Datentyp- und Interfaceeditor für Mobilfunkprotokolle auf der Basis von XML

Tobias Vogler

Konstanz, 31. März 2003

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Fachhochschule Konstanz

Hochschule für Technik, Wirtschaft und Gestaltung

Fachbereich Informatik/Wirtschaftsinformatik

Thema : Datentyp- und Interfaceeditor für Mobilfunkprotokolle auf der Basis von XML

Diplomand : Tobias Vogler, Wittstocker Str. 10, 10553 Berlin

Firma : Texas Instruments Berlin AG,
Alt Moabit 90a, 10559 Berlin

Betreuer : Prof. Dr. Ulrich Hedtstück

Eingereicht : Konstanz, 31. März 2003

Zusammenfassung

Thema	:	Datentyp- und Interfaceeditor für Mobilfunkprotokolle auf der Basis von XML
Autor	:	Tobias Vogler
Studiengang	:	Wirtschaftsinformatik
Eingereicht	:	31. März 2003
Schlagwörter	:	Datenstrukturen ; Mobilfunk ; XML ; Grammatik ; Editor ; Java ; Eclipse ; SWT ; JFace

Diese Arbeit befasst sich mit der Erstellung und Bearbeitung von Datenstrukturen, die für die Entwicklung von Software im Mobilfunksektor benötigt werden. Es wird aufgezeigt, wo diese Datenstrukturen auftauchen und welche Aufgaben ihnen zufallen. Am Beispiel der Texas Instruments Berlin AG wird eine mögliche Implementierung der für deren Handhabung eingesetzten Hilfsmittel im Detail betrachtet sowie auf die diesen innewohnenden Unzulänglichkeiten und Schwachstellen eingegangen. Zur Behebung und Umgehung der aufgezeigten Probleme werden verschiedene Ansätze analysiert und bewertet. Die hierzu benötigten und jeweils kurz vorgestellten Informationen über die Sprache XML und andere Techniken legen ferner offen, warum die letztendlich gewählte Vorgehensweise für die Neuimplementierung sinnvoll ist. Auf die für den reibungslosen Verlauf der Implementierungsphase erforderlichen Planungen wird ebenso wie auf die hierfür im Vorfeld notwendigen Überlegungen und Arbeiten eingegangen. Insbesondere wird auf die Erstellung mehrerer Prototyp-Versionen zur Erleichterung der Einarbeitung und zum besseren Verständnis der zu bearbeitenden Materie hingewiesen. Entwicklung und abschließendes Aussehen einer Grammatik als Grundlage für ein auf XML basierendes Datenformat werden vorgestellt sowie die hierbei auftretenden Problematiken erörtert. Es wird dargestellt, wie die für die Arbeit mit dem neuen Format entworfene Benutzerschnittstelle funktioniert und wie das der dahinterstehenden Funktionalität zugrundeliegende Konzept aussieht. Die Umsetzung dieses Konzepts unter Einsatz der Programmiersprache Java und aufbauend auf der „Eclipse“-Plattform wird umfassend und im Detail erläutert. Begleitende Tests und Evaluierungen werden angesprochen sowie Probleme und Herausforderungen der Implementierungsphase. Es wird darauf eingegangen, wie die Planungen für den zukünftigen Einsatz und die weitere Entwicklung des Projekts unter dem Dach der Texas Instruments Berlin AG aussehen. Und es werden die aus der Durchführung des Projekts – von den ersten Planungen bis hin zum vorläufigen Abschluss der Implementierungsphase und der Vorstellung eines funktionierenden Systems – gewonnenen Schlüsse und Erfahrungen präsentiert und ausgewertet.

Vorwort

Für heutige Computernutzer präsentiert sich ihr Arbeitsgerät zumeist in vielfältiger Buntheit: für jede denkbare Aufgabe scheint eine eigenständige, spezialisierte Lösung bereitzustehen. Dass jedes dieser Werkzeuge mit jeweils einer eigenen sogenannten „grafischen“ und angeblich intuitiv bedienbaren Benutzerschnittstelle ausgestattet ist, versteht sich dabei von selbst – und trägt nicht immer zur Erhöhung der Übersichtlichkeit bei. Enthaltsamkeit von solcherlei Komfort üben dabei häufig ausgerechnet jene unter den Computernutzern, die all die bunten Programme überhaupt erst entwickeln. In der Softwareentwicklung kommen auch heute vielfach noch die selben unspezialisierten und primitiven Werkzeuge zum Einsatz, wie bereits vor einigen Jahrzehnten. Um ein besonders problematisches Beispiel des Einsatzes von wenig geeigneten Werkzeugen in der täglichen Entwicklungsarbeit handelt es sich auch bei jener Anwendung, die neu zu implementieren sich diese Diplomarbeit anschickt. Dafür wurde ein derzeit noch ungewöhnlich erscheinender Weg eingeschlagen, um dem Reigen der Benutzerschnittstellen nicht ein weiteres vollkommen neues und rätselhaftes Exemplar hinzufügen zu müssen.

Die interessanten Möglichkeiten, die sich hieraus ergeben, und die Chance, eine Vielzahl unterschiedlicher Techniken miteinander zu vereinen und zu einem umfangreichen eigenständigen Ergebnis zusammenzuführen, waren für die Wahl des vorliegenden Themas für meine Diplomarbeit letztendlich entscheidend. Zusätzliche motivierende Faktoren waren die Einbettung meiner Tätigkeit in eine professionelle Arbeitsumgebung und die Möglichkeit zur eigenen Planung und Lenkung des Projekts im Rahmen des normalen – realen – Arbeitsablaufs. So bestand ein stabiler Rahmen für das definierte Ziel der Erstellung eines sinnvollen Werkzeugs, und der Dokumentation des Prozesses, der dorthin führen sollte.

Da bei diesem Projekt all Jene, die von der bisherigen anachronistischen Arbeitsweise betroffen waren, die Vorteile der geplanten Veränderungen klar erkennen konnten, waren mir Hilfe und unterstützende Anregungen aus den Reihen meiner Arbeitskollegen bei der Texas Instruments Berlin AG stets gewiss. Besonders möchte ich an dieser Stelle Andreas Kohn danken, der mich bei der Einführung in die Geheimnisse von XML und dessen Anwendung, aber auch mit Tests und Kritik tatkräftig unterstützt hat. Dass trotz vielerlei Ablenkungen, anderen zu erledigenden Arbeiten, einer mehrtägigen Schulung, und einem notwendig gewordenen privaten Umzug dieses Projekt dennoch zu einem Erfolg wurde, ist nicht zuletzt ihm zu verdanken. Auch meine Betreuer, Frank Reglin im Unternehmen selbst und Prof. Dr. Ulrich Hedtstück, haben mit viel Verständnis und Unterstützung das Ihre zum Gelingen beigetragen.

Tobias Vogler, März 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Dieses Dokument beschäftigt sich mit der Entstehung und Funktionsweise eines Editors für Daten im XML-Format. Der Editor ist insbesondere für die Bearbeitung von Daten ausgelegt, welche Definitionen von verschiedenen im Mobilfunkbereich benötigten Datenstrukturen beinhalten. In dieser Funktion soll er die bisher hierfür genutzte Textverarbeitungs-Software Microsoft Word ablösen, und so dem Benutzer den Umgang mit den zu wartenden Datenstrukturen erleichtern. Um auf längere Sicht eine einheitliche Benutzeroberfläche für neu entwickelte Werkzeuge zu erreichen, wird der Editor – in dieser Form zum ersten Mal bei Texas Instruments – als ganzes in eine sogenannte „Plattform"-Software eingebettet. Um alle gesteckten Ziele zu erreichen, sind eine Reihe von Vorarbeiten sowie eine detaillierte Planung erforderlich. Ein wesentliches Element stellt hierbei die Entwicklung einer geeigneten Grammatik für die XML-Daten dar, ein anderes die eingehende Erprobung der für den Editor gewählten Plattform. Die Vorstellung dieser Tätigkeiten sowie der eigentlichen Implementierung sind, neben einigen Erläuterungen zu den verwendeten Techniken und Methoden, Inhalt des vorliegenden Dokuments. Einzig wird dabei – bei der Erklärung der eigentlichen Implementierung – ein gewisses Vorwissen der Programmiersprache Java oder einer ähnlichen Sprache vorausgesetzt. Alles Weitere, insbesondere Konzept und Verwendung von XML, ist im notwendigen Rahmen im Dokument selbst erklärt.

Das Projekt wird unter dem Dach der Texas Instruments Berlin AG durchgeführt, und wird auch hauptsächlich in der in Berlin angesiedelten Softwareabteilung der Mobilfunksparte des Unternehmens zum Einsatz kommen. Das im Zusammenhang mit dem hier dokumentierten Projekt bestehende längerfristige Gesamtkonzept umfasst eine grundsätzliche Neugliederung der Arbeit mit den entsprechenden Daten, was nicht Gegenstand der hier vorgestellten Arbeit ist. Jedoch ist der zu entwickelnde Editor bereits für sich alleine sinnvoll einsetzbar und wird auch baldmöglichst nach Abschluss dieser Diplomarbeit im Unternehmen eingeführt werden. Sollte sich das hier verwendete – und im weiteren Verlauf ausführlich vorgestellte – Konzept bewähren, werden zukünftige Projekte auf dem gesammelten Wissen und den gewonnenen Erfahrungen dieses Projekts aufgebaut werden. Dies soll für die Zukunft zu einer hochgradigen Integration der verschiedenen für den Entwicklungsprozess eingesetzten Werkzeuge und zu einer einheitlichen und robusten Form der Datenhaltung führen. Wie die Grundlagen für diese Zukunftsperspektive geschaffen wurden, welche Überlegungen hinter der gewählten Vorgehensweise standen und welche Mittel hierbei zum Einsatz kamen, kann dem vorliegenden Dokument entnommen werden.

2 Aktueller Stand

Im Bereich der Arbeit mit im Mobilfunksektor benötigten Datenstrukturen ist bei Texas Instruments seit mehreren Jahren bereits ein funktionierendes System etabliert. Seit ebenfalls mehreren Jahren wird versucht, dieses System durch ein neues, schlüssigeres Konzept zu ersetzen. Welche Teile des bei Texas Instruments bestehenden Gesamtsystems dies betrifft, wie hier bisher vorgegangen wird, und welche Schwachstellen das existierende Konzept aufweist, wird im Folgenden eingehend betrachtet. Fachausdrücke aus dem Bereich des Mobilfunks werden dabei soweit möglich vermieden. Wo sie dennoch vorkommen, sind sie jeweils kurz erklärt. Weitere Informationen zu unbekannten Fachwörtern finden sich auch im Glossar und bei [12; 2; 3].

2.1 Abzubildende Datenstrukturen

Für mobile Kommunikation eingesetzte Software ist in aller Regel in der Form eines sogenannten „Protokollstacks“ implementiert. Dabei handelt es sich um eine Reihe von einzelnen Elementen, die zusammen die Übertragung von Information über ein Medium – in unserem Falle über die Luft – ermöglichen. In beiden an der Kommunikation beteiligten Instanzen ist dabei ein jeweils gleichartig aufgebauter Protokollstack implementiert. Zu übertragende Daten durchlaufen die einzelnen Teilbestandteile eines Protokollstacks und werden dabei für die Übertragung vorbereitet, beziehungsweise nach erfolgter Übertragung auf der anderen Seite der „Leitung“ wieder zugänglich gemacht. Die in einem Protokollstack enthaltenen Einheiten sind prinzipiell voneinander unabhängig gestaltet, um eine klare Gliederung der Software zu erreichen und einzelne Module mit nach außen hin gleichartiger Funktionalität unproblematisch gegeneinander austauschen zu können. Im Rahmen des Protokollstack-Konzepts ist die Definition verschiedener Arten von Schnittstellen notwendig. Die beiden in diesem Zusammenhang bedeutendsten und auch das vorliegende Projekt betreffenden Vertreter seien hier kurz vorgestellt.

2.1.1 Service Access Points

Um Kommunikation zwischen den in einem Protokollstack enthaltenen Modulen zu ermöglichen, sind für jedes Modul ein oder mehrere sogenannter „Service Access Points“ (SAP) definiert. Diese SAPs dienen als eindeutig festgelegte Schnittstellen zwischen den verschiedenen Modulen. Ein SAP gehört immer zu einem bestimmten Modul des Protokollstacks und stellt – der Name lässt es vermuten – für andere Module einen Dienst, oder „Service“, bereit.

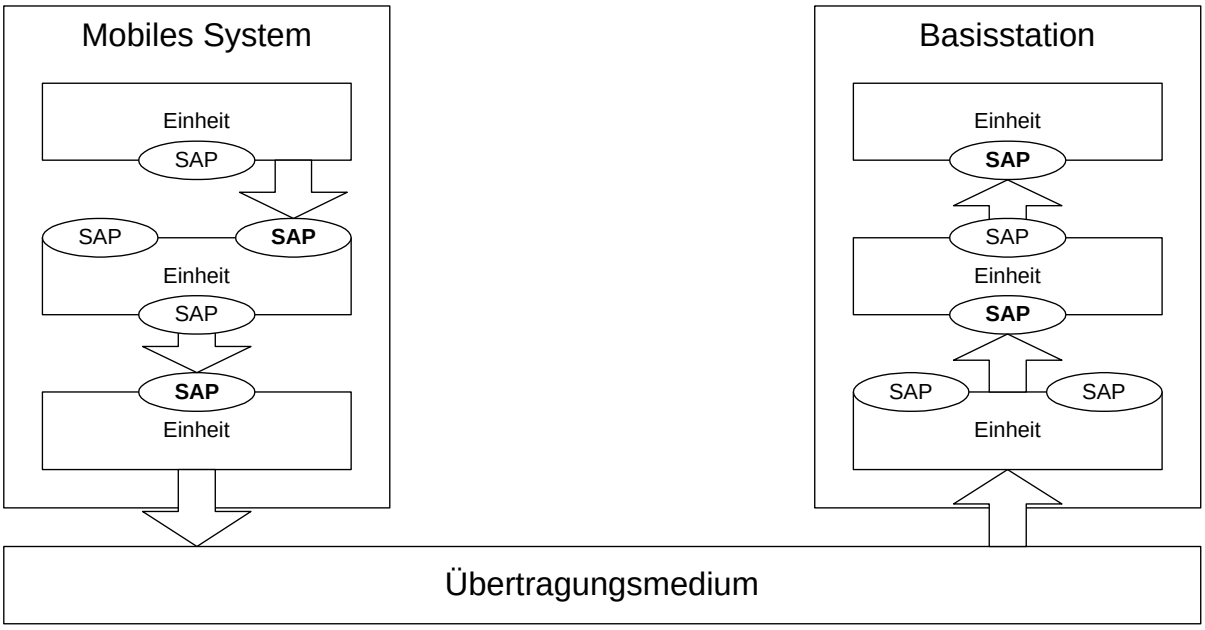

Abb. 2.1: Protokollstack

Die übliche Implementierung eines SAPs, die auch bei Texas Instruments verwirklicht ist, sieht das Versenden von sogenannten „Primitiven“ zur Kommunikation über die definierten Schnittstellen vor. Die Interface-Definition eines SAPs besteht also im Wesentlichen aus der Definition von Primitiven. Diese Primitiven sind kleine Pakete, die Daten oder Steuerinformationen enthalten, und die zwischen zwei Einheiten hin und her geschickt werden. Neben Primitiven können beispielsweise auch Funktionen und anderes in einem SAP definiert sein. Abbildung 2.1 zeigt die Übertragung von Information von einem Mobilsystem zur Basisstation. Die jeweils „unterste“ Einheit im Protokollstack ist dabei für die Steuerung des tatsächlichen physikalischen Sendens und Empfangens der Daten zuständig. Anwendungen, die Daten übertragen möchten, bauen demgegenüber in der Regel auf der „obersten“ Einheit des Protokollstacks auf. Die Struktur der in einem SAP enthaltenen Elemente kann, nur in groben Zügen von Spezifikationen vorgegeben, weitgehend frei bei der Implementierung der dazugehörenden Einheit festgelegt werden. Dem Anwender die Erstellung und Bearbeitung der Definition eines SAP in komfortabler Weise zu ermöglichen ist eine der Hauptaufgaben des hier vorgestellten Projektes.

2.1.2 Air Interface Messages

Auch die „Air Interface Messages“ (AIM) sind Beschreibungen von Schnittstellen. Es handelt sich bei den in einer solchen Schnittstelle definierten Datenstrukturen um direkte Nachrichten zwischen einem mobilen System und einer Gegenstelle im Netzwerk. Diese Nachrichten sind von

ETSI/3GPP[1] standardisiert und sind somit im Gegensatz zu den Datenstrukturen eines SAP in ihrer Struktur bereits vorgegeben. Ebenso wie für SAP-Strukturen soll es mit Hilfe des in diesem Dokument beschriebenen Projekts möglich sein, auf einfache Weise AIM-Strukturen zu erstellen und zu bearbeiten.

2.2 Bisherige Vorgehensweise bei Texas Instruments

Die Definitionen für SAP- und AIM-Schnittstellen sind in speziellen Dokumenten festgehalten. Diese Dokumente müssen maschinenlesbar sein, und es erscheint wohl zunächst sinnvoll, jeder Schnittstelle ein eigenes Dokument zuzuordnen. Darüber hinaus allerdings existieren viele Freiheiten, und es ist nicht auf Anhieb ersichtlich, welche Implementierung möglicherweise die meisten Vorzüge bieten mag. Welcher Weg hier bei Texas Instruments ursprünglich gewählt wurde und warum dies geschah soll in diesem Abschnitt zunächst dokumentiert werden, gefolgt von einer etwas eingehenderen Betrachtung der tatsächlichen Implementierung. Abschließend folgt dann eine Analyse der offensichtlichen Schwachstellen des bestehenden Systems, die zur Forderung nach einer Neuimplementierung geführt haben.

2.2.1 Historische Entwicklung

Als der erste Protokollstack für mobile Endgeräte bei Condat[2] entwickelt wurde, wurde großer Wert auf stets aktuelle und mit der ausgelieferten Implementierung konsistente Dokumentation gelegt. Um dies zu erreichen, sollte, wo immer möglich, die Dokumentation aus dem selben Dokument hervorgehen wie die für die Implementierung benötigte Information. Bei den Bestandteilen des Protokollstacks selbst wurde diese Maxime trotz einiger mehr oder minder erfolgreicher Versuche[3] nie wirklich verwirklicht. Bei den Strukturdefinitionen war dies naturgemäß leichter zu erreichen. Für die hier diskutierten Teilbereiche, die SAP- und AIM-Dokumente, wurde das Format von Microsoft Word als Grundlage festgelegt. Dies erschien sinnvoll, da so stets aktuelle und ohne zusätzlichen Aufwand menschenlesbare Dokumentation vorliegt. Gewisse Probleme bei der Weiterverarbeitung und vor allem in der Wartung der so entstandenen Dokumente traten freilich erst später zu Tage.

2.2.2 Gegenwärtiger Arbeitsablauf

Derzeit wird als Editor für SAP/AIM-Dokumente Microsoft Word verwendet. Als Grundlage für neue Dokumente findet eine Vorlagen-Datei Verwendung. Diese gibt die grundlegende Struktur

[1] ETSI (European Telecommunications Standards Institute) ist eine „non-profit“ Organisation, die sich die Entwicklung von Standards für die Telekommunikation zur Aufgabe gemacht hat. Nähres dazu unter [5]. 3GPP (3rd Generation Partnership Project) ist ein Zusammenschluss von Herstellern und Organisationen, die in der Entwicklung und Umsetzung solcher Standards, insbesondere der neuesten Generation (UMTS, ...), eine wesentliche Rolle spielen. Siehe auch [1].

[2] Heute Texas Instruments Berlin AG.

[3] Auch mit diesem Thema hat sich schon einmal ein Diplomand befasst.

des Dokuments für die zu beschreibende SAP/AIM-Schnittstelle vor. Außerdem werden darin einige Stilelemente für Untergruppen festgelegt, um ein einheitliches Layout zu gewährleisten und so eine automatische Weiterverarbeitung zu ermöglichen. Dies ist zugleich auch die einzige Hilfestellung für den Anwender – das syntaktische Format der einzelnen Datenelemente muss ihm bei der Eingabe bekannt sein.

Um die Elemente einer SAP/AIM-Definition zu erzeugen, wird das Dokument in einzelne Abschnitte gegliedert, die wiederum Tabellen und andere MS Word spezifische Objekte enthalten. Diese Objekte sind Teil der Syntaxdefinition und müssen somit zwingend verwendet werden, um dem nachfolgenden Parser die Erkennung der verschiedene Elemente zu ermöglichen.

Das so entstandene Dokument wird zugleich zur Schnittstellendokumentation genutzt und als Grundlage für die nachfolgenden Werkzeuge in der Toolkette[4]. Um letzterem Zwecke dienen zu können, muss das Dokument ins ASCII-Format umgewandelt werden, da das proprietäre Format von MS Word durch externe Programme nur sehr umständlich zu parsen ist.

Für AIM-Dokumente besteht die Hauptaufgabe bei der Eingabe darin, die von ETSI/3GPP spezifizierten Message-Definitionen in das bei Texas Instruments verwendete interne Format umzuwandeln. Bei SAP-Dokumenten dagegen müssen nach eigenem Ermessen Unterelemente und Tabellen angelegt werden. Um Zeit zu sparen, wird dies häufig durch Kopieren bereits existierender Elemente erledigt. Navigation innerhalb eines Dokuments wird durch Einfügen von Verweisen auf die Definition von Elementen, überall dort, wo diese verwendet werden, ermöglicht. Dies, wie auch die Pflege der entsprechenden Textmarken und Verweise, geschieht manuell.

Die syntaktische Richtigkeit eines Dokuments kann überprüft werden, wenn dieses fertiggestellt ist. Hierzu wird das Dokument zunächst mittels des Tools „doc2txt“ ins ASCII-Textformat umgewandelt. Die sich hieraus ergebende Datei wird von einem weiteren Tool, „xGen“, geparst. Das Ergebnis wird in den internen Formaten pdf (für SAPs) und mdf (für AIM) abgelegt. Aus

[4]Eine sogenannte „Toolkette“ besteht aus einer Reihe von Programmen, die schrittweise eine bestimmte Aufgabe erfüllen. Eine solche Toolkette kommt auch zur Verarbeitung der hier betrachteten Datenstrukturen bei Texas Instruments zum Einsatz.

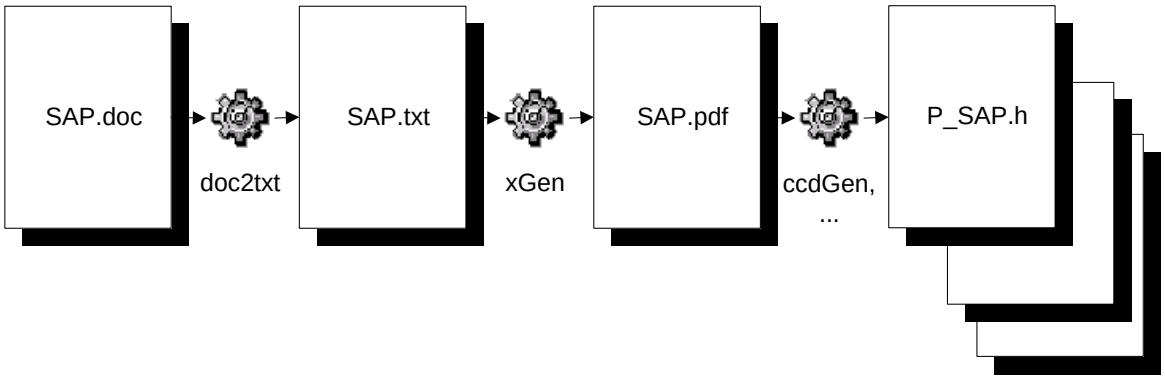

Abb. 2.2: Toolkette

diesen Formaten produziert unter anderem „ccdGen“, ein in der Toolkette nachfolgendes Werkzeug, die zur Protokollstack-Entwicklung gebrauchten C-Header- und Quellcode-Dateien. Abbildung 2.2 auf der Seite vorher zeigt diesen Ablauf exemplarisch an Hand eines SAP-Dokuments mit dem sinnigen Namen „SAP“.

2.2.3 SAP Format

Gegenwärtig werden Service Access Points bei Texas Instruments mit Hilfe von Microsoft Word Dokumenten definiert. Ein solches Dokument beinhaltet die Beschreibung des SAPs in einem festgelegten Format. Diese kann von der nachfolgenden Toolkette weiterverarbeitet und in Formate umgewandelt werden, die für die Entwicklung des Protokollstacks gebraucht werden. Im Wesentlichen handelt es sich hierbei um eine Reihe von C-Include-Dateien, welche die Definition des SAPs als C-Deklaration wiedergeben.

