Seminarunterlagen

Grundlagen des Testens - Bosch Certified Tester -

Dr. Rainer Schmidberger

rainer.schmidberger@informatik.uni-stuttgart.de

 © 2008-2015 Dr. Rainer Schmidberger

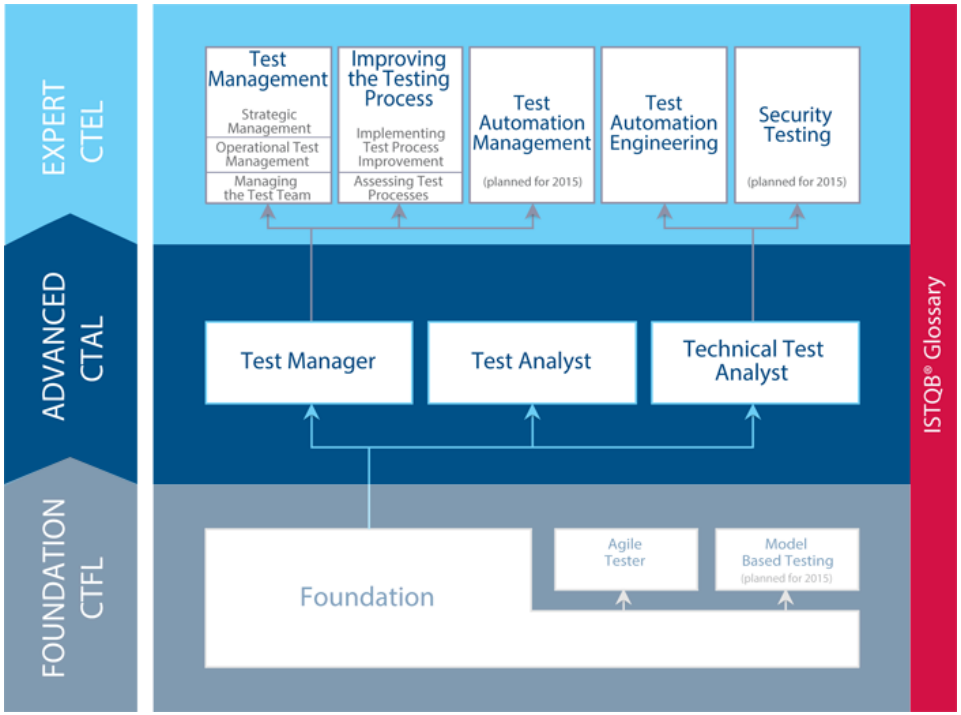
Inhaltsübersicht

1. Tag	2. Tag	3. Tag
Einführung • Übersicht der Prüfverfahren • Motivation und Begriffe • Fundamentaler Testprozess	**Testfallentwurfsverfahren** • Spezifikationsbasiert • Programmcodebasiert • Erfahrungsbasierte Verfahren	**Testdokumentation** • Testplan: IEEE 829 • Testbericht • Testsuite / Testfälle
Testen im Software-Lebenszyklus • V-Modell • Modultest		**Testwerkzeuge** • Typen • Nutzen und Risiken • Einführungsstrategien
Mittagspause 12:45 – 13.45 Uhr		
Testen im Software-Lebenszyklus • Integrationstest • Systemtest • Test nichtfunktionaler Anforderungen	**Praxisbericht**	**Standards** • ISO 15504, IEC 61508, IEC 26262, TPI • ISTQB
	Testmanagement • Organisation und Rollen • Fehlermanagement • Metriken • Strategie	**Abschluss** • Mustertest zum ISTQB Certified Tester • Abschlussrunde

© 2008-2015 Dr. Rainer Schmidberger

Test-Zertifizierung: ISTQB

- International Software Testing Qualifications Board
- german-testing-board.info / istqb.org
- Non-Profit-Organisation, die die Lerninhalte für Test-Seminare und Zertifizierungen festlegt



Quelle: german-testing-board.info und istqb.org

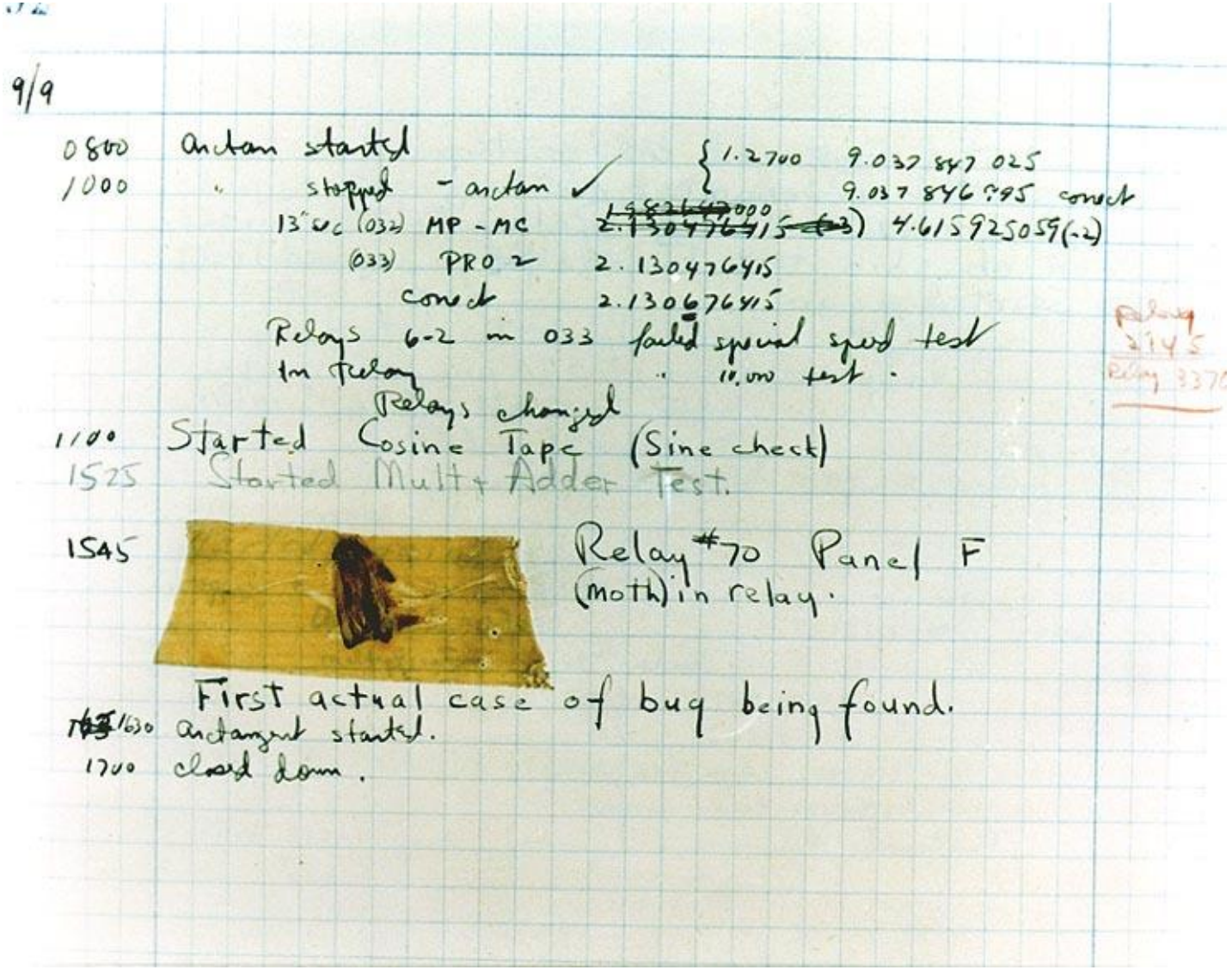
© 2008-2015 Dr. Rainer Schmidberger

Kapitel 1
Einführung

- Übersicht der Prüfverfahren
- Motivation und Begriffe
- Fundamentaler Testprozess
- Testziele
- Testprinzipien nach G. Myers

© 2008-2015 Dr. Rainer Schmidberger

Das Objekt der Begierde ...



[Grace Hopper, Logbuch-Seite des Mark II Aiken Relay Calculator mit dem ersten „Bug“ von 1947]

Was ist Testen?

Testen ist die Ausführung eines Programms mit der Absicht, Fehler zu finden.

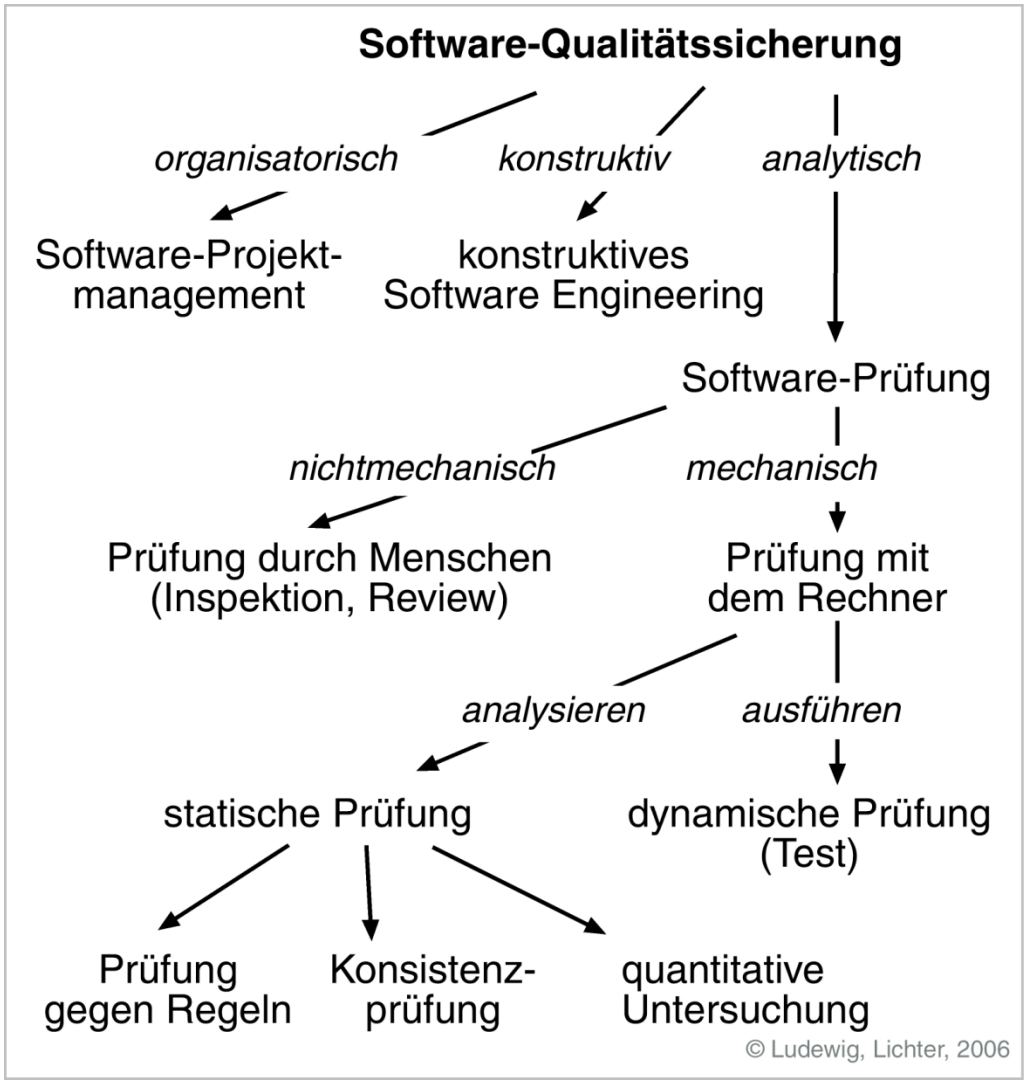
Testen bedeutet die Qualität eines Programms zu messen.

Testen ist der Prozess das Vertrauen zu begründen, dass ein Programm das tut, was es tuen soll.

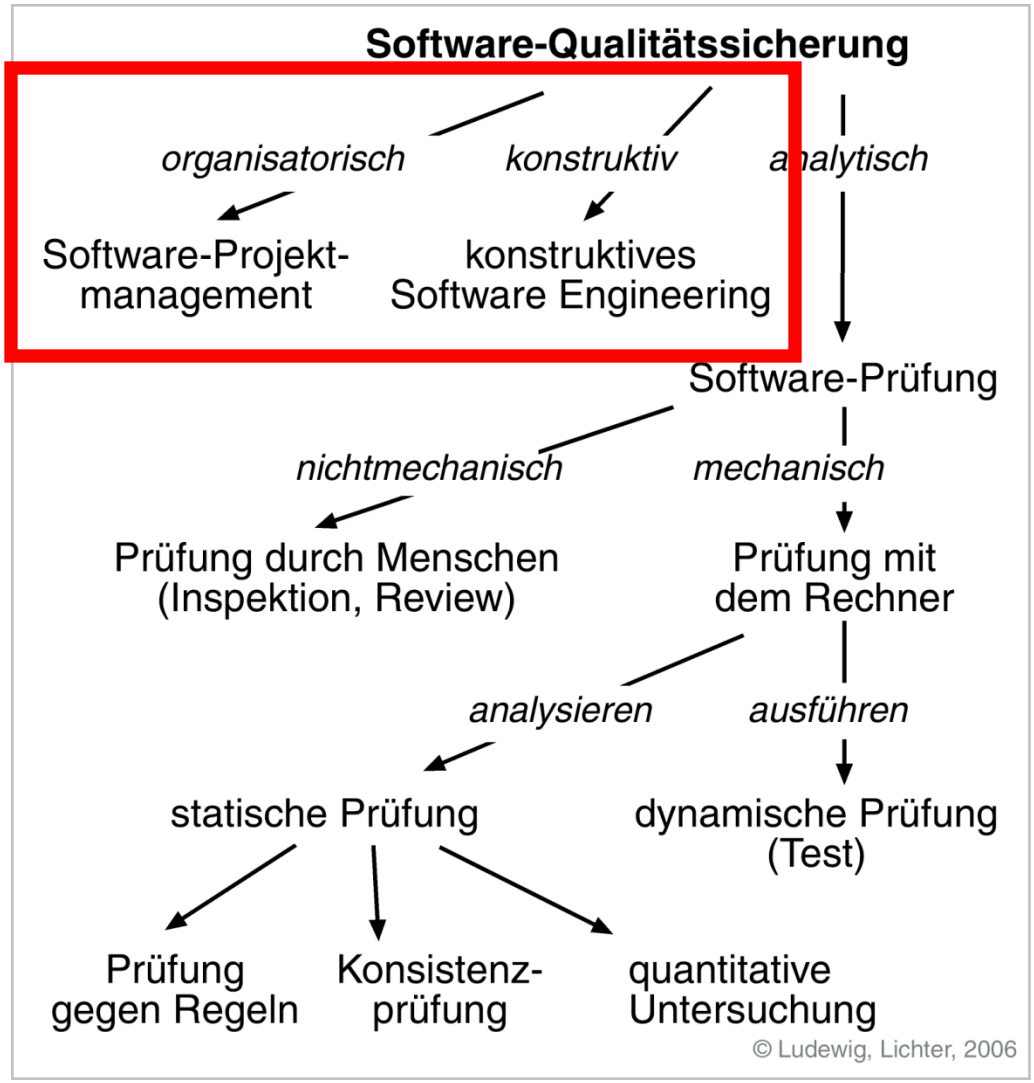
Testen ist jede Aktivität, die darauf ausgerichtet ist eine Eigenschaft oder Funktion eines Programms oder eines Systems darauf hin zu überprüfen, dass die geforderten Ergebnisse eintreffen

[Hetzel, 1988], [Ludewig 2010], [Meyers, 1979]

Prüfverfahren



Prüfverfahren



Das Entstehen von Fehlern wird durch geeignete Maßnahmen während der Entwicklung verhindert:

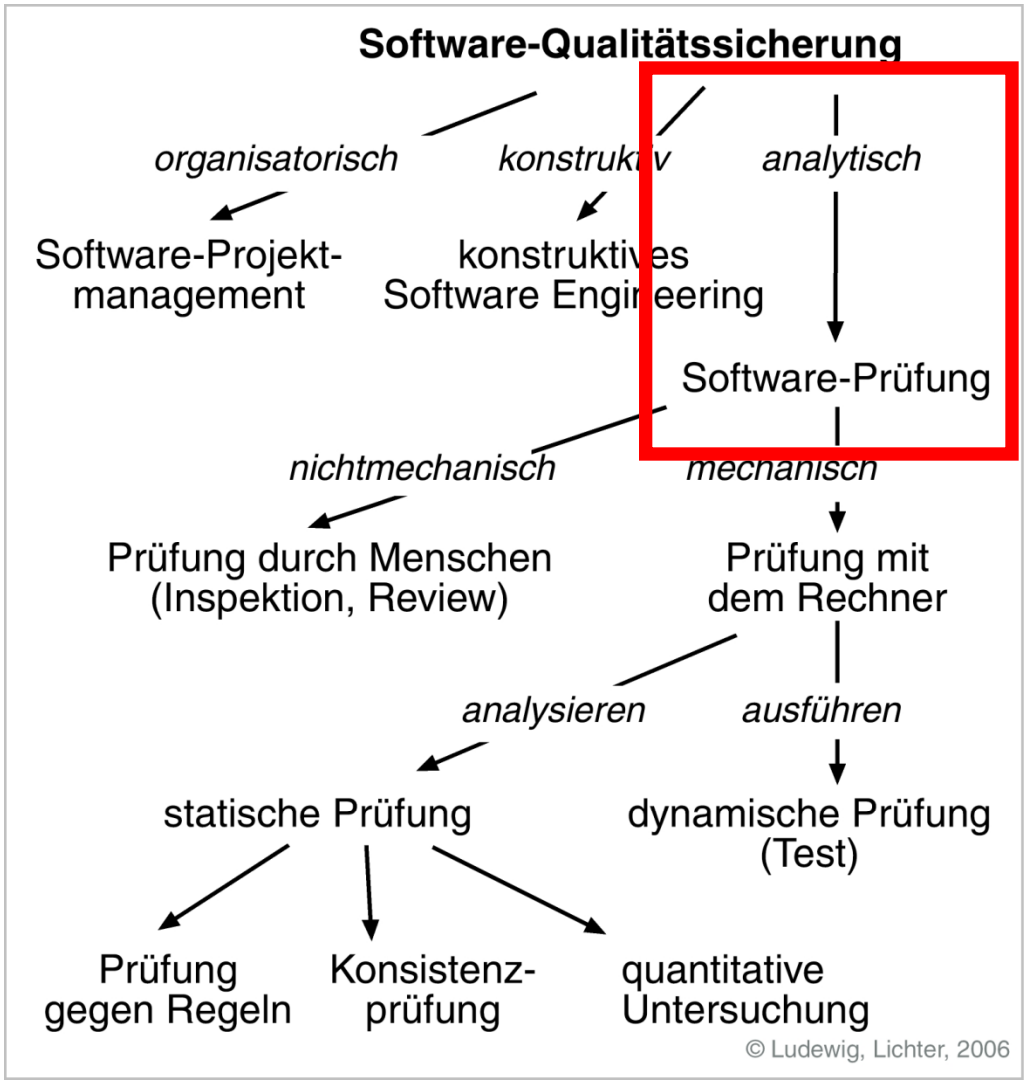
Organisatorisch:

- Regelung von Zuständigkeit
- Regelung von Richtlinien, Standards, Checklisten
- Schulung der Mitarbeiter

Konstruktiv

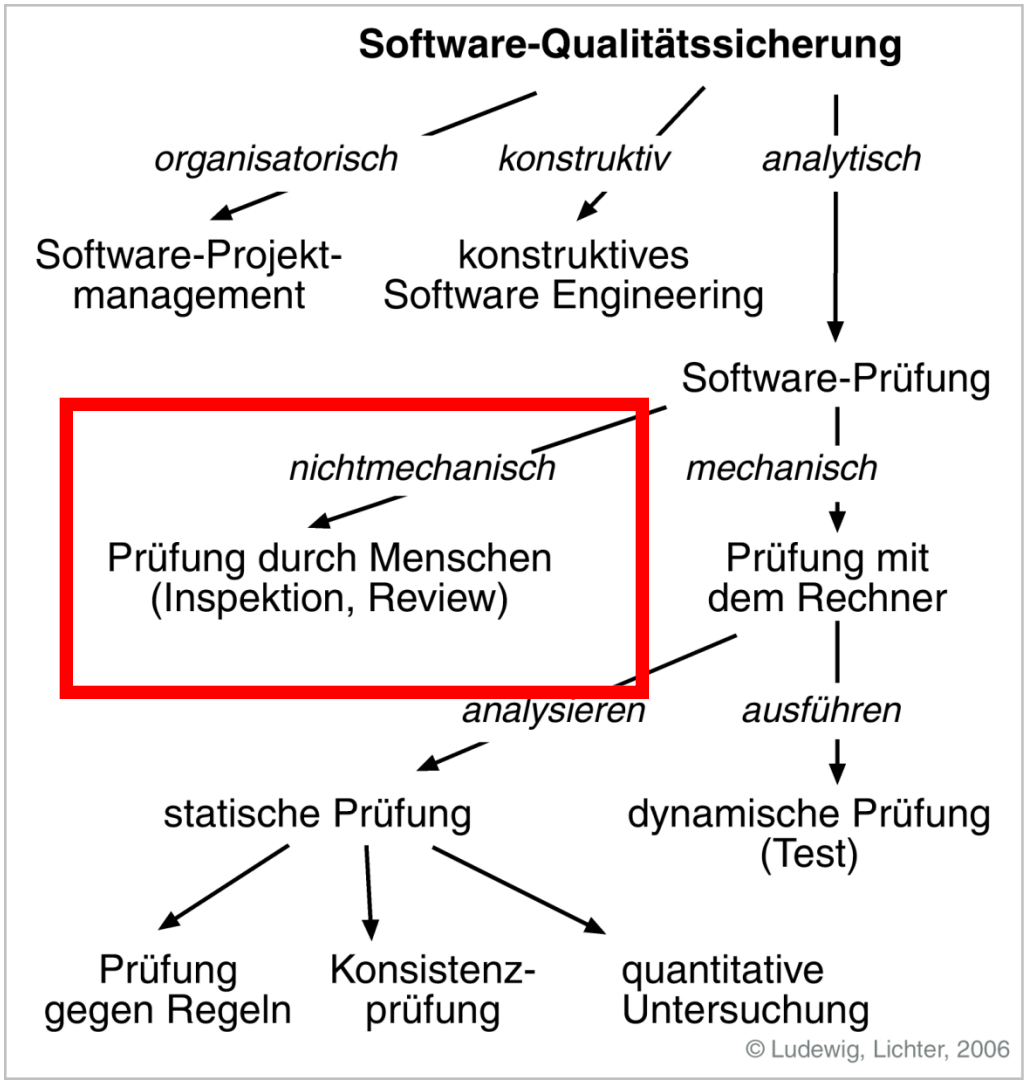
- Einsatz geeigneter Methoden, Sprachen und Werkzeuge.
- Verwendung bestimmter Prozessmodelle

Prüfverfahren



- Der Prüfgegenstand (Teil-, Zwischen- oder Endprodukt) wird auf Fehler hin untersucht
- Die Qualität des Prüfgegenstands wird bewertet
- Es wird geprüft, ob ein Prüfgegenstand bestimmte vorgegebene Qualitätskriterien erfüllt

Prüfverfahren



- **Durchsicht** (Schreibtisch-Test)

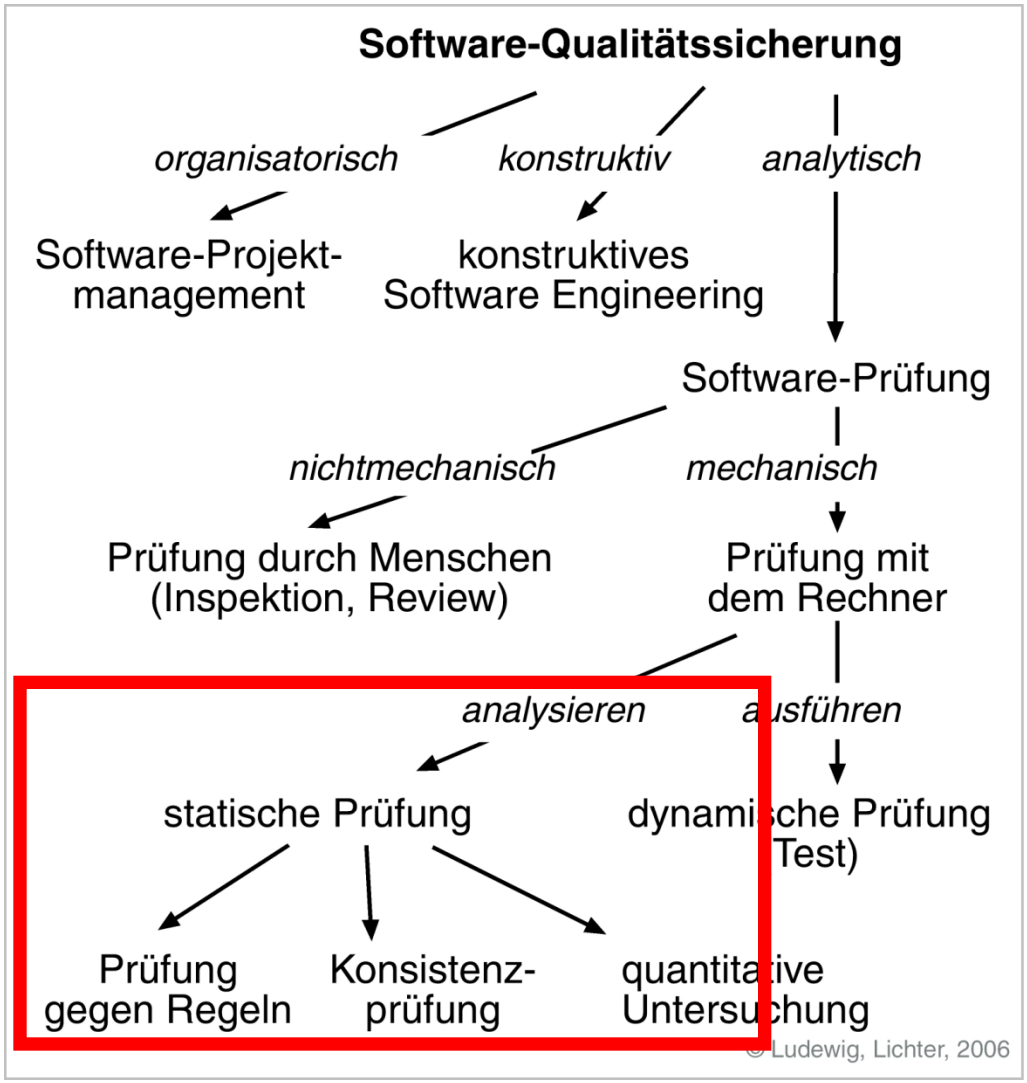
  Informell, wird vom Urheber selbst durchgeführt
- **Stellungnahme**

  Einholen einer anderen Meinung
- **Technischer Review**

  Wohldefiniertes, dokumentiertes Verfahren zur Bewertung eines Prüfgegenstands
- **Walkthrough**

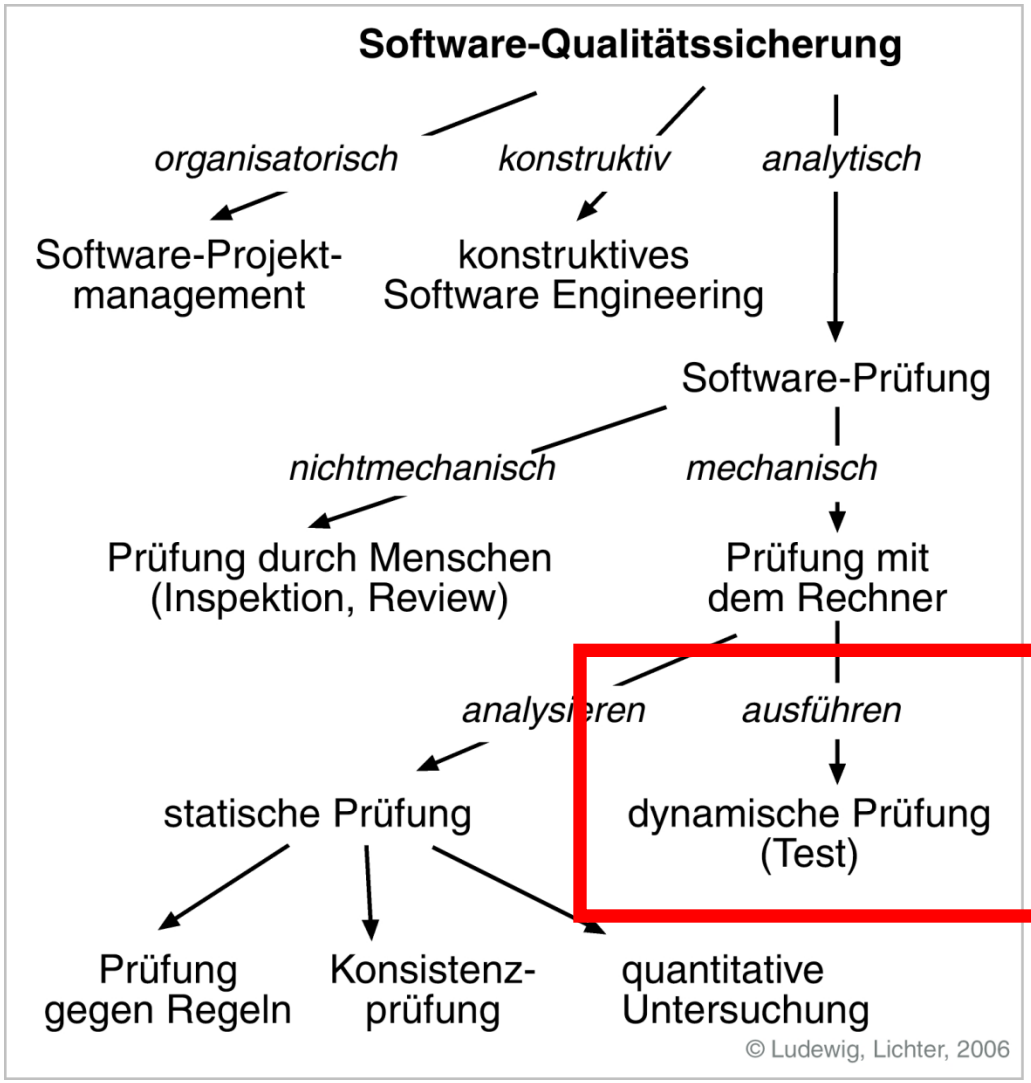
  Mischform zwischen der Stellungnahme und dem technischen Review

Prüfverfahren



- Stilanalysen wie z. B. Konformität zu Programmierrichtlinien
- Architekturanalysen
- Anomalien (z. B. Datenflussanomalien)
- Code-Metriken

Prüfverfahren



„Testen ist die Ausführung eines Programms mit der Absicht, Fehler zu finden“

[Myers, 1979]

und

Dokumentation der genauen Ausführungs-bedingungen wie z.B. Version, Testfälle und Resultate.

Test

An activity in which a system or component is **executed** under specified conditions, **the results are observed** or recorded, and an **evaluation is made** of some aspect of the system or component.

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

© 2008-2015 Dr. Rainer Schmidberger

Testen vs. Prüfung

- Testing im Englischen:
  - Übersetzung: Bewertung, Erprobung, Probe, Probebetrieb, das Prüfen, Prüfung, Schätzung, das Testen, die Überprüfung, die Untersuchung
  - Gemeint ist jede Art von Überprüfung also auch Reviews (der Spezifikationen, des Codes...)
- Testen im Deutschen (nach allgemeinem Verständnis):
  - Ausführung und in der Regel dabei Fehlersuche am ausführbaren / benutzbaren Produkt oder Prototyp

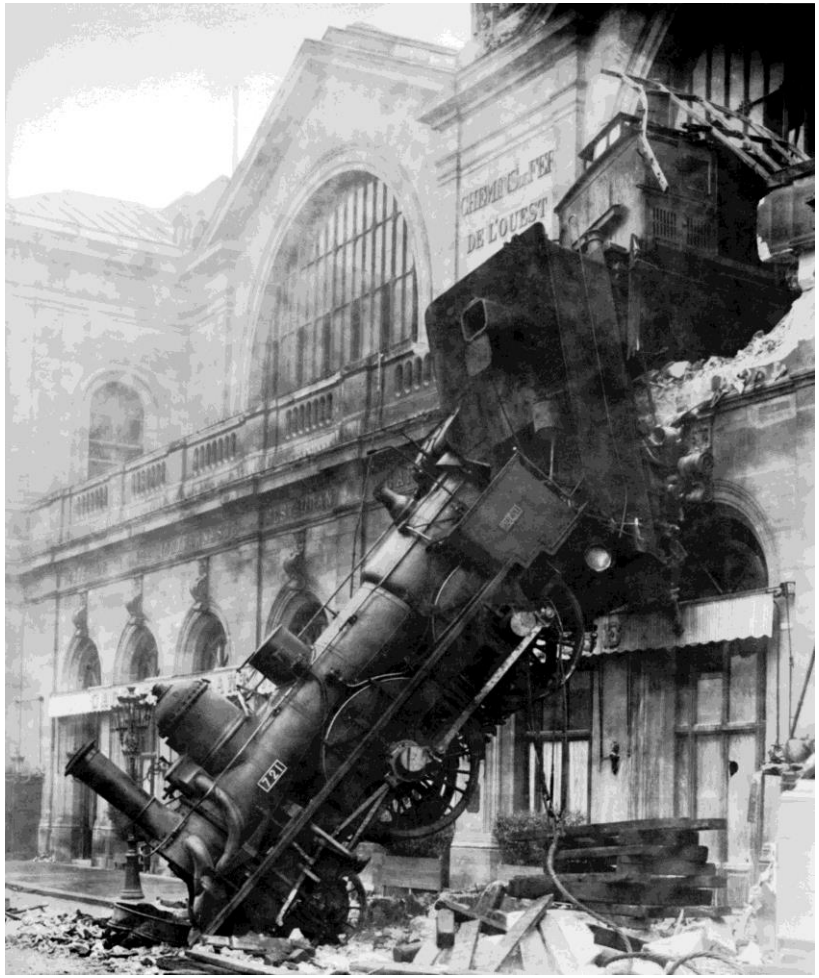
Fehlhandlung, Defekt, Fehlverhalten

Begriff	Definition	Vermeidbar oder erkennbar durch
Error/Mistake (**Fehlhandlung**, Versagen, Irrtum)	Fehlerhafte Aktion einer Person (Irrtum), die zu einer fehlerhaften Programmstelle führt. **Eine Person macht einen Fehler**	Schulung, Konvention, Prozess-verbesserung
Fault/Defect/ Bug (**Defekt**, Fehlerzustand, Fehlerursache)	Fehlerhafte Stelle (Zeile) eines Programms, die ein Fehlverhalten auslösen kann. **Das Programm enthält einen Fehler**	Inspektion, Review, Programmanalyse
Failure (**Fehlerwirkung**, Mangel)	Fehlverhalten eines Programms gegenüber der Spezifikation, das während seiner Ausführung (tatsächlich) auftritt. **Die Anwendung zeigt einen Fehler**	Test

Begriffe gem. IEEE Std. 610 und ISTQB

© 2008-2015 Dr. Rainer Schmidberger

Das Verhüten von Unfällen darf nicht als eine Vorschrift des Gesetzes aufgefasst werden, sondern als ein Gebot menschlicher Verpflichtung und wirtschaftlicher Vernunft.



Werner von Siemens,1884

© 2008-2015 Dr. Rainer Schmidberger

Ariane Flug 501

- ESA, Kourou, Franz. Guyana, 4. Juni 1996
  - Selbstzerstörung der Ariane 5 beim Jungfernflug 39 Sekunden nach dem Start
  - 4 Satelliten werden verloren: 400-500 M Euro, 2 Jahre Verzug im Entwicklungsprogramm: > 500 M Euro, 2 zusätzliche Erprobungsstarts

Der Quellcode

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


© 2008-2015 Dr. Rainer Schmidberger

Ariane: Chronik der Ereignisse – 1

- Die Software für das Trägheitsnavigationssystem wird unverändert von der Ariane 4 übernommen. Ein Test dieser Software unterbleibt daher.
- Die übrigen Systeme der Rakete werden komponentenweise gründlich getestet. Ein gemeinsamer Test der gesamten Steuerungssoftware der Rakete unterbleibt aus Kosten- und Machbarkeitsgründen.
- In der Software für das Trägheitsnavigationssystem gibt es eine Abgleichsfunktion, deren Werte eigentlich nur sinnvoll sind, solange die Rakete noch nicht fliegt. Diese Funktion arbeitet programmgemäß bis ca. 40 s nach H0 weiter, weil das bei der Ariane 4 im Fall eines Countdownabbruchs kurz vor dem Abheben sinnvoll war.
- Flug 501 startet am 4. Juni 1996. Die Triebwerke zünden um H0=9:33:59 Ortszeit. Die ersten 36 Sekunden des Flugs verlaufen normal.

Lions, J.L. (1996). ARIANE 5 Flight 501 Failure. Report by the Inquiry Board. Paris: ESA.
http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
Zusammenfassung von Martin Glinz, Universität Zürich

Ariane: Chronik der Ereignisse – 2

- Da die Ariane 5 eine andere Flugbahn hat als die Ariane 4, berechnet die Abgleichsfunktion einen Wert, der wesentlich größer ist als erwartet.
- Bei der Konvertierung dieses Werts von einer 64 Bit Gleitkommazahl in eine 16-Bit Festkommazahl tritt ein Überlauf ein; der Rechner erzeugt eine Ausnahmebedingung.
- Die Ausnahmebedingung wird nicht behandelt (obwohl dies in der verwendeten Programmiersprache Ada möglich wäre).
- Der Trägheitsnavigationsrechner setzt eine Fehlermeldung an den Steuerrechner der Rakete ab und schaltet sich 36,75 s nach H0 ab.
- Das Trägheitsnavigationssystem ist aus Sicherheitsgründen doppelt ausgelegt. Ein Umschalten auf das zweite System schlägt fehl, da dieses System das gleiche Problem gehabt und sich vor 0,05 s ebenfalls abgeschaltet hat.

Lions, J.L. (1996). ARIANE 5 Flight 501 Failure. Report by the Inquiry Board. Paris: ESA.
http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
Zusammenfassung von Martin Glinz, Universität Zürich

© 2008-2015 Dr. Rainer Schmidberger

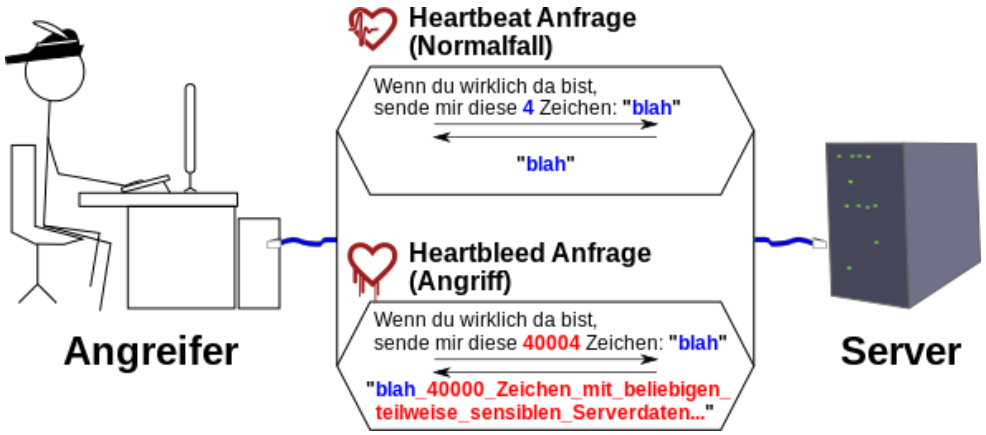
Ariane: Chronik der Ereignisse – 3

- Die Software des Steuerrechners ist auf den Ausfall beider Trägheitsnavigationssysteme nicht ausgelegt und interpretiert die gemeldeten Fehlercodes als Flugbahndaten.
- Dies führt zu völlig unsinnigen Berechnungen und als Folge davon zu unsinnigen Stellbefehlen an die Steuerdüsen der Rakete: Diese werden bis zum maximal möglichen Anstellwinkel ausgeschwenkt.
- Aufgrund der resultierenden Scherkräfte zerbricht die Rakete, worauf der Selbstzerstörungsmechanismus ordnungsgemäß anspricht. Dieser sprengt Rakete und Nutzlast und verhindert damit, dass größere Trümmerteile auf den Boden fallen.

Lions, J.L. (1996). ARIANE 5 Flight 501 Failure. Report by the Inquiry Board. Paris: ESA.
http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
Zusammenfassung von Martin Glinz, Universität Zürich

© 2008-2015 Dr. Rainer Schmidberger

Superdefekt: Der Heartbleed-Exploit



Der sogenannte Heartbeat soll es eigentlich Server und Client ermöglichen, eine TLS-Verbindung am Leben zu halten. Zu diesem Zweck sendet einer der Kommunikations-partner eine Payload mit beliebigem Inhalt an das andere Ende. Der Kommunikationspartner schickt dann exakt die selben Daten zurück, um zu zeigen, dass die Verbindung nach wie vor in Ordnung ist.

"Catastrophic is the right word. On the of 1 to 10, this is an 11."

Bruce Schneier

Der Code vor dem Patch:

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
memcpy(bp, pl, payload);
```

Und nach dem Patch:

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
 return 0; /* silently discard per RFC 6520 sec. 4 */
```

Quelle: Heise-online.de

© 2008-2015 Dr. Rainer Schmidberger

Lessons Learned für den Test

- **Integration:** Bei der Prüfung von Software, die aus mehreren Komponenten besteht, genügt es nicht, jede Komponente nur isoliert für sich zu prüfen.
- **Fehlerbehandlung:** Liefert eine Komponente Fehlerdaten, sind Testfälle erforderlich, die die Fehlersituation prüfen.
- **Systemtest** ist unumgänglich aber aufwändig
- **Fehler** treten gerne dort auf, wo man sie nicht erwartet.
- Fehler haben ein unterschiedliches **Schadens-potential**: Der Test sollte dies berücksichtigen.
- Testen erfordert neben der **Systematik** auch viel **Kreativität**.

Nach Martin Glinz, Universität Zürich

© 2008-2015 Dr. Rainer Schmidberger

Was macht Testen so schwierig?

Die Testziele sind unklar.

Es ist viel zu oft unklar, welches System-verhalten als korrekt oder als falsch gilt.

Der Aufwand für den Test kann vorab nicht geschätzt werden.

Die im Test gefundenen Fehler erweisen sich viel zu oft als falsch.

Testen wird permanent von Fehlerbehebung unterbrochen.

Der Testaufwand ist unverhältnismäßig hoch.

Der Nutzen des Tests kann nicht prognostiziert werden.

Die Testergebnisse sind nicht reproduzierbar.

Ziele des systematischen Tests

- **Systematik:** Ein definiertes Testziel wird angestrebt
- **Planbarkeit**: Aufwand und Nutzen (z. B. gefundene Fehler!) können prognostiziert werden
- **Vollständigkeit**: Es sind Kriterien definiert, die die Vollständigkeit bewerten
- **Reproduzierbarkeit**: Ergebnisse entstehen nicht zufällig sondern systematisch und nachvollziehbar
- **Wirtschaftlichkeit**: Aufwand und Nutzen werden optimiert: Testsuite-Optimierung, Dokumentation, Werkzeugeinsatz, Prozess-Optimierung
- **Risiko- und Haftungsreduktion**: Der systematisch Test gibt Sicherheit, dass kritische Fehler im Betrieb nicht auftreten.

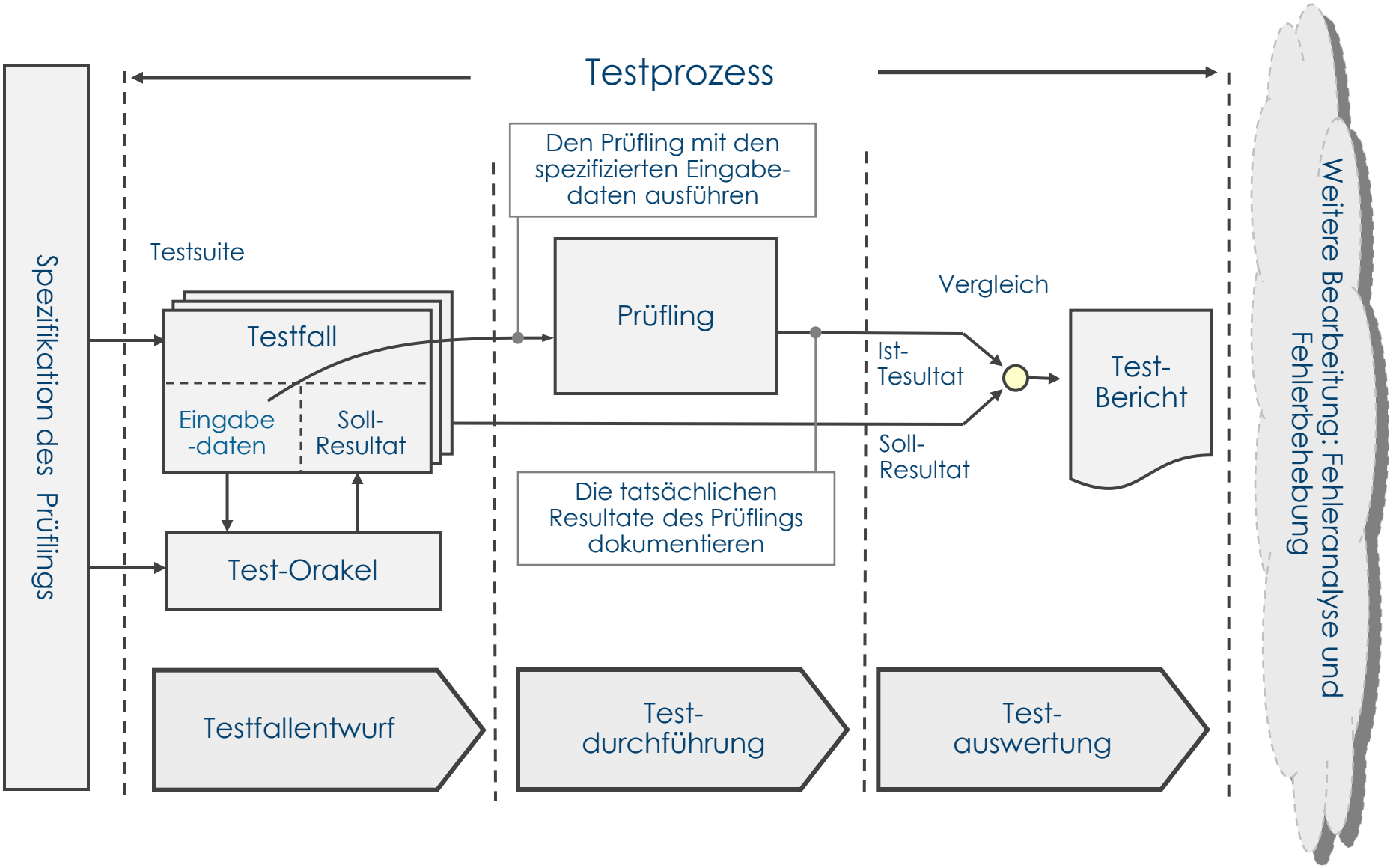
Ad-hoc Test

- Die Testziele sind unklar.
- Es ist viel zu oft unklar, welches Systemverhalten als korrekt oder als falsch gilt.
- Die Testergebnisse sind nicht reproduzierbar.
- Der Aufwand für den Test kann vorab nicht geschätzt werden.
- Der Nutzen des Tests ist stark von der Expertise des Tester abhängig.
- Im Schadensfall drohen Haftungsrisiken.

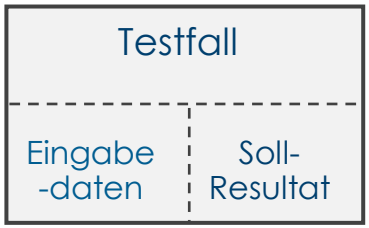
vs. Systematischer Test

- Test ist geplant, eine Testvorschrift liegt vor, die Testziele (z. B. die Vollständigkeit) sind definiert
- Das Programm wird gemäß Testvorschrift – der Test-spezifikation – ausgeführt. Damit ist der Test jederzeit reproduzierbar und nach-vollziehbar.
- Testergebnisse werden systematisch dokumentiert.
- Der Aufwand für ein Testziel kann prognostiziert werden.
- Der systematische Test lässt sich auch systematisch verbessern.

Fundamentaler Testprozess



Was ist ein Testfall?

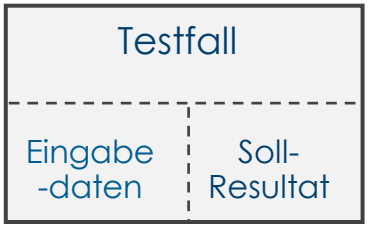


***Testfall** (nach [ISTQB]) umfasst folgende Angaben:*

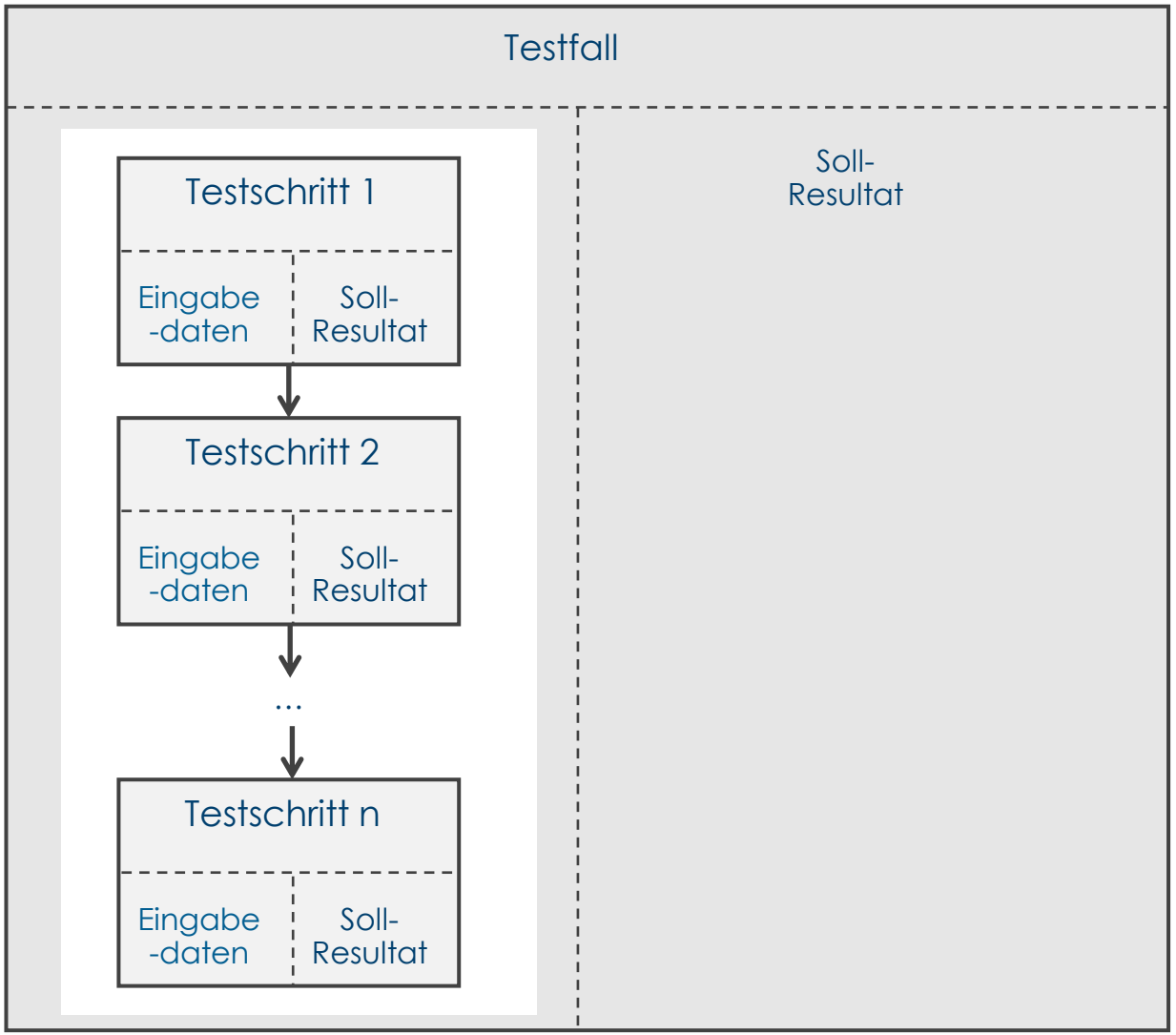
- *die für die Ausführung notwendigen **Vorbedingungen**,*
- *die Menge der **Eingabewerte** (ein Eingabewert je Parameter des Testobjekts),*
- *die Menge der vorausgesagten **Ergebnisse**, sowie die erwarteten Nachbedingungen.*

Testfälle werden entwickelt im Hinblick auf ein bestimmtes Ziel bzw. auf eine Testbedingung, wie z.B. einen bestimmten Programmpfad auszuführen oder die Übereinstimmung mit spezifischen Anforderungen zu prüfen (wie Eingaben an das Testobjekt zu übergeben und Sollwerte abzulesen sind) [nach IEEE 610].

„monolithischer“ Testfall

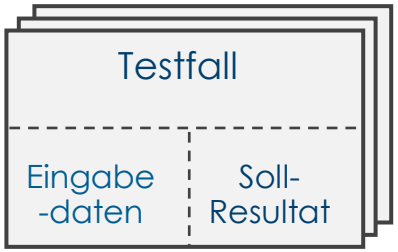


Testfall bestehend aus Testschritten



© 2008-2015 Dr. Rainer Schmidberger

Testsuite



Testsuite *(nach [ISTQB]): Die* ***Zusammenstellung*** *(Aggregation)* ***mehrerer Testfälle*** *für den Test einer Komponente oder eines Systems, bei der Nachbedingungen des einen Tests als Vorbedingungen des folgenden Tests genutzt werden können.*

© 2008-2015 Dr. Rainer Schmidberger

Parametriesierte Testfälle

- Ein oder mehrere Eingangsparameter mit ihren zugehörigen Ausgaberesultaten werden in einer Tabelle „außerhalb“ des Testfalls spezifiziert.

