EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Technische Informatik

Diplomarbeit

Software Integration mit Java und XML unter CICS

Stefan Huster

Betreuer:

Begonnen am:
Beendet am:

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Tübingen, am 30. September 2010

Stefan Huster

Inhaltsverzeichnis

1 Einführung in JCICS

In diesem Abschnitt werden Sie Ihr erstes JCICS-Programm entwickeln und auf dem Mainframe installieren. Es dient als Grundlage für alle weiterführenden Kapitel, in denen Ihnen verschiedene Techniken der Softwareintegration mit Java und XML unter CICS gezeigt werden. Am Ende dieses Kapitels werden Sie in der Lage sein, unter Anleitung JCICS-Programme auf Ihrem System zu entwickeln und diese anschließend selbstständig auf einem Mainframe unter CICS zu installieren.

1.1 Hello JCICS

In diesem Abschnitt werden Sie eine JCICS-Variante des klassischen HelloWorld-Programms erstellen. Das hier erstellte Programm dient im Wesentlichen dazu Ihnen zu zeigen, wie JCICS-Anwendungen auf dem Mainframe portiert und installiert werden.

Erstellen Sie zu Beginn in Ihrer Entwicklungsumgebung ein neues Java-Projekt “Tutorial1”. Im Anschluss legen Sie innerhalb des Projekts eine neue Java-Klasse “HelloWorld” an. Als Paketnamen sollten Sie Ihren Benutzernamen, gefolgt von “tutorial1”, verwenden. In diesem Beispiel entspricht dieser “prak500.tutorial1”.
Das Listing 1.1 zeigt Ihnen den gesamten Code der HelloWorld-Klasse. In den folgenden Absätzen werden wir diesen detailliert besprechen und die Unterschiede zu einem HelloWorld-Programm hervorheben, das Sie auf Ihrem eigenen System auf einer Standard-JVM ausführen würden.

Direkt zu Beginn des Programms fällt Ihnen unter (1) auf, dass das Argument der `main`-Methode nicht wie gewohnt eine Stringarray (`String[]`) ist, sondern ein Objekt vom Typ `CommAreaHolder`. Die Commarea, welche durch dieses Objekt gekapselt wird, ist ein speziell reservierter Speicherbereich. Dieser kann von der CICS-Laufzeitumgebung unter anderem für die Ein- und Ausgabe verwendet werden. In Kapitel [YY] werden wir noch genauer auf die Verwendung der Commarea eingehen.

In Schritt (2) erzeugen und speichern wir eine Referenz auf das `Task`-Singelton (vgl. Singelton-Pattern [GHJV94]). Dieses Objekt dient auch der Kapselung der IBM JCICS API und ermöglicht es auf Informationen der Laufzeitumgebung zuzugreifen.

Unter CICS ist der Stream `System.out` nicht mit der Konsole oder einem Terminal verbunden, wie es bei einem heimischen System der Fall wäre [Bur09]. Die Ausgabe in das CICS-Terminalfenster wird über eine Instanz des `PrintWriter`-Objekts ermöglicht. Diese stellt einen Ersatz für den `System.out`-Stream dar und kann orthogonal zu diesem verwendet werden. Die `PrintWriter`-Instanz wird über eine Factory-Methode (vgl. Factory-Pattern [GHJV94]) des `Task`-Objekts in Schritt (3) bereitgestellt.

Ein weiteres Beispiel für die vom `Task`-Objekt bereitgestellten Informationen sehen Sie unter Punkt (4). Dort lesen wir aus dem `Task`-Objekt die Namen des Benutzers und des aktuell ausgeführten Pro-

gramms mit der dazugehörenden Transaktion aus.
Die korrekte Ausführung der in Schritt (4) getätigten Abfrage kann jedoch nicht immer garantiert werden. Aus diesem Grund ist es notwendig, diese Statements in einem `try-catch`-Block einzufassen, wie Sie ihn bei Punkt (5) sehen können. Dieser soll eventuell auftretende Exceptions abfangen.

1.2 Portierung der Java-Klassen auf den Mainframe

In diesem Abschnitt werden Sie lernen, wie Sie die geschriebenen Java-Programme auf dem Mainframe portieren können.
Auf Ihrem System finden Sie nach der Kompilierung für jede Java-Klasse zwei verschiedene Dateien. Eine Datei hat die Dateierweiterung `.java` und enthält den Quelltext der Klasse. Die zweite Datei trägt die Endung `.class` und enthält den kompilierten Bytecode, der später von der JVM ausgeführt wird. In welchem Ordner die `class`-Dateien gespeichert werden, hängt von der verwendeten Entwicklungsumgebung ab. Eclipse und Netbeans zum Beispiel erstellen hierfür einen seperaten Ausgabeordner innerhalb der angegebenen Workspace.

Öffnen Sie zunächst den Ordner der verwendeten Workspace in einer Dateiverwaltung Ihrer Wahl (Explorer, Finder...). Dieser enthält für jedes angelegte Projekt einen Unterordner entsprechend des Projektnamen. Eclipse legt innerhalb des Projektordners zwei weitere Ordner an. Einer heißt „src“ und einer „bin“. Der bin-Ordner enthält die kompilierten Bytecodevarianten Ihrer Java-Klassen. Für die spätere Installation der JCICS-Anwendungen müssen Sie diese auf den Mainframe laden.

Die kompilierten Klassendateien lassen sich am einfachsten mit Hilfe eines FTP-Klients auf den Mainframe übertragen (vgl. Abschnitt [YY]). Verbinden Sie sich daher über einem FTP-Klienten Ihrer Wahl mit dem Mainframe und navigieren Sie in den Ordner `/u/prak500/classes`.

Achtung: Dieses eine Mal müssen Sie `prak500` **nicht** durch Ihren eigenen Benutzernamen ersetzen! Der Ordner `/u/prak500/classes` wurde als `CLASSPATH` in das JVM-Profil des Mainframes eingetragen. Sie verfügen mit Ihrem Account nicht über die erforderlichen Rechte, um Ihren eigenen Homeordner als `CLASSPATH` einzutragen.

In den Ordner `/u/prak500/classes` können Sie nun die gesamte Struktur des bin-Ordners laden. Diese sollte aus zwei Ordnern und einer class-Datei bestehen. Die Ordner wurden beim Kompilieren durch den Java-Kompiler automatisch angelegt. Ihre Namen entsprechen den gewählten Paketnamen. In diesem Beispiel wurde als Paketname “prak500.tutorial1” gewählt. Für diesen Paketpfad wird zuerst ein Ordner “prak500” angelegt, der selbst wiederum einen Ordner “tutorial1” enthält. In diesem befindet sich dann die class-Datei des HelloWorld-Beispiels. Sollte Ihre Entwicklungsumgebung diese Ordnerstruktur nicht automatisch angelegt haben, ist es notwendig dies auf dem Mainframe per Hand zu tun. Auf jeden Fall sollte die von Ihnen hochgeladene class-Datei am Ende in dem Ordner `/u/prak500/classes/prak500/tutorial1` liegen, wobei Sie das zweite “prak500” durch Ihren Benutzernamen ersetzen müssen.
Diese Struktur erlaubt eine saubere Trennung aller Bearbeitungen dieses Kurses. Ihre penible Einhaltung ist aus diesem Grund zwingend erforderlich.

Listing 1.1: HelloWorld in JCICS

```
1  package prak500;
2
3  import java.io.PrintWriter;
4
5  import com.ibm.cics.server.CommAreaHolder;
6  import com.ibm.cics.server.InvalidRequestException;
7  import com.ibm.cics.server.Task;
8
9  public class HelloWorld
10 {
11   //## (1)
12   public static void main ( CommAreaHolder cah )
13   {
14     //## (2)
15     Task task = Task.getTask();
16     //## (3)
17     PrintWriter out = task.out;
18
19     try {
20       //## (4)
21       out.println("Hello␣" + task.getUSERID() + "␣welcome␣to␣CICS!");
22       out.println();
23       out.println("This␣is␣program␣" + task.getProgramName());
24       out.println("Transaction␣name␣is␣" + task.getTransactionName());
25
26     }
27     //## (5)
28     catch (InvalidRequestException e) {
29       e.printStackTrace();
30     }
31
32   }
33 }
```

1.3 CICS Ressource Definition

CICS verarbeitet Anwendungen mit Hilfe verschiedener Ressource-Definitionen. Für die Installation eines JCICS-Programms als Transaktionen benötigen Sie die Ressourcen:

- `PROGRAM`
- `TRANSACTION`

Die Ressource `PROGRAM` beschreibt ein unter CICS ausführbares Programm und verweist dafür auf die von Ihnen erstellte Java-Klasse. Eine Transaktion wird über die Ressource `TRANSACTION` erstellt. Sie kann direkt vom CICS-Transaktionsmanager ausgeführt werden und arbeitet selbst wiederum mit der `PROGRAM`-Definition zusammen.
In diesem Abschnitt lernen Sie, wie Sie beide Ressourcen erstellen, installieren und CICS-Transaktionen ausführen.
Für die Bearbeitung der folgenden Schritte ist es notwendig, dass Sie sich unter CICS eingeloggt haben. Eine Anleitung dafür finden Sie in Abschnitt [YY].

1.3.1 Das CICS-Programm

Für die Definition, Bearbeitung, Entfernung und Installation der CICS Ressourcen ist die eingebaute CICS Transaktion `CEDA` zuständig. Für die Definition eines neuen Programms hat der Befehl folgende Syntax:

`CEDA DEFINE PROGRAM(`*Programmname*`) GROUP(`*Gruppenname*`)`

Den Programmnamen können Sie frei wählen. Seine maximale Länge beträgt acht Zeichen. Als *Gruppennamen* sollten Sie Ihren Benutzernamen angeben. In diesem Beispiel wäre dies `prak500`. CICS unterscheidet bei der Eingabe über das Terminal standardmäßig nicht zwischen Groß- und Kleinschreibung. Die gesamte Eingabe wird stattdessen in Großbuchstaben konvertiert. Für die Definition des HelloWorld-Programms sieht der Befehl wie folgt aus:

```
CEDA DEFINE PROGRAM(JHELLOW) GROUP(PRAK500)
```

Nachdem Sie den Befehl durch das Drücken der Entertaste bestätigt haben, öffnet sich eine Liste mit Einstellungsmöglichkeiten der Programmdefinition. Einen Ausschnitt dieser Liste sehen Sie in Abbildung 1.1. Wahrscheinlich wird die Liste den Anzeigebereich Ihres Terminals überschreiten. Zum Scrollen können Sie die Tasten `F7` und `F8` verwenden.

Die meisten Einstellungen der Definition müssen nicht geändert werden. Folgende Angaben sind jedoch abweichend von ihrern Standardwerten (in Klammern) zu definieren:

Datalocation:	Any (Below)
Concurrency:	Threadsafe (Quasirent)
JVM:	Yes (No)
JVMClass:	*Pfad zur Klasse*

In das Feld `JVMCLASS` müssen Sie den Pfad zu der Java-Klasse angeben, die Sie für Ihre CICS-Anwendung verwenden wollen. In Projekten, die aus mehreren Klassen bestehen, reicht hier die

Abbildung 1.1: Einstellungsmöglichkeiten für ein CICS Program

Angabe der Klasse, welche die `main`-Methode enthält. Das Feld selbst wird aktiviert, nachdem Sie die Einstellung `JVM=Yes` gesetzt haben. Im Unterschied zu den anderen Einstellungen wird im Feld `JVMCLASS` zwischen Groß- und Kleinschreibung unterschieden. Ausgangspunkt des anzugebenden Pfades ist immer der Ordner `/u/prak500/classes/`. Dies bedeutet, als Pfad zu der HelloWorld-Klasse können Sie *Benutzername*`.tutorial1.HelloWorld` eintragen. In diesem Beispiel entspricht dies der Angabe `prak500/tutorial1.HelloWorld`. Die Angabe der Dateiendung `.class` ist nicht notwendig.

Nachdem Sie alle Einstellungen getätigt haben, können Sie die Definition durch die Betätigung der Eingabetaste speichern. Im Anschluss sollte die Meldung `DEFINE SUCCSESSFUL` in der unteren linken Ecke des Terminals erscheinen (Abbildung 1.2a).

Beenden Sie die CEDA-Umgebung durch Drücken der `F3`-Taste. Diese agiert ähnlich zu der `ESC`-Taste unter Windows. Sie erlaubt es Ihnen immer zu der nächst höheren Anwendungsebene zu springen. Auf der höchsten Ebene sehen Sie nur noch die Meldung `SESSION ENDED`.

Nach der Definition müssen Sie das Programm noch installieren. Dazu geben Sie folgenden Befehl ein:

`CEDA DEFINE PROGRAM(JHELLOW) GROUP(`*Gruppenname*`)`

Im Anschluss sollte die Meldung `INSTALL SUCCESSFUL` erscheinen. Die Angabe *Gruppenname* müssen Sie natürlich wieder durch Ihren Benutzernamen ersetzen. In künftigen Installationen müssen Sie `JHELLOW` durch den Namen des zu installierenden Programms ersetzen.

1.3.2 Die CICS-Transaktion

Theoretisch ist es schon möglich, das oben installierte Programm manuell zu starten. Unser Ziel ist es jedoch, dieses in einer Transaktion einzubinden. Für die Definition einer Transaktion verwenden wir den CEDA-Befehl mit folgender Syntax:

`CEDA DEFINE TRANSACTION(`*`Transaktionsname`*`) GROUP(`*Gruppenname*`)`

Der Name einer Transaktion besteht genau aus vier Zeichen. Auf Grund Ihrer Benutzerrechte ist es Ihnen nur erlaubt Transaktionen auszuführen, die mit einem `X` beginnen. Wir wählen daher den Namen `XJHW`. Der Befehl zur Definition sieht daher wie folgt aus:

`CEDA DEFINE TRANSACTION(XJHW) GROUP(`*Gruppenname*`)`

Wie bei der Definition eines Programms auch erscheint nach dem Absenden der Anweisung durch Drücken der Eingabetaste eine Liste mit Einstellungsmöglichkeiten. In dieser sind folgende Werte abweichend von ihrer Standardbelegung zu definieren:

Program: *Programmname*
Taskdataloc: Any

Der hier angegebene Programmname muss mit dem aus Abschnitt 1.3.1 übereinstimmen. Die Definition kann durch die Betätigung der Eingabetaste gespeichert werden. Es sollte wieder die Meldung `DEFINITION SUCCESFULL` in der unteren linken Ecke des Fensters erscheinen. Verlassen Sie die Anwendungsebene durch Drücken der `F3`-Taste.

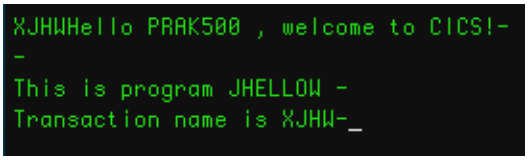
Nach der Definition folgt wieder die Installation. Diese erfolgt durch:

`CEDA INSTALL TRANSACTION (XJHW) GROUP(`*Gruppenname*`)`

Den Gruppennamen müssen Sie wieder durch ihren Benutzernamen ersetzen. In späteren Installationen muss die Angabe `XJHW` durch den Namen der zu installierenden Transaktion ersetzt werden.

Zum Starten der Transaktion verlassen Sie ggf. alle zurzeit verwendeten Transaktionen. Löschen Sie den aktuellen Bildschirm durch den entsprechenden Befehl Ihres TN3270-Emulators. Geben Sie dann den Namen der Transaktion ein und drücken Sie die Eingabetaste. In diesem Beispiel lautet der Befehl `XJHW`. Die entsprechende Ausgabe ist in Abbildung 1.2b dargestellt.

(a) Erfolgmeldung nach einer *DEFINE*-Anweisung



(b) Ausgabe der Transaktion

Abbildung 1.2: Screenshots der CICS-Interaktion

1.3.3 Hinweise

In [IBM99] können Sie mehr über die verschiedenen Einstellungsmöglichkeiten der unterschiedlichen CICS-Ressourcen erfahren.

Anstelle der getrennten Installation von Programm und Transaktion ist es auch möglich, zuerst alle notwendigen Ressourcen zu definieren und im Anschluss die gesamte Gruppe zu installieren. Dafür können Sie folgenden Befehl verwenden:

`CEDA INSTALL GROUP(`*Gruppenname*`)`

Dies ist jedoch an einigen Stellen mit Vorsicht zu genießen, da einige Ressourcen nur einmal installiert werden dürfen und jede Wiederholung der Installation zu einem Fehler führt.

Sollten Sie während der Verwendung der verschiedenen CEDA-Befehle einen Fehler erhalten, können Sie sich durch das Drücken der `F9`-Taste eine genauere Fehlermeldung anzeigen lassen.

Zusammenfassung

JCICS-Programme bedienen sich einer anderen Systemschnittstelle als herkömmliche Java-Programme. Zum Beispiel kann die Eingabe einer main-Methode neben einer String-Array auch einen `Commareaholder` entgegennehmen. Die Ausgabe auf das Terminalfenster erfolgt nicht über den gewöhnlichen `System.out`-Befehl, sondern über einen `PrintWriter`. Dieser wird vom `Task`-Objekt zur Verfügung gestellt, das zusätzlich auch weitere Informationen der Laufzeitumgebung bereitstellt.
Nach der lokalen Entwicklung eines Java-Programms müssen die Java-Class-Dateien auf den Mainframe geladen werden. Dies erfolgt am einfachsten mit Hilfe eines FTP-Klienten. Legen Sie die `class`-Dateien in den entsprechenden Unterorder des Java-Classpath.
Für die Ausführung eines Java-Programms unter CICS ist es notwendig, ein CICS-Programm und eine CICS-Transaktion zu erstellen. Beides sind CICS-Ressourcen. Diese müssen zuerst definiert und im Anschluss installiert werden.

Aufgaben

1. *Entwickeln Sie ein eigenes Hello-Cics-Programm auf Basis des in vorgestellten Codes. Erweitern Sie die Ausgabe um die Namen Ihrer Teammitglieder.*
2. *Informieren Sie sich im CICS 3.2 Informationcenter (http://publib.boulder.ibm.com/infocenter/cicsts/v2r3/index.jsp) über weitere Umgebungsvariablen die durch das ¿¿TASK¡¡-Objekt bereitgestellt werden. Nennen Sie mindestens eine weitere und erklären Sie deren Bedeutung.*
3. *Informieren Sie sich im CICS Transaction Handbuch über weitere Klassen, die der Kapselung der Systemschnittstelle dienen. Nennen Sie eine und erörtern kurz deren Aufgabe.*

2 Externe Programmaufrufe und Datenaustausch unter JCICS

Programme unter CICS können mit einer Vielzahl verschiedener Sprachen entwickelt werden, dazu zählen neben Java unter anderem noch C++, Cobol oder PL/1. Die Systemschnittstelle von CICS ermöglicht es auf sehr einfache Weise, Kommunikationswege zwischen allen unter CICS installierten Programmen aufzubauen, unabhängig von der Sprache, mit der sie implementiert wurden. Für den Anwendungsentwickler stellt dies eine große Vereinfachung dar, da er ansonsten auf Verfahren wie zum Beispiel dem Remote Procedure Call (RPC), der Java Methode Invocation (JMI) oder der Common Object Request Broker Architecture (CORBA) zurückgreifen müsste. Das Problem dieser Techniken ist, dass sie jeweils nur von einer Teilmenge der verfügbaren Sprachen unterstützt werden.

