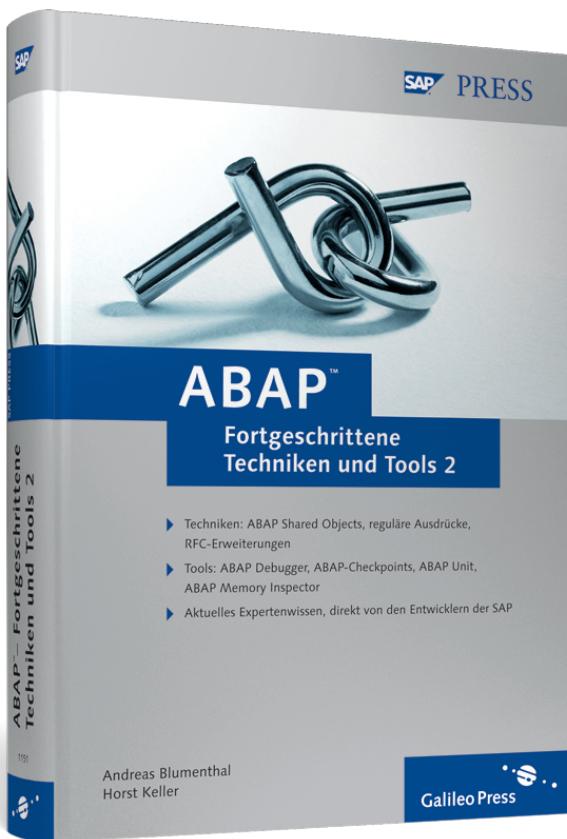


Andreas Blumenthal, Horst Keller

ABAP™ – Fortgeschrittene Techniken und Tools, Band 2



•  •
Galileo Press

Bonn • Boston

Auf einen Blick

1	Effektive Zeichenkettenverarbeitung in ABAP	19
2	Reguläre Ausdrücke für die Zeichenkettenverarbeitung in ABAP	57
3	SAP Simple Transformations	113
4	ABAP Shared Objects: Programmierkonzept zur effizienten Speicherausnutzung	157
5	RFC und RFM – Leitfaden zu ABAP Remote Communications	199
6	bgRFC – Einführung in den Background RFC	265
7	Einsatz der ABAP-Test- und -Analysewerkzeuge in allen Phasen des Entwicklungsprozesses	307
8	Effizientes ABAP Debugging	373
9	Höhere Softwarequalität durch ABAP-Checkpoints	443
10	Testen mit ABAP Unit	477
11	Speicherverbrauchsanalyse mit dem ABAP Memory Inspector	505
12	Switch und Enhancement Framework – der einfache Weg zu konsolidierten Erweiterungen	535

Inhalt

Vorwort	15
---------------	----

1 Effektive Zeichenkettenverarbeitung in ABAP 19

1.1 Datentypen zum Speichern von Zeichen und Bytes	20
1.1.1 Datentypen fester Länge	20
1.1.2 Datentypen variabler Länge	31
1.1.3 Generische Datentypen	36
1.1.4 Auswahl des geeigneten Datentyps	36
1.2 Grundlegende Operationen für die Zeichenkettenverarbeitung	38
1.2.1 Zugriff mittels Offset und Länge	38
1.2.2 Verkettung	40
1.2.3 Zerlegung	41
1.2.4 Verschiebung	42
1.2.5 Formatierung von Zeichenketten	43
1.2.6 Weitere Operationen	45
1.2.7 ABAP-Klassen für die Zeichenkettenverarbeitung	45
1.3 Suchen und Ersetzen	46
1.3.1 Anweisung FIND	47
1.3.2 Anweisung REPLACE	51
1.3.3 Suchen mit logischen Ausdrücken	51
1.4 Fazit	55

2 Reguläre Ausdrücke für die Zeichenkettenverarbeitung in ABAP 57

2.1 Reguläre Ausdrücke in der Textverarbeitung	57
2.1.1 Drei Typen von Textverarbeitungsaufgaben	58
2.1.2 Reguläre Ausdrücke als Retter in der Not	60
2.2 Einführung in die regulären Ausdrücke in ABAP	60
2.2.1 Grundlagen	61
2.2.2 Reguläre Ausdrücke für die Validierung	65
2.2.3 Suchen mit regulären Ausdrücken	68
2.2.4 Reguläre Ausdrücke in ABAP-Programmen	72
2.2.5 Reguläre Ausdrücke außerhalb von ABAP-Programmen	79
2.3 Fortgeschrittene Eigenschaften regulärer Ausdrücke	84
2.3.1 Arbeiten mit Untergruppen	84

2.3.2	Weitere Operatoren und zukünftige Erweiterungen ...	93
2.3.3	Häufige Muster und Probleme	96
2.3.4	Regex-Ressourcen	101
2.4	Technische Aspekte regulärer Ausdrücke	102
2.4.1	Übereinstimmungen, Backtracking und Schnitte	102
2.4.2	Performancekontrolle	107
2.5	Fazit	112
3	SAP Simple Transformations	113
3.1	XML: Esperanto der Computerkommunikation	115
3.1.1	Sprache contra interne Darstellung	117
3.1.2	Interna nach außen kehren?	119
3.2	Strukturtransformationen mit XSLT	122
3.3	Wichtige Funktionen von Simple Transformations	125
3.4	Struktur eines ST-Programms	128
3.5	XML-Content beschreiben	131
3.6	Tiefe Kopien, Schleifen und Richtungsabhängigkeit	135
3.7	Bedingungen	137
3.8	Zuweisung und Variablen	144
3.9	Modularisierung	147
3.10	Vollständiges ST-Beispielprogramm	150
3.11	Simple Transformations im ABAP-Kontext	152
3.12	Fazit	155
4	ABAP Shared Objects: Programmierkonzept zur effizienten Speicherausnutzung	157
4.1	Grundlegende Anwendungsszenarien	159
4.2	Programmiermodell von ABAP Shared Objects	160
4.2.1	Sperren der Gebietsinstanz	161
4.2.2	Zugriff auf Objekte über das Wurzelobjekt	161
4.2.3	Zugriffsmodell für ABAP Shared Objects	163
4.3	Werkzeuge für die Verwaltung und Überwachung des Shared Memorys	164
4.3.1	Transaktion SHMA – Gebiete für Shared Objects definieren	164
4.3.2	Transaktion SHMM – Shared Objects Monitor überwachen	167
4.4	Programmieren mit ABAP Shared Objects	169
4.4.1	Gebietsinstanz erzeugen	171
4.4.2	Auf eine vorhandene Gebietsinstanz zugreifen	173

4.4.3	Gebietsinstanz aktualisieren	174
4.4.4	Objekte lokaler Klasse in einer Gebietsinstanz	175
4.4.5	Fehler behandeln	176
4.4.6	Debugging von Gebietsinstanzen	177
4.5	Erweiterte Programmiertechniken	179
4.5.1	Versionierung von Gebietsinstanzen	180
4.5.2	Automatischer Gebietsaufbau und Einschränken der Lebensdauer	185
4.5.3	Mandantenabhängige Gebiete	189
4.5.4	Synchronisation von Änderungen	190
4.5.5	Festlegen von Speichergrenzen und Verdrängbarkeit	192
4.5.6	Gleichzeitige Änderungssperren auf mehreren Gebietsinstanzen	193
4.5.7	Weiteres zu Gebietshandles	193
4.6	Empfohlenes Programmiermodell	194
4.7	Fazit	196

5 RFC und RFM – Leitfaden zu ABAP Remote Communications 199

5.1	Grundlagen des RFC und mögliche Varianten	200
5.1.1	Grundlegende Terminologie	200
5.1.2	RFC-Kommunikationsprozess	204
5.1.3	Fünf grundlegende RFC-Varianten	207
5.1.4	RFC und Dialoginteraktionen	228
5.2	Remotefähige Funktionsbausteine	233
5.2.1	RFM anlegen	233
5.2.2	Einrichten von RFC-Destinationen	238
5.2.3	Vordefinierte Destinationen	247
5.3	Datenübertragung und Ausnahmebehandlung beim RFC	249
5.3.1	Serialisierung/Deserialisierung von ABAP-Daten beim RFC	249
5.3.2	Ausnahmebehandlung beim RFC	260
5.4	Fazit	264

6 bgRFC – Einführung in den Background RFC 265

6.1	Grundlegende Begriffe	266
6.1.1	bgRFC-Units	266
6.1.2	bgRFC-Destinationen	268

6.1.3	bgRFC-Szenarien	269
6.1.4	QoS-Level für bgRFC	273
6.2	Programmieren mit bgRFCs	282
6.2.1	Erstellen von Inbound-Units	282
6.2.2	Erstellen von Outbound-Units	285
6.2.3	Aufheben einer Sperre für eine Outbound-Unit	286
6.2.4	Erstellen einer bgRFC-Outbound-Unit vom Typ Q	286
6.2.5	Erstellen einer bgRFC-Out-Inbound-Unit vom Typ Q	287
6.2.6	Verbuchung und bgRFC	288
6.2.7	bgRFC-Scheduler	289
6.2.8	Vorteile des Out-Inbound-Szenarios	290
6.3	Konfigurieren von bgRFC-Destinationen	292
6.3.1	Outbound-Destinationen	292
6.3.2	Inbound-Destinationen	293
6.4	Überwachen von bgRFC-Units	295
6.5	Grundlegende Konfiguration eines bgRFC-Schedulers	300
6.6	Fazit	305

7 Einsatz der ABAP-Test- und -Analysewerkzeuge in allen Phasen des Entwicklungsprozesses 307

7.1	Einsatz von Prüf- und Testwerkzeugen während der Entwicklung und in der Testphase	308
7.1.1	ABAP-Programme systematisch testen	308
7.1.2	Werkzeuge für statische Prüfungen	312
7.1.3	Werkzeuge für Unit- und Integrationstests	323
7.1.4	Überprüfen der Programmausführung mit dem Coverage Analyzer	330
7.1.5	Tipps zur Verwendung der Testwerkzeuge	334
7.2	Einsatz von Analysewerkzeugen während der Testphase und des ersten produktiven Einsatzes von ABAP-Applikationen	335
7.2.1	Auswahl des richtigen Werkzeugs	336
7.2.2	Post-Mortem-Analyse	338
7.2.3	Systemprotokoll	339
7.2.4	ABAP-Dumpanalyse	342
7.2.5	Analyse eines beendeten Hintergrundjobs mit der ABAP-Dumpanalyse	347
7.2.6	Analyse der Programmstrukturen mit der ABAP-Trace-Funktionalität	355
7.2.7	Wann Sie den ABAP Debugger brauchen	369
7.3	Fazit	370

8 Effizientes ABAP Debugging 373

8.1	Starten des ABAP Debuggers	375
8.1.1	Starten des Debuggers zu Beginn der Programmausführung	375
8.1.2	Starten des Debuggers während einer Dialogtransaktion	380
8.1.3	Starten des Debuggers für einen Hintergrundjob	383
8.2	Breakpoints und Watchpoints	392
8.2.1	Breakpoints	392
8.2.2	Watchpoints	397
8.3	Neuer ABAP Debugger	405
8.3.1	Gründe für einen neuen Debugger	406
8.3.2	Einstellen des Debuggers	407
8.3.3	Zwei-Prozess-Architektur	408
8.3.4	Benutzeroberfläche	410
8.3.5	Debugger-Werkzeuge	419
8.3.6	Debugging in der Praxis	432
8.4	Fazit	438
8.5	Exklusiver und nicht exklusiver Debugging-Modus	440

9 Höhere Softwarequalität durch ABAP-Checkpoints 443

9.1	ABAP-Checkpoints	444
9.2	Assertions	445
9.3	Aktivierbare Assertions	447
9.3.1	Checkpoint-Gruppen	448
9.3.2	Aktivierungseinstellungen	451
9.3.3	Mehrere Aktivierungseinstellungen und Kontextpriorität	454
9.3.4	Globale Aktivierung und Aktivierung für andere Benutzer	455
9.3.5	Programmspezifische Aktivierung und Priorität von Gültigkeitsbereichen	456
9.3.6	Aktivierungsvarianten	457
9.3.7	Übersicht über alle Aktivierungseinstellungen anzeigen	459
9.4	Breakpoints	460
9.5	Aktivierbare Breakpoints	461
9.6	Logpoints	462
9.6.1	Logpoints sind aktivierbar	462
9.6.2	Speichern von Daten mithilfe des FIELDS-Zusatzes ...	464

9.6.3	Steuern der Aggregation mit dem Zusatz SUBKEY	466
9.7	Wann werden Änderungen an den Aktivierungseinstellungen wirksam?	467
9.8	Weitergehende Funktionalität der ASSERT-Anweisung	469
9.8.1	Betriebsarten	469
9.8.2	Anweisungszusätze für die Protokollierung	471
9.8.3	Systemvarianten für die gruppenübergreifende Aktivierung	472
9.9	Hinweise zur Verwendung von Assertions	474
9.10	Fazit	474

10 Testen mit ABAP Unit 477

10.1	Prinzipien und Vorteile von Modultests	477
10.1.1	Modultests vs. Abnahmetests	478
10.1.2	Modultests sind Entwicklertests	479
10.2	Grundlagen von ABAP Unit	479
10.3	Überprüfen von Testannahmen	481
10.3.1	Assert-Methoden	481
10.3.2	Parameter für Assert-Methoden	483
10.3.3	Assert-Methode fail	483
10.3.4	Assert-Methode abort	484
10.3.5	Spezialisierte Assert-Methoden assert_....	484
10.3.6	Assert-Methode assert_equals	484
10.3.7	Assert-Methoden assert_bound und assert_not_bound	487
10.3.8	Assert-Methode assert_differs	487
10.3.9	Assert-Methoden assert_initial und assert_not_initial	487
10.3.10	Assert-Methode assert_subrc	487
10.3.11	Assert-Methode assert_that	488
10.4	Ausnahmebehandlung in Testmethoden	488
10.5	Beispiel für Modultests einer globalen Klasse	489
10.5.1	Globale Beispielklasse	489
10.5.2	Lokale Testklasse	490
10.6	Ergebnisanzeige	492
10.6.1	Hierarchiedarstellung	492
10.6.2	Fehlerliste	493
10.6.3	Detailsicht	494
10.7	Bereitstellen des Umfeldes	494
10.7.1	Implizite Methoden	495
10.7.2	Delegation	496

10.8 Testisolation	497
10.9 Wiederverwenden von Tests	498
10.10 Massentests mit ABAP Unit	501
10.11 Fazit	503
11 Speicherverbrauchsanalyse mit dem ABAP Memory Inspector	505
11.1 Grundlagen der Speicherverwaltung	506
11.1.1 Dynamische Speicherobjekte in ABAP	508
11.1.2 Löschen von dynamischen Speicherobjekten	511
11.1.3 Funktionsweise des ABAP Garbage Collectors	512
11.1.4 Berechnung des Speicherverbrauchs	514
11.2 Speicherlecks	518
11.2.1 Häufige Ursachen für Speicherlecks oder Speicherverschwendungen	518
11.2.2 Speicherverbrauchsanalyse mit dem ABAP Memory Inspector	519
11.2.3 Starke Zusammenhangskomponenten	522
11.3 Verwendung des ABAP Memory Inspectors	524
11.3.1 Analyse des Speicherverbrauchs eines laufenden Programms	525
11.3.2 Erzeugen von Speicherabzügen	528
11.3.3 Analysieren und Vergleichen von Speicherabzügen	529
11.4 Fazit	534
12 Switch und Enhancement Framework – der einfache Weg zu konsolidierten Erweiterungen	535
12.1 Kurzer Überblick über SAP-Branchenlösungen	537
12.2 Entwicklung von SAP-Modifikationen und -Erweiterungen	540
12.3 Einführung in das Switch und Enhancement Framework	542
12.3.1 Enhancement Framework	544
12.3.2 Switch Framework	546
12.4 Switch und Enhancement Framework in der Praxis	550
12.5 Fazit	558
Herausgeber und Autoren	559
Index	565

Vorwort

Dieses Buch ist der zweite Band des Buchs *ABAP – Fortgeschrittene Techniken und Tools*, das vor vier Jahren bei SAP PRESS erschienen ist. Wie sein Vorgänger ist es eine Zusammenfassung einer Reihe von Artikeln über ABAP und das ABAP-Umfeld, die in englischer Sprache im SAP Professional Journal (<http://www.sapro.com>) erschienen sind. Der Ansatz des ersten Bandes, ABAP-spezifische Artikel des im deutschsprachigen Raum nicht so stark verbreiteten SAP Professional Journals einer erweiterten und zugleich fachlich spezieller interessierten Leserschaft in einem Buch zugänglich zu machen, stieß auf recht positive Resonanz. Deshalb wollen wir dieses Konzept mit dem vorliegenden Band fortführen und präsentieren Ihnen hier die in den Jahren 2004 bis 2007 erschienenen ABAP-spezifischen Artikel des SAP Professional Journals in deutscher Sprache und in einer aktualisierten Fassung.

Wieder handelt es sich um Artikel, die von den Mitarbeitern der SAP, die die Sprache ABAP und die dazugehörigen Werkzeuge entwickeln, selbst geschrieben wurden, und damit um Wissen aus erster Hand. Somit bleiben auch die im Vorwort des ersten Bandes gemachten Aussagen im Wesentlichen gültig. Auch das vorliegende Buch dient der Abrundung anderer SAP PRESS-Bücher, die im Bereich »NetWeaver Core AS&DM ABAP« (das ist der derzeitige Name der ABAP-Gruppe) der SAP entstanden sind. Dies gilt insbesondere für den Einführungsband *ABAP Objects – ABAP-Programmierung mit SAP NetWeaver* (SAP PRESS 2006), in dem die Themengebiete des vorliegenden Buchs entweder nur kurz, mit einem anderen Tenor oder gar nicht erwähnt werden. Im vorliegenden Buch sind diese speziellen Themengebiete als abgeschlossene Einheiten in Kapiteln zusammengefasst.

Alle Artikel dieses Buchs wurden einer eingehenden fachlichen und terminologischen Überarbeitung durch ihre Autoren und die Herausgeber unterzogen, teilweise zusammengefasst, neu geordnet oder sogar neu geschrieben. Während einige Artikel im Original den Stand der Releases 6.20/6.40 beschreiben, spiegelt die vorliegende Überarbeitung das aktuelle Release 7.0 des SAP NetWeaver Application Servers ABAP wider. Auf wesentliche Neuerungen, die für die Releases nach 7.0 entwickelt wurden und die derzeit im Rahmen eines »Feature Set Backports« nach Release 7.0, EhP2 übernommen werden, weisen wir jeweils hin.