Um eine automatische Weiterverarbeitung der SAP-Dokumente zu ermöglichen, ist deren Struktur fest vorgegeben. Neben den bereits in der Vorlage enthaltenen Elementen, der Titelseite, dem Inhaltsverzeichnis, der Dokumentenkontrolle und einer Einführung, sind vier Hauptabschnitte möglich. Es handelt sich hierbei um jeweils einen Abschnitt für Konstanten („constant section“), Primitiven („primitives section“), Funktionen („functions section“) und Parameter („parameters section“), wobei der Abschnitt für Funktionen entfallen kann, wenn keine Funktionen im Dokument definiert sind. Alle anderen Abschnitte müssen vorhanden sein, können aber – zumindest im Falle von Parametern und Konstanten – gegebenenfalls leer sein. Als Konstanten sind alle globalen Konstanten zu finden, die in Primitiven oder Parametern verwendet werden. Der Parameter-Abschnitt enthält alle Parameter von anderen Parametern oder von Primitiven, und die im SAP verwendeten Primitiven sowie die bereitgestellten Funktionen selbst sind jeweils in den entsprechend benannten Abschnitten zu finden. Die Abschnitte müssen im Dokument in dieser Reihenfolge vorkommen. In den Abschnitten für Primitiven, Funktionen und Parametern sind jeweils Unterabschnitte möglich, die je eine einzelne Primitive, eine Funktion oder einen Parameter definieren. Konstanten befinden sich direkt in dem für sie bereitgestellten Abschnitt. Um die einzelnen Abschnitte und Unterabschnitte kenntlich zu machen, werden entsprechend formatierte Überschriften verwendet. Kommentare können in der üblichen C-Syntax in ein Dokument eingefügt werden, allerdings müssen die entsprechenden Zeichenketten, also „/*“ respektive „*/“ jeweils am Anfang beziehungsweise Ende einer Zeile stehen. Die derart kenntlich gemachten Kommentare werden dann von der nachfolgenden Toolkette übergangen.

Die Definition einer Primitive, einer Funktion, eines Parameters oder einer Gruppe von Konstanten beinhaltet verschiedene festgelegte Elemente. Dies können eine Beschreibung (Description), Pragma, Definition, Unterelemente (Elements), Werte (Values) oder die „History“[5] sein. Nicht alle diese Teile sind in allen Abschnitten verwendbar, und einige sind auch optionaler Natur. So müssen beispielsweise Beschreibung, Definition und Unterelemente in allen Abschnitten

[5] Verlauf der bisher an einem Element vorgenommenen Veränderungen.

vorkommen, Pragmas dagegen dürfen nur bei Konstanten vorkommen und sind auch dort nur optional. Einige wichtige Elemente werden im Folgenden besprochen.

Die Beschreibung beinhaltet eine in normalem Text verfasste Erklärung über Zweck und allgemeine Bedeutung des Abschnitts selbst und dessen Inhalts. Die Toolkette ignoriert diese Beschreibung, die nur zur Dokumentation innerhalb des Dokuments selbst gedacht ist.

Die Definition besteht aus einer Tabelle ähnlich der in Abbildung 2.3 dargestellten, in der die Eigenschaften eines Abschnittes näher beschrieben werden. Für Primitiven beispielsweise enthält diese unter anderem ein Element „ID“, welches im gesamten System eindeutig sein muss und mit dessen Hilfe auf eine Primitive zugegriffen werden kann. Des weiteren sind Felder für die Einheiten des Protokollstacks enthalten, welche die jeweilige Primitive versenden und empfangen. Außerdem ist in der Tabelle, welche die Definition der Primitive darstellt, noch ein Feld für den Namen der Primitive enthalten, der von den Werkzeugen der Toolkette verwendet werden soll.

In ähnlicher Weise sind auch die übrigen Abschnitte aufgebaut. Alle relevanten Daten befinden sich innerhalb von in MS Word erstellten Tabellen. Um die Navigation innerhalb eines Dokuments zu vereinfachen, besteht die Vorgabe, an bestimmten Stellen zusätzlich zu den bereits besprochenen Stilelementen manuell Verknüpfungen zwischen Elementen des Dokuments einzufügen. Dies betrifft im Wesentlichen die Elemente im Abschnitt der Parameter.

2.2.4 AIM Format

Auch „Air Interface Messages“ werden bei Texas Instruments im MS Word-Format abgebildet. In den entsprechenden Dokumenten ist beschrieben, wie die Daten je einer Gruppe von AIMs organisiert sind und wie von den Einheiten des Protokollstacks darauf zugegriffen werden kann. Im Gegensatz zu SAPs, die in der Regel auf Bytes als der grundlegenden Einheit aufbauen, sind die zugehörigen Datenstrukturen hier auf Bit-Ebene strukturiert. Die Verarbeitung der Dokumente geschieht analog zur Verarbeitung von SAP-Dokumenten. Auch hier werden als wichtigstes Ergebnis C-Header-Dateien ausgegeben.

Ein AIM-Dokument ist ebenfalls in verschiedene Abschnitte gegliedert, die in ihrer Reihenfolge feststehen:

- Konstanten („Constants“);
- Typen („Types“);
- Nachrichten („Messages“);

short name	ID	direction
FUN_READ_REQ	0x60042023	XX -> FUN

Abb. 2.3: SAP-Element als MS Word-Tabelle

- zusammengesetzte Elemente („Structured Elements“);
- einfache Elemente („Basic Elements“).

Die Abschnitte für Konstanten und strukturierte Elemente sind optional. Neu gegenüber SAP-Dokumenten ist hier im wesentlichen der Abschnitt für „Typen“, in dem die verwendeten Kodierungsarten für die beschriebenen Nachrichten aufgelistet sind. Weiterhin existieren statt eines gemeinsamen Abschnitts für alle „Parameter“ bei AIM-Dokumenten je ein Abschnitt für einfache Elemente und einer für jene, die aus einfachen Elementen und anderen zusammengesetzten Elementen zusammengesetzt sind.

Die Unterteilung der Abschnitte in Unterabschnitte geschieht wiederum analog der für SAP-Dokumente beschriebenen Vorgehensweise. Als Beispiel sei hier der Abschnitt für Nachrichten herausgegriffen. Dieser ist in die vier Unterabschnitte Beschreibung („Description“), Definition („Definition“), Elemente („Elements“) und „History“ gegliedert. Beschreibung und History entsprechen den gleichnamigen Elementen in einem SAP-Dokument. Die Definition beinhaltet in tabellarischer Form den Namen einer Nachricht sowie den Typ der Nachricht und die Richtung in die sie gesendet wird. Optional ist es auch möglich, mittels eines Versionsfeldes zu definieren, für welche Versionen des Codes die Nachricht gebaut werden soll. Des weiteren kann eine Nachricht in einem anderen Dokument definiert werden und im aktuellen Dokument nur als Verweis bestehen. Dies ist dann im ebenfalls optionalen Element „Link“ festgehalten. Hier geht das AIM-Format klar über das für SAPs hinaus. Aus Abbildung 2.4 ist zu erkennen, dass der Namen in einer kurzen und einer langen Form angegeben wird. Die lange Form dient hierbei ausschließlich der Dokumentation. Die Information über die Art der Nachricht ist in der „ID“ enthalten und das Feld „direction“ beinhaltet die Senderichtung.

Der Unterabschnitt der Elemente enthält, ebenfalls in tabellarischer Form, die Elemente der beschriebenen Nachricht, sofern es sich dabei nicht um einen Verweis auf die Definition einer Nachricht in einem anderen Dokument handelt. Da sich hierbei, wie auch in den anderen Abschnitten eines SAP- oder AIM-Dokuments, die prinzipielle Vorgehensweise wiederholt, soll im Rahmen dieser Arbeit nicht weiter darauf eingegangen werden. Detailliertere Angaben zum Aufbau von SAP- und AIM-Dokumenten finden sich bei [9] beziehungsweise [10] und [8].

2.2.5 Probleme mit Editor und Datenformat

Ein Textverarbeitungsprogramm wie Microsoft Word ist nicht zur Eingabe und Pflege komplexer strukturierter Daten vorgesehen. Dies wirft von vorn herein einige prinzipielle Probleme auf.

long name	short name	ID	direction
activate fun mode	activate_fun	0x70042023	downlink

Abb. 2.4: AIM-Element als MS Word-Tabelle

So kann mit Hilfe einer Textverarbeitung zwar sehr gut das Aussehen des bearbeiteten Textes bestimmt werden, zu Inhalt und Struktur des Textes kann das Programm dagegen keine relevanten Hilfen anbieten. Das hat zur Folge, dass der Benutzer keinerlei syntaktische oder logische Hinweise oder Hilfestellungen erhält. Es findet keinerlei Syntax- oder Integritätsüberprüfung des eingegebenen „Textes“ statt, und es können keine Vorschläge für an der gegenwärtig editierten Stelle mögliche und erlaubte Elemente gegeben werden.

Hinzu kommen einige Probleme, die spezifisch für MS Word auftreten. Neben weithin bekannten Unannehmlichkeiten wie der Programmbedienung als solcher und häufig auftretenden Problemen aufgrund untereinander inkompatibler Programm- und Sprachversionen fällt hier vor allem das Dateiformat ins Auge. Es handelt sich hierbei um ein proprietäres Format, was die Weiterverarbeitung durch fremde Programme unnötig erschwert. Zudem ist dieses Format teilweise binär, so dass Dateien vor dem Parsen zunächst in ASCII-Text umgewandelt werden müssen. Unangenehm ist auch die resultierende Unmöglichkeit, beschädigte Dokumente mit einfachen Mitteln zu reparieren. Bereits kleinste Fehler im Dokument machen ein Öffnen unmöglich. Auch das Zusammenführen zweier unterschiedlicher Versionen eines Dokuments ist mit Standardwerkzeugen nicht möglich. Hier ist in aller Regel Handarbeit durch den Benutzer notwendig, was gerade im Rahmen des bei Texas Instruments bestehenden automatisierten Versionskontrollsystems ärgerlich, unnötig und fehleranfällig ist.

Da die für die Repräsentation der Daten gewählte Form sehr viel mehr Wert auf Lesbarkeit und Layout legt, als auf sinnvolle Datenhaltung, werden viele typische Stilelemente einer Textverarbeitung zur Strukturierung verwendet. Diese dienen allerdings nicht nur der optischen Verschönerung, sondern gleichzeitig auch der Ordnung der Daten selbst. Daher müssen die weiterverarbeitenden Werkzeuge auch Stilelemente wie Tabellen, Abschnittsnummerierung und Überschriften beim Auslesen der Daten berücksichtigen. Dies führt zu einer Vermischung von Syntax und Design, was wiederum zu einer ganzen Fülle weiterer Probleme führt. Das Zusammenführen verschiedener Versionen eines Dokuments wird zum Beispiel weiter erschwert. Es muss manuell bestimmt werden, welche Strukturelemente wo zusammengeführt werden müssen und wie dies in der Textverarbeitung zu erfolgen hat. Für den Benutzer ist aber auch schon ohne solcherlei Probleme oft nicht ersichtlich, in welcher Weise unterschiedliche Elemente der Textstrukturierung das Ergebnis beeinflussen und wo die Ursachen von Fehlern in der Weiterverarbeitung zu suchen sind. Letzteres wird dadurch verschärft, dass weiterverarbeitende Systeme wiederum nichts über den eigentlichen Sinn der Layout-Elemente und deren Zuordnung zu dem ihnen vorliegenden Code wissen und daher nicht in der Lage sind, aussagekräftige Fehlermeldungen über Ort und Ursache eines Problems zu produzieren. Da dem Benutzer also auch hier keinerlei Hilfestellung gegeben wird, werden sehr häufig bei der Bearbeitung von Dokumenten einfach bereits existierende Strukturelemente kopiert und entsprechend angepasst.

Auch bei der Verschachtelung von Strukturen oder ihrer Verwendung in verschiedenen Dokumenten müssen häufig Elemente kopiert werden, da nicht die Möglichkeit besteht, mit Referenzen zu arbeiten. Dies führt automatisch zu redundanter Datenhaltung, mit allen weithin bekannten

und gefürchteten Konsistenz- und Wartungsproblemen. Auch daraus resultierende Probleme werden wiederum erst bei der Weiterverarbeitung der Dokumente erkannt und müssen durch Vergleichen und Ausprobieren beseitigt werden. Des weiteren werden Dokumente unnötig aufgebläht und es ist nicht ersichtlich, wo welche Elemente referenziert und verwendet werden.

Bei Verwendung von im selben Dokument definierten Elementen ist es zwar möglich, auf die Definition mittels einer Verknüpfung hinzuweisen, aber diese muss ebenfalls in Handarbeit erstellt werden. Da zum Verknüpfen normalerweise die Kapitelnummern herangezogen werden, ist auch beim Einfügen und Löschen von Elementen die manuelle Nachführung der bereits existierenden Verknüpfungen notwendig.

All dies führt dazu, dass sogar die ursprüngliche Rechtfertigung für die Verwendung von MS Word als Werkzeug zur Datenbearbeitung nicht mehr stichhaltig ist: aus diesem Prozess entstehende Dokumente mögen wohl optisch ansprechend aussehen. Zur Dokumentation taugen sie jedoch nur noch sehr bedingt. Ist schon die Übersicht über ein einzelnes SAP/AIM-Dokument kaum noch gegeben, so bietet das gewählte Format spätestens dann überhaupt keine Hilfe mehr, wenn die Berücksichtigung mehrer Dokumente zugleich notwendig wird. Es ist dem Benutzer nicht möglich, sich in und zwischen verschiedenen Dokumenten zu bewegen, ohne auf für diesen Zweck inadäquate Standard-Suchfunktionen des verwendeten Textverarbeitungsprogramms zurückzugreifen.

Zu den bisher besprochenen, aus den grundsätzlichen Festlegungen für das Format von SAP- und AIM-Dokumenten resultierenden Problemen gesellen sich noch einige weitere, die Folge von spezifischen Details bei der Ordnung von Daten innerhalb eines Dokuments sind. Hier nur drei der augenfälligsten Punkte:

- Unterelemente von Strukturen in SAPs haben keinen Namen;
- in Strukturen, die in einem SAP-Dokument definiert sind, ist es erlaubt, direkt einfache Elemente zu definieren, was leicht dazu führen kann, dass die mehrfache Vergabe eines Namens an verschiedene Element nicht erkannt wird;
- die Formate von SAP- und AIM-Dokumenten unterscheiden sich im Aufbau stark voneinander – der Abschnitt für Parameter in einem SAP-Dokument beispielsweise dient dem selben Zweck wie in einem AIM-Dokument die strukturierten und die einfachen Elemente zusammen.

Das besprochene Format ist aus den diversen hier aufgezeigten Gründen sowohl für die Eingabe und Wartung von Daten als auch für deren Weiterverarbeitung und Analyse, sowohl in automatisierter Form als auch durch menschliche Benutzer, nur wenig geeignet. Die vorliegende Vermischung von Information und Layout führt zu einer unangemessenen Verkomplizierung der notwendigen Arbeitabläufe. Dies hat sich so auch in der praktischen Arbeit mit dem System erwiesen. Insbesondere Wartung, Fehlersuche und Zusammenführung von Dokumenten erfordern ein völlig unangemessenes Maß an zeitlichem Aufwand und sind Grund ständigen Ärgernisses seitens der damit betrauten Software-Entwickler.

3 Verwendete Techniken

In diesem Kapitel werden die für die im Rahmen der vorliegenden Diplomarbeit erfolgte Neuimplementierung verwendeten Techniken und Verfahren vorgestellt sowie deren Anwendung beschrieben. Für das Verständnis der folgenden Kapitel wird lediglich ein grundsätzliches Basiswissen über die Programmiersprache Java und deren Anwendung – beschrieben zum Beispiel in [20; 24] – vorausgesetzt. Alle weiteren im Rahmen des hier vorgestellten Projekts verwendeten Techniken werden in diesem Kapitel oder, soweit nur jeweils dort von Bedeutung, am Ort ihres Auftauchens erklärt.

Das hier vermittelte Wissen hilft zugleich auch beim Verständnis dafür, warum eben die vorgestellten Methoden für die Implementierung ausgewählt wurden. Über die Gründe dieser Auswahl sowie über im Vorfeld diskutierte mögliche Alternativen informiert dann ausführlich ein nachfolgendes Kapitel.

3.1 XML

XML[1] ist eine moderne Sprache, die eine unkomplizierte und robuste Datenhaltung ermöglicht. Eine besondere Stärke von XML liegt in der wohldefinierten Struktur, was ein auf XML basierendes Datenformat zum Austausch zwischen verschiedenen voneinander unabhängigen Einheiten geeignet macht.

3.1.1 XML als Datenformat

Wesentliche Elemente der in diesem Abschnitt gemachten Ausführungen sind inhaltlich aus [21] übernommen. Dort ist auch eine detaillierte Beschreibung der für die Arbeit mit XML unter Java zum Einsatz kommenden Techniken und Werkzeuge zu finden. Weitere Informationen zu XML sind in [18; 22] oder online bei [26] zu finden.

XML gehört zur Familie der Textauszeichnungssprachen[2], wie auch HTML oder SGML. Diese Sprachen zeichnen sich dadurch aus, dass sie Zusatzinformationen zu den eigentlichen Daten schreiben, die genutzt werden, um die Daten zu gruppieren. Auf diese Weise können zudem auch bestimmte Teile der verwalteten Daten für spezielle Aufgaben gekennzeichnet werden. Üblicher-

[1] eXtensible Markup Language.

[2] Ein schönes Beispiel für die „Eindeutschung“ eines typischen Computer-Fachausdrucks, der in seiner ursprünglichen Form für die meisten sehr viel vertrauter klingen dürfte als sein deutsches Pendant: „Markup Language“.

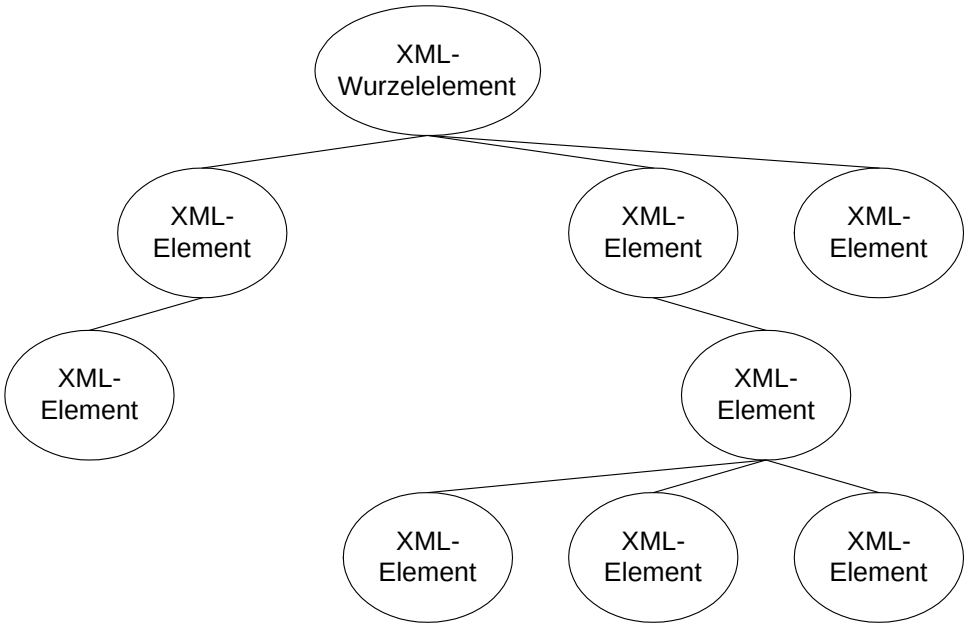

Abb. 3.1: Hierarchische Struktur eines XML-Dokuments

weise werden hierfür spezielle Kennzeichnungen, die sogenannten „Tags“[3] verwendet, welche die jeweils zu einer Untergruppe gehörenden Daten umschließen.

Einige bekannte Textauszeichnungssprachen, beispielsweise HTML oder WML, stellen auf diese Weise im Wesentlichen Mittel zur Verfügung, um die optische Präsentation der gespeicherten Daten zu definieren. Hierzu existieren Tags für Abschnitte, zu verwendende Zeichensätze, Tabellenformate und viele andere Stilelemente. XML dagegen definiert nur einige wenige Tags und grundlegende Regeln, die zur Definition einer eigenen Grammatik beziehungsweise eines eigenen Datenformats genutzt werden können. Hierzu wird es dem Benutzer ermöglicht, seine eigenen benutzerdefinierten Tags anzulegen, um den Inhalt seiner jeweiligen Daten in geeigneter Weise abzubilden. Diese Vorgehensweise ist im folgenden Beispiel gut zu erkennen, das die Bestellung von 10 Kieselsteinen durch einen Kunden „Hugo Müller“ festhält.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Bestellung>
  <Kunde id="k17">Hugo Müller</Kunde>
  <Produkt>
    <Name>Kieselsteine</Name>
    <Anzahl>10</Anzahl>
    <Preis Einheit="EUR">100</Preis>
  </Produkt>
```

[3] Dieser Begriff wird trotz der im Deutschen leicht misszuverstehenden Bedeutung nachfolgend eingesetzt, da keine dem Autor bekannte vernünftige und sinnhaltige Übersetzung ins Deutsche existiert.