Die Hauptaufgabe der Softwareintegration ist es, die Kommunikation zwischen verschiedenen Systemen und Anwendungen zu ermöglichen. In diesem Kapitel werden Sie die CICS-internen Methoden kennenlernen, mit denen Sie eine solche Kommunikation aufbauen können. Im ersten Abschnitt werden Ihnen Befehle vorgestellt, mit denen Sie andere unter CICS installierte Programme aufrufen können. Im Anschluss erhalten Sie einen Einblick in Techniken für den Datenaustausch zwischen verschiedenen CICS-Programmen. Am Ende dieses Kapitels wird Ihnen noch der praktische Einsatz der vorgestellten Methoden und Verfahren in einem kurzen JCICS-Beispiel demonstriert.

2.1 Externe Programmaufrufe

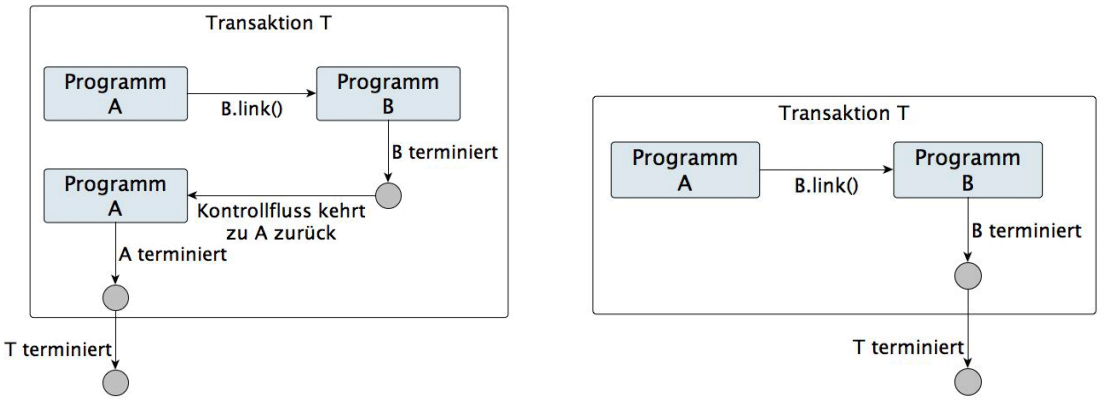
In diesem Abschnitt lernen Sie, wie Sie aus Ihrer CICS-Anwendung andere CICS-Programme aufrufen. Die Systemschnittstelle von CICS stellt Ihnen dafür zwei Befehle zur Verfügung: `LINK` und `XCTL`. Unter JCICS sind beide Befehle als Methoden der Klasse `com.ibm.cics.server.Program` aus der externen Bibliothek `dfjcics.jar` realisiert. Nach ihrer Ausführung werden die `link`- und die `xctl`-Anweisungen direkt unter CICS verarbeitet. Mit ihnen kann daher jedes CICS-Programm aufgerufen werden, unabhängig davon, in welcher Sprache es geschrieben wurde. An dieser Stelle werden Ihnen die konzeptionellen Unterschiede beider Methoden vorgestellt. Die genaue Syntax und die Verwendung unter JCICS werden Sie innerhalb des Abschnitts 2.3 kennenlernen.

2.1.1 LINK

Betrachten wir ein Szenario, bestehend aus einer Transaktion `T`, einem Programm `A` und einem Programm `B`. Innerhalb dieser Transaktion muss Programm `A` ein anderes Programm `B` aufrufen. Mit der abstrakten Anweisung `B.link()` kann das Programm `A` die Laufzeitumgebung auffordern, die momentane Programmausführung in der `main`-Methode von Programm `B` fortzusetzen. Terminiert Programm `B`, springt der Kontrollfluss zurück zu Programm `A` und die Programmausführung wird hinter dem Aufruf von `B.link()` in `A` fortgesetzt. Die gesamte Transaktion `T` terminiert demnach erst, wenn auch Programm `A` terminiert. Das Verfahren der externen Programmaufrufe kann natürlich transitiv fortgesetzt werden, indem auch Programm `B` weitere Programme aufruft.

2.1.2 XCTL

Die Aufgaben des `xctl`-Befehls sind identisch zu denen des `link`-Befehls. Auch die Syntax beider Anweisungen ist orthogonal zu verwenden. Der Unterschied beider Methoden liegt im Programmablauf. Betrachten wir hierfür noch einmal das Szenario aus Abschnitt 2.1.1. In diesem Beispiel hat Programm `A` ein anderes Programm `B` aufgerufen. Gehen wir an dieser Stelle davon aus, dass der Aufruf nicht via `B.link()`, sondern mit `B.xctl()` erfolgt ist. In diesem Fall kehrt der Kontrollfluss nach der Terminierung von `B` nicht mehr zu `A` zurück. Stattdessen terminiert die gesamte Transaktion `T`. Dies bedeutet, dass der `xctl`-Befehl die gesamte Ablaufkontrolle an das aufgerufene Programm überträgt. Den Unterschied zwischen `LINK` und `XCTL` finden Sie noch einmal in der Abbildung 2.1 illustriert.



(a) Der Kontrollfluss einer `link`-Anweisung

(b) Der Kontrollfluss einer `xctl`-Anweisung

Abbildung 2.1: Ablauf des Kontrollflusses bei `LINK`- und `XCTL`-Anweisungen

2.2 Datenaustausch

In diesem Abschnitt lernen Sie, wie Sie Daten zwischen verschiedenen CICS-Programmen austauschen können. Dieser Prozess spiegelt die eigentliche Kommunikation zwischen verschiedenen Softwarelösungen wieder. Diese herzustellen ist ein Ziel der Softwareintegration. Wie im vorangegangenen Abschnitt können die hier beschriebenen Verfahren dazu verwendet werden, Daten zwischen beliebigen CICS-Programmen auszutauschen, unabhängig davon, in welcher Sprache sie geschrieben wurden.

2.2.1 Commarea

Die Commarea ist ein speziell reservierter Speicherbereich. Jedes Programm, das z.B. über eine `link`-Anweisung aufgerufen wird, erhält eine Referenz auf eine übergebene Commarea. Auf diese Weise kann der speziell reservierte Speicher zum Austausch von Daten verwendet werden. Sowohl das aufrufende als auch das aufgerufene Programm verfügen über eine Referenz auf den selben Speicherbereich. Aus diesem Grund kann eine Commarea zur Ein- und zur Ausgabe verwendet werden. Die Größe des reservierten Speicherbereichs wird durch das aufrufende Programm festgelegt und

kann später nicht mehr geändert werden. Ihre maximale Größe beträgt 32kB [Bur09]. Die Übergabe der Daten erfolgt auf Byte-Ebene. Das heißt, auch unter JCICS werden Byte-Arrays ausgetauscht. Es liegt beim Anwendungsentwickler, diese in das richtige Format zu konvertieren und entsprechende Fehlerquellen zu berücksichtigen. Dieses Vorgehen ist sehr systemnah und auf einem sehr niedrigen Abstraktionslevel. Dennoch bildet diese Methode die Basis für jeden Kommunikationspfad innerhalb von CICS.

2.2.2 Channels und Container

Eine andere und auch etwas modernere Variante des Datenaustauschs zwischen verschiedenen CICS-Programmen ist die Verwendung von Channels und Container.
Ein Container dient der Kapselung der zu sendenden Daten. Man könnte ihn mit einem Umschlag für einen Brief vergleichen. Technisch gesehen ist ein Container nicht viel mehr als eine benannte Commarea. Ein Channel ist eine Verbindung zwischen zwei verschiedenen CICS-Programmen, in dem Container transportiert werden können. Der Kanal entspricht daher dem Briefträger, der einen Umschlag vom Sender zum Empfänger bringt. In Abschnitt 2.3 werden Sie die Verwendung von Container und Channels im praktischen Einsatz sehen.

Ein Channel bietet gegenüber der Verwendung der Commarea folgende Vorteile (Auszug aus [IBMa]):

- Channels sind nicht auf eine maximale Größe von 32kB beschränkt. Es besteht keine Limitierung bzgl. der Anzahl und Größe der übertragenen Container. Die einzige Beschränkung ist durch den physisch vorhandenen Speicher des Mainframes gegeben.
- Ein Channel kann Container verschiedener Typen enthalten. Dies ermöglicht, Daten strukturierter zu übertragen, vergleicht man das Vorgehen mit der Commarea, die nur einen einzigen Typ unterstützt.
- Channels sind standardisiert und werden von allen unter CICS- lauffähigen Programmiersprachen unterstützt.

Auf der anderen Seite gilt es auch einige Punkte bei der Verwendung von Channels zu beachten:

- Verwendet man Channels zur Datenübertragung zwischen CICS-Anwendungen, die auf unterschiedlichen Mainframes betrieben werden, kann der Transport größerer Datenmengen zu erheblichen Performanceverlusten führen.
- Für die Übertragung von Daten mit Hilfe von Channels werden mehr Systemressourcen benötigt als bei einer Übertragung via Commarea. Dies liegt daran, dass Container-Objekte in unterschiedlichen Speicherbereichen im System hinterlegt sein können. Dies erfordert unter Umständen häufiger langsame Lesezugriffe. Des Weiteren werden Daten innerhalb der Commarea nur als Referenz übergeben. Die Daten in einem Container werden kopiert.

Um einen Container innerhalb eines Channels zu erzeugen, verwenden CICS-Programme den API-Befehl:

```
EXEC CICS PUT CONTAINER(Name des Containers) CHANNEL(Name des Channels)
```

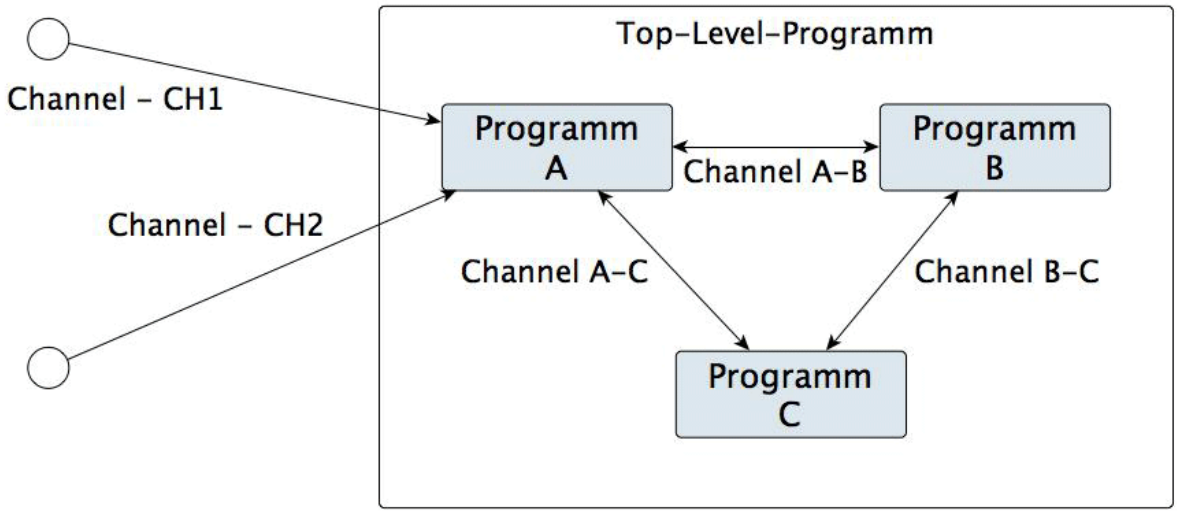


Abbildung 2.2: Channel Multiplexing

Die Umsetzung innerhalb eines JCICS-Programms wird in Abschnitt 2.3 erläutert. Die Übertragung der Daten erfolgt mit Hilfe der oben beschriebenen `link`- oder `xctl`-Kommandos. Das Empfängerprogramm kann die Container über folgenden Befehl lesen:

`EXEC CICS GET CONTAINER(`*Name des Containers*`)`

Der Name des Channels wird automatisch impliziert, da jedes Programm zum Zeitpunkt des Aufrufs genau einen Channel zugewiesen bekommt.

Ein weiterer großer Vorteil von Channels ist die damit verbundene Erweiterungsmöglichkeit von Programmen. In Abbildung 2.2 sehen Sie ein Programm illustriert, welches über zwei verschiedene Channels aufgerufen werden kann. In der IBM-Literatur wird dieser Zustand als “loose coubling” [IBM08] bezeichnet. Damit ist gemeint, dass das Interface zum aufrufenden Klienten nicht eindeutig und fest definiert sein muss, sondern erst zur Laufzeit über die Auswahl des Channels bestimmt werden kann. Der Klient wählt einen entsprechenden Kanal des Hauptprogramms aus. Dieses interpretiert die Daten je nach Kanal anders und leitet sie unter Umständen an das entsprechende Unterprogramm weiter. In Abschnitt 2.3 werden Sie dieses Vorgehen in einem kleinen Beispiel im Einsatz sehen.

2.2.3 Channels vs. Commarea

Wie Sie gesehen haben, verfügen beide Verfahren über ihre Vor- und Nachteile. Die Commarea ist sehr leichtgewichtig und aufgrund des Datenaustauschs auf Basis von Referenzen sehr performant und zudem schnell zu implementieren. Auf der anderen Seite bieten Channels die Möglichkeit, Daten strukturiert zu übergeben und Programme sehr leicht zu erweitern. Dieser Komfort wird dafür mit einem erhöhten Verbrauch von Systemressourcen bezahlt.

Für die Entscheidung der verwendeten Technik stellt IBM in [Bur09] drei einfache Kriterien bereit:

- Ist es notwendig, größere Datenmengen jenseits der Grenze von 32kB zwischen Programmen zu übertragen, sind Channels und Container die eindeutig bessere Wahl.
- Arbeiten Sie mit bestehenden Programmen, die eine Commarea zur Kommunikation verwenden und nur geringe Datenmengen austauschen, lohnt sich die Umstellung auf das Channel-Konzept unter Umständen nicht.
- Entwickeln Sie neue Projekte, ist die Verwendung des Channel-Konzepts in der Regel die bessere Entscheidung.

2.3 Anwendungsbeispiele

In den folgenden zwei Beispielen werden Sie sehen, wie die oben beschriebenen Techniken unter JCICS angewendet werden können.

2.3.1 LINK, XCTL, COMMAREA

In diesem Beispiel verwenden wir zwei Programme, von denen eins das andere einmal über die `link`- und einmal über die `xctl`-Anweisung aufruft. Zusätzlich wird dem aufgerufenen Programm über die Commarea eine Frage übermittelt. Das aufgerufene Programm wird dann die richtige Antwort in die Commarea schreiben.

Das Beispiel der `link`-Variante besteht aus zwei Programmen: “DoLink” und “CallMe”. Deren Quelltexte finden Sie in den Listings lst:DoLink und lst:CallMe. Wir beginnen mit “DoLink”:

Zu Beginn geben wir zur Bestätigung in (1) den Programmnamen (hier “DoLink”) des aufgerufenen Programms aus. Dies zeigt uns, dass der Kontrollfluss zurzeit von der Klasse `DoLink` gesteuert wird.
Als nächstes erzeugen wir in Schritt (2) eine Nachricht, die wir dem aufrufenden Programm senden möchten. Da die Commarea unter JCICS als Byte-Array repräsentiert wird, muss der String vor dem Senden konvertiert werden. Die Größe der Commarea ist damit automatisch festgelegt worden. Sie entspricht der Größe der gespeicherten Zeichenkette.
Unter Punkt (3) erzeugen wir eine neue Instanz der `Program`-Klasse. Dieser Typ wird von der JCICS API bereitgestellt und dient der Abstraktion über CICS-Programme. Dem erzeugten Programm weisen wir im Anschluss einen Namen zu. Dieser darf nicht willkürlich vergeben werden. Er muss mit dem Namen des aufzurufenden Programms übereinstimmen, der innerhalb der entsprechenden CICS-Ressource definiert wurde. Die Übereinstimmung des Klassennamens mit dem Programmnamen ist hingegen nicht vorgeschrieben. Mit dem Befehl `prog.link(commArea)` wird schließlich das zweite Programm des Beispiels aufgerufen. Dessen Analyse folgt im Anschluss an diese Besprechung.
Das von uns aufgerufene Programm wird, wie wir später sehen werden, eine Antwort in die übergebene Commarea schreiben. Diese wird in Schritt (4) ausgegeben.
Sowohl der `link`-Befehl als auch die Abfrage des Programmnamens unter Punkt (1) sind potentielle Fehlerquellen. Es könnte zum Beispiel passieren, dass das aufgerufene Programm nicht existiert. In diesem Fall würde die `link`-Anweisung eine Exception werfen. Diese werden bei Punkt (5) gefangen. Genauere Informationen über die möglichen Exceptions finden Sie in [IBMa] und [IBMb].

Als Nächstes widmen wir uns der Betrachtung des Quelltextes des aufgerufenen Programms. In diesem Beispiel heißt dieses `CALLME`. Die entsprechende Java-Klasse trägt den Namen `CallMe` und ist in Listing lst:CallMe abgedruckt.
Wie auch zuvor beginnen wir das Programm mit einem kurzen Lebenszeichen (1). Den Sinn und Zweck der `println`-Anweisungen besprechen wir später, wenn wir die Ausgabe betrachten und diese mit der des `xctl`-Beispiels vergleichen.
Die übergebene Commarea wird durch den `CommAreaHolder` als Parameter der `main`-Methode gekapselt. Dieser hat genau ein Klassenfeld namens `value` vom Type `byte[]`. Diesem Feld weisen wir in Schritt (2) die richtige Antwort auf die übertragene Frage zu.