© 2008-2015 Dr. Rainer Schmidberger

Testbegriffe – 1

- **Testfall (Test case)**: besteht aus Ausführungs-bedingungen, Eingabedaten und Sollresultat
- **Testsuite**: Sammlung an Testfällen
- **Sollresultat (Expected result)**: das für eine Eingabe spezifizierte Resultat
- **Überdeckung (Coverage)**: Testvollständigkeitsmaß
- **Testware**: die gesamte Testumgebung (das Testgeschirr – test harness)
- **Test-Orakel**: ein Modul, das das Sollresultat für einen Testfall berechnet
- **Testobjekt**: der Prüfling
- **SUT**: System (hier im Sinne der Software!) under Test, das Testobjekt

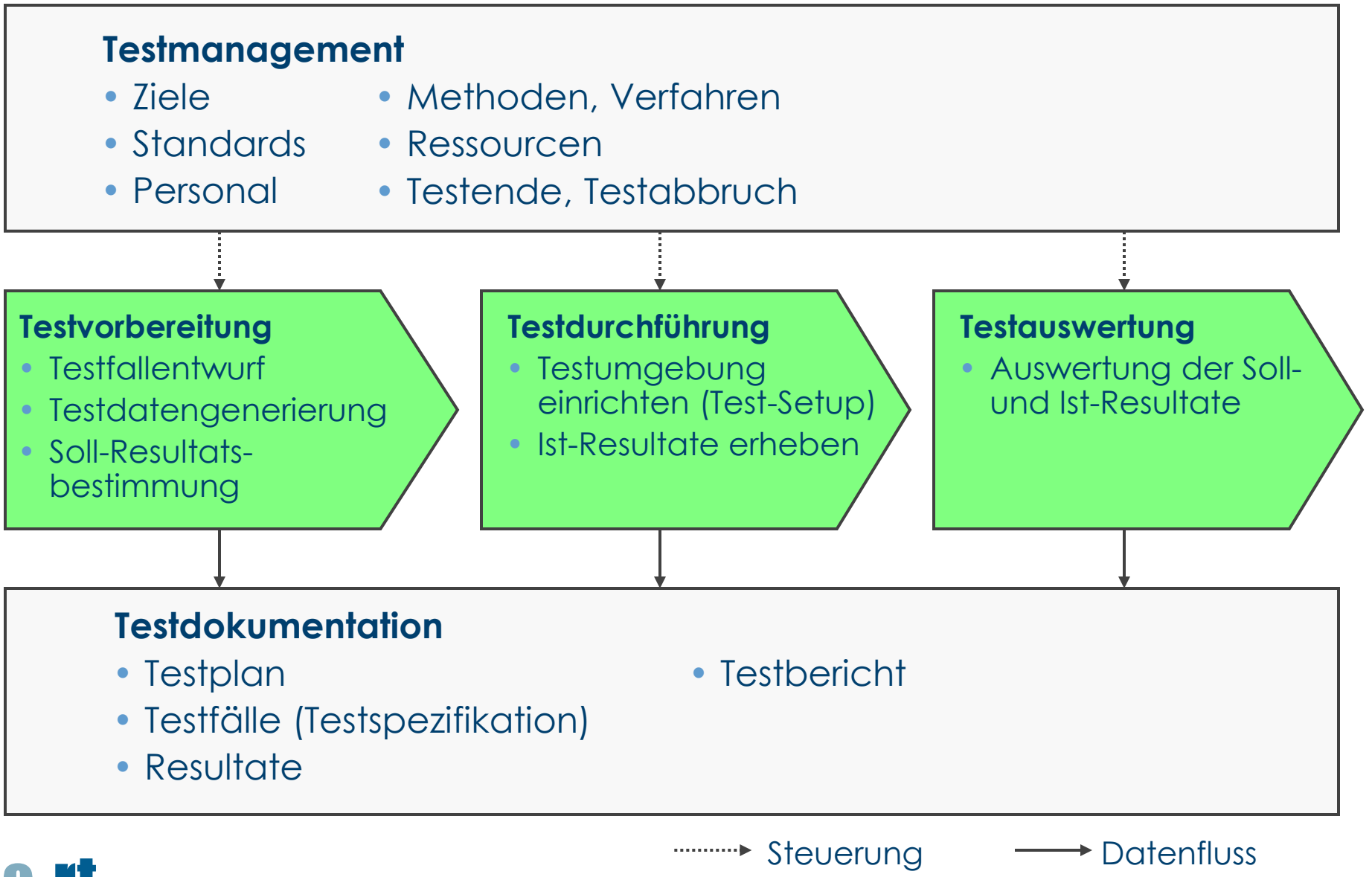
Das ISTQB Glossar

- http://www.german-testing-board.info/news/archiv/istqbRgtb-glossar-21-freigegeben-und-veroeffentlicht.html

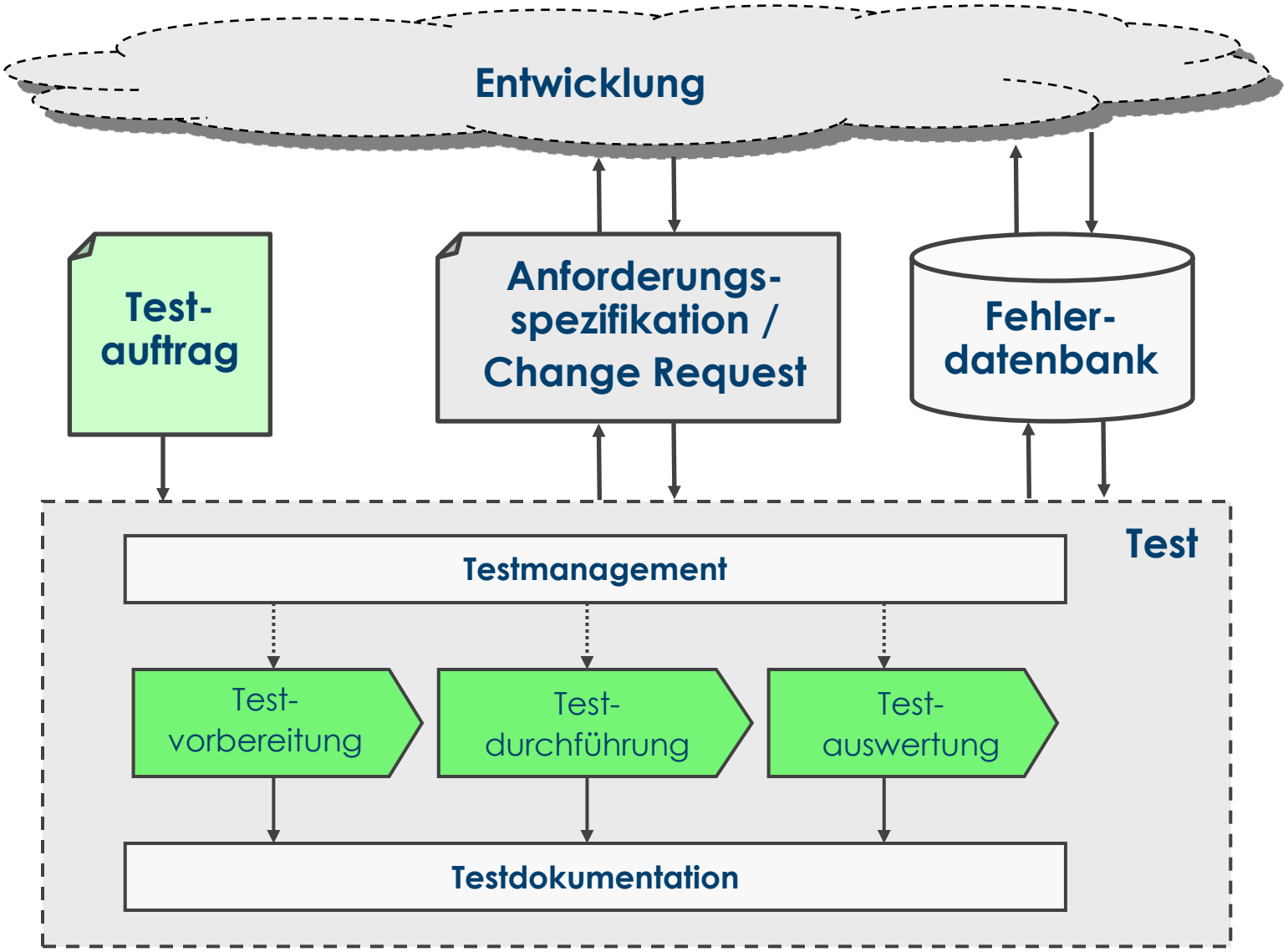
ISTQB/GTB Standardglossar der Testbegriffe Englisch-Deutsch Sortiert nach Term (englisch)

Term	Definition		Begriff	Definition deutsch
A			A	
abstract test case	See high level test case.			
acceptance	See acceptance testing.		Abnahme	Siehe Abnahmetest.
acceptance criteria	The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity [IEEE 610].		Abnahmekriterien	Diejenigen Kriterien, die ein System oder eine Komponente erfüllen muss, um eine Abnahme durch den Benutzer, Kunden oder eine bevollmächtigte Instanz erfolgreich abschließen zu können. [IEEE 610]
acceptance testing	Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system [after IEEE 610].		Abnahmetest	Formales Testen hinsichtlich der Benutzeranforderungen und -bedürfnisse bzw. der Geschäftsprozesse. Es wird durchgeführt, um einem Auftraggeber oder einer bevollmächtigten Instanz die Entscheidung auf der Basis der Abnahmekriterien zu ermöglichen, ob ein System anzunehmen ist oder nicht. [Nach IEEE 610]
accessibility testing	Testing to determine the ease by which users with disabilities can use the component or system [Gerrard].		Zugänglichkeitstest	Testen, um festzustellen, inwieweit Benutzer mit besonderen Bedürfnissen in der Lage sind, eine Komponente oder ein System zu benutzen. [Gerrard]
accuracy	The capability of the software product to provide the right or agreed results or effects with the needed degree of precision [ISO 9126]. See also functionality testing.		Richtigkeit	Die Fähigkeit eines Softwareprodukts, die richtigen oder vereinbarten Ergebnisse oder Wirkungen mit dem benötigten Grad an Genauigkeit zu liefern. [ISO 9126] Siehe auch Funktionalitätstest.
accuracy testing	The process of testing to determine the accuracy of a software product.		Richtigkeitstest	Der Prozess des Testens, um die Richtigkeit eines Softwareprodukts zu bestimmen.
acting (IDEAL)	The phase within the IDEAL model where the improvements are developed, put into practice, and deployed across the organization. The acting phase consists of the activities: create solution, pilot/test solution, refine solution and implement solution. See also *IDEAL*.		Agieren (IDEAL)	Die Phase im IDEAL-Modell, in der die Verbesserungen entwickelt, in die Praxis umgesetzt und unternehmensweit eingesetzt werden. Die Agierenphase besteht aus den Aktivitäten: Lösung erstellen, Lösung erproben/testen, Lösung verfeinern und Lösung umsetzen. Siehe auch *IDEAL*.
action word driven testing	See keyword-driven testing		aktionswortgetriebener Test	Siehe schlüsselwortgetriebener Test.
actor	User or any other person or system that interacts with the system under test in a specific way.		Akteur	Benutzer oder irgendeine andere Person oder ein System, welche mit dem zu testenden System auf eine bestimmte Art interagiert.
actual outcome	See actual result.		tatsächliches Verhalten	Siehe Istergebnis.
actual result	The behaviour produced / observed when a component or system is tested under specified conditions.		Istergebnis	Im Test beobachtetes/erzeugtes Verhalten einer Komponente oder eines Systems unter festgelegten Bedingungen.
ad hoc review	See informal review.		Ad-hoc-Review	Siehe informelles Review.
ad hoc testing	Testing carried out informally, no formal test preparation takes place, no recognized test design technique is used, there are no		Ad-hoc-Testen	Informelles Testen, bei dem keine Testvorbereitung stattfindet und keine anerkannten Testentwurfsverfahren verwendet werden. Es

Aktivitäten beim Test



Testprozess-Schnittstellen



© 2008-2015 Dr. Rainer Schmidberger

Testauftrag

- Prüfling (Version, zu testender Teil des Prüflings, genaue Testrahmenbedingungen)
- Termine
- Verfügbarer Testaufwand und ggf. Ressourcen
- Testgüte, Testvollständigkeit
  - „Rerun-all“
  - Modifikationstest (nur Änderungen testen)
  - Fehlernachtest (nur behobene Fehler testen)
  - Nur Testfälle einer bestimmten Priorität
- Testende- und Testabbruchkriterien

© 2008-2015 Dr. Rainer Schmidberger

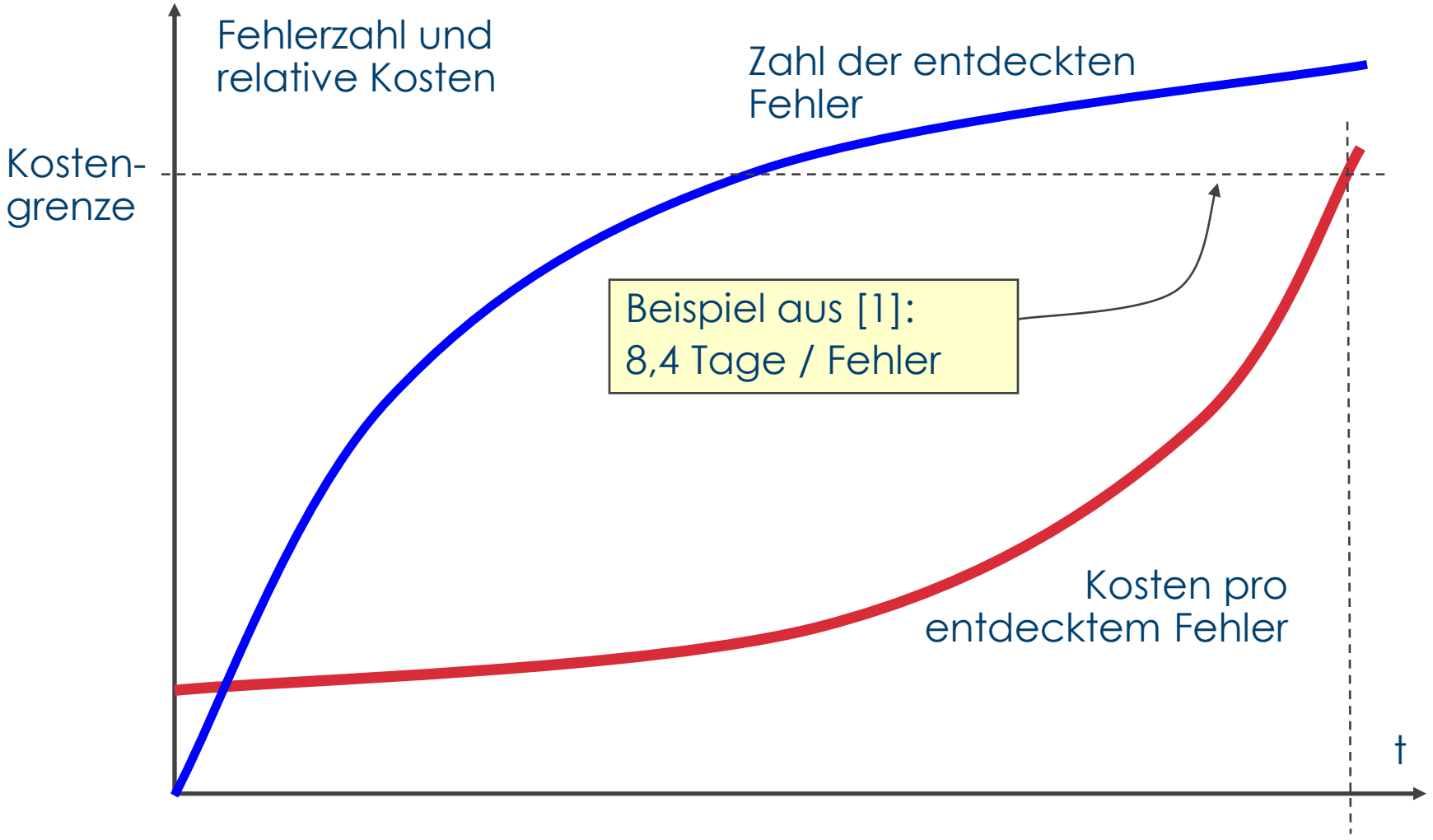
Testendekriterien

- Man kann den Test beenden, wenn
  - alle spezifizierten Testfälle ausgeführt sind
  - die Testfälle einer bestimmten Risikogruppe getestet sind
  - eine bestimmte Zeit kein weiterer Fehler mehr gefunden wird
  - eine bestimmte Anzahl an Fehlern gefunden ist
  - Strukturtest-Kriterien erfüllt sind (z. B. Anweisungsüberdeckung)
  - der vereinbarte Testaufwand geleistet ist

Die Bestimmung des Testendes ist eine der wesentlichen Aufgaben des Testmanagements!

Kosten pro entdecktem Fehler

Aus Ludewig, Lichter, „Software Engineering“, 2. Auflage, Seite 497, dpunkt Verlag, 2010



[1] Ellims, M., Bridges, J., and Ince, D. C. „The Economics of Unit Testing“, Empirical Softw. Engg. 11, 1 (Mar. 2006),

© 2008-2015 Dr. Rainer Schmidberger

Wirtschaftlichkeit des Testens

- Die Wirtschaftlichkeit des Testens wird durch zwei Faktoren bestimmt:
  - Die Kosten, um einen Fehler im Test zu finden (die Testkosten), und
  - die Fehlerkosten, die dadurch entstehen, dass der Fehler unentdeckt bleibt und so lange Schaden anrichtet, bis er in einer späteren Phase, im Betrieb oder gar nicht entdeckt wird.

© 2008-2015 Dr. Rainer Schmidberger

Testbegriffe – 2

True-Positive:
Korrekt gefundener Fehler

False-Positive:
Auf Fehler bewertet, obwohl kein Fehler vorlag.

„Pseudo-Fehler“

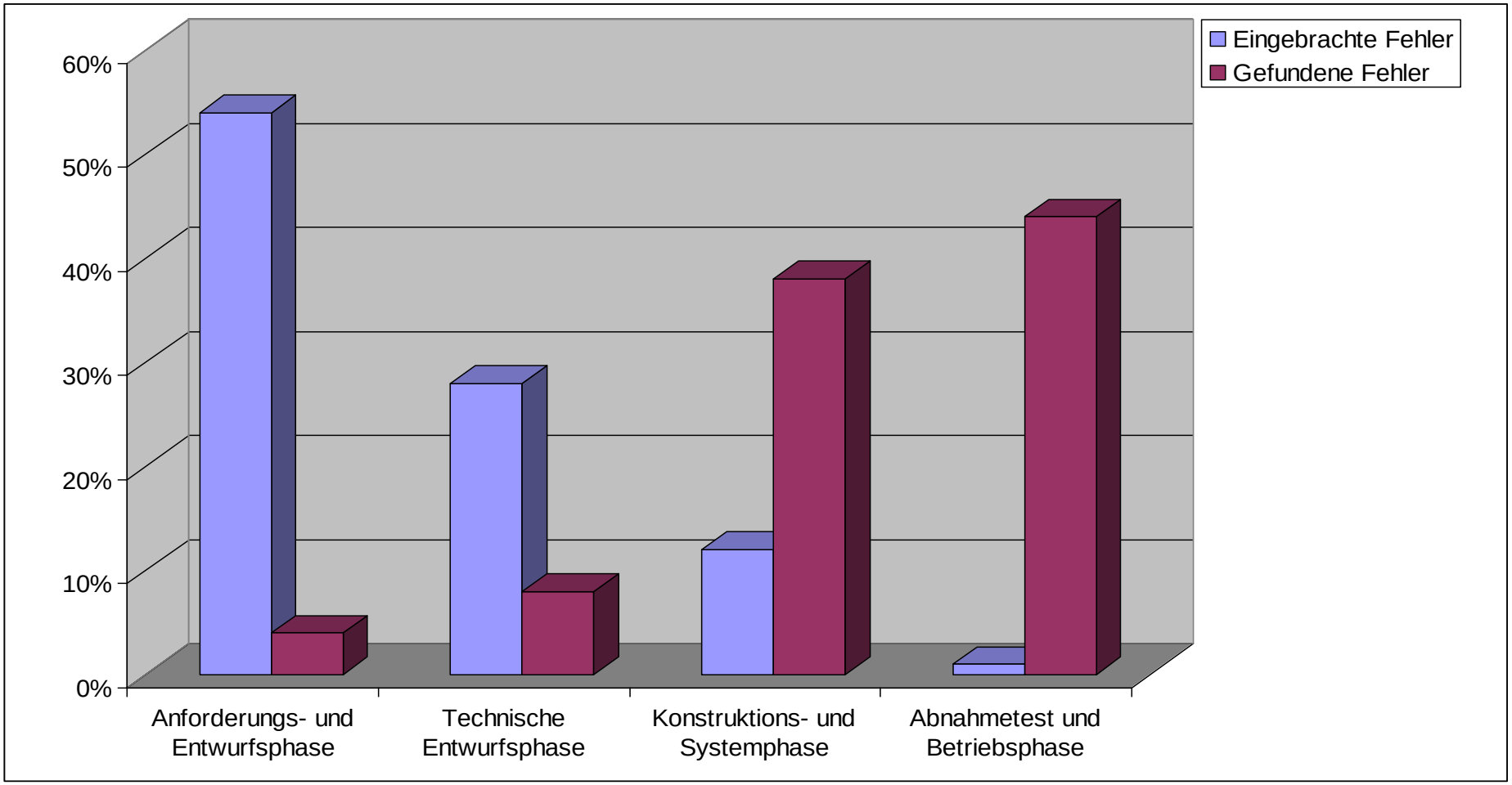
True-Negative:
Korrekt auf „kein Fehler“ bewertet

False-Negative: Auf „kein Fehler“ bewertet, obwohl ein Fehler vorlag.

Problemfälle

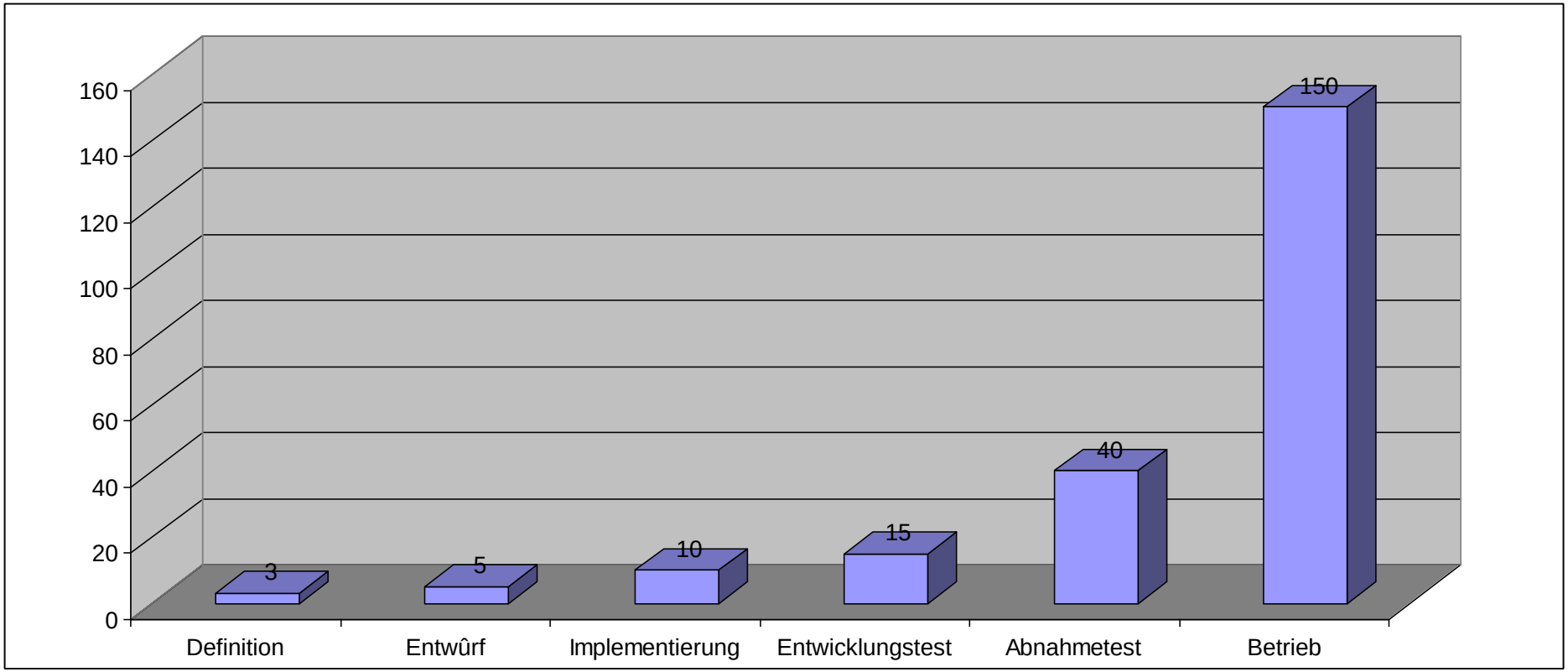
© 2008-2015 Dr. Rainer Schmidberger

Entstehung und Entdeckung von Fehlern
(aus Balzert, IEEE Software, Jan. 1985, S.83)



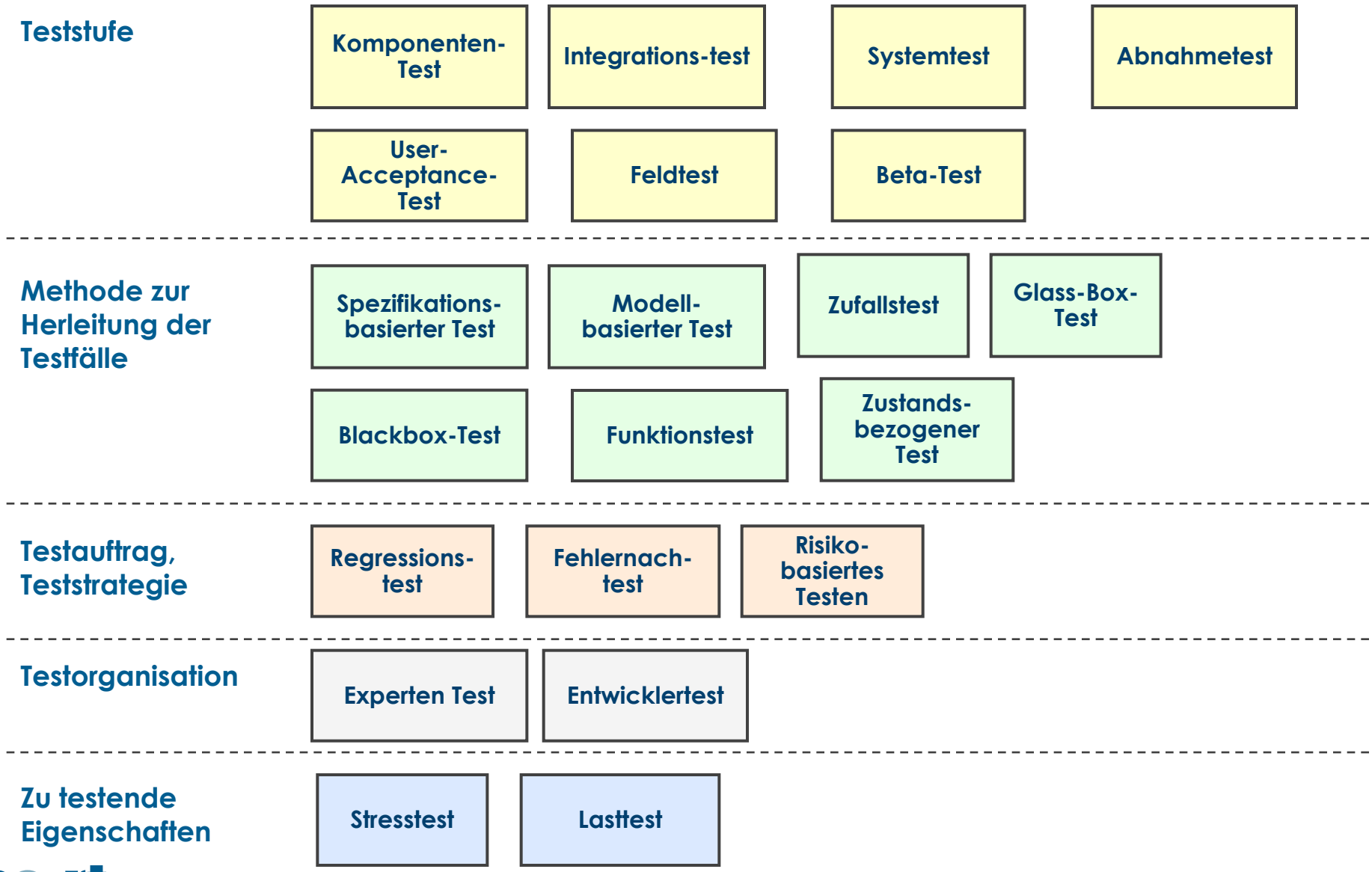
© 2008-2015 Dr. Rainer Schmidberger

Relative Kosten zur Fehlerbeseitigung
(H. Balzert, 1998; Boehm, 1976)



Komponenten-Test
Abnahmetest
Blackbox-Test
User-Acceptance-Test
Systemtest
Integrations-test
Funktionstest
Spezifikations-basierter Test
Modell-basierter Test
Zustands-bezogener Test
Feldtest
Experten Test
Beta-Test
Risiko-basiertes Testen
Fehlernach-test
Regressions-test
Entwicklertest
Glass-Box-Test
Stresstest
Zufallstest
Lasttest

Testarten - Strukturierungsmöglichkeit



© 2008-2015 Dr. Rainer Schmidberger

Auf den Testmix kommt es an …

- Die Testverfahren unterscheiden sich hinsichtlich
  - dem Zeitpunkt im Projektverlauf, zu dem sie angewendet werden können
  - dem Einrichtungsaufwand
  - den Fehlern die gefunden werden können
  - dem Automatisierungsgrad
  - der Effektivität (Aufwand pro gefundenem Fehler)
  - dem Profil des Testers
  - …

Testen besteht in der Regel aus einem Mix an verschiedenen Verfahren. Die Zusammenstellung des „passenden“ Mixes ist eine der zentralen Aufgaben des Testmanagements.

Testprinzipien nach G. Myers – 1

- Ein notwendiger Bestandteil eines Testfalls ist die Definition der erwarteten Werte oder des Resultats.
- Ein Programmierer sollte nicht versuchen, sein eigenes Programm zu testen.
- Die Resultate eines jeden Tests sollen gründlich geprüft werden.
- Testfälle müssen für ungültige und und unerwartete ebenso wie für gültige und erwartete Eingabedaten definiert werden.
- Ein Programm darauf hin zu untersuchen, ob es nicht tut, was es tun sollte, ist nur die eine Hälfte der Schlacht. Die andere Hälfte besteht darin, zu untersuchen, ob das Programm etwas tut, was es nicht tun soll.

© 2008-2015 Dr. Rainer Schmidberger

Testprinzipien nach G. Myers – 2

- Wegwerftestfälle sollten vermieden werden, es sei denn, das Programm ist wirklich ein Wegwerfprogramm.
- Es sollten keine Testverfahren geplant werden unter der stillschweigenden Annahme, dass keine Fehler gefunden werden.
- Testen ist eine extrem kreative und intellektuell herausfordernde Aufgabe.
- Testen ist der Prozess, ein Programm in der Absicht auszuführen, Fehler zu finden.
- Ein guter Testfall ist dadurch gekennzeichnet, dass er mit hoher Wahrscheinlichkeit einen bisher unbekannten Fehler anzuzeigen imstande ist.
- Ein erfolgreicher Testfall ist dadurch gekennzeichnet, dass er einen bisher unbekannten Fehler entdeckt.

Austesten

- Wenn es genau x Möglichkeiten gibt, ein Programm auszuführen, und alle x Möglichkeiten im Test geprüft werden, ist der Test vollständig; das Programm ist **ausgetestet**.
- Aber angenommen, eine Prozedur erwartet 2 Integer-Werte als Eingabedaten
- Bei 32 Bit Länge ergeben sich $2^{32} \times 2^{32}$ viele Kombinationen von Eingabewerten ($\approx 1{,}8 \times 10^{19}$)
- Selbst wenn ein automatischer Testtreiber in 1 ms einen Testfall ausführen kann, dauert der vollständige Test aller Eingabewerte in etwa 570 Mio. Jahre.

➔Das Austesten eines Programms ist (praktisch) unmöglich!

Testen kann die Anwesenheit von Fehlern aufzeigen, aber nie einen Nachweis von Fehlerfreiheit liefern!

Edsger W. Dijkstra, Notes on structured programming, Academic Press, 1972

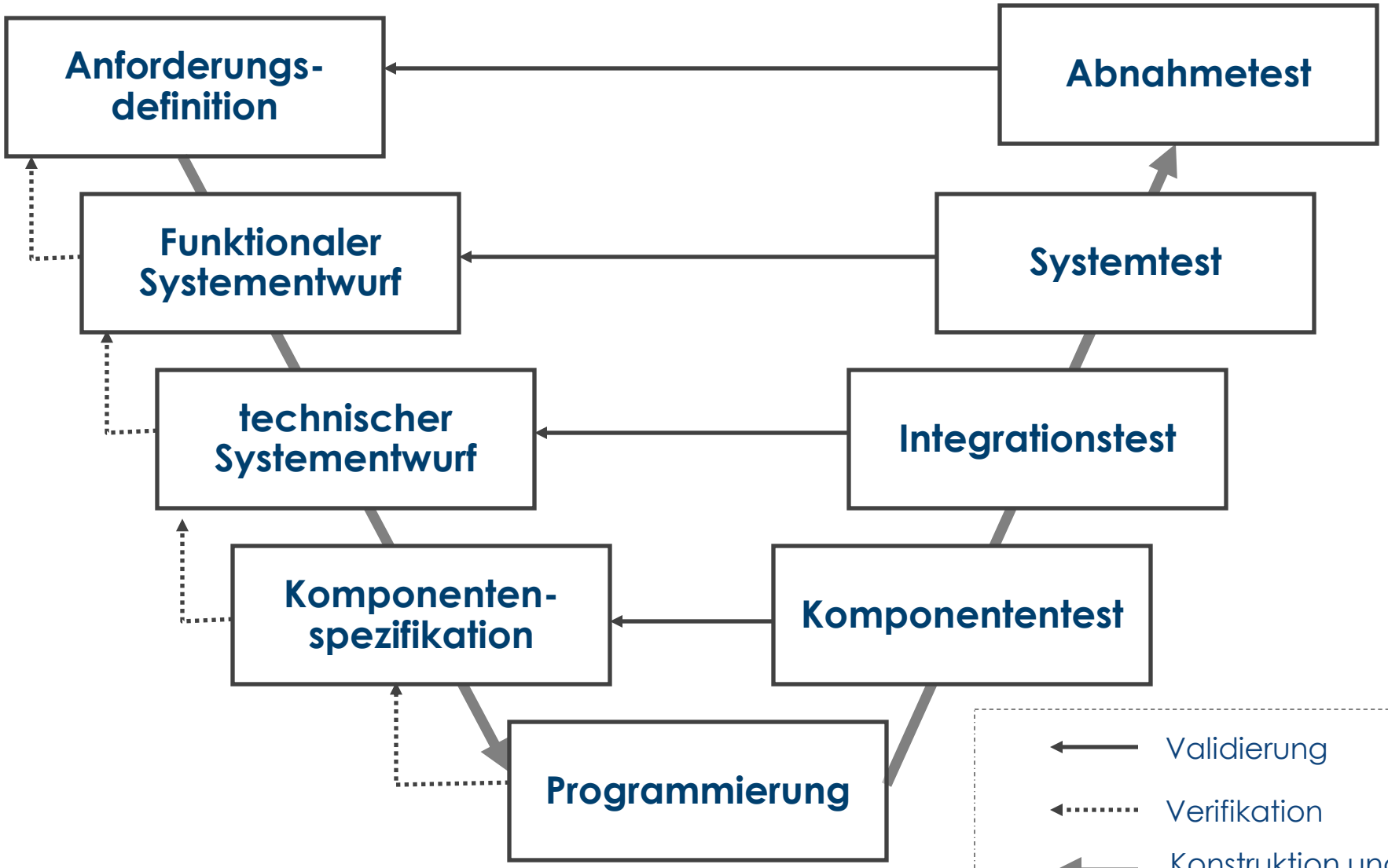
„Keine Fehler“ bedeutet nicht „brauchbares System“

Kapitel 2
Testen im Software-Lebenszyklus

- V-Modell
- Teststufen
- Test nichtfunktionaler Anforderungen

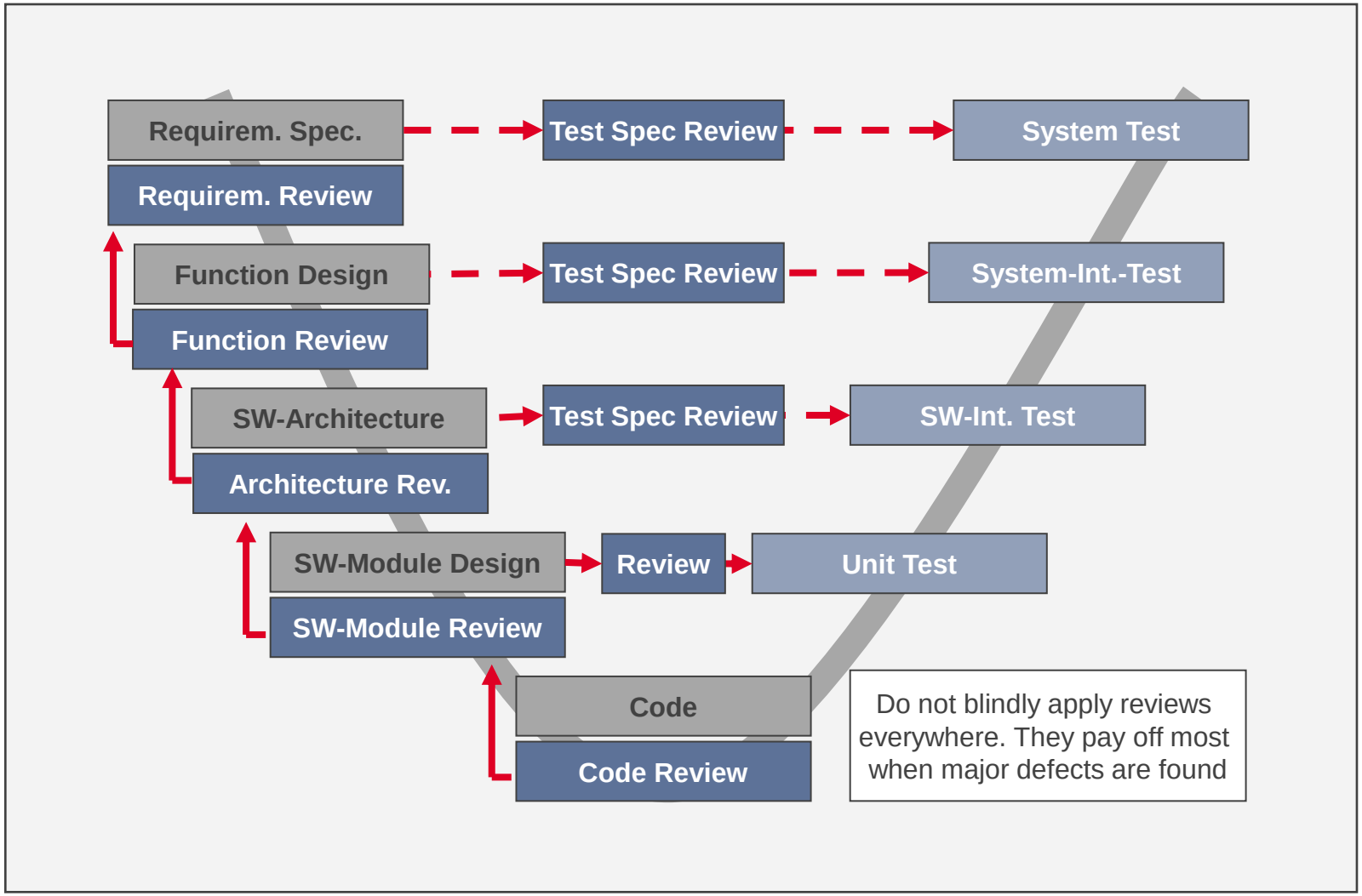
© 2008-2015 Dr. Rainer Schmidberger

V-Modell nach ISTQB



© 2008-2015 Dr. Rainer Schmidberger

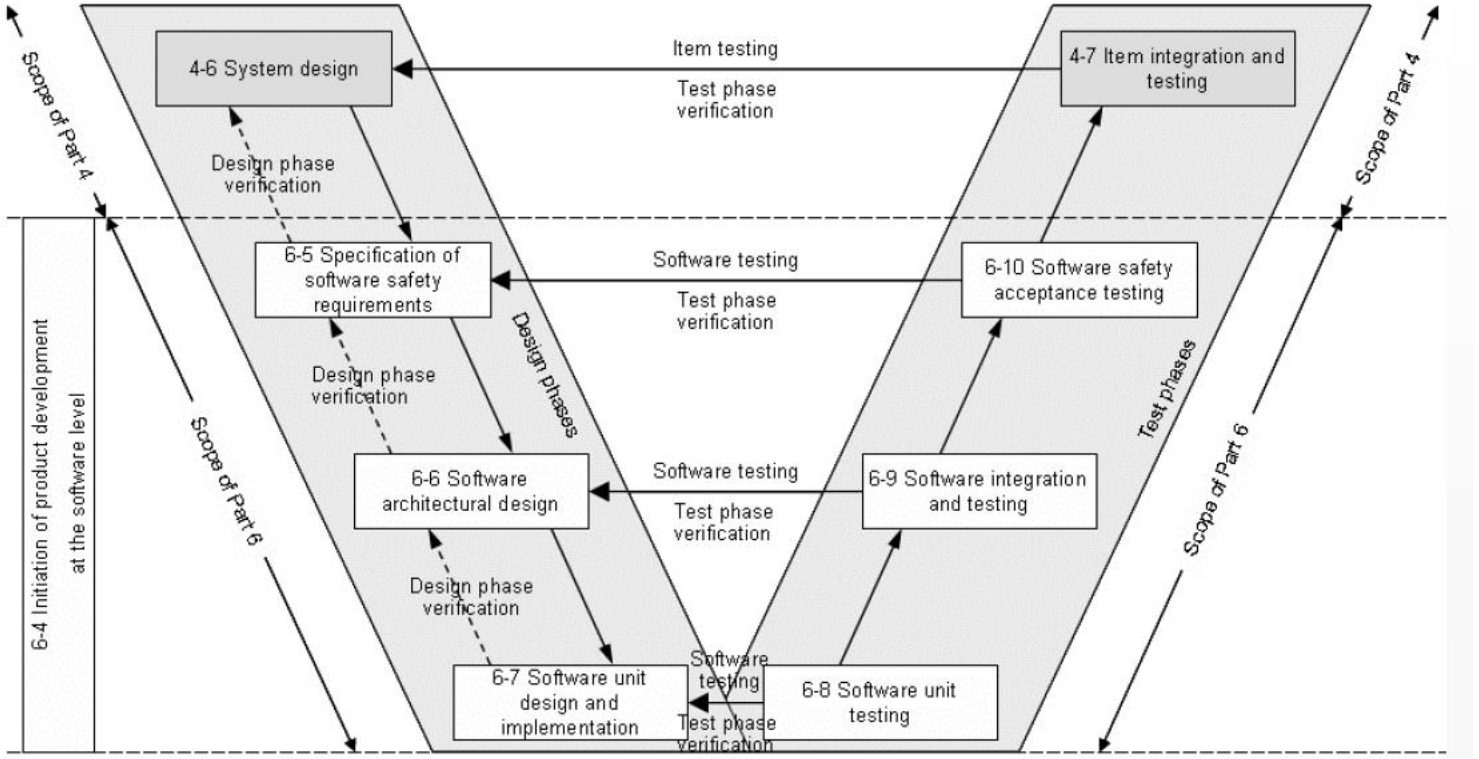
V-Model nach DGS



Quelle: CR/AES-Freimut | 08.09.2008 | © Robert Bosch GmbH 2008. Alle Rechte vorbehalten, auch bzgl. jeder Verfügung, Verwertung, Reproduktion, Bearbeitung, Weitergabe sowie für den Fall von Schutzrechtsanmeldungen.

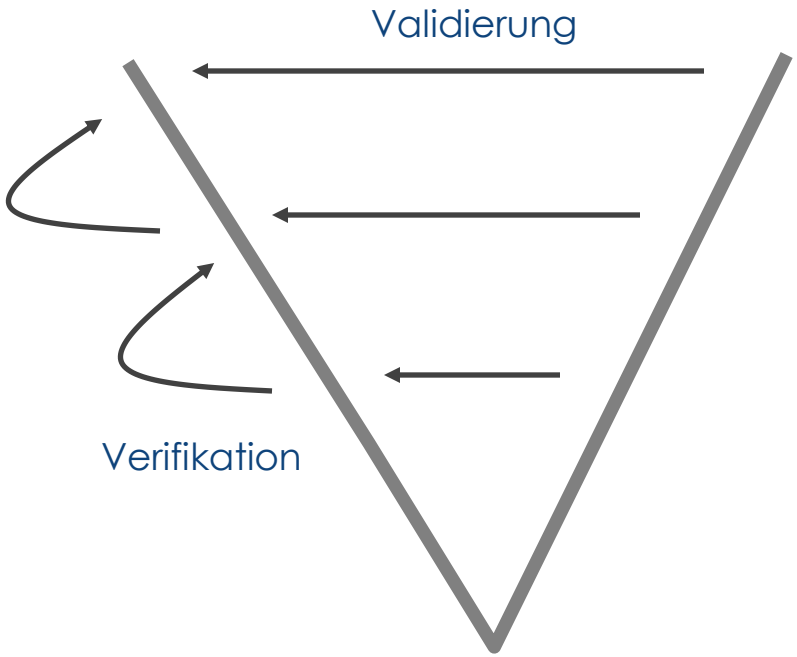
ISO 26262

- Road vehicles – Functional safety



Verifikation und Validierung

- Sprachgebrauch im V-Modell nach ISTQB
  - **Verifikation**: Prüfung, ob die Ergebnisse einer Entwicklungsphase die Vorgaben der Eingangsdokumente erfüllen („Are we doing the thing right?“)
  - **Validierung**: Prüfung, ob ein Entwicklungsergebnis die spezifizierten Anforderungen erfüllt („Are we doing the right thing?“)



Alternativer Sprachgebrauch

- Verifikation = formaler Korrektheitsbeweis
- Validierung = informelle Überprüfung

Testen bedeutet validieren!

Verifikation und Validierung

Verification: (1) The process of evaluating a system or component to determine whether the products of a given development phase **satisfy the conditions imposed at the start of that phase**. **(2)** Formal proof of program correctness.

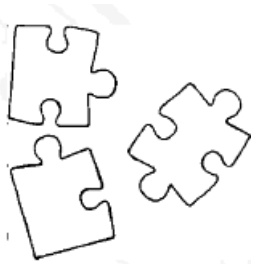
Validation: The process of evaluating a system or component during or at the end of the development process to determine whether it **satisfies specified requirements.**

[IEEE Std 610.12 (1990)]

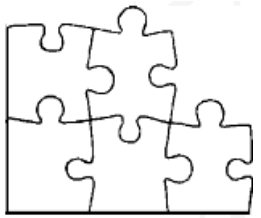
Teststufen - Übersicht

- Testgegenstand sind Komponenten, teilintegrierte Systeme oder Systeme

- Komponententest, Modultest (Unit-Test)
  - ⇨ „autarke“ Komponenten werden isoliert getestet



- Integrationstest
  - ⇨ Teilintegrierte Komponenten testen die Komponenteninteraktion



- Systemtest
  - ⇨ Test des Gesamtsystems

Bilder: Martin Glinz, Universität Zürich

Teststufen

- **Komponententest**
- Integrationstest
- Systemtest

- Weitere Teststufen
  - User Acceptance Test (UAT)
  - Beta-Test
  - Abnahmetest
  - Deployment-Test

- Test gegen nichtfunktionale Anforderungen

© 2008-2015 Dr. Rainer Schmidberger

Komponententest (1)

- „Testen im Kleinen“ (Liggesmeyer)
- Das Testobjekt beim Komponententest ist eine Komponente (=Unit, Modul, Subsystem). Komponenten sind ein Teil einer Applikation, z. B. eine (oder mehrere) Klassen oder eine Prozedur.
- Es gibt weder feste Vorgaben für die Größe einer Komponente (z.B. in LOC) oder für die Anzahl der Komponenten im Gesamtsystem. In der Praxis findet man Systeme, die aus 10 Komponenten aufgebaut sind bis hin zu Systemen, die aus 10.000 Komponenten und mehr aufgebaut sind.
- Im Komponententest wird die Komponente an den Schnittstellen gegen die **Spezifikation** getestet. Dazu ist eine Komponenten-Spezifikation erforderlich!

Komponententest (2)

- Die Komponente sollte möglichst **isoliert** getestet werden; die Isolierung hat dabei den Zweck, komponentenexterne Einflüsse beim Test auszuschließen.
- Die Schnittstelle einer Komponente ist in der Regel eine Programmierschnittstelle. D.h., die Ausführung der Testfälle wird in der Regel in der Programmiersprache des Moduls programmiert.
- Wird ein Fehler gefunden, lässt sich die Ursache in der Regel der getesteten Komponente zuordnen.
- Es stehen Frameworks zur Verfügung: Visual Studio 2008, www.junit.org, www.testng.org, CppUnit, TESSY, Rational Test Realtime

component. One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components.

Note: The terms “module,” “component,” and “unit” are often used interchangeably or defined to be subelements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.

component testing. Testing of individual hardware or software components or groups of related components. Syn: module testing.

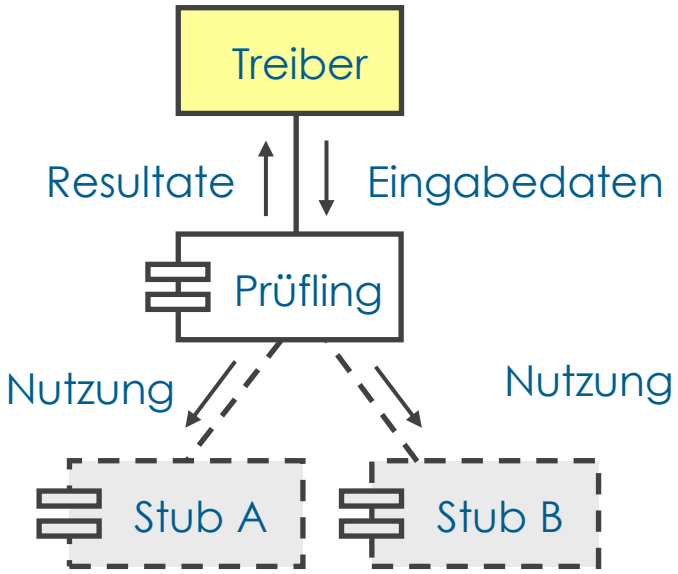
See also: integration testing; interface testing; system testing; unit testing.

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

Komponente vs. Unit

- Der Unit-Test stellt zwar nach IEEE Std 610.12-1990 ein Synonym zum Komponenten-Test dar, in der Praxis wird aber Unit-Test in der Regel im Kontext von Klassen objektorientierter Programme gesprochen.

Testaufbau



Der Testtreiber startet die Ausführung des Prüflings und nimmt Resultate entgegen (Beispiel: JUnit, TTCN-3, Tessy)

Ein Stub (Platzhalter) ersetzt eine vom Prüfling genutzte Funktion

Beispiel: JUnit 4.0

- JUnit ist ein Komponententest-Framework für Java
- Eine beliebige Java-Klasse kann als Testtreiber verwendet werden
- Über Annotationen wird das Verhalten des Testtreibers bestimmt

```
public class MeineTestklasse {

  @Before public void setUp() {
    // Testobjekt initialisieren
  }
  @After public void tearDown() {
    // Belegte Ressourcen wieder freigeben
  }

  @Test public void testMethode1() { ... }
  @Test public void testmethode2() { ... }
}
```

Beispiel: CppUnit – 1

- CppUnit ist die „C/C++“-Variante von JUnit
- http://sourceforge.net/projects/cppunit

```
class MeineTestklasse : public TestFixture
{
    CPPUNIT_TEST_SUITE (MeineTestklasse);
    CPPUNIT_TEST (testfall1);
    CPPUNIT_TEST (testfall2);
    CPPUNIT_TEST (testfalln);
    CPPUNIT_TEST_SUITE_END ();

    public:
        void setUp();
        void tearDown();
    protected:
        void testfall1();
        void testfall2();
        void testfalln();
    private:
        // ... Hier wird in der Regel das Testobjekt referenziert
};
```

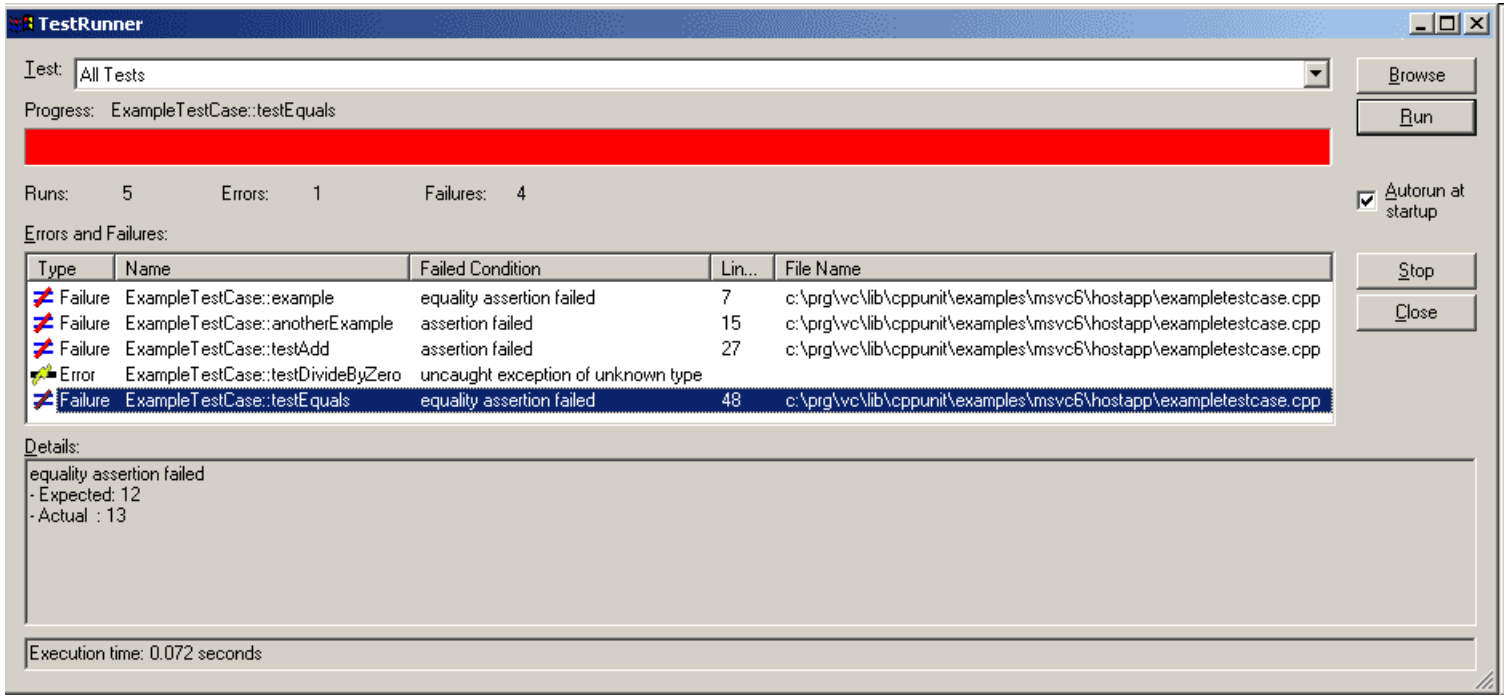
Beispiel: CppUnit – 2

- Beispiel einer Testfall-Implementierung

```
CPPUNIT_TEST_SUITE_REGISTRATION (MeineTestklasse);

void MeineTestklasse::setUp()
{
  // Vorbereitungen treffen; Testobjekt initialisieren
}
void MeineTestklasse::tearDown()
{
  // Freigaben, evtl. das Testobjekt wieder löschen
}

void MeineTestklasse::testfall1()
{
  double sollResultat = ...;

  // Methode methode1 testen
  CPPUNIT_ASSERT_EQUAL (testobjekt->methode1(), sollResultat);
}
...
```

Beispiel: CppUnit – 3

- MFC TestRunner



© 2008-2015 Dr. Rainer Schmidberger

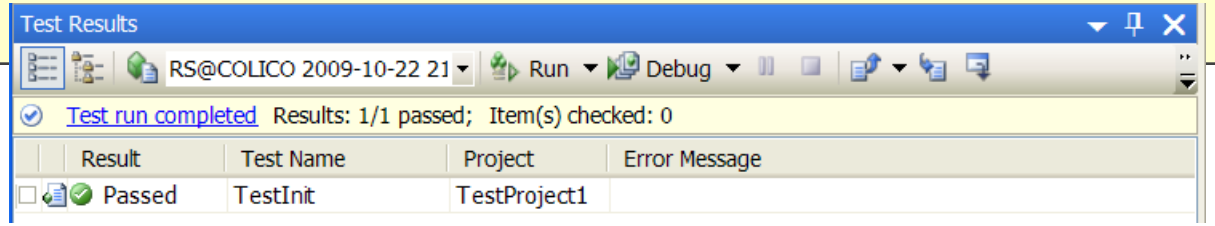
Unit-Test - Beispiel: Klasse „Bruch“

```
public class Bruch {
  private int zaehler;
  private int nenner;
  public override bool Equals(object obj) {
    Bruch b = (Bruch)obj;
    return this.zaehler == b.zaehler && this.nenner == b.nenner;
  }
  public Bruch() {
    nenner = 1;
  }
  public Bruch(int z, int n) {
    zaehler = z;  nenner = n;
  }
  public static explicit operator double(Bruch b) { // cast von Bruch nach double
    return ((double)b.zaehler) / b.nenner;
  }
  static public Bruch operator +(Bruch a, Bruch b) { // hier sitzt der Fehler
    return new Bruch(a.zaehler * b.zaehler + b.zaehler * a.nenner, a.nenner * a.nenner);
  }
  static public Bruch operator !(Bruch a) {
    return new Bruch(a.nenner, a.zaehler);
  }
  public override string ToString() {
    return string.Format("{0} / {1}", zaehler, nenner);
  }
}
```

Unit-Test mit Visual Studio

- Es wird ein Test-Projekt angelegt; das Projekt mit den zu testenden Klassen wird „referenziert“ (Hinweis: die zu testenden Klassen müssen dort `public` sein)
- Testklassen und Testmethoden werden über Attribute gesteuert
- Es ist oft hilfreich, wenn die zu testende Klasse eine `Equals`-Methode implementiert hat

```
[TestClass]
public class UnitTestBruch
{
   [TestMethod]
   public void TestInit()
   {
      Bruch b = new Bruch(1, 2);
      Assert.AreEqual((double)b, 0.5, "Fehler beim Initialisieren");
   }
}
```



Beispiel: Testmethoden für Bruch

- Beispiel Testmethoden

```
[TestMethod]
public void TestAddiere()
{
  Bruch b1 = new Bruch(1, 2);
  Bruch b2 = new Bruch(1, 3);
  Bruch b3 = new Bruch(5, 6);
  Bruch b4 = b1 + b2;
  Assert.AreEqual(b3, b4);
}
[TestMethod]
public void TestKehrwert()
{
  Bruch b1 = new Bruch(1, 2);
  Bruch b2 = new Bruch(2, 1);
  Bruch b3 = !b2;
  Assert.AreEqual(b1, b3);
}
```

Ausgangssituation

Aktion

Resultatsvergleich

Ausgangssituation

Aktion

Resultatsvergleich

© 2008-2015 Dr. Rainer Schmidberger

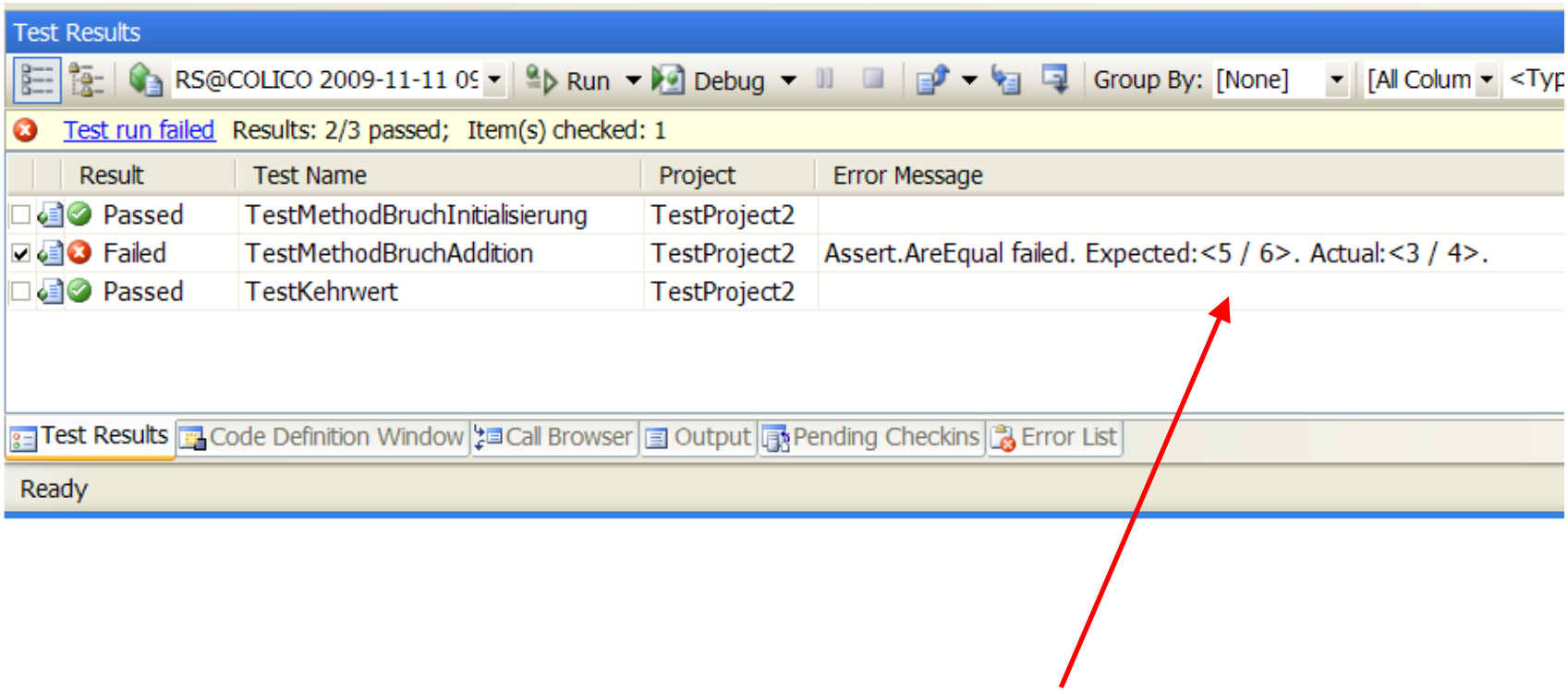
Unit-Test Attribute

- **`[ClassInitialize]`**
  - Wird vor Ausführung des ersten Testfalls `[TestMethod]` aufgerufen. Es können Initialisierungen vor Beginn des Tests vorgenommen werden.
- **`[ClassCleanup]`**
  - Aufräumarbeiten nach Ende aller Testfälle `[TestMethod]`
- **`[TestInitialize]`**
  - Initialisierung, die vor jedem einzelnen Testfall `[TestMethod]` aufgerufen werden
- **`[TestCleanup]`**
  - Aufräumarbeiten, die nach jedem einzelnen Testfall `[TestMethod]` aufgerufen werden
- **`[TestMethod]`**
  - Ein Testfall

© 2008-2015 Dr. Rainer Schmidberger

Beispiel: Test Result für die Klasse Bruch

- Visual Studio 2008



Aufruf der ToString()-Methode

Zusammenfassung Unit-Test

- Die Werkzeugunterstützung und die Verankerung in den (Klassen)-Bibliotheken ist sehr gut.
- Der Fokus liegt auf objektorientierten Programmen, wobei in der Regel Unit=Klasse gilt.
- Bei sehr kleinen Units ist die Chance, Fehler zu finden, eher gering; die Fehler liegen in der Praxis eher in der Interaktion.
- Nicht selten werden zu Diagnose-Zwecken extra öffentliche Methoden aufgenommen.

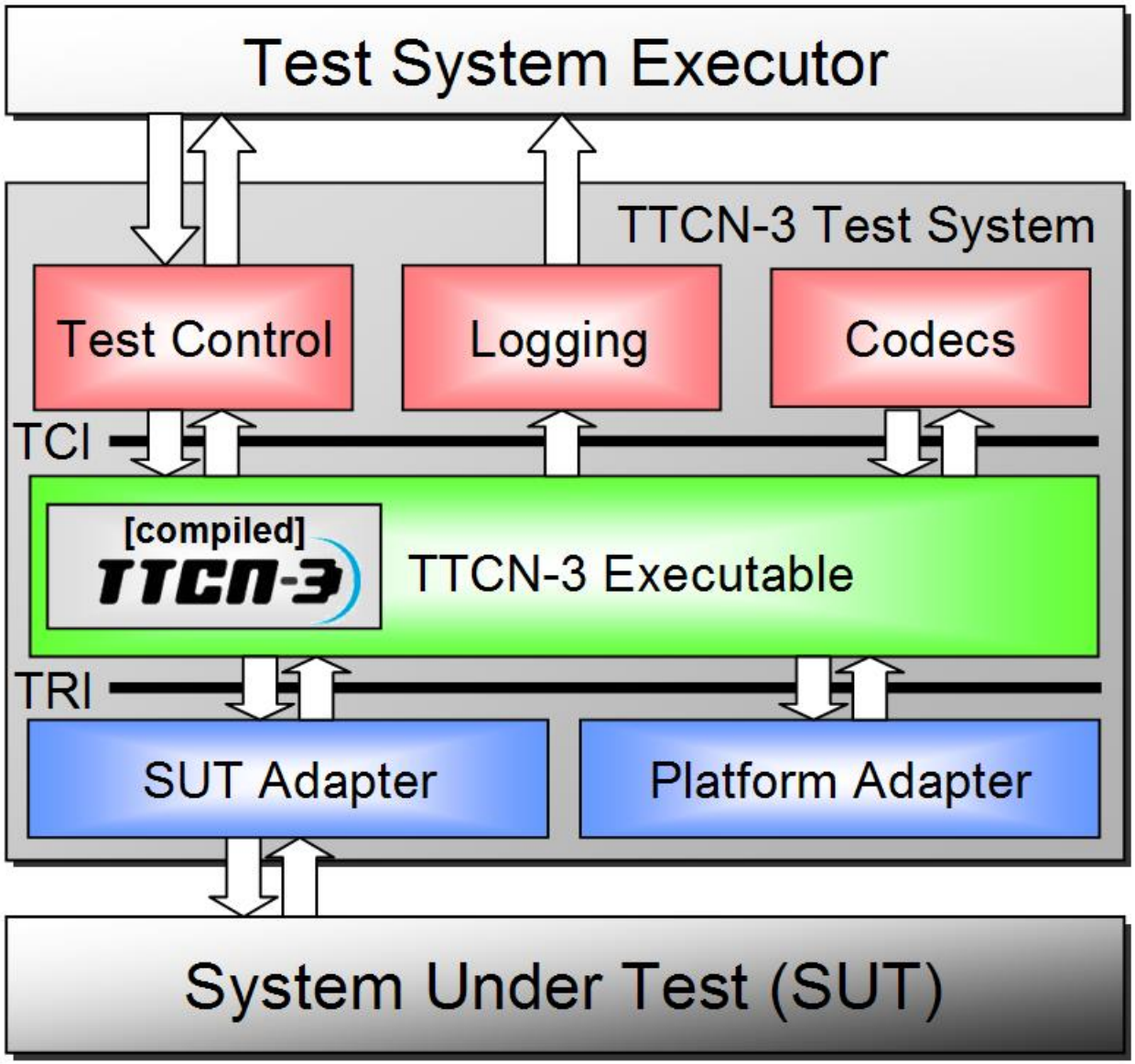
Unit-Tests hinterlassen im Prüfling ihre Spuren!