```
</Bestellung>
```

Regeln und Struktur der selbstdefinierten Textauszeichnungssprache werden dabei in zusätzlichen Dateien festgehalten.

Bei der Entwicklung von XML stand zunächst die Anwendung im Internet im Vordergrund. XML sollte dort zum Einsatz kommen, wo bereits SGML und HTML erfolgreich waren, es sollte Texte und andere Daten in für Menschen anschaulicher Form im Netz präsentieren. Sehr entgegen diesem hehren Ansatz wurde und wird XML in der Praxis allerdings vorwiegend zur Definition von Datenformaten eingesetzt. Und auf diesem Gebiet ist die Sprache außerordentlich erfolgreich. Wie kam es zu diesem unerwarteten Werdegang?

In den letzten Jahrzehnten wurden eine Fülle von Datenformaten für die unterschiedlichsten Zwecke entwickelt. Da viele dieser Formate von einzelnen Unternehmen oder Gruppierungen ins Leben gerufen wurden, sind sie oft nur schwer zugänglich. Probleme können auftreten aufgrund unzureichender oder gar nicht vorhandener Dokumentation, aufgrund unpräziser oder unvollständiger Definition, oder aufgrund von Unzulänglichkeiten des Formats selbst. Häufig unterscheiden sich auch die von verschiedenen Implementierungen erzeugten Daten im Format in subtiler Weise voneinander, so dass selbst die Austauschbarkeit innerhalb eines einzigen Datenformats nicht immer gewährleistet ist.

Diese bekannten Problematiken verleiten dazu, bei der Neuentwicklung eines Datenformats das einfachst möglich erscheinende Format zu wählen. Hier bietet sich beispielsweise ein durch Tabulatoren separiertes zeilenweises Format an, wie es das folgende Beispiel demonstriert:

```
Maier       Franz       01. 01. 1970
Müller      Fritz       02. 03. 1966
Fischer     Heike       05. 05. 1975
```

Hier ist klar ersichtlich, welcher Eintrag welcher Spalte zugeordnet ist. Jedoch ist dies nicht aus der Struktur der Tabelle zu erkennen, sondern nur anhand der darin enthaltenen Daten. Nun ist es natürlich möglich, einfach irgendwo die Definition dafür festzuhalten, welche Spalte welchen Eintrag zu beinhalten hat. Allerdings muss dies dann auch von jedem Programm, welches dieses Datenformat liest und schreibt, beachtet und eingehalten werden. Tritt an einer Stelle ein Fehler auf, werden beispielsweise Vor- und Nachnamen vertauscht, bleibt dieser bei der automatisierten Weiterverarbeitung sehr wahrscheinlich unbemerkt. Läge die Namensliste dagegen im XML-Format vor, wären solche Unklarheiten und Probleme von vorn herein ausgeschlossen:

```
<Namensverzeichnis>
  <Person>
    <Nachname>Maier</Nachname>
    <Vorname>Franz</Vorname>
```

```
    <Geburtsdatum>
      <Tag>01</Tag>
      <Monat>01</Monat>
      <Jahr>1970</Jahr>
    </Geburtsdatum>
  </Person>
  <Person>
    <Nachname>Müller</Nachname>
    <Vorname>Fritz</Vorname>
    <Geburtsdatum>
      <Tag>02</Tag>
      <Monat>03</Monat>
      <Jahr>1966</Jahr>
    </Geburtsdatum>
  </Person>
  <Person>
    <Nachname>Fischer</Nachname>
    <Vorname>Heike</Vorname>
    <Geburtsdatum>
      <Tag>05</Tag>
      <Monat>05</Monat>
      <Jahr>1975</Jahr>
    </Geburtsdatum>
  </Person>
</Namensverzeichnis>
```

Auch das Format, in dem das Datum abgespeichert wird, spielt nun keine Rolle mehr. Unabhängig von der Reihenfolge der Parameter können immer die korrekten Daten ausgelesen werden. Wird es notwendig, Teilelemente zu ergänzen, in diesem Beispiel möglicherweise Adressen oder Geburtsort, so ist in der tabularen Form eine Umdefinition des Formats notwendig, was eine entsprechende Anpassung aller verwendenden Programme notwendig macht. In XML ist es dagegen unproblematisch, ein neues Element hinzuzufügen, das von Programmen, die dieses noch nicht kennen, gegebenenfalls einfach ignoriert wird.

In ähnlicher Weise werden an vielen Stellen die Vorteile von XML deutlich. Wird XML zur Definition eines Datenformats eingesetzt, so ist dieses bereits grundsätzlich robust, einfach zu erweitern, klar gegliedert und erlaubt eine Reihe nützlicher Strukturelemente. In der Definition von XML sind bereits optionale, leere und mehrfach vorkommende Daten-Elemente berücksichtigt. Gerade bei umfangreicheren und komplexen Formaten können diese Vorzüge voll ausgespielt werden.

Auch beim Parsen selbst bietet XML Vorzüge. So ist es lediglich notwendig, für ein selbstentwickeltes Programm einen der zahlreichen Standard-XML-Parser auszuwählen, die auch als freie Software für fast jede Softwareplattform erhältlich sind. Dieser sorgt dann für die ordnungsgemäße Auflösung von Escape-Sequencen, Zeilenumbrüchen, Kodierungsformaten[4] und vielen weiteren Dingen, bis hin zu solchen Details wie der Reihenfolge der Bits und Bytes[5]. Es ist nicht – wie bei der Verwendung einer herkömmlichen Grammatik zur Definition des Datenformats – notwendig, zuerst aus der vorhandenen Grammatik einen Parser zu generieren, der dann zum Parsen von Dokumenten genutzt werden kann. Vielmehr wird der Parser zur Laufzeit anhand der Grammatik – zumeist in Form einer DTD oder eines XML-Schemas – automatisch konfiguriert und kann damit Daten verarbeiten, die dieser Grammatik entsprechen. Soll eine XML-Datei ohne Kontrolle durch eine besondere Grammatik eingelesen werden, so wird dem Parser dies einfach mitgeteilt. Er orientiert sich daraufhin allein an der bekannten Syntax der XML-Textauszeichnungssprache selbst.

Alle diese Eigenschaften prädestinieren die Sprache XML als Standardformat zum Austausch von Daten. Ein in XML definiertes Datenformat macht die darin abgelegten Daten hochgradig portabel, da jedem beliebigen Programm auf einfache und konsistente Weise der Zugriff darauf möglich ist. Dabei ist es für den Benutzer nicht notwendig, sich mit der Struktur der Datenhaltung zu beschäftigen, was es ihm erlaubt, sich voll auf die Daten selbst zu konzentrieren.

Die hier aufgezeigten Vorteile von XML gegenüber proprietären Datenformaten waren und sind Hauptargument für den Siegeszug von XML in den Bereichen der Datenhaltung und des Datenaustauschs. Neue Datenformate werden heute häufig in XML formuliert, und beinahe jedes größere Projekt beinhaltet in der einen oder anderen Form auch das Parsen und die Verarbeitung von XML-Daten.

3.1.2 Struktureller Aufbau

Nachdem die Grundsätze der Sprache XML bereits angesprochen wurden sowie deren Vorteile und Annehmlichkeiten gegenüber anderen Konzepten, soll nun näher auf die eigentliche Struktur der Sprache und die Verwendung der in ihr enthaltenen Elemente eingegangen werden. Anhand der bisherigen Beispiele lassen sich schon anschaulich die in der Sprache vorkommenden Strukturelemente ausmachen. Diese, die sogenannten Tags, sind jeweils von spitzen Klammern („<“, „>“) umgeben. Die zugehörigen Daten stehen dabei immer zwischen einem einleitenden und dem korrespondierenden schließenden Tag. Dieses gleicht dem Einleitenden und ist durch einen vorangestellten Schrägstrich gekennzeichnet. Ein vollständiges Element besteht aus einem einleitenden Tag, einem gleichnamigen schließenden Tag und allem, was dazwischen steht. Dies können Daten sein, oder andere Elemente. In einem einleitenden Tag können auch Attribute definiert werden. Diese bestehen aus einem Name-Wert-Tupel in der Form name="Wert", wie es im Beispiel mit den Kieselsteinen bei der Einheit für den Preis zu sehen ist. Im Gegensatz zu HTML ist zu

[4] Unicode, ASCII, Latin-1, SJIS,...
[5] Big endian, little endian

jedem einleitenden Tag zwingend ein entsprechendes schließendes Tag erforderlich. Kommentare sind ebenfalls von spitzen Klammern umschlossen und werden durch die Zeichenkombination „!−−“ eingeleitet und mit „−−“ abgeschlossen. Sie dürfen an beliebiger Stelle innerhalb eines XML-Dokuments auftauchen und bis auf die einen Kommentar einleitenden und abschließenden Zeichenkombinationen alle im Zeichensatz enthaltenden Zeichen beinhalten.

Jedes XML-Dokument basiert auf einem einzelnen Element, dem Wurzel-Element. Alle anderen Elemente und Daten sind darin enthalten. Lediglich Kommentare und besondere Steueranweisungen sind von dieser Regel ausgenommen. Das Wurzel-Element etwa für das vorangegangene Beispiel ist das Element „<Namensverzeichnis>“. XML-Elemente dürfen sich nicht gegenseitig überlappen. Daher wäre das folgende Fragment kein legaler XML-Code und würde von einem XML-Parser nicht akzeptiert werden:

```
<Name>
  <Nachname>Maier</Nachname>
  <Vorname>Franz</Vorname>
<Straße>
</Name>
  Hausbergstr. 10
</Straße>
```

Neben diesen grundlegenden Strukturen sind bei XML noch einige weitere Dinge zu beachten. So wird bei Element- und Attributnamen zwischen Groß- und Kleinschreibung unterschieden. Diese muss also beispielsweise bei einleitenden und abschließenden Tags übereinstimmen. Attribute müssen zwingend von Anführungszeichen eingeschlossene Werte haben, leere Attribute sind nicht zulässig.

Die hier vorgestellten Eigenschaften und Strukturen von XML sind Teil der vom W3C[6] herausgegebenen Spezifikation von XML. Sind alle Vorgaben dieser Spezifikation erfüllt, so wird ein XML-Dokument als wohlgeformt („well formed“) bezeichnet. Werden zudem noch die optional in einem zusätzlichen Dokument festgehaltenen inhaltlichen Vorgaben erfüllt, wird es als gültig („valid“) bezeichnet. Zwei mögliche Formate für diese Definitionsdokumente, die tatsächlich nichts anderes als eine selbstdefinierte Grammatik enthalten, werden in den beiden folgenden Abschnitten vorgestellt.

Die meisten XML-Dokumente beginnen mit einer XML-Deklaration. Diese ist nicht unbedingt erforderlich, beinhaltet aber einige wichtige Informationen. Dies sind im einzelnen die verwendete Version der XML-Sprache, die für das Dokument verwendete Kodierungsart sowie eine Angabe darüber, ob das Dokument sich auf irgendwelche anderen Dokumente bezieht. Alle drei Angaben sind optional und werden wie folgt in Form von Attributen dargestellt:

[6] Das W3C (World Wide Web Consortium) entwickelt technische Standards des World Wide Web und koordiniert ihre Weiterentwicklung und Verbreitung. Siehe auch [7].

<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>

In der Regel werden XML-Dateien in gewöhnlichem ASCII-Format gespeichert. Es sind aber auch universellere Formate vorgesehen, wie beispielsweise Unicode.

Alias-Namen werden mit dem Schlüsselwort „!ENTITY“ definiert. Dies kann beispielsweise wie folgt aussehen:

<!ENTITY kn "Konstanz">

Im zugehörigen XML-Dokument wird damit jedes Vorkommen der Zeichenkette „&k“ durch das Wort "Konstanz" ersetzt. Für spezielle Zeichen existieren bereits einige vordefinierte Alias-Namen. Diese sind Abbildung 3.2 zu entnehmen.

Das XML-Format bietet außer dem hier Besprochenen noch einiges mehr an Funktionalität. Die bis hierher angesprochenen Elemente genügen allerdings zum weiteren Verständnis. Interessierte seien an die weiterführende Literatur ([18; 22; 26; 21]) verwiesen. Dort werden auch erweitere Konzepte wie Namensräume und Verarbeitungsanweisungen („processing instructions“) behandelt.

Im Folgenden werden die beiden bekanntesten Formate besprochen, die zur Definition der Struktur eines XML-Dokuments genutzt werden können. Es handelt sich hierbei um das Format der „Document Type Definition“ (DTD), und das Format der XML-Schemata. Mit beiden Formaten können die Elemente definiert werden, die zur Repräsentation der eigentlichen Daten genutzt werden sollen, also alle Bestandteile eines Dokuments außer den Daten selbst. Mit Hilfe von XML-Schemata können zudem einige Randbedingungen für die in einem Dokument enthaltenen Daten definiert werden. Auch zu diesen beiden Konzepten finden sich in den angesprochenen Quellen weitergehende Informationen.

3.1.3 Document Type Definition

In einer DTD werden die Elemente definiert, die in den dazugehörigen XML-Dokumenten auftauchen dürfen. Weiterhin werden deren Struktur und erlaubte Reihenfolge, Verschachtelungen, Kardinalitäten und andere Strukturfragen festgelegt. Im wesentlichen ähnelt eine DTD also inhaltlich wie dem Sinn gemäß der Definition einer herkömmlichen Grammatik. Dies kommt auch

Platzhalter	dargestelltes Zeichen
<	<
>	>
"	“
'	’
&	&

Abb. 3.2: Vordefinierte Platzhalter in XML-Dokumenten

in der Wahl der in einer DTD zu verwendenden Elemente zum Ausdruck. Diese lehnen sich eng an aus dem BNF-Format[7] lange bekannte Elemente an. Zur Verdeutlichung hier als Beispiel ein Auszug aus einer möglichen Definition für die XML-Struktur einer SAP-Definition:

```
<!-- Eine einfache DTD für ein SAP-Dokument (dies ist ein Kommentar :) -->
<!ELEMENT sap (prim*) >
<!ELEMENT prim (prim_desc, prim_name, prim_opc, prim_dir, prim_elem, history) >
<!ELEMENT prim_elem (element*) >
<!ELEMENT element ((long_name|short_name), ref, type) >
<!ELEMENT history (his_entry+) >
<!ELEMENT his_entry (date, initials, comment?) >
<!ELEMENT prim_desc (#PCDATA) >
<!ELEMENT prim_name (#PCDATA) >
<!ELEMENT prim_opc (#PCDATA) >
<!ELEMENT prim_dir (#PCDATA) >
<!ELEMENT long_name (#PCDATA) >
<!ELEMENT short_name (#PCDATA) >
<!ELEMENT ref (#PCDATA) >
<!ELEMENT type (#PCDATA) >
<!ELEMENT date (#PCDATA) >
<!ELEMENT initials (#PCDATA) >
<!ELEMENT comment (#PCDATA) >
```

Die Syntax für Kommentare ist gleich der bei XML-Dokumenten verwendeten. Die Definition eines Elements der durch die DTD definierten Grammatik wird durch das Schlüsselwort „!ELEMENT“ eingeleitet und ist von spitzen Klammern umgeben. Dem Schlüsselwort folgt der Name des neu definierten Elements, an den sich die eigentliche Definition anschließt, eingeschlossen in runde Klammern. Diese Definition kann aus verschiedenen Elementen bestehen, welche die Unterelemente des zu definierenden Elements bestimmen sowie deren strukturbezogene Eigenschaften. Dies sind im wesentlichen

- die Namen der enthaltenen Unterelemente, durch Kommata voneinander getrennt, soweit sie nicht bereits durch andere Kontrollzeichen („?“, „|“) eingegrenzt sind;
- „?“ hinter optionalen Elementen,
- „*“ hinter Elementen, die an dieser Stelle in beliebiger Anzahl vorkommen dürfen;
- „+“ hinter Elementen, die zumindest einmal vorkommen müssen, deren Anzahl aber nach oben offen ist;

[7] BNF, Backus-Naur-Form, ist ein Format zur Definition einer Grammatik. Es erlaubt Kardinalitäten, Verschachtelungen und die Auswahl zwischen verschiedenen Unterelementen.

- „|“ zwischen Elementen, von denen wahlweise genau eines an dieser Stelle vorkommen darf, wobei die hiervon betroffenen Elemente von runden Klammern eingeschlossen werden;
- das Schlüsselwort „#PCDATA“[8] als einziges Element der Definition eines Elements, um anzuzeigen, dass dieses Element Daten enthält.

Gruppen von Elementen werden in runde Klammern eingeschlossen. Diese Gruppierungen können dann wie ganz normale Einzelelemente betrachtet und damit also beispielsweise durch Anfügen eines Fragezeichens als optional definiert werden.

Die Angabe von Attributen für ein Element wird ebenfalls durch ein eigenes Schlüsselwort eingeleitet und ist von spitzen Klammern umgeben. Das Schlüsselwort lautet „!ATTLIST“, ein entsprechender Eintrag in unserer SAP-DTD könnte zum Beispiel wie folgt aussehen:

<!ATTLIST sap sapID CDATA #REQUIRED>

An diesem Beispiel werden die Elemente der Definition einer Liste von Attributen deutlich. Es handelt sich hierbei der Reihe nach um

- die Angabe des Elements, dem die aktuelle Liste von Attributen angegliedert werden soll;
- den Namen des Attributes selbst;
- den Datentyp für den Wert des Attributes, wobei CDATA[9] signalisiert, dass es sich hierbei um normalen Text handelt;
- eine Angabe, ob das aktuelle Attribut optional („#IMPLIED“) oder zwingend erforderlich („#REQUIRED“) ist.

In einer Attributliste können beliebig viele Attribute enthalten sein. Hierzu werden die Definitionen der einzelnen Attribute einfach durch Leerzeichen getrennt hintereinandergeschrieben. Mit der optionalen Angabe von vorgegebenen Werten für ein Attribut sowie einer Standardauswahl aus diesen Werten ist eine sehr begrenzte Form der Überprüfung von Eingabedaten möglich. Ein solcher Eintrag kann beispielsweise so aussehen:

<!ATTLIST type typeVal CDATA (BIN|HEX|DEC) “HEX”>

was für das Attribut „typeVal“ des Elements „type“ die drei Werte „BIN“, „HEX“ und „DEC“ erlauben würde und als Standardwert „HEX“ vorgibt.

Ein XML-Dokument ist gültig, wenn es nur die in der zugehörigen DTD definierten Elemente mit ihren Attributen enthält und auch alle sonstigen in der DTD festgelegten Bedingungen erfüllt. Die hier vorgestellten Möglichkeiten zeigen nur einen Teil des DTD-Formats auf. Für weitere Informationen und eingehendere Betrachtungen sei wiederum auf die genannte Literatur verwiesen.

[8] „parseable character data“
[9] „character data“

3.1.4 XML-Schemata

Ein XML-Schema dient den gleichen Zwecken wie eine entsprechende DTD. Jedoch gehen die Möglichkeiten eines XML-Schemas weit über die von DTDs hinaus. Daher ist es im vorgegebenen Rahmen auch nicht möglich, umfassend auf dieses Thema einzugehen. Im Folgenden daher nur ein ungefährer Abriss der Technik und ihrer Möglichkeiten.

XML-Schemata wurden dem XML-Standard erst einige Jahre nach den DTDs hinzugefügt. Im Gegensatz zu diesen werden sie selbst in XML definiert. Dies erlaubt es, sie mit den selben Werkzeugen zu parsen und zu bearbeiten, mit denen auch die eigentlichen XML-Daten bearbeitet werden. So ist es beispielsweise möglich, ein XML-Schema mittels eines entsprechenden XSLT[10]-Stylesheet-Dokuments direkt in eine Eingabemaske für das Internet umzuformen und dort zur Dateneingabe zur Verfügung zu stellen.

Mit Hilfe eines XML-Schemas lassen sich für die Daten eines Dokuments alle einfachen Standarddatentypen wie beispielsweise Integer, Fließkommazahlen oder Zeichenketten festlegen. Zudem ist die Definition und Anwendung von zusammengesetzten Datentypen möglich. Das Format von Daten lässt sich weiterhin durch weitere Mittel wie Längenangaben und Reguläre Ausdrücke näher spezifizieren.

Nun werden einige Sprachelemente der XML-Schema-Sprache anhand einer möglichen Definition für Konstanten im SAP etwas eingehender betrachtet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:simpleType name="valTypeChoice">
    <xs:restriction base="xs:string">
      <xs:enumeration value="DEC"/>
      <xs:enumeration value="BIN"/>
      <xs:enumeration value="HEX"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="Constant">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Alias"/>
        <xs:choice>
          <xs:element ref="Name"/>
          <xs:element ref="Value"/>
        </xs:choice>
        <xs:element ref="Version" minOccurs="0"/>
```

[10] eXtensible Stylesheet Language for Transformations, eine Sprache zur automatischen Umwandlung von XML in beliebige Formate.

```
        <xs:element ref="Comment"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Value">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="ValueType" type="valTypeChoice"
            use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="Alias" type="xs:string"/>
  <xs:element name="Name" type="xs:string"/>
  <xs:element name="Version" type="xs:integer"/>
  <xs:element name="Comment" type="xs:string"/>
</xs:schema>
```

Zunächst fällt auf, dass die Definition von Elementen mit Hilfe eines XML-Schemas sehr viel mehr Platz in Anspruch nimmt als bei Verwendung einer DTD. Dafür wird in einem XML-Schema Dokument sehr viel klarer festgelegt, wie ein gültiges XML-Dokument auszusehen hat.

Die einfachsten Elemente eines XML-Schema Dokuments sind jene, welche am Ende des obigen Beispiels zu finden sind. Die Definitionen für „Alias“, „Name“, „Version“ und „Comment“ weisen diese als Elemente („element“) aus, die lediglich Daten enthalten. Das Attribut „type“ legt hierbei das Format für die Daten fest.

Der selbstdefinierte Datentyp „valTypeChoice“, ganz am Anfang des Dokuments, zeigt eine Möglichkeit der genaueren Spezifikation erlaubter Inhalte in Datenelementen auf. Es handelt sich um einen einfachen Datentypen („simpleType“), der auf Zeichenketten („string“) basiert und für den als Inhalt die Zeichenketten „DEC“, „BIN“ und „HEX“ erlaubt sind. Neben der hier vorgestellten Aufzählung („enumeration“) existiert noch eine ganze Reihe weiterer Möglichkeiten zu Definition von Restriktionen. Als häufig gebrauchte Vertreter sollen hier die Beschränkung der Länge eines Datenelements sowie die Angabe eines regulären Ausdrucks für dessen Inhalt vorgestellt werden. Ersteres geschieht mittels des Elements „length“, dessen Attribut „value“ die erlaubte Länge festlegt. Für letzteres wird das Element „pattern“ verwendet, bei dem im „value“-Attribut der einzuhaltende reguläre Ausdruck steht. Diese Elemente sind jeweils Unterelemente des „restriction“-Elements. Elemente können auch direkt als „simpleType“ definiert werden, was

dann als Unterelement des „element“-Eintrags geführt werden muss und an Stelle des Attributs „type“ tritt.

Die Definition strukturierter Elemente schließlich ist anhand der Elemente „Constant“ und „Value“ nachzuvollziehen. Beide sind vom Typ „complexType“, der, im Gegensatz zu „simpleType“, für Elemente mit eigenen Unterelementen verwendet wird. Letzteres Element, selbst ein Unterelement des Elements „Constant“, besteht lediglich aus Daten in Form einer Zeichenkette und einem einzelnen Attribut, das sozusagen an das Datenelement angehängt ist. Das Attribut, durch das Schlüsselwort „attribute“ gekennzeichnet, ist vom bekannten Datentyp valTypeChoice, kann also eine der drei dafür erlaubten Zeichenketten enthalten. Es darf nicht weggelassen werden, was vom Wert des Attributs „use“, in diesem Falle also „mandatory“ (im Gegensatz zu „optional“ für optionale Elemente) signalisiert wird.

„Constant“ ist das einzige strukturierte Element in diesem Beispiel, das tatsächlich Unterelemente hat. Unterelemente können in den Elementen „sequence“, „choice“ und „all“ enthalten sein, die selbst – mit gewissen Einschränkungen beim Element „all“ – beliebig untereinander geschachtelt werden können. Eine geordnete Folge von Elementen wird hierbei durch die Unterelemente eines „sequence“-Elements vorgegeben, ein „choice“-Element ermöglicht die Auswahl eines einzelnen seiner Unterelemente, und „all“ ermöglicht die Verwendung seiner Unterelemente in beliebiger Reihenfolge. Ein Unterelement selbst wird durch das Schlüsselwort „element“ als Name des Elements bestimmt. Ein solches Element kann verschiedene Attribute enthalten. Zwingend erforderlich ist das Attribut „ref“, als dessen Wert der Name des referenzierten Elements gespeichert ist. Weiterhin sind beispielsweise Attribute zur Angabe der minimalen („minOccurs“) und maximalen („maxOccurs“) erlaubten Anzahl tatsächlich existierender Elemente der angegebenen Art an dieser Stelle möglich, wie beim Unterelement „Version“ demonstriert. Um eine unlimitierte maximale Anzahl zu erlauben, wird dem Attribut als Wert statt einer Zahl die Zeichenkette „unbounded“ zugewiesen. Sind diese Attribute nicht angegeben, wird als Wert für die Kardinalität standardmäßig eins angenommen.

Eine weitere Eigenschaft der Sprache XML als solcher ist in der zweiten Zeile des Beispiels zu erkennen. Es handelt sich hierbei um die Definition eines Namensraums („namespace“) mittels des Schlüsselworts „xmlns“. Im Beispiel hat der Namensraum den Namen „xs“ und als Wert – sozusagen die ID dieses speziellen Namensraums – die als Attributwert angegebene URL. Es ist dabei keineswegs vorgegeben, URLs zu verwenden, jedoch ist dies praktisch sinnvoll, da so die Einmaligkeit der Zeichenkette einfach sichergestellt werden kann. Damit ist nun auch das Rätsel der verdächtigen „xs:“ am Anfang jedes Elements gelöst: Das Kürzel des Namensraums wird in der Schema-Definition jeweils den einzelnen Elementen, durch Doppelpunkt abgetrennt, als Prefix vorangestellt.

3.1.5 Document Object Model

Die Aufgabe, ein im XML-Format vorliegendes Dokument für Programme einfach zugänglich zu machen, übernimmt ein sogenannter Parser. Dieser liest das gewünschte Dokument ein, überprüft

auf Wunsch dessen Gültigkeit bezüglich einer DTD oder eines XML-Schemas, und stellt geeignete Schnittstellen zum Zugriff auf den Inhalt des Dokuments bereit. Die bekanntesten Modelle für Schnittstellen sind derzeit das „Document Object Model“ (DOM) und die „Simple API for XML“ (SAX). Da für die vorliegende Arbeit das erstere Modell zum Einsatz kommt, soll nur dies hier vorgestellt werden.

DOM stellt im Wesentlichen eine Reihe von Schnittstellen zur Verfügung, die Zugriff auf die geparste Form eines XML-Dokuments ermöglichen. Dies geschieht, indem die Elemente und Unterelemente aus dem Dokument gemäß ihrer vorliegenden Hierarchie in eine Baumstruktur eingeordnet werden. Der so entstehende Baum von Elementen wird vollständig im Speicher gehalten. Die einzelnen Elemente sind als Objekte abgebildet, die mittels spezifischer Methoden ausgelesen und manipuliert werden können. Es ist auch möglich, den Baum selbst zu verändern und so XML-Elemente hinzuzufügen, zu entfernen oder zu verschieben. ist das gewünschte Ergebnis erzielt, wird der vollständige Baum einfach wieder zurück in das XML-Format geschrieben, um beispielsweise in einer Datei gespeichert zu werden. Diese Zusammenhänge sind in Abbildung 3.3 auf der nächsten Seite anschaulich zu erkennen.

Das DOM ist somit eine umfassende und einfach anzuwendende Hilfestellung beim Umgang mit XML-Dokumenten. Mögliche Probleme des DOM sind vor allem der bei großen Dokumenten mitunter nicht unerhebliche Speicherverbrauch sowie die Bereitstellung einer großen Zahl eventuell gar nicht benötigter Informationen, da das DOM alle Bestandteile eines XML-Dokuments in den Knoten des erstellten Baums abbildet, einschließlich beispielsweise Leerzeichen, allem enthaltenen Text und allen Attributen.

Wie die Arbeit mit einem DOM-Baum technisch vor sich geht, wird bei der Vorstellung der hier implementierten Lösung im Detail erläutert. Weitere Informationen zum Document Object Model sind beispielsweise bei [25] zu finden. Einige Anmerkungen zu anderen bekannten Programm-Schnittstellen für XML-Dokumente finden sich bei der Begründung zur Auswahl des DOM.