Nachdem der Quelltext beider beteiligten Klassen besprochen wurde, wird es Zeit, einen Blick auf die Ausgabe der entsprechenden Transaktion zu werfen. Diese sehen Sie in Abbildung 2.3. Bereits bei der oberflächlichen Betrachtung sind zwei Punkte besonders auffällig:

1. Es fehlt bei der Ausgabe der Transaktion in Zeile 2, 3 und 4 jeweils der Anfangsbuchstabe.

2. Neben der richtigen Antwort 42 [Ada79] ist in Zeile 5 noch der Rest der Anfrage sichtbar.

Die Erklärung der ersten Beobachtung führt uns direkt zurück zu der Frage, wieso in der Klasse `CallMe` so viele `println`-Anweisungen verwendet wurden. Die `println`-Anweisungen beginnen ihre Ausgabe in der ersten Zeile des Terminalfensters. In dieser Zeile steht i.d.R. jedoch der Name der Transaktion, hier `XDOL`. Aus diesem Grund begannen wir in der Klasse `DoLink` auch die Ausgabe mit einer leeren Zeile. Übergibt man den Kontrollfluss via `LINK` oder `XCTL` an ein anderes Programm, welches ebenfalls Ausgaben auf das Terminalfenster druckt, beginnt die Ausgabe wieder in der ersten Zeile. Dies liegt daran, dass dem aufgerufenen Programm eine neue Task-Umgebung zugewiesen wird. Aus diesem Grund gibt die Klasse `CallMe` zuerst drei leere Zeilen aus, da ansonsten die Ausgabe des `DOLINK`-Programms überschrieben werden würde. Die Ausgabe von `println()` ist jedoch nicht leer, sondern wird im Terminal als "-" dargestellt, um jeweils das Ende einer Zeile zu markieren. Das "-" überschreibt auch den ersten Buchstaben der Zeilen 2–4.
Der Grund für die zweite Beobachtung wurde bereits im Abschnitt 2.2.1 besprochen. Dort wurde festgestellt, dass die Commarea immer eine feste Länge hat, die vom aufrufenden Programm festgelegt wird. In diesem Beispiel entspricht die Größe der Commarea genau dem Platz, der durch die Anfrage "the answer to life the universe and everything" benötigt wird. Die Antwort "42" hingegen ist kürzer und füllt daher nicht die gesamte Commarea aus. Der nicht überschriebene Speicher bleibt unverändert. Dies führt dazu, dass ein Teil der Anfrage auch noch in der Antwort zu lesen ist. Es ist die Aufgabe des Anwendungsentwicklers, dafür zu sorgen, dass gegebene Bedingungen an den Speicherbereich der Commarea erfüllt werden. Eine mögliche Bedingung für dieses Beispiel wäre gewesen, dass die Antwort keinen Teil der Anfrage mehr enthalten darf. In diesem Fall hätte der nicht überschriebene Teil der Commarea manuell gelöscht werden müssen.
Bevor wir unsere Aufmerksamkeit auf die Ausgabe der `xctl`-Variante lenken, betrachten wir noch kurz den Quellcode dieser Alternative. Die entsprechende Klasse heißt `DoXctl` (Listing 2.1) und das CICS-Programm `DOXCTL`. Der Quelltext der Java-Klasse ist an dieser Stelle mehr der Vollständigkeit zuliebe aufgeführt, da die Änderungen verglichen zu der `link`-Varainte nur minimal ausfallen. Die einzigen Unterschiede sind unter Punkt (3) und Punkt (5) zu finden. Im ersten Fall wird einfach anstelle von `prog.link(commArea)` `prog.xctl(commArea)` verwendet. Des Weiteren ist die Anzahl der bei Punkt (5) abzufangenden Exceptions weniger geworden. Die Gründe für die unterschiedliche Anzahl der potentiell auftretenden Exceptions werden hier nicht weiter betrachtet, können

bei Bedarf jedoch in [IBM08] nachgelesen werden.

Die Auswirkung der `xctl`-Anweisung kann direkt in der Ausgabe in Abbildung 2.3b verfolgt werden. Vergleicht man diese mit der in Abbildung 2.3a gezeigten Ausgabe der `DOLINK` Transaktion, fällt auf, dass `DOXCTL` keine vierte Zeile ausgibt. Dies liegt daran, dass die entsprechende Transaktion `DOXCTL` nach der Ausführung von `CALLME` terminiert und nicht wie zuvor zum aufrufenden Programm zurückkehrt. Das heißt, die Zeilen unter Punkt (4) in der `DoXctl`-Klasse werden aufgrund der `xctl`-Anweisung nie erreicht und ausgeführt.
Ein weiterer kleiner Unterschied zu der Ausgabe von `DOLINK` ist das Vorhandensein der ersten Zeichen in Zeile 2 und 3. Dies hat jedoch nichts mit der `xctl`-Methode zu tun, sondern ist das Resultat eines vorangestellten Leerzeichens jeder `println`-Anweisung in `DoXctl`.

```
XDOL◊
◊his is program:DOLINK  ◊
◊y question:the answer to life the universe and everything◊
◊his is program CALLME◊
And the answer is: 42e answer to life the universe and everything◊
```

(a) Ausgabe der Transaktion XDOL

```
XDOX◊
◊This is program:DOXCTL  ◊
◊My question:the answer to life the universe and everything◊
This is program CALLME◊
```

(b) Ausgabe der Transaktion XDOX

Abbildung 2.3: Ausgabe der externen Programmaufrufe

2.3.2 Channels und Container

Im zweiten Beispiel betrachten wir die Kommunikation zwischen verschiedenen CICS-Programmen unter Verwendung von Channels und Container. Des Weiteren wird Ihnen gezeigt, wie Sie über einen Channel-Multiplexer mit einem Programm verschiedene Klienten bedienen können.

Diese kleine Anwendung besteht aus drei verschiedenen Klassen: `NeedEuro`, `NeedDollar` und `MoneyConverter`. Die ersten beiden Klassen erfüllen die Rolle der Klienten, während die dritte Klasse die Rolle eines Serviceproviders einnimmt. Die Idee ist, dass die Klienten jeweils entweder einen Euro- oder einen Dollar-Betrag an die Serviceklasse schicken und als Antwort den entsprechenden Betrag in der jeweils anderen Währung erhalten [1].

Damit der Serviceprovider beide Anfragen unterscheiden kann, verwenden die Klienten Channels mit unterschiedlichen Namen. Natürlich wären an dieser Stelle auch andere Möglichkeiten zur Unterscheidung denkbar. Streng genommen wären diese sogar allein durch unterschiedlich benannte Container möglich, jedoch würde eine solche Optimierung dieses Beispiel zunichte machen.

[1] Bitte entschuldigen Sie, dass der hier verwendete Umrechnungskurs ggf. zum Zeitpunkt Ihrer Bearbeitung nicht mehr aktuell sein könnte. Im Zweifellsfall multiplizieren Sie den Betrag einfach mit 42.

Da beide Klassen bis auf ihren Namen und den Namen des verwendeten Channels identisch aufgebaut sind, begnügen wir uns hier mit der Betrachtung nur eines Quellcodes, dem der `NeedEuro`-Klasse in Listing 2.4. Den Quelltext der anderen Klasse finden Sie im Anhang.

In Schritt (1) erzeugen wir einen neuen Kanal. Dies erfolgt über eine Factory-Methode des `Task`-Objekts (vgl. Factory-Pattern [GHJV94]) mit dem Namen `createChannel`. Wir weisen dem Kanal den Namen "DollarToEuro" zu.
Als nächstes benötigen wir einen Container, den wir über den erzeugten Kanal versenden können. Auch dieser wird in Schritt (2) über eine Factory-Methode erstellt. Der Container erhält den Namen "DOLLAR".
Container sind vom Prinzip her nichts anderes als benannte Commareas. Aus diesem Grund können sie ebenfalls nur Daten als Byte-Array speichern. Dafür erstellen wir in Punkt (3) einen String mit dem zu konvertierenden Betrag als Wert. Im Anschluss wird der String über die `put`-Methode im Container gespeichert. Die Konvertierung in eine Byte-Array erfolgt durch die Methode automatisch.
Schritt (4) sollte Ihnen aus Abschnitt 2.3.1 vertraut vorkommen. Wir erzeugen eine Instanz des `Program`-Objekts und rufen über die `link`-Methode das externe CICS-Programm `MONCONV` auf. Dieses Mal übergeben wir der `link`-Anweisung jedoch keine Commarea, sondern den erzeugten Kanal (5).
Der Wert des Containers wird, ähnlich wie im ersten Beispiel, durch das aufgerufene Programm geändert. Der geänderte Wert wird in Schritt (6) mit der `get`-Methode neu ausgelesen und im Anschluss ausgegeben.

Betrachten wir als Nächstes, wie der Dollarwert durch den Serviceprovider in Euro umgerechnet wird. Den Quelltext der Klasse `MoneyConverter` finden Sie in Listing 2.6.

Zu allererst wird der Channel benötigt, mit dem dieses Programm aufgerufen wurde. Diesen erhalten wir von der Laufzeitumgebung (1), über die unter CICS durch das `Task`-Objekt abstrahiert wird.
Der Channel, den die `Task` zurückgegeben hat, kann unter Umständen auch `null` sein, zum Beispiel dann, wenn das Programm mit einer übergebenen Commarea aufgerufen wurde. Diese Möglichkeit muss überprüft werden (2), bevor mit jeglicher Verarbeitung des Channels begonnen werden kann.
Im Block (3), bestehend aus Block (3-a) und (3-b), findet das Multiplexen des Channels statt. Dafür wird zuerst der Name des übergebenen Channels erfragt und anschließend mit allen unterstützten Namen verglichen. Diese Serviceklasse unterstützt zwei verschiedene Channels: "EuroToDollar" und "DollarToEuro". Die Verarbeitung des Channels erfolgt im Anschluss innerhalb von Block (3-a) bzw. (3-b). Dort wird der erwartete Container mit `ch.getContainer()` aus dem Channel extrahiert und dessen Wert in das gewünschte Format konvertiert. In diesem Fall erwarten wir einen Betrag, den es in eine andere Währung umzurechnen gilt. Der Wert ist daher numerisch und kann mit `Double.valueOf()` konvertiert werden.
Für die Umrechnung in die entsprechende Währung sind die Methoden unter (4-a) und (4-b) verantwortlich. Deren Ergebnis wird mit der `put`-Methode des Containers zurückgeschrieben und kann im Anschluss vom aufrufenden Programm gelesen werden.

Die Ausgabe beider Programme sehen Sie in Abbildung 2.4.

```
XETD◊
Give100 Euros◊
Get130.0 Dollars◊
```

(a) Ausgabe Konvertierung Euro nach Dollar

```
XDTE◊
Give100 Dollars◊
Get76.92307692307692 Euros◊
```

(b) Ausgabe Konvertierung Dollar nach Euro

Abbildung 2.4: Ausgabe der beiden Klienten des Währungsrechners

Hinweis

Die Methoden `LINK` und `XCTL` können auch mit anderen Parametern aufgerufen werden, als die, die in diesem Beispiel verwendet wurden. Eine detaillierte Liste finden Sie in der IBM JCICS API [JAX10].

Channels können auch mehr als einen Container enthalten. Wie im Abschnitt 2.2.1 erwähnt wurde, ist deren Anzahl nur durch die Größe des physischen Speichers beschränkt. In Fällen, in denen mehrere Container übergeben werden, kann es sehr sinnvoll sein, über diese als Liste zu iterieren. Für diesen Zweck stellt die JCICS API einen `ContainerIterator` bereit. Dieser implementiert das bekannte `java.util.Iterator` Interface. Listing 2.7 zeigt ein Beispiel, entnommen aus [IBMa], für die Verwendung dieses Iterators.

Zusammenfassung

Die CICS API stellt zwei Methoden für den Aufruf externer CICS-Programme bereit: `LINK` und `XCTL`. Beide Methoden verfügen über eine ähnliche Schnittstelle. Die `link`-Methode ruft ein anderes Programm auf und kehrt nach dessen Terminierung wieder zum aufrufenden Programm zurück. Die `xctl`-Anweisung hingegen beendet die gesamte Transaktion, nachdem das aufgerufene Programm terminiert ist.
Für den Austausch von Daten zwischen CICS-Programmen existieren ebenfalls zwei verschiedene Verfahren: Datenaustausch über die Commarea und über Channels.
Die Commera ist ein spezieller, geteilter Speicherbereich zweier CICS-Programme. Die Daten werden als Referenz übergeben und müssen vom Type `byte[]` sein. Die Größe einer Commarea wird einmal festgelegt und kann maximal 32kB betragen.
Ein Channel überträgt Daten, die zuvor mit einem Container gekapselt wurden. Channel und Container können benannt und somit auch unterschieden werden. Auch hier werden Daten als Bytes transportiert. CICS gibt jedoch keine Obergrenze für die Anzahl der Container und deren Größe vor.
Sowohl die Methoden zum Aufruf externer Programme als auch die Verfahren zur Datenübertragung funktionieren zwischen allen unter CICS installierten Programmen.

Aufgaben

1. *Erklären Sie, warum im Beispiel 2.4 die `link`-Methode verwendet wurde und nicht die `xctl`-Anweisung.*

2. *Entwickeln Sie einen JCICS-Taschenrechner. Dieser soll die vier Grundrechenarten unterstützen. Ein zweites CICS-Programm soll die verschiedenen Methoden des Taschenrechners aufrufen und das berechnete Ergebnis auf der Konsole ausgeben. Entscheiden Sie selbst, welches Verfahren Sie für den Programmaufruf und für die Datenübertragung verwenden.*

Listing 2.1: Aufrufendes Programm mit LINK

```
1  public class DoLink {
2    public static void main( CommAreaHolder cah ) {
3      Task task = Task.getTask();
4      PrintWriter out = task.out;
5
6      try {
7        //## (1)
8        out.println();
9        out.println( "This␣is␣program:" + task.getProgramName() );
10
11       //## (2)
12       String myMsg = "the␣answer␣to␣life␣the␣universe␣and␣everything";
13       byte[] commArea = myMsg.getBytes();
14       out.println( "My␣question:" + myMsg );
15
16       //## (3)
17       Program prog = new Program();
18       prog.setName( "CALLME" );
19       prog.link( commArea );
20
21       //## (4)
22       out.println();
23       String rply = new String( commArea );
24       out.println("And␣the␣answer␣is:␣" + rply);
25
26       //## (5)
27     } catch (InvalidRequestException e) {
28       out.println("IRE");
29       e.printStackTrace();
30     } catch (LengthErrorException e) {
31       out.println("LEE");
32       e.printStackTrace();
33     } catch (InvalidSystemIdException e) {
34       out.println("ISE");
35       e.printStackTrace();
36     } catch (NotAuthorisedException e) {
37       out.println("NAE");
38       e.printStackTrace();
39     } catch (InvalidProgramIdException e) {
40       out.println("IPE");
41       e.printStackTrace();
42     } catch (RolledBackException e) {
43       out.println("RBE");
44       e.printStackTrace();
45     } catch (TerminalException e) {
46       out.println("TE");
47       e.printStackTrace();
48     }
49   }
50 }
```

Listing 2.2: Aufrufendes Programm mit XCTL

```
1  public class DoXctl {
2    public static void main( CommAreaHolder cah ) {
3      try {
4        Task task = Task.getTask();
5        PrintWriter out = task.out;
6        out.println();
7
8        //## (1)
9        out.println( "␣This␣is␣program:" + task.getProgramName() );
10
11       //## (2)
12       String myMsg = "the␣answer␣to␣life␣the␣universe␣and␣everything";
13       byte[] commArea = myMsg.getBytes();
14       out.println( "␣My␣question:" + myMsg );
15
16       //## (3)
17       Program prog = new Program();
18       prog.setName( "CALLME" );
19       prog.xctl( commArea );
20
21       //## (4)
22       out.println();
23       String rply = new String( commArea );
24       out.print("And␣the␣answer␣is:␣" + rply);
25
26     //## (5)
27     } catch (InvalidRequestException e) {
28       e.printStackTrace();
29     } catch (NotAuthorisedException e) {
30       e.printStackTrace();
31     } catch (InvalidProgramIdException e) {
32       e.printStackTrace();
33     }
34   }
35 }
```

Listing 2.3: Aufgerufens Programm

```
1  public class CallMe {
2    public static void main( CommAreaHolder cah ) {
3      PrintWriter out = Task.getTask().out;
4      out.println();
5      out.println();
6      out.println();
7      //## (1)
8      out.println("This␣is␣program␣CALLME");
9      //## (2)
10     String theAnswer = "42";
11     cah.value = theAnswer.getBytes();
12   }
13 }
```

Listing 2.4: Anfrage mit Channel 1

```
1  public class NeedEuro {
2    public static void main( CommAreaHolder cah ) {
3      try {
4        //## (1)
5        Channel ch = Task.getTask().createChannel("DollarToEuro");
6        //## (2)
7        Container dollarCon = ch.createContainer("DOLLAR");
8        //## (3)
9        String dollarStr = "100";
10       dollarCon.put(dollarStr);
11       //## (4)
12       Program moneyConverter = new Program();
13       moneyConverter.setName("MONCONV");
14       //## (5)
15       moneyConverter.link(ch);
16
17       PrintWriter out = Task.getTask().out;
18       out.println();
19       out.println("Give" + dollarStr + "␣Dollars");
20       //## (6)
21       String euroStr = new String( dollarCon.get() );
22       out.println("Get" + euroStr + "␣Euros");
23
24     } catch (InvalidRequestException e) {
25       e.printStackTrace();
26     } catch (ContainerErrorException e) {
27       e.printStackTrace();
28     } catch (ChannelErrorException e) {
29       e.printStackTrace();
30     } catch (CCSIDErrorException e) {
31       e.printStackTrace();
32     } catch (CodePageErrorException e) {
33       e.printStackTrace();
34     } catch (LengthErrorException e) {
35       e.printStackTrace();
36     } catch (InvalidSystemIdException e) {
37       e.printStackTrace();
38     } catch (NotAuthorisedException e) {
39       e.printStackTrace();
40     } catch (InvalidProgramIdException e) {
41       e.printStackTrace();
42     } catch (RolledBackException e) {
43       e.printStackTrace();
44     } catch (TerminalException e) {
45       e.printStackTrace();
46     }
47   }
48 }
```

Listing 2.5: Anfrage mit Channel 2

```
1  public class NeedDollar {
2    public static void main( CommAreaHolder cah ) {
3
4      try {
5        //## (1)
6        Channel ch = Task.getTask().createChannel("EuroToDollar");
7        //## (2)
8        Container euroCon = ch.createContainer("EURO");
9        //## (3)
10       String euroStr = "100";
11       euroCon.put(euroStr);
12       //## (4)
13       Program moneyConverter = new Program();
14       moneyConverter.setName("MONCONV");
15       //## (5)
16       moneyConverter.link(ch);
17
18       PrintWriter out = Task.getTask().out;
19       out.println();
20       out.println("Give" + euroStr + "␣Euros");
21       //## (6)
22       String dollarStr = new String( euroCon.get() );
23       out.println("Get" + dollarStr + "␣Dollars");
24
25     } catch (InvalidRequestException e) {
26       e.printStackTrace();
27     } catch (ContainerErrorException e) {
28       e.printStackTrace();
29     } catch (ChannelErrorException e) {
30       e.printStackTrace();
31     } catch (CCSIDErrorException e) {
32       e.printStackTrace();
33     } catch (CodePageErrorException e) {
34       e.printStackTrace();
35     } catch (LengthErrorException e) {
36       e.printStackTrace();
37     } catch (InvalidSystemIdException e) {
38       e.printStackTrace();
39     } catch (NotAuthorisedException e) {
40       e.printStackTrace();
41     } catch (InvalidProgramIdException e) {
42       e.printStackTrace();
43     } catch (RolledBackException e) {
44       e.printStackTrace();
45     } catch (TerminalException e) {
46       e.printStackTrace();
47     }
48   }
49 }
```