Inhaltlich bieten wir wieder einen bunten Strauß von Themen, von dem wir hoffen, dass für jeden ABAP-Entwickler etwas dabei ist:

- ▶ Wir beginnen mit zwei Kapiteln über die Zeichenkettenverarbeitung. Stand Release 7.0. Obwohl es hierbei ab Release 7.0, EhP2 erhebliche Verbesserungen durch Zeichenkettenausdrücke und neu eingebaute Zeichenkettenfunktionen geben wird, bleiben viele der in **Kapitel 1** gemachten Aussagen gültig. Dies gilt insbesondere auch für das Verständnis vorhandener Programme, in denen die neuen Sprachkonstrukte noch nicht verwendet werden. Die grundlegende Beschreibung regulärer Ausdrücke in **Kapitel 2** ist ohnehin weitestgehend Release-unabhängig.
- ▶ Nachdem wir im ersten Band bereits XSLT und das Format asXML für die Serialisierung von ABAP-Daten nach XML eingeführt haben, beschäftigt sich **Kapitel 3** mit den Simple Transformations (ST), einer SAP-eigenen XML-basierten Programmiersprache, die rein auf Transformationen zwischen ABAP-Daten und XML zugeschnitten ist und entsprechende Performance-Vorteile liefert.
- ▶ **Kapitel 4** gibt Ihnen eine Einführung in die explizite Verwendung des Shared Memorys eines Applikationsservers mit den Mitteln von ABAP Shared Objects. Nach der Lektüre dieses Kapitels sollten Sie in der Lage sein, wenig veränderliche Daten, die von vielen Programmen gleichzeitig verwendet werden, platzsparend ein einziges Mal im Shared Memory abzulegen und anderen Programmen darauf Zugriff zu gewähren.
- ▶ Die nächsten beiden Kapitel sind der ABAP-Remote-Kommunikation und hierbei insbesondere dem Remote Function Call (RFC) gewidmet – ein äußerst wichtiges Thema, das leider häufig zu kurz kommt. Während **Kapitel 5** einen umfassenden Überblick über die RFC-Programmierung liefert, führt **Kapitel 6** in den neuen Background RFC (bgRFC) ein, der die vorhergehenden Techniken transaktionaler RFC (tRFC) und queued RFC (qRFC) ablöst.
- ▶ Mit den folgenden zwei Kapiteln halten Sie die schriftlich ausgearbeitete Form der erfolgreichsten und meistbesuchten ABAP-Workshops der SAP TechEds der vergangenen Jahre in den Händen. Das »ABAP Trouble Shooting« mit verschiedenen Test- und Analysewerkzeugen (**Kapitel 7**) und der Umgang mit dem ABAP Debugger (**Kapitel 8**) sind Themen, mit denen sich jeder ABAP-Entwickler wohl oder übel auseinandersetzen muss. Hier erhalten Sie viele wertvolle Tipps.
- ▶ **Kapitel 9** führt in die sogenannten Checkpoints ein. Checkpoints instrumentieren ABAP-Programme zu Testzwecken. Sie umfassen neben den vom Debugging bekannten Breakpoints auch Assertions und Logpoints, die alleamt von außerhalb eines Programms gesteuert werden können. Kein modernes ABAP-Programm sollte auf den Einsatz von Checkpoints verzichten.

- ▶ Mit **Kapitel 10** erhalten Sie eine prägnante Einführung in das ABAP-Testwerkzeug schlechthin, nämlich in ABAP Unit für Modultests. Nach der Lektüre dieses Kapitels sollte Sie nichts mehr davon abhalten, die gesamte Funktionalität Ihrer Programme mit solchen Tests zu überprüfen und damit ihre Stabilität zu garantieren.
- ▶ **Kapitel 11** schließt den Kreis der Testwerkzeuge mit einer Einführung in den ABAP Memory Inspector. In Zeiten von mehr und mehr dynamisch erzeugten Objekten und Datenobjekten wird es immer wichtiger, den dadurch zur Laufzeit belegten Speicher zu kontrollieren, um Speicherlecks vorzubeugen. Das Werkzeug ABAP Memory Inspector ist hierfür unerlässlich.
- ▶ Abschließend gibt **Kapitel 12** einen einführenden Überblick über das Enhancement Framework und das Switch Framework, die seit Release 7.0 zur Verfügung stehen. Mithilfe dieser Technologie können verschiedene Branchenlösungen in einem Core-System entwickelt und ausgeliefert werden. Weiterhin läutet das Enhancement Framework eine neue Ära der modifikationsfreien Änderung von SAP-Programmen ein.

Wir danken allen Autoren, die nochmals die Mühe einer Überarbeitung ihrer Artikel auf sich genommen haben, Wellesley Information Services, dass sie uns die Genehmigung für die deutschsprachige Veröffentlichung der Artikel erteilt haben, und Galileo Press für die Übersetzung der Artikel ins Deutsche. Bei Galileo Press sei insbesondere Frau Mirja Werner und Herrn Stefan Proksch vom Lektorat SAP PRESS für die gute Betreuung bei der Erstellung dieses Buchs gedankt.

Wie immer weisen wir abschließend darauf hin, dass wir, wie im Deutschen leider üblich, bei Personenbezeichnungen durchgehend die männliche Form verwenden, diese aber stellvertretend für die männliche und weibliche Form verstehen. Wenn wir also von Entwicklern und Benutzern sprechen, schließt dies selbstverständlich auch Entwicklerinnen und Benutzerinnen ein. Die Verwendung der männlichen Form folgt letztlich auch der Terminologie des SAP NetWeaver Application Servers, die ebenfalls die männliche Form verwendet, beispielsweise Benutzer, Benutzername und Benutzerkonto.

Andreas Blumenthal

Vice President, NetWeaver Core AS&DM ABAP

Horst Keller

Knowledge Architect, NetWeaver Core AS&DM ABAP

Reguläre Ausdrücke sind ein mächtiges und weit verbreitetes Standardwerkzeug zur effizienten Verarbeitung von zeichenartigen Daten. Dieses Kapitel bietet eine allgemeine Einführung in reguläre Ausdrücke und beschreibt deren Integration in die ABAP-Welt. Anhand zahlreicher Beispiele wird gezeigt, wie reguläre Ausdrücke die Performance von ABAP-Programmen verbessern können.

Ralph Benziger und Björn Mielenhausen

2 Reguläre Ausdrücke für die Zeichenkettenverarbeitung in ABAP

2.1 Reguläre Ausdrücke in der Textverarbeitung

Die Textverarbeitung macht einen großen Teil der Informationsverarbeitung aus, insbesondere im Umfeld typischer SAP-Geschäftsanwendungen. Hier bezieht sich Textverarbeitung nicht nur auf die Arbeit mit Text in natürlicher Sprache, sondern auch auf andere Datentypen, die in einer Textdarstellung gespeichert werden, zum Beispiel Datumsangaben, Währungen oder XML-Daten. Auf solche Informationen zuzugreifen, sie zu analysieren und zu ändern sind häufige Aufgaben in einer ABAP-Entwicklung.

Im Allgemeinen lassen sich Textverarbeitungsaufgaben in drei Kategorien gliedern:

► **Validierung**

Stimmen Informationen im Hinblick auf syntaktische Eigenschaften mit einer Spezifikation überein?

► **Extraktion**

Lokalisierung von Informationen in einer größeren Datenmenge, basierend auf dem Kontext, nicht auf der Position

► **Transformation**

Konvertierung von Informationen in unterschiedliche strukturelle Darstellungen

Reguläre Ausdrücke sind ein standardisiertes und weit verbreitetes Werkzeug für die effiziente und effektive Verarbeitung von textbasierten Informationen.

Vor SAP NetWeaver 7.0 gab es in ABAP aber keine native Unterstützung für reguläre Ausdrücke, und sie konnten in ABAP-Programmen nur mithilfe von allerlei Tricks und damit nur sehr eingeschränkt verwendet werden.¹

Vor der eingehenden Beschäftigung mit den Details zur Verwendung von regulären Ausdrücken werden die genannten Kategorien von Textverarbeitungsaufgaben, für die sich reguläre Ausdrücke ideal eignen, etwas näher betrachtet.

2.1.1 Drei Typen von Textverarbeitungsaufgaben

Anhand einiger kleiner Beispiele werden die drei Typen von Textverarbeitungsaufgaben illustriert: Informationen validieren, extrahieren und transformieren. Anschließend wird der erforderliche Aufwand untersucht, um eines dieser Beispiele mit konventionellem ABAP, das heißt ohne die Hilfe von regulären Ausdrücken zu lösen.

Informationen validieren

Angenommen, Sie schreiben einen ABAP-basierten Webservice, um Kreditkarteninformationen als Teil eines E-Commerce-Systems zu verarbeiten. Das Design der Methode sieht vor, dass die Kreditkartensummer und das Gültigkeitsdatum als Zeichenfolgen übergeben werden. Um ein sicheres Design zu gewährleisten, ist es sinnvoll, die Gültigkeit der empfangenen Daten oder zumindest deren Plausibilität zu überprüfen. In diesem Fall soll sichergestellt werden, dass die Kreditkartensummer eine 15- oder 16-stellige Zahl und das Gültigkeitsdatum eine zweistellige Zahl, gefolgt von einem Schrägstrich und einer weiteren zweistelligen Zahl, ist:

```
1234567812345678 02/06 -> akzeptieren
123456781234XXXX 03/07 -> ablehnen
1234567812345678 04.08 -> ablehnen
```

Während sich eine solche Validierung der Kreditkartensummer in ABAP problemlos mithilfe des `CO`-Operators (`contains only`) und der `strlen()`-Funktion (`string length`) implementieren lässt, erfordert die Überprüfung des Gültigkeitsdatums etwas mehr Aufwand.

Im Allgemeinen gilt: Je flexibler die Spezifikation ist, desto schwieriger wird es, sie in eine knappe und präzise ABAP-Implementierung umzusetzen. Wenn

¹ Ein Beispiel ist der Aufruf von JavaScript aus ABAP heraus, nur um dort einen regulären Ausdruck auszuwerten.

die Spezifikation für das Datum beispielsweise einstellige Monatsangaben sowie vier- und zweistellige Jahresangaben zulassen würde, würden sich die Anforderungen an das ABAP-Coding drastisch erhöhen.

Informationen extrahieren

Eine weitere häufige Aufgabe ist das Extrahieren von Informationen aus einer größeren Datenmenge. Angenommen, Sie erstellen einen Webservice zur Bereitstellung von Dokumenten für Benutzer. Das Backend empfängt eine URL vom Frontend, die neben weiteren Informationen die ID des angeforderten Dokumentes enthält:

`http://docserve.sap.com/serve?user=ralph&docid=NW2004&lang=EN`

In diesem Beispiel wird Code zum Extrahieren des Dokumentnamens NW2004 aus der Zeichenfolge benötigt. Weitere typische Beispiele sind die Extraktion von Pfaden und Dateierweiterungen aus Dateinamen oder von E-Mail-Adressen aus Headern von E-Mail-Nachrichten.

Informationen transformieren

Bei dieser Aufgabengruppe geht es um die Umwandlung einer strukturellen Darstellung von Informationen in eine andere. Angenommen, in Ihrer Anwendung werden Telefonnummern in einem lesbaren Format gespeichert, das Leerzeichen, Klammern und Bindestriche umfasst:

(800) 123-4567
+49 6227 747474

Um die Anwendung mit einer externen Anwendung zu verbinden, die Telefonnummern als Ziffernfolgen erfordert, müssen Sie Code schreiben, der alle Zeichen, bei denen es sich nicht um Ziffern handelt, aus den Informationen entfernt. Wenn die Anzahl der Zeichen, die keine Ziffern sind, bekannt und gering ist, kann diese Aufgabe in ABAP problemlos mithilfe einiger REPLACE-Anweisungen ausgeführt werden. Dieser Ansatz wird jedoch schnell mühsam und auch fehleranfällig, wenn die Anzahl der zu testenden Nicht-Ziffern sehr hoch ist.

Zu weiteren Beispielen für die Informationstransformation zählen das Ersetzen von mehreren identischen Zeichen durch ein einziges Zeichen oder das Entfernen von HTML-Tags (< . . . >) aus Dokumenten im HTML-Format.

2.1.2 Reguläre Ausdrücke als Retter in der Not

Obwohl die aufgeführten Beispiele oft nur sehr kleine Teilaufgaben innerhalb einer größeren Entwicklung darstellen, erfordern ihre Lösungen unverhältnismäßig viel ABAP-Code. Genau für solche Aufgaben können reguläre Ausdrücke optimal eingesetzt werden.

Um einen ersten Vorgeschmack auf die Vorteile von regulären Ausdrücken zu erhalten, wird eine der beschriebenen einfachen Aufgaben mit ihrer Hilfe gelöst. Listing 2.1 zeigt zunächst den traditionellen ABAP-Code, mit dem sämtliche Tags aus einem HTML-Dokument entfernt werden, das in der Variablen `mytext` gespeichert ist:

```
WHILE mytext CS '<'.
begin = sy-fdpos.
FIND '>' IN SECTION OFFSET begin OF mytext
      MATCH OFFSET end.
len = end - begin + 1.
REPLACE SECTION OFFSET begin LENGTH len OF mytext WITH ``.
ENDWHILE.
```

Listing 2.1 Traditioneller ABAP-Code

Wie Sie sehen, beinhaltet dieser Code eine Schleife mit zwei Suchvorgängen und eine Anweisung zum Ersetzen, um nur die einfache Aufgabe durchzuführen, alle Elemente zwischen spitzen Klammern sowie diese Klammern selbst auszuwechseln. Dass diese einfache Aufgabe einen derart umfangreichen Code erfordert, ist sicher nicht zufriedenstellend.

An dieser Stelle soll schon einmal gezeigt werden, wie dieses ABAP-Codefragment durch die Verwendung eines regulären Ausdrucks in einen einzeiligen Code umgewandelt werden kann, der, abhängig von der zu verarbeitenden Textmenge, zudem mindestens eine Größenordnung schneller ist:

```
REPLACE ALL OCCURRENCES OF REGEX '<[^>]*>' IN mytext WITH ``.
```

2.2 Einführung in die regulären Ausdrücke in ABAP

Nach einer kurzen Einführung in die Syntax von regulären Ausdrücken und deren Integration in die Sprache ABAP erfahren Sie in diesem Abschnitt, wie Sie reguläre Ausdrücke in Ihren eigenen ABAP-Programmen verwenden können, um die genannten Kategorien der Informationsverarbeitung effizient und effektiv auszuführen.

2.2.1 Grundlagen

Reguläre Ausdrücke, oder kurz *Regexe* (für *Regular Expressions*), sind ein Nebenprodukt der theoretischen Informatik, das in den 1950er-Jahren vom kanadischen Mathematiker Stephen Kleene eingeführt wurde (damals wurde die Informatik natürlich noch Mathematik genannt). Regexe leisteten nicht nur einen Beitrag in der Mathematik, sondern erfreuten sich auch als Teil des UNIX-Betriebssystems und dessen textorientierten Werkzeugen wie *grep*, *awk* und *sed* größter Beliebtheit.

Heute unterstützen praktisch alle modernen Programmiersprachen, einschließlich ABAP, reguläre Ausdrücke entweder nativ oder über zusätzliche Bibliotheken. Frühere Versionen von ABAP boten noch keine Unterstützung für reguläre Ausdrücke und erforderten verschiedene Umwege, um externe Regex-Funktionalität zu nutzen. Ab SAP NetWeaver 7.0 ist die native Regex-Unterstützung nun aber endlich verfügbar.

Grundlegende Ausdrücke

Ein regulärer Ausdruck ist ein *Textmuster*, das stellvertretend für eine oder mehrere *Zeichenfolgen* steht. Die Muster regulärer Ausdrücke bestehen aus normalen Zeichen (Literalzeichen) sowie einigen speziellen Zeichen, die *Operatoren*, *Sonder-* oder *Metazeichen* genannt werden:

. * + ? ^ \$ () { } [] \ |

Der *Punktoperator* ist beispielsweise ein Platzhalter, der ein beliebiges Einzelzeichen darstellt. Folglich steht das Muster `cat` für die Zeichenfolge `cat`, wohingegen das Muster `c.t` Zeichenfolgen darstellt, die aus drei Zeichen bestehen, mit `c` beginnen und auf `t` enden. Beachten Sie daher, dass ein Regex ohne Operatoren genau eine Zeichenfolge darstellt, nämlich sich selbst.

Ein regulärer Ausdruck repräsentiert eine Menge von Zeichenfolgen. Wenn eine Zeichenfolge eine der von einem Regex repräsentierten Zeichenfolgen ist, wird auch gesagt, dass der Regex mit der Zeichenfolge übereinstimmt oder dass der Regex auf die Zeichenfolge passt (Englisch: *matches*). Das Ermitteln von Übereinstimmungen für Zeichenfolgen ist eine der häufigsten Verwendungen von Regexen: Um zu überprüfen, ob eine Zeichenfolge zu einer Gruppe von Zeichenfolgen mit einer bestimmten Eigenschaft gehört, wird ermittelt, ob ein Regex, der diese Eigenschaft codiert, zur Zeichenfolge passt. Im Beispiel codiert der Regex `c.t` die Eigenschaft »eine Zeichenfolge aus drei Zeichen, die mit `c` beginnt und auf `t` endet«, sodass der Regex `c.t` zum Beispiel mit `cut`, `cat` und `cot` übereinstimmt (mit `car` jedoch *nicht*).

Eine weitere Möglichkeit, um mehr als eine Zeichenfolge über einen regulären Ausdruck zu repräsentieren, besteht darin, sämtliche *Alternativen* aufzulisten und diese durch einen *Pipe- bzw. Oder-Operator* (`|`) zu trennen. Der Regex `ten|(twen|thir)ty` passt ausschließlich zu den Zeichenfolgen `ten`, `twenty` und `thirty`. Dieses Beispiel zeigt zudem, dass ähnlich wie in arithmetischen Ausdrücken runde *Klammern* zur Strukturierung des Ausdrucks verwendet werden können.

Soll eine Übereinstimmung für ein Operatorzeichen ermittelt werden, kann diesem Zeichen der *Rückstrich- bzw. Backslash-Operator* vorangestellt werden, um es in ein Literal zu verwandeln. Dieser Vorgang wird als *Escaping* bezeichnet, und der Rückstrich ist ein Fluchtsymbol. Beispielsweise stimmt der Regex `10\.\.0` mit `10.0` überein. Der Regex `10.0` stimmt mit `10.0` überein, darüber hinaus jedoch auch mit `1000`, `10A0` etc. Das Weglassen des Backslash-Operators vor Dezimalpunkten ist ein typischer Anfängerfehler, der häufig unbemerkt bleibt, da der fehlerhafte Regex trotzdem mit dem gewünschten Text übereinstimmt – allerdings nicht nur mit diesem!

Mengen und Klassen

Das ausdrückliche Auflisten zahlreicher Alternativen kann mühsam werden und sich vor allem auch negativ auf die Performance auswirken (siehe Abschnitt 2.4.2, »Performancekontrolle«). Glücklicherweise gibt es als *Mengen* bezeichnete reguläre Ausdrücke, um alternative Einzelzeichen kurz und klar zu definieren.

Der Mengenausdruck `[abc123.!?]` stimmt mit allen innerhalb der eckigen Klammern aufgelisteten Einzelzeichen überein (und nur mit diesen). Beachten Sie, dass alle Metazeichen (Backslash ausgenommen) ihre spezielle Bedeutung innerhalb von Mengen verlieren, sodass der Punkt in der gezeigten Menge nur mit einem literalen Punkt übereinstimmt, nicht jedoch mit einem beliebigen Zeichen.

Darüber hinaus ist die *Negation* einer Menge möglich, indem der Negationsoperator (`^`) als erstes Zeichen in einer Menge hinzugefügt wird. Eine negierte Menge stimmt mit jedem einzelnen Zeichen *außer* den in der Menge aufgelisteten Zeichen überein. Auf diese Weise stimmt der Ausdruck `[^0-9a-f]` zum Beispiel mit allen Einzelzeichen außer einer hexadezimalen Ziffer überein. Beachten Sie, dass das Winkelzeichen für eine Negation an erster Position stehen muss, da es anderenfalls als literales Winkelzeichen interpretiert wird.