3.1.6 Parser

Es existiert eine Vielzahl von Parsern für XML auf dem Markt. Diese Parser unterstützen in der Regel entweder den durch DOM vorgegebenen Standard oder SAX. Für die Erstellung von DOM-Bäumen unter Java sind die beiden Parser „Xerxes“ und „Crimson“ die bekanntesten. Beide sind Teil der „Apache Software Foundation“[11], wobei nur ersterer aktiv weiterentwickelt wird. In Suns „Java Development Kit“ in der aktuellen Version 1.4.1 ist dennoch der mit geringerer Funktionalität ausgestattete „Crimson“-Parser integriert[12]. Die meisten Parser genügen problemlos den Anforderungen, die an sie gestellt werden. Jedoch unterscheidet sich die Handhabung der verschiedenen Implementierungen zum Teil deutlich voneinander.

[11] Aktuelle Informationen zur Apache-Organisation und zu den dort entwickelten Hilfsmitteln für die Arbeit mit XML sind unter [4] zu finden.

[12] Dies hat vorwiegend historische Gründe, da „Crimson“ ursprünglich aus dem von Sun entwickelten „Project X Parser“ hervorgegangen ist.

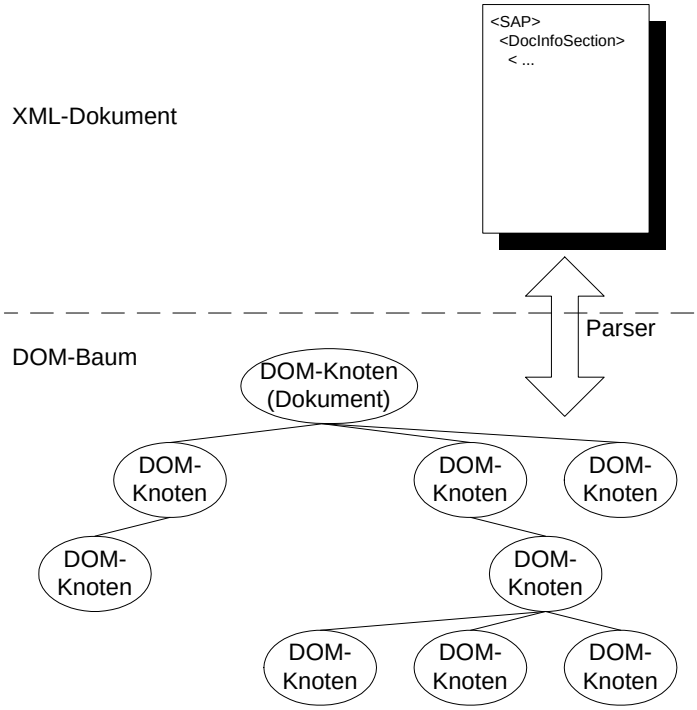

Abb. 3.3: DOM-Baum

Um möglichst viele Implementierungen dennoch einfach gegeneinander austauschbar zu machen, und zugleich dem Benutzer die Handhabung zu erleichtern, wurde von SUN die „Java API for XML Parsing“ (JAXP) entwickelt. JAXP stellt Schnittstellen zur Verfügung, mit deren Hilfe auf einheitliche Weise auf verschiedene Parser zugegriffen werden kann. JAXP ermöglicht das Einlesen von XML-Dokumenten, die Festlegung, ob DTDs oder XML-Schema-Dokumente zur Überprüfung zu verwenden sind und die Aktivierung der meisten anderen Funktionalitäten, welche die unterstützten XML-Parser anbieten. Ein eingehenderes Studium von JAXP ist ebenfalls anhand der vorliegenden Implementierung möglich.

3.2 Eclipse

„Eclipse“ ist der Name eines sehr umfangreichen und ambitionierten Projekts. Ursprünglich angestoßen wurde die Entwicklung von Eclipse durch IBM, von denen es auch heute noch unterstützt wird. Bei Eclipse in seiner heutigen Form handelt es sich in erster Linie um eine „Plattform“. Diese Plattform bildet eine gemeinsame Basis für darin integrierte Programme. Solche Programme, „Plugins“ genannt, bestehen nicht eigenständig, sondern sind Teil der Eclipse-Plattform. Erstellt und integriert werden können sie, die notwendigen Grundkenntnisse vorausgesetzt, von Jedermann. Es ist hierzu nicht notwendig, in den Code der Plattform in irgend einer Weise direkt einzugreifen. Als Programmiersprache empfiehlt sich Java, da auch Eclipse selbst in dieser Sprache geschrieben wurde, und so die verwendeten Konzepte nahtlos verwendet werden kön-

nen. Aber auch die meisten anderen Sprachen sind für die Entwicklung von Plugins problemlos einsetzbar.

Die Eclipse-Plattform ist insbesondere auf die Integration von Plugins zur Unterstützung von Softwareentwicklung ausgelegt. Wenig überraschend ist damit, dass die derzeit bekannteste Anwendung für Eclipse eine integrierte Java Entwicklungsumgebung, das „JDE“, ist. Diese Umgebung stellt, neben den für solche Werkzeuge üblichen Funktonen, auch Hilfen zur Entwicklung von neuen Plugins für Eclipse zur Verfügung. Auch der Java-Teil des hier vorgestellten Projekts ist mit Hilfe dieser Entwicklungsumgebung entstanden.

Da Eclipse für die Verwirklichung des vorliegenden Projekts eine in vielerlei Hinsicht bedeutende Rolle spielt, werden verschiedene Aspekte des Eclipse-Projekts im Folgenden etwas näher beleuchtet. Eine ausführlichere Dokumentation bietet die auch im Programm selbst integrierte Online-Hilfe bei [13].

3.2.1 Prinzipielle Funktionsweise

Eclipse legt das grundsätzliche grafische Erscheinungsbild sowie gewisse elementare zu verwendende Elemente für die integrierten Plugins fest. Damit soll eine einheitliche Benutzerführung gewährleistet werden, um die Eclipse-Plattform mit allen darin integrierten Plugins nach außen hin als ein einheitliches Ganzes erscheinen zu lassen. Die meisten dieser Elemente sind bereits hinlänglich von anderen Anwendungen mit grafischer Benutzerschnittstelle her bekannt. So finden sich beispielsweise sehr bekannt aussehende Menü- und Werkzeugleisten. Es sollte also dem durchschnittlichen Anwender nicht schwer fallen, sich ohne weitere Einarbeitungszeit in Eclipse zurechtzufinden.

Doch wie sind die allgemein genutzten Grafikelemente den verschiedenen Plugins zugeordnet, und wie sind diese untereinander miteinander verknüpft? Eine umfassende Beantwortung dieser für die Eclipse-Plattform zentralen Fragen ist im Rahmen dieser Arbeit nicht möglich – nicht einmal die sehr umfangreiche Online-Hilfe vermag hier in jedem Fall für Klarheit zu sorgen. An dieser Stelle daher nur eine kurze Erläuterung einiger grundlegender Mechanismen.

Plugins können als Grundlage für eigene grafische Elemente auf zwei verschiedene prinzipielle Konstrukte der Eclipse-Plattform zurückgreifen. Dies sind die Elemente „Editor“ und „View“ (Ansicht).

Ein „View“ wird in der Regel verwendet, um Informationen und Daten anzuzeigen, oder dem Anwender die Navigation durch beispielsweise Verzeichnisbäume zu ermöglichen. Als Beispiel für Ersteres kann der „Outline-View“ dienen, in dem jeweils aktuelle Struktur-Informationen zu im gerade aktiven Editor bearbeiteten Daten dargestellt werden. Letzteres ist beispielsweise beim „Navigator-View“ implementiert, der unter anderem die Verzeichnisstruktur innerhalb von Projekten anzeigt. Änderungen, die in einem „View“ vorgenommen werden, werden unmittelbar gespeichert und dargestellt.

Im Gegensatz dazu folgt ein Editor normalerweise dem von Texteditoren her bekannten Konzept, veränderte Daten nur auf Anfrage oder auch in gewissen Intervallen zu speichern und

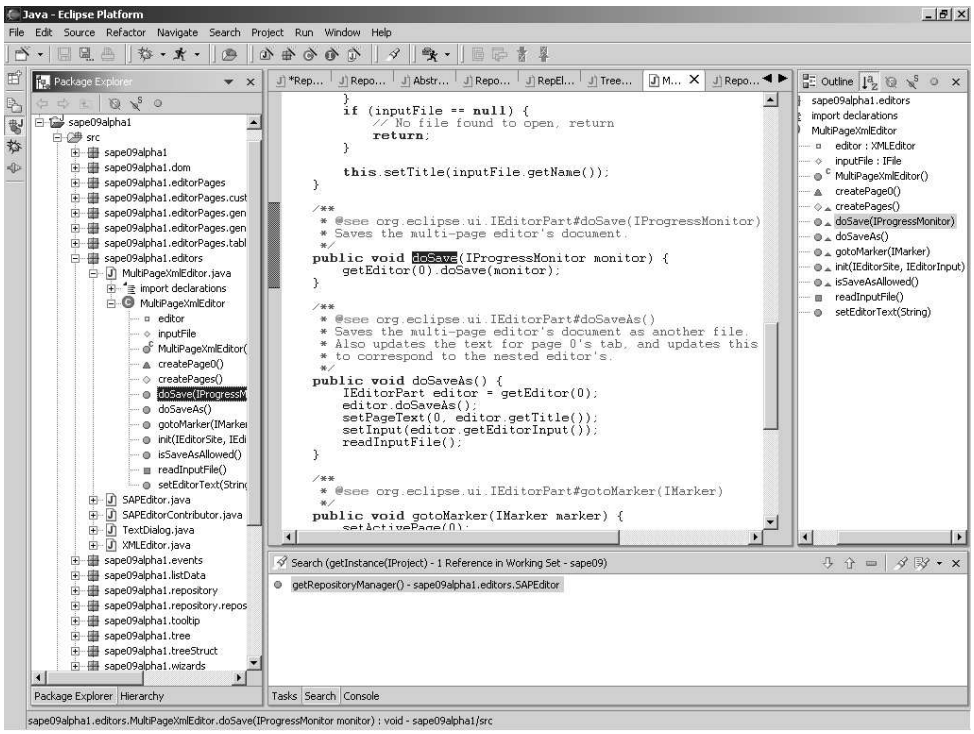

Abb. 3.4: Arbeit mit der in Eclipse integrierten Java-Entwicklungsumgebung

weiterzuverarbeiten. Diese Daten stellen in der Regel die Abbildung eines externen Objekts oder Dokuments dar. Ein typisches Beispiel für einen Editor ist der in der Eclipse-Plattform integrierte Texteditor. Prinzipiell können Editoren aber auch genauso gut dialogbasiert sein oder auf beliebigen anderen Grafikelementen aufbauen.

Die Verbindung eines Editors mit einem oder mehreren Views wird wieder am Beispiel des Text-Editors anschaulich deutlich. In der für die Java-Entwicklungsumgebung erweiterten Version dieses Editors bewirkt ein Klick auf den Namen einer Methode im Outline-View – in dem die Struktur der im aktiven Editor enthaltenen Klasse(n) und Unterelemente dargestellt wird – den Sprung zur Definition dieser Methode in dem im Editor dargestellten Code. Durch solcherlei Zusammenspiel soll möglichst nahtlos integrierte Funktionalität über die gesamte Plattform hinweg erreicht werden.

3.2.2 Grafikelemente

Grafikprogrammierung geschieht für Eclipse-Plugins mit Hilfe der beiden Bibliotheken „SWT“ und „JFace“. SWT ist eine reguläre Grafikbibliothek, vergleichbar etwa mit AWT oder Swing. Alle innerhalb der Eclipse-Plattform verwendete Grafik basiert letztendlich auf SWT.

JFace dagegen deckt vornehmlich komplexere Aspekte der Programmierung von Benutzerschnittstellen ab. Dabei wird auch die Handhabung der in einem Grafikelement dargestellten Daten zumeist mit übernommen.

Als ein praktisches Beispiel soll hier die Implementierung einer Baumstruktur in SWT mit Hilfe von JFace betrachtet werden:

```
// Erzeuge ein neues Objekt zur Anzeige eines Strukturbaums
// innerhalb des Grafikelements 'parentComposite'.
TreeViewer viewer = new TreeViewer(parentComposite);

// Gib dem neuen Objekt Methoden an die Hand, um aus dem später übergebenen
// Daten-Objekt tatsächlich anzuzeigende Daten extrahieren zu können.
viewer.setContentProvider(new WorkbenchContentProvider());

// Dasselbe wie für die eigentlichen Daten,
// hier für die dazu anzuzeigenden Namen.
viewer.setLabelProvider(new WorkbenchLabelProvider());

// Übergebe die tatsächlichen Daten, enthalten im Objekt 'displayTree'.
viewer.setInput(displayTree);

// Öffne alle Elemente des anzuzeigenden Baums bis zur zweiten Ebene.
viewer.expandToLevel(2);
```

Die Klassen `TreeViewer`, `WorkbenchContentProvider` und `WorkbenchLabelProvider` gehören zum JFace-Paket, das Objekt „parentComposite“ ist ein Objekt der Klasse `Composite` aus SWT.

Bei dem hier vorgestellten Programmfragment muss im Rahmen der Implementierung des Baums nicht auf die einzelnen Elemente der Struktur eingegangen werden. Wäre ein solcher Baum nur mit SWT entwickelt worden, wäre es notwendig gewesen, zunächst die im Baum darzustellenden Daten jeweils in ein spezielles Format zu bringen, um diese dann einzeln zusammenzufügen und einem Objekt der `Tree`-Klasse von SWT bekannt zu machen. Bei Veränderungen der Struktur, oder um die darin enthaltenen Informationen wieder auszulesen, wäre der selbe Aufwand zur Ermittlung der Daten und Umformung ins ursprüngliche Format noch einmal angefallen.

Anders wird dies von JFace gehandhabt. Die Daten werden dem TreeViewer-Objekt durch Aufruf der Methode `setInput` in beliebiger Form übergeben. Damit dieses sie dennoch auslesen und bearbeiten kann, wird neben den Daten noch mit Hilfe der Methode `setContentProvider` ein Objekt übergeben, welches das Interface `IContentProvider`[13] bereitstellt. Mit Hilfe der für dieses Interface definierten Methoden kann dann von JFace aus auf die Daten zugegriffen werden. In analoger Weise funktioniert die Bereitstellungen von Beschriftungen für die im Baum darzustellenden Elemente. Hierfür ist das mit der Methode `setLabelProvider` übergebene Objekt zuständig.

[13] Damit tatsächlich die für diesen speziellen Fall einer *B*aumstruktur benötigten Daten extrahiert werden können, muss das Interface vom Subtyp `ITreeContentProvider` sein.

3.2.3 Datenverwaltung

Die beiden im Beispiel des letzten Abschnitts übergebenen Objekte der Klassen `WorkbenchContentProvider` und `WorkbenchLabelProvider` stellen eine einfache Implementierung der beschriebenen Funktionalität dar. Sie erwarten die übergebenen Daten in einer für Eclipse üblichen Form. Diese basiert auf dem Prinzip sogenannter „Adapter“.

Ein Adapter in der Eclipse-Umgebung implementiert in der Regel das Interface `IWorkbenchAdapter`. Um mit den genannten Provider-Klassen zusammenarbeiten zu können, muss das einem `TreeViewer`-Objekt übergebene Daten-Objekt dieses Interface enthalten. Das Interface beinhaltet Methoden zur Ermittlung von über- und untergeordneten Elementen sowie des darzustellenden Namens und Icons für ein Element. Implementiert jedes Element einer darzustellenden Baumstruktur also dieses Interface, so kann der komplette Baum mittels der darin enthaltenen Methoden durch JFace dargestellt werden. Ein wesentlicher Vorteil dieser Vorgehensweise besteht darin, dass viele unterschiedliche Datenstrukturen auf unkomplizierte Weise durch einige wenige standardisierte Grafikelemente dargestellt werden können. Möchte man lieber auf andere Weise auf die Daten zugreifen, so ist auch dies problemlos möglich – es muss lediglich je eine eigene Content- und LabelProvider-Klasse erstellt werden.

Nicht zu verwechseln mit diesen – auf grafische Darstellung ausgerichteten – Adaptern sind jene, die auf dem Interface `IAdaptable` basieren. Dabei handelt es sich um die Möglichkeit, ein Objekt, das diese Schnittstelle implementiert, dynamisch auf andere implementierte Schnittstellen und damit auf die im Objekt enthaltene Funktionalität abzufragen. Dies geschieht mit Hilfe der einzigen Methode des Interfaces:

```
Object getAdapter(Class adapter)
```

Ist das als `Class`-Deklaration übergebene Interface in der Klasse enthalten, wird eine geeignete Instanz dieser Klasse zurückgegeben. Auf diese Weise kann die verfügbare Funktionalität einer Klasse, von der nur bekannt sein muss, dass sie das Interface `IAdaptable` implementiert, zur Laufzeit erweitert werden. Für den Anwender der Schnittstelle ist es dabei nicht notwendig, aur interne Konstrukte zur Abfrage der Struktur der Sprachelemente selbst zurückzugreifen.

Auch für die hier besprochenen Baumstrukturen kommt dieses Konzept im Zusammenhang mit JFace zum Einsatz. So muss im Beispiel das die Daten bereitstellende Objekt „displayTree“ ebenfalls eine Schnittstelle vom Typ `IAdaptable` implementieren. Tatsächlich wird nämlich von den Workbench-Provider Klassen über diese Schnittstelle erst die Schnittstelle `IWorkbenchAdapter` abgefragt und im Erfolgsfall das zurückgegebene Objekt genutzt.

Ein praktisches Beispiel zu den hier gemachten Ausführungen findet sich bei der Besprechung der tatsächlichen Realisierung des Projekts.

4 Anforderungen an eine neue Implementierung

Die Anforderungen an eine Neuimplementierung des Prozesses der Erstellung und Verarbeitung von SAP- und AIM-Dokumenten wurden in einigen langwierigen Diskussionsrunden zusammengetragen. Hieran waren sowohl Entwickler beteiligt, die langjährige Erfahrungen mit der bisherigen Vorgehensweise einbringen konnten, als auch Personen, die im Vorfeld mit der Analyse neuer Möglichkeiten betraut waren. Diese Überlegungen setzten bereits lange vor Beginn meiner Diplomarbeit ein, da die Mängel des bestehenden Systems von Beginn an offensichtlich waren. Die Ergebnisse wurden in internen Protokollen und Anforderungsdokumenten festgehalten, unter anderem in [11; 14; 15].

Daraus ergibt sich für die längerfristige Planung die Vorgabe, die gesamte zur Verarbeitung von SAP- und AIM-Dokumenten eingesetzte Toolkette dergestalt umzubauen, dass zur Datenspeicherung ein einheitliches Format verwendet wird. Dies wird, so die Hoffnung, einige Zwischenschritte – beispielsweise die Umwandlung in das pdf/mdf-Format – in der Verarbeitung unnötig machen und die Zuverlässigkeit und Wartungsfreundlichkeit des Gesamtsystems erhöhen. Jedoch ist diese Thematik nicht Inhalt der vorliegenden Diplomarbeit und wird daher nicht näher behandelt.

An dieser Stelle ist zum einen das neue Datenformat selbst von Bedeutung, zum anderen die Erstellung eines geeigneten Hilfsmittels für dessen Bearbeitung, hier einfach als „Editor“ bezeichnet. Die Ergebnisse der Planungen und Überlegungen, die im Vorfeld bezüglich dieser beiden Komponenten stattgefunden haben, sind im Folgenden wiedergegeben.

4.1 Datenformat

Das bisher genutzte Format zur Speicherung von SAP- und AIM-Daten, das Microsoft Word Dokumentenformat, bringt einige Unannehmlichkeiten mit sich. Diese resultieren hauptsächlich daraus, dass zur Bearbeitung der Daten ausschließlich MS Word eingesetzt werden kann und wurden in dem entsprechenden Kapitel bereits ausführlich diskutiert.

Um Probleme hier zukünftig möglichst zu vermeiden, wurden einige Bedingungen für ein neues Format formuliert. Diese sind im einzelnen:

- ein einfaches Dateiformat, vorzugsweise ASCII;
- klar und umfassend definierte Syntax;

- unproblematische automatische Weiterverarbeitung;
- die Möglichkeit zur manuellen Bearbeitung und Reparatur von Dateien;
- einfache Zusammenführung verschiedener Dokumente oder verschiedener Versionen eines Dokuments im Rahmen der Versionsverwaltung, namentlich des Programmpakets „clearcase“;
- eine Möglichkeit zum Einbinden von Elementen, auch über die Grenzen eines Dokuments hinweg, in Form von Verknüpfungen;
- die möglichst weitgehend automatisierte Konvertierung aller existierenden Dokumente im betroffenen Bereich;
- ein und dasselbe Quelldokument zur Generierung aller benötigten Ausgabedateien, insbesondere von menschenlesbarer Dokumentation und pdf/mdf-Dokumenten.

4.2 Programm zur Unterstützung der Arbeit mit den Daten

Der Editor, mit dem das neue Datenformat bearbeitet werden kann, sollte der gestellten Aufgabe angepasst sein, und spezifische Hilfe und Unterstützung anbieten. Anforderungen an einen neu zu entwickelnden Editor sind insbesondere:

- Microsoft Word in diesem Aufgabenfeld vollständig zu ersetzen;
- alle Möglichkeiten des neu entwickelten Formats anzubieten und zu nutzen, insbesondere die Funktionalitäten zum Einbinden von Verweisen auch aus anderen Dokumenten in Elemente des aktuell bearbeiteten Dokuments;
- die Gefahr menschlicher Fehler im Arbeitsprozess zu minimieren;
- bei der Erstellung syntaktisch richtiger Dokumente behilflich sein;
- an geeigneten Stellen semantische Hilfen anzubieten;
- den Benutzer zur Verwendung bereits existierender Elemente zu ermutigen, und ihm eine einfache Möglichkeit hierzu an die Hand zu geben, um so der weitverbreiteten „Copy/Paste-Mentalität“[1] entgegenzuwirken.

[1] Die Neigung dazu, bereits existierende Elemente zu kopieren und in abgewandelter Form zu einem neuen Zweck einzusetzen. Bei unkritischem Einsatz kann diese Methode zu schwer auffindbaren Fehlern und Problemen und zu unnötigen Redundanzen führen.

5 Auswahl der technischen Grundlagen

Um die formulierten Anforderungen zu erfüllen, wurden verschiedene Ansätze analysiert und auf Tauglichkeit und Zweckmäßigkeit hin untersucht. In diesem Kapitel werden die dabei betrachteten Optionen vorgestellt sowie die Gründe für die letztlich getroffene Auswahl dargelegt.

5.1 Neues Datenformat

An das neu zu entwickelnde Datenformat werden hohe Anforderungen gestellt. Es soll zugleich die beiden alten Formate vereinigen, ihre bisherigen Unzulänglichkeiten beheben, neue Funktionalitäten anbieten, und unkompliziert im Umgang sein. Was dies im Hinblick auf die möglichen Optionen bedeutet, wird im Folgenden diskutiert.

5.1.1 Erweiterung des bestehenden Konzepts

Auch unter Beibehaltung der existierenden Grundlagen sind sehr wohl Verbesserungen für den Arbeitsablauf denkbar. Schon beispielsweise durch eine Anpassung des Formats der SAP-Dokumente an das für AIM-Dokumente verwandte könnte größere Klarheit und Logik im Aufbau der entsprechenden Dokumente und in ihrer Handhabung erreicht werden. Es wäre auch denkbar, Microsoft Word als Editor so anzupassen, dass es einfacher für Anwender wird, sich auf die Eingabe und Bearbeitung der eigentlichen Daten zu konzentrieren. Hierzu wäre beispielsweise der Einsatz von Makros oder Visual Basic Skripten denkbar.

Jedoch kann in dieser Weise nur ein Teil der existierenden Unzulänglichkeiten aufgelöst werden. Kompatibilitätsprobleme zwischen verschiedenen Word-Formaten und -Features würden nach wie vor in bekannter Häufigkeit auftreten, und die Wartung der erzeugten Dokumente wäre nach wie vor ein zeitraubendes Unterfangen. Hierbei sei einmal mehr insbesondere auf die bei Texas Instruments installierte Quellcode-Versionsverwaltung verwiesen, die auch SAP- und MSG-Dokumente betrifft, und in deren Rahmen häufig die Zusammenführung verschiedener Versionen eines Dokuments notwendig ist. Dies erfordert im Falle von MS-Word Dokumenten in aller Regel sehr viel Handarbeit.