Listing 2.6: Java Channel Multiplexer

```
1  public class MoneyConverter {
2    public static void main ( CommAreaHolder cah ) {
3
4      //## (1)
5      Channel ch = Task.getTask().getCurrentChannel();
6
7      //## (2)
8      if( ch != null ) {
9        String chName = ch.getName().trim();
10
11
12       try {
13
14         //## (3)
15         double resValue;
16         if ( chName.equals("EuroToDollar") ) {
17           //## (3-a)
18           Container euroRec;
19           euroRec = ch.getContainer("EURO");
20           double reqVal = Double.valueOf( new String( euroRec.get() ) );
21           resValue = euroToDollar( reqVal );
22
23           euroRec.put( Double.toString( resValue ) );
24
25         } else if ( chName.equals("DollarToEuro") ) {
26           //## (3-b)
27           Container dollarRec =  ch.getContainer("DOLLAR");
28           double reqVal = Double.valueOf( new String( dollarRec.get() ) )
                 ;
29           resValue = dollarToEuro( reqVal );
30           dollarRec.put( Double.toString(resValue) );
31         }
32
33       //## (5)
34       } catch (ContainerErrorException e) {
35         e.printStackTrace();
36       } catch (NumberFormatException e) {
37         e.printStackTrace();
38       } catch (ChannelErrorException e) {
39         e.printStackTrace();
40       } catch (CCSIDErrorException e) {
41         e.printStackTrace();
42       } catch (CodePageErrorException e) {
43         e.printStackTrace();
44       } catch (InvalidRequestException e) {
45         e.printStackTrace();
46       }
47     }
48
49   }
50   //## (4-a)
```

```
51    private static double euroToDollar ( double euro ) {
52      return (euro * 1.30);
53    }
54    //## (4-b)
55    private static double dollarToEuro ( double euro ) {
56      return (euro / 1.30);
57    }
58
59  }
```

Listing 2.7: Beispiel für einen ContainerIterator

```
1  Task t = Task.getTask();
2  ContainerIterator ci = t.containerIterator();
3  while ( ci.hasNext() ) {
4    Container custData = ci.next();
5    // Verarbeitung Container Inhalt
6  }
```

3 Einführung in XML

XML ist die Abkürzung für “Extensible Markup Language”. XML beschreibt einen Sprachstandard, der im Jahr 1998 vom W3C beschlossen wurde und als Metasprache für die Auszeichnungen von Textdokumenten genutzt wird. Sie definiert die Struktur von Dokumenten und damit auch die Struktur textbasierter Datenformate.
Im Bereich der Anwendungsintegration wird XML unter anderem dafür genutzt, Daten zwischen verschiedenen Anwendungen auf heterogenen Systemen auszutauschen. In fast jeder modernen Programmiersprache existieren Bibliotheken zum Lesen und Schreiben von XML-Dokumenten. Dies ist auch der Grund, warum moderne Übertragungsprotokolle wie das “Simple Object Access Protocol” (SOAP) auf XML basieren. Auch die Beschreibungssprache für Webservices (WSDL) verwendet XML-Dokumente.
In diesem Kapitel lernen Sie die Grundlagen von XML kennen. Sie werden lernen, wie XML-Dokumente aufgebaut sind und wie ihre Struktur formal definiert werden kann. In späteren Kapiteln werden Sie darauf basierend lernen, wie Sie XML-Dokumente in Java unter CICS lesen und erzeugen. Nach dem Lesen dieses Kapitels werden Sie in der Lage sein, einfache XML-Dokumente selbstständig zu schreiben und einfache Grammatiken zu definieren. Zudem erhalten Sie ein grundlegendes Verständnis, welches Ihnen ermöglicht weiterführende Literatur wie z.B. [SN09] schnell und einfach zu verstehen.

3.1 Aufbau eines XML-Dokuments

Am einfachsten lässt sich der Aufbau eines XML-Dokuments anhand eines Beispiels erläutern. Listing 3.1 enthält ein XML-Dokument, welches einen einfachen Kalendereintrag repräsentiert. Dieses kleine Beispiel enthält bereits die typischen Bestandteile eines XML-Dokuments: Kopfzeile, Elemente, Attribute und Kommentare.

Hauptbestanteil jedes XML-Dokuments sind die Elemente. Ein Element wird durch eine offene spitze Klammer “<” initialisiert. Daruf folgt der Name des Elements und im einfachsten Fall direkt eine geschlossene spitze Klammer “>”. Diese Zeichenfolge beschreibt den Beginn eines Elements. Das entsprechende und notwendige Ende eines Elements wird ähnlich notiert. Hier folgt nach der offenen spitzen Klammer ein Slash “/”. Zu beachten gilt, dass der Elementname in beiden Fällen identisch sein muss.
Ein XML-Dokument heißt wohldefiniert, wenn es zu jedem Elementbeginn auch ein entsprechendes Ende gibt. In unserem Beispiel sind unter anderem `<date>` und `<time>` ein Element. Aber auch `<appointment>` ist ein Element, auch wenn dieses noch weitere Definitionen im Beginn enthält. Den Bereich zwischen dem Beginn und dem Ende bezeichnet man als Wert des Elements. Dieser kann entweder aus einfachen Zeichenketten bestehen, wie z.B. bei `<date>`, oder wiederum andere Elemente enthalten, wie es bei `<appointment>` der Fall ist. Innerhalb des Elementwertes dürfen keine öffnenden oder schließenden spitzen Klammern und kein Kaufmannsund verwendet werden. Für alle drei Zeichen existieren jedoch Ersatzdarstellungen (vgl. Tabelle 3.1).

Zeichen	Ersatzdarstellung
&	& amp;
’	& apos;
<	& lt;
>	& gt;
“	& quot;

Tabelle 3.1: Sonderzeichen und entsprechende Ersatzdarstellungen in XML

XML-Dokumente beginnen immer mit einem Wurzelelement. Durch die mögliche Verschachtelung können XML-Dokumente als Baumstruktur aufgebaut werden, daher auch der Name “Wurzelelement”.

Ein weiterer wichtiger Bestandteil von XML-Dokumenten sind Attribute. Diese können als Zusatzinformationen innerhalb eines Elementbeginns gespeichert werden. Ihre Syntax ist:

`AttributName=“AttributWert“`

Im gegebenen Beispiel ist unter anderem `user=“stefan“`Attribut des `appointment` -Elements. Der Wert eines Attributes muss eine Zeichenkette sein. Verschachtelungen, wie sie bei Elementen möglich waren, sind an dieser Stelle nicht erlaubt. Verbotene Sonderzeichen innerhalb des Wertes sind öffnende und schließende spitze Klammern “<” “>”, das Kaufmannsund “&” und die einfassenden Anführungszeichen. Die Länge der Attributwerte wird durch keine externe Regel beschränkt, jedoch gilt es als guter Stil diese möglichst kurz zu halten und längere Texte als Elementwert auszulagern.

Der letzte Bestandteil eines XML-Dokuments, der in diesem Abschnitt vorgestellt werden soll, sind Kommentare. Sie werden mit der Zeichenfolge “<!–” eingeleitet und mit “–>” beendet. Innerhalb eines Kommentars kann ein beliebiger Text stehen. Nur die Zeichenfolge “–>” ist aus offensichtlichen Gründen nicht gestattet. Ihre Position kann außerhalb von Elementen frei gewählt werden.

Listing 3.1: Kleines XML-Beispiel

```
1  <!-- A simple appointment as XML document -->
2  <appointment user="stefan" ctime="1234567890" state="confirmed" calendar=
       "university">
3      <date>2010-10-26T10:00:00</date>
4      <duration>90</duration>
5      <title>Talk JEE + Server Technology</title>
6      <location>Sand 2 B313</location>
7      <description>Talk about JEE and SOA on IBM Servers</description>
8      <attendees>
9          <attendee state="confirmed" user="PROF01">
10             <name>Prof. Dr. Ray</name>
11         </attendee>
12         <attendee state="invited" user="DR01">
13             <name>Dr. Taylor</name>
14         </attendee>
15     </attendees>
16 </appointment>
```

3.2 XML-Namen

Bei der Vergabe von eigenen Namen z.B. an Elemente oder Attribute gelten innerhalb von XML spezielle Regeln [SN09]:

Ein XML-Name darf beliebige, auch nichtenglische, alphanumerische Zeichen enthalten. Als Interpunktionszeichen sind Unterstriche "_", Bindestriche "-", Punkte "." und Doppelpunkte ":" erlaubt. Letzterer nimmt in diesem Zusammenhang eine besondere Rolle ein. Der Doppelpunkt ist für die Angabe von Namensräumen, wie sie z.B. aus C++ bekannt sind, reserviert.

Ein XML-Name darf nicht mit Leerzeichen getrennt werden. Auch die Zeichenfolge "xml" sollte in keinem Namen enthalten sein. Dabei spielt es keine Rolle, ob diese Zeichenfolge in Groß- oder Kleinbuchstaben vorliegt. Auch eine Kombination beider Schreibweisen ist verboten [SN09].

Das erste Zeichen eines XML-Namens darf keine Ziffer und kein Interpunktionszeichen sein.

3.3 XML-Grammatik

XML-Dokumente können beliebig aufgebaut werden. Es gibt keine Beschränkung über die Anzahl verschiedener Elementtypen oder über deren Verschachtelungstiefe. Für den Austausch von Daten ist es jedoch wichtig, dass sich Sender und Empfänger auf ein Format der Darstellung einigen. Für XML-Dokumente bedeutet dies, dass der Aufbau eines Dokuments festen Regeln folgen muss. Diese Regeln beschreiben die verschiedenen Typen der Elemente, erlaubte Attribute und die Reihenfolge, in denen Elemente in einem XML-Dokument vorkommen. Die Menge dieser Regeln wird auch Grammatik genannt. Diese kann vom Entwickler entsprechend einer Anwendungs-Domäne selbst definiert werden.

XML-Grammatiken können mit Hilfe der Document Type Definition (DTD) oder über XML-Schemata erstellt werden. In beiden Fällen wird die Grammatik als externe Textdatei bereitgestellt. In den folgenden Abschnitten 3.4 und 3.5 werden beide Techniken anhand eines kleinen Beispiels kurz vorgestellt.

Liegt für ein vorhandenes XML-Dokument eine Grammatik vor, besteht die Möglichkeit das Dokument zu validieren. Das bedeutet, dass die Einhaltung aller definierten Regeln automatisch überprüft wird. Typischerweise wird die Validierung im realen Betrieb vom Sender oder Empfänger durchgeführt. Java verfügt dafür bereits über eingebaute Methoden. Auch einige Entwicklungsumgebungen wie z.B. Netbeans bieten die Möglichkeit XML-Dokumente zu validieren.

Beschäftigen wir uns zum Abschluss noch kurz mit der Frage, wieso die Validierung von XML-Dokumenten überhaupt sinnvoll ist und wieso es nicht ausreicht, sich einfach auf ein Format zu einigen, ohne dessen Einhaltung zu überprüfen. Die Antwort ist relativ einfach. Die wenigsten Anwendungen sind in der Lage, auf Basis falscher oder ungültiger Eingabedaten korrekte Berechnungen und anschließende Ausgaben zu erzeugen. Im Bereich der Anwendungsintegration müssen viele heterogene Systeme miteinander kommunizieren. Dort führt die Möglichkeit der Validierung der übertragenen Datenformate zu zusätzlicher Sicherheit. Dies ist ein großer Vorteil gegenüber hardwarenahen binären Datenformaten, bei denen eine solche Validierung nur sehr schwer möglich ist.

3.4 Document Type Definition

Eine DTD besteht hauptsächlich aus Definitionen für Element- und Attributtypen. Das Listing 3.2 zeigt eine DTD für einen XML-Kalendereintrag. Ein solcher Eintrag enthält ein Datum, eine Uhrzeit, einen Titel, einen Ort, eine Beschreibung und optional eine Liste von Teilnehmern. Zusätzlich kann für jeden Eintrag der Name des Benutzers, der Zeitpunkt der Erzeugung und Informationen über den Kalendertyp und ein Terminstatus gespeichert werden.

Die Definition eines Elementtyps hat die Form:

$$<!\texttt{ELEMENT}\, Name\, (Inhalt)>$$

Der Name des Elementtyps kann nach den im Abschnitt 3.2 beschriebenen Regeln frei gewählt werden. Bei der Definition von `appointment` sehen Sie in Listing 3.2 als Inhalt in Klammern eine Liste von erwarteten Elementtypen. Die einzelnen Typen werden durch ein Komma getrennt hintereinander aufgeführt. Die in dieser Definition gewählte Reihenfolge muss auch im späteren XML-Dokument eingehalten werden.
Hinter einigen Typnamen finden Sie zusätzliche Operatoren wie z.B. “+” oder “?”. Diese beschreiben zusätzliche Regeln über die Häufigkeit des davor beschriebenen Elements innerhalb des definierten Knotens. Das Fragezeichen hinter `location` bedeutet, dass der Knoten `appointment` kein oder maximal ein `location`-Element enthalten darf. Eine Übersicht über weitere Operatoren finden Sie in Tabelle 3.2.

Operator	Ersatzdarstellung
+	Das entsprechende Element muss mindestens einmal vorkommen
?	Das entsprechende Element kann keinmal oder einmal vorkommen
*	Das entsprechende Element kann keinmal oder beliebig oft vorkommen

Tabelle 3.2: Operatoren innerhalb der Document Type Definition

Neben Typlisten können Elemente natürlich auch einfache Zeichenketten enthalten. Dafür gibt es die Typen `PCDATA` und `CDATA`. PCDATA steht für “parsed computer data”. Dieser kann als Inhalt weitere XML-Elemente oder einfachen Text enthalten. Daher dürfen Texte, die als PCDATA deklariert sind, keine spitzen Klammern enthalten. Inhalte im CDATA-Format unterliegen nicht dieser Einschränkung.
Die Typangaben bei der Inhaltsbeschreibung einer Knotendefinition können auch kombiniert werden. Um eine Alternative zwischen verschiedenen Typen zu beschreiben, können Sie den or-Operator ″ | “ verwenden. Zum Beispiel kann der Knoten `users` mit der Definition

```
<!ELEMENT users (name | userid)+>
```

beliebig viele Einträge vom Typ `name` oder `userid` enthalten.

Attribute können für jeden Elementtyp definiert werden. Sie haben folgende Syntax:

$$<!\texttt{ATTLIST}\ \textit{ElementName}\ \textit{AttName}\ \textit{AttTyp}\ \textit{AttEigenschaft}\ \ldots>$$

Als Beispiel finden Sie in Listing 3.2 eine Attributliste für den Knoten `appointment`. Es gibt keine Begrenzung für die maximale Anzahl an Attributen pro Element. Der *ElementName* muss durch

den Namen des Elements ersetzt werden, für welches die Liste definiert werden soll. Jedes Attribut wird in der Folge durch drei Werte beschrieben. Für den Attributnamen *AttName* gelten die Regeln aus Abschnitt 3.2, zudem sollte der Name für den entsprechenden Knoten eindeutig gewählt werden. Als Attributtype *AttTyp* steht unter anderem CDATA zur Verfügung. Elementtypen sind an dieser Stelle aus offensichtlichen Gründen nicht erlaubt. Andere mögliche Werte sind zum Beispiel ID oder Aufzählungstypen, wie sie bei dem Attribut `state` verwendet wurden. Eine vollständige Liste aller möglichen Typen finden Sie in der Spezifikation [xml00]. Als dritter und letzter Wert enthält die Attributdefinition noch eine Angabe, die bestimmt, ob das Attribut verpflichtend (`REQUIRED`), optional (`IMPLIED`) oder fest (`FIXED`) ist. Im letzten Fall wird der Wert des Attributes bereits durch die Grammatik festgelegt.

Listing 3.2: Beispiel für eine DTD eines Kalendereintrags

```
1  <!ELEMENT appointment (date,time,title,location?,description?,attendees?)
       >
2
3  <!ELEMENT date (#PCDATA)>
4  <!ELEMENT time (#PCDATA)>
5  <!ELEMENT title (#PCDATA)>
6  <!ELEMENT location (#PCDATA)>
7  <!ELEMENT description (#PCDATA)>
8  <!ELEMENT attendees (attendee+)>
9
10 <!ATTLIST appointment
11     calender  CDATA   #IMPLIED
12     user      CDATA   #IMPLIED
13     ctime     CDATA   #REQUIRED
14     state     (invited|confirmed|chanceled)   #IMPLIED
15 >
16
17 <!ELEMENT attendee (name)>
18 <!ATTLIST attendee
19     state  CDATA #IMPLIED
20 >
```

3.5 XML-Schemata

XML-Schemata geben eine weitere Möglichkeit Grammatiken für XML-Dokumente zu definieren. Sie werden, anders als DTD-Dokumente, direkt in XML verfasst und bieten umfangreichere Möglichkeiten zur Regelgestaltung. Dieser Abschnitt behandelt die grundlegenden Sprachelemente von XML-Schemata und betont ihre Unterschiede zu DTD. Eine vollständige Dokumentation finden Sie auf der Seite des W3C unter [XSD00].
Listing 3.3 zeigt Ihnen ein Beispiel für ein XML-Schema. Es beschreibt das gleiche Format für Kalendereinträge, das Sie bereits aus Listing 3.2 kennen. Wie auch bei DTD-Dokumenten sind Definitionen von Elementen und Attributen der Hauptbestandteil eines XML-Schemas.

Beginnen wir mit der Definition von Elementen. Diese werden via

`<xsd:element name="`*ElementName*`"</xsd:element>`

definiert. Den Namen des beschriebenen Knotens können Sie wieder nach den Regeln aus Abschnitt 3.2 frei wählen. Alle weiteren Beschreibungen des Elements werden zwischen Beginn und Ende der Knotendefinition verfasst.

XML-Schemata erlauben es dem Entwickler für jedes Element genaue Angaben über dessen Datentyp zu verfassen. Dafür enthält die Sprache zur Erstellung von XML-Schemata bereits eine Grundmenge an Basistypen. Diese reichen von verschiedenen numerischen Typen über Formate für Datum- und Zeitangaben bis hin zu Typen für Zeichenketten. Eine vollständige Liste finden Sie in der Spezifikation unter [XSD00]. Jeder dieser Basistypen kann zusätzlich mit Hilfe weiterer Angaben durch den Entwickler eingeschränkt werden. So ist es zum Beispiel möglich, das Format für die Datumseingabe festzulegen oder über die Verwendung regulärer Ausdrücke nur bestimme Muster von Zeichenketten zuzulassen.
Generell wird innerhalb eines XML-Schemas zwischen zwei Grundformen von Datentypen unterschieden, zwischen den simplen und den komplexen Typen. Die einfachen Typen werden von den Basistypen abgeleitet und ggf. durch zusätzliche Einschränkungen im Wertebereich modifiziert. Komplexe Datentypen zeichnen sich dadurch aus, dass sie wieder aus anderen Elementen zusammengesetzt werden.
Das Element `date` in Zeile 6 innerhalb des Listings 3.3 verwendet zum Beispiel einen simplen Datentyp `date`. In Zeile 33 sehen Sie, wie ein Basistyp als Ausgangspunkt für den neuen Datentyp `appointmentState` verwendet wird. Diesem soll es nur gestattet sein einen von drei vorgegebenen Werten anzunehmen. Dafür wird mit

`<xsd:simpleType name=“`*TypName*`“>`

ein neuer Typ angelegt, dessen Name als Attribut angegeben wird. Darauf folgt in Zeile 34 eine Restriktion, die als Ausgangstyp einen String verwendet. Die Attribute `value` innerhalb der `enumeration`-Elemente zählen dann alle erlaubten Werte für den neuen Typ `appointmentState` auf.