Um einen der Mengenoperatoren oder die schließende Klammer selbst zur Menge hinzuzufügen, können Sie einfach, wie beschrieben, den Backslash-Operator als Fluchtsymbol benutzen. Der Ausdruck `\^\-\`` stimmt beispiels-

weise mit einem Winkelzeichen, einem Bindestrich oder einer schließenden eckigen Klammer überein.

Ein *Bereich* bietet eine noch präzisere Möglichkeit, um Mengen zu definieren. Hier müssen die Zeichen nicht einzeln aufgelistet werden, sondern der Bereichsoperator (-) wird verwendet, um den Start- und den Endpunkt eines Zeichenintervalls festzulegen. Anstelle von `[0123456789abcdef]` kann so die einfachere Schreibform `[0-9a-f]` verwendet werden, um eine einzelne hexadezimale Ziffer zu charakterisieren.

Bereiche sind zwar insbesondere in technischen Domänen sehr nützlich, aber Sie sollten dabei immer beachten, dass Bereiche von der Textumgebung abhängig sind! Beispielsweise hängen vom Bereich `[ä-ß]` abgedeckte Zeichen stark von der aktuellen Codepage ab. Daher ist es schwierig, ABAP-Code zu pflegen, der solche Bereiche verwendet. Glücklicherweise lassen sich die meisten Probleme im Zusammenhang mit Bereichen ganz einfach vermeiden, indem Sie vordefinierte Zeichenuntermengen benutzen, die als *Zeichenklassen* bezeichnet werden.

In Zeichenklassen enthaltene Zeichen sind zwar ebenfalls von der Textumgebung abhängig, jedoch bleibt die zugeschriebene Bedeutung jeder Klasse für alle Codepages einheitlich. Um eine Klasse zu verwenden, fügen Sie den Klassennamen oder die zugehörige Kurzform (siehe Tabelle 2.1) in eine Mengendefinition ein. Zum Beispiel:

- ▶ Die Mengendefinition `[:alpha:][:digit:].[?!\\-"]` passt zu allen Einzelzeichen, die typischerweise in Schriftstücken verwendet werden.
- ▶ Die Mengendefinition `[:word:>@\\-+.]` passt zu allen Einzelzeichen, die typischerweise in E-Mail-Adressen verwendet werden.

Klassenname	Kurzform	Zeichen	Negierte Kurzform
<code>[:alpha:]</code>		alle alphanumerischen Zeichen	
<code>[:digit:]</code>	<code>\d</code>	alle Ziffern	<code>\D</code>
<code>[:upper:]</code>	<code>\u</code>	alle Großbuchstaben	<code>\U</code>
<code>[:lower:]</code>	<code>\l</code>	alle Kleinbuchstaben	<code>\L</code>
<code>[:word:]</code>	<code>\w</code>	alle alphanumerischen Zeichen zuzüglich Unterstrich (<code>_</code>)	<code>\W</code>
<code>[:space:]</code>	<code>\s</code>	alle Leerzeichen, Tabulatoren, Zeilenvorschübe, Zeilenum- brüche und Seitenvorschübe	<code>\S</code>
<code>[:punct:]</code>		alle Interpunktionszeichen	

Tabelle 2.1 Einige vordefinierte Zeichenklassen

Die Kurzformen der Zeichenklassen können auch außerhalb von Mengen verwendet werden, um die Lesbarkeit der Ausdrücke zu verbessern. Die negierten Kurzformen wie \D entsprechen negierten Zeichenmengen wie [^\d] und können daher nicht innerhalb von Mengendefinitionen verwendet werden.

Wiederholungen

Bisher wurden lediglich Platzhalter eingeführt, die für Einzelzeichen stehen. Für die Definition von Platzhaltern für mehrere Zeichen steht eine Vielzahl von *Wiederholungs- bzw. Verkettungsoperatoren* (englisch *Quantifiers*) zur Verfügung (siehe Tabelle 2.2).² Ein Verkettungsoperator wiederholt den unmittelbar vorangestellten Regex und erzeugt damit Verkettungen dieses Regexes.

Verkettungsoperatoren	Bedeutung
a+	mindestens eine Wiederholung des Zeichens a
a*	keine oder mehrere Wiederholungen des Zeichens a
a{ n }	exakt n Wiederholungen des Zeichens a
a{ n , m }	mindestens n und maximal m Wiederholungen des Zeichens a
a?	ein optionales Zeichen a (das heißt keine oder eine Wiederholung des Zeichens a)

Tabelle 2.2 Regex-Verkettungsoperatoren für Wiederholungen

Wie bei Alternativen lässt sich mithilfe von Klammern der Bezug von Verkettungsoperatoren auf größere Unterausdrücke erweitern:

- ▶ abc* wiederholt c und stimmt mit ab, abc, abcc, abccc etc. überein.
- ▶ a(bc)* wiederholt bc und stimmt mit a, abc, abcbc, abcbcabc etc. überein.
- ▶ [abc]* wiederholt [abc] und stimmt mit a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab etc. sowie der leeren Zeichenfolge überein.

Wie im letzten Beispiel gezeigt, wird der quantifizierte Unterausdruck für jede Wiederholung erneut ausgewertet, das heißt, der Unterausdruck stimmt möglicherweise bei jeder Iteration mit einer anderen Zeichenfolge überein. Folglich stimmt [abc]* nicht nur mit Zeichenfolgen überein, die aus identischen

² Das Arsenal an Verkettungsoperatoren ist umfangreicher als eigentlich notwendig. Beispielsweise kann (a+)? anstelle von a* oder aa* anstelle von a+ verwendet werden. Es ist sogar zusätzlich a{ n , } vorhanden, was a{ n }a* entspricht.

Zeichen bestehen, sondern auch mit beliebigen Kombinationen aus den Zeichen a, b und c.

Die richtige Anzahl an Wiederholungen wird für jeden Unterausdruck implizit so bestimmt, dass der gesamte Regex möglichst übereinstimmt, sofern dies überhaupt möglich ist. Das bedeutet, dass der Verkettungsoperator beim Anwenden des Regexes a^*aa auf die Zeichenfolge $aaaa$ eine Übereinstimmung von a^* für die ersten beiden Zeichen a ermittelt und die verbleibenden Zeichen a den literalen Zeichen a im Regex vorbehalten bleiben.

2.2.2 Reguläre Ausdrücke für die Validierung

Nach der Einführung der grundlegenden Operatoren für reguläre Ausdrücke wenden sich die Ausführungen nun einigen Programmieraufgaben zu. Sie werden überrascht sein, wie nützlich bereits die beschriebenen einfachen Regexe eingesetzt werden können.

In diesem Abschnitt sollen Eingaben auf bestimmte Merkmale hin überprüft werden, indem diese Merkmale in einem regulären Ausdruck codiert werden. Idealerweise stimmt der Regex nur dann mit der Eingabe überein, wenn diese das hier interessante Merkmal aufweist. Die Probleme beim Schreiben eines solchen regulären Ausdrucks sind entweder *falsch positive* Ergebnisse (das heißt Eingaben, für die selbst dann eine Übereinstimmung ermittelt wird, wenn sie nicht das gewünschte Merkmal aufweisen) oder *falsch negative* Ergebnisse (das heißt Eingaben, für die selbst dann keine Übereinstimmung ermittelt wird, auch wenn sie das gewünschte Merkmal aufweisen).

Eine bestimmte Anzahl an falsch positiven oder falsch negativen Ergebnissen (jedoch selten beide) ist für zahlreiche Anwendungen sogar akzeptabel. In diesem Fall ist ein Kompromiss zwischen der Komplexität des nötigen Regexes und seiner Performance einerseits und der Anzahl an akzeptablen falschen Übereinstimmungen andererseits erforderlich. Damit die in diesem Artikel verwendeten regulären Ausdrücke lesbar bleiben, wird eine bestimmte Anzahl an falsch positiven, jedoch keine falsch negativen Ergebnisse akzeptiert.

Plausibilitätsprüfung von Kreditkartennummern

Bereits in Abschnitt 2.1.1, »Drei Typen von Textverarbeitungsaufgaben«, wurde die Überprüfung von Kreditkartennummern auf Plausibilität hin angesprochen. Doch was genau *ist* eine plausible Kreditkartennummer? Angenommen, eine Anwendung erfordert, dass keine ungültigen Zeichen in der Nummer enthalten, aber bestimmte Formatierungen der Nummer zugelassen sind.

In diesem Fall kann davon ausgegangen werden, dass eine gültige Kreditkartennummer ausschließlich Ziffern, Leerzeichen und Bindestriche enthält. Ein regulärer Ausdruck, der ausschließlich mit solchen Kreditkartennummern übereinstimmt, lässt sich problemlos als `(\d|\s|-)+` schreiben.

Eine gültige Kreditkartennummer besteht demnach aus einer beliebigen Kombination von Ziffern, Leerstellen und Bindestrichen. Um die Gültigkeit eines in einer Variablen `cc` gespeicherten Eingabewertes zu überprüfen, muss getestet werden, ob der reguläre Ausdruck mit dem Inhalt der Variablen `cc` übereinstimmt. In ABAP lässt sich diese Aufgabe leicht mit der statischen Methode `matches()` der Klasse `cl_abap_matcher` durchführen (siehe Listing 2.2).³

```
IF cl_abap_matcher=>matches(
    pattern = '(\d|\s|-)+'
    text      = cc
) = abap_false.
"invalid credit card number
RETURN.
ENDIF.
```

Listing 2.2 Statische Methode `matches()` der Klasse `cl_abap_matcher`

Abhängig von speziellen Anforderungen, soll jetzt möglicherweise die Anzahl an falsch positiven Ergebnissen reduziert werden, indem zusätzlich überprüft wird, ob die Eingabe exakt 15 oder 16 Stellen umfasst. Werden Leerstellen und Bindestriche zunächst verboten, genügt der einfache Regex `\d{15,16}` für die Prüfung, ob die Kartennummer die richtige Anzahl von Ziffern enthält.

Die Kombination beider Ausdrücke, um die richtige Anzahl an Ziffern zu prüfen und dabei gleichzeitig Leerstellen und Bindestriche zuzulassen, ist nicht so einfach, wie zunächst vermutet werden könnte. Ein erster trivialer Versuch `(\d|\s|-){15,16}` ist falsch, da Leerzeichen und Bindestriche hier ebenfalls als Stellen gezählt werden. Dadurch wird zum Beispiel fälschlicherweise eine Übereinstimmung für eine Zeichenfolge ermittelt, die zehn Ziffern und fünf Leerzeichen umfasst. Dies entspricht jedoch eindeutig nicht dem gewünschten Ergebnis. Um dieses Problem zu lösen, muss die Wirkung des Verkettungsoperators auf das Teilmuster `\d` beschränkt werden, während zusätzliche Leerzeichen und Bindestriche unbegrenzt zulässig sind:

`(\s|-)*(\d)(\s|-)*{15,16}`

³ Mit den Releases 7.0, EhP2 und 7.1/7.2 wurde hierfür sogar eine eingebaute Funktion `matches` eingeführt, die direkt einen Wahrheitswert zurückgibt (eine sogenannte Prädiktfunktion). Der logische Ausdruck vereinfacht sich bei ihrer Verwendung zu `IF matches(...)`.

Zur Erklärung wird mit dem ursprünglichen Ausdruck `\d{15,16}` zum Zählen der richtigen Anzahl von Ziffern begonnen, in den die hervorgehobenen Abschnitte eingefügt werden, um zusätzliche Leerzeichen und Bindestriche zuzulassen. Jede Position der Kreditkartennummer kann nun eine beliebige Anzahl an Leerzeichen und Bindestrichen enthalten.

Validierung von Dezimalzahlen

In einem weiteren Beispiel soll gezeigt werden, wie ein Regex erstellt wird, der mit Dezimalzahlen⁴ übereinstimmt. Generell besteht eine Dezimalzahl aus mehreren Ziffern, denen ein Dezimaltrennzeichen und weitere Ziffern nachgestellt sind. Es soll jedoch die Möglichkeit berücksichtigt werden, dass einige dieser Abschnitte möglicherweise nicht vorhanden sind (zum Beispiel könnten im englischen Sprachraum die Ziffern vor dem Dezimaltrennzeichen fehlen). Wird dies direkt auf einen Regex übertragen, lautet dieser:

```
\d*\.\?\d*
```

Beachten Sie, dass der Backslash-Operator als Fluchtsymbol für den Dezimalpunkt verwendet wird. Ohne diesen Operator behält der Punkt seine Bedeutung als Metazeichen bei. Wenngleich der genannte Ausdruck im Allgemeinen funktioniert, werden auch alleinstehende Dezimalpunkte (.) ermittelt, was zu einem falsch positiven Ergebnis führt. Wenn dieses fehlerhafte Ergebnis ausgeschlossen wird und auch nachgestellte Dezimalpunkte ohne Nachkommastellen (wie in 10.) nicht zugelassen werden sollen, kann der Regex folgendermaßen neu gruppiert werden: `\d*(\.\d+)?`

Durch das Ändern des zweiten Sternchenoperators in einen Plusoperator wird sichergestellt, dass dem Dezimalpunkt mindestens eine Ziffer folgt. Indem der Fragezeichenoperator an das Ende verschoben wird, werden für den gesamten Ausdruck optional auch Ganzzahlen zugelassen.

Um in ABAP mit großen Zahlen vom Typ p zu arbeiten, sollte möglicherweise auch eine Unterstützung für Tausendertrennzeichen (,) hinzugefügt werden. Eine nur vermeintliche Lösung wäre eine Lockerung der Zeicheneinschränkung für den ganzzahligen Abschnitt des Wertes, indem `\d*` durch `(\d|,)*` ersetzt wird. Dadurch stimmt der Ausdruck jedoch mit unsinnigen Zeichenfolgen, wie zum Beispiel 1, , , 0, überein.

⁴ Hier wird sich auf die englischsprachige Notation mit Dezimalpunkt und Komma als Tausendertrennzeichen bezogen, die in technischen Systemen häufig Anwendung findet.

Dies lässt sich vermeiden, wenn eine Eigenschaft für Trennzeichen gefunden wird, die für alle gültigen Ganzzahlen gilt. Gemäß Definition werden Trennzeichen durch exakt drei Ziffern getrennt. Diese Beschreibung kann weiter vereinfacht werden durch die Aussage, dass auf jedes Trennzeichen genau drei Ziffern folgen. Wird diese Eigenschaft in einem Regex codiert, lautet dieser:

```
\d{1,3}(,\d{3})*(\.\d+)?
```

Der Anfangsteil `\d{1,3}` stimmt mit bis zu drei Ziffern vor dem ersten Trennzeichen (falls vorhanden) überein, wohingegen der letzte Unterausdruck `(\.\d+)?` erneut die Nachkommastellen der Zahl abdeckt (falls vorhanden).

2.2.3 Suchen mit regulären Ausdrücken

Bisher bezogen sich die Erläuterungen auf das Übereinstimmen von regulären Ausdrücken mit Zeichenfolgen, etwa um zu prüfen, ob eine Zeichenfolge eine bestimmte Struktur aufweist. In diesem Abschnitt liegt der Schwerpunkt auf einer weiteren wichtigen Anwendungsmöglichkeit für Regexe: das Finden von Informationen durch die *Suche* nach übereinstimmenden Elementen.

Regexe werden im Allgemeinen zum Suchen verwendet, um die Position von mindestens einer Übereinstimmung für den entsprechenden Regex in einer Zeichenfolge zu ermitteln. Trotzdem wird häufig informell die Wendung »nach einem Regex suchen« verwendet, wenn tatsächlich »nach Übereinstimmungen für diesen Regex suchen« gemeint ist. Es ist daher wichtig, dass Sie den Unterschied zwischen Übereinstimmung und *Suchen* im Kopf behalten, wenn Sie den Rest dieses Kapitels lesen. Bedenken Sie zudem, dass in einigen Programmiersprachen, insbesondere Perl, der Begriff Übereinstimmung (Matching) auch für Suchvorgänge verwendet wird, was zusätzlich für Verwirrung sorgt.

Erste längste Übereinstimmungen

Was geschieht bei der Suche nach einem Regex (bzw. nach den *Übereinstimmungen* für einen Regex) in einem bestimmten Text? Zuerst werden alle möglichen Übereinstimmungen des Regexes innerhalb des Textes ermittelt. Anschließend wird in jeder Gruppe aus überlappenden Übereinstimmungen die am weitesten links beginnende Übereinstimmung als Ergebnis zurückgegeben. Ist die Übereinstimmung nicht eindeutig, so wird die längste zurückgeliefert. Als Beispiel soll mit dem Regex `a[ao]*a` in der im Folgenden gezeigten Zeichenfolge gesucht werden:

```
a[ao]*a:   ooaoaoaoXoooaaoooooao
            aooa          (1)
            aooaoa        (2)
            aoa           (3)
            aa            (4)
            aaoooooa    (5)
            aoooooaa   (6)
```

Die ersten beiden Übereinstimmungen (1 und 2) sind die am weitesten links beginnenden Übereinstimmungen in der ersten Gruppe überlappender Übereinstimmungen. Da die zweite Übereinstimmung die längere ist, wird diese als Ergebnis der Suche zurückgegeben. In der zweiten Gruppe handelt es sich bei den Übereinstimmungen 4 und 5 um die am weitesten links ermittelten Übereinstimmungen. Da die zweite Übereinstimmung die längere ist, wird diese als weiteres Ergebnis der Suche zurückgegeben. Nachdem die Ergebnisse auf diese Weise eindeutig bestimmt wurden, muss der Benutzer, der die Suche aufgerufen hat, ein Ergebnis auswählen (in diesem Fall 2 oder 5). Die meisten Suchschnittstellen geben entweder die erste längste Übereinstimmung von links an oder alternativ alle längsten Übereinstimmungen.⁵

Die Regel, dass die am weitesten links beginnende (das heißt erste) längste Übereinstimmung (Leftmost-longest-Regel) zurückgegeben wird, ist mit das natürlichste Verhalten für Suchvorgänge und wurde als POSIX-Standard definiert. Es gibt jedoch auch andere Ansätze für die Suche, die kurz in Abschnitt 2.4, »Technische Aspekte regulärer Ausdrücke«, erwähnt werden.

Beachten Sie, dass ABAP in Wirklichkeit eine intelligenterre Suche durchführt und nicht erst intern alle Übereinstimmungen berechnet, bevor die erste längste Übereinstimmung zurückgegeben wird. Es ist jedoch hilfreich, sich dieses Grundprinzip vorzustellen, wenn Sie den Suchprozess gedanklich nachvollziehen wollen.

Da bei der Suche die längste Übereinstimmung als Ergebnis zurückgegeben wird, wird der Vorgang als *gierig* bezeichnet. Dies gilt nicht nur für den gesamten Regex, sondern auch für jeden einzelnen Unterausdruck und macht sich am meisten bei Wiederholungen bemerkbar. An dieser Stelle spielt diese Tatsache noch keine besondere Rolle, es muss jedoch auf das Problem zurückgekommen werden, wenn auf Untergruppen eingegangen wird (siehe Abschnitt 2.3.1, »Arbeiten mit Untergruppen«).

⁵ Die ABAP-Anweisung FIND macht da keine Ausnahme.