Ein weiteres unangenehmes Problem würde sich beim Übergang vom jetzigen zu einem neuen auf MS Word basierenden Format aus der notwendigen Transformation der bereits existierenden Dokumente ergeben. Dies wäre effektiv nur in Word selbst möglich, was sehr wahrscheinlich erneut zu einer ganzen Reihe zeitraubender Probleme und viel Handarbeit bei der Umstellung führen würde.

Zusammenfassend scheint es somit sicher, dass die gesteckten Ziele durch eine einfache Erweiterung des bestehenden Formats nur teilweise zu erreichen wären. Die geringen Verbesserungen gegenüber dem gegenwärtigen Zustand rechtfertigen den zu erwartenden Aufwand nicht.

5.1.2 Eigenes Format in ASCII

Es wurden ernsthafte Überlegungen angestellt, die Daten aus SAP- und AIM-Dokumenten einfach nur textuell im ASCII-Format abzulegen. Dies sollte in einem neu festzulegenden Format geschehen, das ausreichend strukturiert und übersichtlich ist, um dem Benutzer eine direkte Bearbeitung mit Hilfe eines beliebigen Texteditors zu erlauben. Diesem Plan zu Folge sollte dem Benutzer lediglich Unterstützung bei der Verwaltung von Referenzen auf Elemente der gegenwärtig interessanten Dokumente geboten werden, beispielsweise in Form eines kleinen Hilfsprogramms, das alle möglichen Elemente übersichtlich darstellt.

Dieser Ansatz bietet den Vorteil, dass die resultierenden Dokumente leicht weiterzuverarbeiten sind. Auch ist das Problem des zu verwendenden Werkzeugs auf elegante Weise gelöst, da einfach ein beliebiger bevorzugter Texteditor verwendet werden kann. Defekte Dokumente können einfach repariert werden, und die Zusammenführung von Dokumenten im ASCII-Format wird von einer Reihe von Standard-Werkzeugen zuverlässig und automatisiert erledigt.

Jedoch sind auch hier einige nicht unerhebliche Probleme zu erwarten. So ist es, auch unter Verwendung von Standardwerkzeugen, keineswegs trivial, einen eigenen Parser zu generieren. Genau das wäre aber notwendig, um das selbst definierte Format auch maschinell weiterverarbeiten zu können. Stellt dies noch kein wirkliches Problem dar, so ist die manuelle Bearbeitung schon als weitaus problematischer zu betrachten. Wohl werden einige Probleme des Ausgangsformats gelöst. Es wäre nun nicht mehr nötig, auf Layout und Inhalt der Daten gleichzeitig zu achten, da eine Vermischung der beiden Dinge bei der Definition von vorn herein vermieden werden könnte. Jedoch hingen die syntaktische wie auch die semantische Korrektheit der existierenden Dokumente allein von der Sorgfalt desjenigen ab, der diese Dokumente anlegt beziehungsweise bearbeitet. Eine Syntax-Überprüfung würde erst beim Parsen des Dokuments stattfinden und könnte auch nur dann klare Ergebnisse liefern, wenn im Parser geeignete Funktionalität zur Ausgabe aussagekräftiger Fehlermeldungen integriert ist. Die inhaltliche Überprüfung von Dokumenten, beispielsweise auf die Verwendung sinnvoller Datentypen oder -formate wäre ebenfalls frühestens im nachfolgenden Parser möglich.

Im Anschluss an diesen Parser müsste mittels eines noch zu entwickelnden Programms die automatische Generierung von pdf/mdf-Dokumenten ermöglicht werden. Dies bedeutet zusätzlichen Entwicklungsaufwand gegenüber der bereits existierenden Lösung.

Dazu kommen noch die im entsprechenden Kapitel bereits besprochenen Probleme und Widrigkeiten bei der Verwendung eines einfachen selbstdefinierten Datenformats und dessen resultierende Unterlegenheit im Vergleich zu XML.

5.1.3 XML

XML als Datenformat bietet einige nicht zu bestreitende Vorzüge gegenüber anderen Formaten. Dies wurde bei der Vorstellung von XML bereits eingehend erläutert. Einige der für die Erfüllung der hier vorliegenden Anforderungen besonders herausragenden Merkmale seien im Folgenden noch einmal kurz zusammengefasst. Es handelt sich hierbei um

- die einfache Erweiterbarkeit eines in XML definierten Formats, ohne damit die Konsistenz bereits existierender Daten zu gefährden oder die Funktionsfähigkeit bereits existierender Programme in Frage zu stellen,
- die Existenz fertiger Parser zum Auslesen und Schreiben von Dateien in XML sowie einer großen Zahl weiterer frei verfügbarer Hilfsprogramme für die Arbeit mit XML,
- die allgemeine Robustheit des XML-Formats gegenüber möglichen Fehlern bei der Bearbeitung von darin abgelegten Daten,
- die einfache Überführbarkeit von im XML-Format vorliegenden Daten in andere Datenformate mittels XSLT, beispielsweise zum Zwecke der Erstellung von Dokumentation oder pdf/mdf-Dokumenten.

Diese Vorzüge führen nicht nur zu einer unkomplizierten Implementierung und einfachen praktischen Anwendbarkeit von XML, sie lassen die Definition eines Datenformats für SAP- und AIM-Dokumente in XML auch in Anbetracht der Notwendigkeit einer Verarbeitung durch verschiedene Programmmodule (Editor, Konverter aus den alten Formaten, Konverter nach pdf/mdf, Generator für Dokumentation) als sehr geeignet erscheinen.

Auch die Verwendung von XML bietet nicht nur Vorteile. So ist die Zusammenführung von Dokumenten nicht mehr ganz so einfach möglich, wie bei der Verwendung eines einfachen ASCII-Formats. Dies resultiert daraus, dass bei derlei Operationen im Falle von Entscheidungskonflikten manuelles Eingreifen nötig werden kann. Hierzu muss in diesem Fall entweder der Anwender ein gewisses Grundverständnis des XML-Formats mitbringen, oder er benötigt Hilfe durch ein spezielles Programm. Dieses Problem kann elegant durch die Verwendung eines von zahlreichen bereits existierenden Programmen oder auch die Entwicklung eines neuen Programms gelöst werden, welches das Zusammenführen von XML-Dokumenten auch ohne Kenntnis des zugrundeliegenden Formats ermöglicht. In jedem Falle allerdings wird hier ein gewisser Mehraufwand in der Entwicklungsphase verursacht.

Ein weiterer Punkt, in dem die Verwendung von XML Mehraufwand mit sich bringt, ist die Bereitstellung eines geeigneten Editors. Verwendet man einen einfachen Texteditor, so handelt man sich wiederum die bereits bei der zuletzt vorgestellten Variante des reinen ASCII-Textes erwähnten Unzulänglichkeiten ein: dem Benutzer werden nur unzureichende Hilfen bei der Erstellung und Bearbeitung von Dokumenten geboten, und Fehler werden grundsätzlich erst bei der Weiterverarbeitung gefunden. Hinzu kommt hier nun auch noch die Notwendigkeit der spezifischen Kenntnis von XML. Damit erscheint es bei Verwendung von XML unumgänglich und angebracht,

einen eigenen Editor zur Eingabe und Pflege der SAP- und AIM-Daten bereitzustellen. Dies bietet sehr vorteilhafte Möglichkeiten für die Unterstützung und Führung des Anwenders. Aber es bedeutet auch wiederum nicht unerheblichen Mehraufwand in der Erstellung und Pflege des Editors.

5.1.4 Entscheidung für XML

Neben den bisher vorgestellten Varianten wäre es sicherlich auch möglich, andere Lösungen zu finden. So ist es beispielsweise denkbar, ein selbst entworfenes ASCII-Format mit einem eigenen Editor zu kombinieren. Oder es könnte ein Editor direkt für das pdf/mdf-Format entwickelt werden. Doch bietet die Verwendung von XML in Kombination mit der Erstellung eines spezialisierten Editors in Bezug auf die formulierten Bedingungen die umfassenste und nachhaltigste Lösung. Das zugrundeliegende Datenformat ist damit zugleich robust und leicht zu handhaben, und dem Benutzer können angemessene und effiziente Hilfestellungen beim Umgang mit den Daten gegeben werden. Alle anderen in Betracht gezogenen Kompromisse fallen demgegenüber zumindest in einem der beiden Punkte zurück.

Um die Unterstützung von Verknüpfungen zwischen verschiedenen Dokumenten zu gewährleisten, wird dem Editor eine Einheit zur Datenhaltung, das so getaufte „Daten-Repository“, zur Seite gestellt. Die tatsächliche Implementierung der drei Teilbereiche XML-Grammatik, Editor und Repository wird im entsprechenden Kapitel ausführlich und eingehend behandelt.

5.2 Zu entwickelnde Software

Für die zu entwickelnde Software sind einige Rahmenbedingungen bereits vorgegeben. Die zu entwickelnden Programme müssen unter einem Microsoft Windows Betriebssystem lauffähig sein. Weiterhin ist es wünschenswert, längerfristig auch die Verwendung in einem UNIX-Betriebssystem zu ermöglichen. Diese beiden Kriterien ergeben sich daraus, dass einerseits die Entwicklung der Protokollstack-Software zum gegenwärtigen Zeitpunkt weitestgehend auf Windows-Betriebssystemen – namentlich Windows NT – basiert, andererseits aber in weiten Unternehmensbereichen bei Texas Instruments UNIX-Betriebssysteme zum Einsatz kommen.

5.2.1 Programmiersprache und Grafikbibliothek

Bereits die genannten grundlegenden Bedingungen schließen die Verwendung eines auf ein Betriebssystem festgelegten Systems für Software und Grafik praktisch aus. Die Benutzung beispielsweise der MFC[1] zur Darstellung der benötigten Grafikelemente würde die Notwendigkeit der Portierung auf ein anderes System von vorn herein implizieren.

Es ist glücklicherweise nicht notwendig, die zu erstellende Software betriebssystemnah und auf Schonung von Ressourcen ausgelegt zu entwickeln, da es sich dabei „lediglich“ um Hilfsprogramme handelt. Im Gegensatz zu der Software, die am Ende in Mobiltelefonen oder anderen

[1] „Microsoft Foundation Classes“

integrierten Systemen („embedded systems“) zum Einsatz kommt, sind keine engen Grenzen für RAM- und ROM-Verbrauch einzuhalten, und keine unbedingt zwingenden Geschwindigkeitsvorgaben zu erreichen. Die Performance des Gesamtpakets muss allerdings ausreichend sein, um den Editor und die umgebenden Programmteile auch auf alten Hardwareumgebungen noch in akzeptabler Geschwindigkeit lauffähig zu machen. Dies ist notwendig, da derzeit noch eine große Zahl mehrere Jahre alter PCs zur Softwareentwicklung eingesetzt werden, deren Austausch erst für die nächsten Jahre vorgesehen ist.

Um die Lauffähigkeit sowohl unter Windows als auch UNIX längerfristig mit geringem Aufwand sicherstellen zu können, bieten sich eine Reihe betriebssystemübergreifender Softwareplattformen an. Die nachfolgend beschriebenen Systeme wurden bei der Suche nach einer geeigneten Grundlage in Betracht gezogen.

Perl ist eine bewährte Scriptsprache, die bei TI bereits vielfach eingesetzt wird. Mit Hilfe von Zusatzpaketen ist es möglich, mit Perl grafische Oberflächen zu erstellen. Perl ist auf praktisch allen relevanten PCs bereits installiert, und ein in Perl geschriebenes Programm ist, da es bei der Ausführung interpretiert wird, in der Regel problemlos auch auf anderen Hardwareplattformen lauffähig. Ein Nachteil von Perl ist zweifelsohne die geringe Geschwindigkeit bei der Ausführung, die weitgehend durch das Konzept einer Interpreter-Sprache bedingt ist. Bereits in Perl vorliegende und verwendete Software hat, zum Teil aus diesem Grunde, bei TI nicht mehr den besten Ruf, und so wurde schnell entschieden, dass ein solch umfangreiches Projekt wie das hier geplante nicht in dieser oder auch einer anderen Scriptsprache verwirklicht werden sollte.

Um Microsoft Visual Studio, die bei TI standardmäßig eingesetzte Entwicklungsumgebung, verwenden zu können, wäre es notwendig, C oder C++[2] als Programmiersprache zu wählen. Um eine Interoperabilität zwischen verschiedenen Betriebssystemen zu gewährleisten, bieten sich dann verschiedene Programmpakete an. Zwei prominente Vertreter sind „QT“ und „wxWindows“. Beide Systeme sind zuverlässig und ausgereift und beide stellen ausreichende Möglichkeiten zur Erstellung einer grafischen Oberfläche zur Verfügung, die sowohl unter UNIX- wie auch unter Windowsbetriebssystemen übersetzbar und funktionstüchtig ist. Ein Nachteil von QT ist, dass die kommerzielle Verwendung, zumindest unter Windows, derzeit mit zusätzlichen Lizenzgebühren an Trolltech, die Entwickler von QT, verbunden ist. Ein Nachteil von wxWindows, das von einer unabhängigen Gruppe von Entwicklern betreut wird, ist demgegenüber dessen bis heute nicht sehr gefestigte Akzeptanz für kommerzielle Entwicklungen. Beides jedoch stellt keinen definitiven Hinderungsgrund für den Einsatz eines der beiden Pakete dar. Näher zu analysieren, warum die letztendliche Entscheidung dennoch gegen C und C++ ausgefallen ist, soll weiter unten versucht werden.

Beim Einsatz von Java als Programmiersprache kann auf eines der drei gegenwärtig frei verfügbaren Grafikpakete AWT, Swing und SWT zurückgegriffen werden. Diese entscheiden sich zunächst im Ansatz, mit dem Grafikelemente erzeugt und zur Verfügung gestellt werden. Das

[2] C#, das gegenwärtig in aller Munde ist, wird von der bei TI verwendeten Version des Visual Studio nicht unterstützt und kam schon von daher nicht zur Verwendung in Frage.

„Abstract Windowing Toolkit“ (AWT), von Sun entwickelt und das älteste der drei Pakete, verfolgt hierzu einen minimalistischen Ansatz. Es werden nur solche Grafikelemente zur Verfügung gestellt, die auf allen wesentlichen von Java selbst unterstützten Softwareplattformen existieren. Dies führt dazu, dass in AWT im Großen und Ganzen lediglich grundlegende Grafikelemente wie Linien, Buttons oder Menüs zu finden sind. Alle weitergehenden Konzepte dagegen, seien es umfangreichere vorgefertigte Dialogboxen, Hilfsmittel zur Arbeit mit Baumstrukturen oder komplexere Texteingabefelder, müssen vom Anwender selbst implementiert werden. Zwar führt dieser Weg zu einer akzeptablen Geschwindigkeit, da die wenigen implementierten Grafikelemente direkt auf entsprechenden Betriebssystemfunktionen aufbauen, jedoch hat sich die praktische Arbeit mit dem AWT in den Jahren seiner Existenz vornehmlich wegen des geringen Funktionsumfangs als umständlich und wenig effizient erwiesen.

Aufgrund der Erfahrungen mit AWT hat Sun ein anderes Konzept entwickelt, das einen entgegengesetzten Weg verfolgt. „Swing“, so der – nach [27] – eigentlich inoffizielle, aber dennoch weithin verwendete Name des Projekts, stellt weitgehende Hilfen bei der Implementierung einer grafischen Oberfläche bereit. Es werden nicht nur komplexe Grafikelemente fertig zur Verfügung gestellt, auch Konzepte wie „Drag & Drop“ und „Hot-Keys“, um nur zwei zu nennen, muss der Benutzer nicht mehr selbst implementieren. Um diese weitergehende Funktionalität zu erreichen, wird bei Swing nicht auf die jeweilige Implementierung in einem Betriebssystem zurückgegriffen. Stattdessen werden die entsprechenden Grafikelemente und Systemfunktionalitäten aus einfachen Elementen beziehungsweise grundlegenden Funktionalitäten des Systems nachgebaut. So wird unter Windows nicht der standardmäßig zur Verfügung gestellte Datei-Auswahl-Dialog genutzt, sondern stattdessen ein eigener, von Grund auf selbst aus Linien und anderen einfachen Elementen zusammengesetzter. Dies hat eine ganze Reihe von tiefgreifenden Auswirkungen. Zum einen macht diese Vorgehensweise Swing hochgradig portabel. Es genügt, die wenigen grundlegenden Elemente zur Verfügung zu stellen, auf denen Swing aufbaut, um das System lauffähig zu machen. Weiterhin findet der Anwender eines mit Hilfe von Swing entwickelten Programms unter jedem Betriebssystem die selbe Oberfläche vor. Ohne weitere Voreinstellungen unterscheidet sich das Aussehen einer solchen Anwendung beispielsweise unter LINUX nicht von dem unter Windows. Damit könnte nun argumentiert werden, dass dem Anwender so das Zurechtfinden im Programm erleichtert wird. Allerdings bedeutet dies auch, dass dem an „seine“ Oberfläche gewohnten Anwender zunächst eine Umstellung auf das Aussehen von Swing abverlangt wird. Das einheitliche Aussehen, das bekannte „look and feel“ aller Anwendungen auf *einem* Betriebssystem ist damit nicht mehr gegeben.

Auch das Laufzeitverhalten eines Programms, das mit Swing arbeitet, ist mitunter wenig zufriedenstellend. Dies resultiert zumindest teilweise daraus, dass eben die meisten komplexeren Vorgänge innerhalb der entsprechenden Java-Bibliotheken emuliert werden. Durch diese extrem betriebssystemferne Vorgehensweise und andere, hier nicht näher erläuterte strukturelle Probleme, ist die Arbeit mit Swing-basierenden Programmen für den Anwender oftmals durch deutliche Verzögerungen beispielsweise bei der Auswahl eines Menüs oder beim Druck auf einen Button

gekennzeichnet. Auf langsameren Rechnern kann man häufig genau beobachten, wie sich beim Programmstart oder bei Veränderungen in der Darstellung langsam der Bildschirm aufbaut. Dieser Ruf einer gewissen Trägheit eilt Swing bereits seit der Einführung voraus, und da es sich hierbei leider nicht nur um ein Gerücht handelt, sondern um anschaulich zu zeigende Tatsachen, macht diese Eigenschaft Swing wenig geeignet, um für das geplante Projekt Akzeptanz bei den späteren Anwendern zu gewinnen.

Als dritter und jüngster Vertreter unter den frei verfügbaren Grafikpaketen für Java schließlich ist SWT zu nennen. SWT, das „Standard Widget Toolkit“, wurde von IBM entwickelt und steht somit in direkter Konkurrenz zu AWT und Swing. SWT versucht, die Vorteile der beiden anderen Modelle zu vereinen. Es werden Möglichkeiten zur Verfügung gestellt, die denen von Swing vergleichbar sind. Jedoch werden diese, wo immer möglich, nicht komplett neu implementiert, sondern vom jeweiligen Betriebssystem übernommen. So ist der normale Datei-Auswahl-Dialog von SWT eben *nicht* selbst zusammengebaut, sondern beispielsweise unter Windows genau der gleiche, der auch bei den meisten direkten Windows-Programmen zur Anwendung kommt. Natürlich funktioniert auch das nicht ohne Kompromisse in anderen Bereichen. So ist SWT, zumindest in der gegenwärtig vorliegenden Version 2.0, bei weitem nicht so kongruent und in sich schlüssig wie das Konzept von Swing. In Swing ist es zum Beispiel durchgehend möglich, Elemente innerhalb von anderen Elementen zu platzieren, wohingegen SWT dies an gewissen Stellen nicht zulässt. Dafür ist die tatsächliche Verwendung der eingebauten Möglichkeiten bei SWT häufig geradliniger und unkomplizierter machbar. Wer schon einmal Drag & Drop-Funktionalität in ein auf Swing basierendes Programm einzubauen versucht hat, wird erkennen, was gemeint ist...

Damit stellt SWT den Versuch einer typischen 80/20-Lösung dar. Diese Aussage soll andeuten, dass mit Hilfe von SWT 80% der zu lösenden Aufgaben mit 20% des Einsatzes zu bewältigen sind. Ist dies auch ein wenig plakativ formuliert, so trifft es doch den Kern der Sache, beim Vergleich zwischen SWT und Swing. Bedenkt man noch, dass die zusätzlichen Möglichkeiten von Swing zu einem großen Teil in der Praxis nie genutzt werden dürften – wer würde auf die Idee kommen, einen Titelbalken mit einem Fenster zu „verzieren“ – oder gar den Benutzer auf zeitraubende Irrwege zu führen geeignet sind, so erscheint der Ansatz von SWT als durchaus sehr ausgewogen und sinnvoll.

Um die Ziele von SWT, hohe Geschwindigkeit und einfache Benutzbarkeit, in der geschilderten Weise zu erreichen, wurden einige für Java bisher untypische Design-Entscheidungen getroffen. SWT baut seine Funktionalität auf C-Bibliotheken auf. Da diese, im Gegensatz zu AWT und Swing, nicht in der standardmäßigen Java-Umgebung enthalten sind, muss SWT auf jede Softwareplattform, auf der es verwendet werden soll, zuerst portiert werden. Dies ist für die meisten verbreiteten Betriebssysteme bereits geschehen. Jedoch wäre es wahrscheinlich schwierig, ein auf SWT basierendes Programm ohne weiteres mit der berühmten Java-fähigen Kaffeemaschine zu verwenden. Während dies einen empfindlichen Einschnitt in die hehren Grundsätze von Java darstellt, hat es doch geringe bis überhaupt keine Auswirkungen auf die praktische Arbeit im

vorgesehenen Bereich, da eine SWT-Portierung für alle in Frage kommenden Betriebssysteme existiert.

Auch inhaltlich und in der Vorgehensweise bei der Verwendung unterscheidet sich SWT in einigen Aspekten von der gewohnten Java-Arbeitsweise. Dies liegt darin begründet, dass man bei der Entwicklung von SWT bemüht war, die Grafikbibliotheken bekannter Betriebssysteme möglichst direkt und ohne ein allzu großes Maß an dazwischenliegendem Mehraufwand aus Java heraus verwenden zu können. Augenfällig wird dies beispielsweise beim Speichermanagement[3]. Geschieht dieses bei Java in der Regel automatisch und ohne menschliches Zutun, so erwartet SWT die explizite Freigabe belegter Ressourcen durch den Benutzer, sobald diese nicht mehr gebraucht werden. Auf diese Weise wird die Notwendigkeit vermieden, ein spezielles Speichermanagement sozusagen um die verwendeten Grafikfunktionen herumzubauen, da diese Funktionen selbst – ganz im C-Stil – die Speicherverwaltung voll dem Benutzer überlassen. Dies bedeutet bei der Programmierung keinerlei relevanten Mehraufwand und stellt für Entwickler, die den Umgang mit C oder C++ gewohnt sind, auch keinerlei Problem dar. Ausschließlich in der Java-Welt beheimatete Entwickler dürften möglicherweise gewisse Schwierigkeiten mit dem für sie neuen Konzept haben.

Da eine vergleichsweise hohe Geschwindigkeit für das hier vorgestellte Projekt durchaus von Bedeutung ist, und eine Benutzeroberfläche in bereits bekannter Form für den Anwender durchaus angenehm und von Vorteil ist sowie andererseits alle in der Entwicklung zu erwarteten Probleme mit Hilfe von SWT gelöst werden können, ist diesem gegenüber der Verwendung von Swing der Vorzug zu geben.

Zur Entscheidung zwischen C und Java gäbe es viel zu schreiben. Ganze Glaubenskriege bauen auf den Vorzügen der einen oder der anderen Sprache auf. Tatsache jedoch ist, dass mit beiden Sprachen in ähnlicher Weise gearbeitet werden kann und auch der Aufwand zur Erreichung eines bestimmten Ziels zumeist ähnlich ist. Damit bleibt es, wenn weder Kompatibilitäts- noch Geschwindigkeitsprobleme ausschlaggebend sind, am Ende zumeist den Vorlieben des betroffenen Entwicklers oder „politischen“ Erwägungen überlassen, welche Sprache zum Einsatz kommt.

Letzteres war auch im Falle dieses Projekts ausschlaggebend. Der Einsatz von Java wurde von den ausschlaggebenden Stellen gewollt, und so wurde Java zur Sprache der Wahl. Dies ergibt sich unter anderem aus unternehmensweiten Bestrebungen zur Vereinheitlichung und wohl auch aus dem Glauben an die Überlegenheit und Zukunft von Java. Hierüber soll an dieser Stelle nicht weiter diskutiert werden, da diese Frage für die Verwirklichung des vorgegebenen Projekts nicht von Bedeutung ist.

5.2.2 Plattform

Seit einiger Zeit werden zur Entwicklung von Programmen fertige Plattformen angeboten, in die neu zu entwickelnde Software mehr oder weniger nahtlos eingebunden werden kann. Solche

[3] Speichermanagement ist die Art der Verwaltung des zur Verfügung stehenden Speichers – wann und von wem beispielsweise nicht mehr benötigte Variablen wieder freigegeben werden.