Die Zuweisung eines Typs zu einem Element kann auf zwei Arten erfolgen. Eine Variante ist es, den Typ wie im Falle von `appointmentState` außerhalb der Elementbeschreibungen zu definieren und im Anschluss den Typ als Attributwert im Beginn eines Elements zu setzen. Dieses Vorgehen sehen Sie zum Beispiel in Zeile 8 oder 19. Hier werden bereits definierte Typen mit dem Attribut `type=“ xsd:string“` oder `type=“ appointmentState“` den Elementen zugewiesen.
Ein anderes Vorgehen zeigt die Zeile 4. Hier enthält das Element appointment keine Typangabe. Stattdessen wird der Typ innerhalb des Elements definiert. In diesem Fall wird ein neuer, namenloser komplexer Typ beschrieben. An dieser Stelle muss der neue Datentyp keinen Namen tragen, da die Zuordnung auf Grund der Verschachtelung eindeutig ist.

Komplexe Typen werden aus anderen Elementen zusammengesetzt. XML-Schemata erlauben es, Angaben über die Reihenfolge der enthaltenen Elemente zu tätigen. In Zeile 5 wird dafür eine `sequenz` definiert. Dies bedeutet, dass die enthaltenen Elemente im späteren XML-Dokument in der gleichen Reihenfolge vorkommen müssen, wie sie an dieser Stelle definiert sind. Andere Möglichkeiten bietet die Verwendung von `all`, die keine Reihenfolge vorgibt, oder `choice`. Letztere Angabe definiert eine Alternative zwischen verschiedenen Elementen. Das spätere XML-Dokument muss dann nur ein Element der beschriebenen Auswahl enthalten. Des Weiteren können verschiedene Elemente auch zu Gruppen zusammengefasst werden. Für genauere Informationen sei an dieser Stelle auf die Seite der

W3School [W3S] verwiesen, welche die Verwendung der verschiedenen Indikatoren an vielen Beispielen ausführlich beschreibt.
Neben dem Typ kann für jedes Element auch eine explizite Angabe über das minimale und maximale Vorkommen eines Elements getroffen werden. Dazu dienen die Attribute `minOccurs` und `maxOccurs`. Ein Wert von 0 für `minOccurs` und 10 für `maxOccurs` bedeutet also, dass das entsprechende Element nicht vorkommen muss, jedoch maximal 10-mal vorkommen darf. Die Angaben über die Häufigkeit von Elementen können frei getroffen werden. Dies unterscheidet XML-Schemata von DTD-Beschreibungen, bei denen nur ausgewählte Kardinalitäten definiert werden können.

Als letzter Bestandteil von XML-Schemata soll innerhalb dieses Abschnitts die Definition von Attributen erläutert werden. Diese können innerhalb komplexer Typen beschrieben werden, dürfen selbst jedoch nur simple Typen enthalten. Ein Beispiel für die Definition eines Attributs sehen Sie in Zeile 26. Die allgemeine Syntax ist:

`<xsd:attribute name=“`*AttName*`“ type=“`*AttType*`“ use=“`*AttUsage*`“/>`

Die Werte, die Sie für *AttName* und *AttType* einsetzen müssen, sind analog zu denen bei der Definition von Elementen. Für *AttUsage* können Sie entweder `optional` oder `require` eintragen. Der erste Wert beschreibt ein optionales Attribut, die Standardeinstellung bei Attributen innerhalb XML-Schemata. Der zweite Wert definiert ein zwingend notwendiges Attribut. Zusätzlich können über die Angaben `default` und `fixed` Standard- und Festwerte für Attribute definiert werden.

Listing 3.3: Beispiel für ein XML-Schema eines Kalendereintrags

```
<xsd:schema>

  <xsd:element name="appointment">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="date" type="xsd:date" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="time" type="xsd:time" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="title" type="xsd:string" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="location" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="description" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="attendees" minOccurs="0" maxOccurs="1">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="attendee" minOccurs="1">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                  </xsd:sequence>
                  <xsd:attribute name="state" type="tns:appointmentState" use="optional"/>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="user" type="xsd:string" use="optional"/>
      <xsd:attribute name="time" type="xsd:string" use="required"/>
      <xsd:attribute name="state" type="tns:appointmentState" use="optional"/>
    </xsd:complexType>
  </xsd:element>


  <xsd:simpleType name="appointmentState">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="invited"/>
      <xsd:enumeration value="confirmed"/>
      <xsd:enumeration value="canceled"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

3.6 XML-Namensräume

Zum Abschluss dieser kleinen Einführung in XML betrachten wir noch das Konzept des Namensraums oder im Englischen Namespace. Das Konzept funktioniert in XML analog zu denen bekannter Programmiersprachen wie etwa C++.
Betrachten wir zum besseren Verständnis ein kleines Beispiel. Stellen Sie sich ein Element namens "user" vor. Dieses könnte unter anderem den vollen Namen eines Nutzers enthalten oder eine eindeutige ID, oder beides. Welche Informationen in einem Element "user" gespeichert werden, hängt von der entsprechenden Anwendungsdomäne ab. Stellen Sie sich vor, Sie müssten ein neues XML-Schema entwickeln und würden dafür bereits bestehende Schemata inkludieren. Eines dieser Schemata verwendet dabei die erste, ein weiteres die zweite Interpretation des "user"-Elements. Für den XML-Parser wären beide Elemente ab diesem Zeitpunkt nicht mehr unterscheidbar. Aus diesem Grund existieren Namensräume. Diese erlauben es, jede XML-Grammatik und damit auch die in ihr definierten Elemente und Attribute an einen Namensraum zu binden. Kombiniert man den Namen des Namensraums mit dem Namen des Elements oder Attributs, sind auch doppelt vergebene Namen, wie `user` aus unserem Beispiel, eindeutig ihrer Definition zuzuordnen.
Die Bindung eines Elements an einen Namensraum erfolgt über die Angabe einer Uniform Resource Identification (URI). Eine URI ist eine allgemein gültige ID, wie z.B. eine Internetadresse. Diese kann einfach in den Start-Tag des Elements eingetragen werden:

```
<html xmlns="http://www.w3.org/1999/xhtml">.
```

Der Attributname `xmlns` steht für "XML-Namespace". Eine andere Form der Bindung ermöglicht die Verwendung der Präfixnotation. Das verwendete Präfix kann zum Beispiel in der Kopfzeile eines XML-Dokuments angegeben werden. Die Kopfzeile für eine XML-Schemas Datei sieht zum Beispiel wie folgt aus:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Durch die Erweiterung des xmlns-Attributes um ":xsd" wird der mit `xmlns` angesprochene Namensraum an das Präfix "xsd" gebunden. Im weiteren Verlauf des XML-Dokuments können Sie nun alle Elemente im entsprechenden Namensraum mit "xsd:ElementName" ansprechen. Das Präfix wird also mit einem Doppelpunkt getrennt und einfach vor dem Elementnamen positioniert.

Zusammenfassung

XML ist eine Auszeichnungssprache für textbasierte Dateiformate. Ihre Hauptbestandteile sind Elemente und Attribute. In einem kleinen Beispiel wurde Ihnen gezeigt, wie XML-Dokumente aufgebaut sind und welche Syntax ihnen zugrunde liegt. Sie haben gelernt, dass Elemente in XML-Dokumente verschachtelt werden können und so Baumstrukturen bilden.
Die Regeln für den Aufbau eines gültigen XML-Dokuments werden auch Grammatik genannt. Diese kann mit Hilfe der Document Type Definition oder über XML-Schemata beschrieben werden. Beide Verfahren wurden mit Hilfe eines Beispiels erläutert und ihre grundlegende Syntax sowie deren wichtigste Bestandteile erklärt.
Sie sind nun im Stande, eigenständig XML-Dokumente zu erstellen und einfache Grammatiken mit Hilfe der DTD oder XML-Schemata zu entwerfen.

Aufgaben

1. *Listen Sie die Namen und Werte aller Elemente und Attribute aus Listing 3.1 einzeln auf.*
2. *Erstellen Sie ein weiteres einfaches XML-Beispiel z.B. für den Eintrag eines Buches in einer Bibliothek.*
3. *Schreiben Sie für Ihr in Aufgabe 2 gewähltes Beispiel eine DTD.*
4. *Erläutern Sie kurz die Unterschiede zwischen DTD und XML.*
5. *Erstellen Sie auf Basis des in Listing 3.3 vorgestellten XML-Schemas ein weiteres Beispiel für einen XML-Kalendereintrag. Verwenden Sie dieses Mal alle möglichen Elemente und Attribute.*
6. *Verfassen Sie ein XML-Schema für die in Aufgabe 3 erstellte DTD.*

4 JAXP

JAXP steht für “Java API for XML Processing” und ist Teil des Java-Standards. JAXP bietet somit eine einfache und leichtgewichtige Schnittstelle für die gesamte XML-Verarbeitung mit Java. Neben JAXP sind für Java auch noch externe Bibliotheken mit ähnlichem Funktionsumfang und Schnittstellen verfügbar. Ein Beispiel für eine solche Bibliothek ist das Xerces Projekt der Apache Group [JAX10]. JAXP ist nur eine Schnittstellenbeschreibung und gibt keine Implementierung vor. Diese kann vom Anwender ausgetauscht werden. Die gesamte JAXP-Beschreibung umfasst vier Schnittstellen [SN09]:

- die “simple API for XML” (SAX) zum seriellen Parsen von XML-Dokumenten
- das “Document Object Model” (DOM) zum Parsen und Erstellen von XML-Dokumenten
- eine für die „Extensible Stylesheet Language for Transformations” (XSLT) für XML-Transformationen
- die “Streaming API for XML” (Stax).

In diesem Kapitel wird Ihnen die Verwendung der ersten drei Schnittstellen erörtert. Sie werden lernen, wie man XML-Dokumente mit Hilfe der verschiedenen Methoden parsen und erzeugen kann.

Voraussetzungen

In den folgenden Abschnitten werden Ihnen die verschiedenen XML-Schnittstellen jeweils anhand eines kleinen Beispiels erklärt. Grundlage dafür ist das XML-Dokument aus Listing 3.1. Ziel der verschiedenen Beispiele wird es sein, dieses XML-Dokument in eine entsprechende Objektstruktur umzuwandeln. Bestandteil dieser Struktur sind zwei Klassen: `Appointment` und `Attendee`. Den Quelltext beider Klassen finden Sie im Anhang. Damit die Beispiele der folgenden Abschnitte funktionieren, müssen Sie erst diese beiden Klassen auf dem Mainframe installieren. Laden Sie dafür beide Klassen in das Paket `prak500.cicscalender.model`.
Des Weiteren verwenden die folgenden Beispiele zwei Hilfsklassen: `AppointmentPrinter` und `AppointmentFieldConverter`. Die erste Klasse hat die Aufgabe, die Objektstruktur eines Kalendereintrags auf der Konsole auszugeben. Die zweite Klasse enthält Methoden, um einzelne Informationen des Kalendereintrags in die entsprechende Dateiformate umzuwandeln. Auch für diese Klassen finden Sie die Quelltexte im Anhang. Dieser enthält auch die notwendigen Paketinformationen.
Als letzte Vorbereitung ist es notwendig, das XML-Dokument auf den Mainframe zu laden. Verwenden Sie als Zielordner einen beliebigen Unterordner Ihres Home-Verzeichnisses. Bitte beachten Sie, dass Sie auch Pfadangaben aus den Beispielquelltexten an Ihren Zielordner anpassen müssen.

4.1 SAX

Die “Simple API for XML” oder kurz SAX ist anders als das “Document Object Model” (vgl. Abschnitt 4.2) kein offizieller Standard des W3C. Dennoch wird es von fast allen XML-Bibliotheken, auch außerhalb von Java, unterstützt. Die Unterschiede beider Verfahren werden im Abschnitt 4.2 behandelt.

4.1.1 Parsen mit SAX

An dieser Stelle wird Ihnen gezeigt, wie Sie SAX dafür verwenden können, XML-Dokumente zu parsen. Dazu bedienen wir uns eines kleinen Beispiels. Das Java-Programm aus Listing 4.1 wird den XML-Kalendereintrag aus Listing 3.1 parsen.
Ein SAX-Parser liest das ihm übergebene XML-Dokument seriell vom Anfang bis zum Ende. Während des Lesevorgangs erzeugt der SAX-Parser unterschiedliche Events, abhängig davon, welcher Teil des XML-Dokuments gerade gelesen wird. Ein Beispiel für einen solchen Event ist der Beginn eines neuen Elements. Diese Events werden von Handlern verarbeitet. Diese Handler sind Java-Klassen, die von `DefaultHandler` aus dem Paket `org.xml.sax.helpers` ableiten. Der `DefaultHandler` implementiert für jeden Event eine Methode. Der Handler, der für die Verarbeitung des XML-Dokuments verantwortlich ist, wird zu Beginn beim SAX-Parser registriert. Dieser ruft während des Parsens die entsprechenden Methoden des registrierten Handlers auf. Der vom Entwickler geschriebene Handler muss dabei nur die Methoden implementieren bzw. überschreiben, die für die angestrebte Verarbeitung notwendig sind.
Für unser Beispiel verwenden wir zwei verschiedene Handler: Einen zum Parsen der Appointment- und einen für die Attendee-Informationen. Dieses Vorgehen ist nicht zwingend notwendig. Die Entscheidung für dieses Design hat folgenden Hintergrund: Aus objektorientierter Sicht handelt es sich um zwei getrennte Objekte. Daraus resultieren auch zwei unabhängige Verantwortungsbereiche beim Parsen. Dies erlaubt es später zum Beispiel die Attendee-Klasse um weitere Informationen zu erweitern, ohne den Handler für die Appointment-Klasse zu verändern. Die Aufgabenverteilung ist damit streng geregelt. Diese Form des Softwaredesigns bezeichnet man als “responsibility drivin design” [**?**].

Die Hauptklasse dieses Beispiels sehen Sie in Listing 4.1. JAXP definiert nur eine Schnittstelle, gibt jedoch keine Implementierung vor. Aus diesem Grund muss diese erst über das Factory-Pattern angefordert werden (vgl. Factory-Pattern [GHJV94]). Dafür wird in Schritt (1) zuerst eine Instanz der `SAXParserFactory` erstellt. Diese kann im Anschluss dafür verwendet werden, eine Instanz des eigentlichen Parsers zu erzeugen.
Wie bereits erörtert wurde, verwendet dieses Beispiel zwei Handler. In Schritt (2) wird der Handler für die `Appointment`-Klasse erzeugt. Die Informationen über die Teilnehmerliste sind in die des Kalendereintrags eingebettet. Aus diesem Grund ist es auch Aufgabe des `Appointment`-Handlers die entsprechenden Events an den zweiten Handler weiterzuleiten. Die Kommunikation zwischen beiden Handlern wird in späteren Absätzen behandelt.
Schritt (3) ist schnell erklärt. Hier wird das zu parsende XML-Dokument in einer Instanz der Java-`File`-Klasse gekapselt.
Mit dem Aufruf der `parse()` Methode in Schritt (4) wird der SAX-Parser gestartet. Als Argumente erhält die Methode die in Schritt (3) erzeugte `File`-Instanz und eine Referenz auf den zuvor erzeugten Handler. Die eigentliche Verarbeitung des XML-Dokuments erfolgt in den Handler-Klassen und wird ab dem übernächsten Absatz behandelt.
Als Letztes wird in Schritt (5) über eine `get`-Methode das fertige `Appointment`-Objekt vom Hand-