Um lediglich zu prüfen, ob eine Übereinstimmung für einen Regex in einem bestimmten Text vorhanden ist, kann die statische Methode `contains()` der Klasse `cl_abap_matcher` verwendet werden (siehe Listing 2.3).⁶

```
IF cl_abap_matcher=>contains(
    pattern = '\d+'
    text     = mytext ) = abap_true.
    "match found ...
ENDIF.
```

Listing 2.3 Statische Methode `contains()` der Klasse `cl_abap_matcher`

In Abschnitt 2.2.4 erfahren Sie, wie Sie detaillierte Informationen zum übereinstimmenden Text erhalten, wie beispielsweise die Position oder die Länge der gefundenen Übereinstimmung.

Anker und Grenzen

Mit einer Handvoll Operatoren, die mit *Positionen* anstelle von Zeichen übereinstimmen, können Sie zusätzlich steuern, wo Übereinstimmungen in einem Text gefunden werden sollen.

Die *Anker* `^` und `$` entsprechen dem Anfang bzw. dem Ende einer Zeile. Folglich geben beide Methodenaufrufe in Listing 2.4 den Wert `abap_false` zurück, da `cat` sich weder am Anfang noch am Ende der Zeichenfolge `mytext` befindet. Aufgrund ihres UNIX-Ursprungs reagieren Anker ebenfalls auf in die Zeichenfolge eingebettete Zeilenumbrüche. Falls dieses Verhalten nicht erwünscht ist, kann stattdessen auf die Ankervarianten `\A` und `\z` zurückgegriffen werden, die mit dem tatsächlichen Anfang bzw. Ende der gesamten Zeichenfolge übereinstimmen.⁷

```
mytext = 'the cat sat on the mat'.
rc1 = cl_abap_matcher=>contains(
    pattern = '^cat'
    text     = mytext ).
rc2 = cl_abap_matcher=>contains(
    pattern = 'cat$'
    text     = mytext ).
```

Listing 2.4 Methodenaufrufe für Anker

⁶ Auch hierfür gibt es seit den Releases 7.0, EhP2 und 7.1/7.2 eine eingebaute Prädikatfunktion `contains`.

⁷ Die Unterschiede bei der Groß- und Kleinschreibung lassen sich damit erklären, dass es noch einen weiteren Operator `\Z` gibt, der fast die gleiche Funktion wie `\z` erfüllt, jedoch alle Zeilenumbrüche am Ende des durchsuchten Textes ignoriert.

Die *Wortgrenzen* \< und \> entsprechen dem Anfang bzw. dem Ende eines Wortes:

```
\<.at:      Cathy's cat spat at Matt
.at\>:    Cathy's cat spat at Matt
\<.at\>:  Cathy's cat spat at Matt
```

Technisch gesehen stimmen Wortgrenzen mit Positionen zwischen durch \w und \W darstellbaren Zeichenfolgen überein. Demzufolge ist das Hinzufügen von Wortgrenzen bei dem oberflächlich sehr sinnvoll erscheinenden Regex \<\w+\> aufgrund der Regel der ersten längsten Übereinstimmung eigentlich redundant.

Es gibt noch einige weitere Begrenzungsoperatoren. Der Operator \b entspricht \<|\>, passt also auf Anfang und Ende eines Wortes. Der interessantere Operator \B stimmt mit den Zwischenräumen innerhalb eines Wortes, das heißt Positionen zwischen zwei aufeinanderfolgenden Zeichen \w, überein.

Kontextbasierte Suche nach Informationen

Ein typischer Anwendungsfall für Regexe ist das kontextbasierte Suchen nach Informationen. Erinnern Sie sich an das Beispiel zur Extraktion von Informationen, in dem die Dokument-ID aus einer bestimmten URL extrahiert werden sollte.

Die Position der Informationen wird durch einige *Begrenzungsmerkmale* gekennzeichnet, das heißt eindeutige Zeichen oder Schlüsselwörter, die das Vorhandensein der gewünschten Informationen eindeutig anzeigen. In diesem Beispiel wird der Wert der Dokument-ID durch das begrenzende Schlüsselwort &docid= auf der linken Seite und das nachfolgende Zeichen & auf der rechten Seite präzise angezeigt:

<http://docserve.sap.com/serve?user=ralph&docid=NW2004&lang=EN>

Um diese Informationen schnell zu finden, könnte eine Suche mit dem Regex &docid=\w+& durchgeführt werden, wenn angenommen wird, dass die Dokument-ID nur aus Buchstaben, Ziffern und Unterstrichen besteht. Da dies aber keine Fälle abdeckt, in denen die docid das erste oder letzte Argument in der URL ist, wäre (\?|&)docid=\w+(&|\$) eine bessere Lösung.

2.2.4 Reguläre Ausdrücke in ABAP-Programmen

Nachdem Sie nun die ersten Regexe für das Abgleichen und die Suche schreiben können, wird der Schwerpunkt im Folgenden auf ABAP gelegt und beschrieben, wie Sie die Vorteile von regulären Ausdrücken in Ihrem ABAP-Code nutzen können.⁸ Um diesen Erklärungen folgen zu können, ist es hilfreich, mit den allgemeinen Textverarbeitungstechniken in ABAP vertraut zu sein. Um Ihre Kenntnisse diesbezüglich aufzufrischen, können Sie in Kapitel 1, »Effektive Zeichenkettenverarbeitung in ABAP«, nachschlagen.

Suchen und Ersetzen mit FIND und REPLACE

Die wichtigsten Anweisungen für die Textverarbeitung in ABAP sind die Anweisungen FIND und REPLACE. Ab SAP NetWeaver 7.0 bieten beide Anweisungen eine native Unterstützung für reguläre Ausdrücke. Regexe lassen sich dort einfach dadurch verwenden, indem das Schlüsselwort REGEX vor dem Suchmuster eingefügt wird.⁹

Die gefundenen Teilstufen werden durch die Zusätze MATCH LINE, MATCH OFFSET und MATCH LENGTH zurückgegeben. Um die Übereinstimmungen eines Regexes mypattern in der Zeichenfolge mytext zu finden, wird daher folgender Code verwendet:

```
DATA: length TYPE i, offset TYPE i.
FIND REGEX mypattern IN mytext
      MATCH OFFSET off MATCH LENGTH len.
IF sy-subrc = 0.
  WRITE: / 'found match of length', len,
         'at position', off.
ENDIF.
```

Beachten Sie, dass die Länge des übereinstimmenden Textes (anders als bei einer SUBSTRING-Suche, siehe Kapitel 1, »Effektive Zeichenkettenverarbeitung

-
- ⁸ Eine vollständige Referenz zur Verwendung regulärer Ausdrücke in ABAP sowie die vollständige Erklärung der dort erlaubten REGEX-Syntax finden Sie wie gewohnt in der ABAP-Schlüsselwortdokumentation. Sie brauchen nur nach dem Stichwort »regulär« zu suchen oder aus der F1-Hilfe der entsprechenden ABAP-Anweisungen zu den regulären Ausdrücken zu navigieren.
 - ⁹ Die Anweisungen FIND und REPLACE wurden zu Release 7.0 zudem im Hinblick auf die Verarbeitung von internen Tabellen und die Harmonisierung von Zusätzen überarbeitet: In internen Tabellen mit zeichenartigem Zeilentyp kann über den Zusatz IN TABLE zeilenweise gesucht und ersetzt werden. Zur FIND-Anweisung wurde ein neuer ALL OCCURRENCES OF-Zusatz hinzugefügt, der im Wesentlichen dem gleichlautenden Zusatz der REPLACE-Anweisung entspricht.

in ABAP») von der Länge des Suchmusters abweichen kann. Um die tatsächliche übereinstimmende Zeichenfolge abzurufen, kann der für ABAP typische Offset-/Längenzugriff verwendet werden:¹⁰

```
DATA text_matched TYPE string.  
text_matched = mytext+off(len).
```

Um nach *allen* Übereinstimmungen innerhalb des Textes zu suchen, kann der Zusatz ALL OCCURRENCES OF der FIND-Anweisung verwendet werden. Dann können Sie mit dem Zusatz MATCH COUNT die Anzahl an ermittelten Übereinstimmungen abrufen. Mit den anderen Zusätzen MATCH ... werden jedoch nur Informationen zur *letzten* gefundenen Übereinstimmung zurückgegeben. Die vollständige Liste der Übereinstimmungen kann mit dem Zusatz RESULTS abgerufen werden, der eine interne Tabelle vom Dictionary-Typ match_result_tab mit dem Zeilentyp match_result füllt (siehe Tabelle 2.3), wobei »Ersetzung« sich auf die Verwendung mit REPLACE bezieht.

Komponente	Typ	Bedeutung
line	i	Zeile der gefundenen Übereinstimmung/durchgeführten Ersetzung in internen Tabellen (anderenfalls 0)
offset	i	Position der gefundenen Übereinstimmung/durchgeführten Ersetzung
length	i	Länge der gefundenen Übereinstimmung/durchgeführten Ersetzung
submatches	TABLE OF submatch_result	interne Tabelle für Untergruppen (siehe Abschnitt 2.3.1, »Arbeiten mit Untergruppen«)

Tabelle 2.3 Struktur match_result

Für jede gefundene Übereinstimmung fügt die FIND-Anweisung eine neue Zeile vom Dictionary-Typ match_result in die Tabelle ein. Um auf diese Informationen zuzugreifen, wird in der Regel eine LOOP-Anweisung verwendet (siehe Listing 2.5).

```
DATA res          TYPE match_result_tab.  
FIELD-SYMBOLS <m> TYPE match_result.  
FIND ALL OCCURRENCES OF REGEX mypattern IN mytext RESULTS res.
```

¹⁰ Oder Sie arbeiten gleich mit Untergruppen (siehe Abschnitt 2.3.1 »Arbeiten mit Untergruppen«).

```
LOOP AT res ASSIGNING <m>.
  WRITE / mytext+<m>-offset(<m>-length).
ENDLOOP.
```

Listing 2.5 LOOP-Anweisung für den Zugriff auf Informationen

Der Zusatz RESULTS akzeptiert auch flache Strukturen vom Typ match_result, jedoch bleiben wie auch bei den MATCH ...-Zusätzen lediglich die Informationen zur letzten Übereinstimmung erhalten. Die Verwendung von RESULTS mit Strukturen ist daher nur dafür vorgesehen, um bei der Verwendung von FIRST OCCURRENCE auf die Informationen zu Untergruppen zuzugreifen (weitere Informationen hierzu finden Sie in Abschnitt 2.3.1, »Arbeiten mit Untergruppen«).

Der aus Kapitel 1, »Effektive Zeichenkettenverarbeitung in ABAP«, bekannte Zusatz ACCEPTING|IGNORING CASE kann natürlich auch mit Regexen verwendet werden, um bei der Suche in einem Text die Groß- und Kleinschreibung zu berücksichtigen. Der Zusatz IN BYTE MODE ist bei der Angabe REGEX dagegen ausgeschlossen, da binäre Muster von regulären Ausdrücken nicht unterstützt werden.

Alle bisherigen Erklärungen zu FIND gelten gleichermaßen für die REPLACE-Anweisung. In diesem Fall heißen die entsprechenden Zusätze jedoch REPLACEMENT LINE, REPLACEMENT OFFSET, REPLACEMENT LENGTH und REPLACEMENT COUNT.

Weitere Informationen zu den Anweisungen FIND und REPLACE sowie einige Empfehlungen zur Verwendung mit Zeichenfolgen finden Sie in Kapitel 1.

Reguläre Ausdrücke und ABAP Objects

Für moderne ABAP-Programme wird eine objektorientierte Methodologie als bevorzugtes Programmiermodell empfohlen. Um Regexe in objektorientierten Programmen natürlich verwenden zu können, bietet ABAP die beiden Klassen c1_abap_regex und c1_abap_matcher an.

Die Regex-Klasse c1_abap_regex speichert den eigentlichen regulären Ausdruck in einem internen, vorkompilierten Format (weitere Informationen zur Regex-Erstellung finden Sie in Abschnitt 2.4.2, »Performancekontrolle«). Die Matcher-Klasse c1_abap_matcher verknüpft ein c1_abap_regex-Objekt mit einem Text und dient als zentrale Schnittstelle für die weitere Verarbeitung. Insbesondere bietet diese Klasse Methoden zur sequenziellen Suche, Abfrage und Ersetzung einzelner Übereinstimmungen (*Match-by-Match*).

Um einen Text zu verarbeiten, erstellen Sie zunächst ein Regex-Objekt mit der Anweisung CREATE OBJECT (siehe Listing 2.6).

```

DATA myregex TYPE REF TO cl_abap_regex.
CREATE OBJECT myregex
  EXPORTING
    pattern      = 'a*b'
    ignore_case = abap_true.

```

Listing 2.6 Regex-Objekt mit CREATE OBJECT erstellen

Der optionale Parameter `ignore_case` steuert, ob die Groß- und Kleinschreibung bei der Ermittlung der Übereinstimmungen berücksichtigt werden soll oder nicht. Bei fehlender Angabe wird die Groß- und Kleinschreibung berücksichtigt. Es sind weitere optionale Parameter verfügbar, von denen einige im Folgenden noch beschrieben werden.

Anschließend erstellen Sie ein Matcher-Objekt, um den neuen Regex mit dem zu verarbeitenden Text zu verknüpfen (siehe Listing 2.7).

```

DATA: mytext      TYPE string,
      mymatcher TYPE REF TO cl_abap_matcher.
CREATE OBJECT mymatcher
  EXPORTING regex = myregex
            text   = mytext.

```

Listing 2.7 Matcher-Objekt erstellen

Ein Regex-Objekt kann in beliebig vielen Matcher-Objekten wiederverwendet werden. Aus Gründen der Performance sollte ein mehrfach verwendeter Regex deshalb auch eher als Regex-Objekt erzeugt werden, als immer wieder in Anweisungen oder eingebauten Funktionen (ab den Releases 7.0, EhP2 und 7.1/7.2) neu kompiliert zu werden.

Alternativ bietet die Matcher-Klasse auch eine Factory-Methode für die handliche Erzeugung von Regex und Matcher in einem einzigen Schritt, die dann jedoch keine gemeinsame Nutzung von Regex-Objekten zulässt:

```

mymatcher = cl_abap_matcher->create(
  pattern = 'a*b'
  text   = mytext ).

```

Das Matcher-Objekt speichert eine interne Kopie des zu verarbeitenden Textes, sodass der Inhalt einer Variablen `mytext`, die an den Konstruktor übergeben wird, nicht von nachfolgenden Ersetzungsvorgängen geändert wird. Auf das Ergebnis von Ersetzungen muss in diesem Fall über `mymatcher->text` anstelle von `mytext` zugegriffen werden. Wenn `mytext` vom Typ `string` ist, verursacht das Kopieren von `mytext` in das Objekt dabei zunächst kaum Kosten, da hier

das aus Kapitel 1, »Effektive Zeichenkettenverarbeitung in ABAP«, bekannte Sharing greift. Erst ein Schreibzugriff auf diese Zeichenfolge durch einen Ersetzungsvorgang führt dazu, dass die gemeinsame Verwendung aufgehoben wird, was dann ein potenziell kostenintensiver Vorgang sein kann.

Die Matcher-Klasse bietet eine Vielfalt von Methoden zum Suchen, Abfragen und Ersetzen von Übereinstimmungen (siehe Tabelle 2.4).

Suchmethoden	Abfragemethoden	Ersetzungsmethoden
find_next()	get_match()	replace_found()
match()	get_line()	replace_next()
	get_offset()	
find_all()	get_length()	replace_all()
	get_submatch()	

Tabelle 2.4 Methoden von cl_abap_matcher

Die Interaktion mit einem Matcher-Objekt basiert auf dem Konzept der *aktuellen Übereinstimmung*, das heißt der letzten während des Suchvorgangs gefundenen Übereinstimmung. Eine aktuelle Übereinstimmung wird durch die Methode `find_next()` bestimmt und von der Methode `replace_found()` zerstört. Die Abfragemethoden erfordern eine aktuelle Übereinstimmung, haben jedoch keine Auswirkung auf diese.

Anfänglich ist keine aktuelle Übereinstimmung vorhanden, daher muss zunächst `find_next()` aufgerufen werden. Sobald eine aktuelle Übereinstimmung vorhanden ist, können Sie diese mit den Abfragemethoden untersuchen, um zum Beispiel Informationen wie Offset und Länge zu erhalten. Durch das erneute Aufrufen von `find_next()` wird mit der nächsten Übereinstimmung fortgefahren (falls vorhanden). Das Ergebnis des Aufrufs `find_next()` wird als Boole'scher Wert¹¹ zurückgegeben. Nachdem alle Übereinstimmungen ermittelt wurden, wird die aktuelle Übereinstimmung invalidiert, und weitere Aufrufe von `find_next()` schlagen mit dem Wert `abap_false` fehl. Listing 2.8 zeigt, wie dieses Muster verwendet werden kann, um alle Ganzzahlen zu addieren, die in einem Text enthalten sind.

```
DATA: sum      TYPE i,
      mymatch  TYPE match_result,
      mymatcher TYPE REF TO cl_abap_matcher.
```

¹¹ Natürlich in der ABAP-typischen Ausprägung `abap_bool`.

```

mymatcher = cl_abap_matcher->create(
    pattern = '\d+' text = mytext ).
WHILE mymatcher->find_next( ) = abap_true.
    mymatch = mymatcher->get_match( ).
    sum = sum + mymatcher->text+mymatch-offset(mymatch-length).
ENDWHILE.

```

Listing 2.8 Summierung aller in einem Text enthaltenen Ganzzahlen

Jedes Mal, wenn der Matcher eine aktuelle Übereinstimmung gespeichert hat, kann die Methode `replace_found()` aufgerufen werden, um diese aktuelle Übereinstimmung durch einen neuen Text zu ersetzen. Dadurch wird die aktuelle Übereinstimmung zerstört, sodass alle darauffolgenden Aufrufe einer Abfragemethode eine Ausnahme auslösen. Durch das erneute Aufrufen von `find_next()` wird jedoch die nächste nicht verarbeitete Übereinstimmung ermittelt und als aktuelle Übereinstimmung gesetzt.

Indem Sie abwechselnd `find_next()` und `replace_found()` aufrufen, können Sie die Übereinstimmungen im Text problemlos durchlaufen und für die einzelnen Übereinstimmungen Aktionen ausführen. Dadurch haben Sie die Möglichkeit, zum Beispiel sämtliche Ganzzahlen in einem Textdokument um den Wert 1 zu erhöhen, da der Ersetzungstext berechnet werden kann, *nachdem* die Übereinstimmung ermittelt wurde (siehe Listing 2.9).

```

DATA: value      TYPE i,
      value_text TYPE string,
      mymatch    TYPE match_result,
      mymatcher  TYPE REF TO cl_abap_matcher.
mymatcher = cl_abap_matcher->create(
    pattern = '\d+' text = mytext ).
WHILE mymatcher->find_next( ) = abap_true.
    mymatch = mymatcher->get_match( ).
    value = mymatcher->text+mymatch-offset(mymatch-length) + 1.
    value_text = value.
    mymatcher->replace_found( value_text ).
ENDWHILE.

```

Listing 2.9 Erhöhen aller Ganzzahlen in einem Text um den Wert 1

Abbildung 2.1 zeigt die Statusänderungen eines Matcher-Objektes während eines typischen Nutzungszyklus.