TTCN-3

- Testing and Test Control Notation
- „Konformitätstesten“
- Von der ETSI (European Telecommunications Standards Institute) standardisiert
- TTCN-3 ist eine Skriptsprache
- TTCN-3 wird für die Automatisierung von Tests speziell für kommunikationsbasierte Systeme verwendet
- TTCN-3 wird beispielsweise auch für das Testen von CAN- und MOST-Bussen verwendet
- Es sind kommerzielle und freie Werkzeuge verfügbar
- Siehe: www.ttcn-3.org

© 2008-2015 Dr. Rainer Schmidberger

TTCN-3 Architektur



Quelle: http://www.ttcn-3.org/Tutorials.htm

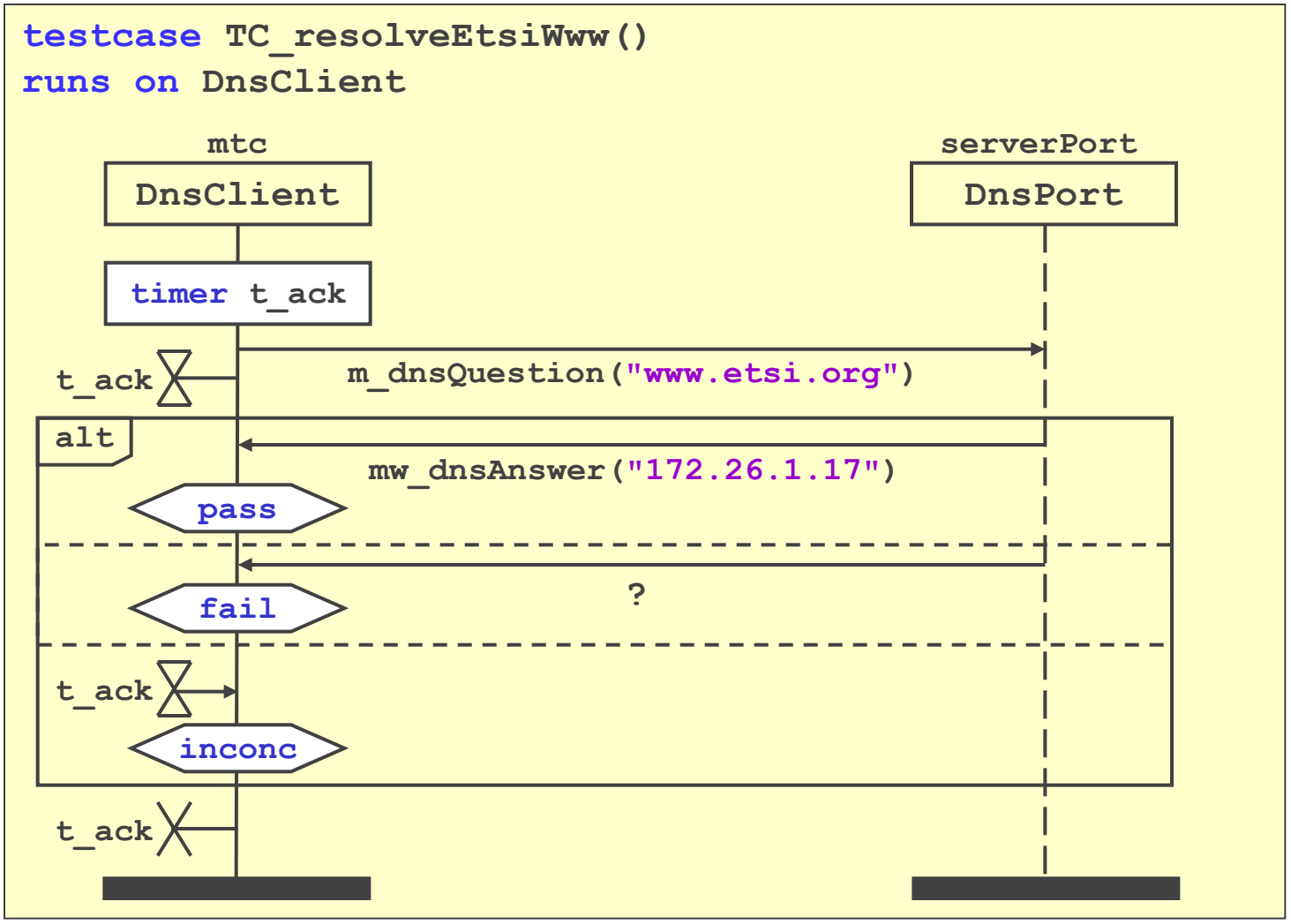
© 2008-2015 Dr. Rainer Schmidberger

TTCN-3: Beispiel – 1

```
testcase TC_resolveEtsiWww() runs on DnsClient
{
    timer t_ack;
    serverPort.send(m_dnsQuestion("www.etsi.org"));
    t_ack.start(1.0);
    alt {
        [] serverPort.receive(mw_dnsAnswer("172.26.1.17")) {
            setverdict (pass);
            }
        [] serverPort.receive { // any other message
            setverdict(fail);
            }
        [] t_ack.timeout {
            setverdict(inconc);
            }
    }
    t_ack.stop;
}
```

Quelle: http://www.ttcn-3.org/Tutorials.htm

TTCN-3: Beispiel – 2



Quelle: http://www.ttcn-3.org/Tutorials.htm

Aus:
TTCN-3 User Conference 2007
Anthony Wiles

http://www.ttcn-3.org/TTCN3UC2007/Presentations/Thu/ETSI%20TTCN-3%20keynote.pdf

Entwurfskriterium: Testbarkeit

- Die Isolierbarkeit einer Komponente ist eine Voraussetzung für Komponententests
- Isolierbare Komponenten entstehen bei Entwurf nicht zwangsläufig – die Isolierbarkeit muss aktiv im Entwurf herbeigeführt werden
- Empfehlung: Testbarkeit sollte bei Entwurfsreviews mit einbezogen werden

© 2008-2015 Dr. Rainer Schmidberger

Test-First-Ansatz

- Es werden zuerst die Testfälle (in der Regel eingebettet in den Testtreiber) implementiert und anschließend der produktive Programmcode
- Test-First entstammt den agilen Vorgehensweisen (Extreme Programming)
- Vorteile
  - QS der Anforderung
  - Automatisierung spart Aufwand
  - Der Test verliert den negativen Beigeschmack
  - Testfälle werden dokumentiert und sind reproduzierbar

- In der Praxis stellt eine unvollständige Komponentenspezifikation ein großes Problem dar!

Zusammenfassung Komponententest

- Test einer einzelnen isolierten Komponente

- Vorteile
  - Der Test einer Komponente ist früh im Entwicklungsprozess möglich
  - Es besteht eine gute Werkzeugunterstützung
  - Testbarkeit ist ein hilfreiches Entwurfskriterium
  - Komponententests sind gut in den Entwicklungsprozess integrierbar
- Nachteile
  - Die Interaktion zwischen den Komponenten wird nicht getestet
  - Z. T. Aufwändiges Erstellen der Stubs

Teststufen

- ✓ Komponententest
- **Integrationstest**
- Systemtest

- Weitere Teststufen
- Test gegen nichtfunktionale Anforderungen

© 2008-2015 Dr. Rainer Schmidberger

Integrationstest

- Der Integrationstest bildet die Brücke zwischen dem Komponententest und dem Systemtest
- Ziel des Integrationstests ist es, Schnittstellen- und Protokollfehler aufzudecken
  - Inkompatible Schnittstellenformate
  - Unterschiedliche Interpretation der übergebenen Daten
  - Timing-Problem: Daten werden richtig übergeben, aber zum falschen oder verspäteten Zeitpunkt oder in zu kurzen Zeitintervallen (Durchsatz- oder Lastproblem).

integration. The process of combining software components, hardware components, or both into an overall system.

integration testing. Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them.

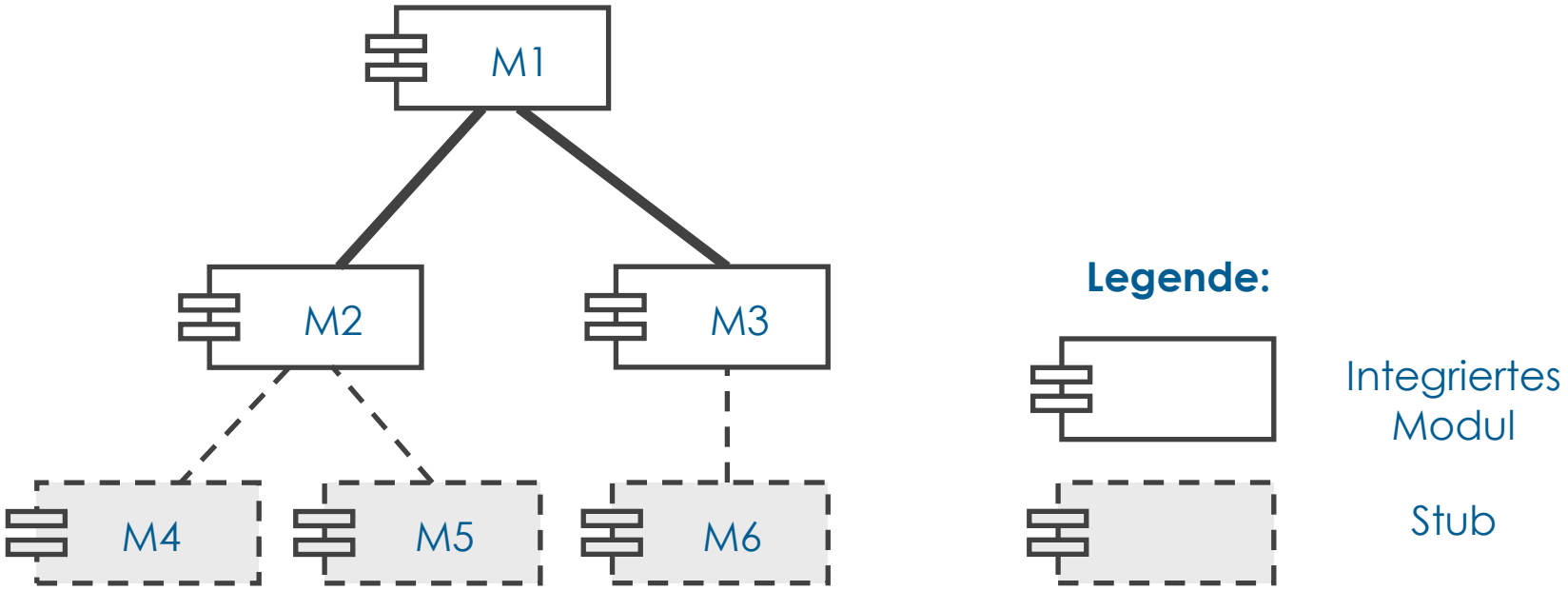
See also: component testing; interface testing; system testing; unit testing.

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

© 2008-2015 Dr. Rainer Schmidberger

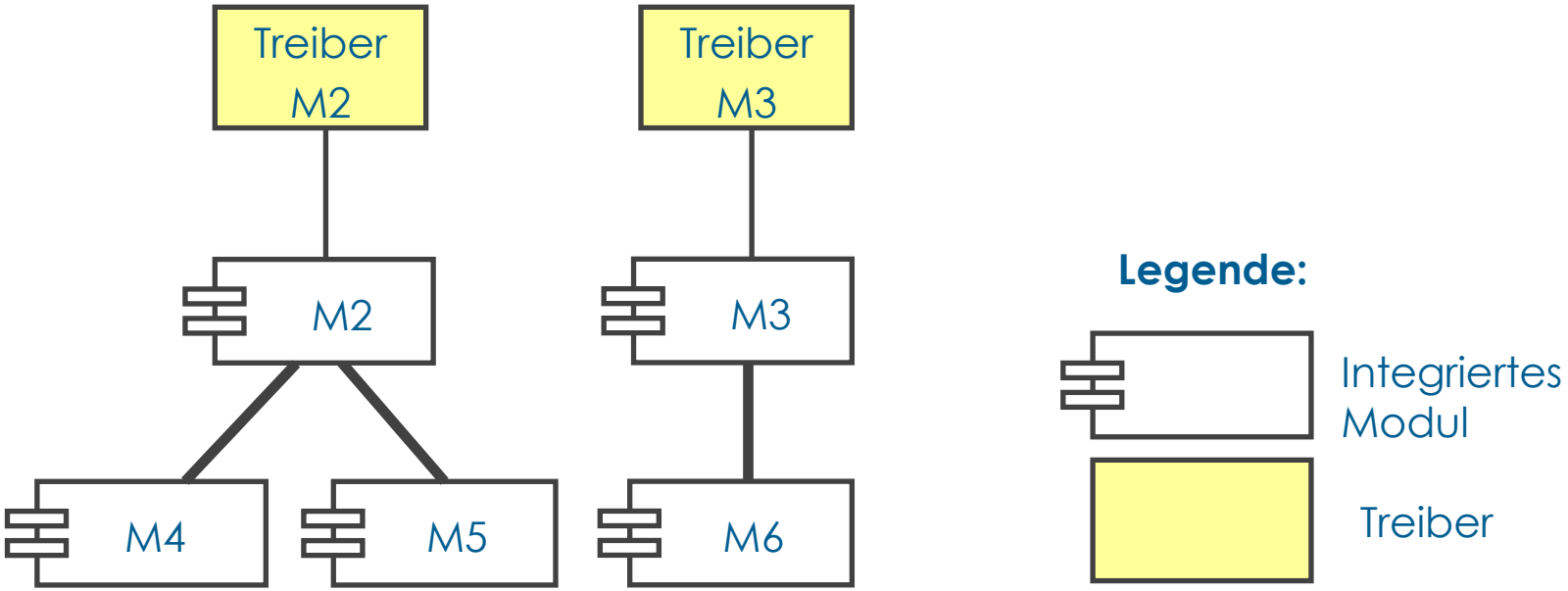
Top-Down-Integration

- Der Test beginnt mit dem „Hauptmodul “
- Vorteil: Keine Testtreiber erforderlich
- Nachteil: Noch nicht integrierte Komponenten müssen durch Platzhalter ersetzt werden.



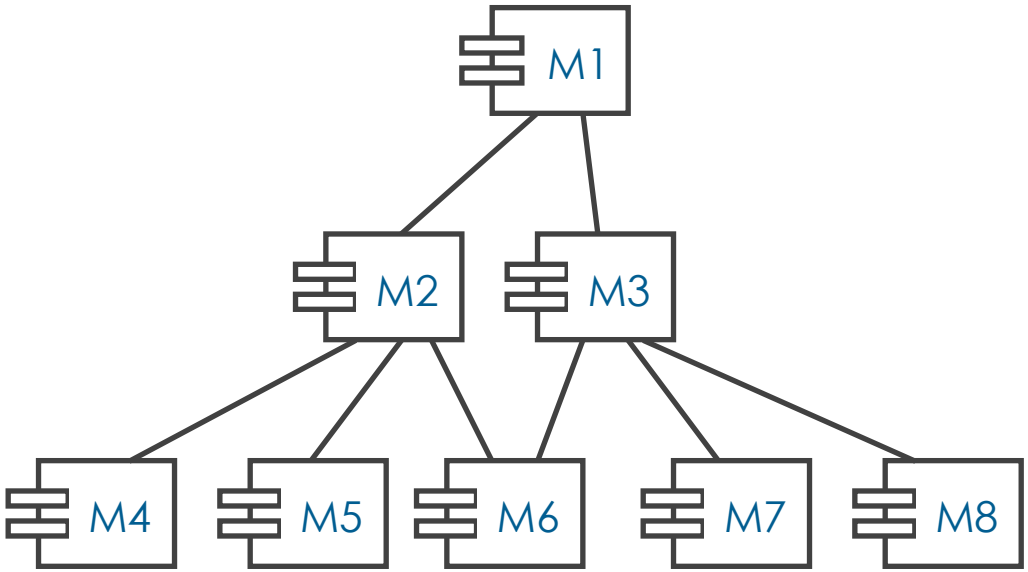
Bottom-Up-Integration

- Der Test beginnt mit dem „unterstem“ Modul
- Vorteil: Keine Platzhalter nötig, „intuitive“ Vorgehensweise
- Nachteil: Übergeordnete Module müssen durch Testtreiber „nachgebildet“ werden. Damit sind viele zusätzliche Testfälle notwendig.



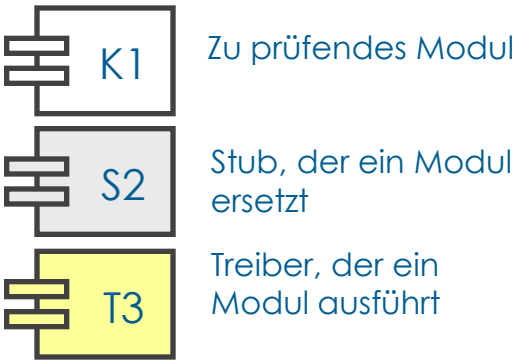
Big-Bang-Integration

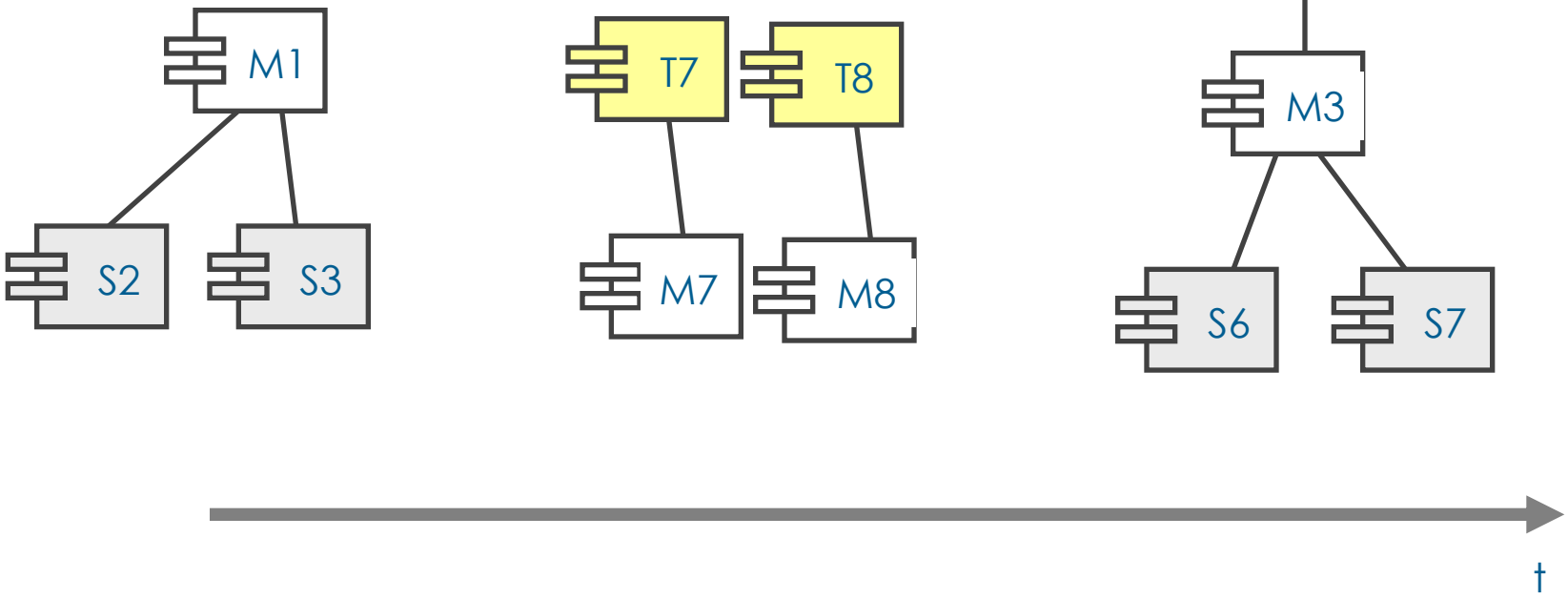
- Der Test beginnt mit dem vollintegrierten System
- Vorteil: Es sind keine Testtreiber und Stubs nötig
- Nachteil: Schwierige Fehlersuche
- Genaugenommen gibt es keine Integrationsstrategie



Ad-hoc-Integration

- Vorteil: Fertiggestellte Module können zeitnah integriert werden.
- Nachteil: Sowohl Testtreiber als auch Platzhalter sind erforderlich





Integrationsstrategien (1)

- Top-Down-Integration
  - Vorteil: Keine Testtreiber erforderlich
  - Nachteil: Noch nicht integrierte Komponenten müssen durch Platzhalter ersetzt werden.
- Bottom-up-Integration
  - Vorteil: Keine Platzhalter nötig
  - Nachteil: Übergeordnete Komponenten müssen durch Testtreiber simuliert werden.
- Ad-hoc-Integration
  - Vorteil: Fertiggestellte Komponenten können zeitnah integriert werden.
  - Nachteil: Sowohl Testtreiber als auch Platzhalter sind erforderlich.
- Big-Bang-Integration

Zusammenfassung Integrationstest

- Test der Interaktion teilintegrierter Komponenten

- Vorteile
  - Aufdecken von Schnittstellenfehlern
  - Test der integrierten Komponenten
  - QS des Entwurfs
  - „How do we eat an elephant?“ – „Bite by bite“

- Nachteile
  - Teilintegration muss in der Regel aufwändig manuell erfolgen
  - Je Teilintegration sind spezielle Testfälle erforderlich
  - Z. T. aufwändige Erstellung der Stubs

Modultest vs. Integrationstest

- Ziel im Modultest: Gründlicher Test möglichst aller Funktionen, die das Modul anbietet
- Ziel im Integrationstest: Gründlicher Test möglichst aller Modul-Interaktionen

- **Achtung:** Die Zielsetzungen sind verschieden! Der eine Test ersetzt damit nicht den anderen!

© 2008-2015 Dr. Rainer Schmidberger

Teststufen

- ✓ Komponententest
- ✓ Integrationstest
- **Systemtest**

- Weitere Teststufen
- Test gegen nichtfunktionale Anforderungen

© 2008-2015 Dr. Rainer Schmidberger

Systemtest

- Test des Gesamtsystems
  - Test des Zusammenspiels aller integrierten Systemkomponenten
  - Test in produktionsnaher Testumgebung
  - Achtung: Der Begriff „System“ ist keines falls eindeutig definiert (SW, SW+HW, SW+HW+Fahrzeug?)

- Testziel
  - Prüfung der spezifizierten Systemeigenschaften

→Kundensicht statt Entwicklersicht

© 2008-2015 Dr. Rainer Schmidberger

System

A collection of components organized to accomplish a specific function or set of functions.

System testing

Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

See also: component testing; integration testing; interface testing; unit testing.

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

Systemgrenzen

- Was gehört alles zum „System“?
  - Die Summe aller Steuergeräte?
  - Ein Steuergerät?
  - Die Software des Steuergeräts?

- Vor Systemtestbeginn (besser vor Projektbeginn) ist diese Frage genau zu klären!

Testziel im Systemtest

- Vollständigkeit
  - Auf Basis der Systemspezifikation
- Leistung bzw. Effizienz
- Zuverlässigkeit und Robustheit
  - Einschl. Misuse-Testfälle
- Sicherheit (Security)
- Benutzbarkeit, Dokumentation

Probleme beim Systemtest

- Testaufbau ist aufwändig
- (blöde, d.h. leicht vermeidbare) Fehler halten den Systemtest immer wieder auf
- Die Fehlerursache ist aufwändig zu finden
- Fehler können Schaden an der Betriebsumgebung anrichten (z.B. bei angeschlossenen Geräten)

→ Um Fehler zu finden ist der Systemtest der ungeeignetste Test! Besser, man findet die Fehler früher.

Test gegen nichtfunktionale Anforderungen – 1

- **Lasttest**: Systemverhalten (Speicherverbrauch, Antwortzeiten) bei steigender Last (z. B. Anzahl parallel arbeitender Anwender, Anzahl Transaktionen).
- **Performanztest**: Messung der Rechenzeit bestimmter Funktionen, in der Regel in Abhängigkeit von steigender Last
- **Volumen-/Massentest**: Systemverhalten bei steigenden Datenmengen
- **Stresstest**: Systemverhalten bei Überlastung
- **Test auf Sicherheit** gegen unberechtigten Systemzugang (siehe www.bsi.de)
- **Test auf Stabilität** und Zuverlässigkeit im Dauerbetrieb

Test gegen nichtfunktionale Anforderungen – 2

- **Test auf Robustheit** gegenüber Fehlbedienung oder Fehlerzuständen der Umgebung, Wiederanlaufverhalten
- **Test auf Kompatibilität**/Datenkonversionen: Prüfung auf Verträglichkeit mit vorhandenen Systemen, Import/Export von Datenbeständen
- **Test unterschiedlicher Konfigurationen**: Betriebssysteme, Hardware, Ländereinstellungen (Dezimalzahlen, Einheiten, Datum), Spracheinstellungen
- **Test auf Benutzungsfreundlichkeit**: Prüfung von Erlernbarkeit und Verständlichkeit, bezogen auf die verschiedenen Benutzergruppen

---

- **Prüfung der Dokumentation**
- **Prüfung auf Änderbarkeit/Wartbarkeit**

Sicherheitslücke bei Fiat Chrysler

Rückruf nach Hacker-Experiment

Stand: 24.07.2015 18:38 Uhr

Nachdem es Computerexperten gelungen war, einen Jeep über das Internet fernzusteuern, hat Fiat Chrysler auf das Hacker-Experiment reagiert. Der Autobauer rief mehr als eine Million Modelle der Marken Jeep, Dodge und Ram zurück.

Der italienisch-amerikanische Autobauer Fiat Chrysler Automobiles hat in den USA 1,4 Millionen PKW zurückgerufen. Grund seien Sicherheitsrisiken bei möglichen Hackerangriffen. Nach Angaben des Konzerns sind verschiedene Modelle der Marken Dodge, Ram und Jeep aus den Jahren 2013 bis 2015 betroffen.

Durch einen Tausch der Software solle verhindert werden, dass Fahrzeuge aus der Ferne manipuliert werden können. Genau das war Sicherheitsexperten in den USA in einem simulierten Hackerangriff mit einem Jeep gelungen.

Radio-Software soll aktualisiert werden

Die Hacker waren über eine Sicherheitslücke im Unterhaltungssystem, das mit dem Internet verbunden ist, bis zur Steuerung des Wagens vorgedrungen. Sie konnten die Bremsen, Geschwindigkeit, Klimaanlage und Radio des Jeep Cherokee fernsteuern.

Das Unternehmen kündigte an, die Software für die Radios zu aktualisieren. Es seien jedoch bislang keine Schadensfälle durch unerlaubtes Eindringen in diese Software bekannt geworden.

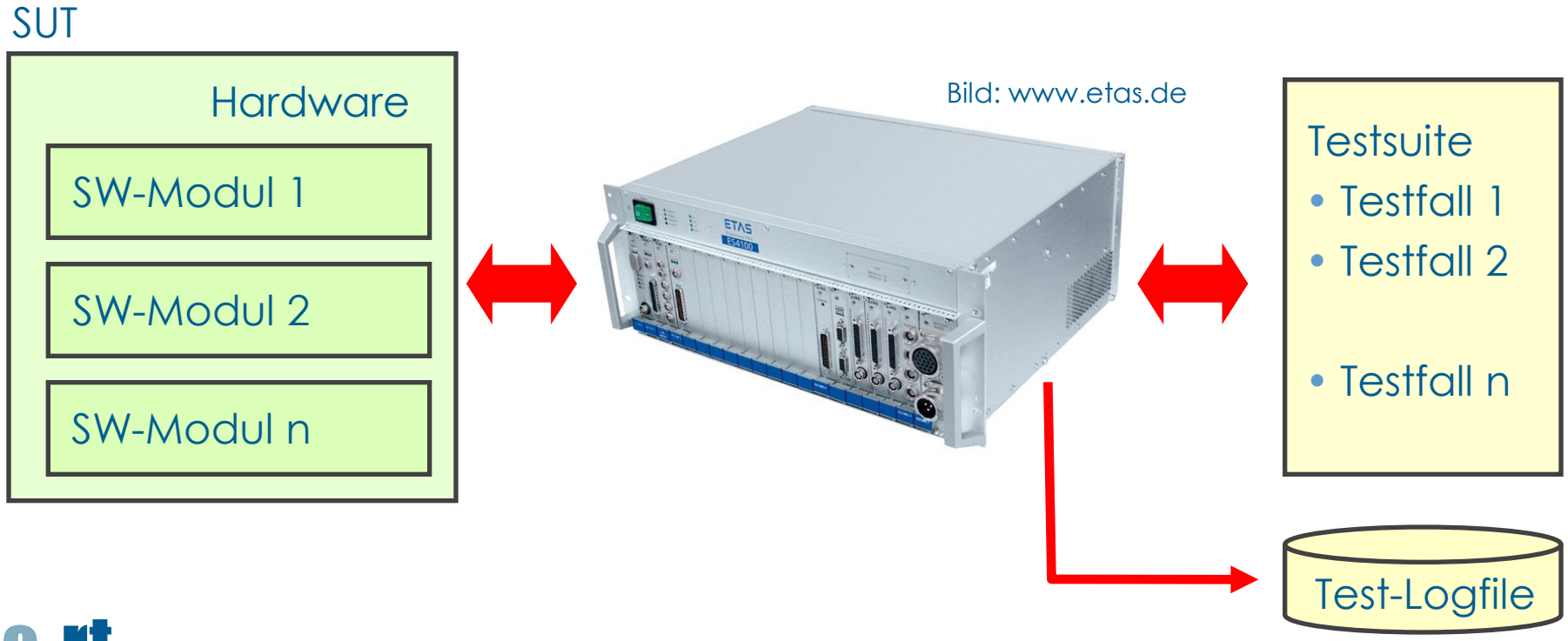
Quelle: Tagesschau.de, 24.07.2015

Penetrationstest

- Prüfung der Sicherheit möglichst aller Systembestandteile eines Netzwerks- oder Softwaresystems mit Mitteln und Methoden, die ein Angreifer (ugs. „Hacker“) anwenden würde, um unautorisiert in das System einzudringen (Penetration).
- Der Penetrationstest ermittelt somit die Empfindlichkeit des zu testenden Systems gegen derartige Angriffe. Wesentlicher Teil eines Penetrationtests sind Werkzeuge, die dabei helfen, möglichst alle Angriffsmuster nachzubilden, die sich aus den zahlreichen bekannten Angriffsmethoden herausbilden.

Systemtest: HiL

- Test realer Steuergeräte (-verbünde)
  - Zielsoftware läuft auf Zielhardware
  - Testen in geschlossenen Regelschleifen
- Testsystem arbeitet Testsuite ab
  - Vorspielen von Dateien mit Verläufen der Stimuli
  - Aufzeichnen von Dateien mit Verläufen der Reaktionen



HiL-Tests

- Vorteile von HiL-Tests
  - Testabdeckung auch wenn reale Umgebung fehlt / nicht verfügbar (z. B. Glatteis)
  - Gezielte Herbeiführung von Grenzsituationen
  - Automatisches Durchschreiten von Parametersätzen
  - Nachbildung von im Feldversuch aufgetretenen Auffälligkeiten
- Probleme
  - Aufwändige (teure) Hardware
  - Testskripte müssen programmiert werden (wer testet diese Skripte?)
  - In der Praxis gibt es eine hohe Zahl an False-Positives
  - Spezifikation von Sollverhalten und Testauswertung ist aufwändig

Automatisch ausgeführter Test

- Effizienz bei häufiger Ausführung
- „Austesten“ – Durchlaufen von Wertebereichen
- Reproduzierbarkeit
- Viele gute Tools sind verfügbar, auch Open-Source Tools
- Probleme mit Falsch-Positiven
- Programmierung und Wartung aufwändig

vs.

Manuell ausgeführter Test

- Schnell einsetzbar, kann direkt vom Fachbereich durchgeführt werden, keine „Starthürde“
- Hohe Effektivität: Erkennen von Fehlwirkungen, die nicht zum spezifizierten Soll-Resultat gehören
- „Unempfindlich“ gegenüber Oberflächenänderungen
- Fehlerwirkungen sind in der Regel unmittelbar fachlich plausibel
- Aufwändig und Tester-Kompetenz muss vorhanden sein

Zusammenfassung Systemtest

- Test des Gesamtsystems gegen die Systemanforderungen

- Vorteile
  - Bedeutsamste Teststufe
  - Test aller Systemkomponenten

- Nachteil
  - Aufwändige Fehlersuche
  - Test erst spät im Entwicklungsprozess möglich
  - Entwurfsänderungen sind nur schwer umsetzbar

© 2008-2015 Dr. Rainer Schmidberger

Regressionstest

- Test auf (unbeabsichtigte Neben-) Effekte bei Korrektur, Anpassung oder Erweiterung.
- „Verhält sich das Programm noch so, wie es sich vor der Modifikation verhalten hatte?“
- Verwendung in der Regel in der Wartung
- Eingabedaten können aus den Betriebsdaten gewonnen werden, Sollresultate werden aus der Version vor der Änderung gewonnen.

regression testing: Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [IEEE Std 610.12 (1990)]

Teststufen

- ✓ Komponententest
- ✓ Integrationstest
- ✓ Systemtest

- **Weitere Teststufen**
  - User Acceptance Test (UAT)
  - Beta-Test
  - Abnahmetest
  - Deployment-Test
- Test gegen nichtfunktionale Anforderungen

© 2008-2015 Dr. Rainer Schmidberger

Weitere Teststufen – 1

- User Acceptance Test (UAT)
  - Meinungen bzw. Bewertungen der Benutzer einholen
  - Resultate sind in der Regel nicht reproduzierbar
- Beta-Test
  - Erprobung durch ausgewählte Benutzer

© 2008-2015 Dr. Rainer Schmidberger

Abnahmetest

- Der **Abnahmetest** ist ein spezieller Systemtest
  - Test auf vertragliche Konformität (juristische Relevanz)
  - In der Regel durch den Kunden/Auftraggeber

- Mögliche Resultate
  - ⇨ Abnahmebescheinigung
  - ⇨ Nachbesserungs-forderungen
  - ⇨ Rückgabe bzw. Minderung

Weitere Teststufen – 2

- Deployment-Test (Test auf Installierbarkeit)
  - Test der Installierbarkeit
  - Die für die Installation zugesicherten Installationsumgebungen werden getestet
  - Virtuelle Umgebungen können den Aufwand zur Bereitstellung der nötigen Testumgebung deutlich reduzieren
- Update-Test
  - Ähnlich dem Deployment-Test
  - Mögliche Ausgangsversion sind zu berücksichtigen
- Test auf Deinstallierbarkeit

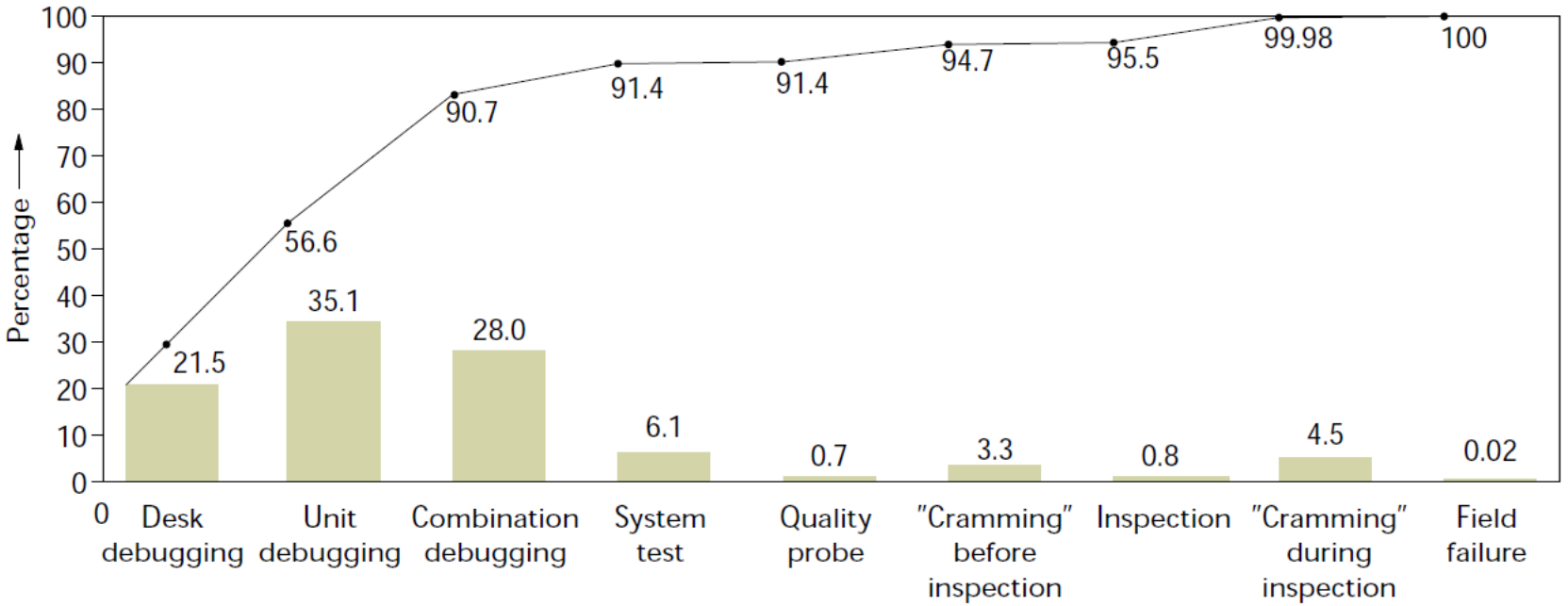
© 2008-2015 Dr. Rainer Schmidberger

Weitere Teststufen – 3

- Smoke-Test: „Vorabtest“, der prüft, ob der Prüfling betriebs- und testbereit ist

© 2008-2015 Dr. Rainer Schmidberger

Fehlerentdeckung über die Teststufen



- Hitachi Software Engineering, 12 Monate, 10 Entwickler, 1990
- 100 kLOC C-Code
- 10.000 Testfälle
- " …We do not use sophisticated tools or state-of-the-art methodology—we simply test programs and fix the bugs detected. …"

Tsuneo Yamaura, Hitachi Software Engineering: "How To Design Practical Test Case", IEEE Software November/ December 1998

© 2008-2015 Dr. Rainer Schmidberger

Kapitel 3
Testfallentwurfsverfahren

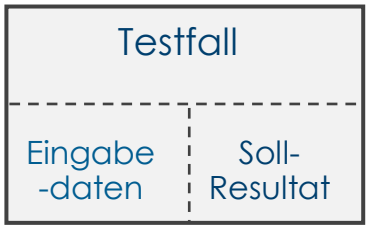
- **Spezifikationsbasierter Test**
- Glass-Box-Test
- Erfahrungsbasierte Verfahren

© 2008-2015 Dr. Rainer Schmidberger

Spezifikationsbasierter Test - Übersicht

- Das SUT wird von außen, an den System-schnittstellen, betrachtet (→ Blackbox-Test)
- Grundlage zur Herleitung der Testfälle bildet die Spezifikation
- Methoden zur Herleitung der Testfälle :
  - Anforderungsbasiertes Testen
  - Äquivalenzklassen-Methode
  - Grenzwertanalyse
  - Klassifikationsbaummethode
  - Entscheidungstabellentest
  - Zustandsbasierter Test
  - Anwendungsfallbasierter Test
  - Zufallstest
  - Modellbasierter Test

Was ist ein Testfall?



***Testfall** (nach [ISTQB]) umfasst folgende Angaben:*

- *die für die Ausführung notwendigen **Vorbedingungen**,*
- *die Menge der **Eingabewerte** (ein Eingabewert je Parameter des Testobjekts),*
- *die Menge der vorausgesagten **Ergebnisse**, sowie die erwarteten Nachbedingungen.*

Testfälle werden entwickelt im Hinblick auf ein bestimmtes Ziel bzw. auf eine Testbedingung, wie z.B. einen bestimmten Programmpfad auszuführen oder die Übereinstimmung mit spezifischen Anforderungen zu prüfen (wie Eingaben an das Testobjekt zu übergeben und Sollwerte abzulesen sind) [nach IEEE 610].

Testfallentwurfsverfahren

- Ein Testfall ist gut, wenn er mit hoher Wahrscheinlichkeit einen noch nicht entdeckten Fehler aufzeigt
- Ein idealer Testfall ist
  - **Repräsentativ**: Er steht stellvertretend für viele andere Testfälle
  - **Fehlersensitiv**: Er hat nach der Fehlertheorie eine hohe Wahrscheinlichkeit, einen Fehler aufzuzeigen
  - **Redundanzarm**: Er prüft nicht, was auch andere Testfälle schon prüfen

- Wir testen bevorzugt …
  - was eine hohe Fehlerdichte hat oder wo wir Fehler am ehesten vermuten
  - wo Fehler besonders viel Schaden anrichten
  - was geringen Aufwand verursacht

© 2008-2015 Dr. Rainer Schmidberger

Anforderungsbasiertes Testen

- Die Grundlage bildet die Anforderungsspezifikation
- Für jede einzelne Anforderung werden Testfälle erstellt (Anforderungsüberdeckung)
- Eine tabellarische Anforderungsspezifikation ist hierfür vorteilhaft
- Requirement-Management-Werkzeuge unterstützen in der Regel das anforderungsbasierte Testen

- Vorteil: Wenn systematisch Testfälle zu den Anforderungen erfasst werden, findet damit auch eine Qualitätssicherung der Anforderungen statt

Ausflug: Requirements Management

- *Nach Auslösen der Wegfahrsperre kann ein Unbefugter nicht mehr starten.*
  - Fragen: Wie wird befugt/unbefugt genau unterschieden?
- *Jeder Passagier hat immer die Kontrolle über das ihm zugeordnete Fenster.*
  - Fragen: Was ist mit der Kindersicherung?
- *Bei einer Störung auf dem Bus, wird die Fernbedienung deaktiviert.*
  - Fragen: Was ist eine Busstörung? Wer deaktiviert für wie lange?
- *Die Diagnosedaten sollen dem Benutzer angezeigt werden.*
  - Fragen: Welche Daten werden welchem Benutzer wie angezeigt?

Quelle: Huhn, Goltz, Prammer: Anforderungsmanagement

Beispiel: Doors Anforderungsdarstellung

- Jede Anforderung erhält eine eindeutige ID
- Anforderungen lassen sich hierarchisch strukturieren
- Textuelle Beschreibung
- In der Regel wird das Tool Doors verwendet

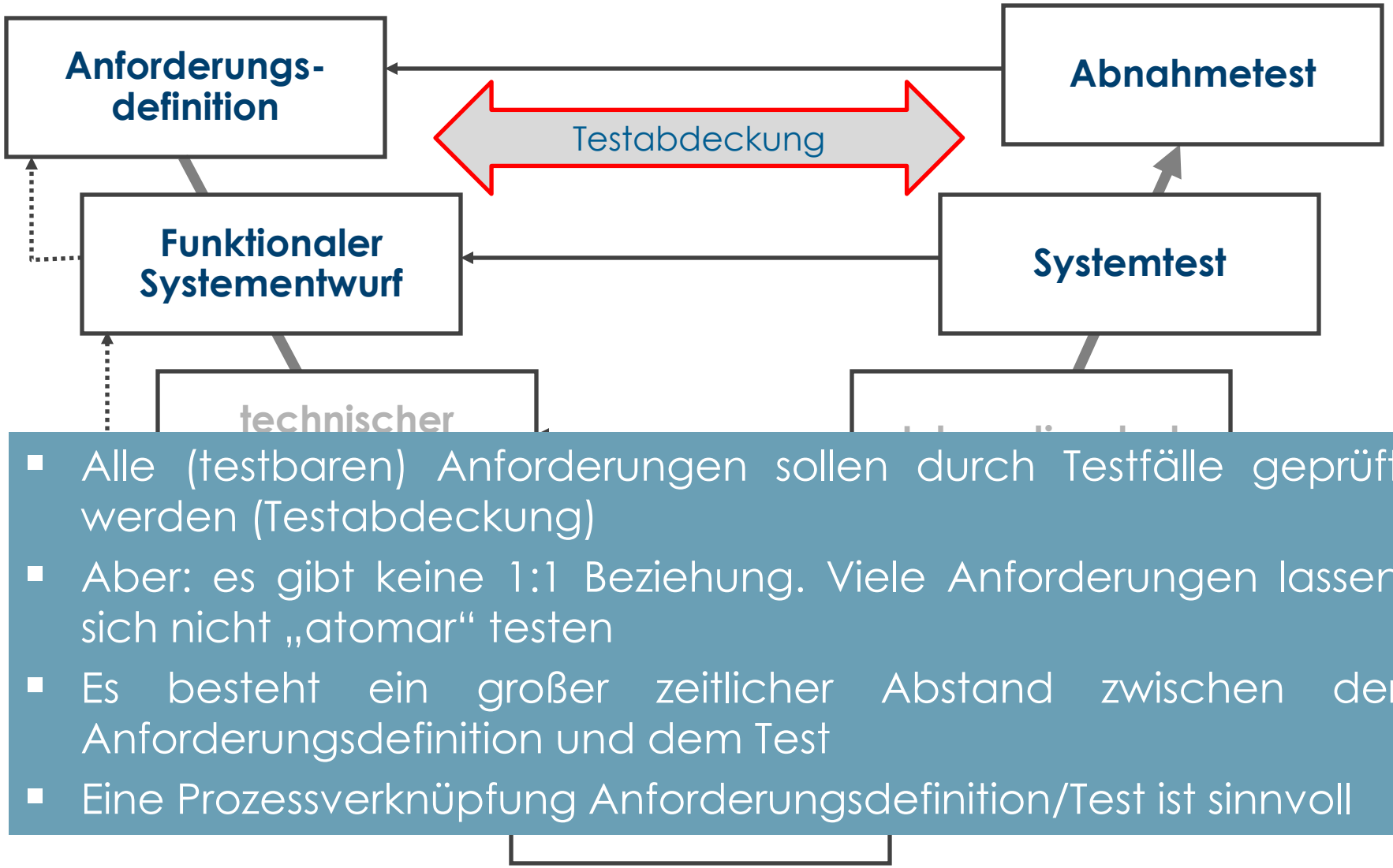
ID	Anforderungen an SMR auf Systemebene
SYA2	**1 Featuregruppe: Wischen und Waschen** *Nachfolgende Informationen dienen als Beispiel ohne jeden Serienbezug*
SYA109	**> 1.1 Globale Vorbedingungen**
SYA232	**> 1.2 Feature: Wischerabschaltung** In Parkstellung verharrender Wischer
SYA10	**> 1.3 Feature: Kontinuierliches Wischen** Gleichmäßige Bewegung des Wischers ohne Pausen beim Reversieren in zwei verschiedenen Geschwindigkeitsstufen
SYA1	**> 1.4 Feature: Sensorgesteuertes Wischen** Steuerung der Wischintervalle und der Wischgeschwindigkeit unter Berücksichtigung der tatsächlichen Regenmenge
SYA11	**> 1.5 Feature: Intervallwischen** Von Intervallpausen fester Länge unterbrochene Wischvorgänge
SYA39	**> 1.6 Feature: Tippwischen** Durch temporäres Betätigen des LSS durch den Fahrer aktiv initiierte Wischvorgänge
SYA12	**> 1.7 Feature: Waschen mit Nachwischen** Der Waschvorgang benetzt die Scheibe mit Waschflüssigkeit. Als Nachwischen werden dann die Wischvorgänge bezeichnet, die in Folge des Waschens initiiert werden. Ziel ist die gewaschene Scheibe wieder von der Waschflüssigkeit zu befreien und dabei die Scheibe zu reinigen.
SYA272	**> 1.8 Feature: Rücksetzen** Bewegen des Wischers in die Parkstellung auf dem *kürzesten* Weg (vgl. "Fertigwischen")
SYA263	**> 1.9 Feature: Fertigwischen** Abschluss eines aktuell durchgeführten Wischvorganges bis zur Parkstellung (im Gegensatz zum Rücksetzen nicht unbedingt auf dem kürzesten Weg)
SYA222	**> 1.10 Feature: Loslaufunterdrückung** Unterdrückung eines nicht-kontinuierlichen, automatisch initiierten Wischvorganges, mit dem Ziel, Personenschäden durch einen plötzlich startenden Wischer zu vermeiden (z.B. bei manueller Reinigung der Scheibe).
SYA241	**> 1.11 Feature: Anlasserunterbrechung** Unterbrechung der laufenden Wischersteuerung im Moment des Motorstarts mit dem Ziel in diesem

Quelle: Grochtmann, TAV 2005

Anforderungsbasierter Tests

- Vorteil
  - Vollständigkeitskriterium: Sind alle Anforderungen „angemessen“ abgedeckt
  - Rückverfolgung: Welche Anforderung ist durch einen Fehlerfall verletzt
  - Änderung der Anforderung: Welche Testfälle sind von Programmänderungen betroffen?
- Nachteil
  - Es wird nur getestet, was auch in den Anforderungen steht. „Common sense“ kann dabei unberücksichtigt bleiben, ebenso indirekte technische Anforderungen.

Anforderungen und Testfälle



- Alle (testbaren) Anforderungen sollen durch Testfälle geprüft werden (Testabdeckung)
- Aber: es gibt keine 1:1 Beziehung. Viele Anforderungen lassen sich nicht „atomar“ testen
- Es besteht ein großer zeitlicher Abstand zwischen der Anforderungsdefinition und dem Test
- Eine Prozessverknüpfung Anforderungsdefinition/Test ist sinnvoll

© 2008-2015 Dr. Rainer Schmidberger

Zuordnung Anforderungen / Testfälle

- Auf welcher Ebene sollen Anforderungen und Tests zugeordnet werden
  - Gruppe von Anforderungen ⇔ Gruppe von Testfällen (effizient aber grob)

Anforderungen

1. ESP: …

2. ABS:

2.1 ABS soll das Blockieren der Räder verhindern, unabhängig von den Reibewerten der Straße.

2.2 ABS soll die Lenkfähigkeit des Fahrzeugs beim Bremsen sicherstellen

Tests

1. ESP-Testfälle …

2. ABS-Testfälle

- Vollbremsung auf glatter Straße, dabei scharf nach links lenken
- Bei sanfter Bremsung nach rechts lenken
- Vollbremsung mit links/rechts unterschiedlichen Reibewerten

Quelle: Grochtmann, TAV 2005

Zuordnung Anforderungen / Testfälle

- Auf welcher Ebene sollen Anforderungen und Tests zugeordnet werden
  - Gruppe von Anforderungen ⇔ Gruppe von Testfällen (effizient aber grob)
  - Einzelne Anforderung ⇔ einzelne Testfälle (naheliegend aber nicht immer angemessen)

Anforderungen

1. ESP: …
2. ABS:

2.1 ABS soll das Blockieren der Räder verhindern, unabhängig von den Reibewerten der Straße.

2.2 ABS soll die Lenkfähigkeit des Fahrzeugs beim Bremsen sicherstellen

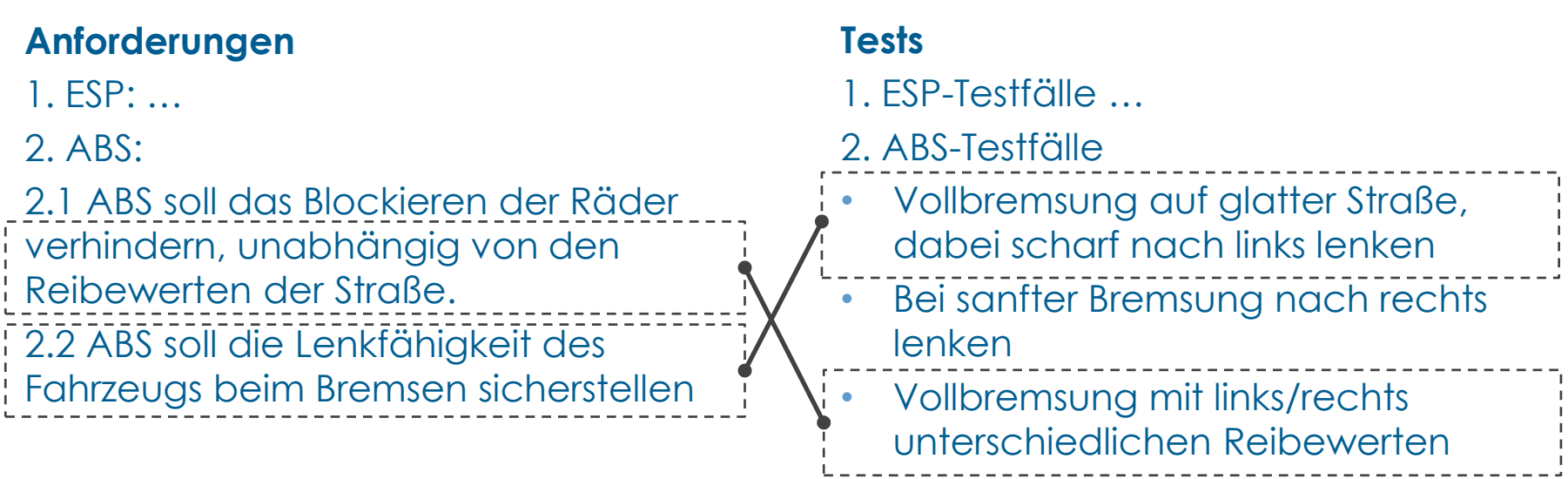
Tests

1. ESP-Testfälle …
2. ABS-Testfälle
   - Vollbremsung auf glatter Straße, dabei scharf nach links lenken
   - Bei sanfter Bremsung nach rechts lenken
   - Vollbremsung mit links/rechts unterschiedlichen Reibewerten

Quelle: Grochtmann, TAV 2005

Zuordnung Anforderungen / Testfälle

- Auf welcher Ebene sollen Anforderungen und Tests zugeordnet werden
  - Gruppe von Anforderungen ⇔ Gruppe von Testfällen (effizient aber grob)
  - Einzelne Anforderung ⇔ einzelne Anforderung (naheliegend aber nicht immer angemessen)
  - Teile von Anforderungen ⇔ Teile von Testfällen (realistisch aber aufwändig)



Quelle: Grochtmann, TAV 2005

Äquivalenzklassenmethode

- Die Äquivalenzklassenbildung hat zum Ziel mit möglichst wenig Testfällen möglichst wirkungsvoll zu testen.
- Die Spezifikation wird nach Eingabegrößen und deren Gültigkeitsbereichen durchsucht
  - Je Gültigkeitsbereich wird eine Äquivalenzklasse definiert
  - Tipp: Wann immer man vermutet, dass Eingabe-werte innerhalb einer Äquivalenzklasse nicht gleichartig behandelt werden, sollte entsprechend in mehrere Äquivalenzklassen aufgeteilt werden.
  - Je „Mitte“ einer Äquivalenzklasse wird ein beliebiges Element **- ein Repräsentant -** als Testfall ausgewählt.
- Vollständigkeit: Alle Repräsentanten sind getestet **(Äquivalenzklassenüberdeckung)**

© 2008-2015 Dr. Rainer Schmidberger

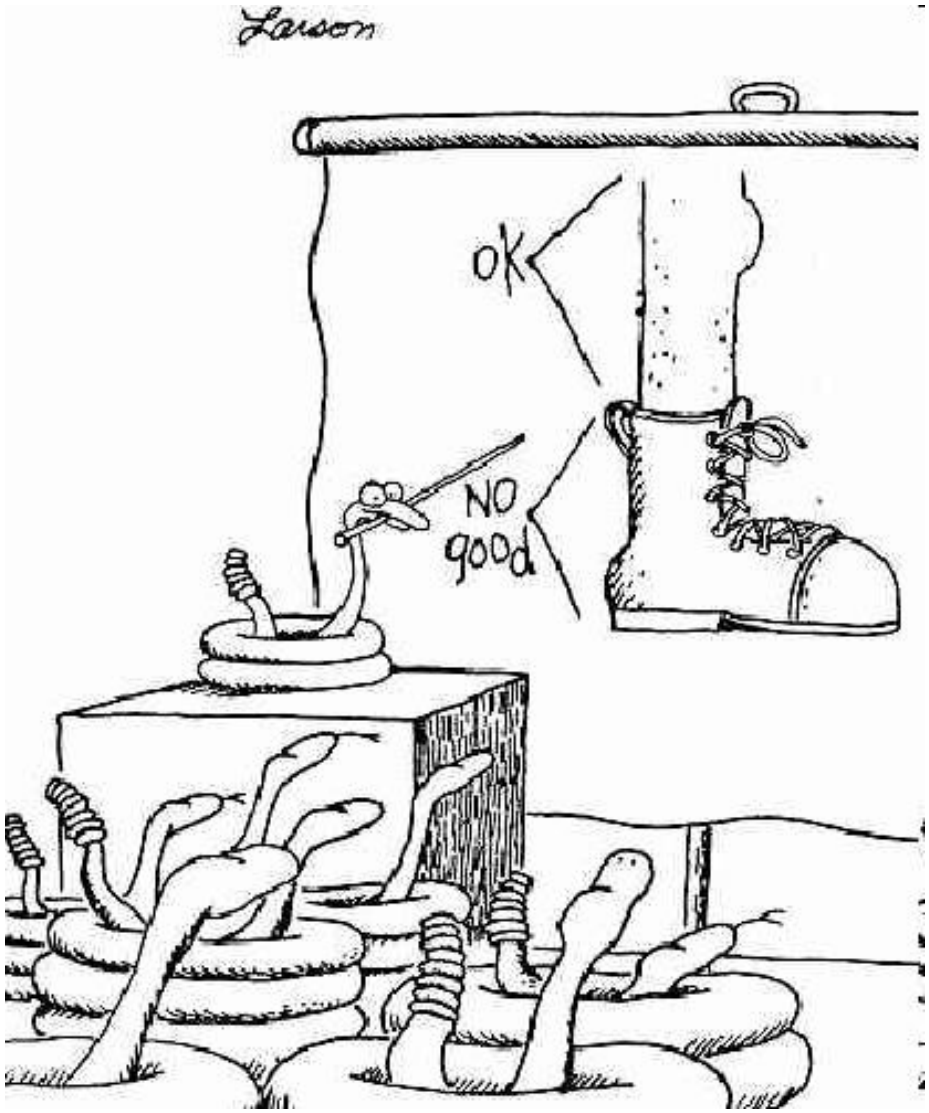
Beispiel zu Äquivalenzklassen

- Ein Sensor meldet an ein Programm die Motor-öltemperatur. Ist die Öltemperatur (T) unter 50°C soll eine blaue Kontrolllampe leuchten, bei T über 150 °C soll eine rote Kontrolllampe leuchten. Sonst soll keine der Lampen leuchten.

Äquivalenzklasse		Soll-Resultat	Repräsentant
1	T < 50°C	blaue Lampe leuchtet	17°C
2	50 <= T <= 150°C	keine Lampe leuchtet	97°C
3	T > 150°C	rote Lampe leuchtet	159°C

- Wenn ein „Verdacht“ besteht, dass z. B. ab Frost mit einem speziellen Verhalten des Sensors oder des Programms zu rechnen ist, so wird die Äquivalenzklasse 1 entsprechend in weitere Äquivalenzklassen aufgeteilt.

Äquivalenzklassenbildung in der Schlangenschule



Quelle: Holger Schlingloff

© 2008-2015 Dr. Rainer Schmidberger

Grenzwertanalyse

- Aufdeckung von Fehlern im Zusammenhang mit der Behandlung der Grenzen von Wertebereichen
- Im Unterschied zur Äquivalenzklassenbildung wird kein beliebiger Repräsentant der Klasse als Testfall ausgewählt, sondern Repräsentanten an den „Rändern“ der Klassen
- Die Methode ist dann anwendbar, wenn die Menge der Elemente, die in eine Äquivalenzklasse fallen, auf natürliche Weise geordnet werden kann
- Hintergrund der Grenzwertanalyse: In Verzweigungen und Schleifen gibt es oft Grenzwerte, für welche die Bedingung gerade noch zutrifft (oder gerade nicht mehr). Diese Grenzwerte sollen getestet werden.

Grenzwertanalyse: Testfälle

- Bereiche
  - Werte auf den Grenzen
  - Werte „rechts“ bzw. „links neben“ den Grenzen (ungültige Werte, kleiner bzw. größer als Grenze)
- Mengen (z.B. bei Eingabedaten, Datenstrukturen, Beziehungen)
  - Kleinste und größte gültige Anzahl
  - Zweitkleinste und zweitgrößte gültige Anzahl
  - Kleinste und größte ungültige Anzahl (oft: leere Menge)
- Bei strukturierten Daten kartesisches Produkt der Grenzwerte
  - Achtung, kombinatorische Explosion!