Plattformen existieren auch für die Programmiersprache Java. Von Bedeutung sind hier gegenwärtig zwei Systeme. Dies sind „Java Beans“, eingeführt und unterstützt von Sun, und Eclipse, das von IBM entwickelt wurde. Beide Plattformen sind freie Software, und beide bieten als eine Anwendung eine integrierte Java-Entwicklungsumgebung an. Neben diesen freien Systemen gibt es eine Reihe kommerzieller Lösungen, die aber für das hier vorgestellte Projekt keine Vorteile bieten[4].

Als Grafikbibliothek liegt den Java Beans – wenig verwunderlich – Swing zugrunde, während Eclipse auf SWT aufbaut. Beide Systeme erben wesentliche Eigenschaften der von ihnen verwendeten Grafiksysteme. Sind die Java Beans ein mittlerweile äußerst umfangreiches Softwarepaket, das eine bemerkenswerte Fülle von Möglichkeiten eröffnet, so beschränkt sich Eclipse – noch – auf die wesentlichen notwendigen Elemente. Dies mag wohl damit zu tun haben, dass auch hier wiederum das Produkt aus der Feder von IBM wesentlich jünger ist als dasjenige aus dem Hause Sun. Auch bei Optik und Performance baut das Verhalten der beiden Pakete – gezwungenermaßen – auf den Charakteristika der jeweiligen Grafikbibliotheken auf. Ist Eclipse relativ schnell und präsentiert sich im Gewande des jeweiligen Betriebssystems, so ist bei der Arbeit mit den Java Beans immer wieder deutlich die typische Java-Trägheit auszumachen und sind die Java Beans optisch sehr „Java-typisch“ gestaltet.

Da die Eigenschaften von Eclipse den Vorgaben für dieses Projekt besser entsprechen als jene der Java Beans, fiel hier die Wahl, parallel zur Entscheidung für SWT, auf Eclipse als Plattform für die neu zu entwickelnde Software. Ohnehin ist es ohne Inkaufnahme größerer Ineffizienzen bei beiden Alternativen nur möglich, für ein eigenes Projekt die jeweils bereits für die Plattform verwendete Grafiklösung zu verwenden. Damit lag es nahe, das Paar SWT/Eclipse zu wählen.

Aber wozu wird nun überhaupt eine „Plattform“ benötigt? Die beiden hier vorgestellten Plattformen bieten beide die für eine integrierte Entwicklungsumgebung benötigte Grundfunktionalität. Verwendet man also eine dieser Plattformen und integriert sein eigenes Projekt in diese Plattform, so steht dem Endanwender damit eine Vielzahl von Funktionen zur Verfügung, die nicht für das Projekt extra entwickelt werden müssen, da sie bereits in der Plattform enthalten sind. Auf diese Weise bleibt den Entwicklern von Werkzeugen erspart, beispielsweise eine eigene Dateiverwaltung, eigene suchen/ersetzen-Dialoge oder ein von Grund auf neu gebautes Menüsystem entwickeln zu müssen.

Um ein Programm in Eclipse zu integrieren, muss dieses lediglich als sogenanntes „plugin“ ausgelegt werden. Das bedeutet im Wesentlichen, dass der Eclipse-Plattform bestimmte Dinge mitzuteilen sind, und auf den Aufruf bestimmter Rückruf-Funktionen zu warten ist. Das Programm, zum Beispiel unser Editor, läuft dann im Rahmen von Eclipse und fügt seine eigene Funktionalität zur Grundfunktionalität der Eclipse-Plattform hinzu.

Somit ist dadurch, dass das hier vorgestellte Projekt auf der Eclipse-Plattform aufbaut, bereits von Beginn an eine bewährte und schlüssige Entwicklungsumgebung als Grundlage des zu

[4] Zumeist handelt es sich hierbei um Erweiterung zum Einsatz im Internet, beispielsweise bei der von IBM verkauften Software „Web Sphere“.

erstellenden Editors gegeben. Zur Erstellung des Editors genügt es nun im Wesentlichen, die eigentliche Editier-Funktionalität zu entwickeln. Da ein solches Vorgehen also sowohl bewährtes Wissen in das Projekt einbringt als auch zu erheblicher Zeitersparnis führen kann, wurde beschlossen, tatsächlich auf der Eclipse-Plattform aufzubauen. Als ein weiterer in die Zukunft gerichteter Grund hierfür wurde betrachtet, dass, wenn auch andere, zukünftige Projekte bei Texas Instruments in diese Plattform integriert werden, automatisch eine einheitliche Benutzeroberfläche für alle Projekte gewährleistet ist. Diese sind damit von vorn herein zu einem gewissen Grade untereinander integriert.

5.2.3 Parser

Es ist im Falle des vorliegenden Projekts von geringer Bedeutung, welcher Parser zum Einlesen der XML-Daten und zur Erstellung des DOM-Baumes zum Einsatz kommt. Alle gängigen Parser sind in der Lage, bei akzeptablem Ressourcenverbrauch zuverlässige und verwendbare Ergebnisse zu erzielen.

Allerdings gibt es verschiedene Möglichkeiten, die Verwendung des Parsers wie auch des generierten Ergebnisses zu vereinfachen. Eine Technik, die einfache und einheitliche Handhabung unterschiedlicher Parser zu ermöglichen, JAXP, wurde bereits vorgestellt. Es handelt sich hierbei um eine Sammlung von Java-Klassen, die einige bekannte Parser, sowohl für DOM als auch für SAX sozusagen umschließen, und deren Funktionalität mit Hilfe einer für alle unterstützten Parser einheitlichen Schnittstelle verfügbar machen. Um einen möglichst generischen Programmierstil zu erreichen, ist die Verwendung von JAXP in jedem Fall sinnvoll, wenn wie hier Java als Programmiersprache und DOM zur Abbildung der XML-Daten zum Einsatz kommen sollen.

Jedoch gibt es einige Erweiterungen und Vereinfachungen für die Baumstruktur, die möglicherweise der Verwendung des DOM Standardmodells vorzuziehen sind. Diese Vereinfachungen zielen im Falle von Java im Wesentlichen auf die Form der Repräsentation der Knoten im Baum selbst ab. Da Java eine objektorientierte Sprache ist, die bereits eine große Zahl von Methoden zur Manipulation der in ihr enthaltenen Objekte mitbringt, scheint es logisch, diese bereits existierende Funktionalität auch für die Objekte des XML-Baums zu verwenden. So erlaubt zum Beispiel JDOM, ein prominenter Vertreter unter den Implementierungen dieses Konzepts, Objekte mit Mitteln der Sprache Java zu kopieren, zu vergleichen, oder auch in Standard-Java-Listen zu speichern und zu manipulieren. Jedes Objekt im Baum enthält eine Liste vom Typ „Vector“, in der die Unterelemente des Objekts in der Hierarchie der Baumstruktur enthalten sind. Im Gegensatz dazu ist es bei der Implementierung des DOM notwendig, alle benötigten Zugriffsmethoden und Datenstrukturen nachzubilden, da die bereits in Java enthaltenen natürlicherweise nicht mit den Definitionen des DOM – das nicht speziell für eine einzelne Programmiersprache entwickelt wurde – übereinstimmen.

Mit dem hier beschriebenen Verfahren kann ein XML-Baum mit sehr viel geringerem Aufwand erstellt und bearbeitet werden als unter Einhaltung der Definition des originalen DOM. Sowohl Speicherbedarf als auch die benötigte Rechenleistung sind in der Regel deutlich geringer als bei

Implementierungen des DOM. Jedoch werden derzeit Konzepte entwickelt, die noch weiter gehen, und auch die inhaltliche Überprüfung der Daten gleich automatisch mit übernehmen.

Ein bei Erstellung dieser Arbeit bereits verfügbares solches System ist „Java Architecture for XML Binding“ (JAXB). Daten werden hier in den passenden, von Java bereitgestellten Datentypen gespeichert. Ist der Wert eines XML-Elements als vom Typ einer Ganzzahl definiert, so enthält das Objekt, welches im XML-Baum das entsprechende Element repräsentiert, also eine Variable vom Typ „Integer“. Es ist also ohne weiteres gar nicht mehr möglich, fehlerhafte Eingaben zu machen. Jedes für die XML-Daten definierte Element wird durch eine eigene Klasse abgebildet, und solange diese in korrekter Weise ineinander geschachtelt werden – Unterelemente stehen auch hier wieder in gewöhnlichen Java-Listen –, ist das daraus erzeugte XML-Dokument in Ordnung. Um eine solche enge Verknüpfung zwischen Java-Klassen und XML-Elementen zu ermöglichen, wird in der hier besprochenen Version von JAXB[5] ein eigenes spezifisches Schema-Dokument verwendet. In diesem vom Benutzer selbst zu erstellenden Dokument sind Daten zur Definition von – unter anderem – Datentyp und Namen jedes einzelnen Struktur-Elements enthalten.

Da dieses Konzept recht vielversprechend und sinnvoll klingt, wurden durch einen der Mitarbeiter, mit dem ich im Rahmen des Gesamtprojekts zusammenarbeite, im Vorfeld bereits intensive Versuche zum Einsatz von JAXB durchgeführt (siehe [16; 17]). Dabei erwies sich das System als im Wesentlichen stabil und einsetzbar. Allerdings kamen auch einige Probleme grundsätzlicher Art zum Vorschein. Zunächst einmal ist es weder angenehm noch wartungsfreundlich, die Daten zur Definition der XML-Struktur in zwei unterschiedlichen Dateien pflegen zu müssen. Da JAXB aber sowohl eine gewöhnliche DTD als auch sein spezielles Schema-Dokument für die vollständige Funktion benötigt, und in beiden Dokumenten jeweils ein Eintrag pro Datenelement enthalten ist, müssen bei Änderungen und Anpassungen auch beide Dokumente bearbeitet werden. Zukünftiger Arbeitsaufwand war bei der Verwendung von JAXB zudem dadurch abzusehen, dass für den Übergang von der damals verfügbaren Version zur ersten öffentlichen Beta-Version eine große Zahl an Veränderungen angekündigt war. Darunter befanden sich auch solch elementare wie der Ersatz des DTD-Formats durch das XML-Schema-Format[6]. Dies würde größere Umbaumaßnahmen beim Übergang auf die neue Version implizieren, der notwendig wäre, da die Arbeit mit einer nicht offiziellen Vorabversion auf Dauer nicht angebracht ist.

Hinzu kommen noch einige Probleme, die sich schlicht aus fehlender Funktionalität ergeben. So existiert zwar eine Funktion „verify“, welche die Gültigkeit der aktuellen Baumstruktur gegenüber dem zugehörigen DTD überprüft. Die einzige Rückmeldung dieser Funktion ist allerdings die Aussage darüber, ob der gesamte Baum in Ordnung ist oder nicht. Dies ist im Fehlerfall aus offensichtlichen Gründen völlig nutzlos. Es ist für das ausführende Programm weder möglich, irgendwelche automatischen Korrekturen vorzunehmen, noch, den Benutzer mit einer aussagekräftigen Fehlermeldung zu unterstützen.

[5] „Version 1.0 Early-Access Release, Specification Version 0.21“, vom 30. 05. 2001

[6] Inzwischen sind tatsächlich neue Versionen von JAXB verfügbar, welche die gehegten Befürchtungen in vollem Maße bestätigen.

Ein weiteres prinzipielles Problem bei der Verwendung von JAXB ergibt sich gerade aus der Besonderheit, dass XML-Elemente mittels einer jeweils eigenen Klasse abgebildet werden. Somit ist es bei der Arbeit mit der erstellten Baumstruktur notwendig, den grundsätzlichen Aufbau dieser Struktur bereits zu kennen. Selbst, wenn man zur Ermittlung der Art der jeweiligen Unterelemente Java-Sprachelemente (`instanceof`, ...) verwendet, muss noch auf jedes ermittelte Element in spezieller Weise eingegangen werden. Wohl ist es in Java auch möglich, die Art und Anzahl der in einem Objekt enthaltenen Variablen zur Laufzeit zu bestimmen, doch führt diese Vorgehensweise schnell zu einer sehr unübersichtlichen Vermischung von Java-Struktur und Programmlogik. Das ausführende Programm wird praktisch gezwungen, seinen eigenen Code zur Laufzeit zu analysieren, was sehr unübersichtlich und verwirrend werden kann. Und ohnehin kann auch auf diese Weise nicht alles erreicht werden, was für eine von einer festgelegten XML-Struktur unabhängigen Implementierung benötigt wird. Es ist beispielsweise nicht ermittelbar, ob es sich bei einem Element um ein optionales Element handelt. Und es ist auch nicht möglich, nur mit diesem Hilfsmittel zu erkennen, welche Art von Elementen als Unterelemente eines Knotens im Baum erlaubt sind. Die Entwicklung eines einigermaßen generischen Programms, das nicht jede Einzelheit der Struktur des verwendeten DTD bereits fest einprogrammiert kennt, erscheint somit beinahe aussichtslos.

Da die Verwendung von JAXB also einige unangenehme Implikationen mit sich bringt, und insbesondere die Gefahr besteht, dass die Struktur des Gesamtprojekts dadurch sehr statisch und unflexibel wird, wurde nach längerer Diskussion und eingehenden Erörterungen entschieden, dieses zunächst nicht einzusetzen. Auch von möglichen Kompromisslösungen wie dem bereits erwähnten JDOM wurde abgesehen, da diese zum Zeitpunkt der Entscheidung weder in ihrer Entwicklung besonders weit fortgeschritten oder ausgereift waren, noch irgendwie standardisiert und damit zukunftssicher[7]. Weiterhin besteht auch für Lösungen wie JDOM in der Regel keine vernünftige Möglichkeit zur Überprüfung einer bestehenden Baumstruktur, wofür unter DOM einige vorgefertigte Möglichkeiten angeboten werden. Auf der anderen Seite fällt der erhöhte Ressourcenverbrauch einer standardisierten DOM Lösung gegenüber speziell auf Java zugeschnittenen Abwandlungen für solch kleine Strukturbäume, wie sie bei der Generierung aus den SAP/AIM-Dokumenten zu erwarten sind, nicht weiter ins Gewicht. Dies wird im Vergleich zu den anderen Teilbereichen des Projekts besonders deutlich. Der Ressourcenverbrauch des Daten-Repository und der grafischen Oberfläche dürfte den des XML-Baums in der Regel zu einer zu vernachlässigenden Größe machen. Die aufgrund der nicht an Java angepassten Schnittstellen und der Strategie des DOM, alle Daten präzise abzubilden, etwas kompliziertere Handhabung soll durch die Kapselung des Zugriffs auf die Knoten des Baums in einer Java-Klasse abgefedert werden.

Mit diesen Überlegungen stand die Entscheidung für JAXP fest. Sollte sich zu einem späteren Zeitpunkt ein – möglicherweise neu- oder weiterentwickeltes – anderes System als überlegen erweisen, so ist eine Umstellung aufgrund der Kapselung der relevanten Elemente nicht weiter schwierig.

[7] Gerade JDOM war zudem bereits seit längerem nicht mehr offiziell weiterentwickelt worden.

Zur Gültigkeitsüberprüfung der XML-Daten ist dem System der XML-Schemata gegenüber DTDs der Vorzug zu geben. Diese bieten zum einen präzisere Möglichkeiten zur Definition des erlaubten Inhalts für einzelne Elemente, wodurch sie die entsprechende Funktionalität des JAXB eigenen speziellen Schemas übernehmen. Zum anderen sind sie einfach mit einem ohnehin verwendeten XML-Parser einlesbar und können somit ebenfalls unter Zuhilfenahme von JAXP ausgewertet werden. Dadurch ist es möglich, auf elegante Weise erlaube Unterelemente, Attribute und Kardinalitäten von Unterelementen eines Elements in der Baumstruktur zu ermitteln. Da allerdings in der aktuellen Version von Suns Java-Umgebung die Gültigkeitsüberprüfung durch den XML-Parser mittels XML-Schemata noch nicht implementiert ist, sollen zunächst auch DTDs gepflegt werden. Es wäre möglich, durch Hinzufügen entsprechender Bibliotheken die Fähigkeiten des Standard-Parsers zu erweitern, doch macht dies eine fehleranfällige manuelle Installation durch den Benutzer notwendig, die zudem zu Inkompatibilitäten mit anderen Programmen führen könnte. Die gleichzeitige Pflege eines DTD- sowie inhaltsgleichen XML-Schema-Dokuments für eine gewisse Zeit erscheint demgegenüber akzeptabel, da es mittels frei verfügbarer Werkzeuge unkompliziert und voll automatisiert möglich ist, aus einer DTD ein XML-Schema zu generieren.

5.3 Die getroffene Auswahl im Überblick

In den vorangegangenen Abschnitten wurden ausführlich die während der Planungsphase des Gesamtprojekts erwogenen Möglichkeiten für ein neues Datenformat und die zu erstellende Software diskutiert. Die getroffenen Entscheidungen wurden analysiert und begründet. An dieser Stelle folgt nun noch einmal eine kurze Zusammenfassung, die zugleich die Konsistenz der einzelnen Teile der Gesamtauswahl untereinander aufzeigt.

Die grundlegenden Entscheidungen betreffen die Wahl der zu verwendenden Programmiersprache sowie der einem neu zu definierenden Datenformat zugrundeliegenden Struktur. Als Grundlage des Datenformats wurde XML ausgewählt. Damit wird zum einen eine gewisse prinzipielle Robustheit gewährleistet und zum anderen die Verwendung einer Vielzahl existierender und bewährter Hilfsmittel bei der Bearbeitung ermöglicht. Als Programmiersprache wurde Java gewählt. Dieses weist keine Nachteile auf, die es für die Verwirklichung des Projekts ungeeignet machen würden, und es wird von entscheidenden Stellen bei Texas Instruments für die Zukunft favorisiert. Um die Erstellung der für den neuen Daten-Editor benötigten grafischen Benutzeroberfläche zu beschleunigen und um auch an dieser Stelle möglichst viel bereits bewährte Technologie einsetzen und dem Anwender verfügbar machen zu können, wurde der Einsatz der Eclipse-Plattform als Basis für das Projekt beschlossen. Mit dieser Plattform kommt zugleich die SWT-Grafikbibliothek, die aufgrund ihrer relativ guten Performance und des „native look and feel“ von mit ihrer Hilfe erstellten Oberflächen gut für die Verwendung im vorliegenden Projekt geeignet ist. Zur Bearbeitung der XML-Daten selbst schlussendlich soll das DOM zum Einsatz kommen. Dieses Modell zur Abbildung von XML-Daten in einer Baumstruktur lässt ein komfortables Lesen, Bearbeiten und Schreiben der Daten zu und ist standardisiert und bewährt.

Als Schnittstelle zur Implementierung des DOM wird JAXP verwendet, um in der Wahl der Implementierung und des zugrundeliegenden Parsers nicht festgelegt zu sein. Zunächst wird hier der standardmäßig in Suns Java-Umgebung enthaltene Parser verwendet um dem Benutzer des fertigen Programms Umbauten an seinem Arbeitsplatz-System zu ersparen.

Diese Bestandteile können zu einem kongruenten Ganzen zusammengefügt werden. Wie die Planungen hierfür aussahen und wie die fertige Implementierung funktioniert, wird im folgenden Kapitel ausführlich und in größerer Tiefe erläutert und analysiert.

6 Planung und Realisierung

In den vorangegangenen Kapiteln wurden die Ziele für eine neue Methode der Arbeit mit SAP/AIM-Daten abgesteckt. Es wurden die Grundlagen der verwendeten Techniken erklärt, verschiedene Alternativen für die Implementierung aufgezeigt und die getroffene Auswahl begründet. Damit ist die notwendige Vorarbeit getan, und wir können uns endlich der eigentlichen Implementierung zuwenden. In diesem Kapitel wird zunächst auf die hierfür notwendige Planung eingegangen. Darauf folgt eine kurze Beschreibung der Zwischenschritte auf dem Weg zur fertigen Implementierung sowie zu Entstehung und Form der DTD- und XML-Schema Dokumente. Abschließend wird näher auf Test und weitere Betreuung des erstellten Gesamtsystems eingegangen. Zusammengenommen repräsentieren die in diesem Kapitel behandelten Themengebiete diejenigen Maßnahmen, die der eigentlichen Softwareentwicklung vorausgehen, beziehungsweise diese unterstützen und damit das Gesamtprojekt unter Gewährleistung von Einsetzbarkeit und Qualität zum gewünschten Ziel führen.

6.1 Planung

Zur erfolgreichen Verwirklichung eines Projekts gehört immer auch ein gewisses Maß an Planung. Da sowohl der Zeitrahmen im Zuge der Diplomarbeit bereits festgelegt war, als auch das geforderte Ergebnis, war im Wesentlichen eine geeignete Aufteilung der verfügbaren Zeit notwendig. Da das Gesamtprojekt in der vorgegebenen Zeit von einer Einzelperson nicht zu bewältigen ist, musste zudem entschieden werden, in welcher Weise die anstehenden Arbeitspakete auf die zur Verfügung stehenden Mitarbeiter verteilt werden sollten. Die einzige Entscheidung, die bereits ganz zu Beginn der Planungsphase feststand, war diejenige über die Wahl des Namens für das neue Projekt: „SAPE“ – SAP Editor.

6.1.1 Abzubildende Prozesse

Für meine Diplomarbeit war es notwendig, einen Teil des Gesamtprojekts zu finden, der auch für sich allein als unabhängiges Ganzes funktioniert und in sich schlüssige und klar darstellbare Funktionalität bietet. Hierfür schien der neu zu erstellende Editor am geeignetsten. Zu diesem Editor gehören als integrale Bestandteile auf der einen Seite das XML-Daten-Repository und auf der anderen Seite die Arbeit mit den XML-Daten selbst, also Parsen, Auswerten, Bearbeiten und Schreiben von XML-Daten.

Andere notwendige Elemente des neuen Konzepts sind verschiedene Werkzeuge zum Umwandeln von Daten im XML-Format in ein anderes benötigtes Format. Hierbei sind drei Ziele zu erreichen. Zum einen soll aus den XML-Daten menschenlesbare Dokumentation entstehen, die online – zum Beispiel in Form von HTML – betrachtet werden kann. Weiterhin soll eine druckbare Form der Dokumentation bereitgestellt werden. Und, für die Funktion des Systems am Wichtigsten, es muss eine Möglichkeit zur Umwandlung von XML in das pdf/mdf-Format geschaffen werden, um die Weiterverarbeitung der Daten durch die bestehende Toolkette zu ermöglichen. Diese drei Teilaufgaben beziehen sich alle auf die Konvertierung vom XML-Format in ein anderes Format. Um dies auf einfachem Wege zu erreichen, ist XSLT prädestiniert[1]. Da sich hieraus also ebenfalls eine sinnvolle Einheit für die Entwicklungsphase ergibt, wurde die Arbeit mit XSLT als ein Block auf einen Entwickler übertragen.

Ein weiterer Entwickler wurde mit der Aufgabe der Umwandlung des alten Word Dokument-Formats in das neue XML-Format betraut. Hierfür wurde derjenige Entwickler ausgewählt, der bereits bisher die automatische Auswertung der in Word erstellten SAP/AIM-Dokumente betreut hatte. Um das gesteckte Ziel zu erreichen, wird auch für diese Umwandlung wieder zuerst gewöhnlicher ASCII-Text aus dem Dokument erzeugt, der dann geparst und in entsprechende XML-Daten umgewandelt wird. Dies muss für jedes existierende Dokument nur einmal geschehen und wird – obgleich so weit wie möglich automatisiert – aufgrund teilweise mangelnder Definition des Word-Datenformats sowie nicht immer ganz korrekter Daten nicht ohne ein gewisses Maß an Handarbeit ablaufen können. Diese sollen die bisher mit den bestehenden Word-Dokumenten befassten Entwickler jeweils selbst leisten.

Die Entwicklung eines neuen XML-Datenformats steht zwischen all diesen Teilaufgaben. Das neue Format bildet sozusagen die Schnittstelle zwischen den verschiedenen neu zu entwickelnden Elementen. Zur ordnungsgemäßen und reibungslosen Funktion des Gesamtprojekts ist eine möglichst schlüssige und sinnvolle Formulierung der neuen Grammatik von großer Bedeutung. Daher wurde entschieden, diese Aufgabe gemeinsam anzugehen. Das entwickelte Konzept sollte von möglichst vielen Personen analysiert, überprüft und verbessert werden, um Fehler und Ungereimtheiten im letztlich verwendeten Format möglichst auszuschließen. Die Aufgabe der Erstellung und Wartung des benötigten XML-Schemas wiederum wurde allein mir übertragen.

6.1.2 Meilensteine

Es ist nicht einfach, ein derart umfangreiches Projekt binnen sechs Monaten von Grund auf neu zu entwickeln. Zumal in diesem Falle keine Erfahrungen im Umgang mit der zu bearbeitenden Materie existierten. Um unter diesen Bedingungen dennoch ein ordentliches Ergebnis zu erzielen, wurden die einzelnen Aufgaben auf drei Abschnitte von jeweils zwei Monaten Dauer verteilt.