Die Matcher-Klasse bietet zum Suchen und Ersetzen noch einige weitere Methoden an. Die Methode `replace_next()` ist eine Kurzform von `find_next()`, unmittelbar gefolgt von `replace_found()`. Die Methode erfordert daher keine aktuelle Übereinstimmung.

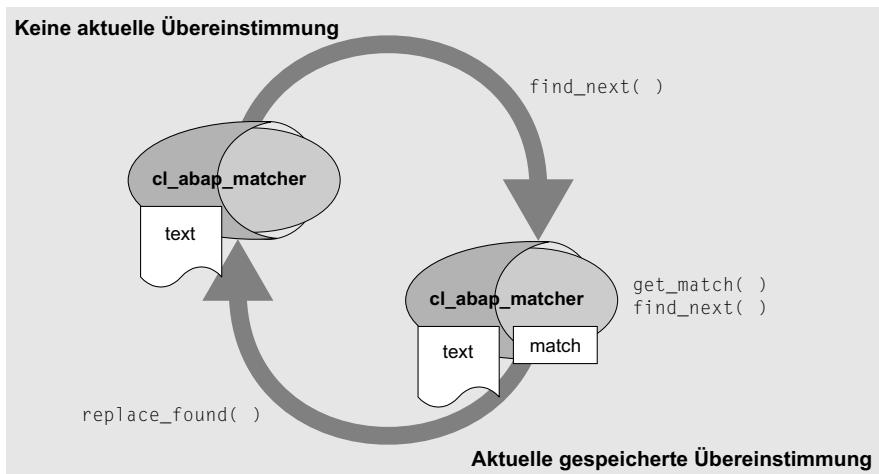


Abbildung 2.1 Typische Verwendung eines cl_abap_matcher-Objektes

Die Methoden `find_all()` und `replace_all()` werden für alle noch übrigen Übereinstimmungen im Text ausgeführt. Die Aufrufe beider Methoden invalidieren zudem die aktuelle Übereinstimmung. Die Methode `match()` vergleicht den Regex mit dem übrigen Text, der noch nicht verarbeitet wurde. Zwar wird die Methode `match()` in der Regel unmittelbar nach der Erzeugung des Matchers aufgerufen, um eine Übereinstimmung für den gesamten Text zu ermitteln, jedoch ist dies nicht zwingend erforderlich.

Statische Methoden

Die statischen Methoden `matches()` und `contains()` wurden bereits in der Einführung in reguläre Ausdrücke erwähnt. Beide statischen Methoden sind für die Verwendung in bedingten Anweisungen als Ersatz für Zeichenfolgenoperatoren wie `CS` und `CP` bestimmt.

Sehr häufig wird lediglich die Information benötigt, ob für einen Regex überhaupt Übereinstimmungen in einem Text vorhanden sind. Es ist jedoch auch möglich, weitere Informationen zu dieser Übereinstimmung abzurufen, wie es auch bei der älteren Vorgehensweise mit Operatoren wie `CS` durch Verwendung von `sy-fdpos` möglich war. Hierzu rufen Sie die statische Methode `get_object()` auf, um ein Matcher-Objekt zu erhalten, das den Status der vorhergehenden Operation `matches()` oder `contains()` darstellt. Anschließend können Sie ganz normal die eingeführten Abfragemethoden einsetzen, um weitere Informationen zu diesem Objekt zu erhalten (siehe Listing 2.10).

```

IF cl_abap_matcher=>contains(
    pattern = mypattern
    text     = mytext      ) = abap_true.
mymatcher =
    cl_abap_matcher=>get_object( ).
off = mymatcher->get_offset( ).
WRITE: / 'found match at offset', off.
ENDIF.
```

Listing 2.10 Informationen zum Objekt abfragen

Theoretisch ist es sogar möglich, die Methoden zum Suchen und Ersetzen für dieses so erhaltene Objekt aufzurufen. Diese Vorgehensweise wird allerdings als schlechter Programmierstil angesehen und deshalb nicht empfohlen.

Hinweis

Ab den Releases 7.0, EhP2 und 7.1/7.2 enthält ABAP einen großen Satz eingebauter Zeichenkettenfunktionen. Von diesen Funktionen nehmen wiederum viele Regexe als Parameter entgegen. Beispiele sind matches, contains, find, replace und match. Diese Funktionen können statt der Methoden der genannten Klassen verwendet werden, wenn es um einzelne kontextfreie Abgleiche geht, in denen die sonstige hier aufgeführte Funktionalität nicht benötigt wird.

2.2.5 Reguläre Ausdrücke außerhalb von ABAP-Programmen

Reguläre Ausdrücke können auch außerhalb von ABAP-Programmen getestet und verwendet werden. Ab SAP NetWeaver 7.0 bietet der neue ABAP Debugger in zahlreichen Suchdialogfenstern Unterstützung für Regexe. Die Unterstützung in weiteren Werkzeugen ist für zukünftige Releases geplant.

Im Folgenden werden das Regex Toy und der Code Inspector als zwei Beispiele vorgestellt, die nützliche Regex-Funktionalität bieten.

Regexe testen mit dem Regex Toy

SAP NetWeaver 7.0 bietet ein nützliches kleines Programm namens DEMO_REGEX_TOY, mit dem Entwickler ihre regulären Ausdrücke schnell und mühe-los testen können.¹² Das interaktive Regex Toy zeigt an, ob und an welchen Positionen Übereinstimmungen für reguläre Ausdrücke in einem vom Benutzer interaktiv bereitgestellten Text gefunden werden. Nach der Eingabe eines Rege-

¹² Falls dieses Programm in Ihrer Installation nicht vorhanden ist, müssen Sie das System auf das Support Package 7 aktualisieren. Ab den Releases 7.0, EhP2 und 7.1/7.2 gibt es auch ein kleineres Programm DEMO_REGEX, das für einfache Abgleiche ausreichend ist.

xes und eines Textes hebt das Programm die erste Übereinstimmung bzw. alle Übereinstimmungen für diesen Regex, wie in Abbildung 2.2 gezeigt, im Text hervor. Im unteren Bildschirmgebiet werden die Untergruppen (siehe Abschnitt 2.3.1, »Arbeiten mit Untergruppen«) für die erste ermittelte Übereinstimmung angezeigt. Wenn Sie die Option REPLACE auswählen, können Sie die ermittelten Übereinstimmungen zudem durch einen beliebigen Text ersetzen.

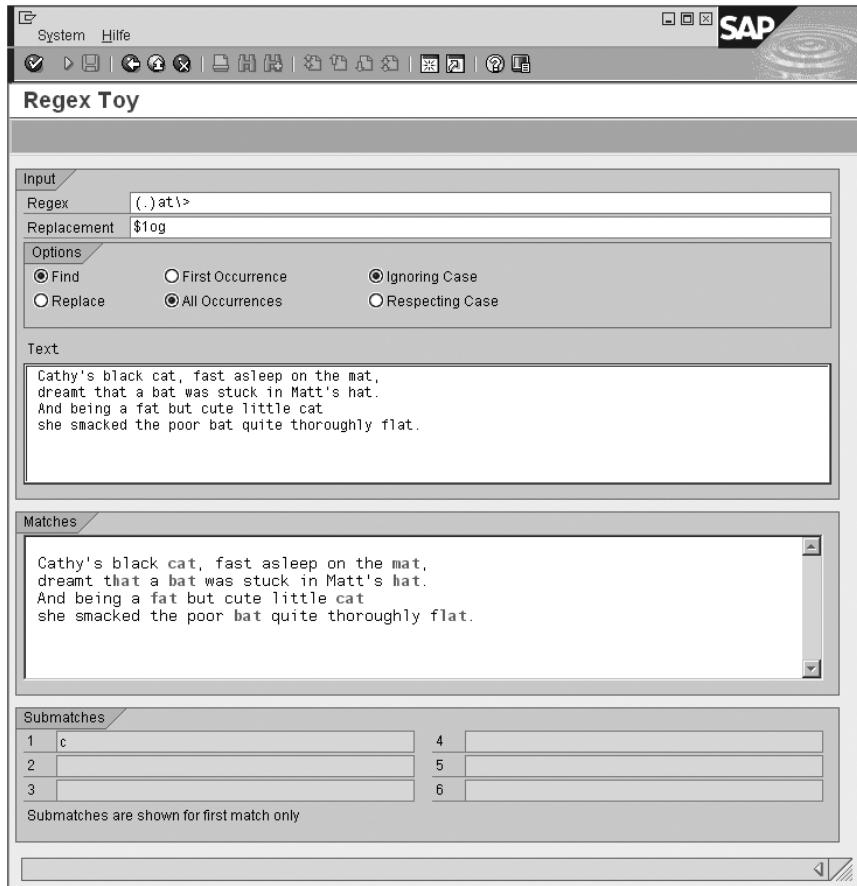


Abbildung 2.2 Regex Toy

Analyse von ABAP-Programmen mit dem Code Inspector

Vielleicht haben Sie in Ihrer Laufbahn als ABAP-Entwickler schon einmal Code geschrieben, den Sie im Nachhinein lieber nicht zur Benutzung freigegeben hätten. Oder Sie möchten die Qualität von übernommenem Code verbessern, indem Sie nach Codemustern suchen, die die Performance verschlechtern oder Sicherheitsrisiken darstellen.

Für die LOOP AT-Anweisung gibt es beispielsweise zwei Hauptvarianten, die die Zusätze INTO wa und ASSIGNING <fs> verwenden. Die Letztgenannte ist im Allgemeinen vorzuziehen, da der Zugriff auf die aktuelle Zeile der durchlaufenen Tabelle nicht über einen Wertetransport, sondern effizient über eine Referenz erfolgt. Die Aufrufe von Systemfunktionen über CALL name können dagegen zu Sicherheitsproblemen führen, wenn sie nicht ordnungsgemäß durch ABAP-Code mit entsprechenden Authentifizierungsprüfungen geschützt werden.

Angenommen, Sie möchten nun anhand dieser Informationen Ihren Code verbessern und eine einfache Codeprüfung durchführen – wie sollten Sie vorgehen, um die relevanten Anweisungen zu finden? Der einfache Suchdialog des klassischen ABAP Editors weist diverse Einschränkungen auf, die den Nutzen dieses Dialogs für die vorliegende Aufgabe erheblich reduzieren:

- ▶ Die Suche erfolgt zeilenweise, sodass bei der Suche nach dem Muster LOOP*INTO keine mehrzeiligen LOOP-Anweisungen ermittelt werden.
- ▶ Doppelpunkt-Komma-Ausdrücke ändern den syntaktischen Aufbau von Anweisungen, sodass geschachtelte LOOP-Anweisungen hinter einem LOOP AT: nicht gefunden werden.
- ▶ Kommentare und Zeichenliterale enthalten möglicherweise die Suchbegriffe, was zu falsch positiven Ergebnissen führt.
- ▶ Negative Suchvorgänge das heißt eine Suche nach Wörtern, die nicht mit einem bestimmten Suchbegriff übereinstimmen, werden nicht unterstützt. Dies ist jedoch im zweiten Beispiel erforderlich (CALL name), um Anweisungen wie CALL SCREEN, CALL FUNCTION oder CALL METHOD auszuschließen, die in der Regel keine Gefahren darstellen.¹³

Der Code Inspector ist ein mächtiges Werkzeug zur Quellcodeanalyse und genau richtig für diese Art von Aufgabe. Das Programm ist im Lieferumfang sämtlicher Releases von SAP NetWeaver enthalten und löst ältere Werkzeuge wie die ohnehin nur zur internen Verwendung vorgesehene Transaktion SAMT ab. Der Code Inspector ist von keinem der genannten Probleme betroffen: Er verarbeitet die Anweisungen in einem normalisierten Quellcode nacheinander, das heißt mit aufgelösten Doppelpunkt-Komma-Ausdrücken, und ermöglicht den Ausschluss von Kommentaren. Eine ausführliche Einführung in das Werkzeug Code Inspector ist an dieser Stelle nicht möglich (siehe hierzu Kapitel 10, »Qualitätsüberprüfung mit dem Code Inspector« in Band 1 von *ABAP – Fortgeschrittene Techniken und Tools*), doch soll trotzdem anhand eines

¹³ Beachten Sie, dass bei der klassischen Suche nach CALL ' zwar Systemaufrufe ermittelt werden, ohne über CALL SCREENS und ähnliche Elemente zu stolpern; durch Variablen oder Konstanten definierte Systemaufrufe werden allerdings möglicherweise übersehen.

vollständigen Beispiels gezeigt werden, wie Sie sich dort ab den Releases 7.0, EhP2 und 7.1/7.2 Regexe zunutze machen können.

Nachdem Sie den Code Inspector mit Transaktion SCI oder über den Menüeintrag im ABAP Editor gestartet haben, definieren Sie zunächst einen neuen Objektsatz REGEX_DEMO, der sämtliche Programme enthält, die Sie untersuchen möchten. Anschließend erstellen Sie eine neue Prüfvariante, mit der alle Objekte innerhalb dieses Objektsatzes durchsucht werden (siehe Abbildung 2.3).

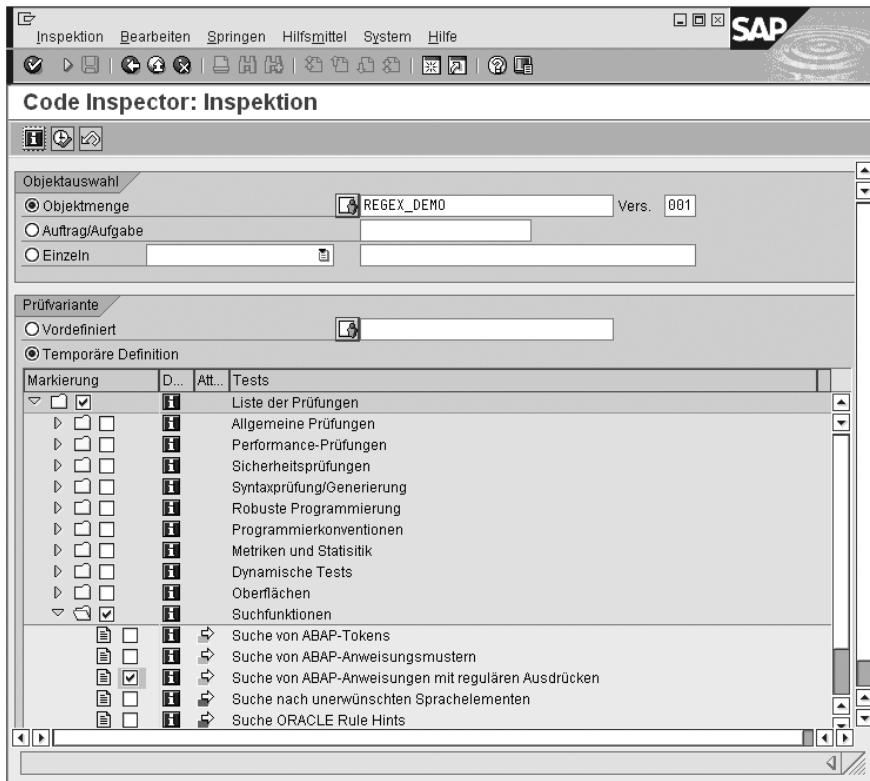


Abbildung 2.3 Erstellen einer neuen Inspektion im Code Inspector

Es stehen zahlreiche vordefinierte Analysen zur Auswahl.¹⁴ Zum Zweck dieser einmaligen Ausführung entscheiden Sie sich jedoch für eine temporäre Definition.¹⁵ Nachdem Sie den Teilbaum SUCHFUNKTIONEN aufgeklappt haben, klicken

-
- 14 Dank der umfangreichen vorhandenen Analysen ist es nur selten erforderlich, von Grund auf neue Analysen für den Code Inspector zu schreiben. Bei speziellen Anforderungen finden Sie in der Dokumentation zum Code Inspector weitere Informationen.
 - 15 Hierfür müssen Sie im Einstiegsbild des Code Inspectors das Eingabefeld für den Namen der Inspektion leer lassen und ANLEGEN auswählen.

Sie auf den kleinen Pfeil neben dem seit den Releases 7.0, EhP2 und 7.1/7.2 vorhandenen Element SUCHE VON ABAP-ANWEISUNGEN MIT REGULÄREN AUSDRÜCKEN, um das in Abbildung 2.4 gezeigte Dialogfenster für die Regex-Suche zu öffnen.

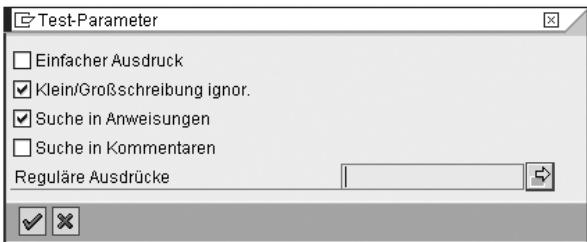


Abbildung 2.4 Dialogfenster für die Regex-Suche

Stellen Sie sicher, dass die Option SUCHE IN ANWEISUNGEN aktiviert ist, und geben Sie den folgenden Regex in das Texteingabefeld ein:

```
(\<LOOP\s+AT\>(!.+|\<ASSIGNING\>))|  
(\<CALL\s+(?!FUNCTION|METHOD|SCREEN))
```

Dieses Muster sucht nach LOOP- und CALL-Anweisungen, die auf die Beschreibung der fraglichen Programmierkonstrukte passen. Um die Genauigkeit der Übereinstimmungen zu erhöhen, werden viele Wortgrenzen und Leerzeichensmengen verwendet. Der neue Operator (!...), der hier bereits integriert wurde, wird als *negative Vorausschaubedingung* bezeichnet. Dieser Operator ermittelt eine Übereinstimmung, wenn der Text, der von dem in den Klammern angegebenen Teilausdruck repräsentiert wird, *nicht* gefunden wird. Dieser nützliche Operator wird in Abschnitt 2.3.2, »Weitere Operatoren und zukünftige Erweiterungen«, detailliert beschrieben. Das Regex-Eingabefeld kann erweitert werden, um Ausnahmen und Alternativen anzugeben. Abgesehen von einer verbesserten Lesbarkeit, wird dadurch jedoch keine neue Funktionalität zur Suche hinzugefügt, da diese Kombination von Regexen bereits durch Regexe selbst ausgedrückt werden kann.

Nachdem Sie das Dialogfenster über den Eingabeknopf (✓) geschlossen haben, kann der Inspektionslauf gestartet werden. Wenn Sie im oberen Bildschirmbildbereich auf den Knopf zum Ausführen klicken oder die Taste F8 drücken, führt das System den Lauf durch. Nach kurzer Zeit sollten eine Meldung über die erfolgreiche Ausführung und ein neuer Knopf zur Anzeige der Ergebnisse erscheinen (siehe Abbildung 2.5).