Beispiel zur Grenzwertanalyse

- Wie im Beispiel zu den Äquivalenzklassen: Ein Sensor meldet an ein Programm die Motoröltemperatur. Ist die Öltemperatur (T) unter 50°C soll eine blaue Kontrolllampe leuchten, bei T über 150 °C soll eine rote Kontrolllampe leuchten. Sonst soll keine der Lampen leuchten.

Grenzwerte		**Soll-Resultat**
Testfall 1	T =49°C	blaue Lampe leuchtet
Testfall 2	T =50°C	keine Lampe leuchtet
Testfall 3	T =51°C	keine Lampe leuchtet
Testfall 4	T = 149°C	keine Lampe leuchtet
Testfall 5	T = 150°C	keine Lampe leuchtet
Testfall 6	T = 151°C	rote Lampe leuchtet

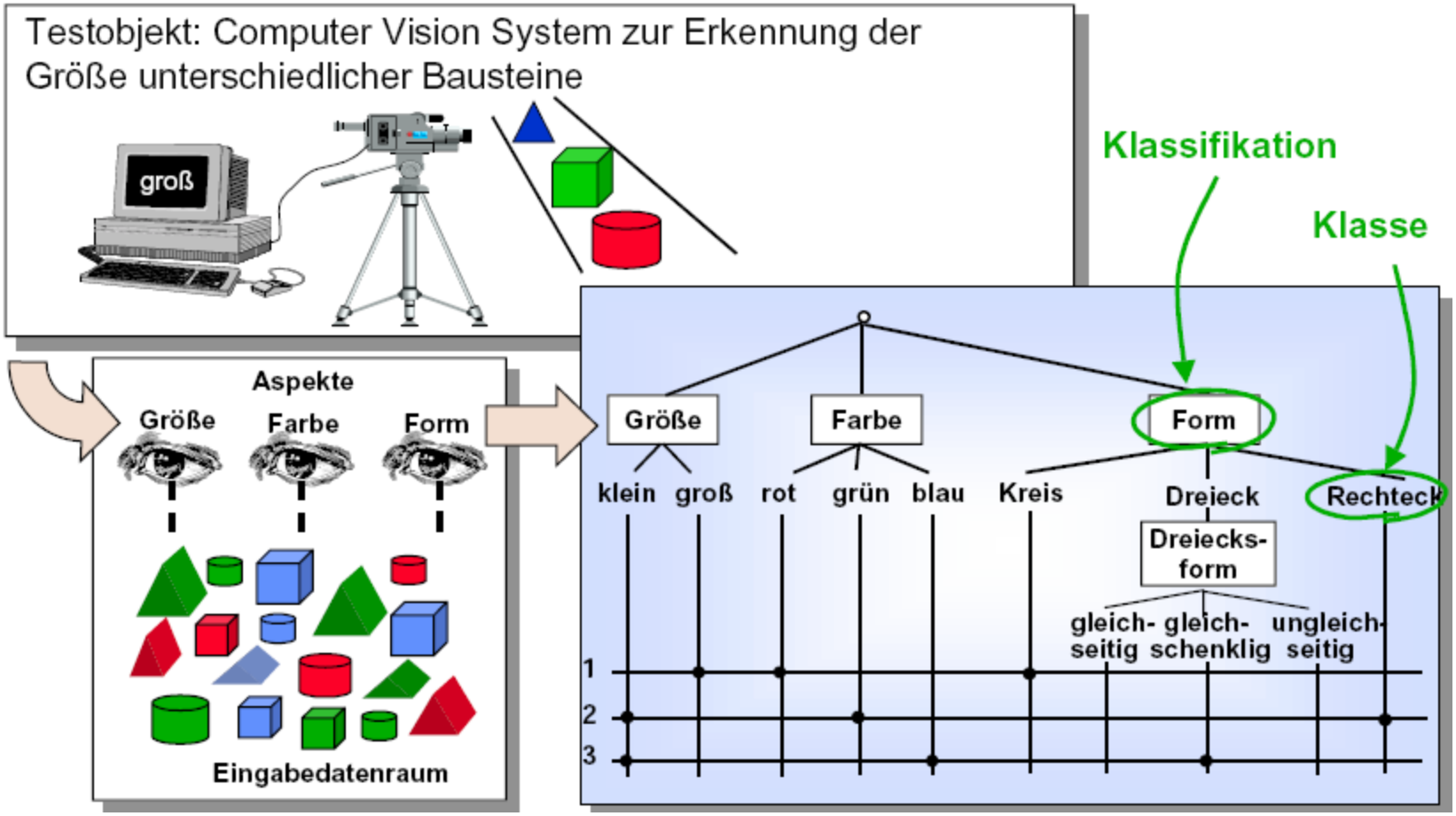
© 2008-2015 Dr. Rainer Schmidberger

Grenzwertanalyse Vor- und Nachteile

- Vorteile
  - An den Grenzen von Äquivalenzklassen sind häufiger Fehler zu finden als innerhalb dieser Klassen
  - "Die Grenzwertanalyse ist bei richtiger Anwendung eine der nützlichsten Methoden für den Testfallentwurf“ (Myers)
  - Effiziente Kombination mit anderen Verfahren, die Freiheitsgrade in der Wahl der Testdaten lassen
- Nachteile
  - Rezepte für die Auswahl von Testdaten schwierig anzugeben
  - Bestimmung aller relevanten Grenzwerte schwierig
  - Kreativität zur Findung erfolgreicher Testdaten gefordert
  - Oft nicht effizient genug angewendet, da sie zu einfach erscheint

Klassifikationsbaummethode

- Behandelt das Problem der kombinatorischen Vielfalt



Aus: Pitschinetz, R.; Grochtmann, M.; Wegener, J.: Test eingebetteter Systeme
http://www.systematic-testing.com/

Klassifikationsbaum Übersicht

- Einfache grafische Methode zur Unterstützung der Testfall-Erzeugung
- Werkzeugunterstützung: **C**lassification **T**ree **E**ditor (e**X**tended **L**ogics)
- **Begriffe**
  - **Klassifikation**: Variable oder Eingabeparameter, erwartetes Ergebnis
  - **Klasse**: Wertebereich einer Klassifikation. Der Wertebereich kann z.B. eine Äquivalenzklasse sein
  - Die **Klassen müssen disjunkt** sein und den Wertebereich der Klassifikation vollständig wiedergeben

Klassifikationsbaum Methode

1. **Klassifikationen wählen**

   Der gesamten Eingabe- oder Ausgabedatenraum des Prüflings wird in testrelevante Gesichtspunkte untergliedert → Klassifikationen

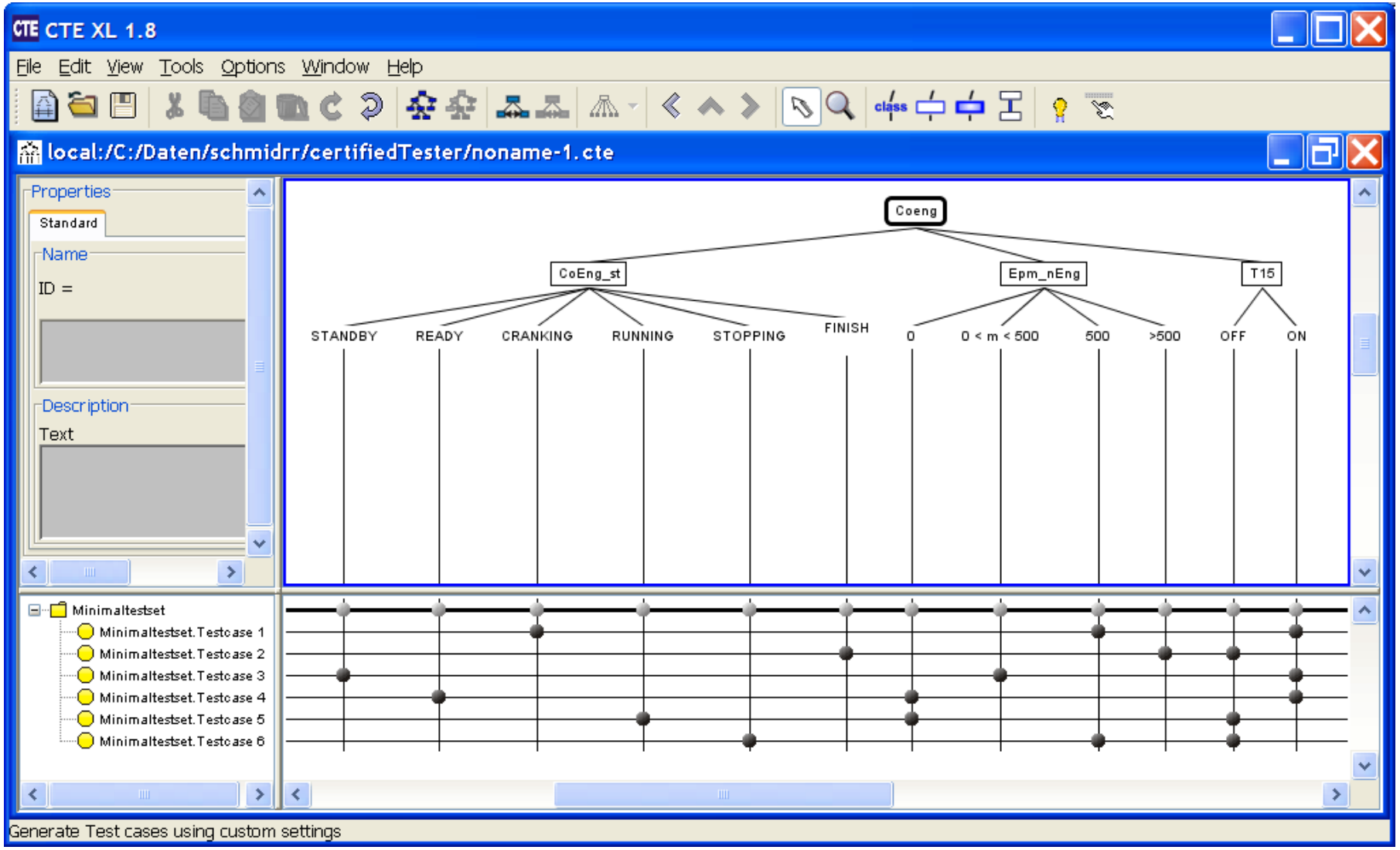
2. **Klassen wählen**

   Jede Klassifikation wird in Wertebereiche unterteilt. Wertebereiche können z.B. Äquivalenzklassen sein. Die Wertebereiche (Klassen) sind disjunkt.

3. **Testfälle wählen**

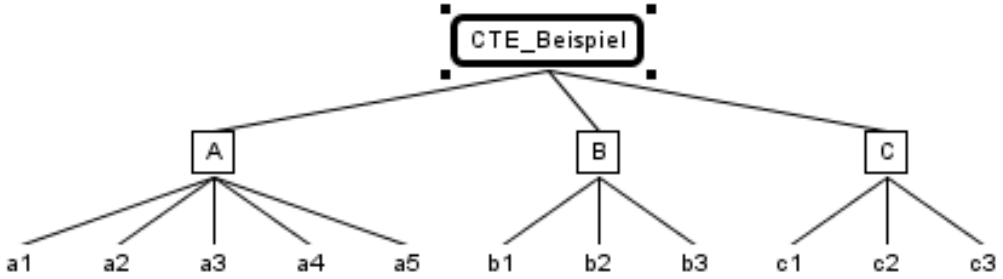
   Testfälle entstehen durch die Kombination von Klassen unterschiedlicher Klassifikationen

Klassifikationsbaum – ein Beispiel



Klassifikationsbaum: Testfälle erzeugen

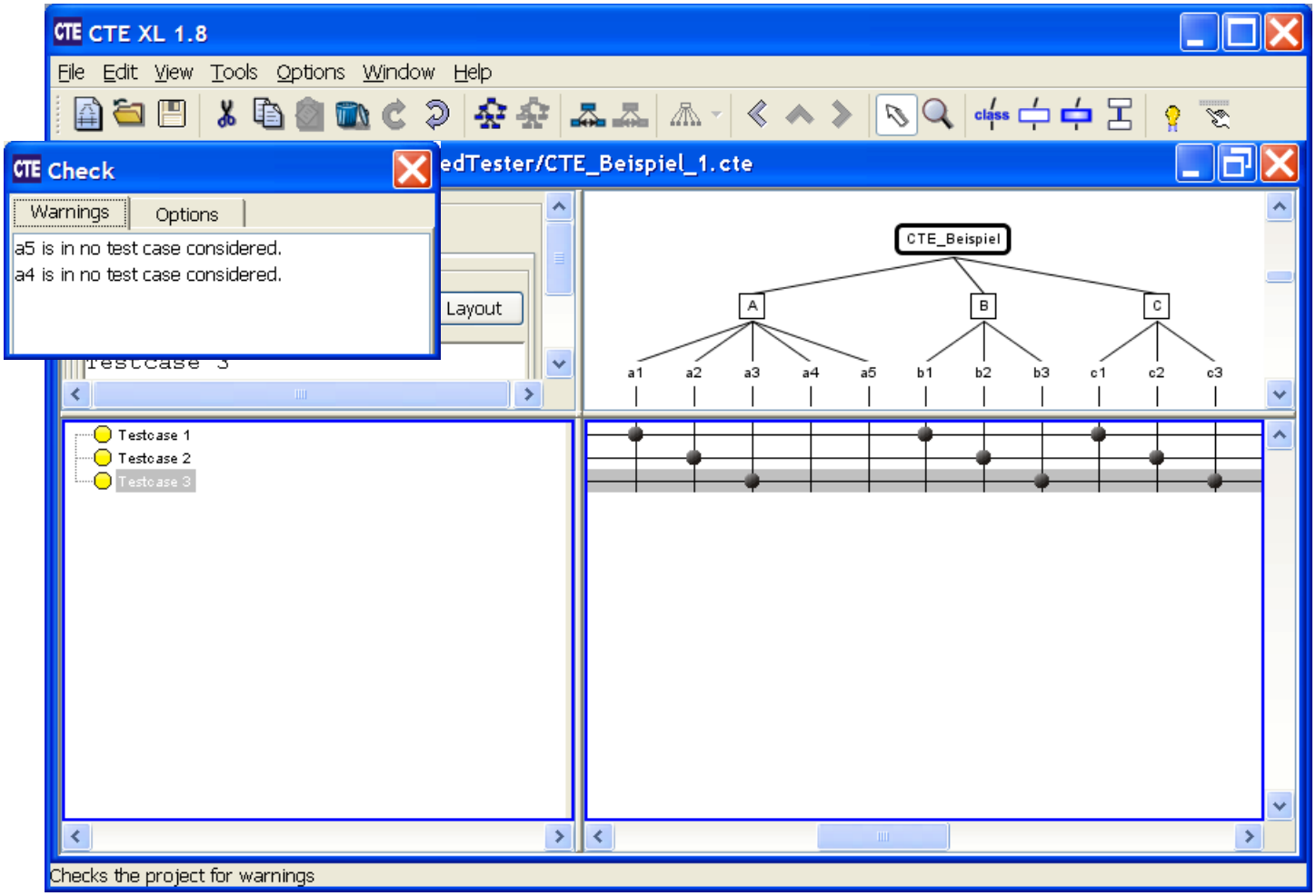
- 3 Klassifikationen (A, B, C) mit folgenden Klassen:



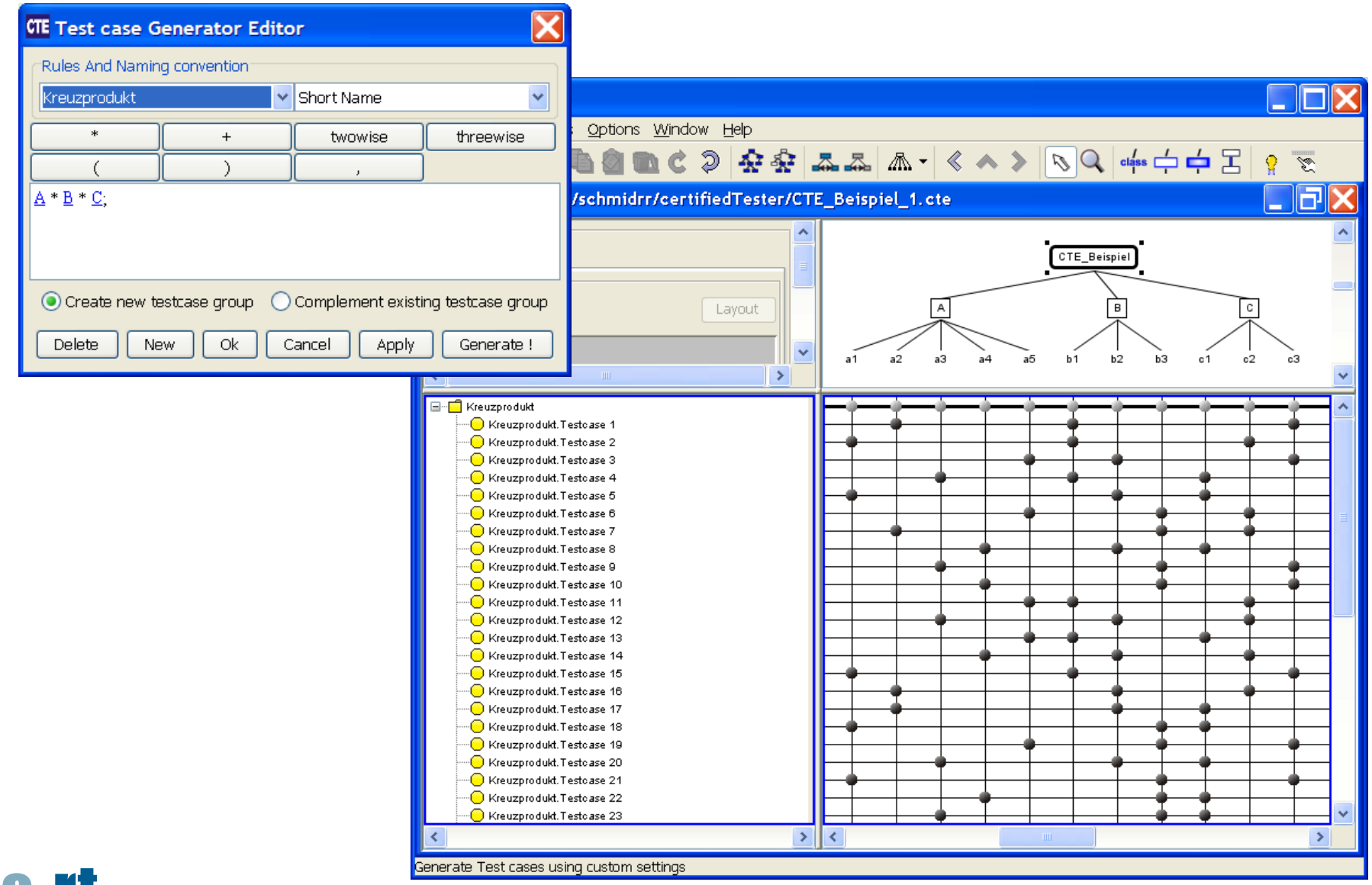
- Um jede Klasse mindestens einmal in einem Testfall zu berücksichtigen sind 5 Testfälle erforderlich
- Vollständige Kombination aller Klassen ergibt 5 x 3 x 3 Testfälle

Manuel Testfälle erzeugen

- Es wird ein Testfall neu angelegt und je Klassifikation eine Klasse selektiert

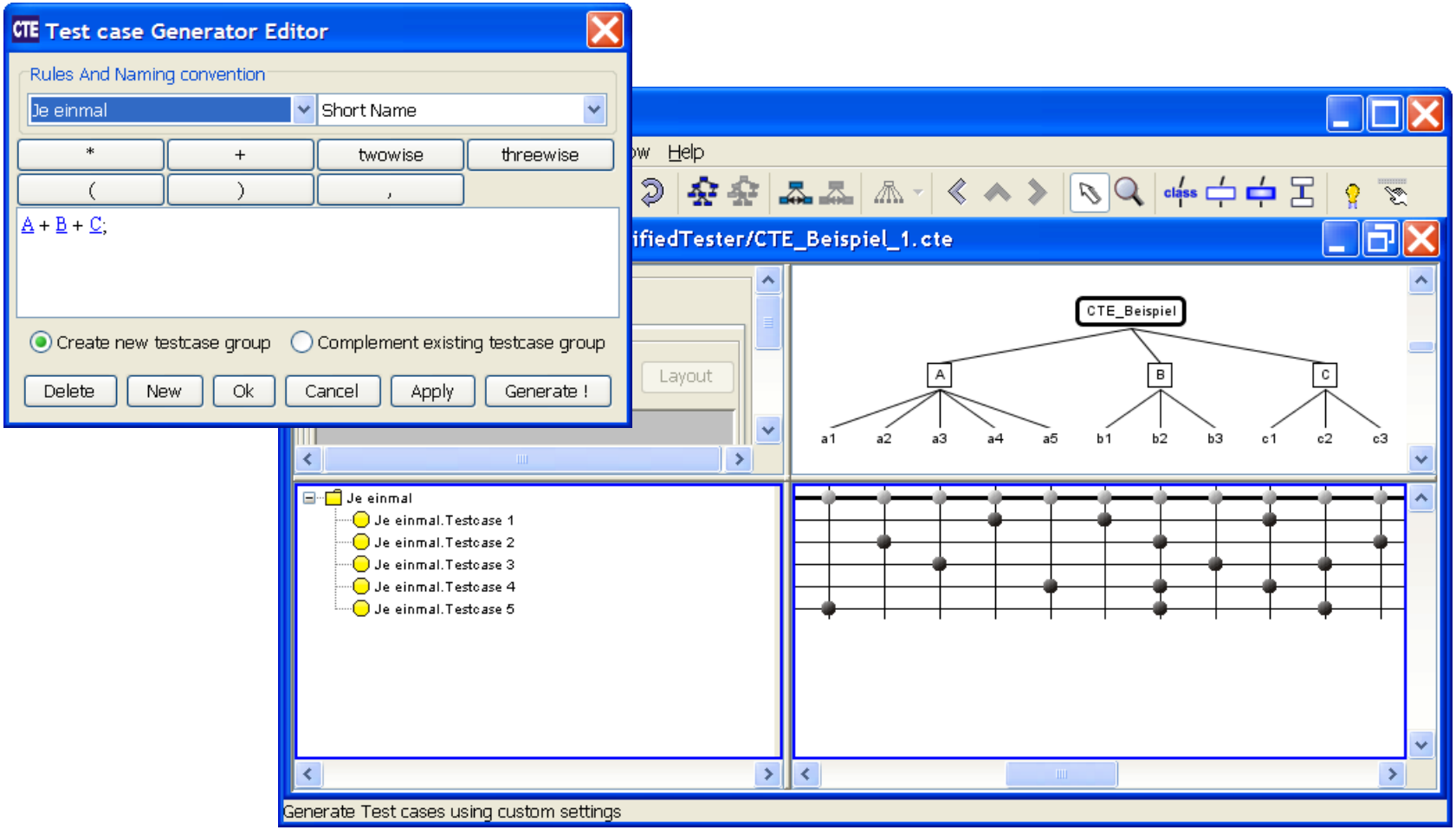


Testfälle generieren: Vollständig



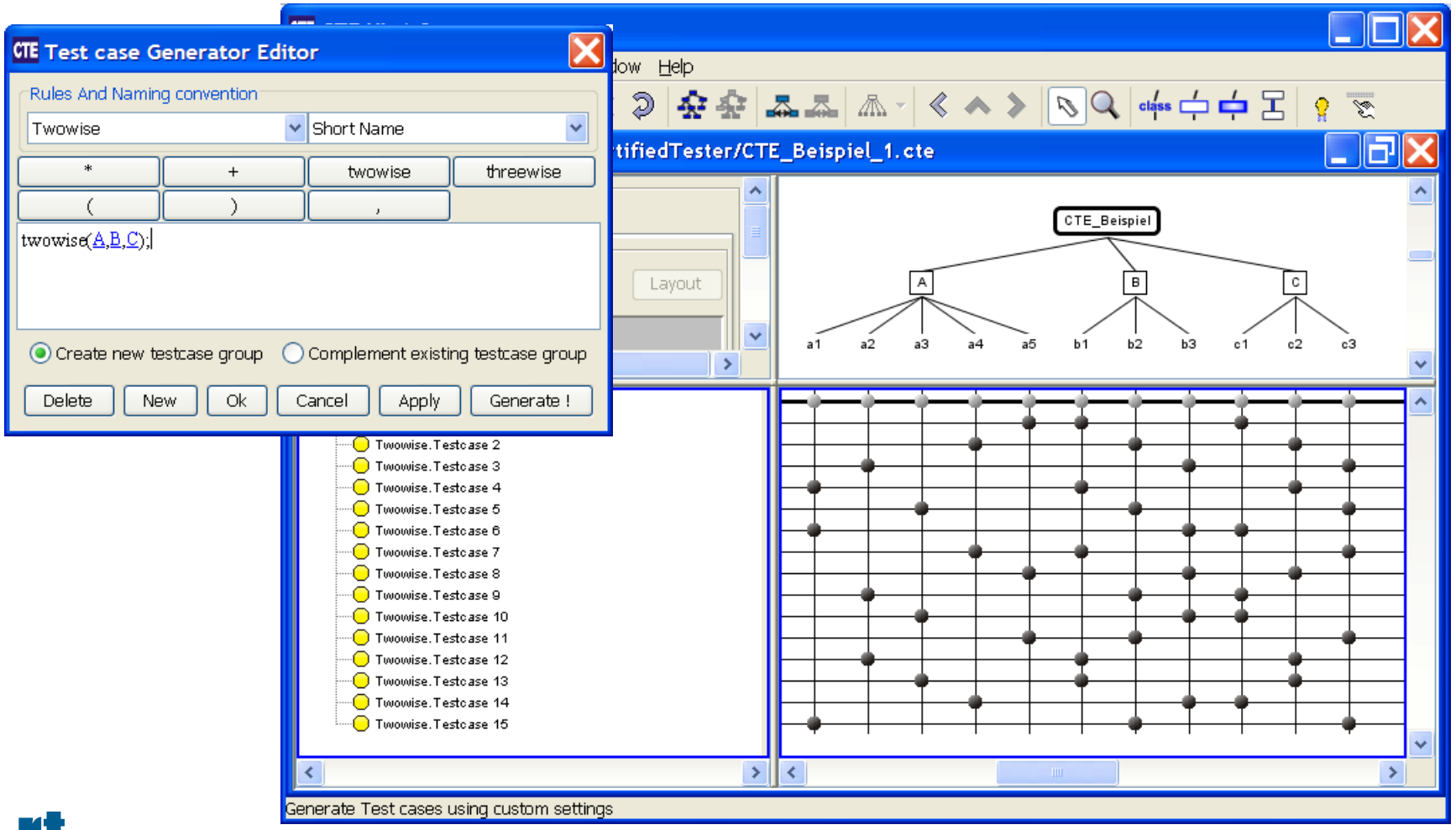
Testfälle generieren: Einfach markieren

- Jede Klasse wird einmal markiert. Die Auswahl geschieht zufällig.



Testfälle generieren: Twowise

- Kombiniert jeweils zwei der angegebenen Klassifikation vollständig miteinander.

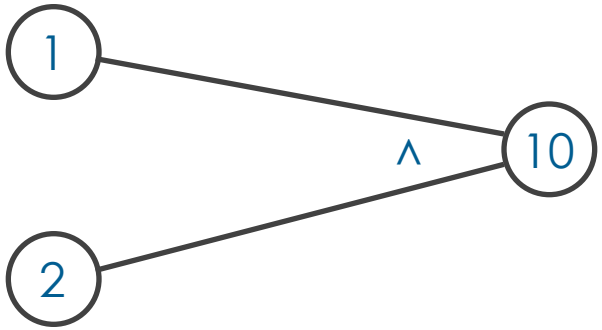


Ursache-Wirkungs-Graph -1

- Grafische Darstellung von Wirkungen zwischen Eingabedaten und Ausgabedaten bzw. Aktionen

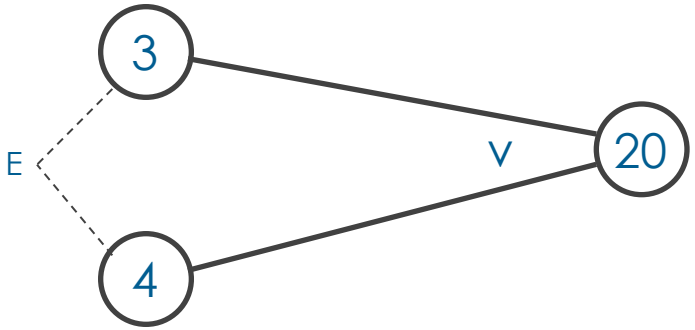
1: Parken im Parkverbot
2: Polizist kommt vorbei

10: Strafzettel erhalten



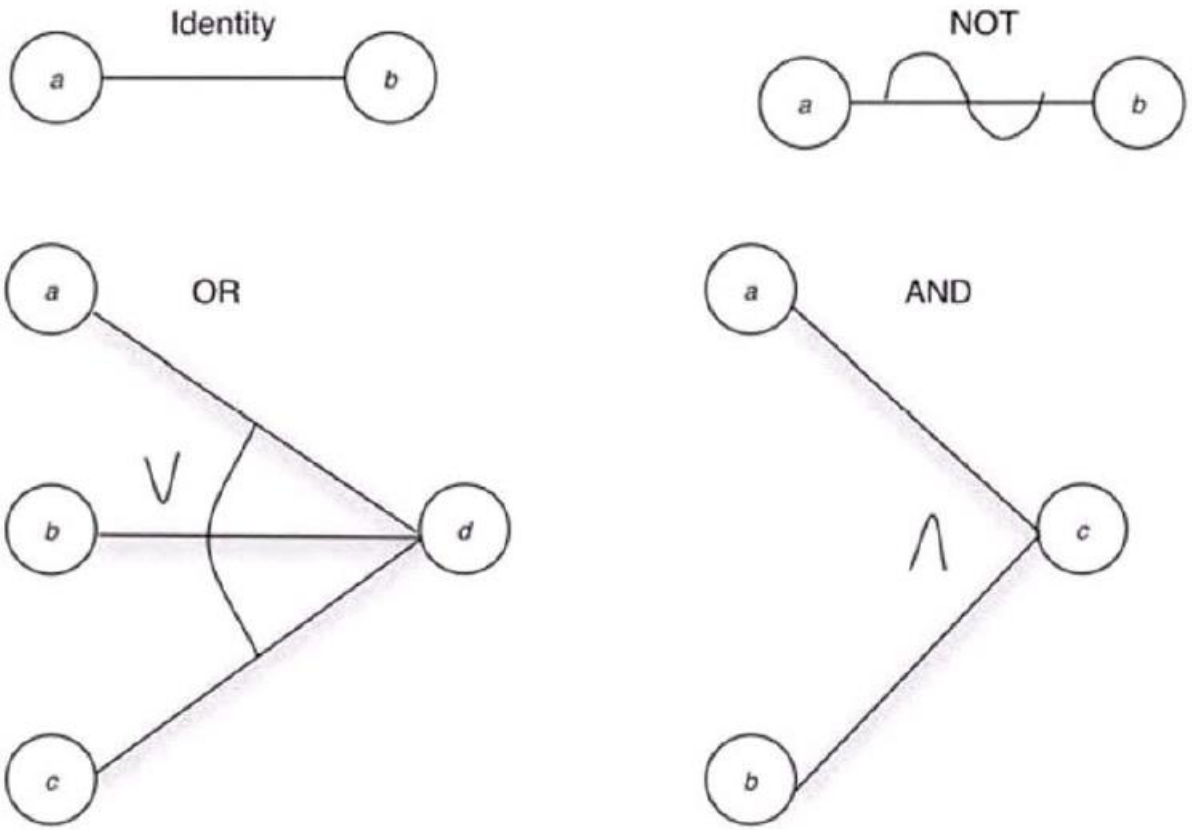
3: Rückwärtsgang eingelegt
4: Langsam vorwärts fahren

20: Abstandswarner aktivieren



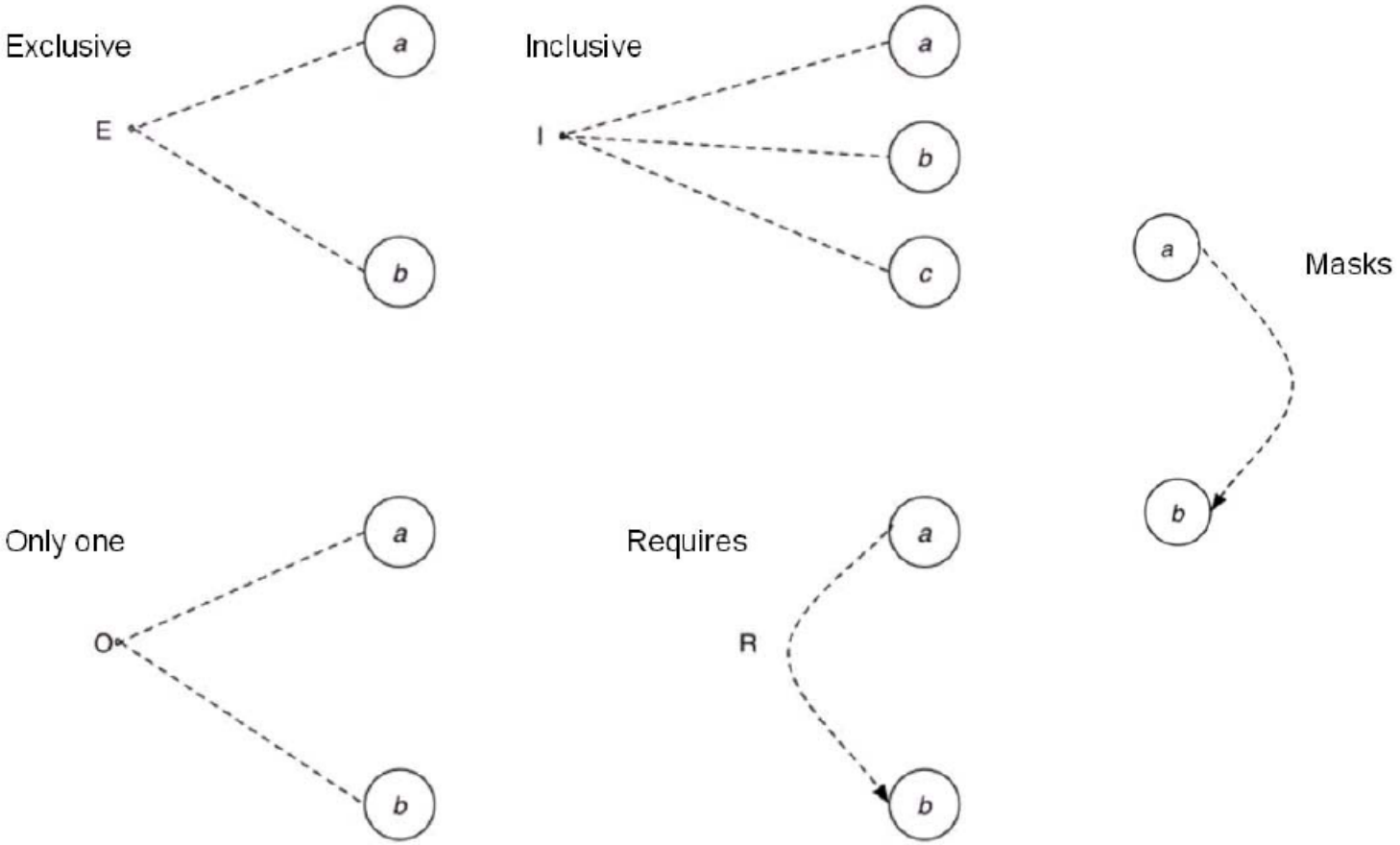
Ursache-Wirkungs-Graph - 2

- Logische Symbole



Ursache-Wirkungs-Graph - 3

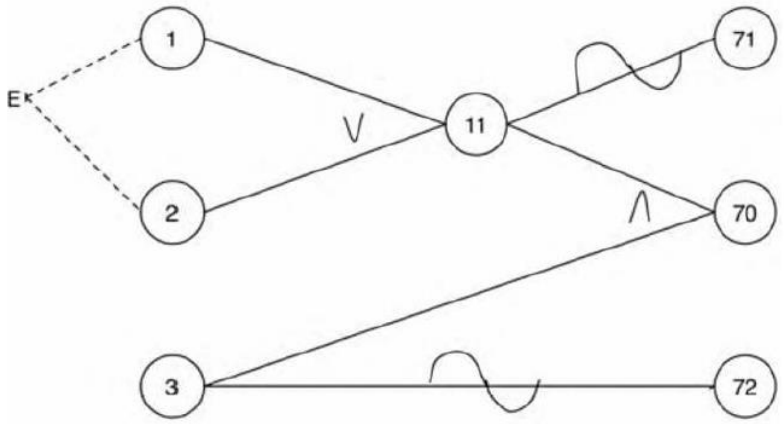
- Einschränkungen



© 2008-2015 Dr. Rainer Schmidberger

Ursache-Wirkungs-Graph - 4

- Übertragung in eine Entscheidungstabelle



Ursache / Wirkungen	T1	T2	...
1	1	0	...
2	0	0	...
3	1	1	...
70	1	0	...
71	0	1	...
72	0	0	...

Entscheidungstabellentest – 1

	Regeln				
Bedingungen	Regel 1	Regel 2	Regel 3	…	*Regel k*
Bedingung 1	T	F	T	Beding-ungen ver-wenden Eingabe-daten	F
Bedingung 2	-	T	F		F
Bedingung i	F	-	-		-
Aktionen					
Aktion 1		x	x	Aktionen erzeugen Ausgabe-daten	x
Aktion 2	x		x		
Aktion j	x				

Don't care

```
IF (Bedingung 1) AND (Not Bedingung i)
  Aktion 2
  Aktion j
END-IF
```

Laurence I. Press, „Conversation of Decision Tables to Computer programs", Comm. ACM 8, No. 6, 385-390 (Juni 1965)

Entscheidungstabellentest – 2

- Entscheidungstabellen beschreiben sehr anschaulich (Geschäfts-) Regeln der Art „wenn ... dann ... sonst“.
- Typische Anwendungssituation: Abhängig von mehreren logischen Eingangswerten sollen verschiedene Aktionen ausgeführt werden
- Entscheidungstabellen unterstützen durch ihren Aufbau die Vollständigkeit des Tests
- Jeder Tabellenspalte (Regel) entspricht ein Testfall
- Regelüberdeckung: Je Regel wird mindestens ein Testfall gewählt.
- Das Soll-Resultat ergibt sich durch die entsprechend ausgeführten Aktionen

Beispiel: Kaffeeautomat

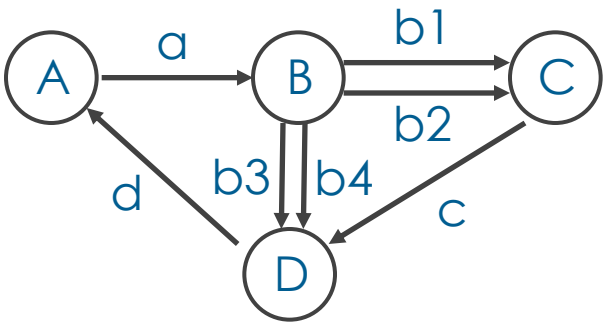
	Regeln				
Bedingungen	Regel 1 Kaffee bestellen	Regel 2 Cappu-ccino bestellen	Regel 3 Tee bestellen	Regel 4 Zutaten fehlen	Regel 5 Bestellung abgebrochen
Ausgewähltes Getränk	Kaffee	Cappu-ccino	Tee	-	-
Zutaten und Becher vorhanden	Ja	Ja	Ja	Nein	-
Bezahlung ist erfolgt	Ja	Ja	Ja	Nein	-
Bestellung ist abgebrochen	Nein	Nein	Nein	Nein	Ja
Aktionen					
Betrag einziehen	x	x			
Milch aufschäumen		x			
Wasser erhitzen	x	x	x		
Kaffee aufbrühen	x	X			
Vorgang abbrechen				x	x

Beispiel: Kaffeeautomat Testfälle

Bedingungen	Testfall 1 Kaffee bestellen	Testfall 2 Cappu-ccino bestellen	Testfall 3 Tee bestellen	Testfall 4 Zutaten fehlen	Testfall 5 Bestellung abgebrochen
Ausgewähltes Getränk	Kaffee	Cappu-ccino	Tee	-	-
Zutaten und Becher vorhanden	Ja	Ja	Ja	Nein	-
Bezahlung ist erfolgt	Ja	Ja	Ja	Nein	-
Bestellung ist abgebrochen	Nein	Nein	Nein	Nein	Ja
Aktionen / Soll-Resultate					
Betrag einziehen	x	x			
Milch aufschäumen	x	x			
Wasser erhitzen	x	x	x		
Kaffee aufbrühen	x				
Vorgang abbrechen				x	x

Zustandsbezogener Test

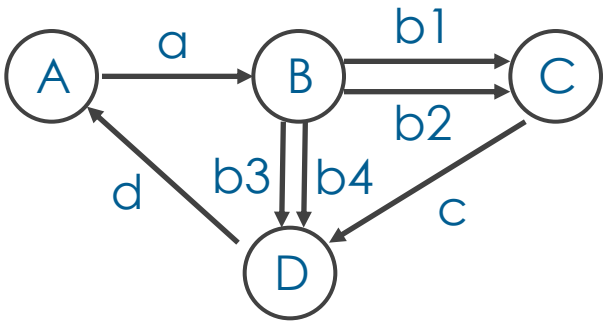
- Ausgangsbasis bildet die Spezifikation des Programms als Zustandsgraph (Endl. Automat, state chart)
- Zustände und Zustandsübergänge sind so beschrieben
- Beispiel:



- Ein Testfall wird aus
  - ⇨ dem Ausgangszustand,
  - ⇨ dem Ereignis (→Ein-gabedaten) und dem
  - ⇨ Soll-Folge-Zustand (→Soll-Resultat)

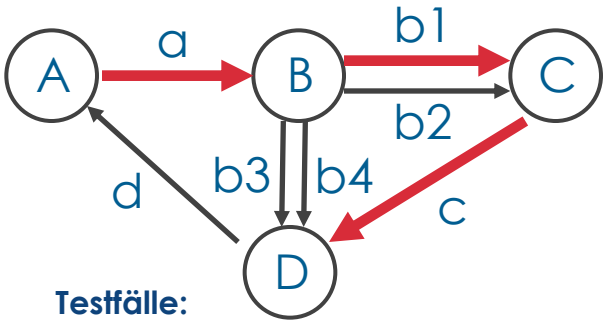
  gebildet

- Testumfang für einen betrachteten Zustand
  - ⇨ Alle Ereignisse, die zu einem Zustandswechsel führen
  - ⇨ Alle Ereignisse, die auftreten können, aber ignoriert werden
  - ⇨ Alle Ereignisse, die auftreten können und eine Fehlerbehandlung erfordern



Zustands-überdeckung

Jeder Zustand muss mindestens einmal erreicht werden

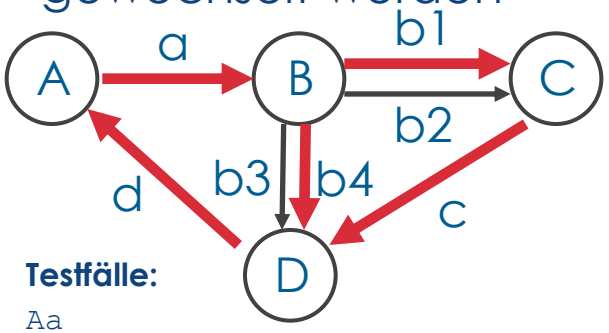


Testfälle:

```
Aa
Bb1
Cc
```

Zustandspaar-überdeckung

Von jedem Zustand muss in jeden möglichen Folgezustand gewechselt werden

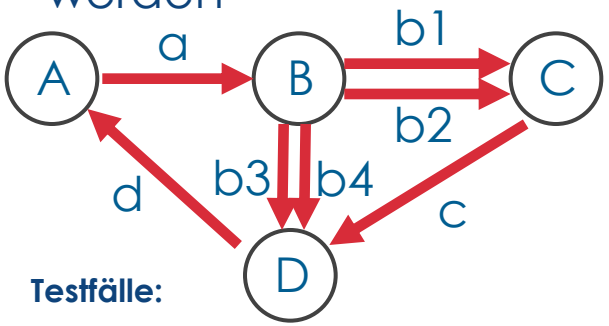


Testfälle:

```
Aa
Bb1
Bb4
Cc
Dd
```

Transitions-überdeckung

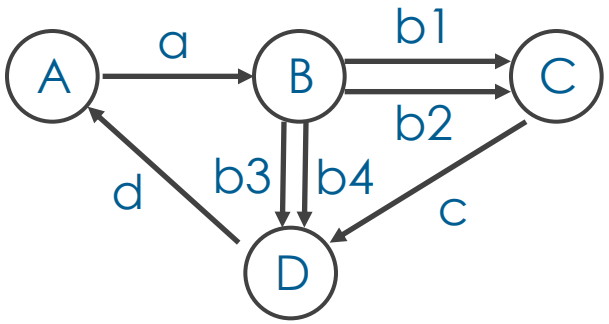
Alle Zustandsübergänge müssen mindestens einmal wirksam werden



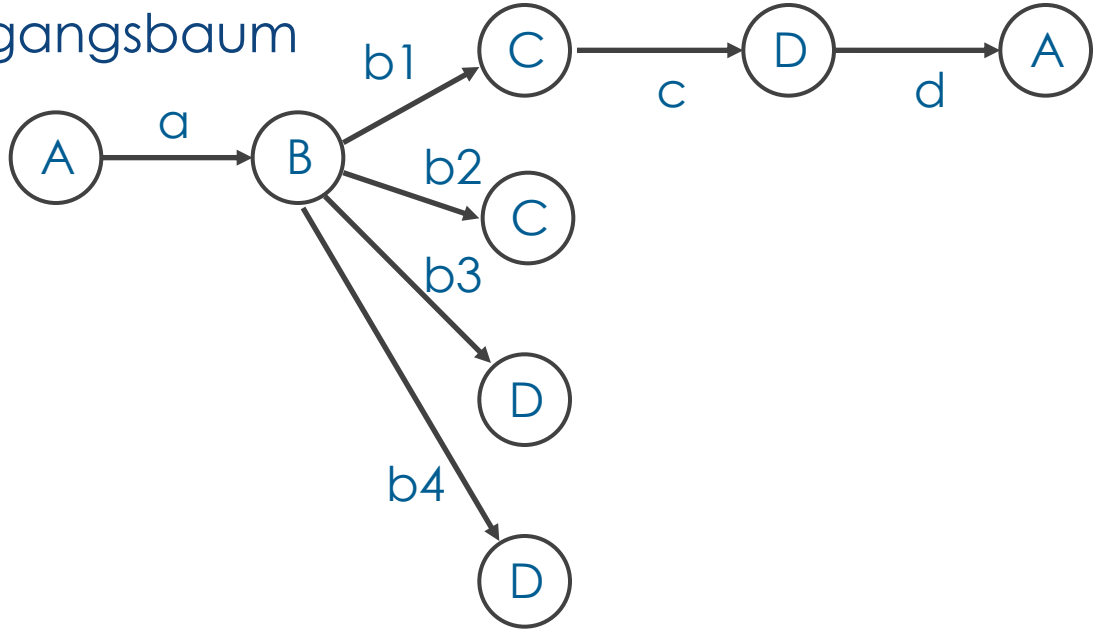
Testfälle:

```
Aa
Bb1     Bb4
Bb2     Cc
Bb3     Dd
```

Roundtrip-Folgen



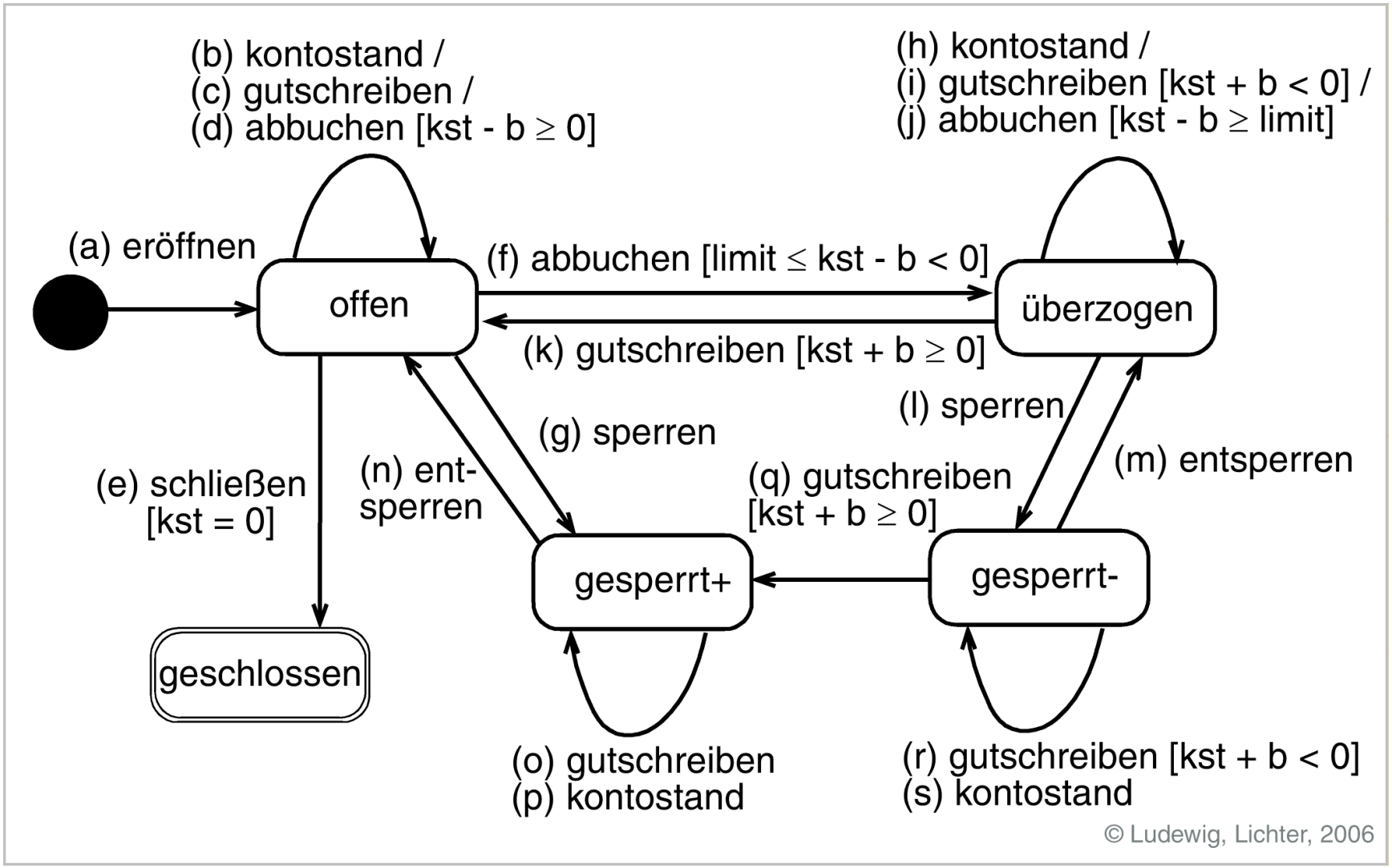
- Roundtrip-Folgen
  - Beginnend im Startzustand
  - Endend in einem Endezustand oder in einem Zustand, der bereits in dieser oder einer anderen Roundtrip-Folge enthalten war
  - Zustandsübergangsbaum



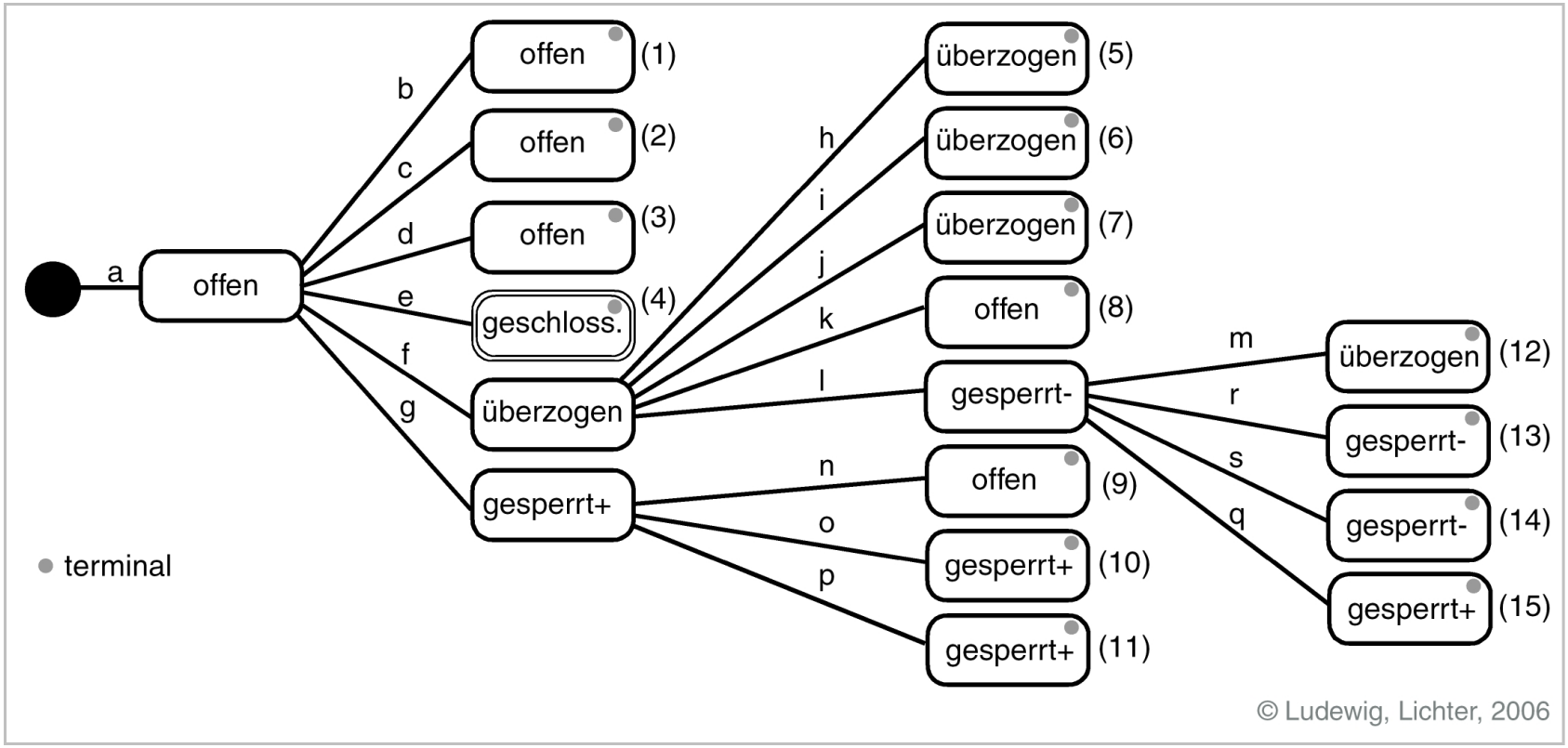
Testfälle:

```
Aa Bb1 Cc Dd
Aa Bb2
Aa Bb3
Aa Bb4
```

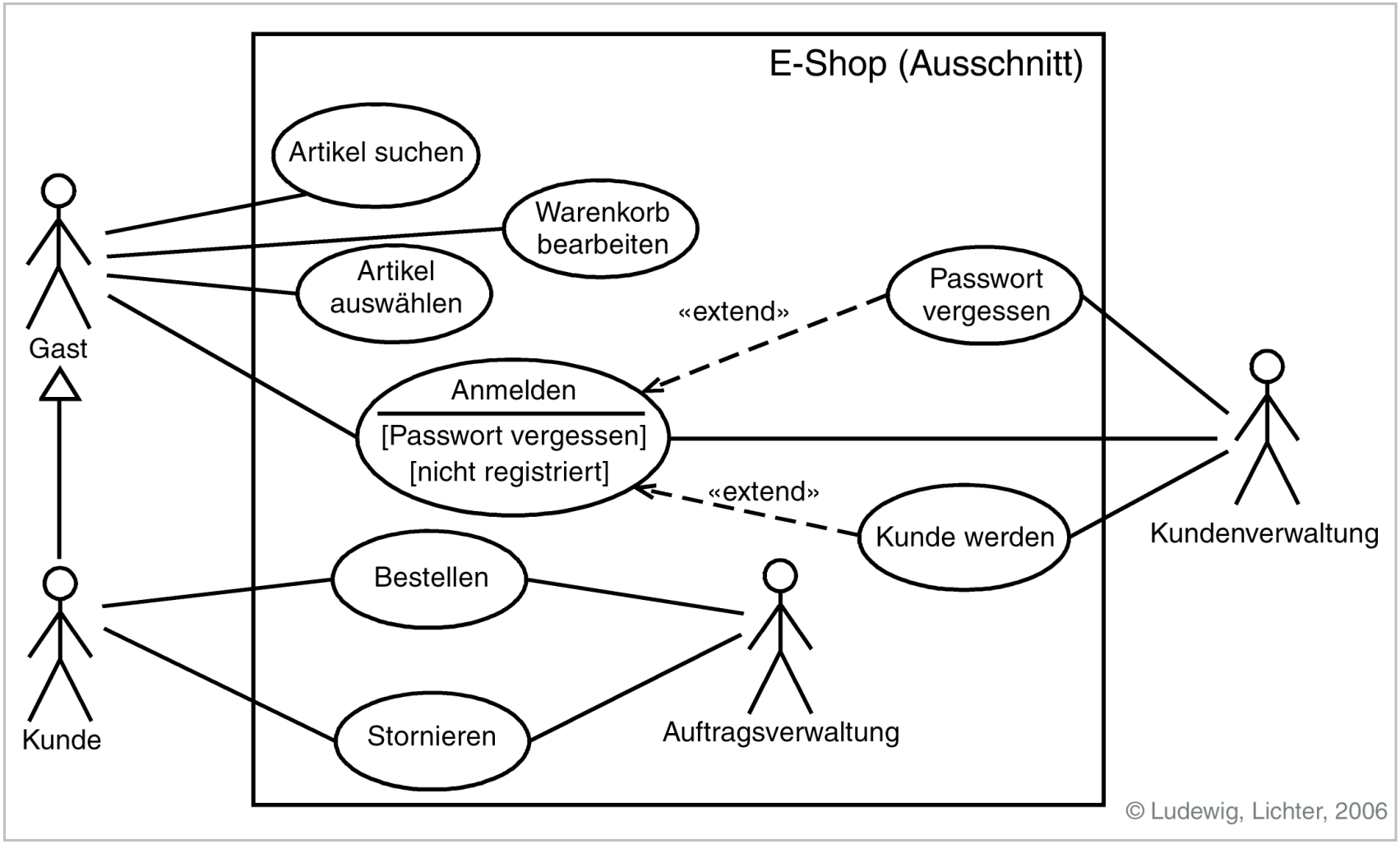
Beispiel - Zustandsautomat



Beispiel: Zustandsübergangsbaum



Anwendungsfallbasierter Test



Anwendungsfallbasierter Test – 2

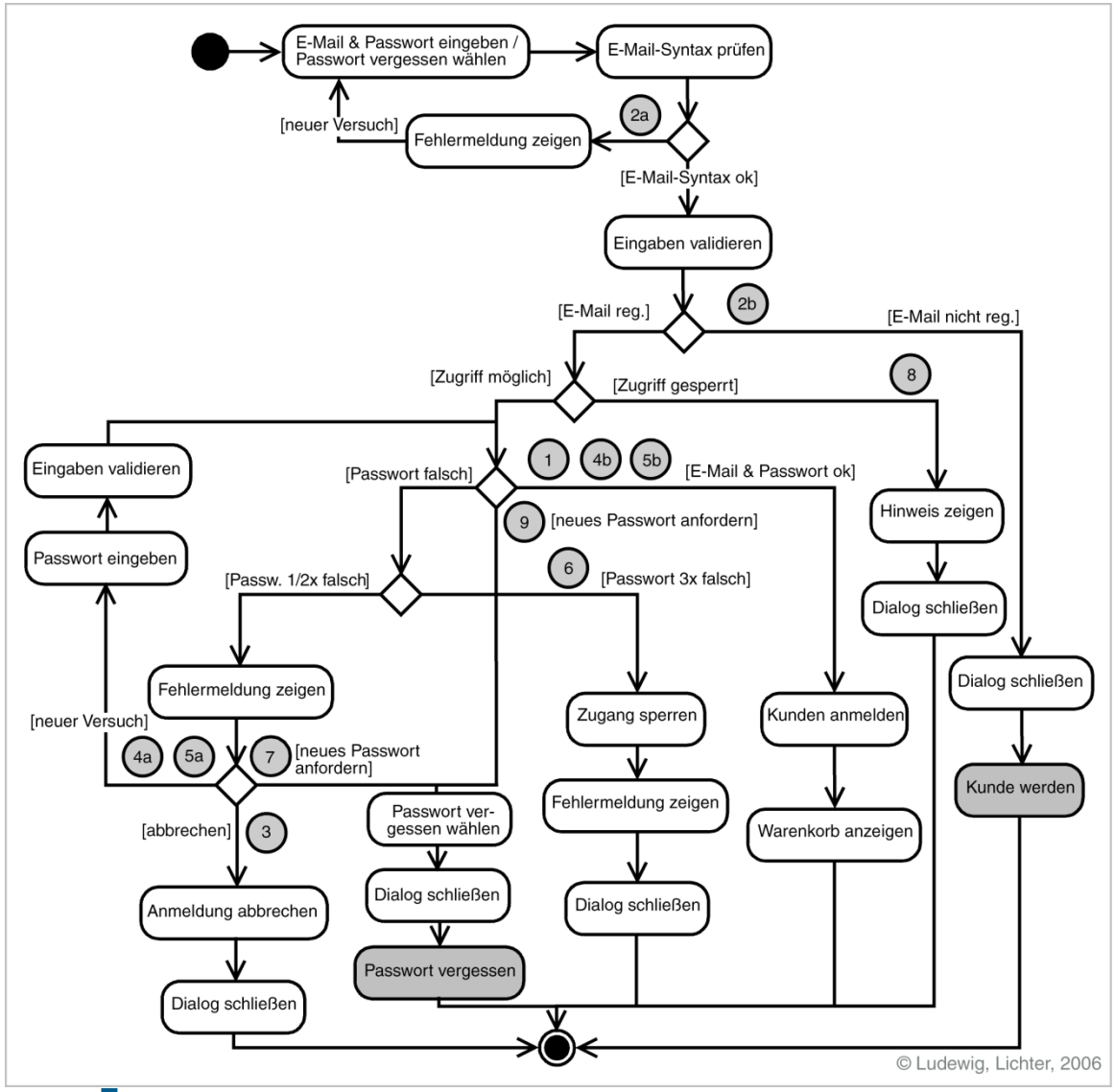
Vorteil

- Gutes Strukturierungsinstrument
- Anwendungsfälle beschreiben Vor- und Nachbedingungen sowie die Benutzeraktionen.
- Sonderfälle werden behandelt.

