

Inhaltsverzeichnis

I	Warum überhaupt testen?	1
1	Komplexe Systeme führen zu Fehlern	3
1.1	Kommunikation	3
1.2	Gedächtnis	6
1.3	Fachlichkeit	6
1.4	Komplexität	8
1.5	Erstes Fazit	8
2	Programmiersprachen sind fehleranfällig	9
2.1	Die Venussonde Mariner 1	9
2.2	Der Jungfernflug der Ariane 5	10
2.3	Zweites Fazit	12
2.3.1	Überschätzte Typisierung	13
2.3.2	Was hätte geholfen?	15
3	Qualität, Fehler, Test: Begriffsbildung	17
3.1	Qualität	17
3.1.1	Aspekte von Softwarequalität	18
3.1.2	Quality in use – der Nutzen im Fokus	20
3.2	Anforderung	22
3.3	Fehler oder Mangel?	23
3.3.1	Fehlhandlung, Fehlerzustand und Fehlerwirkung	23
3.3.2	Fehler und Mangel unterscheiden	24
3.4	Test	25
3.4.1	Demonstratives und destruktives Testen	26
3.4.2	Whitebox- und Blackbox-Testverfahren	26
3.4.3	Testfall und Testdaten	27
3.4.4	Unit Tests: Klassen-, Ketten- und Modultests	28
3.4.5	Regressionstests	30
3.4.6	Debugging	31
4	Was schließen wir daraus?	33
4.1	Tests automatisieren	33
4.2	Modellbasiertes Testen	34

II	Verfahren des Softwaretests	37
5	Lösungen für technische Probleme	39
5.1	Unterstützung durch den Compiler	39
5.1.1	Warning-Level	39
5.1.2	Programmierrichtlinien	39
5.2	Selbstdokumentierender Coding Style Guide	40
5.2.1	Allgemeines zum Thema Layout	40
5.2.2	Layout von Kontrollstrukturen	43
5.2.3	Selbstdokumentierender Code	45
5.2.4	Kommentare und selbstdokumentierender Code	46
5.3	Programmierregeln	48
5.3.1	Sprachunabhängige Regeln	48
5.3.2	Sprachabhängige Regeln	49
5.3.3	Was nützen statische strenge Typprüfungen?	49
5.4	Debugging	50
5.4.1	Einplanung der Fehlersuche in Produkt und Prozess	50
5.4.2	Vorbereitung und Ausführung des Debuggings	51
5.4.3	Der Debugging-Vorgang	52
6	Lösungen für analytische Probleme	57
6.1	Scope: Was will ich testen?	57
6.2	Fachliche Testfälle finden	58
6.2.1	Testdaten ableiten	61
6.2.2	Unit-Testfälle ableiten	62
6.2.3	Kettentests ableiten	62
6.2.4	Systemtestfälle ableiten	63
6.3	Risikoorientiertes Testen	63
6.3.1	Risiken bewerten	63
6.3.2	Mit Risiken umgehen	65
6.3.3	Risiken finden	67
6.3.4	Produkt- und Projektrisiken	68
7	Lösungen für methodische Probleme	69
7.1	Psychologie des Testens	69
7.2	Codereviews	71
7.2.1	Interne Codereviews	72
7.2.2	Externe Codereviews	73
7.2.3	Dokumentreviews	73
7.3	Die richtigen Testdaten finden	74
7.3.1	Grenz- und Extremwerte	74
7.3.2	Testdaten als Designkriterium	78
7.3.3	Fehlersensibilität	78