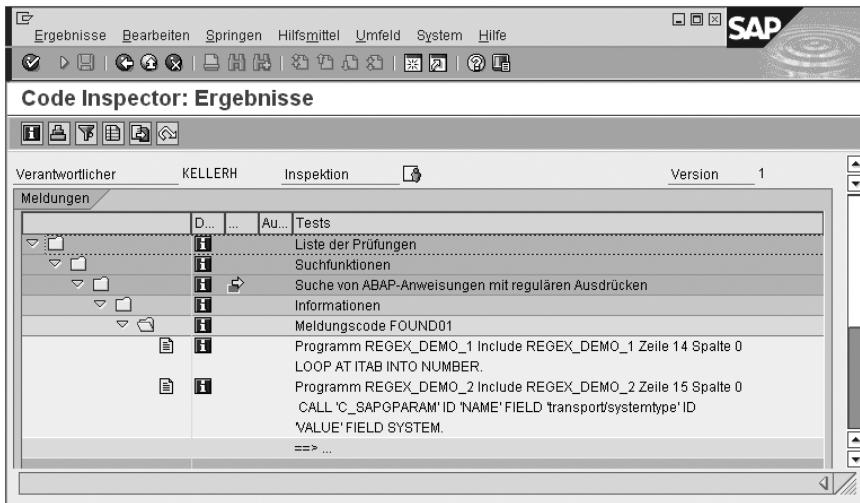


Abbildung 2.5 Ergebnisse der Inspektion

In diesem Beispielauf hat der Code Inspector zwei Codezeilen ermittelt, die mit den Suchkriterien übereinstimmen. Durch Doppelklicken auf die Ergebniszellen lässt sich der Quellcode anzeigen, um zu überprüfen, ob und wie sich der Code verbessern lässt.

2.3 Fortgeschrittene Eigenschaften regulärer Ausdrücke

Nach der vorangegangenen Einführung in die elementaren Eigenschaften von regulären Ausdrücken und ihre Verwendung sowohl in ABAP-Programmen als auch in Werkzeugen des SAP NetWeaver Application Servers ABAP wird nun auf einige fortgeschrittene Aspekte dieses mächtigen Konzeptes eingegangen.

2.3.1 Arbeiten mit Untergruppen

Untergruppen gehören zu den nützlichsten Aspekten von regulären Ausdrücken überhaupt. Um reguläre Ausdrücke effektiv zu nutzen, müssen Sie mit dem Konzept der Untergruppen vertraut sein.

Um die wichtigsten Eigenschaften von Untergruppen kurz vorwegzunehmen: Die bereits bekannten runden Klammern strukturieren nicht nur Ihren Regex, sondern speichern zudem eine interne Kopie der Unterfolge, die mit dem eingeschlossenen regulären Ausdruck übereinstimmt, intern ab. Diese sogenannten *Untergruppen* können anschließend verwendet werden, um Teile der untersuchten Zeichenfolge

- ▶ kontextbasiert zu extrahieren oder
- ▶ wertebasiert zu transformieren.

Informationen zu den abgespeicherten Untergruppen können mit den Anweisungen `FIND` und `REPLACE` sowie mit `c1_abap_matcher` abgefragt werden. Darüber hinaus kann sowohl in regulären Ausdrücken als auch in Ersetzungstexten auf diese Untergruppen zugegriffen werden.

Untergruppen

Mithilfe von Klammern lassen sich reguläre Ausdrücke in *Unterausdrücken* strukturieren. Diese Unterausdrücke werden von links nach rechts nummeriert (einsetzend mit 1) und beginnen mit einer öffnenden Klammer. Auch eine Schachtelung ist möglich:

```
a+(b+)(c(e+)|d(e+))f
<--><----->
 1   2
      <-->  <-->
    3       4
```

Wenn Sie Übereinstimmungen für diesen Regex innerhalb des Textes `aabbdeef` ermitteln, erhalten Sie als Ergebnis die folgenden Übereinstimmungen:

Text:	...aabbdeef...
Übereinstimmung:	aabbdeef
Untergruppe 1:	bbb
Untergruppe 2:	dee
Untergruppe 3:	(leer)
Untergruppe 4:	ee

Die Übereinstimmungen mit den Unterausdrücken werden als Untergruppen abgespeichert. Um auf diese Untergruppen zuzugreifen, kann der Zusatz `SUBMATCHES` der Anweisung `FIND` verwendet werden:

```
FIND REGEX 'a+(b+)(c(e+)|d(e+))f'
IN 'aabbdeef'
SUBMATCHES s1 s2 s3 s4 s5.
```

Diese Anweisung speichert die Werte `bbb`, `dee` und `ee` in den Variablen `s1`, `s2` und `s4` und initialisiert die Variablen `s3` und `s5`.

Der Zusatz `SUBMATCHES` füllt somit alle angegebenen Variablen mit der entsprechenden Zeichenfolge der Untergruppe. Falls weniger Unterausdrücke als Variablen vorhanden sind, werden die überzähligen Variablen initialisiert (wie `s5` im Beispiel). Das Gleiche gilt für Variablen, für deren Unterausdrücke keine Übereinstimmungen gefunden wurden (wie `s3` im Beispiel).

Um weitergehende Informationen zu erhalten, können Sie den Zusatz `RESULTS` verwenden, der, wie bereits in Abschnitt 2.2.4, »Reguläre Ausdrücke in ABAP-Programmen«, erwähnt, Ergebnisse der Typen `match_result` oder `match_result_tab` zurückgibt und auf deren Komponente `submatches` zugreift. Diese Komponente ist eine interne Tabelle vom Dictionary-Typ `submatch_result_tab`, deren Zeilen vom Typ `submatch_result` den Offset und die Länge für jede Untergruppe enthalten.

Alternativ kann auch wieder die Klasse `cl_abap_matcher` aus Abschnitt 2.2.4, »Reguläre Ausdrücke in ABAP-Programmen«, mit ihren Abfragemethoden eingesetzt werden, um dieselben Informationen zu erhalten (siehe Listing 2.11).

```
DATA: mymatcher TYPE REF TO cl_abap_matcher,
      str      TYPE string,
      off      TYPE i,
      len      TYPE i.
mymatcher =
  cl_abap_matcher->create_matcher(
    pattern = 'a+(b+)(c(e+)|d(e+))f'
    text     = 'aabbbdeef' ).
```

`mymatcher->find_next().`
`str = mymacher->get_submatch(1).`
`off = mymacher->get_offset(2).`
`len = mymacher->get_length(4).`

Listing 2.11 Klasse `cl_abap_match`

Die Methode `get_submatch()` gibt ähnlich wie der Zusatz `SUBMATCHES` einfach die Zeichenfolge der *nten* Untergruppe zurück. Wenn Sie eine leere Untergruppe angeben, wird auch eine leere Zeichenfolge zurückgegeben. Übergeben Sie jedoch den Index einer *nicht vorhandenen* Gruppe, wird der Ausnahmefehler `cx_sy_invalid_submatch` ausgelöst. Der spezielle Index 0 bezieht sich auf die gesamte Übereinstimmung, sodass über `get_submatch(0)` der Text der gesamten Übereinstimmung abgerufen werden kann.

Die übrigen Abfragemethoden aus vorangegangener Tabelle 2.4 unterstützen ebenfalls einen optionalen `index`-Parameter, der die Zeile, den Offset oder die Länge der entsprechenden Untergruppe zurückgibt.

Wenngleich das Konzept der Untergruppen bisher recht unkompliziert ist, gibt es einige nennenswerte Feinheiten bei deren Verwendung innerhalb von Wiederholungen. Genau wie die wiederholten Unterausdrücke in jeder Iteration neu angewendet werden, wird auch jede potenzielle Untergruppe bei jeder Iteration neu gefüllt. Bei der Anwendung des Regexes

```
(([ab])|([cd])|([ef]))*
<----->
1      <----> <----> <---->
2          3          4
```

auf den Text `aebe` werden zum Beispiel für die Untergruppen 1 bis 4 die Übereinstimmungen `e`, `b`, (`empty`) bzw. `e` zurückgegeben, da jede Gruppe den letzten übereinstimmenden Zeichenfolgenwert beibehält.

Ein weiterer interessanter Aspekt von Untergruppen betrifft die »Gierigkeit« von regulären Ausdrücken, die auch für Unterausdrücke gilt. Während die Regel zur ersten längsten Übereinstimmung (siehe Abschnitt 2.2.3, »Suchen mit regulären Ausdrücken«) die zurückzugebende Übereinstimmung bei der Suche nach Regexen eindeutig definiert, bleibt zunächst unklar, wie Untergruppen bestimmt werden sollen. Betrachten Sie zum Beispiel folgende Anweisung:

```
FIND REGEX '([ab]+)([bc]+)' IN 'ooaabbcoco'
SUBMATCHES s1 s2.
```

Die Regel für die erste längste Übereinstimmung legt nicht fest, ob `s1` der Wert `aa`, `aab` oder `aabb` zugewiesen wird. Da die Gierigkeit jedoch auch für Unterausdrücke, und zwar von links nach rechts, gilt, wird der Variablen `s1` der Wert `aabb` zugeordnet, und `s2` erhält den Restwert `cc`. Es ist jedoch wichtig zu verstehen, dass trotz gieriger Unterausdrücke primär der gesamte Regex übereinstimmen muss. Vergleichen Sie das vorhergehende Beispiel mit dem folgenden:

```
FIND REGEX '(a+)(ab)' IN 'ooaabboo'
SUBMATCHES s1 s2.
```

Hier erhält trotz Gierigkeit `s1` nicht `aa` zugewiesen, da sonst der gesamte Regex nicht mehr übereinstimmen würde. Tatsächlich erfordert die einzige mögliche Übereinstimmung für den gesamten Regex, dass `s1` den Wert `a` und `s2` den Wert `ab` erhält.

Diese Analyse mag Ihnen zunächst recht theoretisch erscheinen. Es ist aber hilfreich, an dieses Verhalten zu denken, wenn Sie Untergruppen zum Extrahieren von Informationen verwenden, was im Folgenden vorgestellt wird.

Extrahieren von Teilinformationen mit Untergruppen

Im ersten Teil dieses Kapitels wurde gezeigt, wie Sie reguläre Ausdrücke für die kontextbasierte Suche nach Informationen verwenden (siehe Abschnitt 2.2.3, »Suchen mit regulären Ausdrücken«). Häufig sind diese Informationen jedoch zusätzlich unterstrukturiert und lediglich bestimmte Teile von ihnen interessant.

Im Folgenden wird dies wieder anhand eines Beispiels gezeigt: Angenommen, Sie schreiben eine Webanwendung, die eine URL wie die folgende mit den Anmeldeinformationen des Benutzers erhält:¹⁶

```
http://sap.com/handler?sess=1&login=ralph@geheim&lang=EN
```

Die Anmeldeinformationen werden über den `login`-Parameter angegeben, der Benutzername und Kennwort enthält, die durch ein @-Zeichen getrennt sind. Diese Informationen lassen sich problemlos mithilfe von `FIND REGEX` finden (siehe Listing 2.12).

```
DATA: login TYPE string,
      user  TYPE string,
      pass  TYPE string.
FIND REGEX '&login=\w+@\w+' IN url
      MATCH OFFSET off MATCH LENGTH len.
CHECK sy-subrc = 0.
login = url+off(len).
```

Listing 2.12 Anmeldeinformationen mit `FIND REGEX` ermitteln

Die Unterfolge `&login=ralph@geheim` mit den Anmeldeinformationen ist nun in der Variablen `login` gespeichert. Um den Benutzernamen und das Kennwort zu extrahieren, überspringen Sie zunächst den Namen des Parameters, indem Sie auf `login+7` verweisen und anschließend die restliche Zeichenfolge beim Trennzeichen abtrennen:

```
SPLIT login+7 AT '@' INTO user pass.
```

Dieser Code ist noch etwas umständlich, da Sie die Anzahl der zu überspringenden Zeichen selbst zählen müssen. Darüber hinaus müssen Sie einige der

¹⁶ Dieses konkrete Anwendungsbeispiel dient ausdrücklich nur zur Illustration und ist aufgrund seiner Sicherheitsmängel nicht für den Produktivbetrieb geeignet.

Eingaben zweimal durchsuchen, um an die interessanten Teile zu gelangen. Mithilfe von Untergruppen können Sie dieses Verfahren auf einen einzigen Vorgang reduzieren, der zudem weniger anfällig für Änderungen an der URL-Syntax ist:

```
FIND REGEX '&login=(\w+)(@\w+)' IN url  
SUBMATCHES user pass.  
CHECK sy-subrc = 0.
```

Die erste Untergruppe des RegExes speichert den in der URL enthaltenen Benutzernamen und die zweite Untergruppe das Kennwort. Die Schlüsselwörter und Begrenzungszeichen, die zur Suche nach den Informationen eingesetzt werden, werden zwar für die tatsächliche Übereinstimmung benötigt, jedoch nicht als Teil des Ergebnisses zurückgegeben. Dies ist einer der entscheidenden Vorteile bei der Verwendung von Untergruppen.

Als weiteres, etwas kompliziertes Beispiel sollen Sie nun einen Pfadnamen wie */usr/sap/work/profile.txt* in *Verzeichnis*, *Basisname* und *Erweiterung* unterteilen:

```
FIND REGEX '(.*/)?([^.]*)(\..*)?' IN path  
SUBMATCHES dir base ext.
```

Den ersten Teil des Pfadnamens bildet das Verzeichnis, zu dem auch der letzte Schrägstrich (/) zählt. Die Erweiterung ist der letzte Teil des Pfades und beginnt mit dem ersten Punkt (.) von links, der nicht zum Verzeichnis gehört. Der übrige Teil zwischen dem Verzeichnis und der Erweiterung ist der Basisname. Jeder dieser Teile kann leer sein.

Der erste Unterausdruck *(.*/)?* stimmt mit dem Teil des Pfadnamens bis zum letzten Schrägstrich überein (sofern vorhanden, anderenfalls bleibt die zugehörige Untergruppe leer). Dieses Ergebnis entspricht dem Verzeichnisteil. Anschließend wird der Text bis zum ersten Punkt ermittelt, der dem Basisnamen der Datei entspricht. Falls kein Punkt vorhanden ist, handelt es sich bei dem übrigen Abschnitt des Pfades um den Basisnamen. Auch hier gilt, dass der Basisname leer sein kann. Zuletzt wird eine Übereinstimmung für den Punkt (falls vorhanden) und den übrigen Teil des Pfades ermittelt, der für die Erweiterung des Pfades steht.

In Fällen wie in diesem Beispiel, in denen Sie Informationen eigentlich nur unterteilen anstatt suchen wollen, sollten Sie einen Abgleich statt einer Suche durchführen, um auszuschließen, dass Sie falsch positive Ergebnisse für eine passende Teilfolge erhalten. Wenn Sie für diese Aufgabe nicht direkt ein Mat-

cher-Objekt verwenden möchten, können Sie für die FIND-Anweisung das Durchführen eines Abgleiches erzwingen, indem Sie den Regex in Zeilenanker `^ ... $` einschließen. In diesem Beispiel mit dem Pfadnamen ist dies allerdings nicht erforderlich, da der verwendete Regex so gierig ist, dass er stets entweder mit dem gesamten Text oder gar nicht übereinstimmt.

Ersetzen von Übereinstimmungen

Ein weiterer nützlicher Aspekt von Untergruppen ist, dass Sie innerhalb eines Ersetzungstextes¹⁷ auf Untergruppen verweisen können. Sowohl die Anweisung REPLACE als auch die replace-Methoden kennen das spezielle Konstrukt `$n`, das die *nte* Untergruppe der aktuellen Übereinstimmung mit einem Regex bezeichnet.¹⁸ Dies ist sehr hilfreich, um mit einer einzigen Anweisung Informationen hinzuzufügen oder vorhandene Inhalte zu transformieren.

Angenommen, Sie möchten Datumsangaben im amerikanischen Format MM/DD/YY in das europäische Datumsformat DD.MM.YY konvertieren. Mit dem richtigen Regex wird hierfür lediglich eine Zeile benötigt:

```
REPLACE ALL OCCURRENCES OF REGEX '(\d+)/(\d+)/'
IN mytext WITH '$2.$1.'.
```

Zunächst wird nach den Datumsangaben im amerikanischen Format anhand der begrenzenden Schrägstriche gesucht, und die relevanten Teile mit den Monats- und Tagesdaten werden in Untergruppen extrahiert. Dann werden die alten Datumsangaben durch das neue Format ersetzt, bei dem die Reihenfolge von Tag und Monat umgekehrt ist und Punkte anstelle von Schrägstrichen verwendet werden.

Es ist sogar möglich, in einem Schritt Datumsangaben vom amerikanischen in das japanische Format (YY-MM-DD) und umgekehrt zu konvertieren, indem Sie von der Eigenschaft profitieren, dass Untergruppen ohne Übereinstimmung leer bleiben:

```
REPLACE ALL OCCURRENCES OF
REGEX '(\d+)/(\d+)/(\d+)|(\d+)-(\d+)-(\d+)'
IN mytext WITH '$2$6.$1$5.$3$4'.
```

¹⁷ Das ist der Text, der in der Anweisung REPLACE hinter WITH angegeben wird.

¹⁸ Für die ab den Releases 7.0, EhP2 und 7.1/7.2 verfügbaren eingebauten replace-Funktionen gilt das natürlich genauso.

Die Verarbeitung dieses Codes ist sogar etwas schneller als die Verwendung von zwei separaten REPLACE-Anweisungen für jeweils eine Umwandlungsrichtung.

Wie in der Einführung zu Untergruppen bereits beschrieben, steht die Untergruppe mit dem Index 0 gemäß Konvention für die gesamte Übereinstimmung. Daher können Sie auch \$0 verwenden, um die gesamte aktuelle Übereinstimmung in den Ersetzungstext aufzunehmen. Dies ist insbesondere für das Hinzufügen von Informationen zum ursprünglichen Text hilfreich. Nehmen Sie beispielsweise an, dass die Fettschrift-Tags und zu allen Zahlen in einem HTML-Dokument hinzugefügt werden sollen. So einfach sieht das dann aus:

```
REPLACE ALL OCCURRENCES OF REGEX '\d+'  
IN mytext WITH '<b>$0</b>'.
```

Beachten Sie, dass die \$n-Operatoren ausschließlich in Ersetzungstext verwendet werden können. Sie sind zwar nicht Teil der Regex-Syntax, es ist aber eine enge Verknüpfung vorhanden. So führt eine ungültige Angabe für Untergruppen, wie zum Beispiel \$*, zum Ausnahmefehler cx_sy_invalid_regex_format. Wenn Sie ein literales Dollarzeichen in die Ersetzungstexte einfügen müssen, können Sie einfach das für Regexe übliche Fluchtsymbol verwenden: \\$.

Um den Abschnitt über Ersetzungen abzuschließen, sollen Sie sich nochmals an Kapitel 1 erinnern: Beachten Sie beim Arbeiten mit REPLACE, dass die von REPLACE zurückgegebenen Informationen sich anders als bei FIND nicht auf die gefundenen Übereinstimmungen, sondern immer auf die eingefügte Zeichenfolge beziehen:

```
mytext = '-x-x-'.  
REPLACE ALL OCCURRENCES OF REGEX '\w+'  
IN mytext WITH 'ooo'  
REPLACEMENT COUNT cnt REPLACEMENT OFFSET final_off.  
WRITE: / mytext, cnt, final_off.
```

Als Ergebnis wird -ooo-ooo- 2 5 ausgegeben – das heißt, final_off weist bei der Fertigstellung den Wert 5 und nicht 3 auf.