Nachteil

- Die Anwendungsfälle enthalten wenig Details und sind oft unvollständig.

Test auf Basis des Aktivitätendiagramms

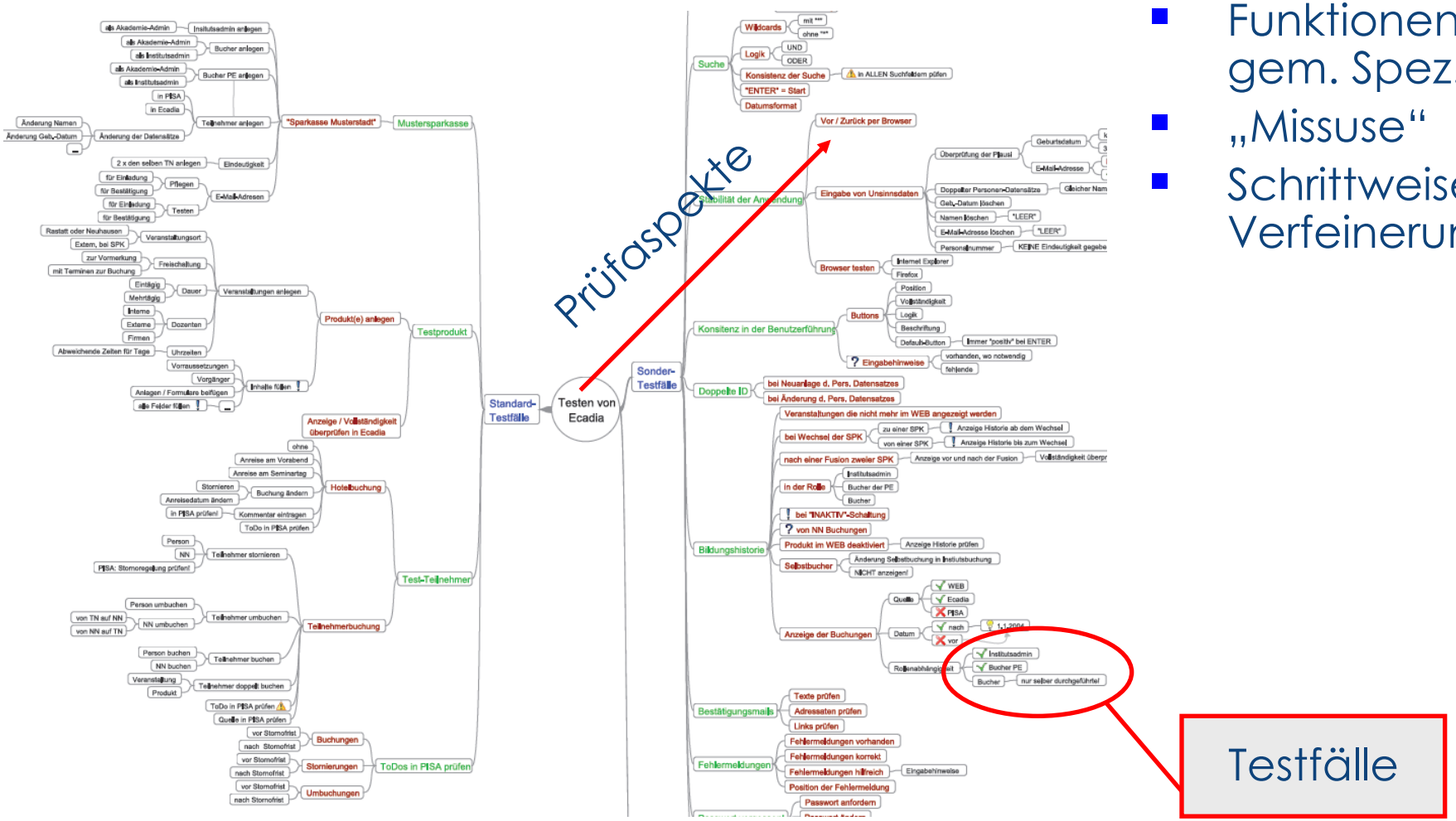


- Aktivitäten- und Kanten-überdeckung als Vollständigkeits-kriterium

Zufallstests

- Aus dem Wertebereich der Eingabedaten werden zufällig Testfälle erzeugt
- Voraussetzungen
  - Datentypen und Datenstrukturen der Eingabedaten sowie die Wertebereiche müssen bekannt sein
  - Sollresultate für die (zufällig gewählten) Eingabedaten müssen automatisch berechnet werden können (Testorakel)
- Ablauf
  - Testfälle mit Sollresultaten werden vorab generiert, und zu einem späteren Zeitpunkt ausgeführt
  - Eingabedaten werden während der Testausführung zufällig generiert und das Sollresultat wird „on-the-fly“ ermittelt

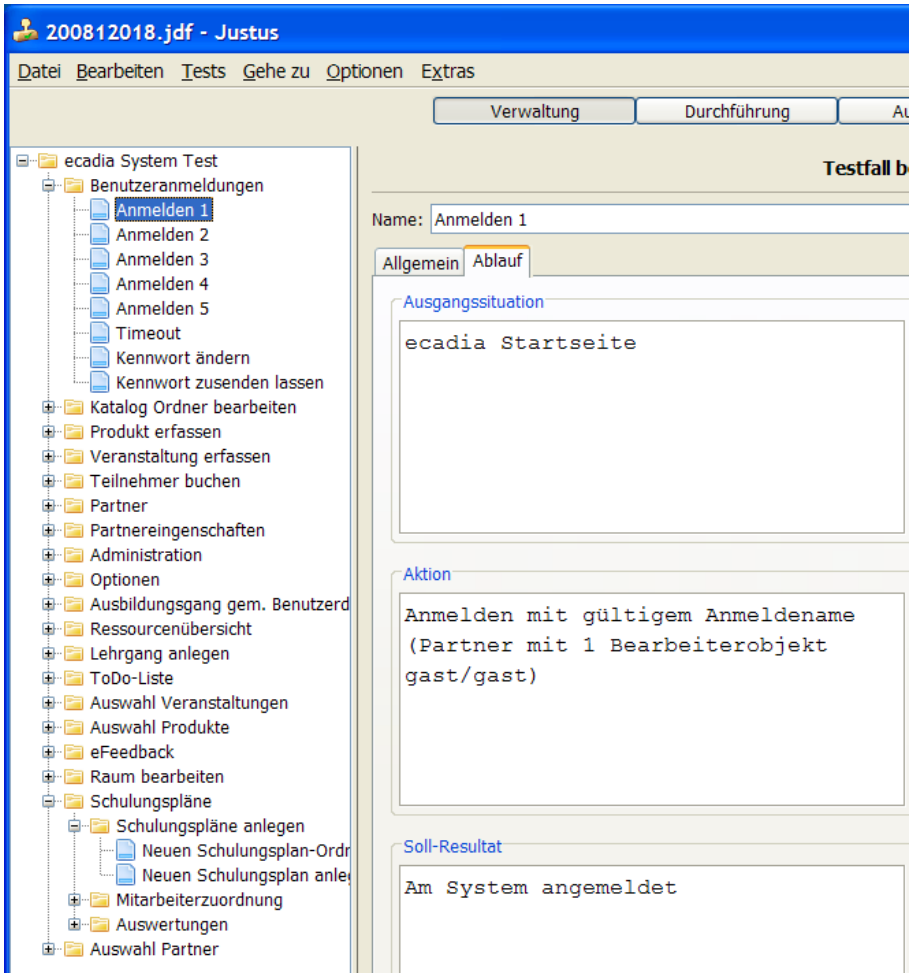
Testfallentwurf: Methodik



- Funktionen gem. Spez.
- „Missuse“
- Schrittweise Verfeinerung

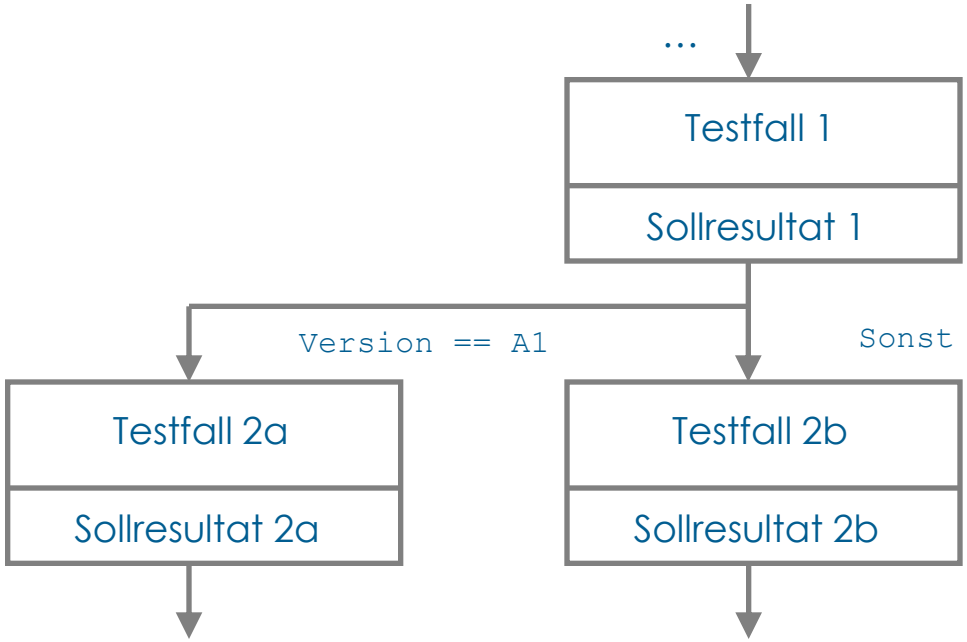
Darstellung von Testfällen - Text

- Text- bzw. tabellenbasierend
  - Testfälle werden in Testsequenzen gruppiert
  - Die Abfolge der Testfälle ergibt die Reihenfolge
  - Vorteile: Einfach, „Excel“ als Werkzeug möglich
  - Nachteil: wenig anschaulich, Prüfung auf Vollständigkeit ist schwierig



Darstellung von Testfällen – Grafisch

- Ein Knoten beschreibt einen Testfall
- Kanten können mit Bedingungen (ähnl. Zustandsdiagramm) versehen werden
- Kanten beschreiben die Abfolge der Testfälle
- Werkzeug – oft mit eigenen Anpassungen – erforderlich
- Alternative Ausführungspfade bei Varianten sind darstellbar



Modellbasierter Test – MBT

- Die (verbale) Spezifikation wird in ein Modell (z. B. Zustandsautomaten) übertragen.
- Aus den Modellen werden Testfälle generiert.
- Die Testfälle werden (automatisiert) ausgeführt.

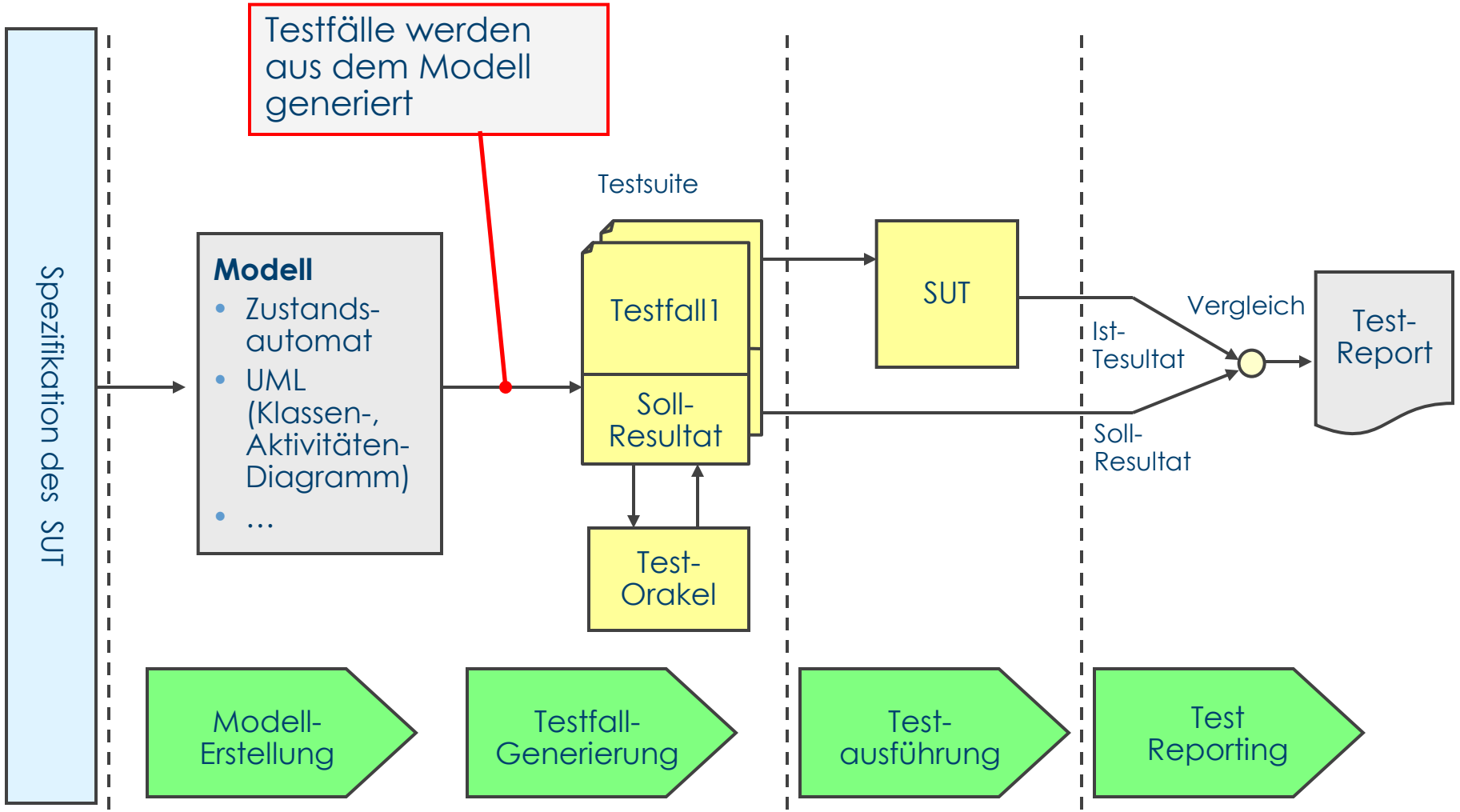
Vorteile des MBT

- Qualitätssicherung der Spezifikation durch die Modellierung; Aufdecken von Spezifikationsmängeln
- Anschauliche Modelle statt (sehr) vieler Testfälle
- Automatische Generierung der Testfälle aus den Modellen

Nachteile des MBT

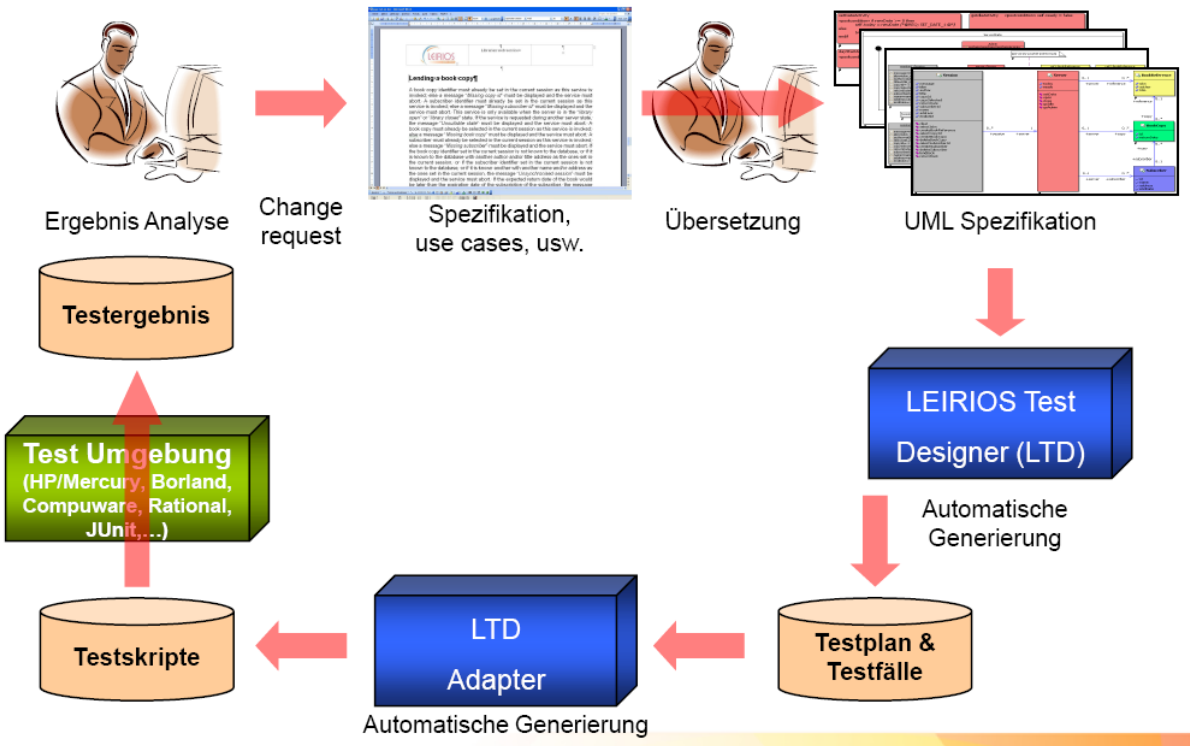
- Die Modelle sind oft nicht mächtig genug; nur ein Teil der Anforderungen lassen sich im MBT behandeln.
- Die Modelle der modell-basierten Entwicklung können nicht verwendet werden.

Datenfluss beim modellbasierten Testen



Beispiel: Leirios

Das neue Verfahren: LTD im Einsatz



Quelle: www.smartesting.com

© 2008-2015 Dr. Rainer Schmidberger

Zusammenfassung MBT

- Der MBT bildet einen interessanten Ansatz: Modelle sind in vielen Fällen anschaulicher d. h. lesbarer/prüfbarer als Text oder Code.
- Aus den Modellen werden (viele) Testfälle generiert; eine automatische Ausführung ist erforderlich.
- Die Modelle sind – zumindest in den Werkzeugen am Markt – in der Regel aus der UML (Klassendiagramm, Zustandsdiagramm, Interaktionsdiagramm, …)
  - Zur weiteren Spezifizierung wird oft noch die OCL (Object Constraint Language) hinzugenommen
  - UML deckt vieles, aber eben nicht alle Testanforderungen ab: Nicht alle Systemmerkmale können mit dem UML-MBT getestet werden
- Interessante Ansätze zeigen domänenspezifische Modelle; oft werden Werkzeuge dazu selbst entwickelt oder stark adaptiert.
- Vorsicht beim Einsatz der Modelle der modellbasierten Entwicklung!

Kapitel 3
Testfallentwurfsverfahren

- ✓ Spezifikationsbasierter Test
- **Glass-Box-Test (GBT)**
- Erfahrungsbasierte Verfahren

© 2008-2015 Dr. Rainer Schmidberger

Glass-Box-Test

- Alternative Bezeichnungen: Coverage-Test, Whitebox-Test, Strukturtest.
- Der Glass-Box-Test (GBT) beurteilt die Vollständigkeit eines Tests anhand der vollständigen Ausführung des Programmcodes
- Verschiedene „Typen“ an Programmcode werden betrachtet: Anweisungen, Verzweigungen, Schleifen oder Bedingungsterme.
- Als Resultat des Glass-Box-Tests entsteht das Glass-Box-Test-Protokoll. Daraus kann u. a. der Grad an Ausführung des Programms angeben werden → Überdeckung, Coverage.
- Eine Übersicht zu GBT-Werkzeugen steht unter

`http://www.iste.uni-stuttgart.de/se/menschen/schmidberger/glassboxtools.html`

Structural testing.

Testing that takes into account the internal mechanism of a system or component.

Types include branch testing, path testing, statement testing.

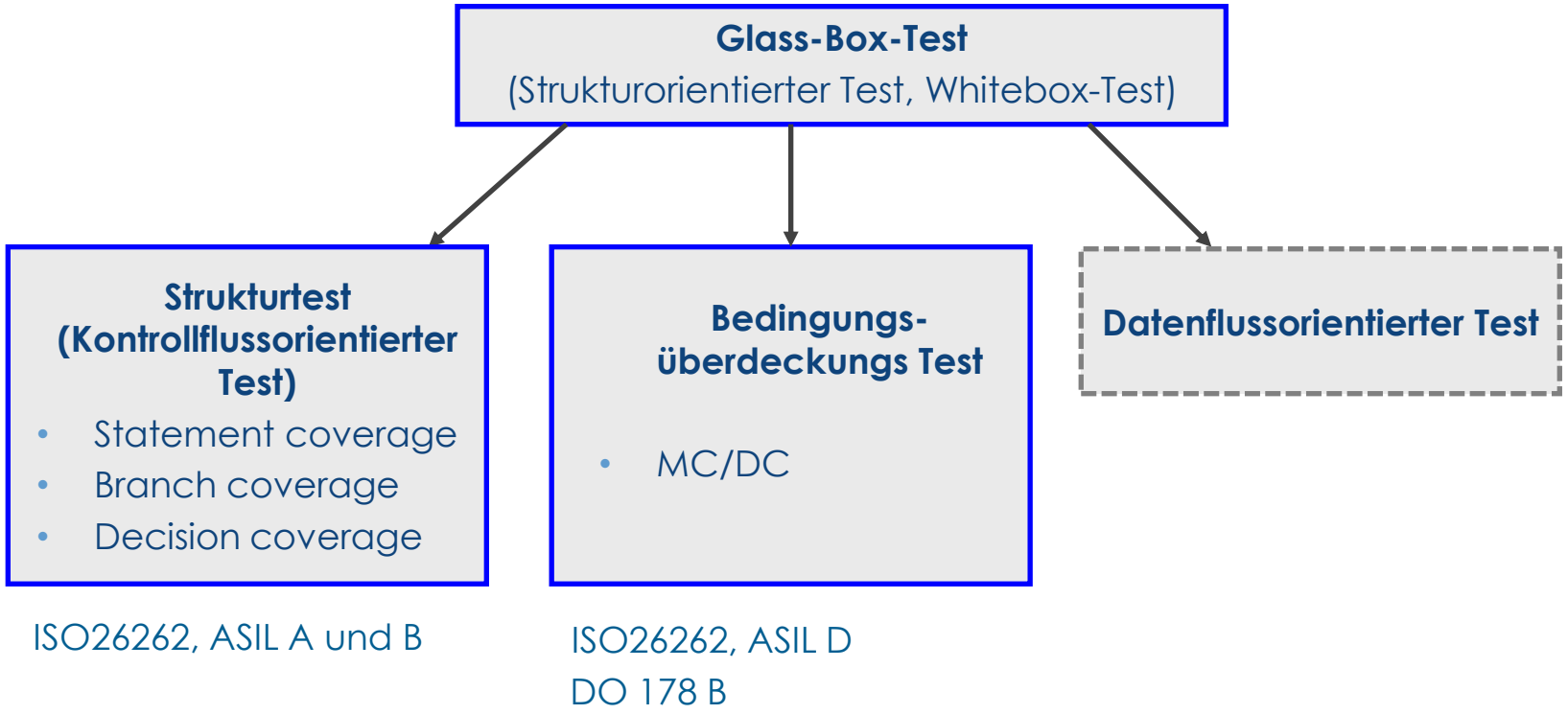
Synonym: glassbox testing; white-box testing.

Contrast with: functional testing (1).

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

Glass-Box-Test Übersicht

- In der Praxis wird der Glass-Box-Test in der Regel in drei Gruppen aufgeteilt:



Nutzen des Glass-Box-Tests

1. **Codeüberdeckungsmaße als Metrik zur Testgüte**

   Objektives Vollständigkeitskriterium, das beispielsweise als Testendekriterium verwendet werden kann

2. **Testsuite-Erweiterung**

   Der GBT zeigt Programmcode, der für eine Testsuite nicht ausgeführt wird und damit ungetestet bleibt.

3. **Grundlage für selektiven Regressionstest**

   Anstelle der „rerun-all“-Strategie sollen nur einzelne, ausgewählte Testfälle ausgeführt werden.

4. **Testsuite-Reduktion**

   Für den Regressionstest soll die Testsuite verkleinert werden, ohne dabei aber (wesentlich) an Testgüte einzubüßen

© 2008-2015 Dr. Rainer Schmidberger

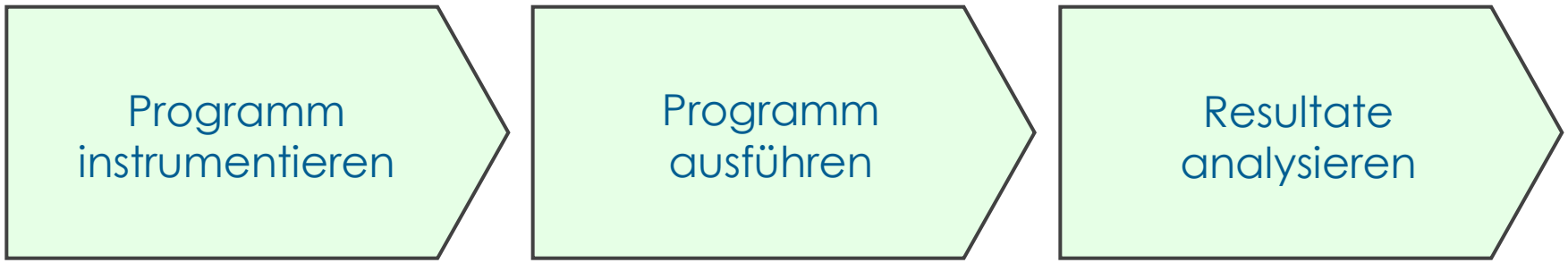
Zertifizierungsrichtlinien

- IEC61508

  Für alle Sicherheitsstufen wird für den Modultest ein Glass-Box-Test empfohlen (für SIL 3 und 4 „highly recommended“), (Tabelle B.2 in 61508-3 [IEC61508]). Zudem wird dort auf 61508-7, C.5.8 verwiesen, wo „a large percentage of the program code exercised“ gefordert wird. Es wird empfohlen, das Überdeckungsziel vorab zu spezifizieren. Neben Anweisungs- und Zweigüberdeckung wird auch unter „compound conditions“ die MC/DC-Überdeckung referenziert

Teilschritte beim Glass-Box-Test

- Beim Glass-Box-Test wird das Programm speziell „präpariert“ um die Ausführung protokollieren zu können (→instrumentieren). Hierzu sind Werkzeuge erforderlich.
- Die Ausführung erfolgt genau gleich wie beim Blackbox-Test
- Das Resultat des Glass-Box-Tests ist die Information welcher Programmcode ausgeführt wurde und welcher nicht



Instrumentierung

- **Quelltextinstrumentierung**

  Im Quelltext werden vor der Kompilierung Anweisungen ergänzt (eingewoben), die die Protokollierung der Laufzeitinformation durchführen. Nachteil dieser Instrumentierung ist der aufwändigere Build-Vorgang.
- **Bytecode-Instrumentierung**

  Der bereits kompilierte Programmcode wird instrumentiert.
- **„On-the-fly"-Instrumentierung**

  Die Laufzeitumgebung wird so modifiziert, dass die gewünschten Informationen erhoben und ausgegeben werden. Nur bei Programmiersprachen mit ausgeprägter Laufzeitumgebung (i. d. R. Interpreter) möglich.

- Generelle Probleme sind das „Application blow up" und das „Execution slow down"

Strukturtest (1)

- **Anweisungsüberdeckung**
  - Anteil der ausgeführten Anweisungen (Statements)
  - Statement coverage
  - Auch als c0-Test bezeichnet
- **Zweigüberdeckung**
  - Anteil der ausgeführten Programmzweige
  - Branch coverage
  - Grundlage bildet in der Regel der Kontrollflussgraph (Kantenüberdeckung)
  - Auch als c1-Test bezeichnet

Strukturtest (2)

- **Pfadüberdeckung**
  - Anteil der ausgeführten Programmpfade (Achtung: die Gesamtmenge der Pfade ist in der Regel unendlich groß!)
  - Auch als c2a-Test bezeichnet
- **Entscheidungsüberdeckung** (decision coverage)
  - Vollständigkeit bezieht sich auf Programmverzweigungen einschl. Ausdrücke mit Kurzschlusssemantik, bedingter Ausdrücke und Ausnahmenbehandlung
  - Federal Aviation Administration, 2007, Software Verification Tools Assessment Study, DOT/FAA/AR-06/54, http://www.tc.faa.gov/its/worldpac/techrpt/ar0654.pdf

Strukturtest (3)

- **Schleifenüberdeckung**
  - Anteil der Schleifen, die nicht, einmal oder mehrfach durchlaufen werden
  - Auch als Boundery-Interiour-Test oder als c2b-Test bezeichnet
- **Funktionsüberdeckung**
  - Anteil der ausgeführten Funktionen

- Blocküberdeckung, Instruction-Überdeckung, …

Überdeckung	Ausprägung	Beispiel
Anweisung/ Statement	ausgeführt / nicht ausgeführt	`System.out.println("Hallo CodeCover!");`

Block (Zweig)/ Branch	ausgeführt / nicht ausgeführt	`if(i == 99) {`
		`  System.out.println("Hallo World!");`
		`} else {`
		`  System.out.println("Hallo Stuttgart!");`
		`}`

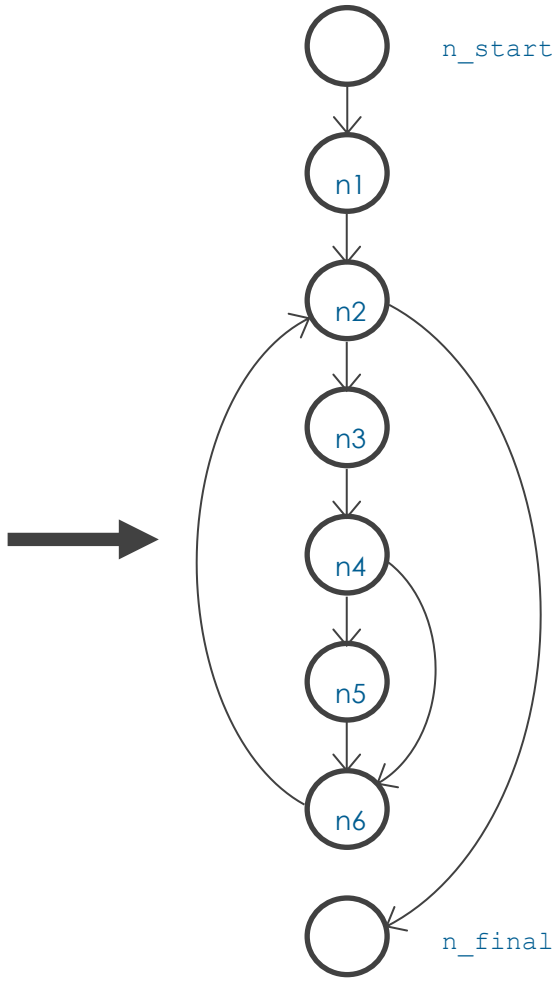
Term	true- und false-Wirksamkeit je Einzelterm	`if(i > 5 || j > 1 || k == j) {`
		`  System.out.println("Termüberdeckung");`
		`}`

?-Operator	Ausdrücke wirksam: keiner / einer von beiden / beide	`k = i < 3 ? 2 : 1;`

Schleife	Schleifenkörper: nicht / einmal / mehrfach ausgeführt	`while(i < 5) {`
		`  System.out.println("Schleifenüberdeckung");`
		`  i++;`
		`}`

Der Kontrollflussgraph (CFG)

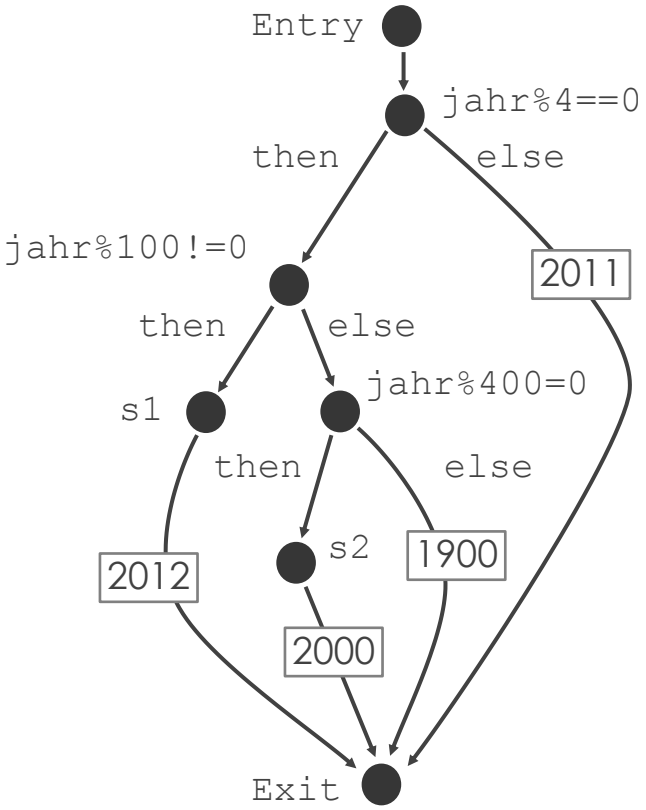
```
void ZaehleZchn(int& VokalAnzahl,
                        int& Gesamtzahl)
{
  char Zchn;

  cin >> Zchn;                              // n1
  while(Zchn >= 'A' && Zchn <= 'Z' &&       // n2
          Gesamtzahl < INT_MAX)
  {
    Gesamtzahl = Gesamtzahl + 1;            // n3

    if(Zchn == 'A' || Zchn == 'E' ||        // n4
       Zchn == 'I' || Zchn == 'O' ||
       Zchn == 'U')
    {
      VokalAnzahl =  VokalAnzahl + 1;       // n5
    }
    cin >> Zchn;                            // n6
  }
}
```



Beispiel Strukturtest

- Wie viele Testfälle sind erforderlich?

```
boolean istEinSchaltjahr(int jahr) {

  // Defaultannahme: kein Schaltjahr
  boolean istSchaltjahr = false;

  if(jahr % 4 == 0) {
    if(jahr % 100 != 0) {
      istSchaltjahr = true; // s1
    } else {
      if(jahr % 400 == 0) {
        istSchaltjahr = true; // s2
      }
    }
  }
  return istSchaltjahr;
}
```



Zur Anweisungsüberdeckung: 3 Testfälle, z.B. 2000, 1900, 2008
Zur Zweigüberdeckung: 4 Testfälle, z.B. 2000, 1900, 2008, 2005

© 2008-2015 Dr. Rainer Schmidberger

Strukturtest (2)

- „Kurzschlusssemantik“

```
// Variante 2
boolean istSchaltjahr(int jahr) {

  return jahr%4 == 0 && jahr%100 != 0 || jahr%400 == 0;

}
```

- Bedingter Operator

```
// Variante 3
boolean istSchaltjahr(int jahr) {

  return jahr%4 == 0 ? (jahr%100 != 0 ? true :
                (jahr%400 == 0 ? true : false)) : false;

}
```

→ Wie sieht es hier mit der Zweigüberdeckung aus?

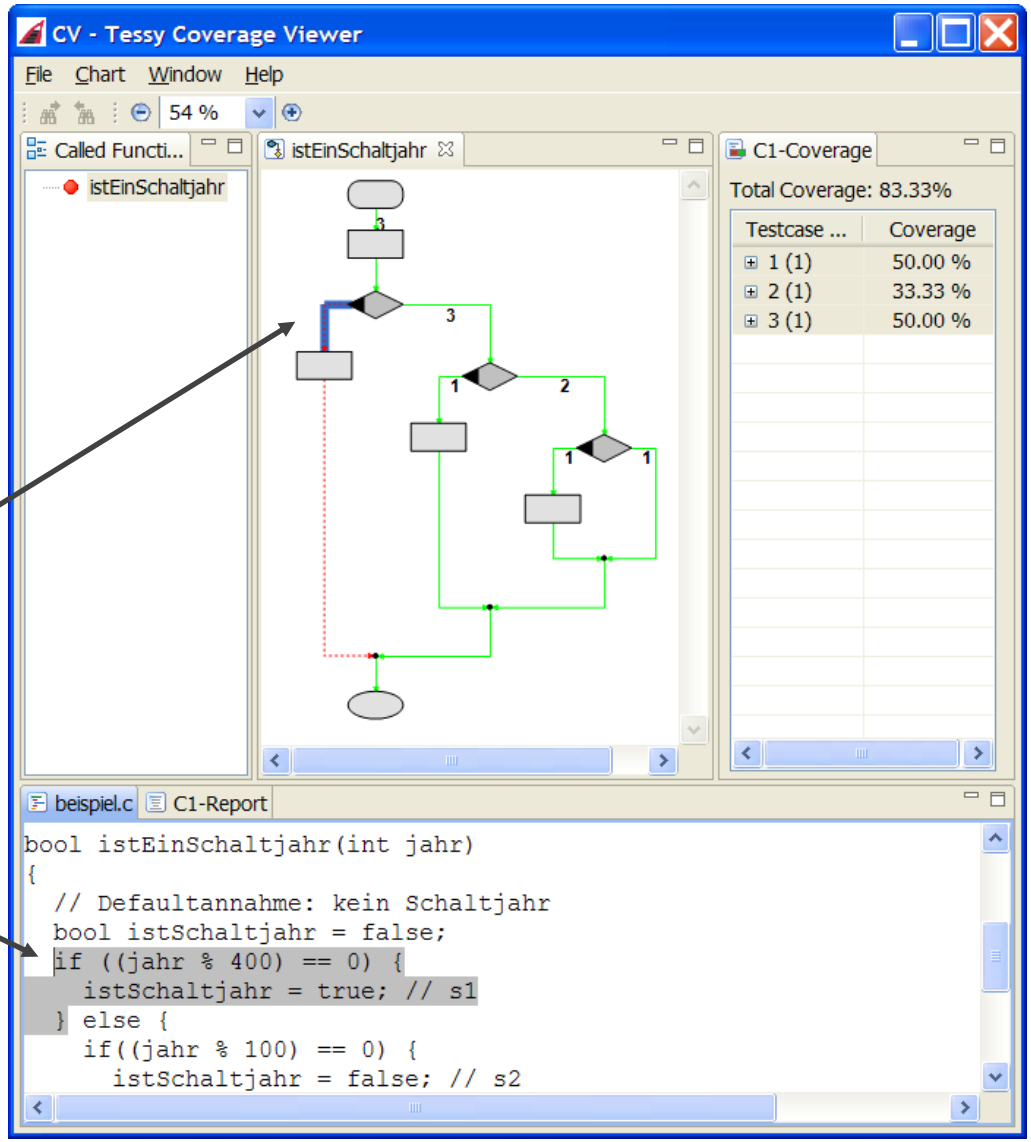
Beispiel: Werkzeug Tessy (1)

- Test mit den Eingabedaten:

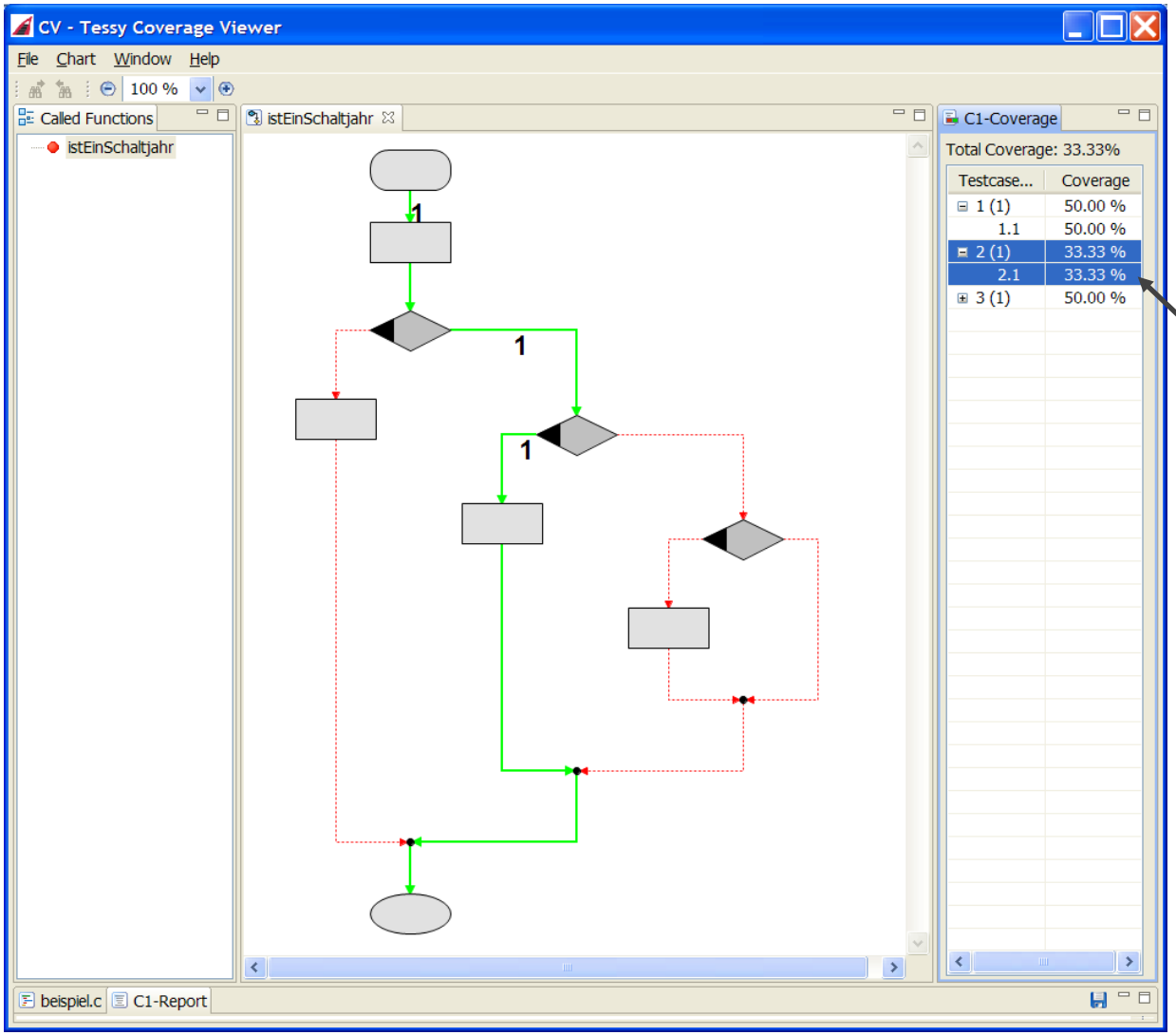
1(1) 2005	Kein Schaltjahr (Normalfall)
2(1) 1900	Ein 100er Jahr
3(1) 2008	Normales 4er Schaltjahr

Unit | Properties

Nicht ausgeführter Zweig



Beispiel: Werkzeug Tessy (2)

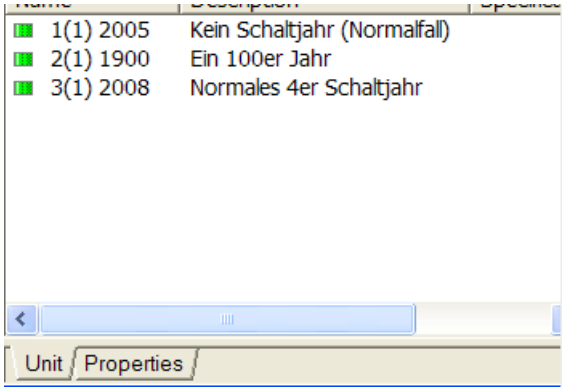


- Beispiel Tessy: Ausführungen werden je selektiertem Testfall angezeigt

Selektierter Testfall

Tessy „c1-Report“

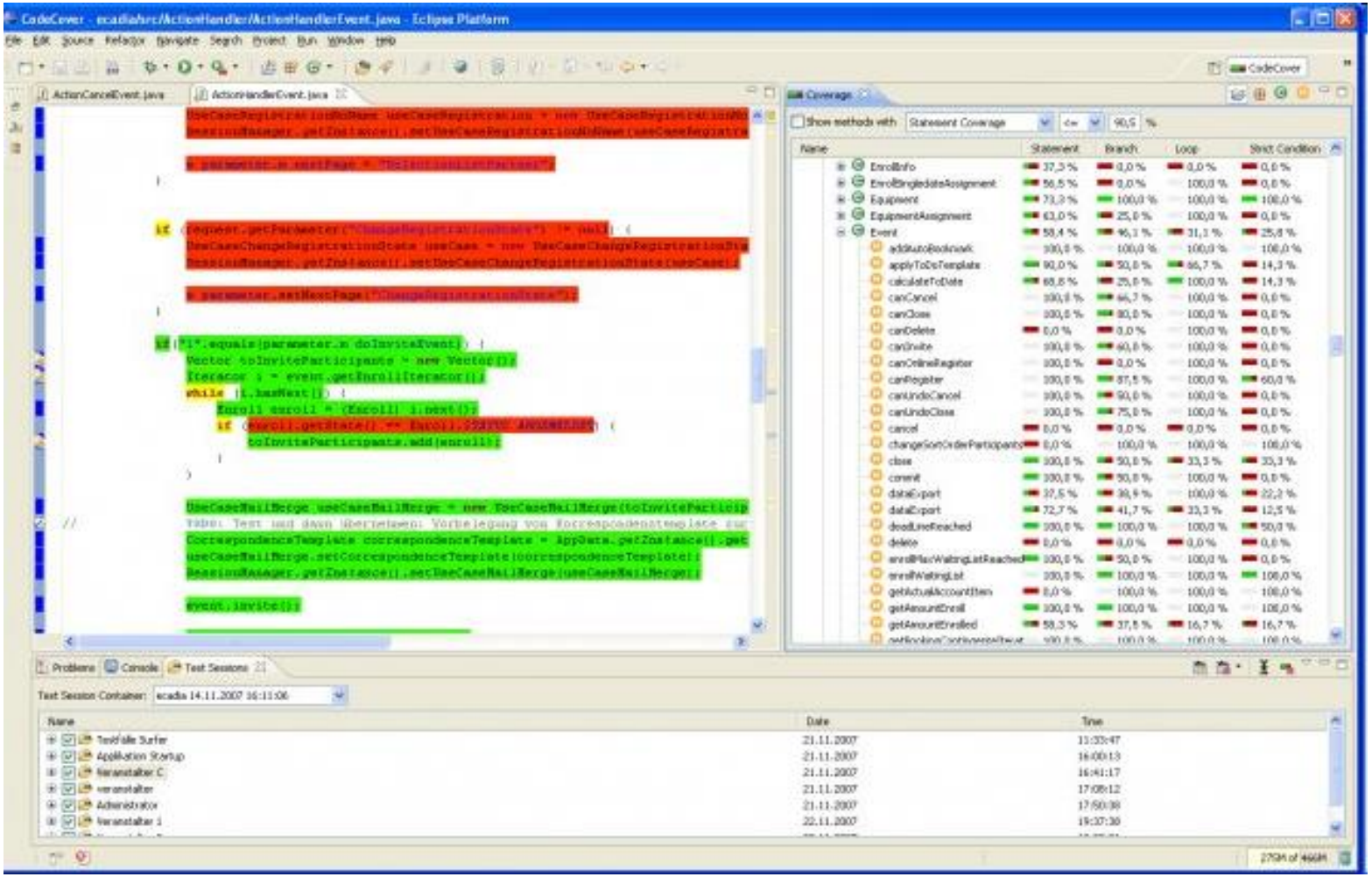
- Test mit den Eingabedaten:




```
*****************************************************************************
* MODULE                    Quadrat
* TESTOBJECT                istEinSchaltjahr
* DATE OF INSTRUMENTATION   08.10.2008 08:49:33
*****************************************************************************
* C1-COVERAGE               83.33 %
*****************************************************************************
* BRANCHES                  6
* REACHED BRANCHES          5
* UNREACHED BRANCHES        1
*****************************************************************************
  Function:  istEinSchaltjahr  File:      C:\dummy\TessyBsp\beispiel.c
  Line:      6                 Coverage:  83.33 %
  Branches:  6                 Reached:   5
  Unreached: 1
---------+-------------------------------------------------------------------
         | bool istEinSchaltjahr(int jahr)
         | {
         |   // Defaultannahme: kein Schaltjahr
         |   bool istSchaltjahr = false;
         |   if ((jahr % 400) == 0) {
****** 0 |
         |     istSchaltjahr = true; // s1
         |   } else {
         |     if((jahr % 100) == 0) {
         |       istSchaltjahr = false; // s2
       3 |
         |     } else {
         |       if((jahr % 4) == 0) {
       1 |
         |         istSchaltjahr = true; // s3
         |       }
         |     }
         |   }
       2 |
         |   return istSchaltjahr;
         | }
       1 |
```

Beispiel: CodeCover

- www.codecover.org



Datenflusstest

- Grundidee: Es wird der Datenfluss im Programm betrachtet, um Testziele zu definieren
- Variablenzugriffe werden in Klassen eingeteilt
  - Variablenzuweisung (def)
  - Verwendung in Berechnung (c-use)
  - Verwendung als Bedingungsprädikat (p-use)
- Für jede Variable muss ein Programmpfad durchlaufen werden, für den bestimmte Zugriffskriterien zutreffen (def/use-Kriterien)
  - all defs: Für jede Definition (all defs) einer Variablen wird eine Berechnung oder Bedingung getestet
  - all p-uses: Jede Kombination aus Definition und prädikativer Benutzung wird getestet
  - all c-uses: Jede Kombination aus Definition und berechnender Benutzung wird getestet
- Der Datenflusstest hat kaum praktische Bedeutung

Bedingungstest

- Testen zusammengesetzter Bedingungsausdrücke, z. B.

```
if( (A and B) OR (C AND D) ) { … }
```

- Verbreitete Metriken
  - Einfache Bedingungsüberdeckung (simple condition coverage)
  - Bedingungs-/Entscheidungsüberdeckung (condition/decision coverage)
  - Mehrfach-Bedingungsüberdeckung (multiple conditon coverage)
  - Modifizierte Bedingungs-/Entscheidungsüberdeckung (modified condition/decision coverage, MC/DC)

Bedingungstest – 1

- Einfache Bedingungsüberdeckung (simple condition coverage)
  - Je Einzelterm wird der Test auf wahr und falsch gefordert
  - Zwei Testfälle genügen
  - Achtung: Die einfache Bedingungsüberdeckung subsumiert die Zweigüberdeckung nicht!

```
(A and B) OR (C AND D)
```

Testfall	A	B	C	D	Resultat
1	wahr	wahr	falsch	falsch	wahr
2	falsch	falsch	wahr	wahr	wahr

Bedingungstest – 2

- Bedingungs-/Entscheidungsüberdeckung (condition/decision coverage)
  - Wie Einfache Bedingungsüberdeckung, aber der Gesamtausdruck muss je einmal wahr und einmal falsch ergeben
  - Anzahl Testfälle: 2
  - Subsumiert die Zweigüberdeckung

```
(A AND B) OR (C AND D)
```

Testfall	A	B	C	D	Resultat
1	wahr	wahr	wahr	falsch	wahr
2	falsch	falsch	falsch	wahr	falsch

Bedingungstest – 3

- Mehrfach-Bedingungsüberdeckung (multiple conditon coverage)
  - Test aller Wahrheitswertekombinationen der Einzelterme
  - Anzahl Testfälle: 2^n, d.h. in der Regel nicht praktisch anwendbar

```
(A and B) OR (C AND D)
```

Testfall	A	B	C	D	Resultat
1	falsch	falsch	falsch	falsch	falsch
2	wahr	falsch	falsch	falsch	falsch
3	wahr	wahr	falsch	falsch	wahr
	…				
16	wahr	wahr	wahr	wahr	wahr

Bedingungstest: MC/DC – 1

- Modifizierte Bedingungs-/Entscheidungsüberdeckung (modified condition/decision coverage, MC/DC)
- Stammt aus dem Standard der Luftfahrtindustrie DO-178B „Software Considerations in Airborne Systems and Equipment Certification“ der RTCA und wird für sicherheitskritische Software (Level A) gefordert.
- **Definition nach RTCA, DO-178B:**

  Every point of entry and exit in the program has been invoked at least once,

  every condition in a decision in the program has taken on all possible outcomes at least once,

  and each condition has been shown to affect that decision outcome independently.

  A condition is shown to affect a decision's outcome independently by varying just that decision while holding fixed all other possible conditions.

Quelle: Software Considerations In Airborne Systems and Equipment Certification, DO-178B. RTCA, 1992

© 2008-2015 Dr. Rainer Schmidberger

Bedingungstest: MC/DC – 2

- Anzahl Testfälle: n + 1, bei n Teiltermen

```
(A and B) OR (C AND D)
```

Testfall	A	B	C	D	Resultat
1	wahr	wahr	falsch	wahr	wahr
2	falsch	wahr	falsch	wahr	falsch
3	wahr	falsch	falsch	wahr	falsch
4	wahr	falsch	wahr	wahr	wahr
5	wahr	falsch	wahr	falsch	falsch

MC/DC in der Praxis

1. Die im Rahmen des „normalen“ Modul-Black-Box-Tests vorhandenen Testfälle ausführen
2. Die tatsächlich aufgetretenen Wertebelegung der Einzelterme aufschreiben
3. Terme markieren, die bereits die MC/DC-Forderung erfüllen
4. „Sinnvolle“ Eingabedaten suchen, die zu einer fehlenden Wertebelegung führen. Dann weiter mit 2.

Nutzen des Glassbox-Tests

1. **Codeüberdeckungsmaße als Metrik zur Testgüte**

   Objektives Vollständigkeitskriterium, das beispielsweise als Testendekriterium verwendet werden kann

2. **Testsuite-Erweiterung**

   Der Glassbox-Test zeigt Programmcode, der für eine Testsuite nicht ausgeführt wird und damit ungetestet bleibt.

3. **Grundlage für selektiven Regressionstest**

   Anstelle der „rerun-all"-Strategie sollen nur einzelne, ausgewählte Testfälle ausgeführt werden.

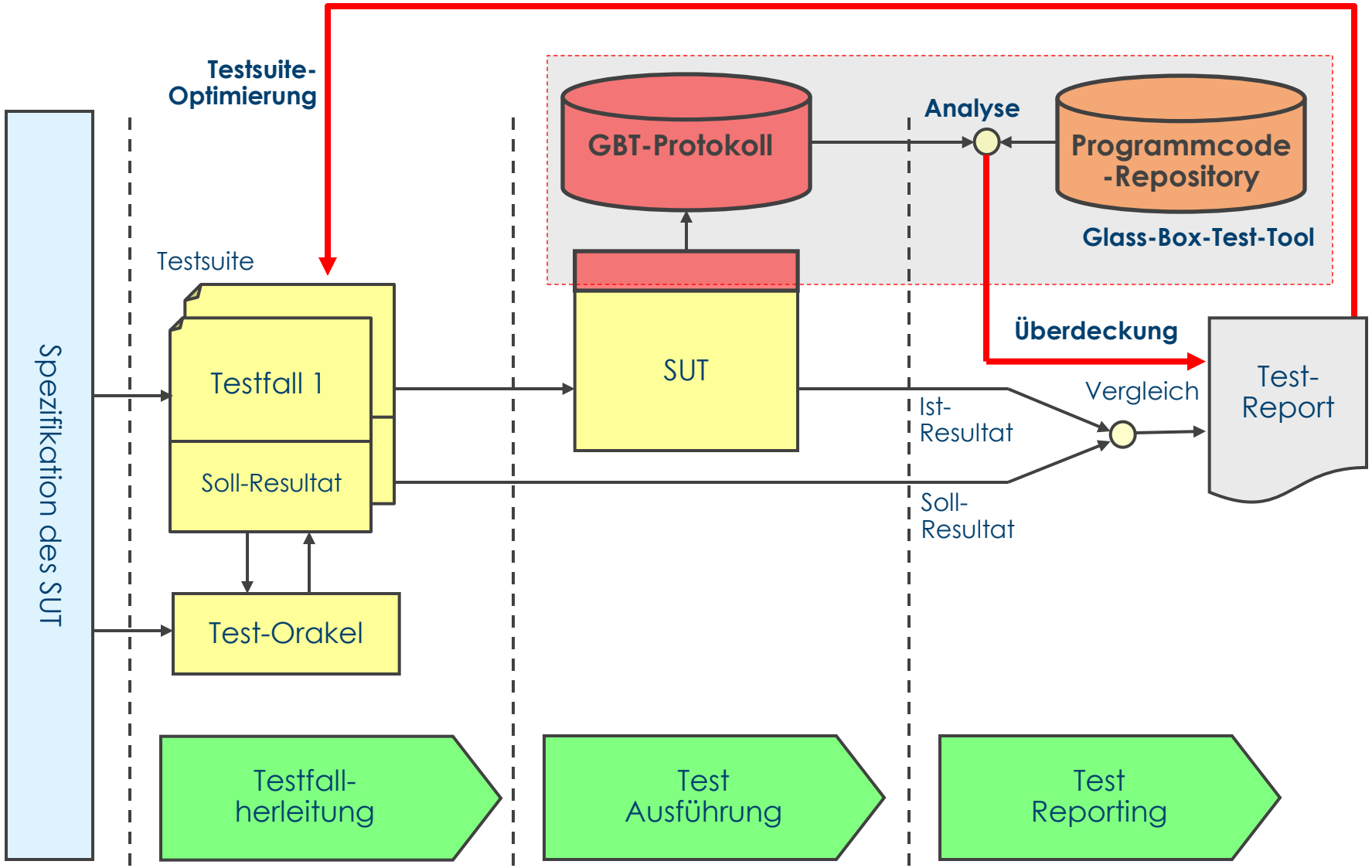
4. **Testsuite-Reduktion**

   Für den Regressionstest soll die Testsuite verkleinert werden, ohne dabei aber (wesentlich) an Testgüte einzubüßen

5. **Unterstützung beim Programmcodeverständnis**

   Der Glassbox-Test zeigt, welcher Programmcode von welchem Testfall ausgeführt wird.