```
XSAX◊
APPOINTMENT ########################################◊
university◊
◊
26.10.2010◊
10:00-11:30◊
Sand 2 B313◊
◊
Talk JEE + Server Technology◊
◊
state: CONFIRMED◊
----------------------------------------◊
Talk about JEE and SOA on IBM Servers◊
----------------------------------------◊
attendees:◊
◊
    -Prof. Dr. Ray (CONFIRMED)◊
    -Dr. Taylor (INVITED)◊
◊
################################################◊
```

Abbildung 4.1: Ausgabe des SAX-Parsers

ler angefordert. Dieses wird daraufhin mit Hilfe der `AppointmentPrinter`-Klasse auf der Konsole ausgegeben. Das Ergebnis dieser Ausgabe finden Sie in Abbildung 4.1.

Die Aufgabe der Hauptklasse war es, eine Instanz des Handlers und des Parsers zu erzeugen. Der Handler musste beim Parser registriert werden, bevor der Parser gestartet werden konnte. In den folgenden Absätzen widmen wir uns nun der Funktionsweise des Handlers. Den Quellcode für den `Appointment`-Handler finden Sie in Listing 4.2.
Der SAX-Parser durchläuft das zu parsende XML-Dokument seriell. Dadurch ist kein wahlfreier Zugriff auf die einzelnen Informationen des Kalendereintrags möglich. Alle gelesenen Werte müssen daher zuerst zwischengespeichert werden, bevor sie in einer Objektinstanz zusammengefasst werden können. Im Abschnitt (1) des Quellcodes sehen Sie daher eine Reihe an Variablen, die als Zwischenspeicher dienen. Ihre detaillierten Aufgaben werden in den folgenden Absätzen genauer beschrieben. Die Methode `startElement()` bei (2) ist eine vererbte Methode des `DefaultHandlers`. Sie wird vom SAX-Parser aufgerufen, sobald ein neues XML-Element beginnt. Die Methode erhält vier Argumente. Die `uir` ist der verwendete Präfix des Namensraums. Die Zeichenkette `localName` enthält den Namen des geöffneten Elements ohne Präfix. Beide Werte werden nur dann belegt, wenn

die Verarbeitung von Namensräumen aktiviert ist . Eine Alternative dazu ist die Verwendung des qualifizierten Namens. Dieser wird in `qName` gespeichert und enthält den gesamten Namen des Elements inklusive eventueller Präfixe. Das letzte Argument ist die Liste der Attribute. Dieses Beispiel arbeitet mit qualifizierten Namen. Aus diesem Grund ist es notwendig, das Präfix manuell vom Namen zu trennen. Dies geschieht in Schritt (2-b). Liegt der Name des Elements vor, kann im Anschluss entschieden werden, wie es verarbeitet werden soll. Wurde ein `appointment`-Element geöffnet, müssen die möglichen Attribute gesichert werden. Dieser Schritt erfolgt unter (2-c). Mit der Methode `getValue()` und dem Namen des Attributes als Argument kann dessen Wert aus der Attributliste ausgelesen werden. Sollte das Attribut nicht in der Liste enthalten sein, wird eine leere Zeichenkette zurückgegeben. Alle Attributwerte werden in Klassenvariablen im Block für einfache Werte (1-c) temporär gespeichert. Wird hingegen ein `attendee`-Element begonnen, ist das Vorgehen ein anderes. In diesem Fall ist es die Aufgabe dieses Handlers, das Ereignis an den verantwortlichen `Attendee`-Handler weiterzuleiten. Dies erfolgt über das Flag `withinAttendee`, welches als Klassenvariable (1-b) realisiert ist. Dieses Flag wird beim Beginn eines `attendee`-Elements in Schritt (2-d) auf `true` gesetzt. Im weiteren Verlauf des Programms wird das Ereignis in Schritt (2-e) immer dann weitergeleitet, wenn das Flag auf `true` gesetzt wurde.
Das Gegenstück zu `startElement()` ist die Methode `endElement()` bei Abschnitt (3) des Quelltextes. Wie der Name bereits vermuten lässt, wird sie immer dann vom SAX-Parser aufgerufen, wenn ein XML-Element geschlossen wird. Die Argumente sind identisch zu denen der `startElement`-Methode, nur dass an `endElement()` keine Attributliste übergeben wird. Zuerst wird wieder das Präfix des Elementnamens gelöscht (3-a) und das Ereignis ggf. an den `Attendee`-Handler weitergeleitet (3-b). Im Abschnitt (3-c) werden die restlichen Standardinformationen des Kalendereintrags zwischengespeichert. Dies erfolgt immer in Abhängigkeit vom Namen des beendeten Elements. Ein besonderes Vorgehen ist notwendig, wenn ein `attendee`-Element geschlossen wird. In diesem Fall kann bereits eine Instanz der `Attendee`-Klasse erzeugt werden. Des Weiteren muss auch die Weiterleitung der Ereignisse an den `Attendee`-Handler gestoppt werden. Beides erfolgt in Schritt (3-d). Zuerst wird das Flag wieder zurück auf `false` gesetzt. Im Anschluss erfragen wir die geparste Instanz der `Attendee`-Klasse vom `Attendee`-Handler. Diese wird daraufhin in einer globalen Liste zwischengespeichert.
Die letzte Kernmethode für die Verarbeitung eines XML-Dokuments sehen Sie unter (4). Die Methode `characters()` wird vom SAX-Parser aufgerufen, wenn ein Text zwischen Beginn und Ende eines Elements lesbar ist. Das bedeutet, mit Hilfe dieser Methode ist es möglich die Werte eines XML-Elements auszulesen. Dafür erhält sie drei Argumente. Die Array `ch` enthält in der Regel das gesamte XML-Dokument. Im Detail hängt dies von der Implementierung von der SAX Schnittstelle ab. Auf jeden Fall entspricht ihre Größe der Anzahl an Zeichen des XML-Dokuments. Der übergebene Index `start` bezieht sich auf die Array `ch` und dokumentiert die Position des ersten zu lesenden Zeichens. Der dritte Wert `length` enthält die Anzahl der zu lesenden Zeichen. Solle ein `attendee`-Element offen sein, wird das Ereignis bei (4-a) wieder weitergeleitet. Ansonsten werden die zu lesenden Zeichen bei Schritt (4-b) in einem globalen Puffer zwischengespeichert. Wird das entsprechende Element geschlossen, kann dann aus dem Puffer dessen Wert ausgelesen werden. Dieses Vorgehen konnten Sie bereits in Schritt (3-c) beobachten. Beginnt ein neues Element, kann davon ausgegangen werden, dass die Informationen im Puffer bereits verarbeitet sind. Auf jeden Fall werden sie nicht weiter gebraucht. Aus diesem Grund kann der Puffer bei jedem Beginn eines neuen Elements geleert werden, wie es in Schritt (1-a) erfolgt.
Die vorletzte Methode bei Punkt (5) dient als Schnittstelle zur Hauptklasse. Sie wird von dieser dazu genutzt die geparste Instanz der `Appointment`-Klasse anzufordern. Dieses Vorgehen ist analog zu dem Verhalten zwischen diesem und dem `Attendee`-Handler in Abschnitt (3-d). Die Methode

`getAppointment()` überprüft zuerst, ob bereits eine Instanz vorliegt. Wenn dies nicht der Fall ist, wird eine erstellt. Grundlage dafür sind die Werte, die wir in den vorhergehenden Schritten zwischengespeichert haben. Die so erzeugte Instanz wird innerhalb des Handlers ebenfalls zwischengespeichert. Auf diesem Wege kann sie mehrfach von außerhalb angefordert werden, ohne immer wieder neu erzeugt zu werden. Als Konsequenz dieser Designentscheidung ist eine `reset`-Methode notwendig, wie sie unter Punkt (6) zu sehen ist. Sie kann von außen aufgerufen werden, um alle lokal gespeicherten Werte zu verwerfen.

Der Aufbau des `Attendee`-Handlers ist komplett analog zu dem des `Appointment`-Handlers. Aus diesem Grund wird an dieser Stelle darauf verzichtet, ihn im Detail zu erläutern. Den zugehörigen kommentierten Quelltext finden Sie im Anhang. Damit dieses Beispiel funktioniert, muss der Handler im Paket `prak500.cicscalender.parser.sax` enthalten sein.

Listing 4.1: Der Quellcode der Hauptklasse des SAX-Parsers

```
1  public class SaxTutMain {
2    public static void main ( CommAreaHolder cah ) {
3
4      PrintWriter out = Task.getTask().out;
5      out.println();
6
7      try {
8        //## 1
9        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
10       SAXParser parser = parserFactory.newSAXParser();
11
12       //## 2
13       AppointmentHandler32 appHandler = new AppointmentHandler32();
14
15       //## 3
16       File appXML = new File("/u/prak500/classes/prak500/tutorials/sax/
             DateF1-001.xml");
17
18       //## 4
19       parser.parse( appXML, appHandler );
20
21       //## 5
22       Appointment23 appointment = appHandler.getAppointment();
23       AppointmentPrinter31.printAppointment(out, appointment);
24
25     } catch (ParserConfigurationException ex) {
26       out.println("Parser␣Conf␣Error");
27     } catch (SAXException ex) {
28       out.println("SAX␣Error");
29     } catch (Exception ex) {
30       ex.printStackTrace(out);
31       out.println("IO␣Error");
32     }
33   }
34 }
```

Listing 4.2: Quelltext der AppointmentHandler-Klasse

```
1  public class AppointmentHandler32 extends DefaultHandler {
2
3    //# (1)
4    //# (1-a)
5    private StringBuffer buffer = new StringBuffer();
6    //# (1-b)
7    private boolean withinAttendee = false;
8    private AttendeeHandler attendeeHandler = new AttendeeHandler();
9    //# (1-c)
10   private String  date = "";
11   private String  duration = "";
12   private String  location = "";
13   private String  title = "";
14   private String  description = "";
```

```
15    private String  calender = "";
16    private String  state = "";
17    private String  user = "";
18    private String  ctime = "0";
19    private LinkedList<Attendee> attendees = new LinkedList<Attendee>();
20    //# (1-d)
21    private Appointment appointment = null;
22
23    //# (2)
24    @Override
25    public void startElement(   String      uri,
26                  String      localName,
27                  String      qName,
28                  Attributes  attributes ) throws SAXException {
29
30      //# (2-a)
31      this.buffer = new StringBuffer();
32
33      //# (2-b)
34      if( qName.contains(":") ) {
35        qName = qName.substring( qName.rindexOf(":") + 1 );
36      }
37
38      //# (2-c)
39      if( qName.equals("appointment")) {
40        this.user = attributes.getValue("user");
41        this.ctime = attributes.getValue("ctime");
42        this.calender = attributes.getValue("calendar");
43        this.state = attributes.getValue("state");
44      }
45
46      //# (2-d)
47      else if(qName.equals("attendee")) {
48        this.withinAttendee = true;
49      }
50
51      //# (2-e)
52      if( this.withinAttendee ) {
53        this.attendeeHandler.startElement(uri, localName, qName, attributes
              );
54      }
55    }
56
57    //## (3)
58    @Override
59    public void endElement( String  uri,
60                String  localName,
61                String  qName ) {
62
63      //## (3-a)
64      if( qName.contains(":") ) {
65        qName = qName.substring( qName.rindexOf(":") + 1 );
66      }
```

```
67
68      //## (3-b)
69      if( this.withinAttendee ) {
70          this.attendeeHandler.endElement(uri, localName, qName);
71      }
72      //## (3-c)
73      else if ( qName.equals("date") ) {
74          this.date = buffer.toString();
75      }
76      else if(qName.equals("duration")) {
77        this.duration = buffer.toString();
78      }
79      else if(qName.equals("location")) {
80        this.location = buffer.toString();
81      }
82      else if(qName.equals("title")) {
83        this.title = buffer.toString();
84      }
85      else if(qName.equals("description")) {
86        this.description = buffer.toString();
87      }
88      else if(qName.equals("calender")) {
89        this.calender = buffer.toString();
90      }
91      else if(qName.equals("state")) {
92        this.state = buffer.toString();
93      }
94      else if(qName.equals("user")) {
95        this.user = buffer.toString();
96      }
97      else if(qName.equals("ctime")) {
98        this.ctime = buffer.toString();
99      }
100
101     //## (3-d)
102     if(qName.equals("attendee")) {
103       this.withinAttendee = false;
104       Attendee attendee = this.attendeeHandler.getAttendee();
105       if(attendee != null ) {
106         this.attendees.addLast(attendee);
107         this.attendeeHandler.reset();
108       }
109     }
110   }
111
112   //## (4)
113   @Override
114   public void characters( char[]  ch,
115               int     start,
116             int     length) {
117
118     //## (4-a)
119     if( this.withinAttendee ) {
```

```
120       this.attendeeHandler.characters(ch, start, length);
121       return;
122     }
123
124     //## (4-b)
125     for( int i=start; i < start+length; i++) {
126       this.buffer.append( ch[i] );
127     }
128   }
129
130   //## (5)
131   public Appointment getAppointment() {
132     if( this.appointment == null ) {
133       Calendar ldateAndTime = AppointmentFieldConverter.parseDateAndTime(
              this.date);
134       long lctime = new Long(this.ctime);
135       int lduration = AppointmentFieldConverter.parseDuration(this.
              duration);
136       State lstate = AppointmentFieldConverter.parseStateStr(this.state);
137       this.appointment = new Appointment  ( ldateAndTime, lduration, this
              .title,this.description,this.location,this.calender,lstate,this.
              user,lctime);
138
139       if( this.attendees.size() > 0) {
140         for( Attendee22 attendee : this.attendees ) {
141           this.appointment.addAttendee(attendee);
142         }
143       }
144     }
145
146     return this.appointment;
147   }
148
149   //## (6)
150   public void reset() {
151       this.appointment = null;
152       this.buffer = new StringBuffer();
153       this.attendees = new LinkedList<Attendee>();
154       this.date = "";
155       this.duration = "";
156       this.location = "";
157       this.title = "";
158       this.description = "";
159       this.calender = "";
160       this.state = "";
161       this.user = "";
162       this.ctime = "";
163   }
164 }
```

4.2 DOM

Das “Data Object Model” oder kurz DOM ist ein offizieller Standard des W3C zur Verarbeitung von XML-Dokumenten. Die DOM API kann dafür verwendet werden XML-Dokumente zu parsen, aber auch dafür sie zu erzeugen. In beiden Fällen wird im Speicher des Computers ein vollständiges Abbild des XML-Dokuments erzeugt. Dieses besteht aus unterschiedlichen Objekt-Typen und deren Verknüpfungen. Dies ist ein elementarer Unterschied im Vergleich zu SAX. Dort wurde ein XML-Dokument sequentiell eingelesen. Es konnte immer nur das verarbeitet werden, was gerade gelesen oder zuvor manuell zwischengespeichert wurde. Der Speicherverbrauch von DOM ist daher bei großen XML-Dokumenten erheblich größer als der von SAX. Auf der anderen Seite erlaubt DOM sehr schnelle Zugriffzeiten auf die einzelnen Elemente eines XML-Dokuments.

4.2.1 Parsen mit DOM

In diesem Abschnitt wird Ihnen gezeigt, wie Sie die DOM API dafür einsetzen können, XML-Dokumente zu parsen. In dem gezeigten Beispiel wird ein XML-Kalendereintrag vollständig in ein DOM-Objekt überführt und anschließend in eine Instanz der `Appointment`-Klasse umgewandelt, die Sie bereits aus den Beispielen des Abschnitts 4 kennen. Das zu parsende XML-Dokument ist identisch mit dem des SAX-Beispiels und in Listing 3.1 zu finden.

Den Quellcode für den DOM-Parser finden Sie in Listing 4.3. An dieser Stelle sei angemerkt, dass es natürlich auch für DOM Möglichkeiten gibt, die Verantwortungsbereiche des Parsers sauber aufzuteilen, so wie es im SAX-Beispiel geschehen ist. Die Methoden und Überlegungen sind jedoch sehr ähnlich. Aus diesem Grund ist das Ziel dieses Beispiels nicht, eine saubere Architektur zu beschreiben, sondern einen schnellen Einblick in die DOM API zu ermöglichen.

Wie Sie bereits gelernt haben, bietet JAXP nur eine Schnittstellenbeschreibung. Die Implementierung des DOM-Parsers wird daher wieder über eine Factory erzeugt (vgl. Factory-Pattern in [GHJV94]). Für DOM existieren verschiedene Factory-Klassen, je nachdem welche Implementierung Sie verwenden möchten. Das Zielsystem dieses Beispiels schränkt uns jedoch in der Auswahl ein. Aus diesem Grund erzeugen wir in (1) eine Instanz der `DocumentBuilderFactory`, die wir direkt im Anschluss dafür verwenden, eine Instanz der `DocumentBuilder`-Klasse zu erzeugen. Diese repräsentiert den eigentlichen DOM-Parser. Diesen Parser könnten wir bereits dafür verwenden, Eingaben vom Typ `InputSource` zu parsen. Dies könnten `InputStreams` oder unter anderem Instanzen diverser `Reader`-Klassen sein. In diesem Beispiel beschreiten wir jedoch einen anderen Weg.

Zum eigentlichen Parsen von XML-Dokumenten verwenden wir nicht direkt den DOM-Parser, sondern eine Instanz der `LSParser`-Klasse. Das „LS“ steht für „load and save“, also für Laden und Speichern. Die Schnittstelle dieses Parsers erlaubt es auf sehr einfache Weise, XML-Dokumente über die Angabe von Dateipfaden zu parsen und später auch DOM-Objektstrukturen in Dateien zu schreiben. Der LSParser wird im Beispiel 4.3 bei Schritt (2) erzeugt. Über die verschiedenen Modi, die vom LSParser unterstützt werden, können Sie sich unter [DOM03] informieren.

Kernstück jeder Verarbeitung eines XML-Dokuments mit DOM ist die DOM-Objektstruktur. Diese wird bei Punkt (3) erzeugt. Dank der Verwendung des LSParsers reicht ein einziger Befehl zum Laden und Parsen des gesamten Dokuments. Der Rückgabewert `doc` ist eine Referenz auf die erzeugte DOM-Objektstruktur. Streng genommen ist damit bereits der gesamte Vorgang des Parsens

abgeschlossen. Alle weiteren Schritte beschäftigen sich nur noch damit, die unterschiedlichen Felder des Kalendereintrags zu extrahieren und ggf. zu konvertieren.

Um auf die verschiedenen Elemente des geparsten XML-Dokumentes zuzugreifen, verfügt die DOM API über zahlreiche Befehle. An dieser Stelle können wir nicht jeden Befehl einzeln vorstellen. Eine genaue Liste aller Methoden finden Sie in der Dokumentation der DOM API [DOM03]. Schritt (4) zeigt Ihnen, wie Sie aus dem gesamten XML-Dokument alle Elemente eines bestimmten Namens auslesen können. Dies erfolgt über den Befehl `getElementsByName(ElementName)`. Selbst wenn es nur ein Element mit diesem Namen geben sollte, wie es hier der Fall ist, ist der Rückgabewert immer eine Liste vom Typ `NodeList`. Diese Liste enthält alle Knoten mit dem angegebenen Namen. Der Typ `Node` ist die Superklasse aller möglichen XML-Bestandteile. Auf die einzelnen Knoten der Liste können Sie mit der Methode `item(Index)` zugreifen. Da wir an dieser Stelle wissen, dass die Liste nur einen Knoten enthalten kann, können wir ohne Probleme einfach den Index 0 verwenden. Alternativ wäre auch das Iterieren über die gesamte Liste möglich, wie wir es später in Schritt (9) noch sehen werden. Da es sich bei `Node` nur um die Superklasse handelt, muss noch ein entsprechender Cast nach `Element` vollzogen werden.
Ein völlig anderer Weg, einzelne Knoten aus der DOM-Objektstruktur zu extrahieren, bietet die Methode `getElementByID()`. Um diese Methode zu verwenden, muss das XML-Element jedoch über ein ID-Attribut mit eindeutigem Wert verfügen. Der Umweg über die `NodeList` bliebe uns bei diesem Vorgehen dadurch erspart.

Nachdem im vorhergehenden Schritt der Wurzelknoten `appointment` extrahiert wurde, kann im Schritt (5) damit begonnen werden, auf die eigentlichen Werte des Kalendereintrags zuzugreifen. Die Adressierung der einzelnen Elemente erfolgt auf dem gleichen Wege wie in Schritt (4). Der einzige Unterschied ist, dass an dieser Stelle der Wurzelknoten als Ausgangspunkt verwendet wird und nicht mehr das Gesamtdokument. Des Weiteren wird die `NodeList` nicht mehr extra zwischengespeichert, sondern direkt weiterverarbeitet. Der letzte Methodenaufruf `getTextContent()` gibt den Wert des Elements als `String` zurück. Dieses Vorgehen ist für alle Elemente des Kalendereintrags gleich.

Der Kalendereintrag verwendet für die Speicherung auch Attribute. Diese werden in Schritt (6) verarbeitet. Die entsprechende Methode heißt `getAttribute()`. Als Argument erhält diese Methode den Namen des Attributs. Aufgerufen wird diese Methode auf dem `Element`-Objekt des entsprechenden Knotens. In diesem Fall sind die Attribute Teil des `appointment`-Knotens. Die Methode wird daher auf der Instanz `appointmentEle` angewendet. Sollte das gesuchte Attribut nicht gefunden werden oder keinen Wert enthalten, ist der Rückgabewert der Methode eine leere Zeichenkette.

Alle Werte des Kalendereintrags liegen bisher nur als Zeichenkette vor. Einträge wie das Datum oder die Länge des Termins müssen jedoch noch in die entsprechenden Datenformate konvertiert werden. Dies erfolgt im Schritt (7). Die Konvertierung der einzelnen Werte wird, aus Gründen der Übersichtlichkeit dieses Beispiels, von einer externen Klasse vorgenommen. Die dafür notwendigen Schritte sind kein Bestandteil der eigentlichen XML-Verarbeitung und werden daher an dieser Stelle nicht weiter erörtert. Bei Bedarf finden Sie den kommentierten Quellcode der Klasse `AppointmentFieldConverter` im Anhang dieser Arbeit.
Ist die Konvertierung abgeschlossen, liegen alle Informationen vor, die für die Erstellung einer Instanz der `Appointment`-Klasse notwendig sind. Den Aufruf des Konstruktors sehen Sie im Schritt (8).

Als Letztes müssen noch die Teilnehmer des Termins geparst werden. Dies erfolgt innerhalb des Schritts (9). Das Vorgehen zum Extrahieren der Elementwerte und Attribute unterscheidet sich nicht von dem, was bisher angewendet wurde. Neu ist an dieser Stelle, dass es mehrere `attendee`-Elemente geben kann, während alle anderen bisher betrachteten Einträge maximal nur einmal vorkommen konnten. Damit alle `attendee`-Einträge betrachtet werden können, werden sie über den bekannten Befehl `getElementsByTagName()` innerhalb einer `NodeList` gespeichert. Über diese Liste wird im Anschluss iteriert. Dafür wird in einer `for`-Schleife ein Zähler initialisiert und bis zu der Anzahl der Knoten innerhalb der Liste hochgezählt. Diese kann über `getLength()` ermittelt werden. Alle weiteren Schritte im Rumpf der Schleife unterscheiden sich nicht von denen der Schritte (5)-(9).
Der Kalendereintrag ist nun vollständig in die entsprechende Objektstruktur überführt. Das Ergebnis wird in Schritt (10) in der Konsole ausgegeben. Dafür wird wieder die externe Klasse `AppointmentPrinter` verwendet. Den kommentierten Quelltext finden Sie im Anhang. Die Ausgabe des Parsers ist identisch zu der des SAX-Parsers. Diese sehen Sie in Abbildung **??**.

Listing 4.3: XML-Dokumente parsen mit DOM