Rückwärtsreferenzen

Ein weiterer recht ausgefeilter, jedoch eher selten verwendeter Operator für Untergruppen ist der *Rückwärtsreferenzoperator* (\n), mit dem auf die nte unter-

geordnete Übereinstimmung im Regex selbst verwiesen werden kann. Oder mit anderen Worten: Jeder Untergruppe ist ein Operator `\1, \2, \3 ...` zugeordnet, der hinter dem entsprechenden Unterausdruck angegeben werden kann und rückwärts auf die Untergruppe verweist. Er wirkt also im Regex als Platzhalter für die Zeichenfolge der Untergruppe. Dies mag zunächst seltsam klingen, aber nehmen Sie zum Beispiel an, dass Sie über eine einfache Datenbank mit Benutzernamen und Passwörtern verfügen, die als Zeilen aus `user:pass`-Einträgen in einer internen Tabelle `userdata` gespeichert sind. Dann können Sie die Anweisung

```
FIND ALL OCCURRENCES OF REGEX '^(\w+):\1$' IN TABLE userdata
RESULTS res.
```

verwenden, um alle nachlässigen Benutzer zu ermitteln, deren Kennwort mit dem Anmeldenamen übereinstimmt.

An dieser Stelle ist es nochmals wichtig, den grundlegenden Unterschied zwischen *nicht übereinstimmenden* und *nicht vorhandenen* Untergruppen herauszustellen. Eine Rückwärtsreferenz auf eine Untergruppe ohne Übereinstimmung ist möglich und steht für eine leere Zeichenfolge, das heißt, dass sie praktisch ignoriert wird. Der Regex

```
regex:      (["])?\w+\1
found here: This "is" 'not' 'it'!
```

stimmt beispielsweise mit Wörtern überein, die auch in Anführungszeichen stehen können. Dabei müssen die Anführungszeichen jedoch konsistent sein. Wird der Fragezeichenoperator innerhalb des Unterausdrucks platziert, ändert sich zwar die Funktionsweise innerhalb dieses Ausdrucks geringfügig, dies hat allerdings keine Auswirkungen auf das Endergebnis der Übereinstimmung.

Das Verweisen auf nicht vorhandene Untergruppen oder auf Gruppen, die erst hinter dem Rückwärtsreferenzoperator kommen, ist allerdings nicht zulässig und löst die Ausnahme `cx_sy_invalid_regex` aus. Als Faustregel gilt, dass Sie stets überprüfen sollten, ob der *nten* Rückwärtsreferenz mindestens *n* schließende Klammern nachgestellt sind.

Im Gegensatz zu den meisten anderen Regex-Implementierungen ist die Anzahl an Rückwärtsreferenzen in ABAP nicht auf `\1` bis `\9` begrenzt. Stattdessen ist eine beliebige Anzahl von Referenzen, wie zum Beispiel `\69`, zulässig. In anderen Sprachen ist dies nicht möglich, da solche Ausdrücke in der Regel als Steuerungscodes für ASCII-Zeichen interpretiert werden.

Index

.NET Connector 204

A

ABAP

Debugging 373
Programmierfehler 345
Remote Communication 199
Serialization XML 120
Webservice 125
XML-Mapping 113, 125

ABAP Debugger 177, 178, 179, 307, 335, 336, 337, 338, 343, 369, 374, 377, 392, 406, 460, 467, 470, 525
Fehlermeldung 391
Interface → ADI
Nachteil 406
Neuer ABAP Debugger → Neuer ABAP Debugger
Starten 375, 380, 383

ABAP Dictionary 352, 353, 354, 538
Datentyp 354
Element 548

ABAP Editor 334
Programmeigenschaft 381

ABAP Serialization XML → asXML

ABAP Shared Objects 157, 158, 169
Administration 164
erweiterte Programmietechnik 179
Programmiermodell 160
Shared Objects Monitor 164, 167, 168, 169
Zugriffsmodell 163

ABAP Test Cockpit → ATC

ABAP Unit 309, 311, 323, 325, 477, 479, 500
Browser 326
Delegation 496
Detailsicht 494
Ergebnisanzeige 492
Hierarchiedarstellung 492
Massentest 501
Testisolation 497
Vorgehen 324

ABAP-Checkpoint 443

ABAP-Code 548

ABAP-Dump 390

ABAP-Dumpanalyse 337, 340, 342, 344, 347

ABAP-Laufzeitanalyse 336, 337, 355, 358, 365

Verwenden 364

ABAP-Laufzeitfehler 264, 345

ABAP-Modultest 308

ABAP-Programm 316

Testen 308

ABAP-Trace 355, 356, 358, 368, 376, 380

Ergebnis 360

paralleler Modus 365

Starten 364

Tipp 362

Variante 363, 368

ABAP-Verbindung 238, 242, 303

Abdeckungsdaten 332

abgebrochener Job 338

Ablauflogik 553

Abnahmetest 479

ABORT 470

ACCEPTING CASE 74

Add-on 549

ADI 409

aktivierbare Assertion 447

aktivierbarer Breakpoint 461

Aktivierung

benutzerspezifische 452

Einstellung 451, 459, 467

globale 452

globale Variante 472

gruppenspezifische 451

programmspezifische 451, 456

serverspezifische 452

Variante 457

Aktualparameter 425

ALE 199

ALL OCCURRENCES 48

ALL OCCURRENCES OF 73

ALL_ABORT_ABORT 472

ALL_BREAK_ABORT 472

ALL_BREAK_LOG 472

- ALL_LOG_LOG 472
Alternative 62
Analyse
 Post-Mortem-Analyse 338
Analysewerkzeug 307
Änderungssperre 193
Anker 70
Anmeldebild 244
Anmeldeinformation 205
anonymes Datenobjekt 508
Ansatz
 Inside-Out 119
 Outside-In 118
Anweisung 395
 deklarative 444
 operative 444
Anwendungsprotokoll 261, 299
Anzeigefilter 363
Append 547
Application Link Enabling → ALE
Application Log 261, 299, 462
ArchiveLink 199
AS ABAP 201, 224
ASSERT 444, 446
ASSERT CONDITION 448
ASSERT FIELDS 472
ASSERT ID 448
ASSERT SUBKEY 472
Assertion 445, 470, 474
 aktivierbare 447
 Bedingung 447, 453, 470, 474
 Protokoll 470
ASSERTION_FAILED 446, 450, 470
Assert-Methode 325, 481, 494
 abort 484
 assert_... 484
 assert_bound 487
 assert_differs 487
 assert_equals 484
 assert_initial 487
 assert_not_bound 487
 assert_not_initial 487
 assert_subrc 487
 assert_that 488
 fail 483
 Parameter 483
 spezialisierte 484
asXML 120, 127, 235
asynchroner RFC → RFC
ATC 322, 502
Atomic 267
Aufruf 357
 zustandsloser 210
Aufrufebene 360, 361
Aufrufhierarchie 360, 367
Aufrufstack 419, 422
Ausführungskontext 452
Ausführungszeitpunkt 226
Ausgabeparameter 226, 227
Ausnahme 396
 vordefinierte 261
Ausnahmebehandlung
 Zusichern 488
Ausnahmefehler 493
Automaten-Cache 107
automatischer Gebietsaufbau 185
-
- B**
- Background RFC → bgRFC
Backslash-Operator 62
Backtracking 103, 110
BAdI 540, 542, 543, 546, 549
basXML 234, 235, 236, 238, 246, 250,
 259, 292
bedingter Checkpoint 447
Befehlstyp 383
Begrenzung 99
Begrenzungsoperator 71
Begrenzungszeichen 97
Benutzer 245
Benutzergruppe 243
Benutzerkontext 366
Benutzersitzung 202
benutzerspezifische Aktivierung 452
Benutzerstammsatz 21
Benutzerwechsel 226, 227
Berechtigungsobjekt 211
Berechtigungsprüfung 237
Berechtigungstest 245
Bereichsoperator 63
bestimmte Einheit 368
Betriebsart 452, 469
bgRFC 265, 291
 ACID 267
 API 280
 Destination 268, 272
 Destination konfigurieren 292

- bgRFC (Forts.)**
- Framework* 266, 292
 - Inbound* 271, 281
 - Outbound* 270, 281
 - Out-Inbound* 273, 281, 290
 - Programmieren* 282
 - Scheduler* 270, 273, 283, 289
 - Scheduler konfigurieren* 300
 - Szenario* 269
 - Unit* 266, 274, 278, 279, 283, 284, 300
 - Unit überwachen* 295
 - Verarbeiten* 299
 - Verbuchen* 288
- Big Endian** 250
- Binärdaten** 19
- Typ* 25
- Binärmodus** 25
- Boyer-Moore-Algorithmus** 50
- Branchenerweiterung** 538
- Branchenlösung** 535, 537, 539, 549
- branchenspezifische Funktionalität** 549
- BREAK** 470
- BREAK-POINT** 393, 426, 444, 460
- Breakpoint** 370, 384, 392, 400, 419, 422, 434, 460, 470
- aktivierbarer* 461
 - Anlegen* 428
 - Debugger-Breakpoint* 393, 394, 426, 428
 - dynamischer* 395
 - Externer Breakpoint* 394, 426, 428
 - flüchtiger* 461
 - Session-Breakpoint* 426
 - Setzen* 426
 - Symbol* 427
 - Typ* 427
 - Typ ändern* 430
 - Werkzeug* 467
 - Wertehilfe* 429
 - Zeilen-Breakpoint* 394
- BREAK-POINT ID** 461
- Breakpoints**
- Session-Breakpoint* 393
- Broker-Klasse** 194
- Buffer**
- Exclusive Buffer* 160
- Business Add-in → BAdI**
- Business Function** 554
- Business Function Set** 547
- BYTE MODE** 25, 50
- BYTE-CA** 53
- BYTE-CN** 53
- BYTE-CO** 53
- Bytecode** 355
- BYTE-CS** 53
- Bytefolge** 19
- BYTE-NA** 53
- Bytereihenfolge** 250
-
- ## C
- CA** 52
- CALL FUNCTION** 208
- CALL TRANSFORMATION** 121, 123, 125, 146, 152, 153
- Callback** 212
- Destination* 248
 - Mechanismus* 227
- Casting**
- implizites* 30
- CATT** 329
- CFW** 204
- Checkpoint** 393, 443
- bedingter* 447
 - Gruppe* 448
- cl_abap_char_utilities** 22, 46
- cl_abap_conv_in_ce** 46
- cl_abap_conv_out_ce** 46
- cl_abap_conv_x2x_ce** 46
- cl_abap_matcher** 70, 74, 86, 108
- cl_abap_memory_utilities** 528
- cl_abap_regex** 74, 108
- cl_abap_string_utilities** 46
- cl_abap_weak_reference** 518
- cl_aunit_assert** 481
- cl_sxml_string_reader** 153
- Class Builder** 491, 500
- Class-Pool** 491
- Testmethode* 492
- CLEAR** 511
- clike** 36
- CN** 52
- CO** 52
- Code Inspector** 80, 308, 309, 315, 321, 334, 480, 502
- Codepage** 21, 63, 251, 253
- Umgebungs-Codepage* 252, 258

- Codepage (Forts.)
 verwendete 255
- COMMIT WORK 218, 440
- Component under Test → CUT
- CONCATENATE 40
- CONDENSE 45
- Consistency 267
- contains 52, 78, 79
- Control 204
- Copy-on-Detach 195
- Copy-on-Write 35, 195, 509
- Coverage Analyzer 309, 310, 330, 332, 334
 Anzeige 334
- CP 54, 78
- CREATE DATA 521
- CREATE OBJECT 521
- CS 53, 78
- csequence 36
- Customer-Exit 540, 544
- Customizing 540
- Customizing-Anpassung 549
- CUT 489, 494
- cx_st_error 154
- cx_sy_range_out_of_bounds 39
- cx_transformation_error 154
- cx_xslt_exception 154
-
- D**
- Data Explorer 406, 420, 425
- Data-Dictionary-Objekt 308
- Dateizugriff 357
- Datenbank
 Commit 219
- Datenbankzugriff 357
- Datenobjekt 423
 Analysieren 423
 anonymes 508
- Datenreferenz 508
 Vergleich 487
- Datentyp
 generischer 36
- Datumsfeld 28
- Debuggee 409
- Debugger Engine 409
- Debugger-Breakpoint 393, 394, 426, 428
- Debugger-Variante 417
 Sichern 418
- Debugging 226, 295, 375, 432
 Einstellung 412
 Gebietsinstanz 170, 177
 Hintergrundjob 385
 Layer-aware Debugging 388
 Szenario 433
- Debugging-Modus 375, 378, 379, 387
 exklusiver 441
 nicht exklusiver 440
- Definition
 globale 321
 lokale 321
- deklarative Anweisung 444
- Delegation 496
- DEMO_REGEX_TOY 79
- Deserialisierung 118, 126, 249
- Desktop 413
- DESTINATION 208
- Destination 210, 238, 268, 285, 295, 297
 Destinationskennung 249
 dynamische 205
 Sperren 298
 Supervisor-Destination 303
 vordefinierte 247
- Dezimalzahl
 gültige 67
- Dialoginteraktion 226, 227, 228, 229
- Dialogtransaktion 380
- Dialogverarbeitung 384
- Dialog-Workprozess 202
- Difftool 406, 411, 420, 436, 437
- Dispatcher 201
- Dispatcher-Queue 291
- Document Object Model → DOM
- Document Type Declaration → DTD
- DOM 113, 116, 124
- DTD 153
- Dumpanalyse 374, 384
- Dumpkontext 344
- Durable 267
- dynamische Destination 205
- dynamischer Breakpoint 395
- dynamisches Speicherobjekt 507, 508
- Dynpro 228, 538
- Dynproanalyse 420
- Dynpro-Element 547, 548

E

eCATT 309, 310, 323, 328
Vorteile 329
 Edit Control 421
 Einheit
bestimmte 368
 Einsatz von Analysewerkzeugen 335
 Einzelanzeige 435
 Einzelfeld 420
 Einzelschritt 400
 Endlosschleife
temporäre 386
 Enhancement Framework 544, 545, 547
 Enhancement Spot 545
 ENHANCEMENT-POINT 545
 ENHANCEMENT-SECTION 545
 Enqueue-Workprozess 202
 Entwicklertest 479
 Entwicklungsobjekt 546
Typ 548
 Ergebnis
falsch negatives 65
falsch positives 65
 Ergebnisanzeige 492
 erste längste Übereinstimmung 68
 erweiterte Programmprüfung 308, 309,
 312
 Erweiterung 554
Implementierung 545
konsolidierte 535
verfügbare 556
 Erweiterungspunkt 554, 556
 Erweiterungstechnologie
klassische 542
 Exactly Once 216, 226
 Exactly Once In Order 216, 226
 Exchange Infrastructure → SAP NetWeaver Process Integration
 Exclusive Buffer 160
 extended Computer Aided Test Tool →
 eCATT
 Extensible Markup Language → XML
 Extensible Stylesheet Language Transformations → XSLT
 Externer Breakpoint 394, 426, 428
 externer Fehler 346
 Extreme Programming 328

F

Factory-Klasse 273
 Factory-Methode 271, 273, 286
 falsch negatives Ergebnis 65
 falsch positives Ergebnis 65
 falsche Kommunikationsart 254
 Fehler
externer 346
interner 345
 Fehleranalyse 348, 485
 Fehlerliste 493
 Fehlermeldung 212
ABAP Debugger 391
 Fehlerursache
Ermitteln 347, 348
 Feld 352
Inhalt überprüfen 351
vorbelegtes 375
 Feldattribut 553
 Feldsymbol 36
 feste Länge 20
 FIFO 275
Pipe 275, 276
 FIND 46, 47, 72, 85, 107
 find 79
 First In, First Out → FIFO
 Fixture 494, 497
 flüchtiger Breakpoint 461
 Fluchtsymbol 53, 99
 Formalparameter 36
 Formatierung 43
Regel 44
 Framework 373
 FREE 511, 519
 Function Builder 204, 425, 556
 funktionale Methode 474
 funktionaler Methodenaufruf 474
 Funktionalität
branchenspezifische 549
 Funktionsbaustein 199, 228, 288, 396
Aufrufen 207, 217
remotefähiger → RFM 233
 Funktionsgruppe 236

G

Garbage Collector 425, 507, 512
 Gateway 201

- Gebiet
 mandantenabhängiges 189
- Gebietsaufbau
 automatischer 185
- Gebietshandle 174, 193
- Gebietsinstanz 160, 163, 179
 Aktualisieren 170, 174
 Aktualisierungszeit 188
 Debugging 170, 177
 Erzeugen 170, 171
 Fehler behandeln 170, 176
 Lebensdauer 188
 Leerlaufzeit 188
 Objekt lokaler Klasse 175
 Speichern von Daten 170
 Sperren 161
 transaktionale 191
 Verfallszeit 188
 Versionierung 180
 Wurzelobjekt 161
 Zugreifen 170, 173
- Gebietsklasse 164
- Gebetskonsruktur 186
 Implementieren 187
- generischer Datentyp 36
- genügsame Übereinstimmung 97
- gierige Übereinstimmung 69, 96
- Gierigkeit 87
- Gleitkommazahl
 Vergleich 486
- globale Aktivierung 452
- globale Definition 321
- globale Testklasse 498
- globale Testmethode 499
- globale Variable 424
- Graph 512
 gerichteter 512
- Gruppe
 Logon-Gruppe 240
 RFC-Servergruppe 240
- GUI-Verknüpfung 382
 Erstellen 382
- gültige Dezimalzahl 67
- gültige Kreditkartennummer 66
- Gültigkeitsbereich 451, 456
-
- H**
- Hash-Tabelle 436
- Hauptmodus 232
- Haupt-Template 147
- Header 34
- heterogene Kommunikation 253, 254, 256, 257
- Hexadezimalwert 25
- Hierarchiedarstellung 492
- Hilfsmethode
 Testklasse 497
- Hintergrundjob 383, 388
 Debuggen 385
 im Debugging-Modus starten 387
- Hintergrundprozess 367
- Hintergrundverarbeitung 460, 461
- Hintergrund-Workprozess 202
- hmusa 528
- homogene Kommunikation 253, 254, 256
- Hook 540
- HTML
 Tag 99
- HTTP-Service 462
-
- I**
- ICM 201
- IDoc 199
- id-Transformation 121
- IGNORING CASE 50, 74, 111
- implizite Methode 495
- IN TABLE 48
- Inbound-Destination 293, 294
- Inbound-Scheduler 221, 290
- Inbound-Szenario 272, 281
- Inbound-Unit
 Erstellen 282
 Typ Q 285
 Typ T 283
- Include-Programm 316, 348
- INITIAL SIZE 519
- Initialisierungsfehler 325
- Inkonsistenz
 Feststellen 353
- Inside-Out-Ansatz 119
- Inspektion 316
 Anlegen 320

Inspektion (Forts.)
Ausführen 320
Starten 320
 Inspektionslauf 321
 Installationsfehler 346
 Instanzattribut 434
 Instanzkonstruktor 436
 Instanzmethode 500
parameterlose 480
 Integrationstest 328
 Intermediate Document → IDoc
 interne Tabelle 236, 357, 403, 414
Vergleich 485
 interne Verbindung 248
 interner Fehler 345
 interner Modus 237, 467, 518
 Internet Communication Manager → ICM
 INTO TABLE 41
 Isolation 267
 iXML
Bibliothek 116
Parser 152

J

Java Connector 204
 Job
abgebrochener 338
 Job-Log 384
 Jobübersicht 339, 387

K

kanonische XML-Darstellung 119
 Kante 512
 Knoten 512
 Kommentar 443
 Kommunikation
heterogene 253, 254, 256, 257
homogene 253, 254, 256
 Kommunikationsart 246, 254
falsche 254
 Kommunikationsfehler 216, 260
 kontextbasierte Suche 71
 Kontextknoten 129
 Kontrollbereich 413
 Kontrollblock 402