Datenfluss beim Glassbox-Test



Glassbox-Test Werkzeuge

Product	Vendor	Language	SC	BC	Licence
Agitar	www.agitar.com	Java	x	x	
Bullseye	www.bullseye.com	C/C++	x	x	
CC Analyser	www.caseconsult.de	C/C++, COBOL, Java, PL/1	x		
Clover	www.cenqua.com/clover	Java, .net	x	x	
Cobertura	cobertura.sourceforge.net	Java	x	x	GPL
CodeCover	www.codecover.org	Java, COBOL	x	x	EPL
coverlipse	coverlipse.sourceforge.net	Java	x		CPL
Dynamic	www.dynamic-memory.com	C/C++	x	x	
EMMA	emma.sourceforge.net	Java	x		CPL
gcov	gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html	C/C++	x		GPL
Insure++	www.parasoft.com	C/C++	x	x	
JTest	www.parasoft.com	Java , .net	x	x	
JCover	www.mmsindia.com/JCover.html	Java	x	x	
Koalog	www.koalog.com/php/kover.php	Java	x		
LDRA Testbed	www.ldra.co.uk/softwaretesting.asp	C/C++, Ada, COBOL	x	x	
Rational Test Realtime	www-306.ibm.com/software/awdtools/test/realtime	C, C++, Java, Ada	x	x	
Rational PurifyPlus	www-306.ibm.com/software/awdtools/purifyplus	C/C++, Java, , .net	x	x	
→Semantic Designs	www.semdesigns.com	C/C++, C#, PHP, COBOL,	x	x	
TCAT	www.soft.com/Products/index.html	C/C++, Java, Ada	x	x	
Tessy	www.razorcat.com	C	x	x	
Telelogic Logiscope	www.telelogic.com/products/logiscope	C/C++, Java, Ada	x	x	
Testwell CTC++	www.verifysoft.com	C/C++	x	x	

© 2008-2015 Dr. Rainer Schmidberger

SC = Statement coverage **BC** = Branch coverage

GBT-Werkzeuge

	Anweisungs-überdeckung	Zweig-überdeckung
CodeCover [CodeCover] Version: 1.0.2.0	62,2 %	47,8 %
Clover [Clover] Verssion: 3.1.0	57,8 %	
Emma [Emma] Version: v2.1.5320	Line: 62,0 %	Block: 54,0 %
EclEmma [EclEmma] Version: 1.5.3	55,7 %	
eCobertura [eCobertura] Version: 0.9.8	64,3 %	
CodePro [CodePro] Version: 7.1.0	Instruction: 58,5 % Line: 57,1 %	Block: 60,6 %
jCoverage [jCoverage] Version: 1.10.304	Line: 38,0 %	25,0 %
PureCoverage [PureCoverage] Version: 2003.06.16.910	Line: 68,3 %	

Wir stellen fest:

- Für die gleiche Programmausführung liefern verschiedene GBT-Werkzeuge unterschiedliche Überdeckungswerte
- Die verschiedenen Überdeckungsmetriken sind keinesfalls präzise definiert

Piwowarski, P., Ohba, M., and Caruso, J. „Coverage measurement experience during function test“, *Proceedings of the IEEE 15th international Conference on Software Engineering, 1993*

[...]

When test coverage had not previously been measured, testers tended to overestimate coverage of their test cases. The first time testers measured coverage during function test, they found that the coverage was in the range of 50% to 60%. **The testers were surprised at the low percentage of coverage** they were getting. They expected a much higher percentage of code coverage. Some testers estimated that their coverage was 90% or higher.

[...]

Glass-Box-Test-Prozess

- Vorrangig wird der spezifikationsbasierte Test (Blackbox-Test) durchgeführt.
- Sind alle Möglichkeiten des spezifikationsbasierten Tests angewendet, d. h. es können keine weiteren Testfälle über die spezifikationsbasierten Methoden gefunden werden, kommt der Glass-Box-Test zum Einsatz.
- Die Testfälle werden nun mit „eingeschaltetem“ Glass-Box-Test-Werkzeug erneut ausgeführt[1].
- Aus den Resultaten des Glass-Box-Tests werden weitere neue Testfälle ermittelt. Die Soll-Resultate werden aus der Spezifikation entnommen.
- Diese weiteren Testfälle werden ausgeführt, das Ist-Resultat mit dem Soll-Resultat verglichen.

[1]Wenn davon ausgegangen werden kann, dass der Glass-Box-Test die Funktion des Systems nicht verändert, kann bereits der spezifikationsbasierte Test mit aktiviertem Glass-Box-Test-Werkzeug durchgeführt werden.

© 2008-2015 Dr. Rainer Schmidberger

Übliche Fallen beim GBT

- Die Überdeckung steigt, aber die neuen Testfälle prüfen keine kritischen Funktionen
- Die Überdeckung steigt, aber die auftretenden Fehler werden nicht wahrgenommen
- In „wildem Aktionismus“ steigt die Zahl der Testfälle, ohne die Überdeckung zu erhöhen
- Eine hohe Überdeckung führt zu einer erheblichen Überbewertung der erreichten Produktqualität

© 2008-2015 Dr. Rainer Schmidberger

Zusammenfassung Glass-Box-Test

- Der Glassbox-Test zeigt ausgeführten sowie nicht ausgeführten Programmcode an.
- Mit IEC61508 und ISO26262 wird Anweisungs- und Zweigüberdeckung (ASIL A und B) und MC/DC (ASIL D) verlangt
- Der Glassbox-Test
  - dient der Testsuite-Optimierung: Z.B. können aus dem nicht ausgeführten Programmcode neue, fehlende Testfälle abgeleitet werden.
  - liefert eine objektive Testgütemetrik
- Ein Glassbox-Test-Werkzeug ist erforderlich.
- Der Glassbox-Test bildet keine Alternative sondern eine Ergänzung zum Blackbox-Test.

© 2008-2015 Dr. Rainer Schmidberger

Kapitel 3
Testfallentwurfsverfahren

- ✓ Spezifikationsbasierter Test
- ✓ Glass-Box-Test
- **Erfahrungsbasierte Verfahren**

© 2008-2015 Dr. Rainer Schmidberger

Erfahrungsbasierte Verfahren

- „Error guessing“
  - Einsatz in höheren Testebenen, um systematische Tests zu ergänzen.
  - Kann von systematisch erstellten Testfällen „inspiriert“ werden.
  - Error Guessing kann äußerst unterschiedliche Grade von Effizienz erreichen, abhängig von der Erfahrung des Testers.
- Exploratives Testen
  - „Ad-hoc“-Testfallentwurf, Testdurchführung, Testprotokollierung
  - Die Testgüte hängt in hohem Maße von der Kompetenz der Tester ab
  - Der Ansatz, erscheint dann als sinnvoll, wenn es nur wenig oder ungeeignete Spezifikationen gibt
- Test-Ideen (RUP)
  - Grundlage bildet ein „Brainstorming-Prozess“
  - Test Ideen werden in einer Liste gesammelt und sukzessive verfeinert.

Kapitel 4
Testmanagement

- Testorganisation
- Testrollen
- Fehlermanagement
- Testmetriken
- Teststrategie

© 2008-2015 Dr. Rainer Schmidberger

Testorganisation

Typ1: Testen liegt ausschließlich in der Verantwortung des einzelnen Entwicklers. Jeder Entwickler testet seine eigenen Programme.

Typ2: Testen liegt in der Verantwortung des Entwicklungsteams. Die Entwickler testen ihre Programme gegenseitig.

Typ3: Mindestens ein Mitglied des Entwicklerteams ist für Testarbeiten abgestellt. Es erledigt alle Testarbeiten des Teams.

Typ4: Es gibt ein oder mehrere dedizierte Testteams innerhalb des Projekts (die nicht an der Entwicklung beteiligt sind).

Typ5: Eine separate Organisation (Testabteilung, externer Testdienstleister, Testlabor) übernimmt das Testen.

Test-Rollen

- **Testmanager**

  Testplanung und Teststeuerung (Kompetenz: Qualitäts-management, Projektmanagement, Personalführung)
- **Testdesigner**

  Erstellung der Testfälle und der Testspezifikation, Priorisierung (Domänenwissen, Anwenden der Testmethoden zur Herleitung der Testfälle, Dokumentation)
- **Testautomatisierer**

  Testautomatisierung (Testgrundlagenwissen, Programmiererfahrung und sehr gute Kenntnisse der eingesetzten Testwerkzeuge).
- **Testadministrator**

  Installation und Betrieb der Testumgebung (Systemadministration).
- **Tester**

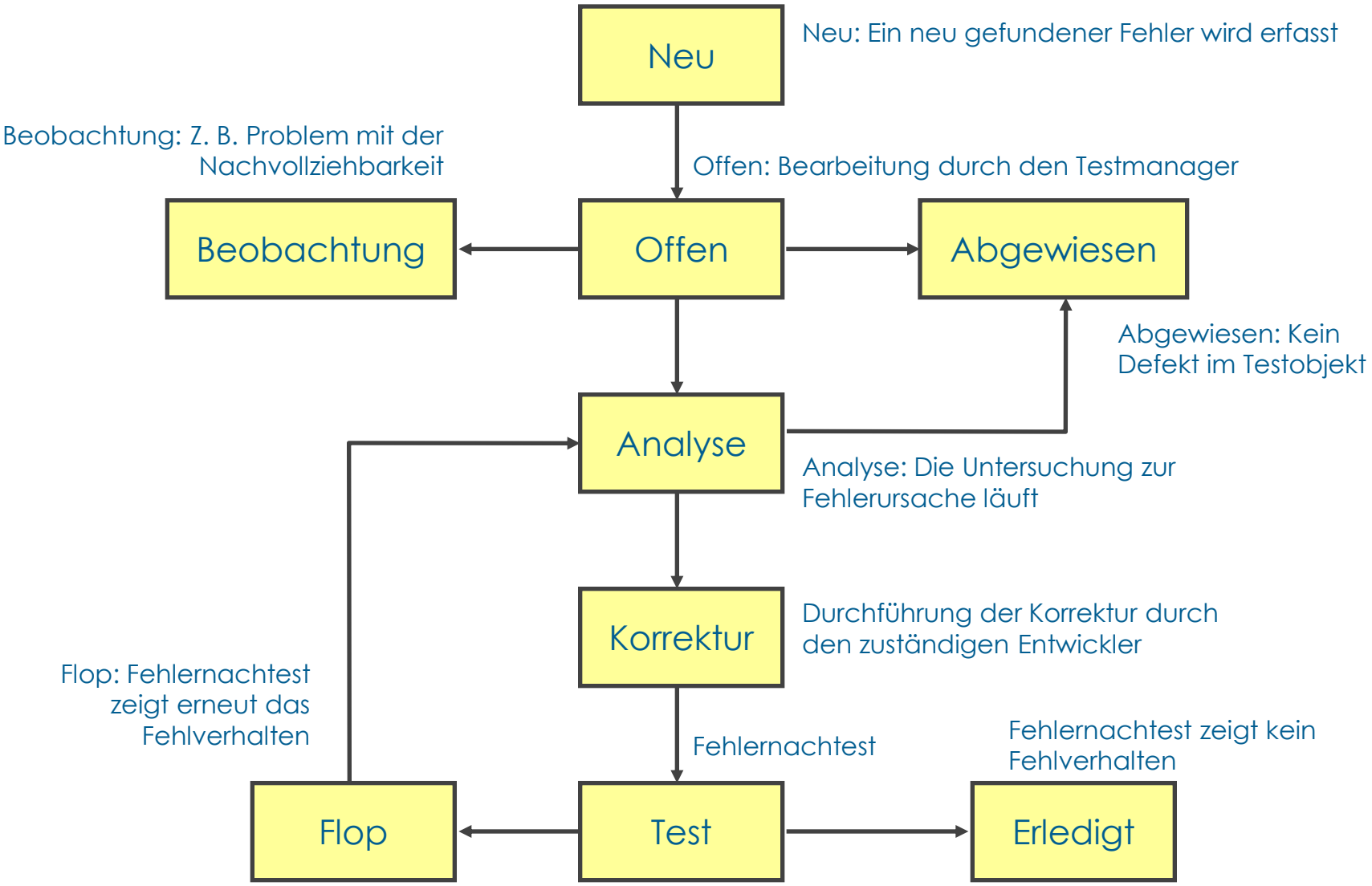
  Testdurchführung und Erstellung der Fehlerberichte (IT-Grundlagen, Testgrundlagenwissen, Bedienung der eingesetzten Testwerkzeuge, Verständnis des Testobjekts)

Kapitel 4
Testmanagement

- ✓ Testorganisation
- ✓ Testrollen
- **Fehlermanagement**
- Testmetriken
- Teststrategie

© 2008-2015 Dr. Rainer Schmidberger

Ablauf einer Fehlermeldung



Nach Linz, T, Spillner, A.: Basiswissen Softwaretest, dpunkt Verlag

© 2008-2015 Dr. Rainer Schmidberger

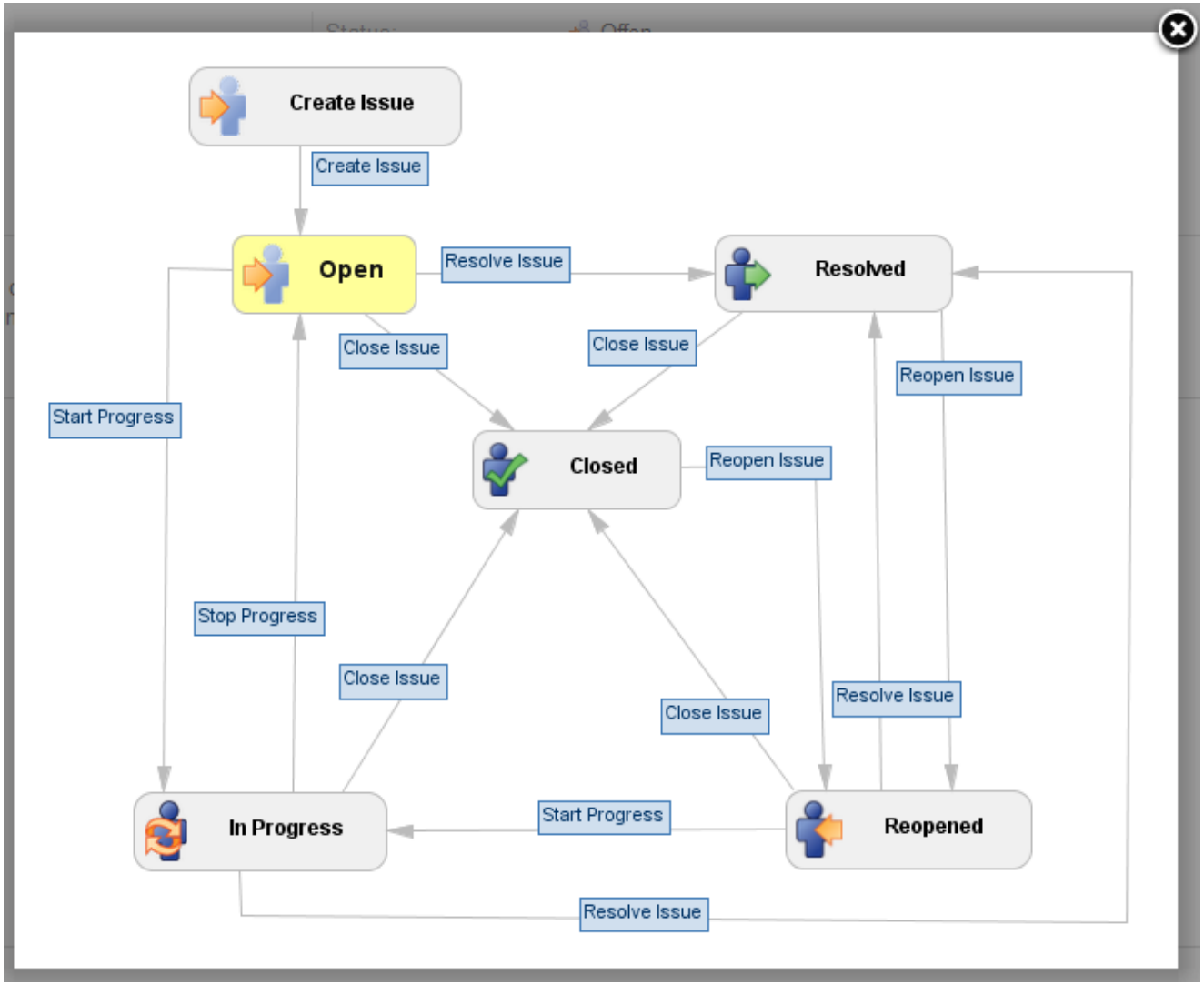
Fehlermeldung – 1

- **Nummer:** laufende, eindeutige Meldungsnummer (oft auch als „Ticket“ bezeichnet)
- **Testobjekt:** Bezeichnung des Testobjekts, in welchem System bzw. Subsystem trat der Fehler auf
- **Version:** Identifikation der genauen Version des Testobjekts
- **Testfall**: Beschreibung des Testfalls (Name, Nummer) bzw. der Schritte, die zur Reproduktion der Fehlerwirkung führen
- **Plattform:** Identifikation der HW-/SW-Plattform bzw. der Testumgebung
- **Entdecker:** Identifikation des Testers (ggf. mit Teststufe)
- **Entwickler:** Name des für das Testobjekt verantwortlichen Entwicklers oder Teams
- **Erfassung:** Datum, ggf. Uhrzeit, an dem das Problem beobachtet wurde
- **Teststufe:** Komponenten-, Intergations-, System-, Beta- … -Test

Fehlermeldung – 2

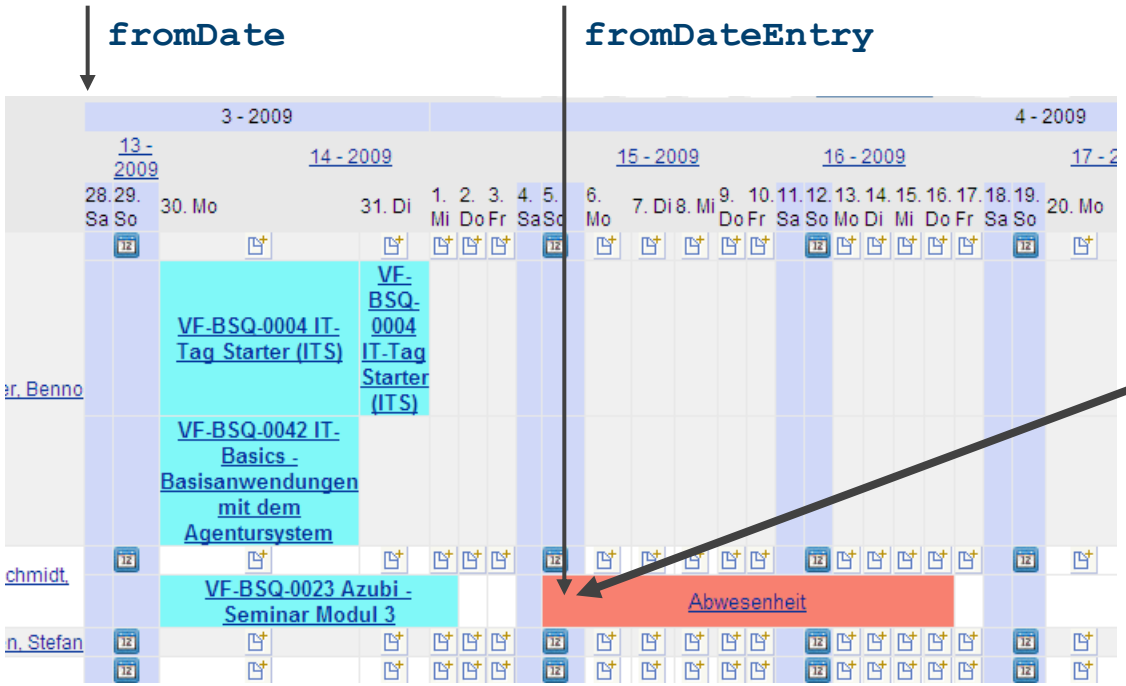
- **Problem**: Beschreibung der Fehlerwirkung; erwartete vs. tatsächliche Ergebnisse bzw. Verhalten
- **Kommentar**: Stellungnahmen der Betroffenen zum Meldungsinhalt
- **Status**: Bearbeitungsfortschritt der Meldung; möglichst mit Kommentar und Datum des Eintrags
- **Klasse**: Klassifizierung der Schwere des Problems
- **Priorität**: Klassifizierung der Dringlichkeit einer Korrektur (Auftrittswahrscheinlichkeit und Schadensmaß)
- **Anforderung**: Verweis auf die (Kunden-)Anforderung, die wegen der Fehlerwirkung nicht erfüllt oder verletzt ist
- **Fehlerquelle**: Soweit feststellbar, die Projektphase, in der die Fehlhandlung begangen wurde (Analyse, Design, Programmierung); nützlich zur Planung prozessverbessernder Maßnahmen
- **Verweis**: Querverweise auf andere zugehörige Meldungen
- **Korrektur**: Korrekturmaßnahmen des zuständigen Entwicklers mit Angabe des fehlerhaften Artefakts („injected in/by“)

Issue-Status bei Jira



Quelle: http://www.atlassian.com/software/jira

Ein Beispiel-Fehler im System „ecadia“



Fehlerwirkung: Im Kalender werden Belegungen falsch eingetragen

Z.B.: Start der Belegungen am Sonntag und nicht am Montag

```
...
double dTmp = fromDateEntry.getTime().getTime() - fromDate.getTime().getTime();
double dTmpInDays = dTmp / 1000 / 60 / 60 / 24;
long offset =  (long)dTmpInDays;
dTmp = toDateEntry.getTime().getTime() - fromDateEntry.getTime().getTime();
dTmpInDays = dTmp / 1000 / 60 / 60 / 24;
long duration = (long)dTmpInDays + 1;
...
```

- Fehlerkorrektur:
  - Die Verschiebung beim Wechsel Winterzeit/Sommerzeit muss berücksichtigt werden!

```
...
double dTmp =     fromDateEntry.getTime().getTime() +
                  fromDateEntry.get(Calendar.DST_OFFSET) -
                  fromDate.getTime().getTime() -
                  fromDate.get(Calendar.DST_OFFSET);

double dTmpInDays = dTmp / 1000 / 60 / 60 / 24;
long offset =  (long)dTmpInDays;

...
```

Fehlerdatenblatt

1. Fehler finden und dokumentieren

Gefunden durch	Betrieb, Benutzer
Aufwand zur Fehlererhebung	2 Stunden Sachbearbeiter und Helpdesk
Schadenswirkung, Risiko	Sehr lästig, verschobene Kalenderanzeige, Kalender damit praktisch nutzlos
Reproduzierbarkeit	Leicht
Priorität zur Fehlerbehebung	Hoch

2. Fehler beheben und neues Release ausliefern

Aufwand zur Fehlerlokalisierung	1 Stunde
Fehler beheben	4 Stunden
Aufwand zur Neurelease-Erstellung	2 Stunden
Fehlerstelle	Code
Fehlertyp	Codierfehler, Berechnungsfehler, Fehler im Ausdruck, API-Fehlbenutzung
Fehlerursache, Verursacher	
Zeitpunkt oder Phase der Fehlereinfügung	Größerer Umbau des Kalenders im Sommer 2008

3. Schlussfolgerungen zur Prozessverbesserung

Testsuite	Neue Testfälle zu Test des Kalenders für Winter-/Sommerzeitwechsel werden ergänzt
Prozess	Programmcode wird nach ähnlichen Programmstellen durchsucht, Kommunikation der Fehlerursache an die anderen Entwickler

Fehlerklassifikationen

- „Anleitung“ zur Fehlerabstraktion
- Bestehen in der Regel aus mehreren Kategorien mit je mehreren Klassifikationen
- Ein Fehler soll nachvollziehbar und reproduzierbar einer Klassifikation zugeordnet werden können
- Verwendungszweck
  - „In-Process“ Feedback, Prozessverbesserung, aus (häufiger wiederkehrenden) Fehlern lernen
  - Entscheidungsgrundlage zur Fehlerbehebung: „major/minor“
  - Behandlung von Gewährleistungsansprüchen, Service-Levels, Abnahmebehinderung

Fehlerklassifikationen in der Literatur

- **ODC** (Orthogonal Defect Classification)
  - Chillarege et al., „Orthogonal defect classification-a concept for in-process measurements“, TSE 1992
- **IEEE Std 1044-1993**, IEEE Standard Classification for Software Anomalies
- „**HP approach**“
  - Robert Grady 1992: “Practical software metrics for project management and process improvement”, Englewood Cliffs, NJ: Prentice Hall
- **Fehler-Taxonomie von Beizer**
  - Boris Beizer, „Software Testing Techniques“, van Nostrand Reinhold, 2. Auflage1990
- **BITKOM** Leitfaden Fehlerklassifikation für Software
  - http://www.bitkom.org/de/publikationen/38337_50074.aspx

ODC - Orthogonal Defect Classification

Fehler finden und dokumentieren

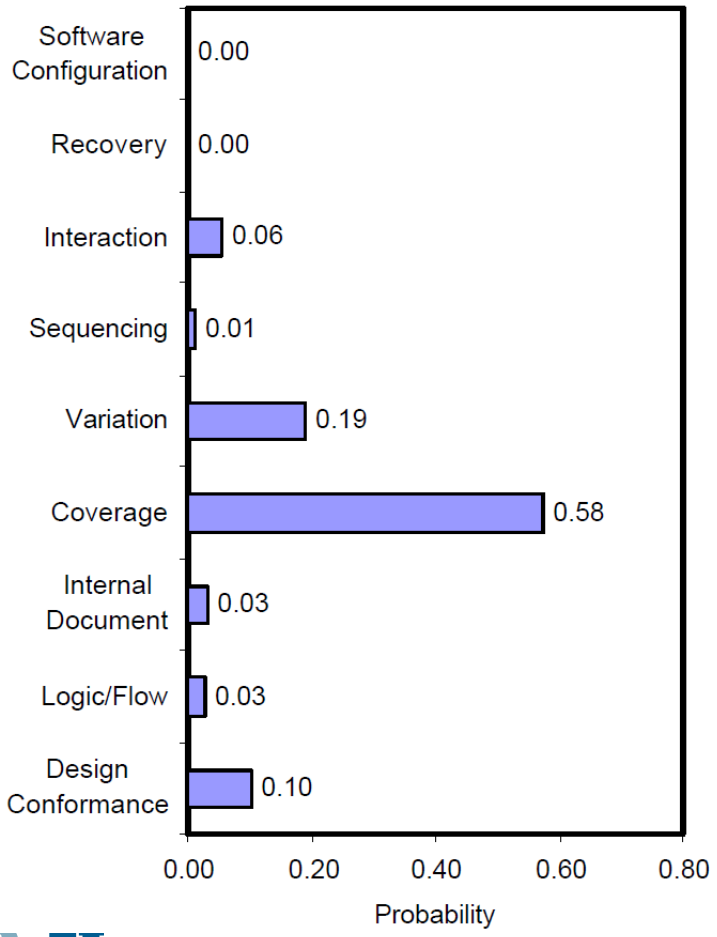
Activity	Aktivität zur Fehlerentdeckung: Code/Design Test Unit-, Function-, System (wobei mit System hier die spezielle Betriebsumgebung gemeint ist)
Defect trigger	Fehlerentdeckungstechnik, eine Untergliederung von Aktivität
Impact	Schadensauswirkung für den Benutzer

Fehler beheben

Target	Das betroffene Dokument (Entwurf, Code, …)
Defect type	Art des Fehlers (assignment, checking, algorithm, timing/serialization, interface, function, build/package/merge, documentation)
Defect qualifier	Art der Dokumentänderung (missing, incorrect, irrelevant, obscure)
Age	Zeitpunkt der Fehlereinfügung (base, new, rewritten, refix)

ODC Beispiel: Auswertung nach „Trigger"

Ram Chillarege, Kothanda Ram Prasad: "Test and Development Process Retrospective – a Case Study using ODC Triggers", IEEE DSN 2002



DESIGN CONFORMANCE: A review trigger where the implemented design is compared against a reference - either design document, pattern, or guideline.

LOGIC/FLOW: Checking for correctness or flaws using knowledge of the practice.

BACKWARD COMPATIBILITY: Examining compatibility with prior version of the product.

LATERAL COMPATIBILITY: Examining for compatibility with other products and platforms that need to work with this release.

CONCURRENCY: Serialization, shared resources, multithreaded tasks, timing, etc.

INTERNAL DOCUMENT: Inconsistencies in prologs, and sections in the same work product.

LANGUAGE DEPENDENCY: Programming standards, specific implementation considerations, environment restrictions, execution modes, etc.

SIDE EFFECTS: Usage behavior beyond design, but relevant in context. Do A; B happens.

RARE SITUATION: Unusual issues related to idiosyncrasy of environment, hardware, or software.

SIMPLE PATH: White box - Straightforward usage of code path and branches.

COMPLEX PATH: White box - Exercising conditionals, and circuitous coverage.

COVERAGE: Black box - Straightforward usage of function or single parametrized execution.

VARIATION: Black box - straightforward like coverage, but with a variety of parameters.

SEQUENCING: Black box - multiple functions, with a specific sequence.

INTERACTION: Black box - when two or more bodies of code are involved.

WORKLOAD STRESS: Pushing the limits of performance, resources, users, queues, traffic, etc. Recovery: Invoke exception handling, recovery, termination, error percolation, etc.

STARTUP/RESTART: Major events of turning on, off or changing the degree of service availability.

HARDWARE CONFIGURATION: Issues surfaced as a consequence of changes in hardware setup.

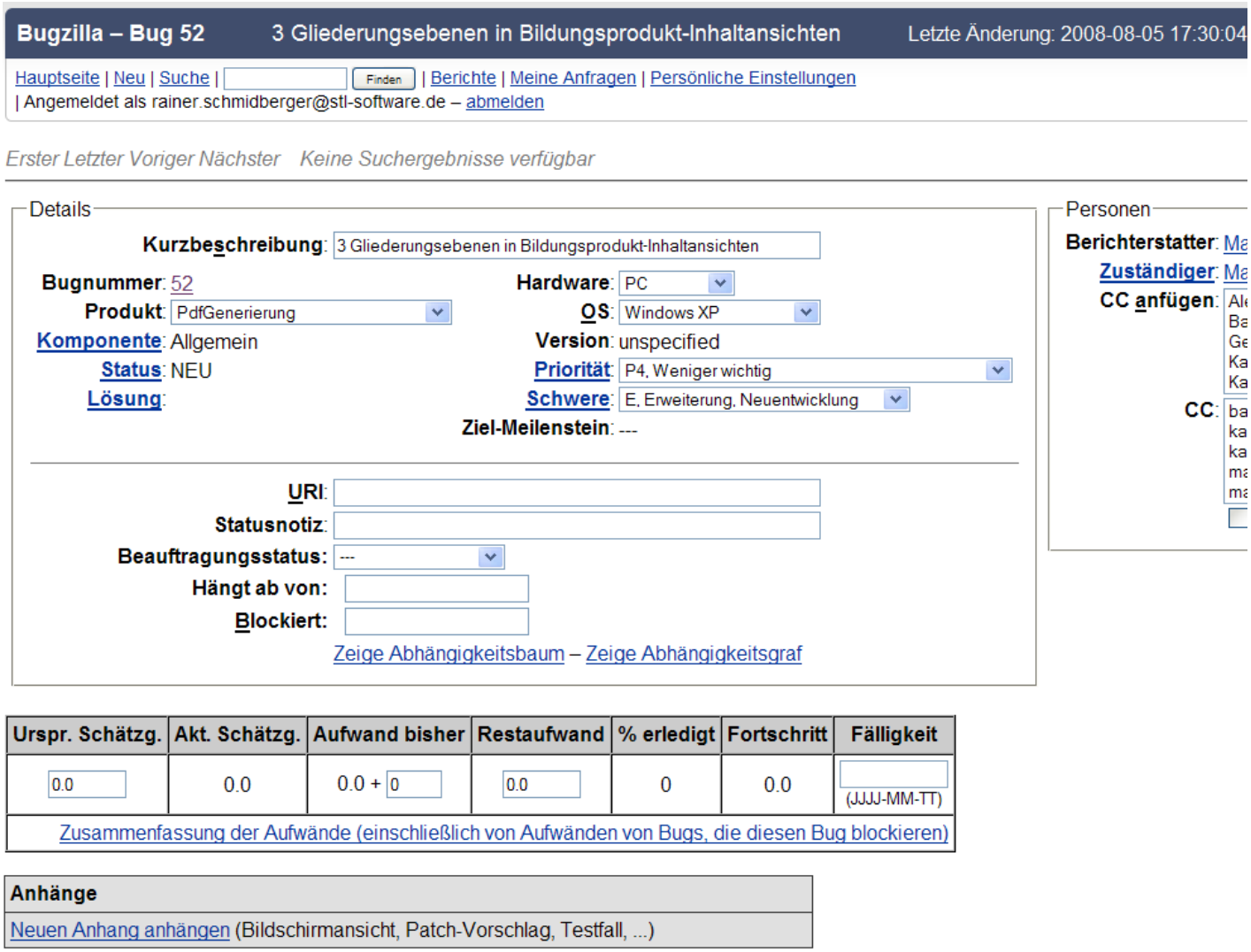
SOFTWARE CONFIGURATION: Issues surfaced as a consequence of changes to software setup.

BLOCKED TEST/NORMAL MODE: Test could not run during system test. Normal Mode - archaic: customer found nonspecific trigger.

Werkzeuge für das Bug-Tracking

- Übliche Funktionen
  - Fehler- und Änderungsmanagement
  - Native und Web-Client Versionen
  - Konfigurierbare Oberfläche und Datenfelder
  - Konfigurierbarer Workflow
  - Reporting- und Statistik-Funktionen
  - Einsatzmöglichkeit verschiedener Datenbanksysteme
- Verbreitete Werkzeuge
  - Rational ClearQuest
  - HP Quality Center (vormals Mercury)
  - Und viele weitere (http://www.testingfaqs.org/t-track.html)

Beispiel Bug-Tracking: Bugzilla – 1



Bugzilla – Bug 52 3 Gliederungsebenen in Bildungsprodukt-Inhaltansichten Letzte Änderung: 2008-08-05 17:30:04

Hauptseite | Neu | Suche | Finden | Berichte | Meine Anfragen | Persönliche Einstellungen
| Angemeldet als rainer.schmidberger@stl-software.de – abmelden

Erster Letzter Voriger Nächster Keine Suchergebnisse verfügbar

Details

Kurzbeschreibung: 3 Gliederungsebenen in Bildungsprodukt-Inhaltansichten

Bugnummer: 52
Produkt: PdfGenerierung
Komponente: Allgemein
Status: NEU
Lösung:

Hardware: PC
OS: Windows XP
Version: unspecified
Priorität: P4, Weniger wichtig
Schwere: E, Erweiterung, Neuentwicklung
Ziel-Meilenstein: ---

URL:
Statusnotiz:
Beauftragungsstatus: ---
Hängt ab von:
Blockiert:

Zeige Abhängigkeitsbaum – Zeige Abhängigkeitsgraf

Personen

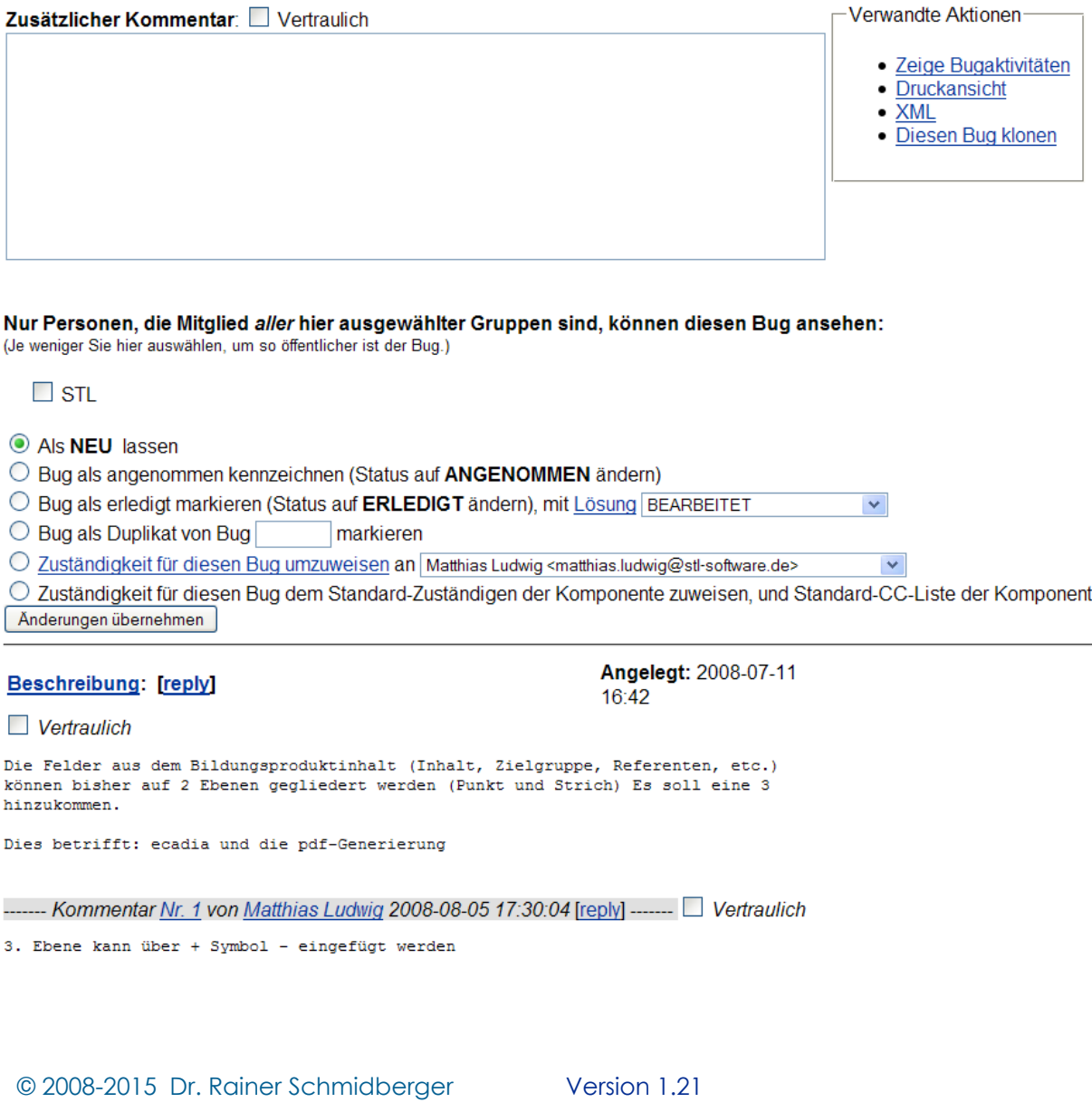
Berichterstatter: Ma
Zuständiger: Ma
CC anfügen:
CC:

Urspr. Schätzg.	Akt. Schätzg.	Aufwand bisher	Restaufwand	% erledigt	Fortschritt	Fälligkeit
0.0	0.0	0.0 + 0	0.0	0	0.0	(JJJJ-MM-TT)

Zusammenfassung der Aufwände (einschließlich von Aufwänden von Bugs, die diesen Bug blockieren)

Anhänge
Neuen Anhang anhängen (Bildschirmansicht, Patch-Vorschlag, Testfall, ...)

Beispiel Bug-Tracking: Bugzilla – 2



Beispiel Bug-Tracking: Bugzilla – 3

Bugzilla – Bugliste

Hauptseite | Neu | Suche | Finden | Berichte | Meine Anfragen | Persönliche Einstellungen
| Angemeldet als rainer.schmidberger@stl-software.de – abmelden

Die Bugs auf dieser Liste sind nach ihrer Relevanz absteigend sortiert.

Wed Aug 6 2008 12:27:53

ID	Ausw	Pri	Betriebssystem	Zuständiger	Status	Lösung	Kurzbeschreibung
55	E,	P4,	Wind	r...ar...	NEU		Performance-Optimierung ecadia-Partnersuche
42	E,	P4,	Wind	r...ar...	NEU		Einbindung Pdf-Generierung in ecadia
44	E,	P4,	Wind	r...ar...	ERLED	BEARB	Aufnahme der Qualifizierungsbereiche in Excel Export
52	E,	P4,	Wind	r...ar...	NEU		3 Gliederungsebenen in Bildungsprodukt-Inhaltansichten
43	E,	P4,	Wind	r...	NEU		Änderungen an der Produktbeschreibungsseite
38	F4,	P3,	Wind	r...ar...	ERLED	BEARB	Einrichtung Bugzilla
39	E,	P3,	Wind	k...re.de	NEU		E-Mail-Versand
8	F4,	P2,	Wind	r...ar...	ERLED	WIRDN	Zentraler Server für PDF-Generierung

8 Bugs gefunden.

Langformat | XML | Zeiterfassungsbericht

CSV | Feed | iCalendar | Spalten ändern |
Mehrere Bugs auf einmal ändern |
E-Mail an die Zuständigen dieser Bugs senden |

Suche bearbeiten | Speichere die Suche als

Aktionen: Hauptseite | Neu | Suche | Finden | Berichte | Meine Anfragen | Persönliche Einstellungen
| Angemeldet als rainer.schmidberger@stl-software.de – abmelden

Bearbeiten: Systemparameter | Benutzereinstellungsvorgaben | Systemzustandsüberprüfung | Benutzerkonten | Produkte
| Markierungstypen | Zusatzfelder | Feldwerte | Gruppen | Schlüsselwörter

Gespeicherte Suchen: Meine Bugs
alle offenen
Schlagwort ( ) zuordnen (zu Bugs) Sichern

Estimating software quality

- „If the test cases are properly developed, you can easily estimate the quality of the target software in the midst of debugging by applying the fish-in-the-pond analogy: If you detect four bugs after executing 100 test cases out of 1000, common sense says the software will carry about 40 bugs altogether. State-of-the-art software reliability theory is not that simple, but this quick and easy estimation gives you a rough idea of the software's quality."

Tsuneo Yamaura, Hitachi Software Engineering: "How To Design Practical Test Case", IEEE Software November/ December 1998

Kapitel 4
Testmanagement

- Testorganisation
- Testrollen
- Fehlermanagement
- **Testmetriken**
- Teststrategie

© 2008-2015 Dr. Rainer Schmidberger

Testmetriken – 1

Ziel	Metriken		Erhebung
Aufwands-transparenz, Nachweis von Effekten einer Prozess-verbesserung	#1	Aufwand zur Erstellung eines Testfalls	Zeiterfassung
	#2	Aufwand zur Testdurchführung je Testfall und für die gesamte Testsuite	Zeiterfassung
	#3	Setup der Testumgebung	Zeiterfassung
Transparenz der Automatisierung: Aufwände, Wirtschaftlichkeits-rechnung	#4	Testskript-Erstellungsaufwand	Zeiterfassung
	#5	Testskript-Wartungsaufwand (z. B. Aufwand je veränderter Seite oder Seitenabfolge)	Zeiterfassung
	#6	Testdurchlaufzeiten	Log-File
Qualität der Testware bewerten	#7	Einarbeitungsaufwand	Zeiterfassung
	#8	Aufwand zur Qualitätssicherung der Testware (Testsuite, Testsystem, Testdaten, Testskripte, ...)	Zeiterfassung
	#9	Anzahl der False-Positives	Fehlerdatenbank

Testmetriken – 2

Ziel	Metriken		Erhebung
Nachweis und Verbesserung der Wirksamkeit (Effektivität) Objektive Testgüteangabe	#10	Gefundene Fehler: Fehler-auswertung und Fehler-klassifizierung (gruppiert nach Teststufe, Subsystem, Ursache, ...)	Fehlerdatenbank
	#11	Anforderungsüberdeckung	Requirements-management, Traceability
	#12	Codeüberdeckung	Coverage-Werkzeug
	#13	Fehlerfindungsrate bei „Error seeding“	Fehlerdatenbank
	#14	Anzahl gefundener Restfehler (Delivered defects)	Fehlerdatenbank
	#14a	Fehlerentdeckungsrate #10 / (#10 + #14)	
	#15	Mean time to failure	Fehlerdatenbank
	#16	Mean time between failures	Fehlerdatenbank

Testmetriken – 3

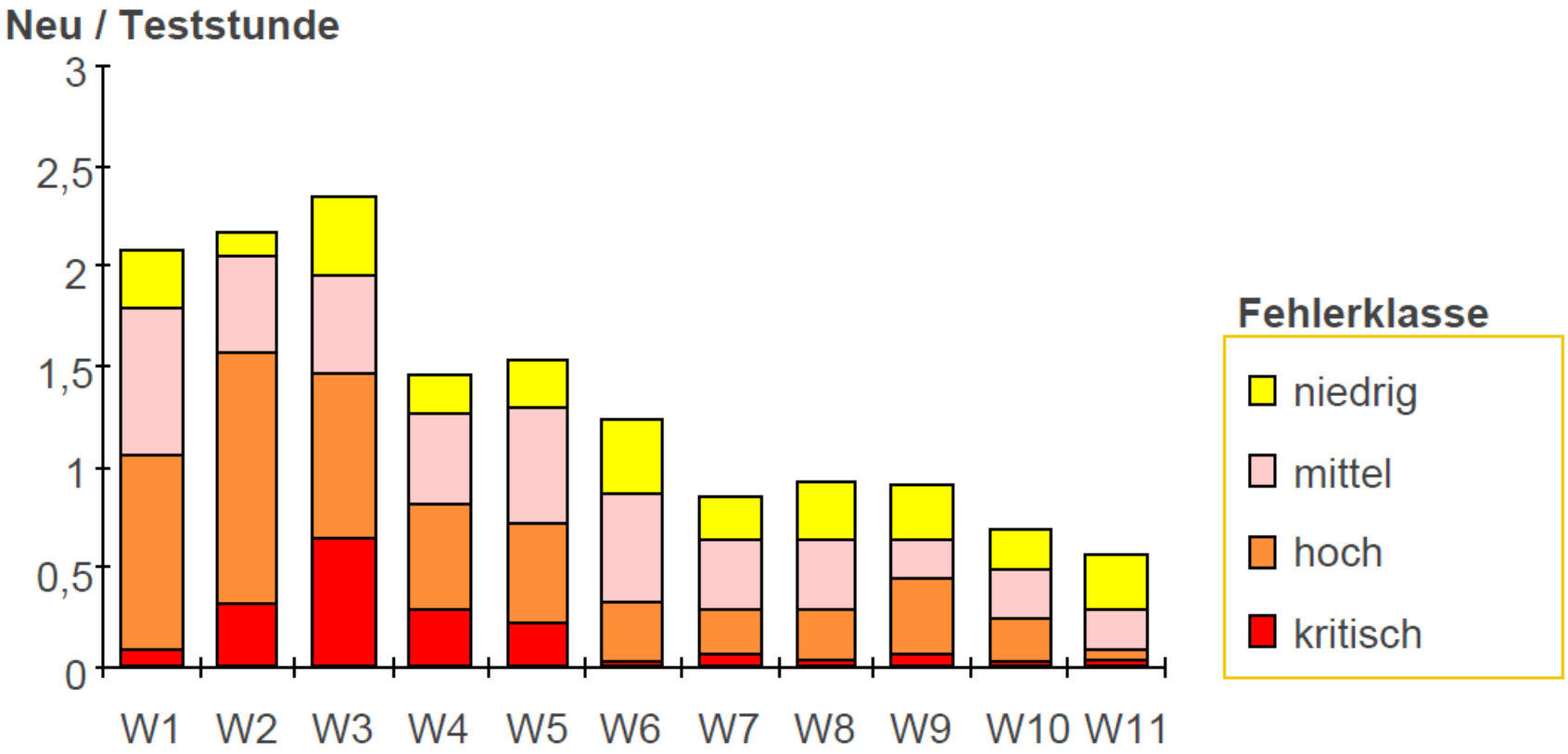
Ziel	Metriken		Erhebung
Bewertung der Testeffizienz	#17	Testeffizienz: Die Effizienz ist eine abgeleitete Metrik aus der Anzahl an gefundenen Fehlern und dem hierfür erforderlichen Testaufwand.	
Bewertung des Fehlerbehebungs prozesses	#18	Bad-fix Injections	Fehlernachtest
	#19	Defect-removal efficiency	Fehlerdatenbank
Vergleichbarkeit, Normierung	#20	Programmcode Basismetriken: LOC, Prozeduren, Schnittstellen, ggf. Function Points	Statische Analyse
	#21	Kenngrößen der Testsuite: Anzahl Testfälle	Testsuite
Qualitäts-bewertung des Prüflings Fehlermetriken	#22	Fehlerdichte	
	#23	Defektpotential	Expertenschätzung, Erfahrungswerte

Testmetriken – 4

Ziel	Metriken		Erhebung
Transparenz der Ressourcen/ Testware-Auslastung	#24	Auslastung und Verfügbarkeit	Log-Files
Personal	#25	Anteil der speziell für Testaufgaben qualifizierten Mitarbeiter im Projektteam (z. B. „Certified Tester“ o. ä.)	Personal-entwicklung
Testfortschritts-messung	#26	Anzahl durchgeführter Testfälle (mit/ohne gefundenem Fehler), Anzahl blockierter Tests (z. B. wegen nicht beseitigter Fehler)	Testprotokoll

Testmetriken – Beispiel 1

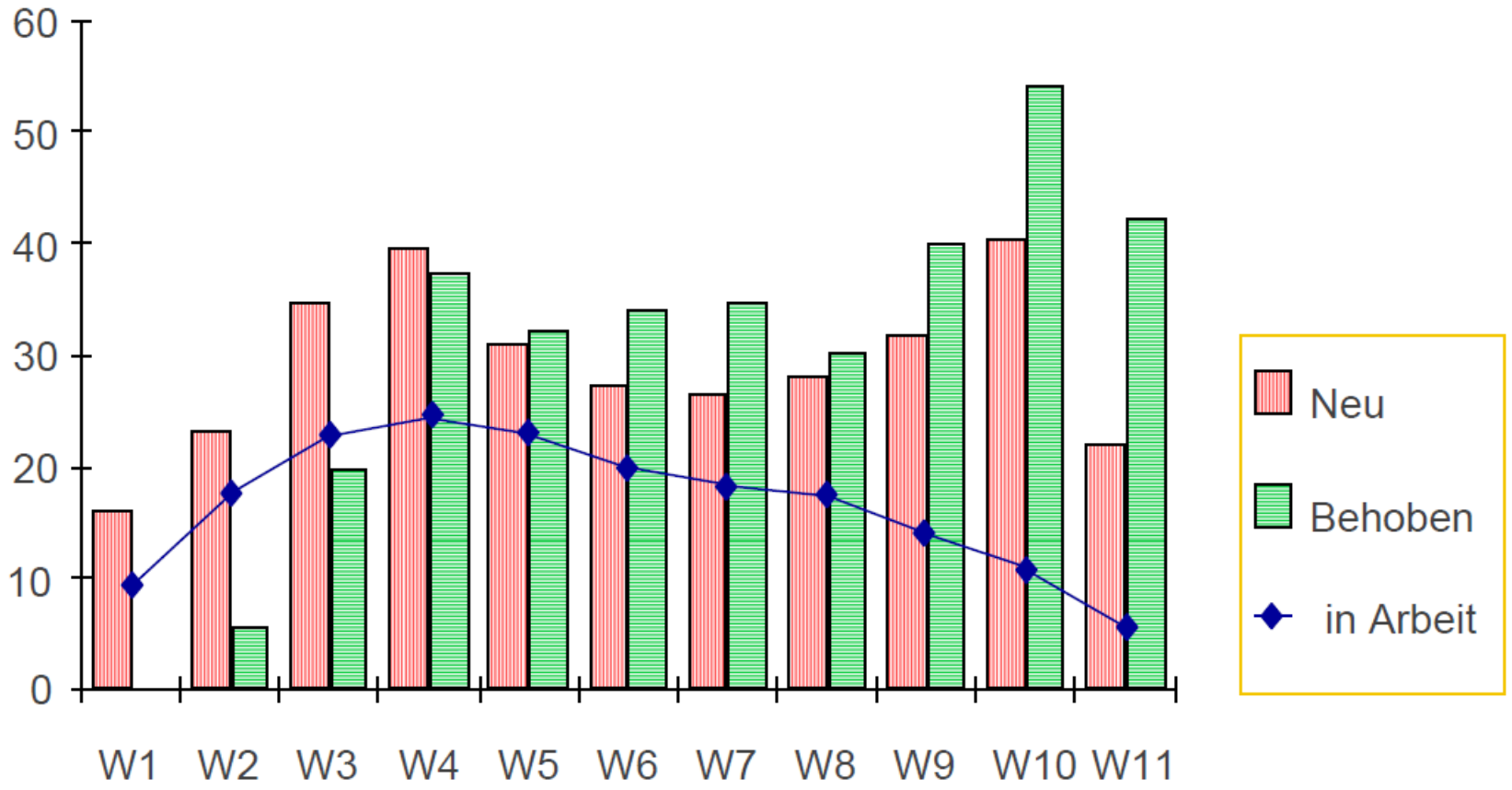
- Gefundene Fehler nach Fehlerklasse



Nach F. Fraikin,E.-H. Riedemann, A. Spillner, M. Winter

Testmetriken - Beispiel 3

- Vergleich gefundene / behobene Fehler



Nach F. Fraikin,E.-H. Riedemann, A. Spillner, M. Winter

Kapitel 4
Testmanagement

- ✓ Testorganisation
- ✓ Testrollen
- ✓ Fehlermanagement
- ✓ Testmetriken
- **Teststrategie**

© 2008-2015 Dr. Rainer Schmidberger

Teststrategie

- **Testziel** festlegen: Orientierung an Abnahmekriterien, Risiken und Folgekosten
- Festlegung von **Überdeckungskriterien** je Teilsystem
- **Priorisierung** einzelner Teilsysteme oder Anwendungsfunktionen → Risikobasiertes Testen
- Festlegung der **Aufwandsverteilung** zwischen den Teststufen
- Werkzeugeinsatz festlegen
- Festlegung des **Automatisierungsgrads**: Orientierung an der Wiederholrate der einzelnen Testausführungen

Risikobasiertes Testen

- Das risikobasierte Testen hat zum Ziel, die Fehler im Programm zu finden, von denen eine hohes Risiko auf den Projekterfolg ausgeht
- Als Risikoindikator wird in der Regel das Produkt von Fehlerauftrittswahrscheinlichkeit und erwartetem Schaden angenommen
- Die Testfälle werden priorisiert
- Je nach verfügbarem Testaufwand werden die höher priorisierten Testfälle ausgeführt

Priorisierung der Testfälle – 1

- **Eintrittswahrscheinlichkeit** der Fehlerwirkung im normalen Betrieb
  - Funktionen, die viel genutzt werden, sollten intensiver getestet werden
- **Fehlerschwere**
  - Intensiver Test auf Fehler, die hohen Schaden anrichten
- (Subjektive) **Wahrnehmung**
  - Intensiver Test auf Fehler, die einem Benutzer besonders auffallen und „stören“
- **Priorität der Anforderung**
  - Testfälle zu wichtigen Anforderungen werden bevorzugt getestet
  - Zentrale und architekturbeeinflussende Anforderungen werden bevorzugt getestet

Priorisierung der Testfälle – 2

- **Komplexität** der einzelnen Komponenten
  - Intensiver Test komplexer (im Sinne von schwer verständlicher, komplizierter) Komponenten
- **Testaufwand**
  - Testfälle mit geringem Aufwand werden bevorzugt (z. B. Bevorzugung automatisierter Testfälle)
- **Fehlerprognose**
  - Testfälle, die aus der „Erfahrung“ oft Fehler aufdecken
  - Neue Funktionen, geänderte Funktionen
- **Testabdeckung**
  - Testfälle mit hoher Abdeckung werden bevorzugt

Kapitel 5
Testdokumentation

- Testplan: IEEE 829
- Testbericht
- Testsuite / Testfälle

© 2008-2015 Dr. Rainer Schmidberger

Testplan-Inhaltsverzeichnis IEEE 829

1. Testplanbezeichnung
2. Zusammenfassung des Testziels, der Testobjekte und der zu testenden Eigenschaften
3. Testobjekte (genaue Beschreibung, Spezifikation und Version)
4. Zu testende Leistungsmerkmale
5. Leistungsmerkmale, die nicht getestet werden

Was

6. Teststrategie, Lösungskonzept, Vollständigkeitskriterien
7. Abnahmekriterien, Soll-Resultate je Testobjekt beschreiben
8. Kriterien für den Testabbruch und Testfortsetzung
9. Beschreibung der zu erstellenden Testdokumentation (Testplan, Testaufbau, Testfälle, Log-Dateien, Bericht)
10. Testtätigkeiten
11. Testinfrastruktur, Testumgebung

Wie

12. Verantwortlichkeit und Zuständigkeit
13. Personal und Training

Wer

14. Zeitplan

Wann

15. Risiken
16. Genehmigung/Freigabe

Testspezifikation nach IEEE 829

- Die **Test-Spezifikation** (test specification) umfasst
  - **Testentwurfspezifikation** (test design specification): Die Testentwurfsspezifikation 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.
  - **Testfallspezifikation** (test case specification): Die Testfallspezifikation 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.
  - **Testablaufspezifikation** (test procedure specification): Beschreibung aller Schritte zur Durchführung der spezifizierten Testfälle und Implementierung des zugehörigen Test-Designs.