7.3.4	Äquivalenzklassen	79
7.4	Überdeckungen: Wege durch die Kombinatorik	83
7.4.1	Anweisungs-, Zweig- und Pfadüberdeckung	83
7.4.2	Vereinfachte Schleifenüberdeckung	86
7.4.3	Test von Bedingungen: die Termüberdeckung	87
7.5	Erfahrung: Error Guessing und laterale Tests	89
8	Lösungen für fortgeschrittene Probleme	91
8.1	Zustandsraumbasiertes Testen	91
8.1.1	Zustandsübergänge modellieren	91
8.1.2	Zustandsmodelle testen	93
8.2	Rekursive und iterative Algorithmen	96
8.3	Parallele Prozesse	99
8.3.1	Prozesskommunikation und -synchronisation	99
8.3.2	Race Conditions	102
8.3.3	Parallelität auf Ein-Prozessor-Maschinen	105
9	Test objektorientierter Software	109
9.1	Testreihenfolge in objektorientierter Software	110
9.1.1	Assoziationen	111
9.1.2	Vererbung	112
9.1.3	Verflechtung von Assoziationen und Vererbung	115
9.1.4	Testreihenfolgen für Methoden	115
9.2	Vererbung, das zweischneidige Schwert	117
9.2.1	Das Rechteck-Quadrat-Drama	117
9.2.2	Sind dauerhafte Lösungswege möglich?	118
9.3	Prinzipien zur objektorientierten Vererbung	120
9.3.1	Das Substitutionsprinzip nach Liskov	120
9.3.2	Das Zusicherungs-Verantwortlichkeitsprinzip	121
9.3.3	Clean Code – wartbar und fehlerarm	122
9.3.4	Flattening: Welche Methoden sind zu testen?	122
9.3.5	Zufällige Korrektheit durch Vererbung	123
9.3.6	Typische Fehler in Vererbungshierarchien	125
9.3.7	Teststrategie bei Vererbung	127
9.4	Testmuster: Tipps für die Praxis	129
9.4.1	Modale Klasse	129
9.4.2	Modale Hierarchie	131
9.4.3	Nicht modaler, polymorpher Server	134
9.5	Struktur von objektorientierten Programmen	136
9.5.1	Test von Mehrschicht-Architekturen	137
9.5.2	Test von SOA-Services	140
9.6	Zusammenfassung	140

10	Test von Realtime und Embedded Systems	141
10.1	Was bedeutet eigentlich RTES?	141
10.2	Was ist ein sicheres System?	144
10.3	Warum sind RTES so besonders schwierig?	145
10.3.1	Reaktives System	145
10.3.2	Nebenläufigkeit und Verteilung	145
10.4	Besondere Testverfahren	146
10.4.1	Failure Mode and Effect Analysis – FMEA	147
10.4.2	Fault Tree Analysis – FTA	148
10.4.3	Classification Tree Method – CTM	149
10.4.4	Testbare und robuste Architektur: Watchdog-Pattern	153
10.4.5	Vom Watchdog- zum Safety-Executive-Pattern	154
10.5	Gemischte Signale und Timing-Diagramme	155
10.5.1	Timing-Informationen	156
10.5.2	Timing-Diagramme	157
10.6	Entwicklungs- und Testumgebungen	161
10.6.1	Stufenweises Vorgehen	162
10.6.2	In-the-Loop: Regressionstests	163
10.6.3	Besonderheiten von RTES-Entwicklungsumgebungen	165

III Umsetzungsstrategien **167**

11	Lösungen für organisatorische Probleme	169
11.1	Vorgehensweisen in Projekten	169
11.1.1	Das Wasserfallmodell	170
11.1.2	Das V-Modell – Erweiterung des Wasserfalls	172
11.1.3	Inkrementell-iteratives Vorgehen	173
11.2	Iteratives Vorgehen und Agilität	177
11.2.1	Planen von inkrementell-iterativen Prozessen	177
11.2.2	Scrum	181
11.2.3	eXtreme Programming	183
11.2.4	Innen inkrementell-iterativ, außen V-Modell?	186
11.3	Testgetriebenes Design	187
11.3.1	Konkretes Vorgehen in der Entwicklung	189
11.3.2	Testen von Objektketten	190
11.4	Reviews und Retrospektiven	190
11.4.1	Der formale Reviewprozess	191
11.4.2	Der pragmatische Reviewprozess	194
11.4.3	Retrospektiven durchführen	194
11.5	Refactoring	195
11.5.1	Was ist Refactoring?	195
11.5.2	Wie funktioniert Refactoring?	197