Konvertierung 29
Exit 44
Regel 29
 Kreditkartennummer
gültige 66
 Kurzdump 263, 342, 370, 390
Ent sperren 343
Navigation 346
Sperren 343
 Kurzdumpanzeige
Kategorie 345
 Kurzdumptext 347

L

Länge
feste 20
variable 20
 Lastausgleich 269, 302
 Lastverteilung 239, 241
 Laufzeitanalyse 361
 Laufzeitfehler 342, 353, 384, 493
 Laufzeitprüfung 309, 310, 311
 Layer-aware Debugging 388
 Leerzeichen
schließendes 23, 33, 38
 Leftmost-longest-Regel 69
 Legacy-Framework 292
 LENGTH 23, 26, 48
 Lesehandle 174
 Lesesperrre 184
 Literaloperator 22, 95, 99
 Little Endian 250
 Loader-Klasse 194
 LOG 470
 Logical Unit of Work → LUW
 logische Verbindung 302
 logisches System 203
 Logon-Gruppe 240, 241
 LOG-POINT 444, 463
FIELDS 464
ID 463
SUBKEY 466
 Logpoint 462, 470
 lokale Definition 321
 lokale Testklasse 490
 Loopback-Destination 248

- Lösung
 Ausarbeiten 353
LUW 203, 215, 219, 267, 283, 288, 440
-
- ## M
- mandantenabhängiges Gebiet 189
Mapping-Liste 134
Massentest 316, 480
match 79
MATCH COUNT 73
MATCH LENGTH 47, 72
MATCH LINE 72
MATCH OFFSET 47, 72
MATCH_RESULT_TAB 49, 73
Matcher-Klasse 75
Matcher-Objekt 75
matches 52, 78, 79
Matching-Operation 104
MDMP 252
 System 21, 252, 255, 258
Mehrfachverwendung 34
Meldungsart 493
Memory Inspector 420, 506, 519, 525
Mengenausdruck 62
Menüelement 548
MESSAGE 262
Message-Server 241
Metazeichen 53, 61
Methode 396
 Assert-Methode 325
 funktionale 474
 implizite 495
 statische 496
Methodenaufruf
 funktionaler 474
Methodendefinition 488
Modifikation 542
Modifikationsassistent 541
Modultest 311, 323, 326, 328, 477, 478,
 479, 481, 489, 497
 Ablauf 494
 Ablaufdiagramm 498
 globale Klasse 489
 Implementieren 491
Modus
 interner 237, 467, 518
Modusnummer 412
Modustyp 412
- MOVE 29, 510
Multibyte-Codepage 21, 25
Multi-Display, Multi-Processing-System
 → MDMP
Muster 53
-
- ## N
- NA 52
Nachricht
 Senden 433
 Typ 263
Negationsoperator 62
negativer Vorausschauoperator 93
Neuer ABAP Debugger 405
 Benutzeroberfläche 410, 414
 Detailsicht 419, 424
 Einstellen 408
 Prozessinformation 412
 Variante 417
 Werkzeug 419
 Zwei-Prozess-Architektur 408
nicht vorhandenes Programm 356
Nicht-Unicode-System 252, 255
numerischer Text 27
-
- ## O
- Oberklasse 397
Object Navigator 545
Objekt 420, 508
 Referenz 508
Objektliste 554
Objektmenge 316, 334
 Definieren 317
Objektreferenz 283, 284, 286, 403
 Vergleich 487
Objektsicht 435
OFFSET 48
Offset-/Längenzugriff 38
operative Anweisung 444
Outbound-Destination 271, 292
Outbound-Queue 286
Outbound-Scheduler 221, 273, 290
Outbound-Szenario 271, 281
Outbound-Unit
 Erstellen 285, 287
 Sperre aufheben 286
 Typ Q 287

Typ T 285
 Out-Inbound-Szenario 281, 290, 291
 Out-Inbound-Unit 287
queued 287
 Outside-In-Ansatz 118

P

Paket 546
 Paketzuordnung 542
 paralleler RFC → RFC
 Parallelisierung 222, 225, 275
 Parameter
tiefer 236
 parameterlose Instanzmethode 480
 Passwort 245
 Performanceassistent 377
 Perl 103, 106
 Personalisierung 540
 Pipe-Operator 62
 Plausibilitätsprüfung 65
 Plusoperator 99
 Positionsdetail 550
 POSIX 103
Standard 69, 106
 Post-Mortem-Analyse 338
 Produktivbetrieb 460
 Produktivsystem 460, 536
 Profilparameter 328, 357
 Program Execution Area → PXA
 Programm
nicht vorhandenes 356
 Programmablauf
Analysieren 375
 Programmprüfung
erweiterte 308, 309, 312
 Programmzustand 445
 Protokoll
Eintrag 463, 470
Speicher 463
 Prozessübersicht 385, 442
 Prüffehler 493
 Prüfung
statische 309, 312
 Prüfvariante 316
Definieren 319
 Pseudokommentar 315
 Punktoperator 61
 PXA 161

Q

QoS 266, 270, 273
Typ 270
 QoS BE 271, 272
 QoS EO 270, 271, 272, 274, 282
 QoS EOIO 270, 271, 274, 275, 278, 281
 qRFC-Scheduler 220
 Quality of Service Best Effort → QoS BE
 Quality of Service Exactly Once In Order
 → QoS EOIO
 Quality of Service Exactly Once → QoS EO
 Quality of Service → QoS
 Quelltext 419
Überprüfen 348
 Quelltextanzeige 314, 422, 437
 Quelltextebene 309
 Quelltexterweiterung 543
 Quelltextinformation 413
 Quelltext-Plug-in 543, 545
 Quelltextposition 392
 Queue
Präfix 294
Sperre löschen 298
Statusanzeige 297
virtuelle 280
 queued RFC → RFC

R

Rangliste 520
 Realtime Information and Billing System
 → RIVA
 RECEIVE RESULTS 213
 Referenz
Anzeigen 425
schwache 518
Semantik 508
Zähler 513
 Referenzvariable 435
 REFRESH 511
 REGEX 72
 Regex 57, 61
Erstellung 107
Klasse 74
Muster 101
Objekt 74
Performance 107

- Regex (Forts.)
 Regex Toy 79
 Ressource 101
- Regressionstest 308, 501
- regulärer Ausdruck → Regex
- Reliable Messaging 272, 274
- Remote Function Call → RFC
- Remote-enabled Function Module →
 RFM
- remotefähiger Funktionsbaustein → RFM
- Remote-Generierung 442
- Remote-System 295
- REPLACE 46, 51, 72, 90, 107
- replace 79
- REPLACEMENT COUNT 74
- REPLACEMENT LENGTH 51, 74
- REPLACEMENT LINE 74
- REPLACEMENT OFFSET 51, 74
- Representational State Transfer → REST
 Paradigma
- RESPECTING CASE 50
- Ressourcenengpass 346
- Ressourcenmanagement 289
- REST
 Paradigma 114
- RESULTS 49
- RFC 199, 265
 Aktion im Workprozess 206
 asynchroner RFC (aRFC) 210, 212, 223,
 228, 231
 asynchroner RFC (aRFC) mit Antwort
 212
 asynchroner RFC (aRFC) ohne Antwort
 210
 Ausnahmebehandlung 249, 260
 Auswahl 225
 Basistyp 215
 Benutzer 202
 Bibliothek 209
 binäres Protokoll 249
 Client 203
 Datenübertragung 249
 Destination 238, 253
 Eigenschaft 226
 Einschränkung 265
 Fehlerprotokoll 263
 Grundlagen 200
 Kommunikation 241, 256, 257
 Kommunikationsfehler 260
- RFC (Forts.)
 Kommunikationsprozess 204
 Nachricht 262
 paralleler RFC (pRFC) 222, 223, 224,
 229
 Protokoll 235
 Prozess 205
 qRFC-Scheduler 220
 queued RFC (qRFC) 215, 220, 229, 248
 Schnittstellendaten 206
 Server 203
 Servergruppe 240, 294
 Sitzung 206
 synchroner RFC (sRFC) 208, 212, 228
 Systemfehler 261
 Trace 246
 transaktionaler RFC (tRFC) 215, 218,
 229, 248
 Variante 207
 Workprozess-Ablauf 207
- RFM 200, 204, 231, 232, 233
 Anlegen 233
 asynchroner RFM 212
 Ausführen 211
 Debugging 389
 Performance 235
 Robustheit 237
 Sicherheit 236
 RIGHT-JUSTIFIED 44
- RIVA 538
- ROLLBACK WORK 218
- Rollbereich 467
- Rollbereichsende 409
- Roll-In 203
- Roll-Out 203
- Round-Robin-Verfahren 276
- Rückgabewert 212, 350
- Rückwärtsreferenzoperator 91
-
- S**
-
- S_MEMORY_INSPECTOR* 529
SAP Control Framework → CFW
SAP Core System 535
SAP ERP 6.0 535
SAP LUW → LUW
SAP NetWeaver 535, 537
SAP NetWeaver Application Server ABAP
 → AS ABAP

- SAP NetWeaver Process Integration 114,
125
SAP Simple Transformations → ST
SAP-Server 393
SAPshortcut 381
SAP-Spool-System 387
Schalter 543, 546, 552
Scheduler
 Benutzer 300, 304
Schleifenzähler 405
schließendes Leerzeichen 23, 33, 38
Schnittoperator 106
schwache Referenz 518
Screen Painter 536, 552
SEARCH 46
Secure Network Communications → SNC
SEPARATED BY 40
Serialisierung 113, 118, 126, 249
 Standardserialisierung 139
Servergruppe 272
 Pflegen 294
serverspezifische Aktivierung 452
Servicebenutzer 300
Session-Breakpoint 393, 426
SET COUNTRY 43
Shared Buffer 159
Shared Memory 157, 160, 332
 Aufbau 160
 Überwachung 164
 Verwaltung 164
 Werkzeug 164
Shared Objects Monitor 164, 167, 168,
169
Sharing 34, 37, 509
SHIFT 42
Simple Transformations → ST
Single Sign-on 243
Single-Byte-Codepage 25
SNC 244
Software-Lebenszyklus 311
Softwarequalität 443
Sortierung
 topologische 277, 279
space 24
Speicher
 Abzug 528, 529
 allokierter 515
 Analyse 525
 benutzter 515
Speicher (Forts.)
 gebundener 514
 Problem 505
 referenzierter 514
 Speicherleck 506, 518
 Verbrauch 507, 514
 Verwalten 506
Speicheranalyse 420
Speichergrenze 192
Speicherobjekt
 dynamisches 507, 508
 statisches 507
Speicherverbrauch 359
Sperre 179
Sperren
 mit Versionierung 182
 ohne Versionierung 181
SPLIT 41
Spool-Workprozess 202
Sprache 244
SQL Trace 337, 433
ST 125
 Anweisung 129
 Programm 128, 150
 Wurzel 129
stabile Struktur 237
Standardnotation 279
Standardprüfvariante 321
starke Zusammenhangskomponente
 522, 523
statische Methode 496
statische Prüfung 309, 312
statischer Test 311
statisches Speicherobjekt 507
Statusanzeige
 Queue 297
Steueroperation 104
Steuerzeichen 22
String 32, 195, 237, 508
 Literal 32
 Referenz 508
Struktur 30, 420
 Komponente 30
 stabile 237
 tiefe 36
Strukturkomponente 397, 399
Strukturkomponenten-Selektor 30
SUBMATCHES 85
SUBSTRING 47

- Suche 68
 kontextbasierte 71
- Suchergebnis 361
- Supervisor-Destination 303
- Support Package 541
- Switch Framework 546, 547
 Architektur 547
- Switch → Schalter
- Switch und Enhancement Framework 535, 536, 558
 Einführung 542
 Einsatz 550
 Funktionsweise 543
 Integration 536
 Vorteile 536
- symmetrische Transformation 126
- synchrone RFC → RFC
- Synchronisation 289
- Syntaxprüfung 312
- System
 Debugging 461
 logisches 203
 Programm 461
 Protokoll 461, 462
- SYSTEM_NO_ROLL 505
- Systembefehl 383
- Systembereich 420
- Systembereichsanzeige 505
- System-Exception 396
- Systemfehler 261
- Systemfeld 349, 402, 413
- Systemmodul 380
- Systemprotokoll 337, 339, 340, 384
 Ausgeben 340
 Dateigröße 341
 Spalte 341
- T**
-
- T005X 43
- Tabelle 420, 538
 Hash-Tabelle 436
 Header 404
 interne 195, 236, 357, 403, 414
- Tabellenkörper 508
- Tabellenreferenz 508
- Tabellenvergleich 485
- Taskhandler 289
- TDD 490
- Template 129
 Haupt-Template 147
 Unter-Template 148
- temporäre Endlosschleife 386
- temporäres Testprogramm 479
- Test 477
 einrichten 310
 statischer 311
 Wiederverwenden 498
- Testannahme
 Überprüfen 481
- Test-Driven Development → TDD
- Test-Fixture 326
- testgetriebene Entwicklung → TDD
- Testgruppe 331
- Testhierarchie 492
- Testisolation 497
- Testklasse 480, 492
 globale 498
 Hilfsmethode 497
 lokale 490
 Vererben 501
- Testlauf 316
- Testlogik 480
- Testmethode 326, 480, 492, 496, 500
 Ausnahmebehandlung 488
 globale 499
- Testperformance 310
- Testprogramm
 temporäres 479
- Teststrategie 311
- Testwerkzeug 307
 Software-Lebenszyklus 311
 Überblick 309
 Verwenden 334
- Text
 extrahieren 59
 Muster 61
 numerischer 27
 transformieren 59
 validieren 58
 verarbeiten 57
- Textfeldliteral 22
- Textsprache 258, 260
- Textübersetzung 425
- Textumgebung 21
- Thread 277
- tiefe Struktur 36
- Tiefenanalyse 373

- tiefer Parameter 236
 topologische Sortierung 277, 279
 Trace 246
 Trace-Ergebnis 361
 Trace-Variante 359
 Tracing 357, 368
 Transaktion
 PFCG 300
 RZ11 207
 RZ12 224, 321
 SAAB 449, 463
 SBGRFCCONF 282, 293, 305
 SBGRFCMON 295
 SCI 308, 315, 480, 502
 SCOV 309, 331
 SE09 375, 398, 402
 SE24 325
 SE30 336, 355, 358
 SE37 425
 SE93 378, 398
 SECATT 309
 SFW5 548
 SHMA 164, 188
 SHMM 164, 167
 SLG1 299
 SLIN 308, 313
 SM21 339
 SM37 338, 387
 SM50 385
 SM51 393
 SM58 216, 219, 255
 SM59 205, 239, 292
 SMLG 241, 294
 SMQ2 227
 SMQS 222
 SMT2 242
 ST05 433
 ST11 263
 ST22 340, 342
 VA01 442
 Transaktion *SM37* 339
 transaktionaler RFC → RFC
 Transaktionskennung 216
 Transaktionspflege 378, 398
 Transformation 125
 id-Transformation 121
 symmetrische 126
 Transformationsrichtung 118
 umkehrbare 126
- TRANSLATE 45
 Transport Organizer 375, 398, 402
 Trusted System 242
 TSV_TNEW_PAGE_ALLOC_FAILED 505
 tt:apply 148
 tt:assign 144
 tt:attribute 132
 tt:call 149
 tt:clear 144
 tt:cond 138
 tt:context 148
 tt:copy 135
 tt:deserialize 137
 tt:group 142
 tt:include 149
 tt:lax 143
 tt:loop 136
 tt:parameter 146
 tt:read 145
 tt:ref 129
 tt:root 129
 tt:serialize 137
 tt:skip 137
 tt:switch 138
 tt:switch-var 146
 tt:template 129
 tt:text 132
 tt:transform 128
 tt:value 130, 132
 tt:value-ref 130
 tt:variable 146
 tt:with-parameter 149
 tt:with-root 148
 tt:write 145
 TYPE REF TO 509
-
- U**
-
- Übereinstimmung
 erste längste 68
 genügsame 97
 gierige 69, 96
 unerwünschte 98
 Übertragungsprotokoll 235, 246, 292
 Überwachung 226, 227
 Umfeldmethode 496
 Umgebungs-Codepage 252, 258
 umkehrbare Transformation 126
 unerwünschte Übereinstimmung 98

- ungültiger Wert 26
Unicode 21
Unicode-System 251, 252, 254, 255
Uniform Resource Locator → URL
Unit 278, 289, 478
 Detail 298
 ID 298
 Objekt 273
 Referenz 284, 285
 Typ 295
Unit Browser 502
Unterausdruck 85
 ohne Registrierung 108
Untergruppe 84
Unterprogramm 396
Unter-Template 148
Update-Prozess 537
URL 88
User Exit 540
-
- V**
- Variable
 Anzeigen 423
 globale 424
variable Länge 20
Variante
 Aktivieren 457
 Deaktivieren 459
 Pflege 458
Verarbeitungsart 226
Verbindung
 interne 248
 logische 302
Verbindungsparameter 205
Verbuchung 460
 Debugging 461
Verbuchungs-Task 271, 288
Verbuchungs-Workprozess 202
Verdrängbarkeit 192
Vererbung 543
Vererbungshierarchie
 Anzeigen 435
verfügbare Erweiterung 556
Vergleich
 interne Tabelle 485
Verkettungsoperator 64
Versionierung
 Gebietsinstanz 180
- Versionierung (Forts.)
 lange Sperrzeit 185
Verwendungs-nachweis 475
Verzweigungspunkt 105
virtuelle Queue 280
Voraballokation 517
Vorausschauoperator 93
 negativer 93
vorbelegtes Feld 375
vordefinierte Ausnahme 261
vordefinierte Destination 247
-
- W**
- W3C 122
Warnung 493
Wartung 446
Watchpoint 397, 399, 402, 430, 437
 Anlegen 398, 400, 430
 für interne Tabelle 431
 Tabellen-Header 404
 Tipp 432
 Variable 402
 Wertevergleich 437
Webservice 114
 ABAP 125
Werkzeug 413
Wert
 ungültiger 26, 29
 Untersuchen 349
 Vergleich 484
Wertehilfe
 Breakpoint 429
Wertesemantik 508
Wertevergleich 437
Workprozess 201, 268, 291, 304, 365, 366, 441
 Dialog-Workprozess 202
 Enqueue-Workprozess 202
 Hintergrund-Workprozess 202
 Spool-Workprozess 202
 Verbuchungs-Workprozess 202
World Wide Web Consortium → W3C
Wortgrenze 71
WRITE TO 43
WRITE_MEMORY_CONSUMPTION_FILE 528
Wurzelmenge 512
Wurzelobjekt 161

X

XHTML 127
XI → SAP NetWeaver Process Integration
XML 113, 115, 235
 Infoset 115
 kanonische Darstellung 119
 Kommunikation 117
 Parser 116
 Schema 117
 Stream-Reader 126, 152
 Tag 115
XML Browser 425
XPath 124
xRFC 249
xsequence 36
XSLT 122
 Programm 123
xstring 33

Z

Zeichendarstellung 251
Zeichendatentyp 21
Zeichenfolge 19
Zeichenkette
 Vergleich 485
Zeichenketten-Template 43
Zeichenklasse 63
Zeichenkonvertierung 247
Zeichenliteral 22
Zeichenmodus 25
Zeilen-Breakpoint 394
Zeitfeld 28
Zusammenhangskomponente 522
 starke 522, 523
zustandsloser Aufruf 210
Zuweisung 29
Zwei-Phasen-Commit 219