Testfallspezifikation nach IEEE 829

- **Eindeutige Kennung** jedes Testfall
- **Testgegenstände**: Funktionen bzw. Eigenschaften, die von einem Testfall überprüft werden.
- **Spezifikation der Eingaben**: Detaillierte Angabe aller Eingaben und deren Beziehungen (zum Beispiel (die zeitliche) Abfolge).
- **Spezifikation der Ausgaben**: Auflistung aller erwarteten Ausgaben und weiterer Eigenschaften (wie z.B. Antwortzeiten), die zu erfassen sind und deren Werte bzw. Wertebereiche (Toleranzen)
- **Umgebungsanforderungen**: Detaillierte Beschreibung der Hardware und Softwareanforderungen und anderer Anforderungen. Definition aller speziellen Beschränkungen oder Eigenheiten
- **Wechselbeziehungen** zu anderen Testfällen: Auflistung aller Testfälle, die vor diesem Testfall auszuführen sind und Angabe der Art der Wechselbeziehungen

Teststatusbericht (Zwischenbericht)

- Ein Teststatusbericht sollte folgende Informationen enthalten:
  - Testobjekt(e)
  - Teststufe
  - Testzyklus-Datum von ... bis
  - Testfortschritt (geplante/gelaufene/blockierte Tests)
  - Fehlerstatus (neue/offene/korrigierte Fehler)
  - Risiken (neue/veränderte/bekannte Risiken)
  - Ausblick (Planung des nächsten Testzyklus)
  - Gesamtbewertung (Beurteilung der Freigabereife des Testobjekts)

Testabschlußbericht

- Prüfling, genaue Version
- Tester, Ort, Umfeld, Zeitrahmen
- Ausgeführte Testfälle
  - Angabe der Testfall-Kennungen
  - Aufwände
  - Ressourcenverbrauch
  - Verwendetes Testgeschirr
- Glassbox-Bericht
- Resultate
  - Gefundene Fehler
- Auffälligkeiten
  - Ggf. Verbesserungsvorschläge
  - Ergänzungen

Kapitel 6
Testwerkzeuge

- Typen
- Nutzen und Risiken
- Einführungsstrategien

© 2008-2015 Dr. Rainer Schmidberger

Testwerkzeuge Typenübersicht

- Unit-Test-Werkzeuge (siehe Kapitel Komponententest)
- Glassbox-Test-Werkzeuge (siehe Kapitel Glassbox-Test)

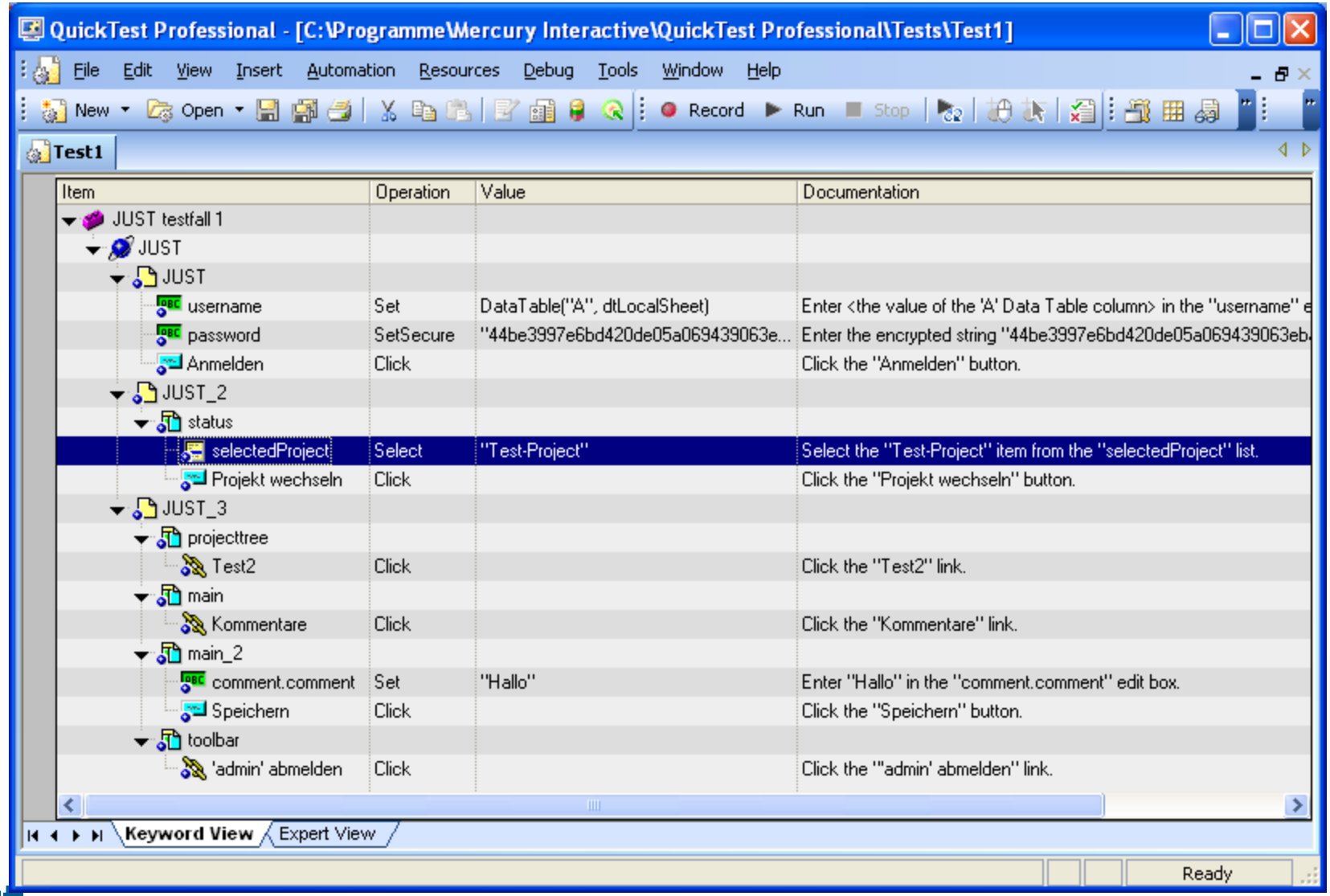
- GUI-Test-Werkzeuge
- Manual-Test-Werkzeuge
  - Verwaltung der Testfälle, die manuell ausgeführt werden
  - Unterstützung bei der Testdurchführung und Berichtserstellung
- Bug-Tracking-Werkzeuge (siehe Kapitel Fehlermanagement)

- Viele kleine „Helferlein“
  - Testdaten-Generierung
  - Werkzeug zum Vergleich zweier Text-Dokumente
  - Automatisches Einspielen von Testdatenbanken mit z. B. Anonymisierung

GUI-Testwerkzeuge

- Capture Replay:
  - Eine einmal aufgezeichnete (oder programmierte) Eingabefolge kann wiederholt ausgeführt werden
- Positiv:
  - Viele Werkzeuge sind am Markt verfügbar
  - Sinnvoller Einsatz z. B. bei Belastungstests oder Stresstests
- Negativ:
  - Abhängigkeit von der Technologie der GUI
  - Problematisch bei Änderungen der GUI; hoher Aufwand für Pflege und Wartung der Skripte

GUI-Test, Beispiel „Quicktest“

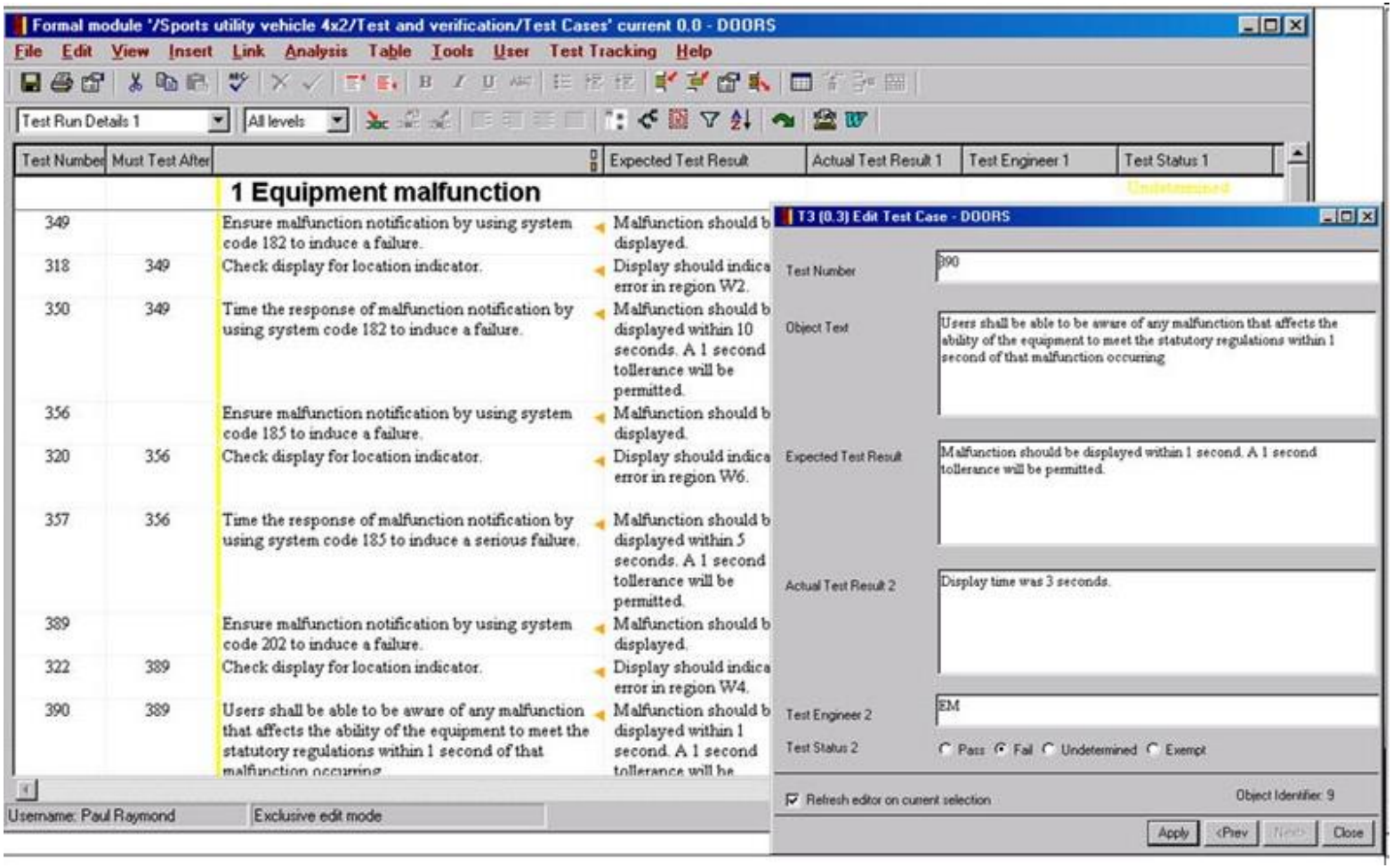


„Manual Tests“

- Testfallbeschreibungen
- Unterstützung bei der Testdurchführung
- Berichtserstellung

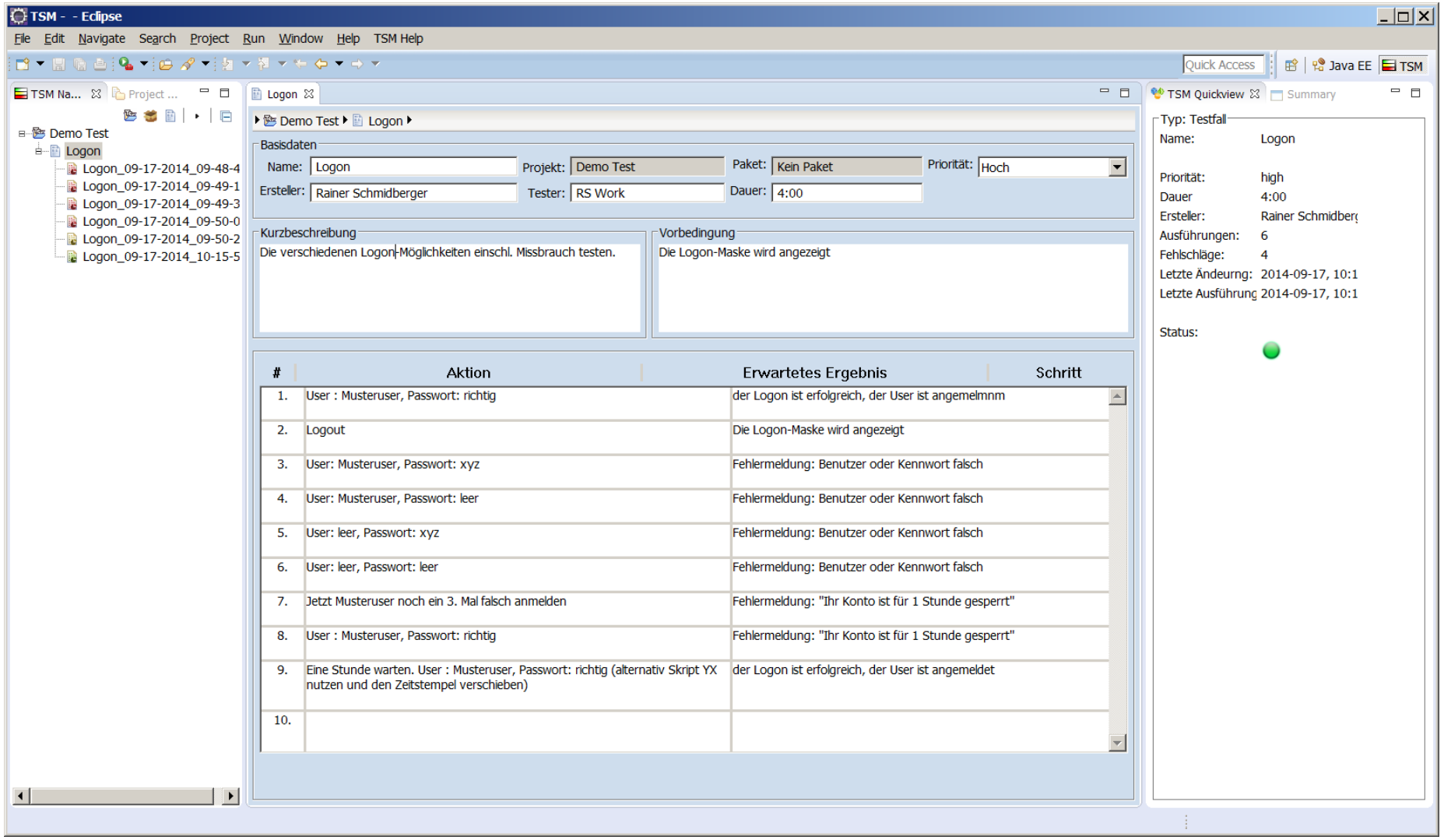
© 2008-2015 Dr. Rainer Schmidberger

Beispiel: Doors (IBM/Telelogic)



Quelle: http://www.ibm.com

Das Werkzeug TSM



Das Werkzeug TSM

- TSM ist ein Testwerkzeug, das den manuellen Test unterstützt:
  - Erfassen der Testfallspezifikation
  - Ausführen der Testfälle und protokollieren der Ergebnisse
  - Reporting
- TSM wurde 2012 im Rahmen eines Studienprojekts an der Universität Stuttgart entwickelt und steht unter EPL Lizenz.
- TSM ist derzeit als Eclipse-Plugin oder als „Stand-alone“ RCP-Anwendung verfügbar.
- Ein web-Frontend ist in einer ersten Version verfügbar: https://bitbucket.org/softwareprojectmst2014

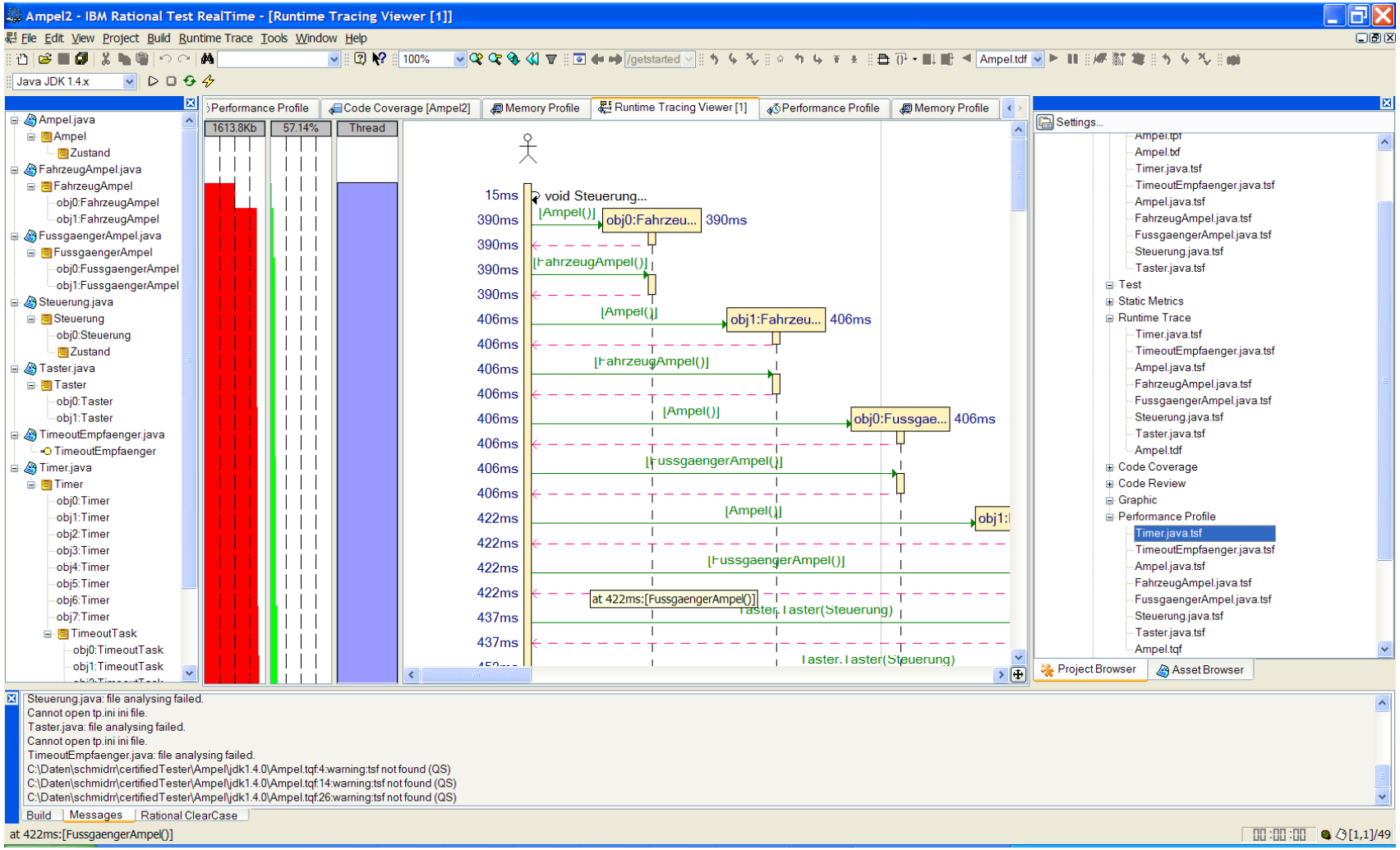
- http://tsmtest.sourceforge.net

Laufzeitanalyse

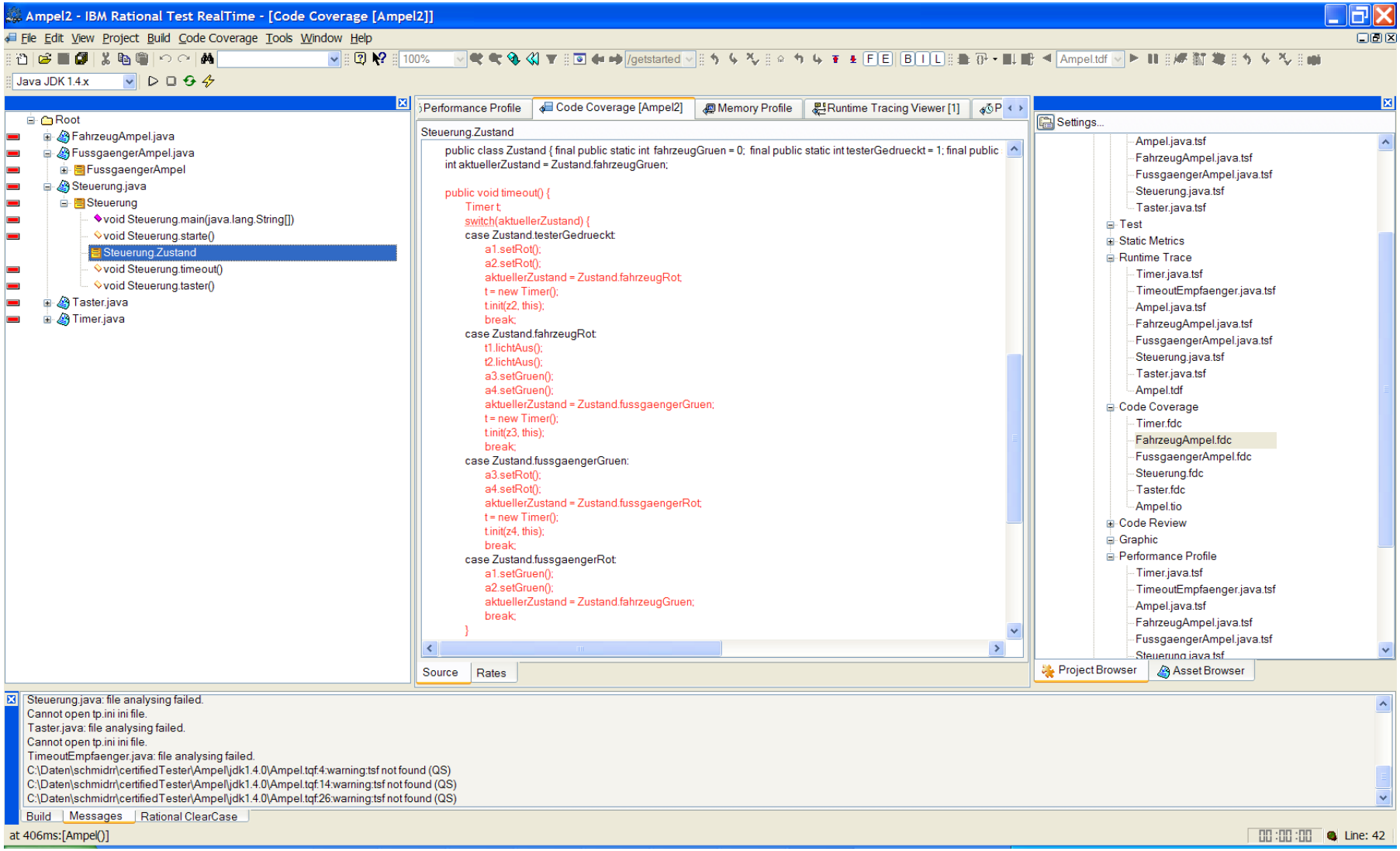
- Speicherbedarf (Memory Profile)
- Glassbox-Test-Resultate (z. B. Anweisungsüberdeckung, Zweigüberdeckung, …)
- Visualisierung von Botschaftsflüssen
- Laufzeiten (Performance Profile)

© 2008-2015 Dr. Rainer Schmidberger

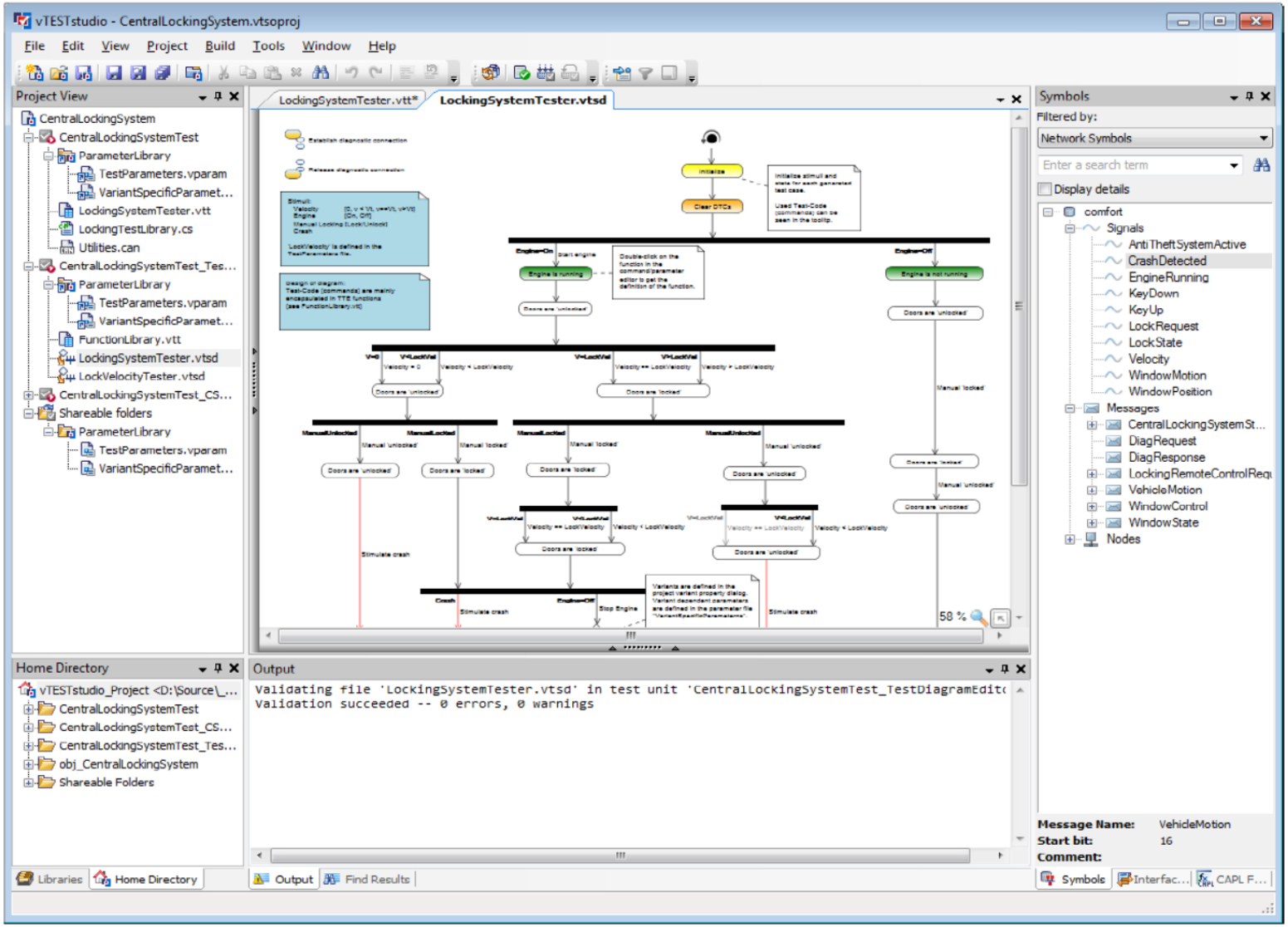
Beispiel: Rational Realtime – 1



Beispiel: Rational Realtime – 2

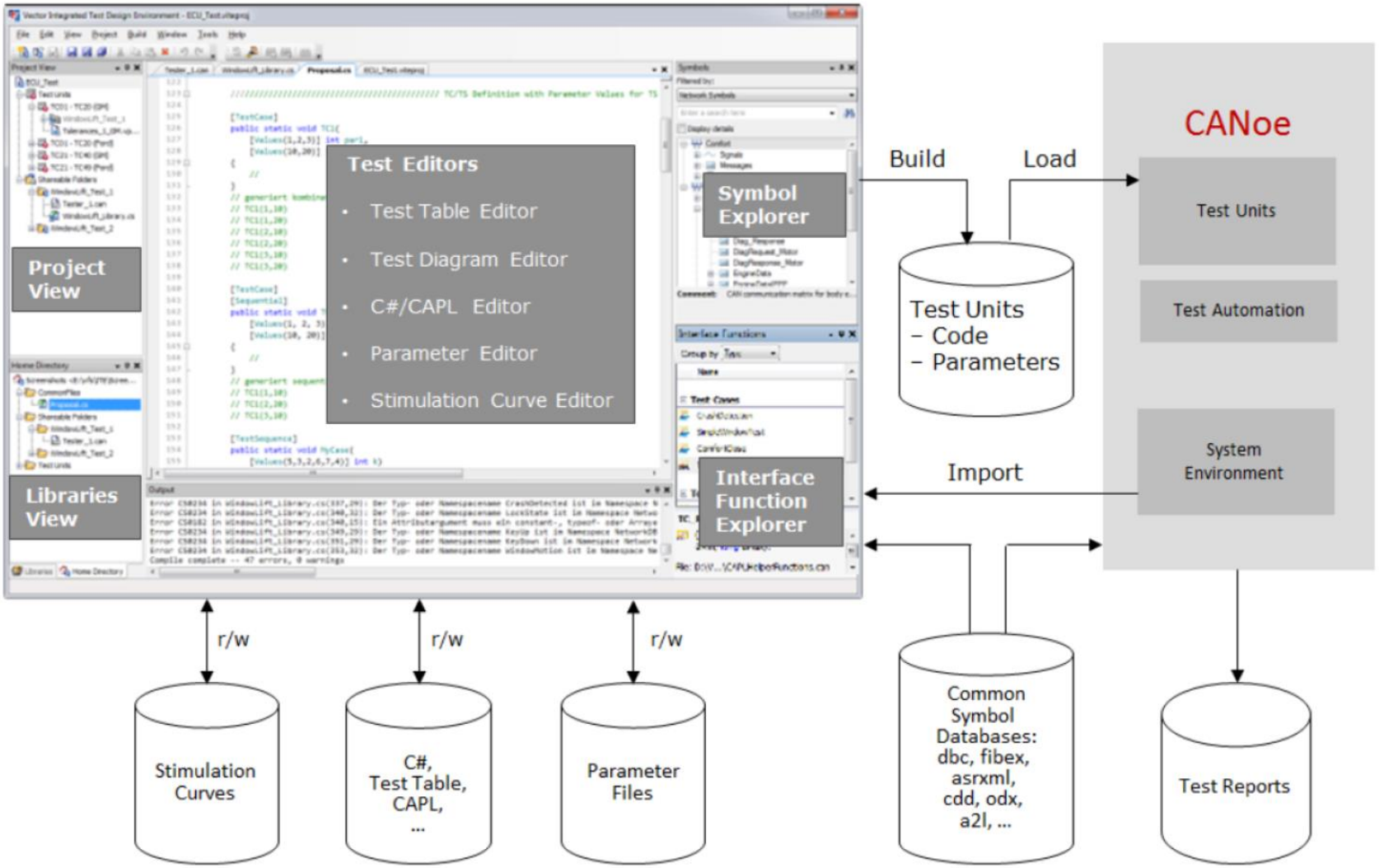


vTeststudio (1)



Quelle: Vector Informatik

vTeststudio (2)



Quelle: Vector Informatik

Werkzeugauswahl und -einführung

1. Kriterien für Werkzeugeinsatz festlegen, Commitment des Managements
2. Marktstudie
3. Bewertung vornehmen (anhand Werkzeugdemo, Probeeinsatz, Produktinformationen, ...)
4. Werkzeugfestlegung und Beschaffung

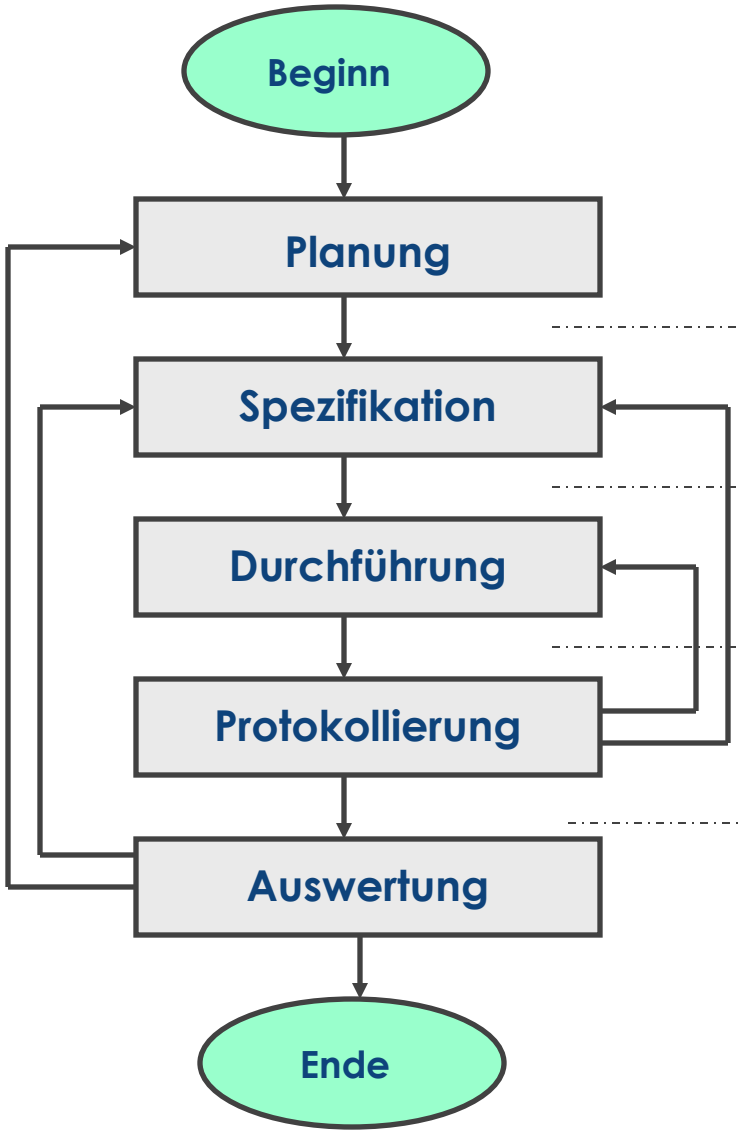
5. Pilotprojekt
6. Erfahrungen des Pilotprojekts zusammenfassen und Benutzungsunterlagen erstellen
7. Anwenderschulung
8. Breiteneinführung
9. Begleitendes Coaching

Kapitel 7
Test-Standards

- BS 7925-2, Software Testing
- IEEE 829, IEEE 1008
- ISTQB
- Testprozess-Reifegradmodelle (ISO 15504, IEC 61508, IEC 26262, TPI)
- RUP

© 2008-2015 Dr. Rainer Schmidberger

BS 7925-2, Software testing



Ziel, Strategie, Zeitplan, Personal, Ressourcen

parametrisierte (=logische) Testfälle, konkrete Testfälle(definierte Eingabedaten)

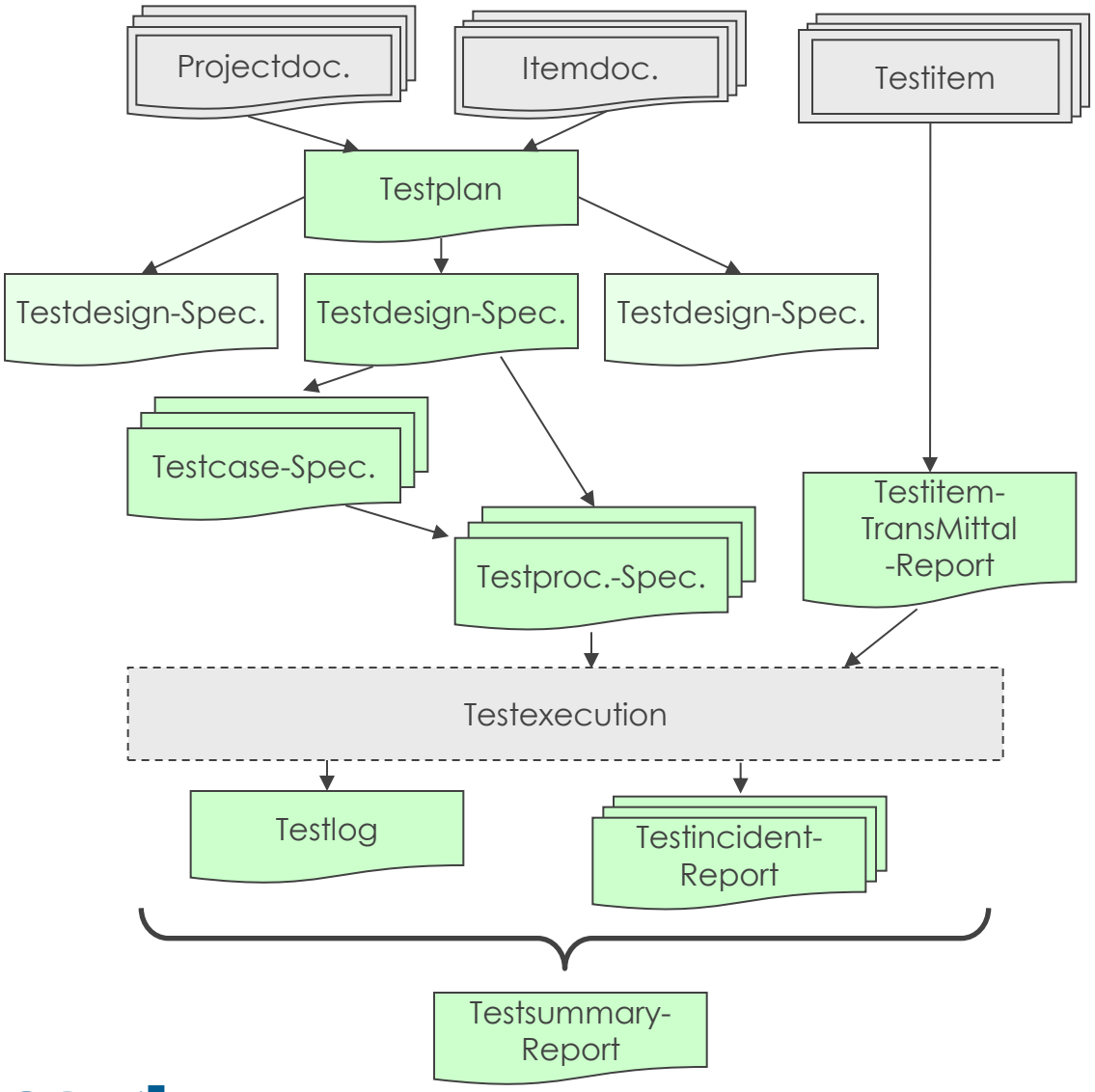
Ausführung der Testfälle

Präzise und detaillierte Protokollierung der Testdurchführung

Bewertung der Zielerreichung, Aufwandsbewertung, Fehlermanagement

Quelle: British Standard 7925-2, Software testing, Part 2; Software component, 1998

Testprozess nach IEEE 829



Die "**test design specification**"

- verfeinert den Testplan und
- beschreibt die von Test überdeckten Funktionen.
- Beschreibt die Testfälle und den Testprozess sowie
- die Annahme/Ablehnungskriterien

Die "**test case specification**"

- definiert die einzelnen Testfälle

Die "**test procedure specification**"

- definiert das genaue Ausführungs-prozedere

Document specified by this Standard

Document not specified by this standard

Testitem (not specified by this standard)

Process not specified by this standard

Quelle: IEEE Std 829-1998

© 2008-2015 Dr. Rainer Schmidberger

ISTQB

- International Software Testing Qualifications Board
- Standard Glossary of Terms
- „Certified Tester“ Ausbildungsprogramm
- Siehe www.istqb.org und www.german-testing-board.de

Qualitätsmerkmale ISO 9126

- **Funktionalität**: Inwieweit besitzt die Software die geforderten Funktionen?
- **Zuverlässigkeit**: Kann die Software ein bestimmtes Leistungsniveau unter bestimmten Bedingungen über einen bestimmten Zeitraum aufrechterhalten?
- **Benutzbarkeit**: Welchen Aufwand fordert der Einsatz der Software von den Benutzern und wie wird er von diesen beurteilt?
- **Effizienz**: Wie ist das Verhältnis zwischen Leistungsniveau der Software und eingesetzten Betriebsmitteln?
- **Änderbarkeit**: Welchen Aufwand erfordert die Durchführung vorgegebener Änderungen an der Software?
- **Übertragbarkeit**: Wie leicht lässt sich die Software in eine andere Umgebung übertragen?

ISO 9126: Funktionalität

- Vorhandensein von Funktionen mit festgelegten Eigenschaften
- Diese Funktionen erfüllen die definierten Anforderungen
- Richtigkeit: Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen, z.B. die benötigte Genauigkeit von berechneten Werten
- Angemessenheit: Eignung der Funktionen für spezifizierte Aufgaben, z.B. aufgabenorientierte Zusammensetzung von Funktionen aus Teilfunktionen.
- Interoperabilität: Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken
- Ordnungsmäßigkeit: Erfüllung von anwendungsspezifischen Normen, Vereinbarungen, gesetzlichen Bestimmungen und ähnlichen Vorschriften
- Sicherheit: Fähigkeit, unberechtigten Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten zu verhindern.

ISO 9126: Zuverlässigkeit

- Reife: Geringe Versagenshäufigkeit durch Fehlzustände
- Fehlertoleranz: Fähigkeit, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren
- Wiederherstellbarkeit: Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen

Testprozess-Reifegradmodelle

Reifegradmodell	Erscheinungsjahr und Urheber	Kurzbeschreibung
CMM / CMMI SPICE / ISO 15504 IEC 61508, ISO 26262	SEI, 1991 ISO, 1998, 2011	▪ Test liegt zwar nicht im Fokus wird aber behandelt
TMM (Test Maturity Model)	Burnstein et al., 1996 (einige Publikationen, ein Buch)	▪ An CMM angelehnt mit fünf Stufen; „Maturity goals“ enthalten die (abstrakten) Ziele je Stufe ▪ Fokus auf Organisationsstrukturen und Managementaktivitäten
TPI Test Process Improvement	Koomen und Pol, 1999 (ein Buch)	▪ Beschreibt 20 Schlüsselbereiche (Test Strategie, Werkzeuge, Metriken , ...) ▪ Erfüllungsgrad wird durch „Checkpoints“ bestimmt
TIM Test Improvement Model	Ericson et al., 1997 (eine Publikation)	▪ An CMM angelehnt mit fünf Stufen sowie fünf Schlüsselbereichen ▪ Je Stufe und Schlüsselbereich sind (abstrakt) Aktivitäten beschrieben

ISO 15504 (SPICE)

- Spice ist ein Prozessreifegradmodell und prüft Prozesse, definiert aber keine!

Abfolge im „V“

Engineering Process Group (ENG)

- ENG.1 Requirements elicitation
- ENG.2 System requirements analysis
- ENG.3 System architectural design
- ENG.4 Software requirements analysis
- ENG.5 Software design
- ENG.6 Software construction

- ENG.7 Software integration
- **ENG.8 Software testing**
- ENG.9 System integration
- **ENG.10 System testing**
- ENG.11 Software installation
- ENG.12 Software and system maintenance

© 2008-2015 Dr. Rainer Schmidberger

Spice - Software testing

Process ID	ENG.8
Process Name	Software testing
Process Purpose	The purpose of the Software testing process is to confirm that the integrated software meets the defined software requirements.
Process Outcomes	As a result of successful implementation of this process: 1) a **strategy is developed** to test the integrated software according to the priorities and categorization of the software requirements; 2) a **test specification** for software test of the integrated software is developed that demonstrates compliance to the software requirements; 3) the integrated software is verified **using the test cases**; 4) **results of software testing are recorded**; 5) **consistency** and bilateral traceability are established between software requirements and software test specification including test cases; and 6) a **regression test strategy** is developed and applied for re-testing the integrated software when a change in software items occur. NOTE 1: The test specification for software testing includes the test design specification, test procedure specification and test case specifications. NOTE 2: The verification is performed according to the test cases NOTE 3: The test results for software testing include the test logs, test incident reports and test summary reports.

IEC 61508

- Software safety lifecycle: overview

		modified.				
9.3	Software design and development	**Software module testing:** To verify that the requirements for software safety (in terms of the required software safety functions and the software safety integrity) have been achieved – to show that each software module performs its intended function and does not perform unintended functions.	Software modules.	7.4.7	Software module test specification; source code listing; code review report.	Software module test results; verified and tested software modules.
9.3	Software design and development	**Software integration testing:** To verify that the requirements for software safety (in terms of the required software safety functions and the software safety integrity) have been achieved – to show that all software modules, components and subsystems interact correctly to perform their intended function and do not perform unintended functions.	Software architecture; software system.	7.4.8	Software system integration test specification.	Software system integration test results; verified and tested software system.

IEC 61508

7.4.7 Requirements for software module testing

NOTE 1 – See also tables A.5, B.2, B.3 and B.6.

NOTE 2 – Testing that the software module correctly satisfies its test specification is a verification activity (see 7.9). It is the combination of code review and software module testing that provides assurance that a software module satisfies its associated specification, i.e. it is verified.

7.4.7.1 Each software module shall be tested as specified during software design (see 7.4.5).

7.4.7.2 These tests shall show that each software module performs its intended function and does not perform unintended functions.

NOTE 1 – This does not imply testing of all input combinations, nor of all output combinations. Testing all equivalence classes (see C.5.7 of IEC 61508-7) or structure based testing (see C.5.8 of IEC 61508-7) may be sufficient. Boundary value analysis (see C.5.4 of IEC 61508-7), control flow analysis (see C.5.9 of IEC 61508-7) or sneak circuit analysis (see C.5.11 of IEC 61508-7) may reduce the amount of test cases to an acceptable number. Analysable programs (see C.2.7 of IEC 61508-7) make the requirements easier to fulfil.

NOTE 2 – Where the development uses formal methods (see C.2.4 of IEC 61508-7), formal proofs (see C.5.13 of IEC 61508-7) or assertions (see C.3.3 of IEC 61508-7), such tests may be reduced in scope.

NOTE 3 – Statistical evidence may be used as well (see annex D of IEC 61508-7).

7.4.7.3 The results of the software module testing shall be documented.

7.4.7.4 The procedures for corrective action on failure of test shall be specified.

IEC 61508

7.4.8 Requirements for software integration testing

NOTE 1 – See also tables A.5, B.2, B.3 and B.6.

NOTE 2 – Testing that the software is correctly integrated is a verification activity (see 7.9).

7.4.8.1 Software integration tests shall be specified concurrently during the design and development phase.

7.4.8.2 The specified software integration tests shall specify the following:

a) the division of the software into manageable integration sets;

b) test cases and test data;

c) types of tests to be performed;

d) test environment, tools, configuration and programs;

e) test criteria on which the completion of the test will be judged; and

f) procedures for corrective action on failure of test.

IEC 61508

Table A.5 – Software design and development: software module testing and integration (see 7.4.7 and 7.4.8)

	Technique/Measure*	Ref	SIL1	SIL2	SIL3	SIL4
1	Probabilistic testing	C.5.1	---	R	R	HR
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
3	Data recording and analysis	C.5.2	HR	HR	HR	HR
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	HR
5	Performance testing	C.5.20 Table B.6	R	R	HR	HR
6	Interface testing	C.5.3	R	R	HR	HR

NOTE 1 – Software module and integration testing are verification activities (see table A.9).

NOTE 2 – Appropriate techniques/measures shall be selected according to the safety integrity level.

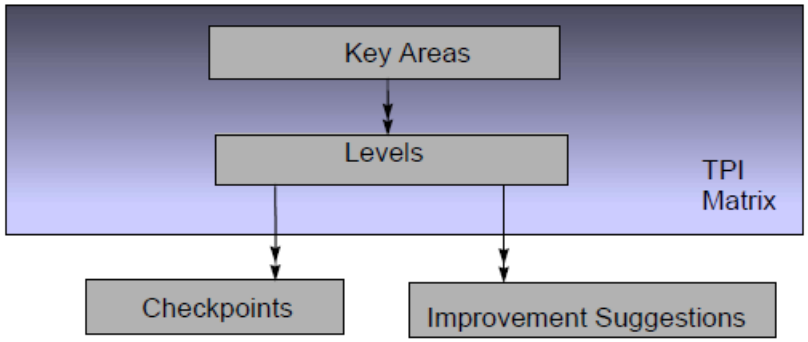
* A numbered technique/measure shall be selected according to the safety integrity level.

HR: Highly recommended
R: Recommended
NR: not recommended

© 2008-2015 Dr. Rainer Schmidberger

TPI Automotive

- Automotive-Variante des „Test Process Improvement“ von Sogetti
- 21 Key Areas
- Die Grundlagen von TPI sind allerdings nicht (z. B. empirisch) belegt
- Das Modell von TPI „ähnelt“ CMM



© 2008-2015 Dr. Rainer Schmidberger

TPI Automotive Key Areas

Key Area	Key Area
Test strategy	Test functions and training
Life-cycle model	Scope of methodology
Moment of involvement	Communication
Estimation and planning	Reporting
Test design techniques	Defect management
Static test techniques	Testware management
Metrics	Test process management
Test automation	Evaluation
Test environment	Low-level testing
Office and laboratory environment	Integration testing
Commitment and motivation	

- Beispiel: Checkpoint für die Key Area „Test Stategy“

Checkpoints

1.B.1 Coordination takes place between the different high-level tests, often the system-, acceptance- and production acceptance test, or supplier and commissioner side testing, in the field of test strategy (risks, quality characteristics, area of consideration of the test and planning).

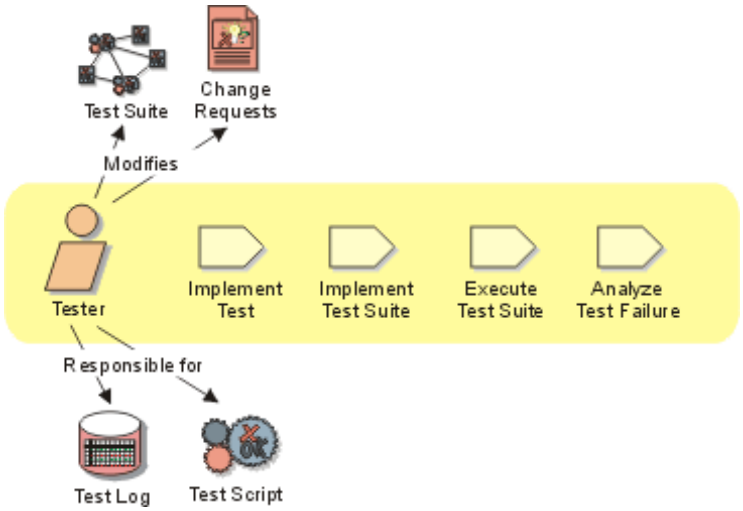
1.B.2 The result of the coordination is a coordinated strategy, which is documented. During the total test process this strategy is controlled.

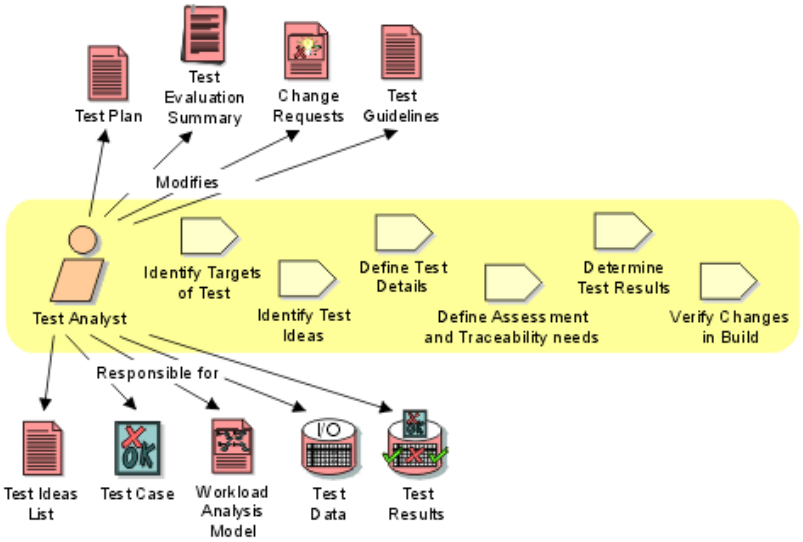
TPI Test Maturity Matrix

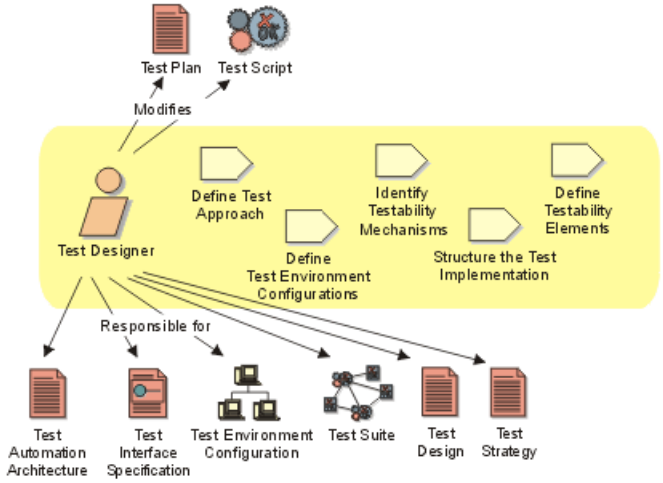
	Key area													
1	Test strategy		A					B				C		D	
2	Life-cycle model		A			B									
3	Moment of involvement			A				B				C		D	
4	Estimating and planning				A							B			
5	Test design techniques		A		B						C				
6	Static test techniques					A		B							
7	Metrics						A			B			C		D
8	Test automation				A				B			C			
9	Test environment				A				B						C
10	Office and laboratory environment				A										
11	Commitment and motivation		A				B						C		
12	Test functions and training				A			B			C				
13	Scope of the methodology					A		B				C			D
14	Communication			A		B							C		
15	Reporting		A			B		C					D		
16	Defect management		A				B		C						
17	Testware management			A			B				C				
18	Test process management		A		B								C		
19	Evaluation			A				B			C				
20	Low level testing					A		B		C					
21	Integration testing				A			B				C			

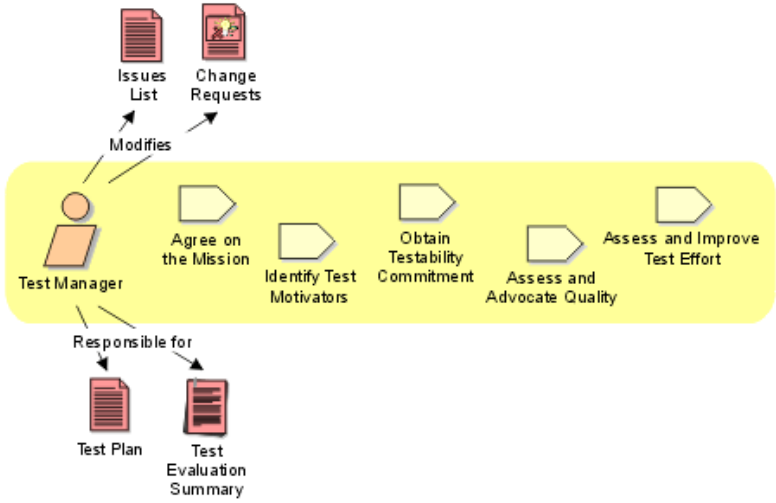
- A, B, C, D stehen für einen „Maturity Level“
- Je Key Area sind die Checkpoints für den jeweiligen Level definiert
- D subsumiert C subsumiert B …

RUP: Test-Rollen









Quelle: www.rational.com

Literatur – 1

- Beizer, B. (1995). Black-Box Testing. New York, etc.: John Wiley & Sons
- Frühauf, K., J. Ludewig, H. Sandmayr (1991). Software-Prüfung. Eine Fibel. Zürich vdf und Stuttgart Teubner
- Graham, D. et al. (2007), Foundations of Software Testing, Thomson Learning
- Hetzel, Wiliam, (1988), The compete guide to software testing, John Wiley & Sons
- IEEE (1987). Standard for Software Unit Testing. ANSI/IEEE Std 1008-1987. IEEE Computer Society Press
- IEEE (1998a). IEEE Standard for Software Test Documentation. ANSI/IEEE Std 829-1998. IEEE Computer Society Press
- Liggesmeyer, P. (2002). Software-Qualität: Testen, Analysieren und Verifizieren von Software. Berlin: Spektrum Akademischer Verlag
- Ludewig, J., Lichter. H. (2010), Software Engineering, zweite Auflage, dpunkt Verlag
- Myers, G.J. (1979). The Art of Software Testing. New York, etc.: John Wiley & Sons. [in dt. Übersetzung: Methodisches Testen von Programmen. 4. Auflage. Oldenbourg, München, 1991.]
- Pol, M., T. Koomen, A. Spillner (2000). Management und Optimierung des Testprozesses. Heidelberg: dpunkt.verlag
- Spillner, A., T. Linz (2002). Basiswissen Softwaretest. Heidelberg, dpunkt Verlag
- Zeller, A. (2006). Why Programs Fail: A Guide to Systematic Debugging. San Francisco: Morgan Kaufmann und Heidelberg, dpunkt Verlag

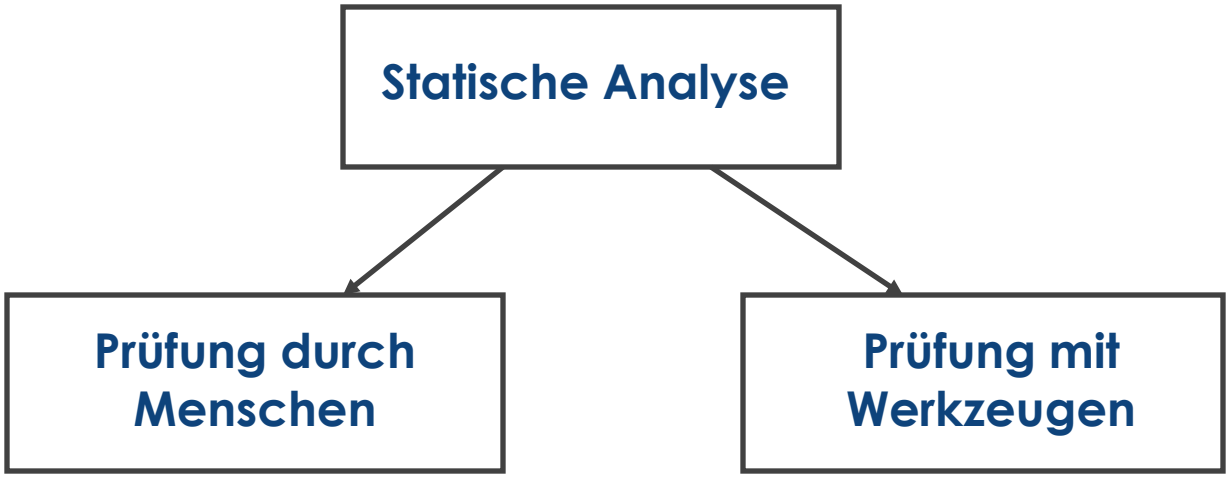
Statische Prüfungen

- Statische Prüfungen ohne Rechner
- Rechnergestützte statische Prüfungen

© 2008-2015 Dr. Rainer Schmidberger

Statische Analysen

- Statische Analyse wird oft als Obergriff für alle Prüfverfahren verwendet, bei denen keine Ausführung des Testobjekts erfolgt
- Achtung: Viele Defekte werden erst bei der Ausführung erkennbar → (dynamischer) Test



- Aufwändig, teuer
- Struktur und Semantik kann geprüft werden

- Prüfling (Dokument) muss eine formale Struktur haben
- In der Regel nur Strukturprüfungen

Strukturierte Gruppenprüfungen

- Walkthrough
  - Präsentation (oft durch den Autor) mit kritischen Zuhörern, wenig formaler Ablauf
- Inspektion
  - Formaler Ablauf, Checklisten, Gutachter, Prüfkriterien
- Technisches Review
  - Geregeltes, dokumentiertes Verfahren zur Bewertung eines Dokuments
  - Vorbereitung, Reviewsitzung, Nachbereitung
  - Rollen: Moderator, Gutachter, Protokollant, Autor
- Informelles Review
  - Abgeschwächte Form des Reviews (wenig/keine Vorbereitung, Autor übernimmt die Moderatorenrolle)

Vorteile von Reviews

- Reviews lassen sich auf alle Dokumente anwenden, wenn sie für die beteiligten Personen lesbar sind
- Mängel können frühzeitig, direkt nach ihrem Entstehen, erkannt und beseitigt werden.
- Die Prüflinge werden einem größeren Kreis an Personen bekannt

© 2008-2015 Dr. Rainer Schmidberger

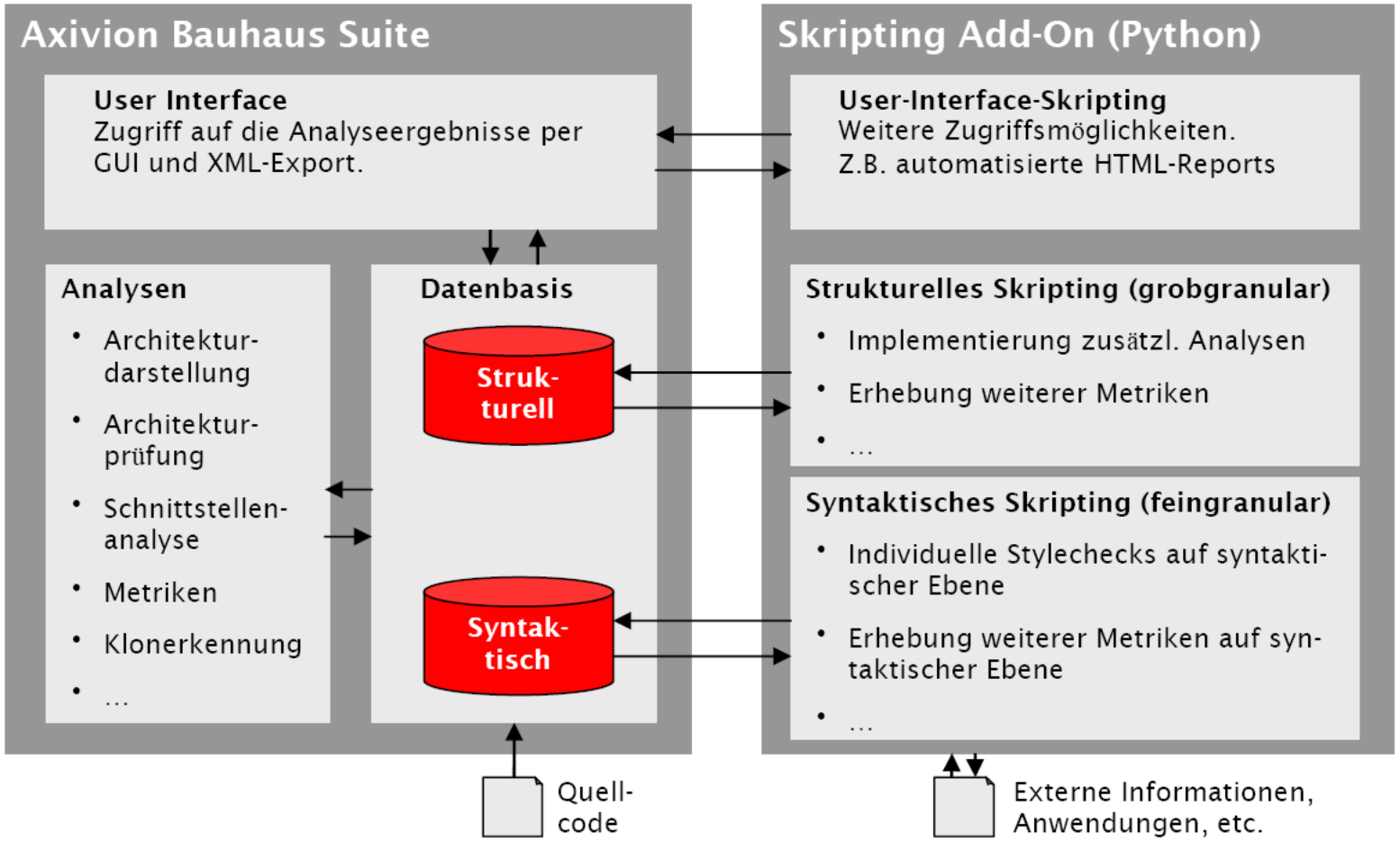
Statische (werkzeugbasierte) Analysen

- Überprüfung, Vermessung und Darstellung eines Dokuments bzw. eines Programms oder einer Komponente
- Dokumente müssen eine formale Struktur haben
- Typischerweise wird Quelltext analysiert
  - Z.B. Prüfung auf „toten“ Code oder Klone
- Prüfung auf Einhaltung von Konventionen und Standards
  - Schachtelungstiefe, Kommentierungsgrad, ...
- Datenflussanalyse
  - Anomalien z. B. ur-, du-, dd- (u: undefiniert, d: definiert, r: referenziert)
- Erhebung von Metriken

Datenflussanalyse: Anomalien

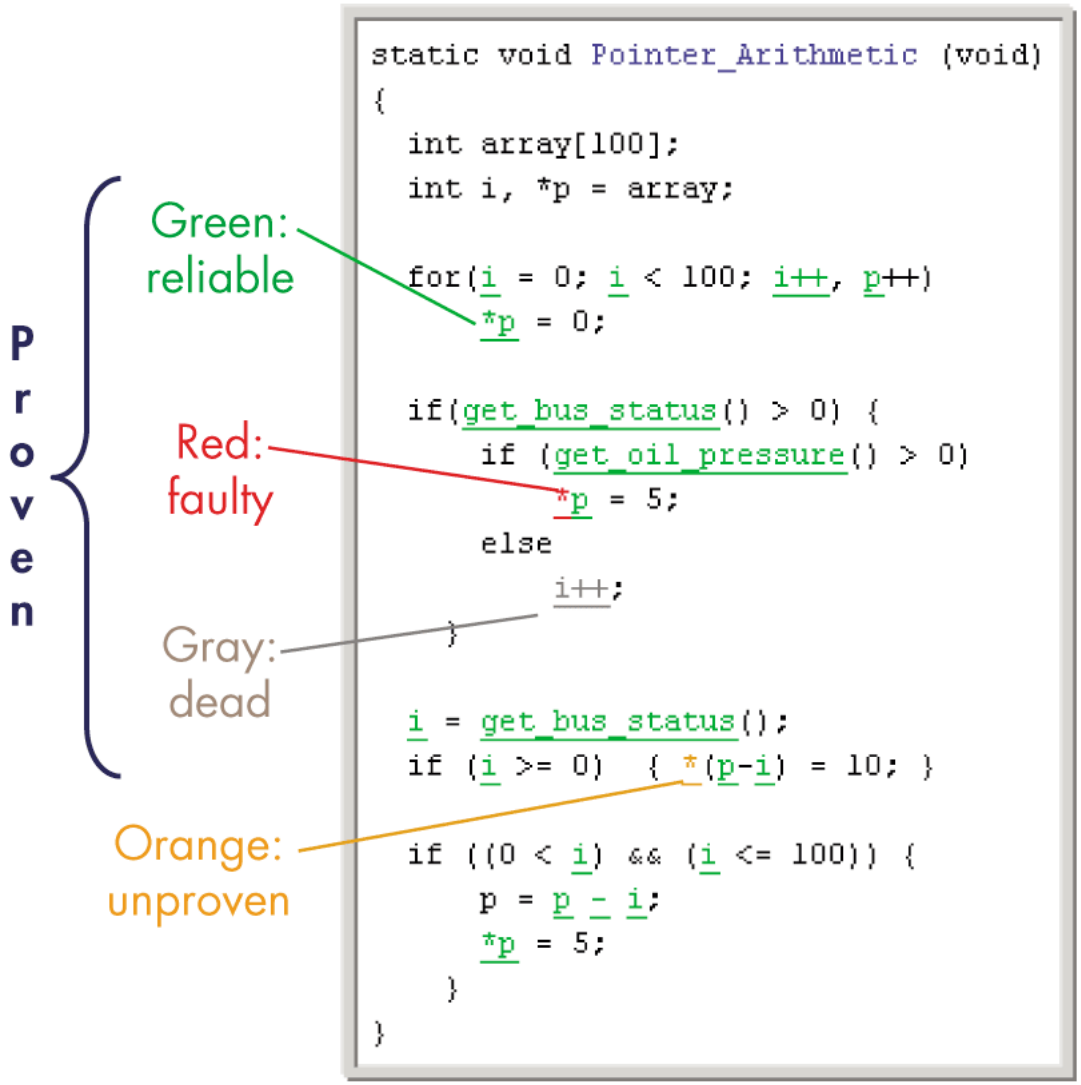
- **ur-Anomalie:** Ein undefinierter Wert (u) einer Variablen wird auf einem Programmpfad gelesen (r).
- **du-Anomalie:** Die Variable erhält einen Wert (d), der allerdings ungültig (u) wird, ohne dass er zwischenzeitlich verwendet wurde.
- **dd-Anomalie:** Die Variable erhält auf einem Programmpfad ein zweites Mal einen Wert (d), ohne dass der erste Wert (d) verwendet wurde.