```
1  public class ParserDom {
2    public static void main( CommAreaHolder cah ) {
3
4      try {
5      PrintWriter out = Task.getTask().out;
6      out.println();
7
8      //## (1)
9      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance()
           ;
10     DocumentBuilder builder = factory.newDocumentBuilder();
11
12     //## (2)
13     DOMImplementationLS domImpl = (DOMImplementationLS) builder.
           getDOMImplementation();
14     LSParser parser = domImpl.createLSParser(DOMImplementationLS.
           MODE_SYNCHRONOUS, null);
15
16     //## (3)
17     Document doc = parser.parseURI("/u/prak500/classes/prak500/tutorials/
           sax/DateF1-001.xml");
18
19     //## (4)
20     NodeList appointmentNodes = doc.getElementsByTagName("appointment");
21     Element appointmentEle = (Element) appointmentNodes.item(0);
22
23     //## (5)
24     String dateStr = appointmentEle.getElementsByTagName("date").item(0).
           getTextContent();
25     String durationStr = appointmentEle.getElementsByTagName("duration").
           item(0).getTextContent();
26     String title = appointmentEle.getElementsByTagName("title").item(0).
           getTextContent();
27     String location = appointmentEle.getElementsByTagName("location").
           item(0).getTextContent();
28     String description = appointmentEle.getElementsByTagName("description
           ").item(0).getTextContent();
29
30     //## (6)
31     String stateStr = appointmentEle.getAttribute("state");
32     String user = appointmentEle.getAttribute("state");
33     String calendar = appointmentEle.getAttribute("calendar");
34     int ctime = (new Integer(appointmentEle.getAttribute("ctime"))).
           intValue();
35
36     //## (7)
37     Calendar dateAndTime = AppointmentFieldConverter.parseDateAndTime(
           dateStr);
38     int duration = AppointmentFieldConverter.parseDuration(durationStr);
39     State state = AppointmentFieldConverter.parseStateStr(stateStr);
40
```

```
41      //## (8)
42      Appointment appointment = new Appointment(  dateAndTime, duration,
            title, description,ocation, calendar,state, user, ctime);
43
44      //## (9)
45      NodeList attendeeNods = appointmentEle.getElementsByTagName("attendee
            ");
46      for( int i=0; i < attendeeNods.getLength(); i++ ) {
47          Element attendeeElement = (Element) attendeeNods.item(i);
48          State attState = AppointmentFieldConverter.parseStateStr(
                attendeeElement.getAttribute("state"));
49          String attUser = attendeeElement.getAttribute("user");
50          String attName = attendeeElement.getElementsByTagName("name").
                item(0).getTextContent();
51          Attendee21 attendee = new Attendee21(attName, attUser, attState);
52          appointment.addAttendee(attendee);
53      }
54
55      //## (10)
56      AppointmentPrinter.printAppointment(out, appointment);
57
58      } catch (ParserConfigurationException e) {
59        out.println("Parser␣Exception");
60      }
61    }
62  }
```

4.2.2 Erzeugen von XML-Dokumenten

Bisher wurde gezeigt, wie Sie mit SAX und DOM XML-Dokumente parsen können. In diesem Abschnitt drehen wir die Dinge einmal um und verwenden DOM, um ein XML-Dokument zu erzeugen. Als Beispiel wollen wir einen Kalendereintrag erzeugen. Den Quelltext des Programms finden Sie im Listing 4.4.
Dieser Abschnitt zeigt Ihnen die Kernmethoden der DOM API, die zum Erzeugen von XML-Dokumente benötigt werden. Eine Dokumentation aller anderen API-Befehle, die hier nicht behandelt werden können, finden Sie in [DOM03]. Weitere Beispiele können Sie in [SN09] nachlesen.

Wie auch beim Parser beginnen wir in Schritt (1) mit dem Aufruf der Factory-Methode. Wir erzeugen zuerst eine Factory für den `DocumentBuilder` und laden eine Instanz des Builders. Auch dieses Mal beschreiten wir den Umweg über die `DOMImplementationLS` (vgl. 2. Absatz Abschnitt 4.2.1), um ein einfaches Laden und Speichern des erstellten XML-Dokuments zu ermöglichen.

Den zuvor erzeugten `DocumentBuilder` verwenden wir in Schritt (2) dafür, eine leere Instanz eines DOM-Dokuments zu erzeugen. Wir erinnern uns, das DOM-Objekt ist die Objektstruktur im Speicher, die das XML-Dokument repräsentiert. Innerhalb des leeren DOM-Dokuments erzeugen wir mit der `createElement`-Methode den Wurzelknoten. Als Argument erhält diese Methode den Namen des zu erzeugenden Elements. In unserem Fall ist dies „appointment“.

Bevor wir damit beginnen die Kinderelemente des `appointment`-Knotens zu erzeugen, ergänzen wir diesen in Schritt (3) um dessen Attribute. Diese Aufgabe wird durch die Methode `setAttribute()` erledigt. Diese erhält zwei Argumente. Das erste ist der Name des zu erzeugenden Attributes und der zweite dessen Wert. Der Aufruf dieser Methode erfolgt auf einer Instanz der Element-Klasse.

In Schritt (4) beginnen wir mit dem Erzeugen der Kinderknoten. Das Schema ist dabei für jeden Knoten dasselbe. Zuerst wird mit `createElement` das entsprechende Kind erzeugt. Als nächstes enthält dieses seinen Wert. Dafür wird auf der erzeugten `Element`-Instanz die Methode `setTextContent` aufgerufen. Das übergebene Argument ist vom Typ `String` und enthält den Wert des Elements. Als Letztes muss der erzeugte Knoten noch an den entsprechenden Elternknoten angehängt werden. Dies erfolgt mit dem Befehl `appendChild()`. Er wird als Methode des Elternknotens aufgerufen und erhält als Argument die Instanz des anzuhängenden Kindknotens.
Da dieses Vorgehen für alle Elemente gleich ist, zeigt das Beispiel auch nicht die Erzeugung aller Elemente. Den vollständigen kommentierten Quelltext finden Sie im Anhang.

Im letzten Schritt (5) verwenden wir die in (1) erzeugte Instanz der `DOMImplementationLS`, um einen Serializer zu initialisieren. Dieser kann das erzeugte DOM-Dokument automatisch in eine Zeichenkette umwandeln. Diesen Vorgang nennt man Serialisieren. Auf der Instanz des Serializers wird dafür die Methode `writeToString` aufgerufen, als Argument dient das DOM-Dokument. Die so erzeugte Zeichenkette kann im Anschluss einfach in eine Datei geschrieben werden oder, wie in diesem Beispiel, in der Konsole ausgegeben werden. Die Ausgabe sehen Sie in Abbildung 4.2. Leider sind die semantischen Hilfen wie Zeilenumbrüche und Einrückung nicht Teil der Serialisierung.

```
xdom◊
<?xml version="1.0" encoding="UTF-16"?><appointment calendar="university" ctime=
"1282918186" state="confirmed" user="stefan"><date>2010-10-01T10:22</date><durat
ion>90</duration><title>Using LP for production optimization</title><location>Sa
nd A113</location><description>Talk about real world project</description><atten
dees/></appointment>◊
```

Abbildung 4.2: Ausgabe des DOM-Parsers

Listing 4.4: XML-Dokumente erstellen mit DOM

```
1  class DomCreator {
2    public static void main ( CommAreaHolder cah ) {
3      PrintWriter out = Task.getTask().out;
4      out.println();
5
6      try {
7
8        //## (1)
9        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance
             ();
10       DocumentBuilder builder = factory.newDocumentBuilder();
11       DOMImplementationLS domImpl = (DOMImplementationLS) builder.
             getDOMImplementation();
12
13       //## (2)
14       Document doc = builder.newDocument();
15       Element appointment = doc.createElement("appointment");
16
17       //## (3)
18       appointment.setAttribute("user", "stefan");
19       appointment.setAttribute("ctime", "1282918186");
20       appointment.setAttribute("calendar", "university");
21       appointment.setAttribute("state", "confirmed");
22
23       //## (4)
24       Element name = doc.createElement("date");
25       name.setTextContent("2010-10-01T10:22");
26       appointment.appendChild(name);
27
28       Element duration = doc.createElement("duration");
29       duration.setTextContent("90");
30       appointment.appendChild(duration);
31
32       Element title = doc.createElement("title");
33       tile.setTextContent("Using␣LP␣for␣production␣optimization");
34       appointment.appendChild(title);
35
36       /* .... */
37
38       Element attendees = doc.createElement("attendees");
39       Element attendee1 = doc.createElement("attendee");
40       attendee1.setAttribute("user", "ray");
41       attendee1.setAttribute("state", "invited");
```

```
42        attendee1.appendChild(attendees);
43
44        Element att1Name = doc.createElement("name");
45        att1Name.setTextContent("Prof. Dr. Ray");
46        attendee1.appendChild(att1Name);
47
48        appointment.appendChild(attendees);
49
50        //## (5)
51        LSSerializer serializer = domImpl.createLSSerializer();
52        String xmlStr = serializer.writeToString(appointment);
53        out.println(xmlStr);
54
55     } catch (ClassCastException e) {
56              out.println("Class Cast Exception");
57     } catch (ParserConfigurationException e) {
58              out.println("Parser Configuration Exception");
59
60     }
61   }
62 }
```

4.3 XPATH und XSLT

XSLT ist die Abkürzung für “Extensible Stylesheet Language Transformation”. Diese ermöglicht es, Dokumente auf XML-Basis als Eingabe zu verarbeiten und in beinahe beliebiger Form wieder auszugeben. Grundlage für diesen Vorgang ist neben der XML-Eingabe eine XSL-Schablone, welche das Zielformat definiert. Diese beschreibt mit Hilfe von XPATH, welche Informationen aus der Eingabedatei extrahiert werden müssen und wie sie innerhalb des neuen Zielformates wiedergegeben werden. Die eigentliche Transformation wird im Anschluss auf Basis der Schablone durch einen XSLT-Prozessor durchgeführt. Das bedeutet, dass der Entwickler nur die Schablone zu definieren hat und keinen weiteren Programmcode, wie etwa einen eigenen Parsen, schreiben muss. Dieses Verfahren bietet daher einen sehr schnellen und effizienten Weg, Datenformate ineinander zu überführen. Ein Vorgang, der sich im Umfeld heterogener Computersysteme, wie sie bei der Softwareintegration meistens anzufinden sind, hervorragend dafür eignet Adapter zu entwickeln.
In den folgenden beiden Abschnitten erhalten Sie eine Einführung in XPATH und XSLT. Es werden Ihnen die Grundlagen beider Techniken erläutert und ein Teil der entsprechenden API vorgestellt. Dies versetzt Sie in die Lage bei Bedarf mit Hilfe der offiziellen Spezifikation beider Sprachen schnell und einfach weitere Aspekte der Sprachen kennenzulernen. Im Anschluss an die Einführung wird Ihnen anhand eines kleinen Beispiels der praktische Einsatz von XSLT demonstriert.

4.3.1 XPATH

Die “XML Path Language” oder kurz XPATH ist eine vom W3C entwickelte Abfragesprache. Sie dient dazu, Teile eines XML-Dokuments zu adressieren und so Information zu extrahieren. Neben dem Einsatz innerhalb der XSL-Transformation basieren noch andere Techniken wie XQuery oder XPointer auf dieser Sprache.

Für die Adressierung unterscheidet XPATH zwischen folgenden Knotentypen eines XML-Dokuments [SN09]: Wurzelknoten, Elementknoten, Textknoten, Attributknoten, Namensraumknoten, Verarbeitungshinweise, Kommentare.

Innerhalb dieser kurzen Einführung und dem später folgenden Beispiel werden wir nicht auf alle aufgeführten Typen zugreifen. Aus diesem Grund sei an dieser Stelle für weitere Informationen auf die Spezifikation von XPATH [XPA10] verwiesen.
Für die Beschreibung der verschiedenen Verfahrung zur Adressierung betrachten wir in den folgenden Absätzen das XML-Beispiel 3.1.

Für die Lokalisierung von Knoten innerhalb eines XML-Dokuments verwendet XPATH Pfadangaben, die in ihrer Syntax denen eines UNIX/Linux-Systems ähneln. Ein Beispiel für einen solchen Pfad ist `/appointment/time`. Dieser Pfad selektiert das `time`-Element unterhalb eines `appointment`-Knotens. Die Lokalisierungspfade verwenden dafür verschiedene Symbole, wie etwa den Slash “/”. Eine Liste der häufigsten Symbole und deren Bedeutung finden Sie in Tabelle 4.1.

Ausruck	Beschreibung
/	Trennt zwei Knoten
//	Selektiert alle Knoten unabhängig von deren Position
.	Selektiert den aktuellen Knoten
..	Selektiert den Elternknoten
@	Selektiert ein Attribut

Tabelle 4.1: Ausdrücke innerhalb XPATH-Lokalisierungspfade

Betrachten wir nun ein paar Beispiele für die oben aufgeführten Pfadausdrücke.
`/appointment/attendees/attendee` selektiert alle `attendee`-Knoten innerhalb des `attendees`-Elements.
`//name` wählt alle `name`-Elemente innerhalb des Dokuments aus, unabhängig von deren Position. In diesem Beispiel wären dies die Elemente mit den Werten "Prof. Dr. Rayn" und "Dr. Taylor".
`/appointment/@state` selektiert das `state`-Attribut des `appointment`-Knotens.
Des Weiteren sind alle oben aufgeführten Ausdrücke beinahe beliebig miteinander kombinierbar. Der Ausdruck `//attendee/@state` selektiert alle `state`-Attribute der `attendee`-Knoten, unabhängig von deren Position.
Hilfreich bei der Gestaltung von XPATH-Abfragen ist ein XPATH-Visualisierer, der die durch einen XPATH-Ausdruck selektierten Elemente innerhalb eines XML-Dokuments hervorhebt. Abbildung 4.3 zeigt den Einsatz eines solchen Hilfsmittels.

Zusätzlich zu den in Tabelle 4.1 vorgestellten Ausdrücken unterstützt XPATH auch noch den Einsatz von Wildcards. Diese können unter anderem dafür eingesetzt werden, Knoten zu adressieren, deren Namen nicht bekannt ist. Es existieren drei verschiedene Wildcards, die in Tabelle 4.2 aufgelistet sind.

Wildcard	Beschreibung
*	Passt auf alle Elementknoten
@*	Passt auf alle Attributknoten
node()	Passt auf alle Knotentypen

Tabelle 4.2: Wildcards innerhalb XPATH-Lokalisierungspfad

Durch den Ausdruck `//@*` werden zum Beispiel alle Attribute des XML-Dokuments selektiert.

Die bisher verwendeten Formen der Lokalisierungspfade adressieren immer Knotenmengen anhand deren Namen oder Position. XPATH erlaubt es jedoch auch, Knoten auf Basis ihres Wertes zu filtern. Diese Aufgabe wird innerhalb der Sprache durch Prädikate erfüllt. Dies sind Ausdrücke, die sich zu `wahr` oder `falsch` auswerten lassen. Sie werden dafür verwendet Kriterien zu definieren, anhand derer eine ausgewählte Knotenmenge gefiltert werden kann. In XPATH werden Prädikate in eckigen Klammern "[]" eingefasst. Der Pfad `//attendee[@state = 'confirmed']` wählt auf diese Weise alle Teilnehmer aus, die ihre Teilnahme bereits bestätigt haben.

Innerhalb der Prädikate können verschiedene Operatoren eingesetzt werden. Dabei handelt es sich im Allgemeinen um die Standardoperatoren, wie sie auch aus anderen Programmiersprachen bekannt sind. Eine Auswahl der Operatoren finden Sie in Tabelle 4.3.

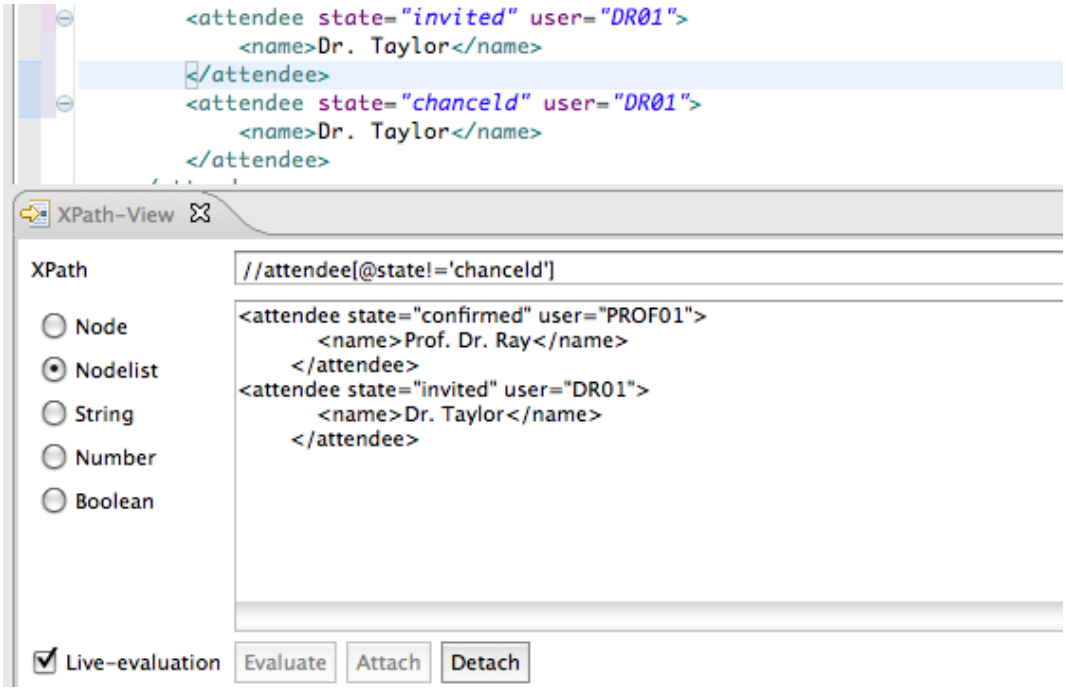


Abbildung 4.3: Visualisierung eines XPATH-Pfades in Eclipse mit Hilfe des Plugins XPath-View

Operator	Beschreibung
>	Größer als
<	Kleiner als
=	Test auf Gleichheit
>=	Größer oder gleich
<=	Kleiner oder gleich
!	Negation
and	LogischeS Und
or	Logisches Oder
|	Verknüpft zwei Ausdrücke

Tabelle 4.3: Operatoren für XPATH-Prädikate

Der Operator zur Verknüpfung zweier Ausdrücke würde im Beispiel

```
//attendee[@state='confirmed'] | //attendee[@state='invited']
```

alle Teilnehmer selektieren, die ihre Teilnahme noch nicht abgesagt haben. Das gleiche Ergebnis könnte mit der Abfrage `//attendee[@state!='chanceld']` erzielt werden.

Als letztes Sprachelement sollen an dieser Stelle die eingebauten Funktionen vorgestellt werden. Die Syntax eines Funktionsaufrus ist `Funktionsname(Argumente)`. XPATH versucht den Typ der übergebenen Argumente entsprechend zu interpretieren. Unterstützte Typen sind Strings, numerische Werte, Knotenmengen und boolesche Werte. Eine Liste der unterstützten Funktionen können Sie der XPATH-Spezifikation [XPA10] entnehmen. Ein Beispiel für eine eingebaute Funktion ist `count`. Die

Abfrage `count(//attendee)` gibt als Rückgabewert die Anzahl der eingetragenen Teilnehmer eines Termins zurück. Innerhalb einer Liste von Terminen könnte mit `//title[count(//attendee)>=3]` der Titel aller Termine mit mindestens drei Teilnehmern ausgegeben werden.

4.3.2 XSLT

Die "Extensible Stylesheet Language" (XSL) wird innerhalb der XSL-Transformation (XSLT) dafür verwendet, die Schablonen für das Ausgabeformat zu definieren. Diese Schablonen beschreiben die Struktur der Ausgabe, enthalten selbst jedoch noch keine Informationen aus der Eingabedatei. Stattdessen verwendet XSL Lokalisierungspfade, um so mit Hilfe von XPATH zu beschreiben, welche Daten aus der Eingabedatei extrahiert werden sollen. Bei der Verarbeitung der Transformation werden diese Lokalisierungspfade durch den XSL-Prozessor durch die entsprechenden Inhalte der Eingabedatei ersetzt.

Jede XSL-Schablone beginnt mit der Definition der verwendeten Version und des Namensraums

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                                                    version="1.0">
```

Kernstück der XSL-Schablone sind die definierten Templates:

```
<xsl:template match="XPATH-Ausdruck">
```

Der XPATH-Ausdruck im `match`-Attribut beschreibt die Knotenmenge der Eingabedatei, auf die das Template angewendet werden soll. Innerhalb einer Schablone können mehrere Templates definiert werden.