11.5.3	Testkoordination	199
11.6	Aufwandsbetrachtungen	200
11.6.1	Schätzungen	202
11.6.2	Fehlerkorrekturen und Nachtests	203
11.7	Fehlermodelle zur Aufwandsschätzung	204
11.7.1	Thesen und empirische Werte	204
11.7.2	Ein Rechenmodell im Detail	205
11.7.3	Rechenmodelle für die Entwicklung	207
11.8	Testverwaltung	208
12	Strategien für die Testautomatisierung	211
12.1	Testfallfindung vs. Testautomatisierung	211
12.1.1	Testautomatisierung und testgetriebenes Vorgehen	211
12.1.2	Anforderungen an die Testautomatisierung	212
12.1.3	Das Konzept hinter xUnit	214
12.2	Entwicklertests mit JUnit	216
12.2.1	Aus der Entwicklungsumgebung heraus testen	216
12.2.2	Testklassen realisieren	218
12.2.3	Exceptions testen	222
12.2.4	Datenbanken im Test ansprechen	223
12.3	Design for Testability	225
12.3.1	Wenn public zu wenig ist!	225
12.3.2	Reihenfolgen im Design und Test	226
12.3.3	Kohäsion und Kopplung	227
12.4	Stellvertreter: Stub, Dummy und Mock	228
12.5	Ein komplexes JUnit-Beispiel	229
12.5.1	Die Modelle als Ausgangspunkt	230
12.5.2	Die Testmethodik anwenden	233
12.5.3	Der konkrete JUnit 3-Test	234
12.6	... und was ist mit C++ ?	236
12.6.1	CppUnit	237
12.6.2	.NET – C++ und NUnit	239
12.6.3	Die Boost Library	241
12.7	Die C++ Boost Test Library	241
12.7.1	Unit Test Framework	242
12.7.2	Testtools	248
12.8	Testautomatisierung über die GUI	251
12.8.1	Grundregeln für Oberflächentests	254
12.8.2	Aufbau von Testskripten	255
12.9	Weitere Aspekte einer Testautomatisierung	258
12.9.1	Stresstest-Automatisierung	258
12.9.2	Test von Mehrschicht-Anwendungen	259
12.9.3	Fehlerinjektion: Wie gut sind unsere Tests?	260

12.9.4	Ausblick: Mehrwert automatisierter Tests	261
12.9.5	Vielfalt nutzen	262
13	Was bringen uns die Verfahren?	263
13.1	Kriterien für erfolgreiche Projekte	263
13.2	Anforderungen an das Entwicklungsteam	265
13.3	Anforderungen an den Projektleiter	266
14	Teststrategie: Der Weg ist das Ziel	269
14.1	Strategien umsetzen	269
14.2	Inhalte einer pragmatischen Teststrategie	272
14.3	Fehlerkultur	275
14.3.1	Konstruktive Fehlerkultur: Aus Fehlern lernen	276
14.3.2	Fehlerkultur und Kreativität	277
14.3.3	Beurteilung und Konsequenzen von Fehlern	278
IV	Modellbasiertes Testen	281
15	Was ist modellbasiertes Testen	283
15.1	Modellbasierte Blackbox-Tests	283
15.1.1	Dynamik über Aktivitätsdiagramme modellieren	284
15.1.2	Statik – Fachklassen- bzw. Domänenmodelle	285
15.2	Modellbasierte Whitebox-Tests	285
15.3	Was steckt hinter UTP und TTCN-3?	286
15.3.1	Das Metamodell der UML	287
15.3.2	Profile – der Erweiterungsmechanismus der UML	288
15.3.3	Die Sprache TTCN-3	290
16	UML und die Modellierung von Tests	291
16.1	U2TP – das UML 2-Testprofil	291
16.1.1	Testarchitektur	292
16.1.2	Testverhalten	294
16.1.3	Testdaten	295
16.1.4	Zeitverhalten	296
16.2	Ein Anwendungsbeispiel	298
16.2.1	Unit Test – die Geld-Klassen	299
16.2.2	Testmodellierung – der Geldautomatentest	301
16.3	Die Abbildung des UML-Testprofils	304
16.3.1	UTP und JUnit	304
16.3.2	UTP und TTCN-3	307
16.3.3	Wann ist eine Testmodellierung sinnvoll?	308

17	Echtzeit- und Performance-Tests	311
17.1	Nicht funktionale Qualitätskriterien	311
17.1.1	Anforderungen an den Test von Echtzeitverhalten	312
17.1.2	Anforderungen an Performance-Tests	313
17.2	Tests auf Echtzeitverhalten modellieren	315
17.2.1	Harte Echtzeitverhaltensanforderungen	315
17.2.2	Weiche Echtzeitverhaltensanforderungen	316
17.3	Performance-Tests modellieren	318
18	Zusammenwachsen von Entwicklung und Qualitätssicherung	321
18.1	Ziele für Entwicklung und Qualitätssicherung	321
18.2	Gemeinsame Aufgabenteilung	322
V	Anhang	325
	Danksagung	327
	Literatur	329
	Index	335