Beispiel: Axivion Bauhaus Suite



Quelle: www.axivion.com

Beispiel: Polyspace



Statische Analysen:

- die Abwesenheit von Überläufen,
- Divisionen durch Null,
- unzulässigen Array-Zugriffen
- …

Dies geschieht ohne Ausführung und Instrumentierung des Codes und ohne die Erzeugung von Testfällen sondern durch eine abstrakte Interpretation mithilfe formaler Methoden.

http://www.mathworks.de/products/polyspace/

Übung
Tessy

- Produktübersicht
- Einführendes Beispiel „quadrat“
- Beispiel CoEng: Interface-Analyse, Testfall erstellen, Test ausführen, Testbericht

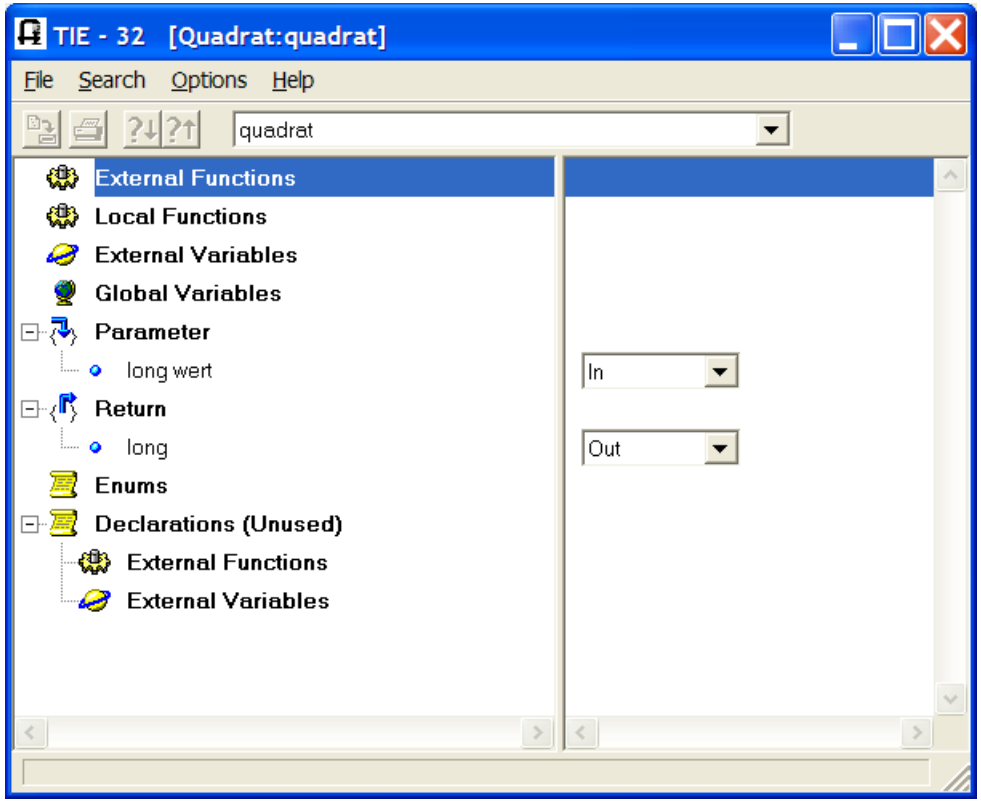
© 2008-2015 Dr. Rainer Schmidberger

Übersicht

- Tessy ist ein Unit-Test-Werkzeug für in C geschriebene Programme
- In Tessy können externe Compiler integriert werden; der gcc wird standardmäßig mitgeliefert
- Funktionen
  - Automatische Interface-Analyse von C-Funktionen
  - Erfassung und Verwaltung von Testfällen für C-Funktionen
  - Ausführung von Testfällen
  - Flexible Berichtserstellung (HTML, Word, …)
- Tessy stammt wie der CTE aus der Daimler-Forschung
- www.hitex.de

Beispiel „quadrat“ - Analyse

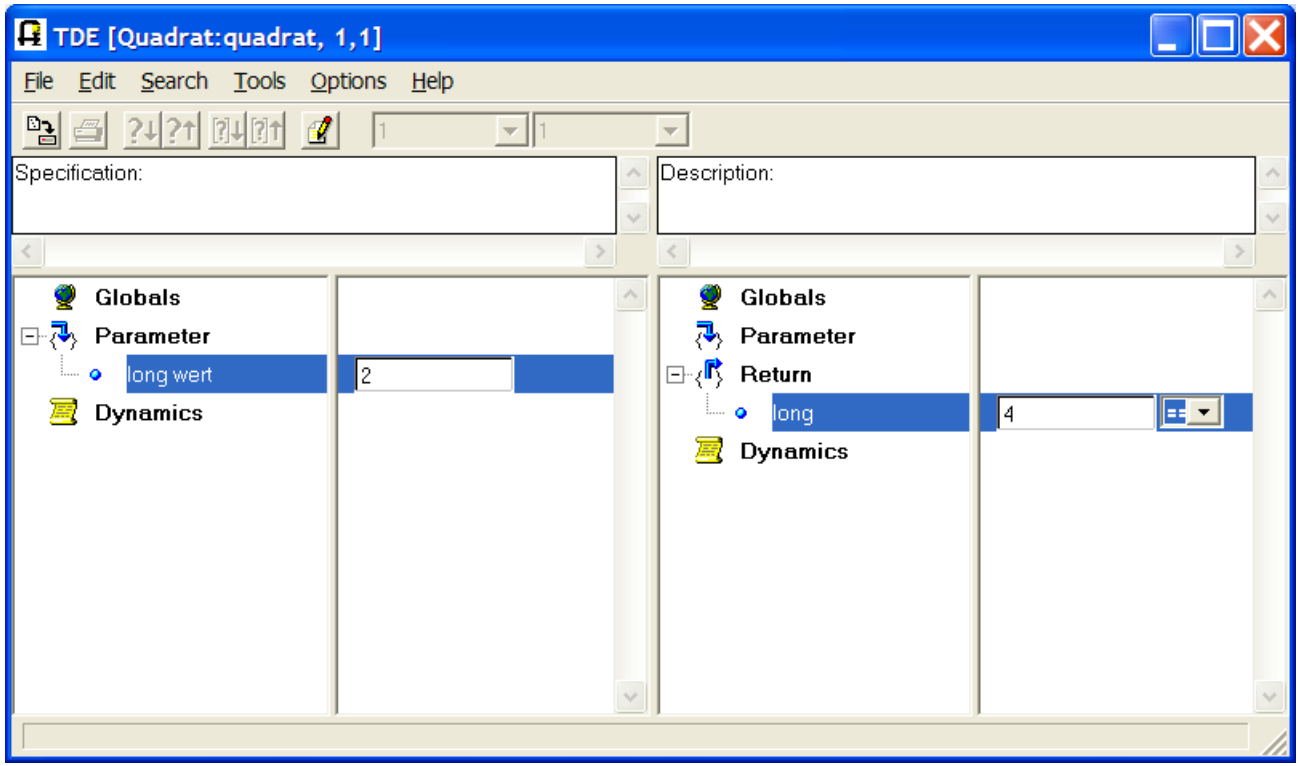
- Eine einfache Funktion „quadrat“ wird mit Tessy analysiert



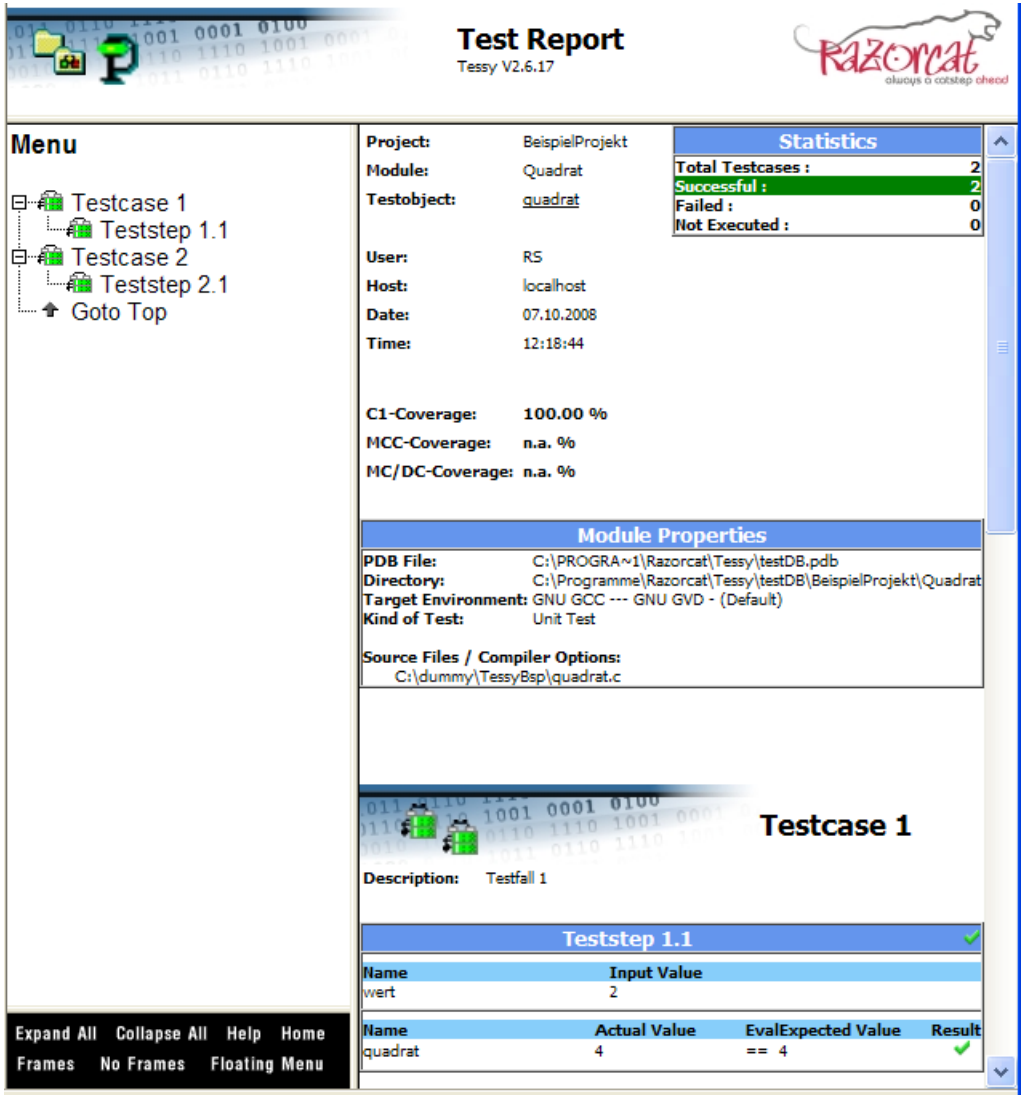

```
int quadrat(int wert)
{
  return wert * wert;
}
```

Beispiel „quadrat“ - Testfall

- Eingabewerte werden spezifiziert
- Für Ausgabewerte wird ein Soll-Resultat spezifiziert

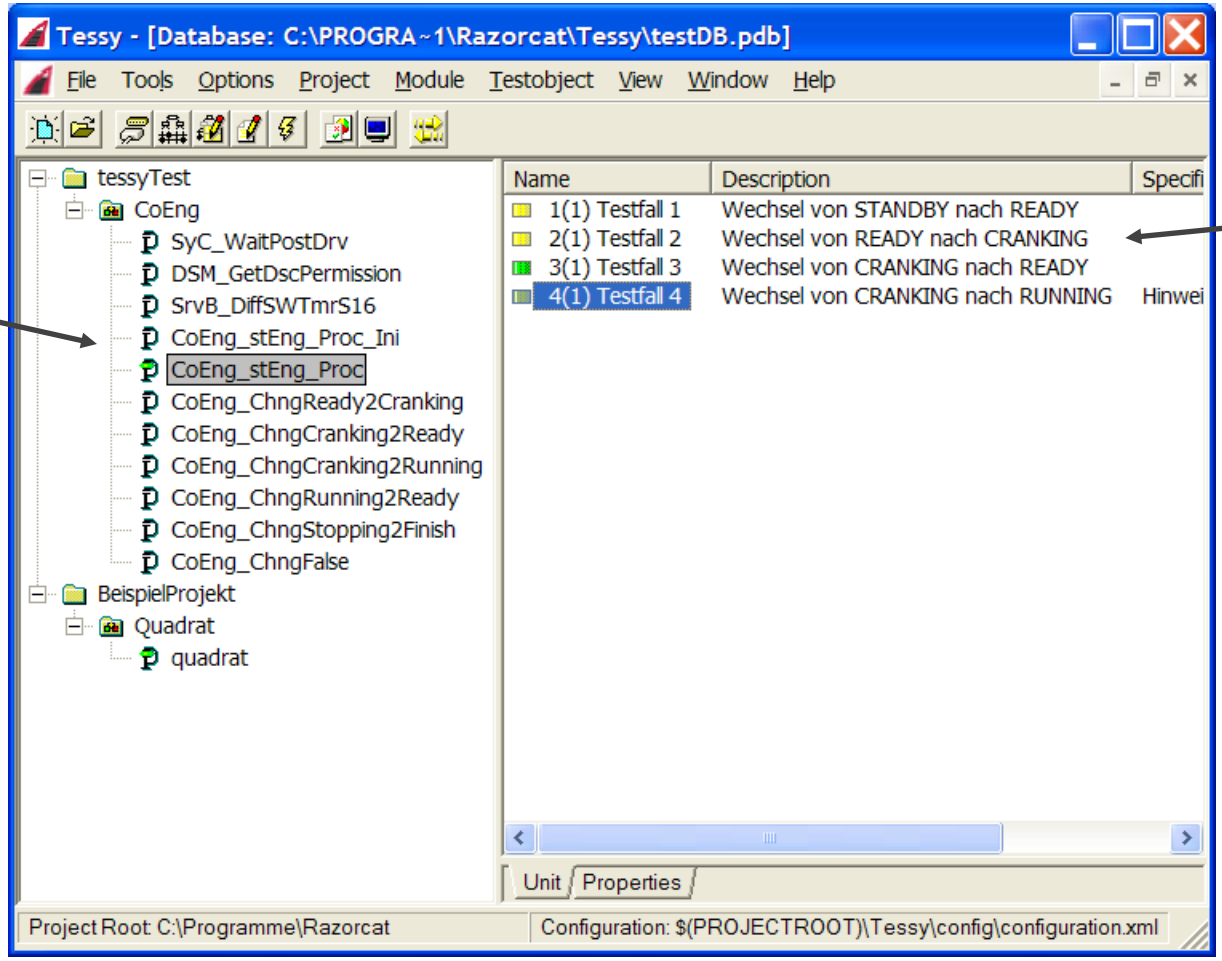


Beispiel „quadrat“ - Reportserstellung



- Formate
  - HTML
  - Word
  - Excel
- Inhalt
  - Standardinhalte
  - Konfigurations-möglichkeit über Python Skripte

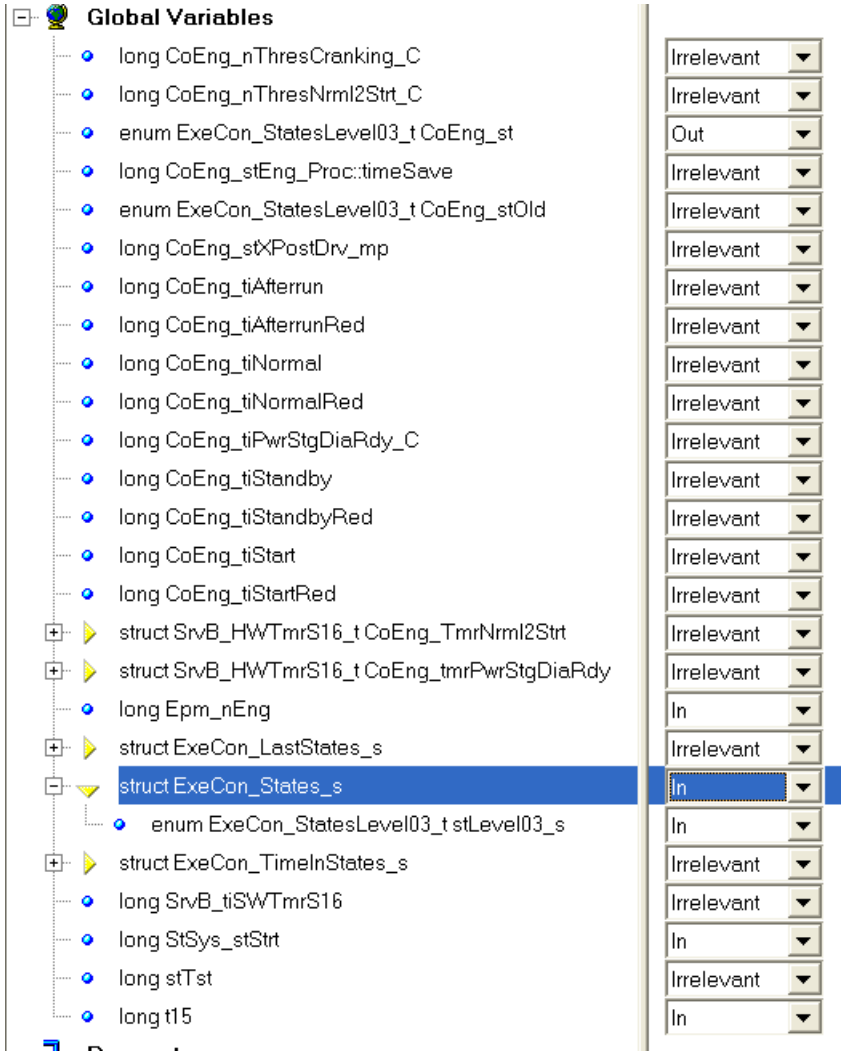
Beispiel CoEng: Projektanzeige



Testobjekte

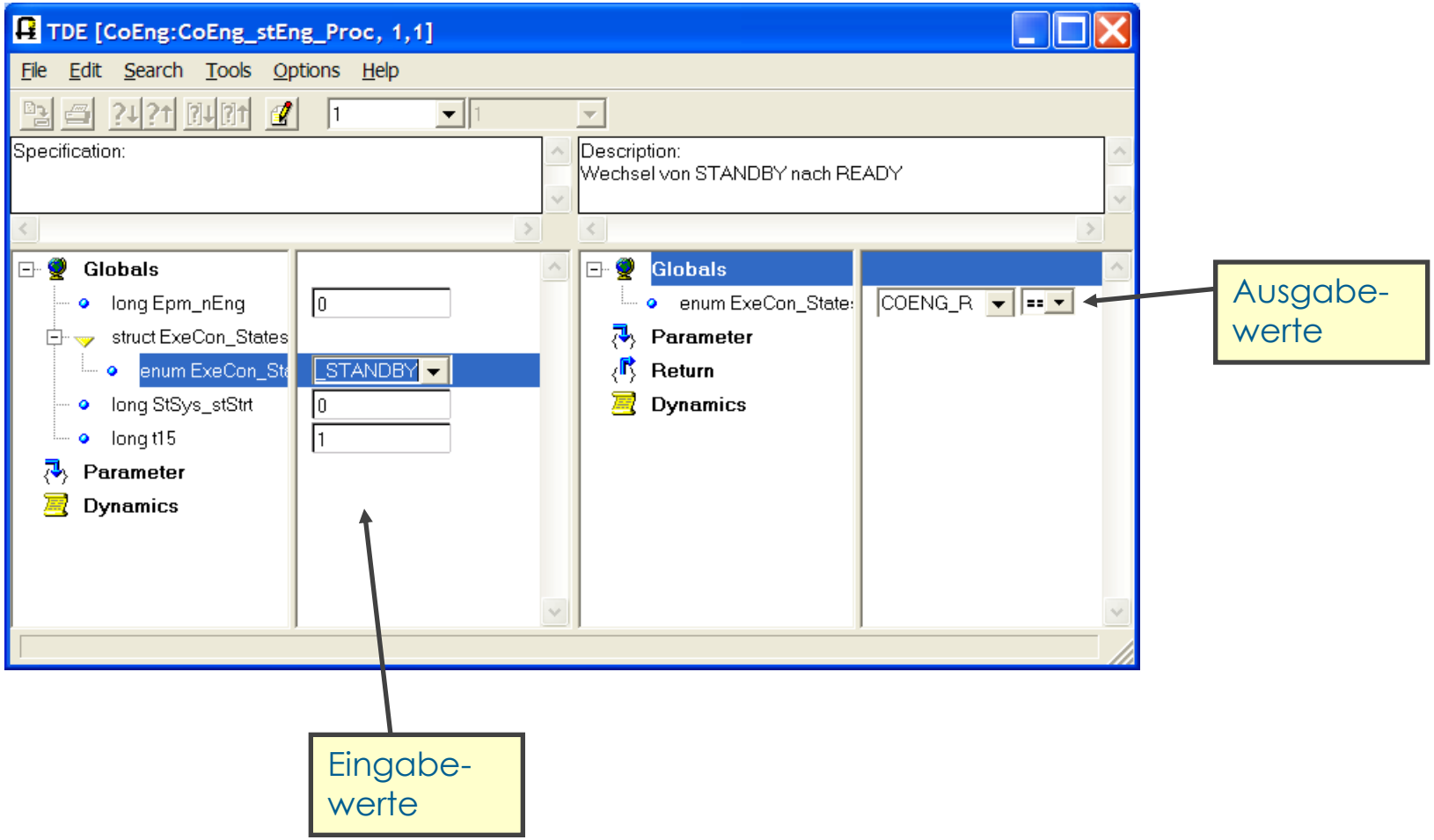
Testfälle

Beispiel CoEng: Interface-Analyse

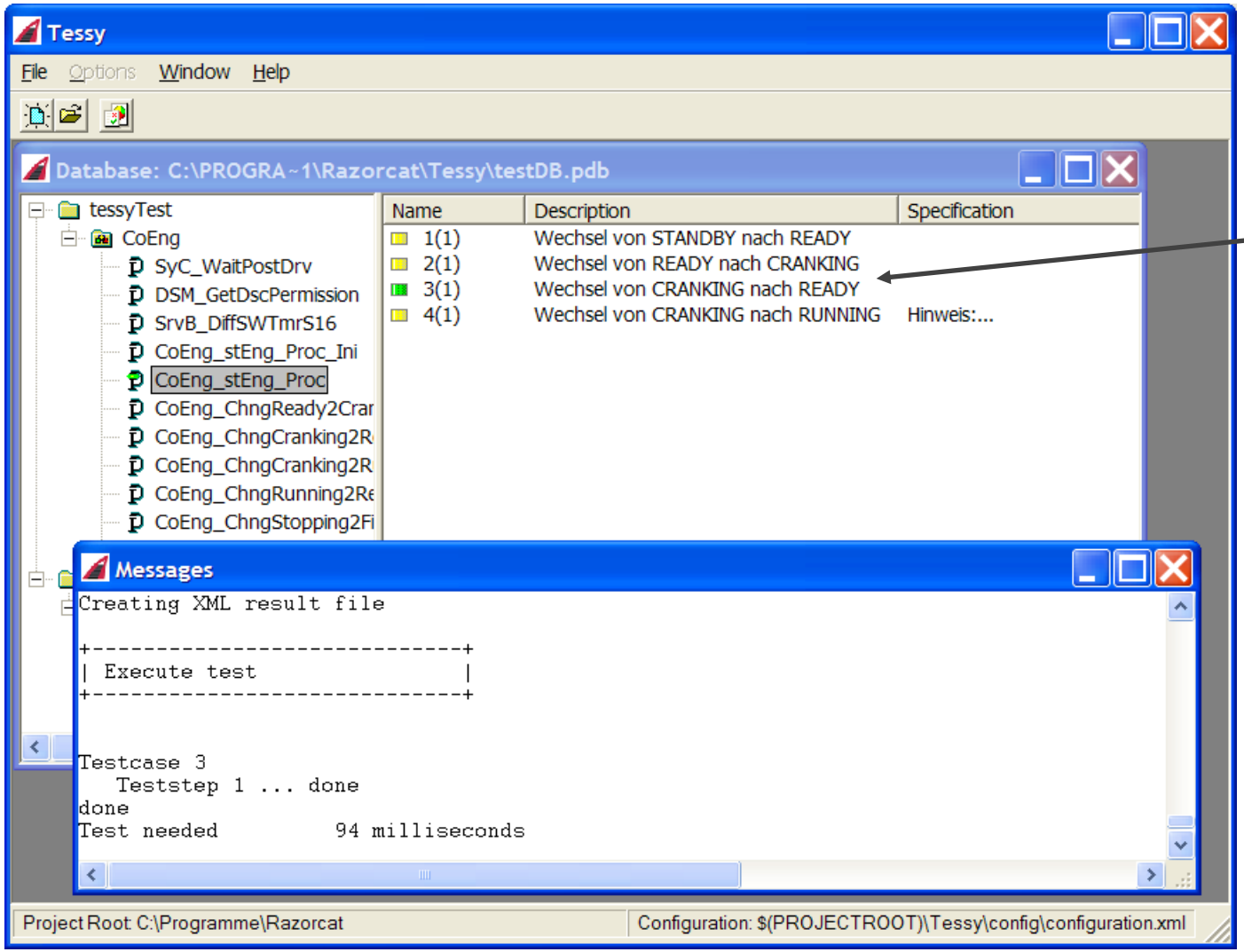


- Je Testobjekt (=C-Funktion) wird das Interface analysiert
  - Parameterübergaben
  - Rückgaben
  - Globale Variablen: Die Auswirkung auf den Test (in, out, in/out, irrelevant, Extern) kann je einzeln spezifiziert werden

Beispiel CoEng: Testfall erfassen

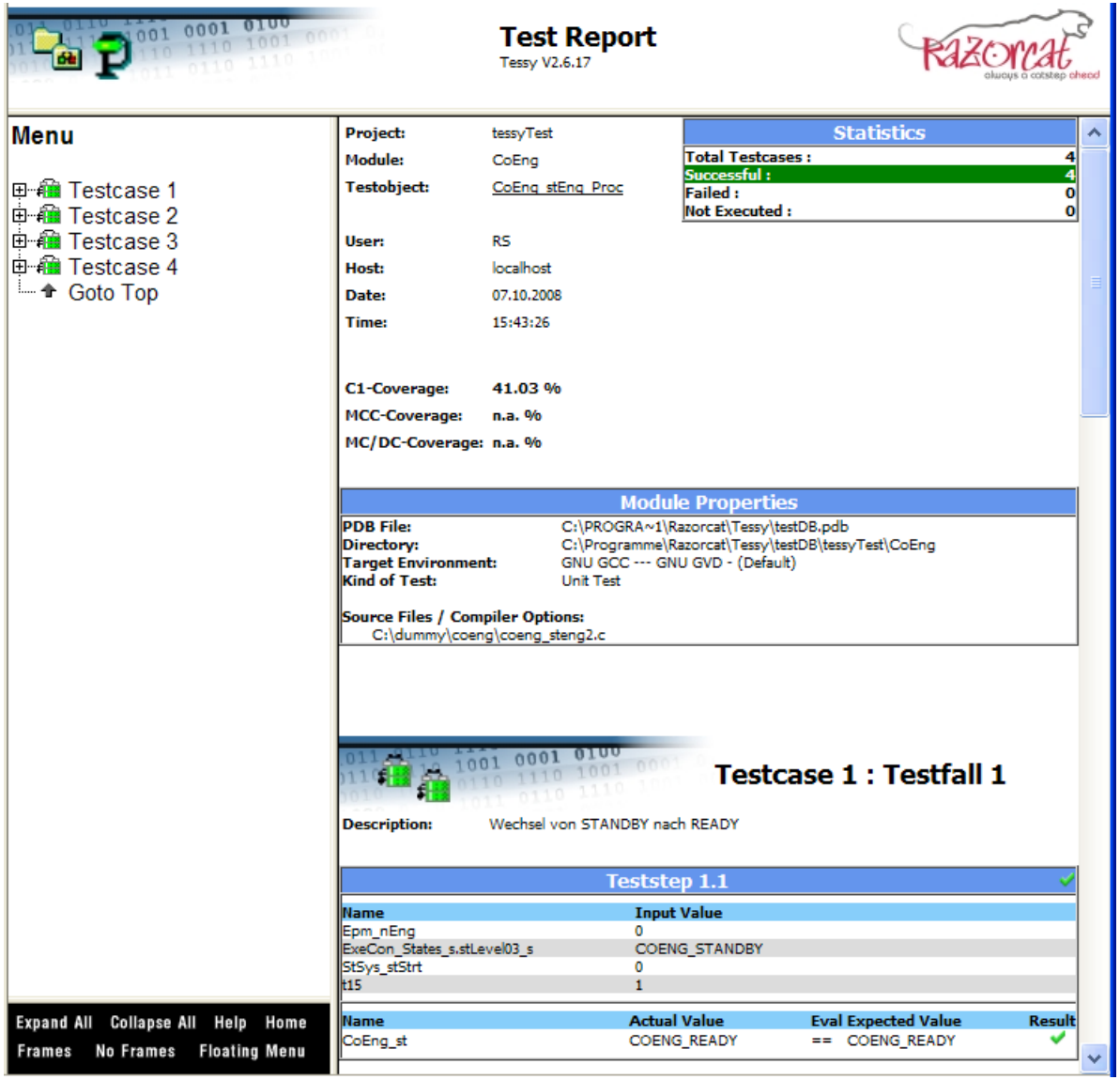


Beispiel CoEng: Testfall ausführen



Testfall 3 wurde einzeln ausgeführt (grün), die anderen Testfälle sind noch nicht ausgeführt (gelb)

Beispiel CoEng: Testbericht als HTML



Test Report
Tessy V2.6.17

Menu

- Testcase 1
- Testcase 2
- Testcase 3
- Testcase 4
- Goto Top

Expand All Collapse All Help Home
Frames No Frames Floating Menu

Project:	tessyTest
Module:	CoEng
Testobject:	CoEng_stEng_Proc
User:	RS
Host:	localhost
Date:	07.10.2008
Time:	15:43:26
C1-Coverage:	41.03 %
MCC-Coverage:	n.a. %
MC/DC-Coverage:	n.a. %

Statistics

Total Testcases :	4
Successful :	4
Failed :	0
Not Executed :	0

Module Properties

PDB File:	C:\PROGRA~1\Razorcat\Tessy\testDB.pdb
Directory:	C:\Programme\Razorcat\Tessy\testDB\tessyTest\CoEng
Target Environment:	GNU GCC --- GNU GVD - (Default)
Kind of Test:	Unit Test

Source Files / Compiler Options:
C:\dummy\coeng\coeng_steng2.c

Testcase 1 : Testfall 1

Description: Wechsel von STANDBY nach READY

Teststep 1.1

Name	Input Value
Epm_nEng	0
ExeCon_States_s.stLevel03_s	COENG_STANDBY
StSys_stStrt	0
t15	1

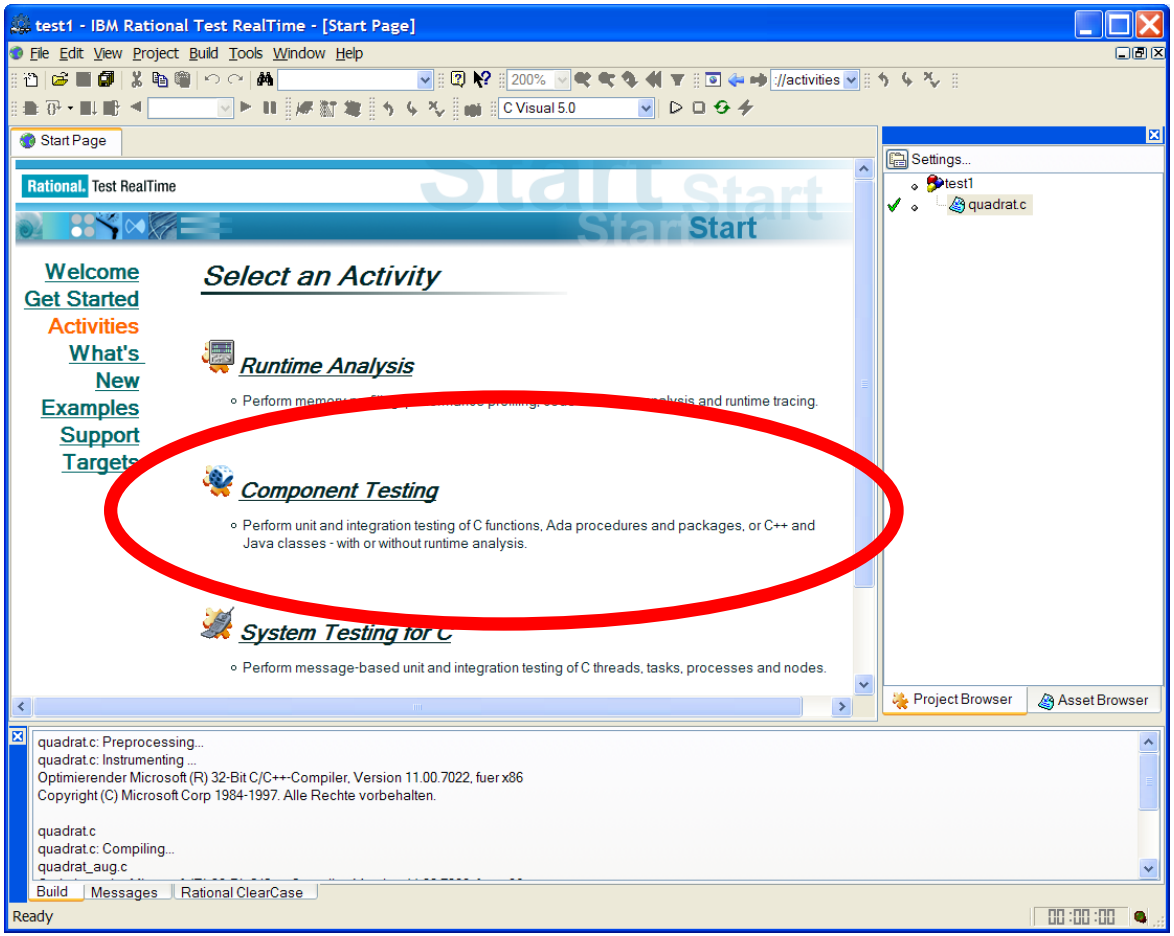
Name	Actual Value	Eval Expected Value	Result
CoEng_st	COENG_READY	== COENG_READY	✔

Übung
Rational Test RealTime

- Produktübersicht, Komponenten-Test
- Komponenten-Test erstellen
- Skriptsprache
- Ausgaben

© 2008-2015 Dr. Rainer Schmidberger

Component Testing



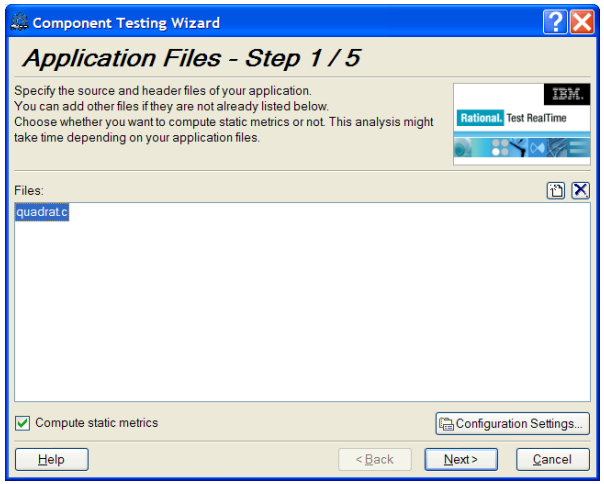
- Einzelne C-Funktionen können getestet werden
- Übergabewerte und Rückgabe-werte sowie globale Variablen können gesetzt und evaluiert werden

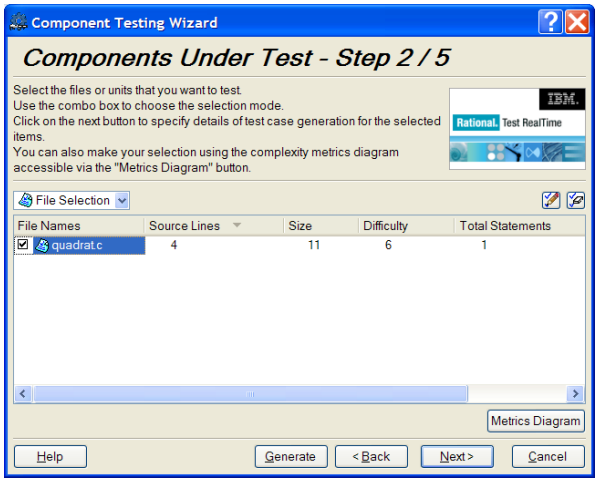
Übersicht

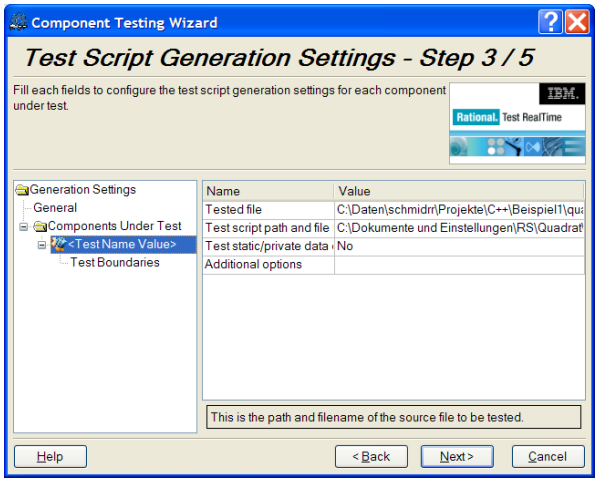
- Die Verbindung zu Compiler und Linker wird über sogenannte „Target Deployment Ports“ eingerichtet
  - Gängige Entwicklungsumgebungen sind bereits vorbereitet (z.B. MS Visual C++)
- Für UTE17 gibt es bei DS/ESQ1 bereits viel Erfahrung
  - Eine Ausführliche Dokumentation ist hierzu verfügbar; zudem umfangreiche Standardisierungen und Tipps
- Komponenten-Test
  - Einfache Skriptsprache zur Erstellung der Testfälle
  - Umfangreiches Reporting
  - Glassbox-Test-Metriken

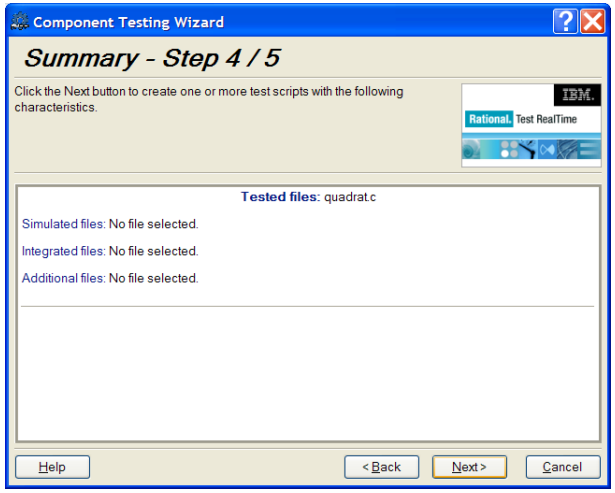
Komponenten-Test

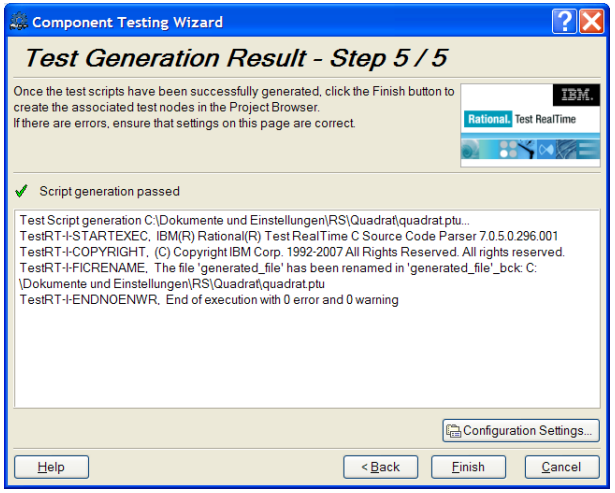
- Assistent zur Generierung eines Testskripts











Testskript – Struktur

Beispiel.ptu

```
HEADER Testskript, 28.08.2008, Version 1.0
<Variablendeklarationen für das Test Script>
BEGIN

SERVICE Testskript
  <Lokale Variablendeklarationen>
  # variable1, variable2, variable3: integer;
TEST Testfall1
  FAMILY nominal
  ELEMENT
    VAR variable1, INIT=0, EV=0
    VAR variable2, INIT==, EV=0
    VAR variable3, INIT=1, EV=INIT
    #<call to the procedure under test>
  END ELEMENT
  END TEST

END SERVICE
```

HEADER: Testskriptname und Versionsangabe. Für Dokumentationszwecke, Angaben erscheinen im Testbericht.

SERVICE: Enthält die Testfälle, üblicherweise für eine C-Funktion

TEST: Beschreibt den Testfall, hat einen eindeutigen Bezeichner

FAMILY: Testfall-Typ. Liste kann frei festgelegt werden. Beispiele: nominal, structure

ELEMENT: Umfasst einen Testfall. Kann durch **NEXT_TEST** erweitert werden

Testskript – Funktionsweise

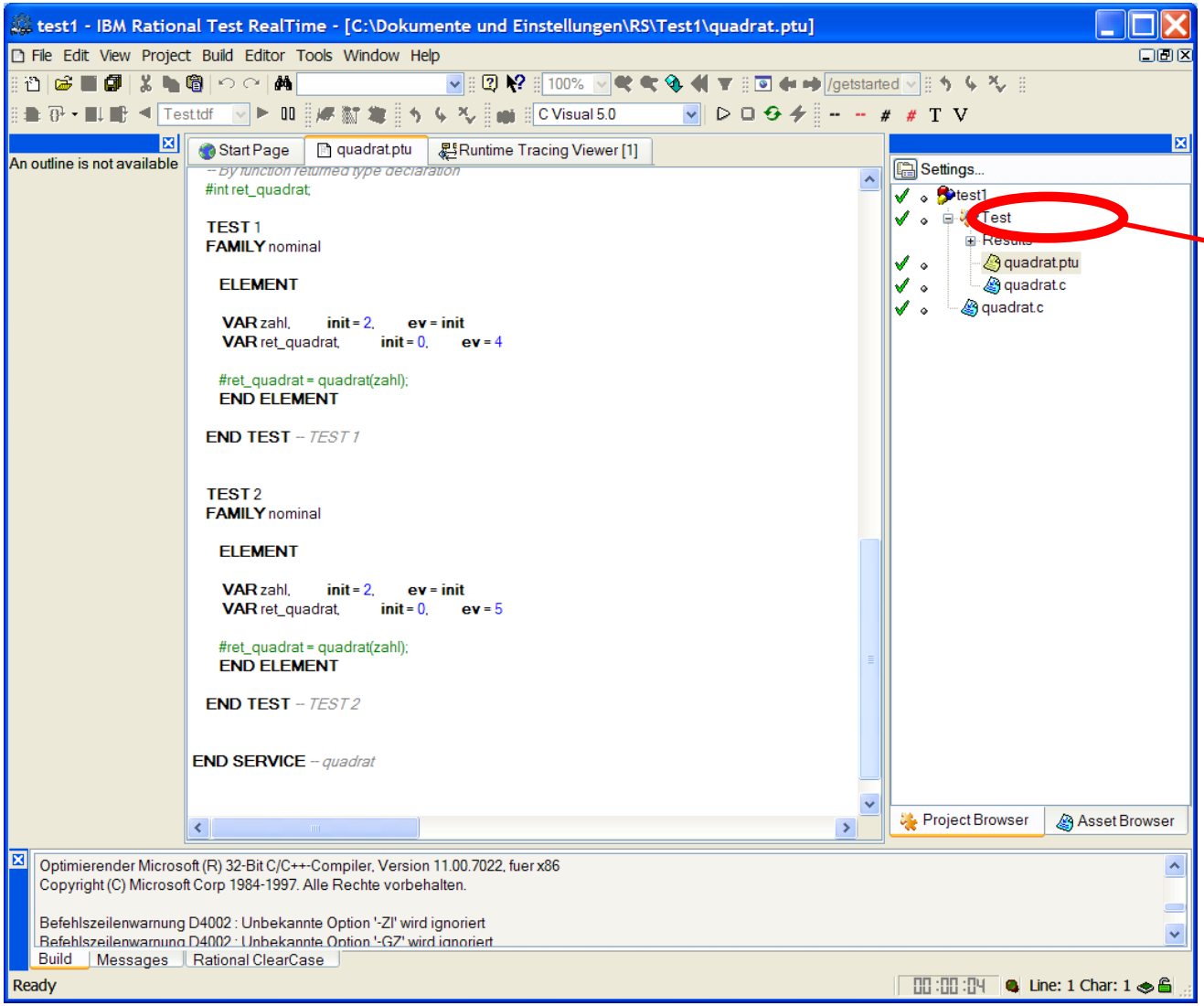
Testskript (.ptu)

```
TEST 1
FAMILY nominal

 ELEMENT
  var a, init = 0 ,   ev  =  1
  var b, init=2, ev=2
  var ergebnis, init=0, ev=4

  #ergebnis=testFunktion(b);

 END ELEMENT

END TEST -- TEST 1
```

Testobjekt: C-Funktion

```
/* extern */ int a;

int testFunktion(int b)
{
  /* nur ein Beispiel */
  if(b == 2)
  {
    a = 1;
  }
  return b * b;
}
```

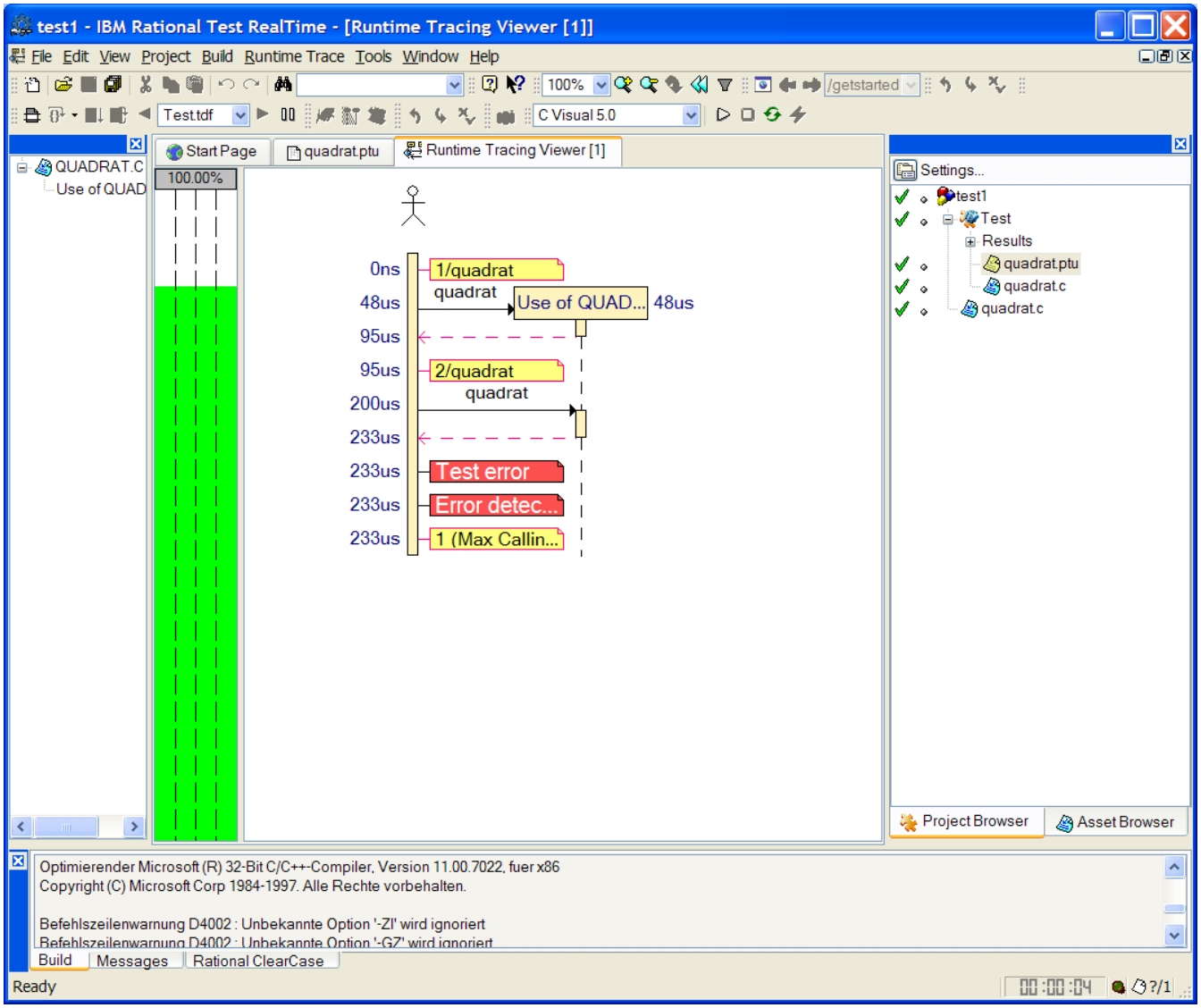
Testskript – Editor



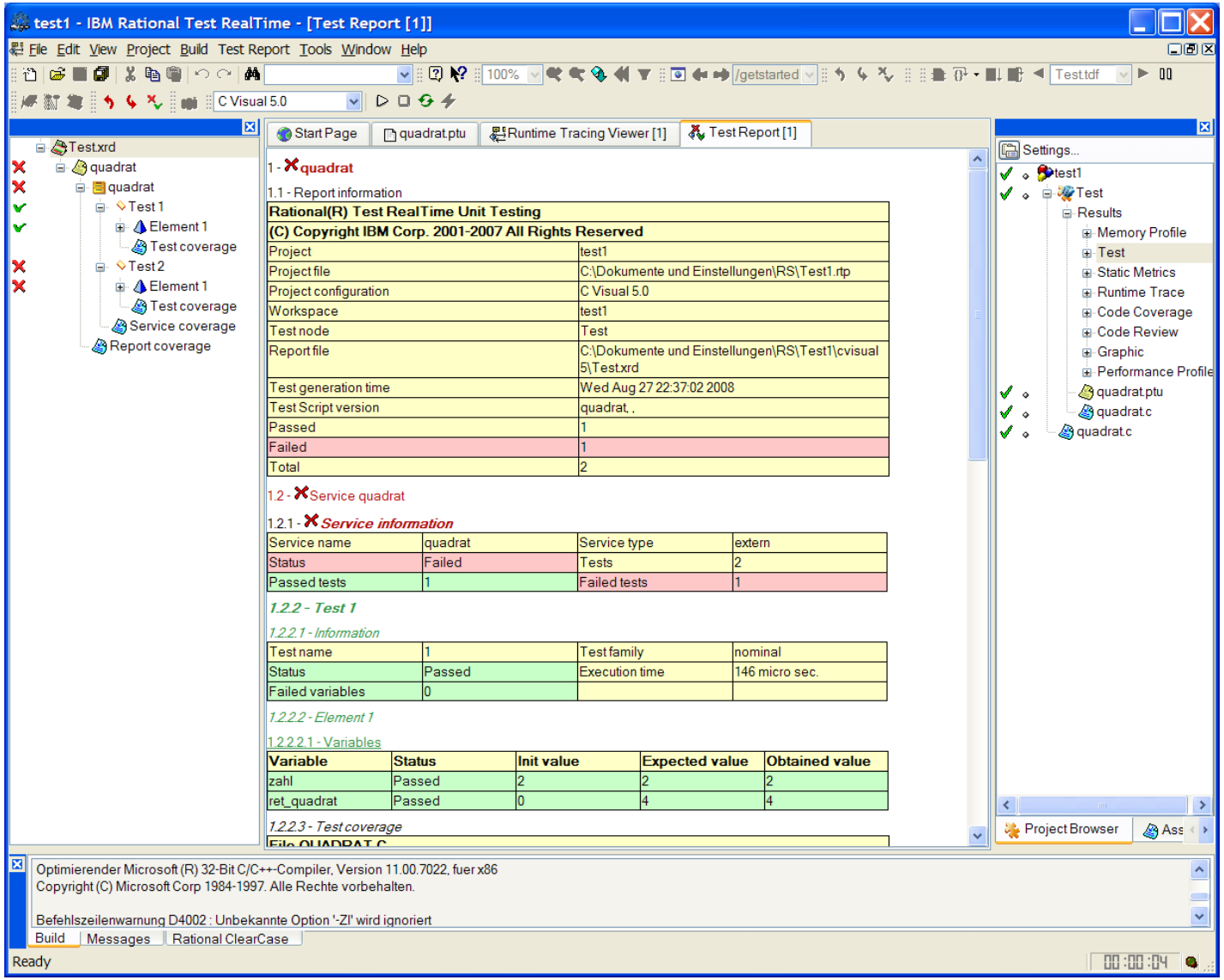
Rechte Maustaste:

Kompilieren und Test ausführen

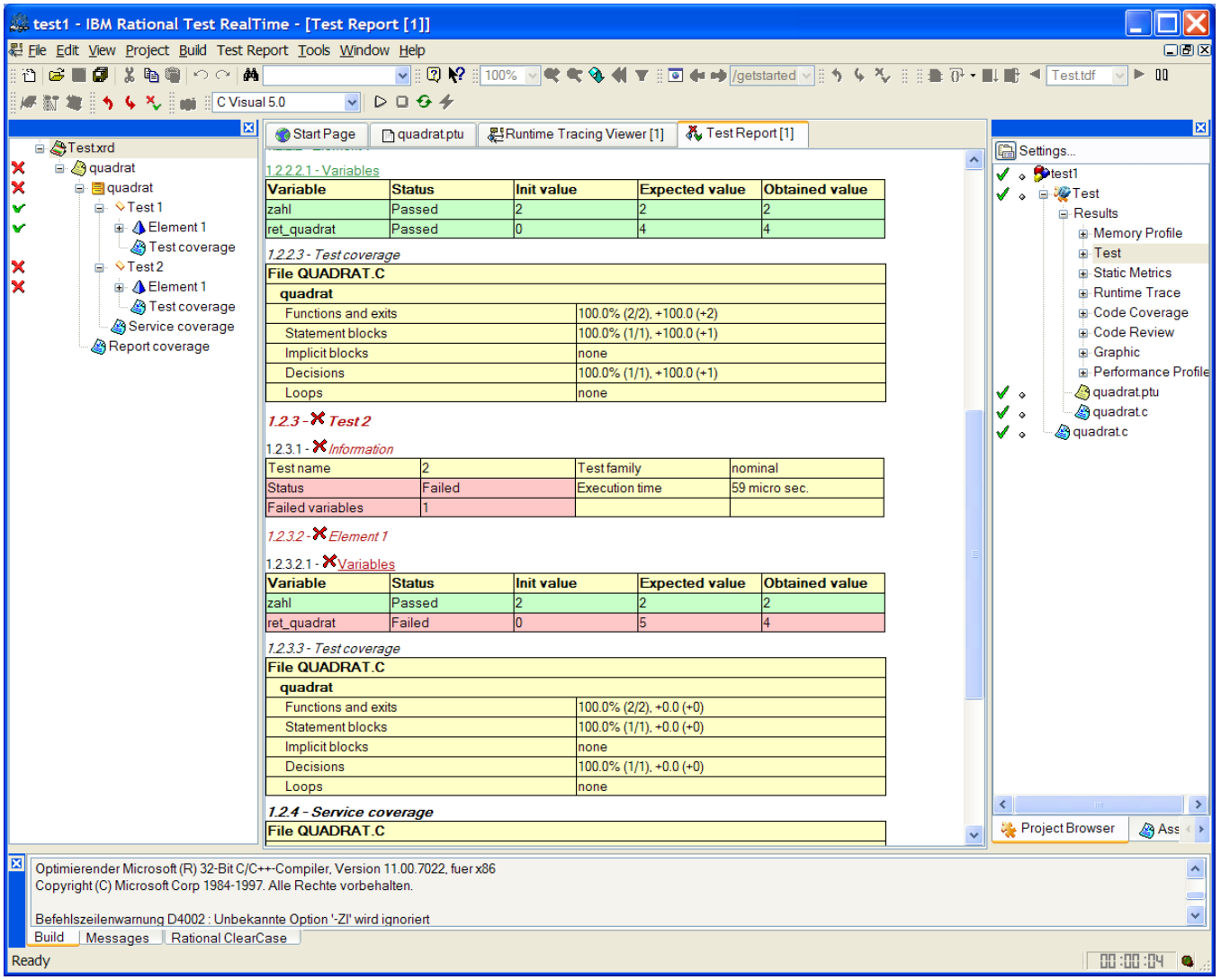
Ausgaben: Runtime Tracing Viewer



Ausgaben: Test Report – 1



Ausgaben: Test Report – 2



1.2.2.2.1 - Variables

Variable	Status	Init value	Expected value	Obtained value
zahl	Passed	2	2	2
ret_quadrat	Passed	0	4	4

1.2.2.3 - Test coverage

File QUADRAT.C	
quadrat	
Functions and exits	100.0% (2/2), +100.0 (+2)
Statement blocks	100.0% (1/1), +100.0 (+1)
Implicit blocks	none
Decisions	100.0% (1/1), +100.0 (+1)
Loops	none

1.2.3 - Test 2

1.2.3.1 - *Information*

Test name	2	Test family	nominal
Status	Failed	Execution time	59 micro sec.
Failed variables	1		

1.2.3.2 - Element 1

1.2.3.2.1 - Variables

Variable	Status	Init value	Expected value	Obtained value
zahl	Passed	2	2	2
ret_quadrat	Failed	0	5	4

1.2.3.3 - Test coverage

File QUADRAT.C	
quadrat	
Functions and exits	100.0% (2/2), +0.0 (+0)
Statement blocks	100.0% (1/1), +0.0 (+0)
Implicit blocks	none
Decisions	100.0% (1/1), +0.0 (+0)
Loops	none

1.2.4 - Service coverage

File QUADRAT.C