Innerhalb des Templates werden weitere XSL-Instruktionen dafür verwendet, die eigentlichen Daten der Eingabedatei zu verarbeiten. Eine der wichtigsten Instruktionen ist:

```
<xsl:value-of select="XPATH-Ausdruck"/>
```

Hiermit könen Sie den Wert des adressierten Knotens extrahieren. Bei der Verarbeitung der Transformation wird dann der gesamte XSL-Befehl durch den gelesenen Wert der Eingabedatei ersetzt. Über diesen Befehl können Sie auch die Rückgabewerte von XPATH-Funktionen ausgeben.

Neben Befehlen zur Datenextraktion existieren innerhalb von XSL auch Kontrollstrukturen. Eine Fallunterscheidungen definieren Sie mit:

```
<xsl:if test="XPATH-Prädikat">
```

Der Rumpf des `if`-Knotens wird dann nur ausgewertet, wenn das verwendete Prädikat nach `wahr` ausgewertet wird.
Knotenmengen verarbeiten Sie zum Beispiel mit:

```
<xsl:for-each select="XPATH-Ausdruck">
```

Der Code zwischen dem Start- und End-Knoten der `for-each`-Anweisung wird dann für jeden Knoten ausgewertet, der durch den XPATH-Ausdruck selektiert wurde. Sie können die Knoten einer

Menge sogar sortieren:

```
<xsl:sort select="XPATH-Ausdruck">
```

Die Knoten werden dann auf Basis der Werte des adressierten Knotens sortiert.

Ein Beispiel für den Einsatz von XSL-Schablonen finden Sie in Listing 4.5. Dieses wird im Anschluss in Abschnitt 4.3.3 besprochen. Natürlich enthält XSL noch weitaus mehr Möglichkeiten der Eingabeverarbeitung. Eine gesamte Auflistung würde jedoch den Rahmen dieses Kapitels sprengen. Eine vertiefende Einführung finden Sie in [SN09]. Eine Beschreibung des gesamten Sprachumfangs enthält die Spezifikation unter [XSL10].

4.3.3 XSLT am Beispiel

Ziel dieses Beispieles ist es, einen Termin in XML-Form für die Ausgabe in eine HTML-Datei zu überführen. Für diesen Vorgang benötigen wir drei Teile:

1. Die Eingabedatei. Dies wird das XML-Dokument sein, das Sie bereits aus Listing 3.1 kennen.
2. Eine Schablone für das HTML-Dokument, welches erzeugt werden soll. Den Quelltext der Schablone finden Sie in Listing 4.5.
3. Eine Java-Klasse, die eine Instanz des Transformers erstellt und die eigentliche Transformation startet. Diese heißt innerhalb dieses Beispiels "TutXpathMain" und ist in Listing 4.6 abgedruckt.

In den folgenden Absätzen werden wir die Schablone und die Java-Klasse näher betrachten und uns mit ihrer Funktionsweise vertraut machen.

Abbildung 4.4: Die transformierte HTML-Ausgabe der Kalendereintrags

Beginnen wir mit dem Quellcode der Schablone aus Listing 4.5.
Direkt im Anschluss an den Header sehen Sie bei Punkt (1) die Definition des Templates. Diese Schablone verwendet nur ein Template, dessen Geltungsbereich das gesamte XML-Dokument der Eingabe umfasst. Dies wird erreicht, indem der Wert des `match`-Attributs auf den Wurzelknoten „/“ gesetzt

wird.
Die erste Extrahierung eines Wertes folgt in Schritt (2). Für den Titel des HTML-Dokuments wird der Titel des Termins ausgelesen. Da das XML-Format, welches wir transformieren wollen, nur ein `titel`-Element enthält, reicht für die Adressierung der positionsunabhängige Pfad `//title`.
Auf den Wert eines Attributs wird in Schritt (3) zugegriffen. An dieser Stelle wird der Name des verwendeten Kalenders über den Pfad `//@calender` extrahiert. Der XSL-Befehl zum Einsetzen des Wertes ist derselbe wie in Schritt (1). Es wird die `value-of`-Instruktion verwendet. Dies gilt für alle Stellen, an denen Informationen der Eingabe in die Schablone eingefügt werden sollen.
Schritt (4) zeigt eine etwas aufwändigere Verarbeitung der Eingabedaten. An dieser Stelle wird auf das Datum des Termins zugegriffen. Dieses ist innerhalb der Eingabedatei jedoch in einem anderen Format gespeichert, als wir es für die Ausgabe wünschen. Aus diesem Grund teilen wir in Punkt (4-a) die Zeichenkette des Datums und extrahieren jeweils nur einen Teil des Datums. Die einzelnen Teile werden dann neu angeordnet und ausgegeben. Zum Teilen einer Zeichenkette verwenden wir die XPATH-Funktion `substring()`. Als Argumente übergeben wir die Ausgangszeichenkette, die erste Position, die wir lesen wollen, und die Länge des zu lesenden Bereichs. Zu beachten ist, dass Index-Werte in XPATH bei 1 und nicht bei 0 beginnen. Auch die Dauer des Termins wird weiterverarbeitet. Ziel ist es, die Angaben in Minuten, in Stunden und Minuten aufzuteilen. Dafür werden die XPATH- internen Rechenoperatoren verwendet. Der Stundenanteil wird mit der Division `div` und anschließendem Abrunden `floor()` berechnet. Der restliche Minutenanteil kann dann über den `mod`-Operator ermittelt werden. Dieser gibt den Rest einer Modulorechnung aus. Für beide Rechnungen muss der Wert des Elements zuerst noch in eine Zahl umgewandelt werden. Dies erfolgt über die Funktion `number()`.
Den Einsatz einer Kontrollstruktur sehen Sie in Schritt (5). Die hier eingesetzte `if`-Anweisung testet, ob das Feld für die Terminbeschreibung leer ist. Sollte dies nicht der Fall sein, wird der Inhalt inklusive einer farbigen Box in die HTML-Ausgabe eingefügt. Sollte keine Beschreibung vorliegen, wird auch die Box ausgeblendet. Gerade für optionale Knoten innerhalb der Eingabe ist dieses Vorgehen sehr hilfreich.
Die Liste der Teilnehmer erzeugen wir mit Hilfe einer `for-each`-Anweisung in Punkt (6). Diese enthält als Eingabe alle `attendee`-Elemente, die über den Pfad `//attendee` adressiert wurden. Innerhalb der `for-each`-Anweisung können dann die Kinderknoten der `attendee`-Elemente über den `./ELEMENTNAME` angesprochen werden. Der Punkt am Beginn des Pfades verweist dabei immer auf den aktuell ausgewählten Knoten der Schleife.

Nachdem wir die Funktionsweise der Schablone kennengelernt haben, widmen wir uns nun der Java-Hauptklasse aus Listing 4.6.
Als Erstes erzeugen wir in Schritt (1) auf gewohnte Art und Weise über das Factory-Pattern eine Instanz der `TransformerFactory`. Diese verwenden wir später für die Erstellung des eigentlichen XSL-Prozessors.
Zuvor sorgen wir in Schritt (2) noch dafür, dass eine schreibbare und vor allem leere Ausgabedatei auf dem System vorhanden ist.
Schritt (3) nutzen wir für die Erzeugung des Templates, eine Java-Objekt- Variante unserer Schablone. Diese wird über die `newTemplate`-Methode der in (1) erstellten Factory erstellt. Als Argument erhält diese Methode einen Eingabestrom, der auf die zu verwendende Schablone verweist. Diese heißt in diesem Beispiel “tansformer.xsl”. Auf Grund der Darstellung als Java-Objekt muss das zeitintensive Einlesen der `xsl`-Datei nur einmal durchgeführt werden.
Der XSL-Prozessor wird in Schritt (4) erstellt über eine Factory-Methode, der Template-Instanz. Das bedeutet, dass für jede Schablone ein neuer XSL-Prozesser erzeugt werden muss. Dies hat den Hinter-

gund, dass jeder Prozessor über die Factory-Methode speziell auf die Schablone zugeschnitten wird [SN09]. Auf diese Weise ist es möglich, die Funktionsweise des Prozessors auf das Benötigte zu reduzieren und so eine sehr effiziente Verarbeitung der Eingabe zu erlauben.
Bevor wir die Transformation starten, erzeugen wir in Schritt (5) und (6) noch die Ein- und Ausgabeströme für die Eingabe- und Ausgabedatei.
Zum Schluss wird bei Punkt (7) die Transformation gestartet. Das Ergebnis können Sie dann in Form der erzeugten Ausgabe „date.html“ betrachten. Ein Screenshot der HTML-Ausgabe sehen Sie in Abbildung 4.4.

Zusammenfassung

In diesem Kapitel wurden Ihnen drei Schnittstellen für die Verarbeitung von XML mit Java vorgestellt: SAX, DOM und XSLT.
Die SAX-Schnittstelle kann dafür verwendet werden XML-Dokumente seriell einzulesen und zu verarbeiten. SAX verarbeitet Dokumente eventbasiert mit Hilfe eines Eventhandlers. Auf Grund der seriellen Verarbeitung ist SAX besonders gut dafür geeignet große XML-Dokumente zu verarbeiten, da nie eine gesamte Kopie des Dokuments im Speicher gehalten werden muss.
Das Document Object Model (DOM) dient ebenfalls dem Parsen von XML-Dokumenten. DOM erzeugt dafür eine komplette Kopie der Dokumentstruktur im Speicher. Dies erlaubt auf der einen Seite einen schnellen Zugriff auf die einzelnen Elemente, verbraucht bei großen Dokumenten jedoch auch erheblich mehr Speicher. Des Weiteren kann DOM auch dafür verwendet werden, XML-Baumstrukturen zu modifizieren oder komplette XML-Dokumente zu erzeugen.
Die Extensible Stylsheet Transformation hat die Aufgabe, XML basierte Dokumente zu transformieren, das bedeutet, von einem Datenformat in ein anderes zu überführen. Basis dieses Vorgangs ist der XSLT-Prozessor, der eingehende Dokumente parst und sie mit Hilfe einer Dokumentenschablone in das Zielformat überführt. Für die Extrahierung einzelner Informationen aus dem Eingabedokument wird XPATH verwendet. Dies ist eine Methode für die Adressierung von XML-Elementen.

Aufgaben

1. *Entwickeln Sie einen SAX-Parser für das im Anhang [YY] aufgelistete XML-Format einer Musiksammlung. Verwenden Sie den Parser analog zum Beispiel [YY] dafür, die einzelnen Elemente des XML-Dokuments innerhalb einer Java-Objektstruktur darzustellen und deren Werte auf dem CICS Terminal auszugeben.*

2. *Entwickeln Sie für das Beispiel [YY] einen DOM-Parser. Dessen Aufgabenbereich ist äquivalent zu dem des SAX-Parsers.*

3. *Verwenden Sie die DOM-API für die Erstellung einer neuen Instanz der Musiksammlung. Ihr Quellcode sollte ähnlich zu dem des Beispiels [YY] funktionieren.*

4. *Entwickeln Sie eine XSL-Transformation, die als Eingabe einen XML-Eintrag der Musiksammlung erhält und diesen als HTML-Datei zurückgibt.*

Listing 4.5: Die XSL-Schablone für die Transformation

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version=
       "2.0">
3  <xsl:output method="html"/>
4  <!-- ## (1) -->
5    <xsl:template match="/">
6      <html>
7        <head>
8          <!-- ## (2) -->
9          <title>Appointment: <xsl:value-of select="//title" /></title>
10       </head>
11       <body style="font-family:Arial; font-size:1em;">
12
13         <div style="width:50em; padding:0.5em; border:1px solid #000000;"
               >
14
15           <!-- ## (3) -->
16           <span style="font-size:1.2em; color:#3e3e3e">calendar: <
                 xsl:value-of select="//@calendar" /></span>
17           <h1 style="color:#467cba">Appointment: <xsl:value-of select="//
                 title" /></h1>
18           <h3><xsl:value-of select="//location" /></h3>
19
20           date:
21           <!-- ## (4a) -->
22           <xsl:value-of select="substring(//date,9,2)" />.
23           <xsl:value-of select="substring(//date,6,2)" />.
24           <xsl:value-of select="substring(//date,1,4)" />
25           <br/>
26           begin:
27           <xsl:value-of select="substring(//date,12,2)" />:
28           <xsl:value-of select="substring(//date,15,2)" />
29           <br/>
30           duration:
31           <!-- ## (4b) -->
32           <xsl:value-of select="floor(number(//duration) div 60)" /> hour
33           <xsl:text disable-output-escaping="no"> </xsl:text>
34           <xsl:value-of select="(number(//duration) mod 60)" /> minutes
35
36           <!-- ## (5) -->
37           <xsl:if test="//description != ''">
38             <div style="background-color:#e2e3d4; padding:1em;">
39               <xsl:value-of select="//description" />
40             </div>
41           </xsl:if>
42
43           <h4 style="font-size:1em; margin-bottom:5px;">Attendees:</h4>
44           <!-- ## (6) -->
45           <xsl:for-each select="//attendee">
46             <li>
47               <!-- ## (7) -->
```

```
48              <xsl:value-of select="./name"/>
49              (<xsl:value-of select="./@state"/>)
50            </li>
51          </xsl:for-each>
52
53        </div>
54      </body>
55    </html>
56  </xsl:template>
57 </xsl:stylesheet>
```

Listing 4.6: Die Java-Hauptklasse der XSL-Transformation

```
1  package prak500.tutorials.xpath;
2
3  import com.ibm.cics.server.CommAreaHolder;
4
5  import java.io.File;
6  import java.io.FileInputStream;
7  import java.io.FileNotFoundException;
8  import java.io.FileOutputStream;
9
10 import javax.xml.transform.Result;
11 import javax.xml.transform.Source;
12 import javax.xml.transform.Templates;
13 import javax.xml.transform.Transformer;
14 import javax.xml.transform.TransformerConfigurationException;
15 import javax.xml.transform.TransformerException;
16 import javax.xml.transform.TransformerFactory;
17 import javax.xml.transform.stream.StreamResult;
18 import javax.xml.transform.stream.StreamSource;
19
20 public class TutXpathMain {
21
22   public static void main ( CommAreaHolder cah ) {
23
24     //## (1)
25     TransformerFactory factory = TransformerFactory.newInstance();
26
27     //## (2)
28     File output = new File("date.html");
29     if( output.exists() )
30       output.delete();
31
32     try {
33
34       //## (3)
35       FileInputStream fisT = new FileInputStream("transform.xsl");
36       Source tmplSource = new StreamSource(fisT);
37       Templates template = factory.newTemplates( tmplSource);
38
39       //## (4)
40       Transformer transformer = template.newTransformer();
```

```
41
42          //## (5)
43          FileOutputStream fos = new FileOutputStream("date.html");
44          Result result = new StreamResult( fos );
45
46          //## (6)
47          FileInputStream fisS = new FileInputStream("date.xml");
48          Source source = new StreamSource(fisS);
49
50          //## (7)
51          transformer.transform(source, result);
52
53      } catch (FileNotFoundException e) {
54          e.printStackTrace();
55      } catch (TransformerConfigurationException e) {
56          e.printStackTrace();
57      } catch (TransformerException e) {
58          e.printStackTrace();
59      }
60
61
62    }
63
64  }
```

Literaturverzeichnis

[Ada79] ADAMS, Douglas: *The Hitchhiker's guid to the galaxy*. delrey, 1979

[Bur09] BURGESS, Chris Rayns; G.: *Java Application Development for CICS*. IBM, Februar 2009

[DOM03] http://www.w3.org/2003/01/dom2-javadoc/

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E. ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1. Addison-Wesley, 1994

[IBMa] IBM (Hrsg.): *CICS Transaction Server for z/OS Release Guide*. 3.1. IBM, ftp://ftp.software.ibm.com/software/htp/cics/tserver/library/CICS_TS_V32_ReleaseGuide.pdf

[IBMb] IBM: *IBM JCICS API*. http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp?topic=/com.ibm.cics.ts31.doc/dfhpk/com/ibm/cics/server/package-summary.html, Abruf: 2010

[IBM99] IBM (Hrsg.): *CICS Resource Definition Guide*. 3. IBM, March 1999

[IBM08] IBM (Hrsg.): *Java Applications in CICS*. 3.2. IBM, 2008

[JAX10] https://jaxp-sources.dev.java.net/nonav/docs/api/

[SN09] SCHOLZ, Michael ; NIEDERMEIER, Stephan: *Java und XML*. 2. Galileo Computing, 2009

[W3S] http://www.w3schools.com/schema/default.asp

[xml00] http://www.edition-w3c.de/TR/2000/REC-xml-20001006/

[XPA10] http://www.w3.org/TR/xpath20/

[XSD00] http://www.w3.org/XML/Schema#dev

[XSL10] http://www.edition-w3.de/TR/2000/REC-xml-20001006/