Das erste Drittel der Zeit wurde dem Sammeln von Erfahrungen und der Festlegung der Vorgehensweise zur Erreichung einer verwendbaren Version der zu entwickelnden Software gewidmet.

[1] Mittels einiger einfacher Regeln kann das XML-Format mit XSLT aus und in praktisch jedes beliebige denkbare Format umgewandelt werden.

In diese Zeit fiel ebenfalls die konzertierte Ausarbeitung der Definition des neuen Datenformats in Form von DTD. Dieses wurde von einigen Mitgliedern der Tools-Projektgruppe auf Konsistenz und Sinnhaftigkeit überprüft und, nach einigen Umarbeitungen, bis auf wenige Verfeinerungen und Ergänzungen in der heutigen Form festgelegt. Um die Möglichkeiten der gewählten Hilfsmittel kennen zu lernen und den Umgang mit ihnen zu erlernen, wurde ein erster Prototyp für einen Editor entwickelt.

Dieser Editor-Prototyp beinhaltet noch keine für SAP/AIM-Dokumente spezialisierte Funktionalität und ist – mit gewissen funktionalen Einschränkungen – als generischer Editor für XML ausgelegt. XML-Daten können damit eingelesen werden und werden in einer Baumstruktur dargestellt. Anhand des zugehörigen XML-Schema-Dokuments werden mögliche Unterelemente eines ausgewählten Elements dargestellt. Weiterhin ist es möglich, Elemente zum Baum hinzuzufügen und die Baumstruktur zurück in eine XML-Datei zu schreiben. Mögliche Fehler im resultierenden XML-Dokument werden in einer Liste dargestellt. Die grafische Benutzerschnittstelle des Prototyps ist in Abbildung 6.1 auf der folgenden Seite dargestellt.

Durch die Implementierung dieser Funktionalität konnte sowohl eine gewisse Souveränität im Umgang mit SWT, Eclipse und DOM erlangt als auch die prinzipielle Machbarkeit der gesamten Unternehmung mit den gewählten Mitteln gezeigt werden. Nach seiner Fertigstellung im eben umrissenen Rahmen wurde der Prototyp einem geschlossenen Publikum – im Wesentlichen die ebenfalls mit dem Projekt befassten Entwickler sowie der Leiter des Projekts – vorgestellt. Daraus wurden Anforderungen für einen zweiten, spezifischen Prototypen und für die Erarbeitung des fertigen Editors abgeleitet.

Die folgenden beiden Monate dienten der Erstellung des zweiten Prototypen und des daraus abgeleiteten Endprodukts. Binnen zweier Wochen war ein in groben Zügen funktionsfähiger Prototyp implementiert. Dieser Prototyp bietet neben der Baumstruktur speziell auf die einzelnen Elemente eines SAP-Dokuments zugeschnittene Eingabemasken an, ist aber ansonsten in seiner Funktionalität auf das für die Vorstellung im Rahmen einer offiziellen firmeninternen Präsentation notwendige Maß beschränkt.

Zu dieser Präsentation wurden eine Reihe von Projektleitern und sonstigen Entscheidungsträgern aus den Reihen von Texas Instruments Berlin eingeladen. Der im Rahmen dieser Präsentation vorgestellte zweite Prototyp für den Editor fand großen Anklang beim Publikum und es wurden einhellig die Fortsetzung des Projekts und der praktische Einsatz im Unternehmen befürwortet. Auch über die weitere Fortführung des Projekts nach Abschluss der Diplomarbeit bestand im Ergebnis dieser Präsentation bereits Einvernehmen. Aus dem Prototypen abgeleitete Wünsche für das Endergebnis wurden in die weiteren Planungen aufgenommen.

In den verbliebenen sechs Wochen wurden Editor, Daten-Repository und XML-Manager implementiert. Dies bereitete keine prinzipiellen Probleme und ging aufgrund der intensiven Vorbereitungsphase gut und schnell von der Hand. Wo immer möglich wurden Elemente der bereits existierenden Prototypen weiterverwendet und auf in der Eclipse-Plattform verfügbare Funktio-

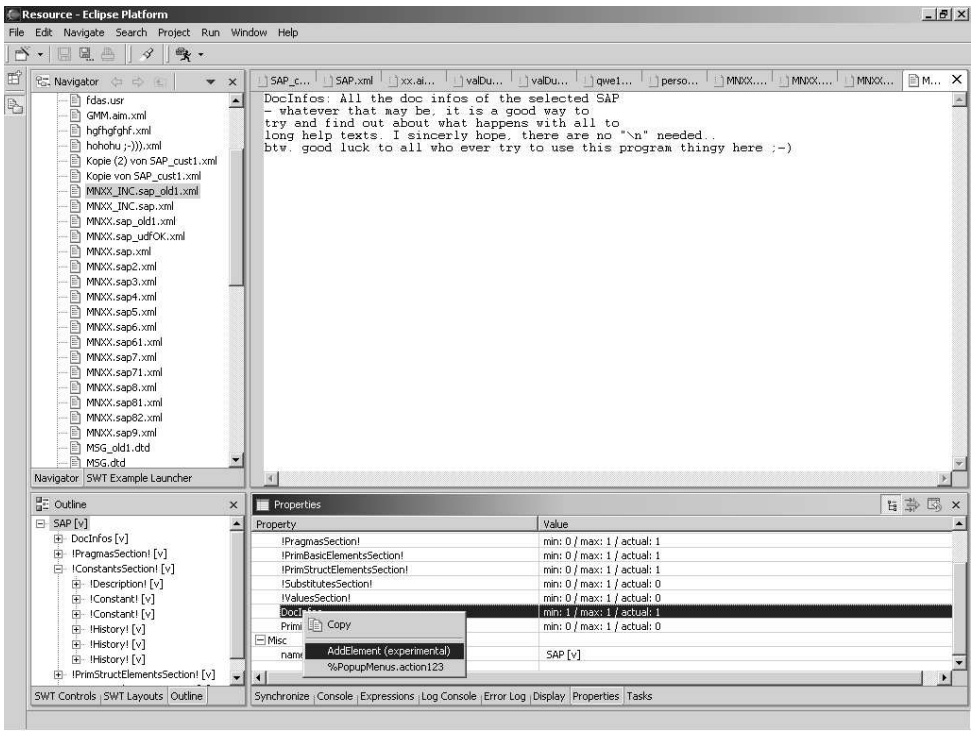

Abb. 6.1: Der erste Prototyp

nalitäten zurückgegriffen. Auf diese Weise konnte binnen kurzer Zeit ein großer Funktionsumfang zur Verfügung gestellt werden, ohne die Stabilität des Gesamtprodukts nachhaltig zu gefährden.

Die letzten zwei Monate wurden der Nachführung der Benutzerdokumentation, dem Test der erstellten Software sowie letzten Detailverbesserungen gewidmet. Auch die Erstellung des schriftlichen Teils dieser Diplomarbeit fällt in diesen Zeitraum. Tests wurden hauptsächlich von jenem Entwickler durchgeführt, der auch mit den XSLT-Transformationen befasst war. Damit war er eine sehr gute Besetzung zum Test der Software, da er einerseits Details der Implementierung nicht kannte, andererseits aber genauestens über die zu erwartenden Ergebnisse und Effekte bezüglich der bearbeiteten Daten informiert war.

Als Ergebnis der Tests wurde die prinzipielle Einsatzbereitschaft der erstellten Software festgestellt. Aus den Tests resultierende Vorschläge für zukünftige Erweiterungen und Verbesserungen wurden einer bereits bestehenden bei der Entwicklung selbst erstellten Liste entsprechender Anregungen zugefügt. Diese Liste wird, nachdem eine grobe Sortierung nach Relevanz und Dringlichkeit durchgeführt wurde, im Anschluss an die vorliegende Diplomarbeit von mir Stück für Stück abgearbeitet werden. dass dabei bis auf weiteres kein Ende abzusehen ist, versteht sich von selbst...

Zusätzlich sind für die Zeit im Anschluss an diese Arbeit eine weitere offizielle Präsentation des erreichten Standes sowie die allgemeine Einführung des neuen Formats und damit des erstellten Editors bei Texas Instruments vorgesehen.

Die zeitlichen Vorgaben für die nicht von mir zu bearbeitenden Teile des Gesamtprojektes wurden so gewählt, dass diese ebenfalls zum Ende des Zeitraums von sechs Monaten einsatzbereit sind.

6.2 Erstellung von DTDs und XML-Schemata

Die Hauptarbeit bei der Erstellung der DTDs wurde, wie bereits erwähnt, von einem ebenfalls mit dem hier vorgestellten Projekt betrauten Entwickler geleistet. Da ich jedoch eng in diese Entwicklung eingebunden war, und mit meiner eigenen Aufgabe zugleich auch einer der Hauptbetroffenen, erscheint es angebracht, die Entwicklung und Funktion der vorliegenden DTDs näher zu dokumentieren. Zudem habe ich alle Entwicklungen im Bereich der XML-Schema Dokumente alleine durchgeführt, was ohnehin auch eine Beschreibung der zu Grunde liegenden DTDs notwendig macht.

6.2.1 DTDs

Da sich das Format von SAP-Dokumenten schon aufgrund des unterschiedlichen Inhalts von dem von AIM-Dokumenten unterscheidet, ist es notwendig, für beide Formate jeweils eine eigene DTD bereitzustellen. Da jedoch in beiden Dokumenttypen auch viele gemeinsame Strukturen auftauchen, ist darauf zu achten, dass diese jeweils so weit wie möglich übereinstimmen. Um die Handhabung identischer Elemente zu vereinfachen und unnötige Redundanzen zu vermeiden, werden in einer dritten DTD alle gemeinsamen Elemente gesammelt. Diese dritte DTD ist in die anderen beiden eingebunden. Dies geschieht in den DTDs jeweils mittels der folgenden Anweisungen:

```
<!--SAP/MSG common definitions can be found in SAPMSGcommon.dtd-->
<!ENTITY % SAPMSGcommon SYSTEM "SAPMSGcommon.dtd">
<!--Include SAP/MSG common elements-->
%SAPMSGcommon;
```

Hierbei definiert die „ENTITY“-Anweisung ein neues Alias, das in der darauf folgenden Anweisung sogleich verwendet wird. Damit wird an dieser Stelle der Inhalt der Datei „SAPMSGcommon.dtd“ eingebunden und bei Verwendung der DTD mit berücksichtigt[2].

Die einzelnen in den drei resultierenden DTDs enthaltenen Elemente orientieren sich eng an den Elementen, welche bereits im bisherigen Format von AIM-Dokumenten definiert sind. Dies bedeutet im Falle der SAP-Dokumente einige Änderungen, die aber wenig kompliziert und zumeist offensichtlich in ihrer Umsetzung sind. Das rührt daher, dass sich trotz der bisher unterschiedlichen Formate die Art der in beiden Dokumenttypen enthaltenen Daten nicht wesentlich unterscheidet. So verwundert es auch kaum, dass die für SAP- wie AIM-DTD eingebundene DTD

[2] Das Auftauchen der Abkürzung „MSG“ soll uns an dieser Stelle nicht weiter beunruhigen. Es handelt sich hierbei lediglich um eine weitere Form der Abkürzung für Air Interface Messages (AIM). In allen neueren Dokumenten wurde „AIM“ anstelle von „MSG“ verwendet.

die unfangreichste von den Dreien darstellt. Anhand dieser DTD soll hier der grundsätzliche Aufbau aller drei Dokumente vorgestellt werden.

Für jeden Typ von zu speichernder Information wird ein Abschnitt („Section“) bereitgehalten. Diese Abschnitte, beispielsweise die „DocInfoSection“ oder die „ConstantsSection“, enthalten die jeweils interessanten Daten. Im Falle des letztgenannten Abschnitts sieht dies dann so aus:

```
<!ELEMENT ConstantsSection (Description, Constant+, History+)>
```

Damit ist – im DTD-Format – festgelegt, dass ein gültiges Element vom Typ „ConstantsSection“ je ein Element vom Typ „Description“ sowie mindestens je ein Element der Typen „Constant“ und „History“ enthalten muss[3]. Diese Unterelemente sind ebenfalls in der DTD definiert, „Constant“ beispielsweise wie folgt:

```
<!ELEMENT Constant (Alias, (ItemLink | Value),
  Version?, Group?, Comment, Note?)>
```

Dessen optionales Unterelement „Version“ beispielsweise ist wie folgt definiert:

```
<!ELEMENT Version (#PCDATA)>
```

Damit bildet es einen Endpunkt in der Kette der Verschachtelungen, da seine Definition besagt, dass es lediglich ASCII-Text enthält. In ähnlicher Weise sind auch alle anderen Elemente der SAP/AIM DTD-Dokumente definiert.

Für die SAP- und AIM-DTD ist jeweils ein Wurzelelement definiert. Im Falle des SAP trägt dieses – wenig überraschend – den Namen „SAP“. Es ist wie folgt definiert:

```
<!ELEMENT SAP (DocInfoSection, PragmasSection?, ConstantsSection?,
  PrimitivesSection?, FunctionsSection?, PrimStructElementsSection?,
  PrimBasicElementsSection?, SubstitutesSection?, ValuesSection?)>
```

Es sind also eine ganze Reihe von Abschnitten in einem SAP-Dokument möglich, die – bis auf die „DocInfoSection“, die Informationen zu einem Dokument als solchem enthält – alle optional sind, also weggelassen werden können, wenn keine entsprechenden Daten vorliegen.

Das Wurzelelement eines AIM-Dokuments, „MSG“, sieht dem eben vorgestellten sehr ähnlich:

```
<!ELEMENT MSG (DocInfoSection, PragmasSection?, ConstantsSection?,
  MessagesSection, MsgStructElementsSection?, MsgBasicElementsSection?,
  SubstitutesSection?, ValuesSection?)>
```

Die einzigen wirklichen Unterschiede zum Wurzelelement der SAP-DTD sind das Fehlen der „FunctionsSection“, die schlicht für AIM-Dokumente nicht gebraucht wird sowie das zwingende Erfordernis einer „MessagesSection“. Die letztere Vorgabe resultiert daraus, dass ein AIM-Dokument, das keine einzige Air Interface Message enthält, wenig sinnvoll ist[4].

[3] Nähre Informationen zur Funktionsweise von DTD sowie zur verwendeten Syntax sind im Kapitel „Grundlagen“ nachzulesen.

[4] Dagegen kann ein SAP-Dokument ohne Primitiven durchaus sinnvoll sein, wenn es stattdessen beispielsweise Funktionen enthält.

Die Parallelen zum an anderer Stelle vorgestellten alten AIM-Format sind bei beiden neuen Formaten klar zu erkennen. Es wurde im Grunde lediglich die textuell in [9; 10; 8] bereits existierende Definition für entsprechende Dokumente mit den formalen Mitteln der DTD nachgeführt. Im Laufe des Entwicklungsprozesses sowie im Zuge der durchgeführten Reviews[5] wurde das hieraus hervorgegangene Ergebnis vervollständigt und abgerundet, so dass nun – es steht zu hoffen – eine vollständige Grammatik für DTD-/AIM-Dokumente existiert.

6.2.2 Umwandlung in XML-Schema-Dokumente

Es ist ein Leichtes, aus den vorliegenden DTDs XML-Schema-Dokumente zu erzeugen. Die vorliegenden Schema-Dokumente wurden automatisch mit Hilfe des bei [6] abrufbaren frei verfügbaren Programms „dtd2xs“ generiert. Um eine möglichst einfache Handhabung zu gewährleisten, werden die gemeinsamen Definitionen einfach direkt in das SAP-Schema-Dokument übertragen. Dies verringert die Anzahl der zu verwaltenden Dateien und hat keinerlei Auswirkungen auf das Funktionieren des Gesamtprogramms, da XML-Schema-Dokumente derzeit nicht zur Gültigkeitsüberprüfung verwendet werden, sondern nur zum Ermitteln von zusätzlichen Informationen über Aufbau und Struktur der bearbeiteten Dokumente.

Sobald die gleichzeitige Wartung von DTD- und XML-Schema-Dokumenten nicht mehr notwendig ist – wenn also die Unterstützung für XML-Schemata in die Java Standardimplementierung von Sun integriert worden ist –, werden die DTD-Dateien verworfen und die XML-Schema-Dateien so behandelt, wie bisher die DTDs. Damit wird es, nach der heutigen Planung, schlussendlich also lediglich *drei* XML-Schema-Dokumente und überhaupt keine DTDs mehr geben.

Das Ergebnis der automatischen Umwandlung ist am anschaulichsten durch ein Beispiel zu demonstrieren:

```
<xs:element name="ConstantsSection">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Description"/>
      <xs:element ref="Constant" maxOccurs="unbounded"/>
      <xs:element ref="History" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Es werden – wie zu sehen ist – schlicht die Informationen aus dem DTD direkt in ein XML-Schema-Dokument übertragen.

[5] Begutachtungen und Stellungnahmen durch andere, nicht direkt mit der betroffenen Sache betraute Personen.

6.2.3 Benutzerdefinierte Modifikationen

Der vorgesehene Übergang auf die alleinige Verwendung von XML-Schema-Dokumenten ist schon deswegen erstrebenswert, da für Teile des Editors die Möglichkeit besteht, zusätzliche Informationen aus den entsprechenden Dokumenten zu beziehen. Es handelt sich hierbei derzeit nur um Informationen zur Darstellung der Elemente, jedoch sind auch Erweiterungen dieses Konzepts denkbar. Der Vorteil besteht darin, dass der Benutzer die Darstellung an seine persönlichen Bedürfnisse anpassen kann, ohne Veränderungen am Programm selbst vornehmen zu müssen. Jedoch ist dieses Konzept noch nicht wirklich sinnvoll zu benutzen, solange XML-Schema-Dokumente noch automatisch aus den entsprechenden DTDs generiert werden, da danach jeweils alle erfolgten Anpassungen am Schema-Dokument von Hand nachgeführt werden müssen. Dennoch wurde es in den – bis auf weiteres entgültigen – aktuellen XML-Schema-Dokumenten eingesetzt. Die Funktionsweise wird schnell am folgenden Beispiel deutlich:

```
<xs:element name="Primitive">
  <xs:complexType>
    <xs:annotation>
      <xs:appinfo>
        <replaceName PrimDef.Name=""/>
      </xs:appinfo>
    </xs:annotation>
    <xs:sequence>
      <xs:element ref="Description"/>
      <xs:element ref="PrimDef"/>
      [...]
```

Alle im Schema-Dokument enthaltenen Anweisungen an den Editor sind in sogenannten „annotation“ Elementen, und hierin wiederum im Unterelement „appinfo“ enthalten. Diese Art der Konstruktion ist in der offiziellen Spezifikation für XML-Schemata zu eben diesem Zweck vorgesehen. Derzeit existieren zwei Typen möglicher Unterelemente, „addName“ und „replaceName“, deren Attributnamen jeweils als Pfadangaben dienen. Diese Pfade beziehen sich relativ auf die Position des aktuellen Elements und weisen den Weg zu einem Datenelement, dessen Wert als Namen herangezogen wird. Dieser Name wird dann an bestimmten Stellen im Editor in Ergänzung („addName“) zum beziehungsweise anstatt („replaceName“) des eigentlichen Namens des Elements verwendet. Die einzelnen in der Pfadangabe vorkommenden Elemente sind voneinander jeweils durch einen Punkt getrennt.

6.3 Wartung und Test

Funktionale Tests der erstellten Software wurden von mir laufend während des Entwicklungsprozesses vorgenommen. Weiterhin wurde ständig der aktuelle Entwicklungsstand mit den zu

erfüllenden Anforderungen verglichen, um eventuelle Irrwege und Fehlentwicklungen frühzeitig zu erkennen und damit möglichst zu vermeiden. Die hierbei gefundenen Probleme und noch zu erledigenden Details wurden in eine Liste aufgenommen oder direkt gelöst.

Zu bestimmten Zeitpunkten während der Entwicklung, und insbesondere nach Fertigstellung der ersten tatsächlich verwendbaren Version des Gesamtsystems wurden, wie bereits erwähnt, ausführliche Tests durch einen außenstehenden Mitarbeiter durchgeführt. Diese beinhalteten, neben allgemeinen Funktionsprüfungen, die Erstellung eines kompletten SAP-Dokuments mit dem Editor sowie die Bearbeitung verschiedener aus dem MS-Word Format konvertierter SAP- und AIM-Dokumente. Des weiteren wurde erneut der aktuelle Ist-Zustand mit den für das Projekt festgelegten Anforderungen verglichen. Tests dieser Art werden weiterhin parallel zur laufenden Entwicklungsarbeit durchgeführt.

Die Gewährleistung der Qualität ist somit durch andauernde Tests, Präsentationen und - im Falle der DTD - Reviews gegeben. Um das System nun auch tatsächlich für den Endanwender nutzbar zu machen, wurde eine geeignete Benutzerdokumentation entwickelt. Diese beinhaltet die Erklärung der grundlegenden Mechanismen des Editors und der mit ihm verbundenen Elemente sowie eine Beschreibung zum Umgang mit der grafischen Oberfläche. In einer deutschsprachigen Fassung wurde diese Dokumentation im Abschnitt „Bedienung“ auch in das hier vorliegende Dokument eingearbeitet.

Für die Zukunft ist die Abhaltung von Schulungen vorgesehen, welche die Benutzer an das neue Konzept heranführen und ihnen das für eine erfolgreiche Arbeit notwendige Wissen vermitteln sollen. Ein weiterer Zweck dieser Schulungen sowie der stattfindenden Präsentationen ist die Erhöhung der Akzeptanz durch die Anwender, da immerhin zu einem bestimmten Zeitpunkt das alte System komplett abgeschafft und für alle verbindlich durch das neu entwickelte ersetzt werden soll. Um dies möglichst unproblematisch zu gestalten und einen reibungslosen Betrieb zu gewährleisten, werden eine ständige Weiterentwicklung und Verbesserung der Software sowie weitere geeignete Tests stattfinden. Letzteres ist in größerem Umfange möglich, sobald alle umgebenden Bestandteile des neuen Konzepts in ausreichendem Maße funktionieren. Dies betrifft insbesondere die Konverter für das neue XML-Format aus dem MS-Word Format sowie in das noch verwendete pdf/mdf-Format.

7 Funktionsweise der Software

Die im Rahmen des vorliegenden Projekts entwickelte Software gliedert sich im Wesentlichen in drei Bestandteile. Eine Einheit zur Bearbeitung, zum Lesen und zum Schreiben von Daten im XML-Format, ein Daten-Repository zur Verwaltung von Referenzen innerhalb von Dokumenten sowie über Dokumentgrenzen hinweg und ein Editor, der mit Hilfe einer grafischen Benutzeroberfläche dem Anwender in einfacher und strukturierter Form die Bearbeitung der zugrundeliegenden Daten erlaubt. Die Zusammenhänge zwischen den einzelnen Modulen sind aus Abbildung 7.1 auf der nächsten Seite zu entnehmen. Der Editor ist hierbei das bei weitem umfangreichste und komplexeste Teilprojekt, das zudem eng mit der verwendeten Eclipse-Plattform verwoben ist, weswegen seine Beschreibung in diesem Dokument naturgemäß den größten Raum einnimmt.

Neben diesen Hauptbestandteilen existieren noch einige Hilfseinheiten, beispielsweise zur Verwaltung von Listen oder zum Lesen und Schreiben festgelegter Elemente in XML-Strukturen. Diese Teile werden ebenfalls bei der Vorstellung des Editors besprochen, da sie in diesem Zusammenhang ihre wesentliche Verwendung finden.

7.1 Bedienung

Um den Sinn der einzelnen Teilelemente sowie deren Zusammenspiel und Funktionsweise verständlicher zu machen, folgt hier zunächst eine kurze Einführung in die Bedienung des Gesamtpaketes aus Benutzersicht.

7.1.1 Installation

Die Installation des Systems unter Windows gestaltet sich denkbar unkompliziert. Es ist lediglich notwendig, die ausgelieferte Zip-Datei in ein beliebiges Verzeichnis zu entpacken. Das System, bestehend aus der Eclipse-Plattform sowie einigen Plugins – darunter auch der hier vorgestellte Editor – ist damit sofort lauffähig. Da es sich hierbei trotz Benutzung der Eclipse-Plattform noch immer um ein Java-Programm handelt, muss allerdings eine Java-Laufzeitumgebung installiert sein. Die vorliegende Programmversion 0.9 wurde mit Suns „Java Runtime Environment“ in der Version 1.4.1 entwickelt und getestet, das damit zugleich die Mindestanforderung darstellt. Um alles zusammen in akzeptabler Geschwindigkeit ausführen zu können, wird ein mindestens 300 MHz schnelles System mit 128 MB RAM-Speicher und 60 MB freiem Festplattenspeicher empfohlen.

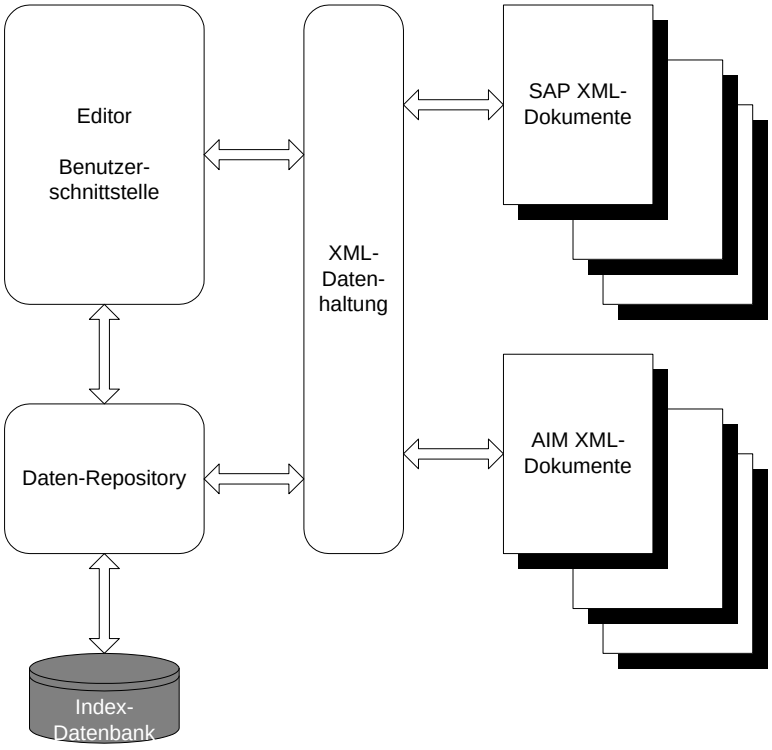

Abb. 7.1: Interner Aufbau des Projekts

7.1.2 Verwendung

Der hier vorgestellte Editor ermöglicht, zusammen mit den von der Eclipse-Plattform bereitgestellten Bedienelementen, die grafische Bearbeitung von im XML-Format vorliegenden SAP/AIM-Daten. Hierzu wurden für das neu eingeführte Plugin zwei grafische Elemente erstellt. Dabei handelt es sich zum einen um ein Fenster, in dem mit Hilfe von speziell angepassten Eingabemasken SAP- und AIM-Daten bearbeitet werden können. Zum anderen wurde ein weiteres Fenster erstellt, in dem die Struktur des gerade aktiven XML-Dokuments in Form eines Baums angezeigt wird. Dieses Fenster dient der Gesamtübersicht über ein Dokument sowie der Navigation innerhalb des Dokumentes. Wird ein Element der angezeigten Baumstruktur vom Anwender ausgewählt, so öffnet sich im anderen Fenster eine zu diesem Element passende Eingabemaske. Abbildung 7.2 auf der folgenden Seite zeigt eine Momentaufnahme aus der Arbeit mit dem Editor.

Die Eingabemasken zur Bearbeitung der Daten, die zu einem XML-Element gehören, bestehen aus mehreren Tabellen. Wo es nicht möglich oder sinnvoll ist, alle Tabellen auf einer Seite darzustellen, existieren zusätzlich ganz oben im Fenster einige Buttons[1], mit deren Hilfe zwischen den verschiedenen Seiten umgeschaltet werden kann, welche die einzelnen Tabellen beinhalten.

[1] „Knöpfe“, Schaltflächen – da das englische Wort hier sehr viel gebräuchlicher ist, werden an dieser Stelle ausnahmsweise wieder einmal alle möglichen deutschen Übersetzungen geflissentlich ignoriert...

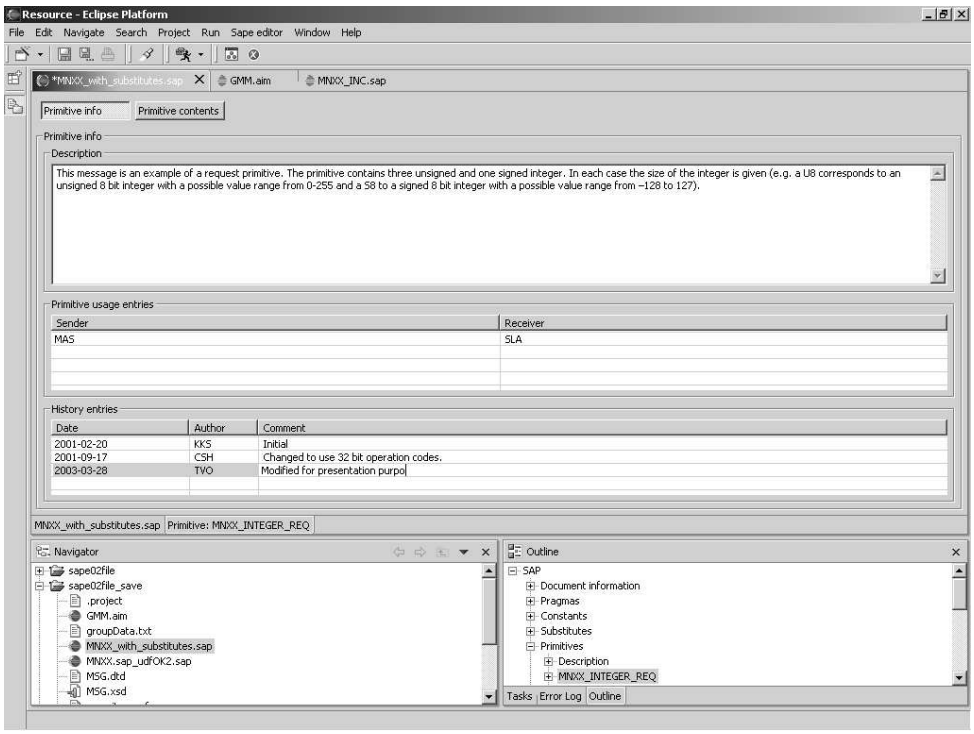

Abb. 7.2: Arbeit mit dem Editor

Ein Element in einer Tabelle kann bearbeitet werden, indem zuerst durch einfachen Mausklick auf die enthaltende Tabellenzeile diese Zeile ausgewählt wird, und dann durch nochmaligen einfachen Mausklick auf das Element dieses zur Bearbeitung ausgewählt wird. Je nach Art des Elements wird dann weiter verfahren. Hierbei existieren die folgenden drei Varianten:

- einfache Textelemente, deren Inhalt durch gewöhnliche Texteingabe verändert werden kann, und die daran zu erkennen sind, dass beim Editieren einfach nur der zum Element gehörende Text angezeigt wird;

- Tabellenelemente, die beim Editieren an einem kleinen Pfeil in der rechten Ecke des Tabellenfeldes zu erkennen sind, und für die nach einem Klick auf diesen Pfeil die in der daraufhin erscheinenden Auswahlliste aufgelisteten Zeichenketten als Werte für das bearbeitete Feld ausgewählt werden können;

- Verweise auf andere Elemente, bei denen statt eines Pfeils in der rechten Ecke ein kleiner grauer Button zu sehen ist, dessen Aktivierung die Öffnung einer Auswahlbox mit der baumförmigen Ansicht aller möglichen an dieser Stelle zu verwendenden Verknüpfungen bewirkt.

Ein Doppelklick auf die Elemente des letzteren Typs bewirkt den Sprung direkt zu dem Element, auf das in der zugehörigen Tabellenzeile verwiesen wird. Ist dieses in einem anderen Dokument zu

finden, so wird zunächst das entsprechende Dokument geöffnet und sodann darin das betreffende Element angezeigt.

Um eine Zeile in einer Tabelle zu löschen und damit die dieser zugrundeliegenden Daten, muss diese markiert und im zugehörigen Kontextmenü die Option „Delete element“ ausgewählt werden. Um dagegen ein neues Element in die Tabelle aufzunehmen, steht im selben Kontextmenü die Option „Add new element“ zur Verfügung. Bei Auswahl dieses Punktes wird in der Tabelle eine neue Zeile angelegt, deren Felder mit vorgegebenen Standardwerten initialisiert sind.

In manchen Tabellen, deren vollständige Darstellung zu viel Raum in Anspruch nehmen würde, sind Spalten mit nicht zwingend erforderlichen Daten zunächst ausgeblendet. Um solche Spalten ein- oder auszublenden, genügt ein Klick auf ihren jeweiligen Namen in der am rechten Rand der Tabelle befindlichen Liste, die alle betroffenen Spalten enthält.

Die beim Start angezeigt Seite, die zu den Wurzelelementen der XML-Struktur, den „SAP“- und „MSG“-Elementen gehört, hat ein eigenes Format. Dabei werden alle möglichen und tatsächlich existierenden Unterelemente, also die möglichen Abschnitte eines SAP- beziehungsweise AIM-Dokuments, in einer Liste untereinander angezeigt. Über dieser Liste befinden sich zwei Buttons, die sich jeweils auf das gerade in der Liste markierte Element beziehen. Deren einer, „Go into“, bewirkt den Sprung zu der Editor-Seite, die das Element vollständig anzeigt, mit dem anderen, „Add new element“, kann ein neues Unterelement des markierten Typs erzeugt werden, wenn nicht bereits eines existiert. Auf diese Weise können noch nicht vorhandene Abschnitte in ein Dokument integriert werden, beispielsweise die „PrimitivesSection“, wenn in einem neuen SAP-Dokument die erste Primitive angelegt werden soll.

Zur besseren Strukturierung des Arbeitsablaufs existieren zudem zwei Reihen von Karteireitern, eine am oberen Ende des Fensters und eine am unteren Ende. Die untere Reihe beinhaltet die gerade bearbeiteten Elemente des Dokuments, so dass auch mehrer Elemente parallel bearbeitet werden können. Dies ist besonders beim gerade angesprochenen Sprung zu einem referenzierten Element nützlich, um nicht die Übersicht zu verlieren. Geschlossen werden können diese Elemente wieder mit Hilfe des Menüpunkts „Close page“ im Menü „SAP editor“ oder durch Klick auf das kleine Kreuz in der Werkzeugleiste. Im Fenster, das die Baumstruktur des Dokuments enthält, ist immer das Element aktiviert, das gerade auch im anderen Fenster dargestellt wird. Wählt der Benutzer in einem der beiden Fenster ein neues Element aus, so wird dieses automatisch im jeweils anderen Fenster ebenfalls angezeigt.

Die obere Reihe von Karteireitern erlaubt den direkten Zugriff auf alle gerade geöffneten Dokumente. Jedes offene Dokument wird dabei von einer eigenen Instanz des Editors repräsentiert, zu der jeweils ein mit dem Dokumentnamen beschrifteter Karteireiter gehört. Wechseln zu einem neuen Dokument ist durch Klick auf die entsprechenden Karteireiter möglich.

Das Öffnen und Bearbeiten von Dokumenten geschieht im Explorer-Stil mit Hilfe des von der Eclipse-Plattform bereitgestellten „Navigator“-Plugins. In dessen Fenster werden in hierarchi-

scher Form die Dateien angezeigt, die zum aktuellen Projekt[2] gehören. Die Auswahl einer dieser Dateien durch Doppelklick führt – so es sich dabei um eine SAP- oder AIM-Datei handelt – zum Öffnen dieser Datei und zur Darstellung ihres Inhalts in den beiden anderen hier beschriebenen Fenstern.

Um die vorgenommenen Veränderungen auf syntaktische Richtigkeit zu überprüfen, steht im Menü „SAP editor“ der Menüpunkt „Verify xml document“ zur Verfügung. Die durch dessen Auswahl aktivierte Funktion überprüft den aktuellen Stand der XML-Daten daraufhin und stellt die Ergebnisse in einem Popup[3]-Fenster dar.

Gespeichert werden kann ein verändertes Dokument in gewohnter Weise durch Klick auf die entsprechenden Buttons in der Werkzeugleiste oder durch Auswahl der Menüpunkte „Save“ oder „Save as“ aus dem „File“-Menü. Dass ein Dokument gegenüber der gespeicherten Fassung verändert wurde, ist an einem kleinen Stern rechts von dessen Name im zugehörigen Karteireiter zu erkennen.

7.2 Konventionen

Vor der tatsächlichen Vorstellung des in Java programmierten Teils des Projektes sind einige grundlegende Dinge zu klären sowie einige Konventionen für die Darstellung von Code und Pseudocode zu vereinbaren. Dies soll im Folgenden geschehen.

Das gesamte Projekt einschließlich aller Dokumentation liegt in englischer Sprache vor. Dies ergibt sich daraus, dass diese Sprache die bei Texas Instruments als international operierendem Unternehmen einzige „offizielle“ Sprache darstellt. Da Dokumentation und Quellcode Mitarbeitern in Frankreich oder Amerika genauso zugänglich sein müssen, wie den Kollegen in Deutschland, bleibt auch vom praktischen Standpunkt her gar keine andere Wahl. Somit entsprechen die in diesem Dokument vorkommenden Auszüge aus dem Quellcode und der Dokumentation des Projekts nicht eins zu eins dem jeweiligen Original, sondern sind, wo dies angemessen ist, entsprechend in die deutsche Sprache übersetzt.

Klassendiagramme sind nur dort bei den jeweiligen Methoden dargestellt, wo sie auch aussagekräftig sind. Auf die grafische Darstellung einfacher linearer Vererbungen zwischen zwei oder drei Klassen wurde zumeist verzichtet. Die Notation der vorhandenen Klassendiagramme ist sehr einfach – jede beteiligte Klasse ist in Form eines Rechtecks dargestellt, das den Namen der Klasse enthält, und die Rechtecke sind untereinander durch Pfeile verbunden, die jeweils von den untergeordneten – abgeleiteten – Klassen zu den ihnen übergeordneten – vererbenden – Klassen zeigen. Auf die Darstellung von Methoden in den Klassendiagrammen wurde ebenfalls verzichtet, da dies die Übersichtlichkeit stark reduzieren würde, und aufgrund der oft systembedingt nicht allzu aussagekräftigen Methodennamen ohnehin nicht viel zum Verständnis einer Klasse

[2] Zur Arbeit mit Projekten in Eclipse und zu allen weiteren Fragen, die allgemein die Bedienung von Eclipse betreffen, sei hier auf die sehr umfangreiche Online-Hilfe verwiesen.

[3] Das Wort „Popup“ bedeutet in diesem Zusammenhang in etwa „Unabhängig vom Rest der grafischen Oberfläche neu im Vordergrund geöffnet“.

beiträgt. Die jeweils interessanten Methoden werden stattdessen explizit im Text erwähnt und zumeist auch näher beschrieben.

Pseudocode wird zur Beschreibung von komplexeren Algorithmen eingesetzt. Einfache Beispiele und Strukturen aus dem Quellcode werden dagegen direkt im Java-Code wiedergegeben. Triviale Probleme und größere Zusammenhänge sind textuell beschrieben. Die Struktur des Pseudocodes ist stark an Java angelehnt, Sprachelemente für Schleifen und Entscheidungen sowie die Struktur von Funktionsköpfen wurden direkt übernommen. Auch Zuweisungen sind sofort wiederzuerkennen, genauso wie das Prinzip der Bildung von Blöcken mit Hilfe geschweifter Klammern. Tatsächliche Funktionalität ist dagegen in den meisten Fällen im Klartext durch Anführungszeichen eingeschlossen wiedergegeben.

Prinzipiell ist keineswegs jede einzelne Methode und Variable aus jeder einzelnen Klasse im Folgenden dokumentiert. Vielmehr wurde großer Wert darauf gelegt, die Gesamtstruktur des verwendeten Codes erkennbar zu machen, und das Zusammenspiel der einzelnen funktionalen Elemente zu verdeutlichen. Einfache Funktionalitäten im Zusammenhang mit Bibliotheken von SWT, JFace oder der Java-Umgebung – beispielsweise die Funktionsweise der Klasse `ArrayList` oder der Methoden zur Darstellung von Grafikelementen – wurden im Interesse dieser Zielsetzung zumeist nicht näher dokumentiert. Ebenfalls weggelassen wurden Strukturen, die geradlinig implementiert und nicht weiter interessant sind, oder die an anderer Stelle bereits in ähnlicher Form vorgestellt wurden. Auf diese Weise soll unnötiger Ballast vermieden werden, um die Dokumentation des Codes stattdessen so umfassend, verständlich und interessant wie möglich zu gestalten.

7.3 Allgemeine Datenstrukturen

Es existieren in dem hier besprochenen Projekt einige Datenstrukturen, die über den gesamten Quellcode hinweg immer wieder auftauchen. Die wichtigste darunter ist die zur Kapselung des DOM-Baumes verwendete Baumstruktur. Daher sollen an dieser Stelle beispielhaft Aufbau und Funktionsweise dieser Struktur besprochen werden.

Die hier vorgestellte Baumstruktur besteht aus zwei Klassen. Objekte der Klasse `TreeNodeElement` bilden die Knoten des Baumes ab, solche der Klasse `TreeLeafElement` die Blätter, also jene Elemente, die selbst über keine eigenen Unterelemente verfügen. Aus dieser Aufteilung ergibt sich, dass bis auf das Wurzelelement jedem Element im Baum genau ein Element vom Typ `TreeNodeElement` direkt übergeordnet ist. Ihre grundsätzliche Funktionalität erben die beiden genannten Klassen von der Klasse `AbstractTreeElement`, wie dem Klassendiagramm aus Abbildung 7.3 auf der nächsten Seite zu entnehmen ist.

Die Klasse `AbstractTreeElement` soll als erstes näher betrachtet werden. Wie aus der Abbildung zu erkennen ist, erbt die Klasse die Funktionalität der Klasse `DOMNodeData`, deren Objekte jeweils einen Knoten des DOM-Baumes enthalten und die an anderer Stelle näher beschrieben ist. Weiterhin stellt sie die beiden Schnittstellen `IAdaptable` und `IWorkbenchAdapter` zur Ver-

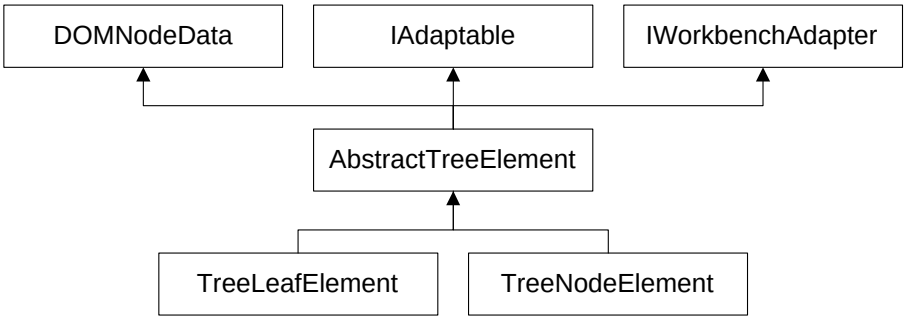

Abb. 7.3: Klassendiagramm: Baumstruktur

fügung. Beide wurden bereits bei der allgemeinen Betrachtung von Eclipse besprochen. Die Implementierung von `IAdaptable` dient hier dazu, auf generische Weise den Zugriff auf das andere implementierte Interface zu ermöglichen. Im Code sieht dies wie folgt aus:

```
public Object getAdapter(Class adapter) {
  if (adapter == IWorkbenchAdapter.class) {
    return this;
  }
  return null;
}
```

Ähnlich arm an überraschenden Offenbarungen gestaltet sich auch der Rest der Implementierung der vorliegenden Baumstruktur. Die Klasse `AbstractTreeElement` enthält private Felder für Namen und Elternelement eines Elements im Baum. Zusätzlich werden zur Verwendung im Editor-Teil der Implementierung aus einer statischen Integer-Variablen eine eindeutige Id-Nummer generiert und gespeichert sowie ein Feld für den internen Namen eines Elements vorgehalten. Zu allen Feldern existieren entsprechende Zugriffsmethoden. Weiterhin sind noch die Methoden `getTreeRoot` – die das Wurzelelement des Baums zurückgibt – und `findElement` beziehungsweise `findTextElement` erwähnenswert. Die beiden letzteren Methoden suchen ab der aktuellen Position des Elements im Baum nach Unterelementen beziehungsweise ASCII-Daten, die aus der ihnen jeweils übergebenen Zeichenkette bestehen. Zur Illustration für diese wie auch die meisten anderen ähnlichen Methoden hier der Code der Methode `findElement`. An diesem Codebeispiel wird zugleich die Verwendung der Methoden aus `DOMNodeData` deutlich.

```
public AbstractTreeElement findElement(String string) {
  try {
    // Ist das aktuelle Element ein normales Element des Baums,
    // und entspricht sein Name der übergebenen Zeichenkette,
    // so ist es das gesuchte Element.
    if (getNode().getNodeType() == Node.ELEMENT_NODE
```

```
      && getNodeName().compareTo(string) == 0) {
      return this;
    }
  } catch (NullPointerException e) {
    // Ignoriere den aufgetretenen Fehler und setze die Suche fort.
  }

  if (this.isTreeNode()) {
    AbstractTreeElement[] children =
      ((TreeNodeElement) this).getChildren();

    // Durchsuche rekursiv die Unterelemente des aktuellen Elements
    // nach dem gesuchten Element.
    for (int i = 0; i < children.length; i++) {
      AbstractTreeElement subElement =
        children[i].findElement(string);
      // Wurde ein passendes Element gefunden,
      // so breche die Suche an dieser Stelle ab.
      if (subElement != null) {
        return subElement;
      }
    }
  }

  // Nichts gefunden.
  return null;
}
```

Neben diesen Methoden enthält die Klasse Implementierungen der bereits bekannten Methoden, die in den beiden Schnittstellen enthalten sind. Um ohne Rückgriff auf die Analyse von Java-Elementen zur Laufzeit ermitteln zu können, ob es sich bei der aktuellen Instanz einer abgeleiteten Klasse um einen Knoten oder einen Endpunkt im Baum handelt, existieren weiterhin die beiden Methoden `isTreeNode` und `isTreeLeaf`, die beide als Rückgabewert „false“ zurückgeben. Um zu signalisieren, dass es sich um ein entsprechendes Element handelt, muss eine abgeleitete Klasse jeweils die passende Methode überschreiben, und dort stattdessen „true“ zurückgeben.

Die Klasse `TreeNodeElement` fügt dieser Funktionalität noch weitere Methoden zur Arbeit mit eventuell vorhandenen Unterelementen hinzu. Hierbei handelt es sich um Methoden zum erstellen, finden und löschen von Unterelementen. Die meisten dieser Methoden beziehen sich auf den JFace-Baum, es existieren aber auch entsprechende – geerbte – Methoden für den DOM-Baum

beziehungsweise zur Bearbeitung beider Baumstrukturen zugleich. Letzteres ist beispielsweise beim Hinzufügen neuer Elemente mittels der folgenden Methode zu sehen:

```
public TreeNodeElement createTreeNode(String string) {
  TreeNodeElement node = createDisplayNode(MessageUtil.getString(string));
  node.setDOMNode(createDOMNode(string));
  return node;
}
```

Einzelne dieser Methoden werden bei der Besprechung des Editors auftauchen. Da ihre interne Funktionalität jedoch nicht besonders interessant ist, und die Funktion zumeist schon anhand des Namens klar wird, soll hier auf eine ausführlichere Dokumentation verzichtet werden.

Auch über die Klasse `TreeLeafElement` ist nichts weiter zu berichten. Sie erweitert die abstrakte Klasse `AbstractTreeElement` um zwei Methoden, wovon die eine – `isTreeLeaf` – schlicht ein „true“ zurückgibt, und die andere, die für das Interface `IWorkbenchAdapter` benötigte Methode `getChildren(Object)`, eine neu angelegte leere Liste von Elementen des Typs `Object`. Dies erklärt sich daraus, dass ein Endelement der Baumstruktur naturgemäß über keinerlei Unterelemente verfügt und somit auch neben den bereits in `AbstractTreeElement` enthaltenen Feldern keine weitere Funktionalität benötigt.

Die hier vorgestellte Datenstruktur ist die umfangreichste und am häufigsten vorkommende Struktur im vorliegenden Projekt. Neben dieser existieren noch eine Reihe weiterer Strukturen, die, zumeist in ähnlicher Form aufgebaut, ebenfalls zur Verwaltung von Daten in Form einer baumartigen oder anderen hierarchischen Struktur gedacht sind. Da sich die meisten enthaltenen Elemente dabei wiederholen, dürfte das Verständnis auch der hier nicht ausführlicher dargelegten Varianten nicht weiter schwer fallen.

7.4 Allgemeine Hilfsklassen

Innerhalb des Gesamtprojektes existieren einige Klassen, deren Funktionalität der allgemeinen Unterstützung der eigentlichen Implementierung dient. Den elementarsten Vertreter dieser Sparte stellt die als Interface definierte Klasse `Constants` dar, deren Inhalt aus einigen global genutzten Konstanten besteht. Dies sind derzeit im einzelnen die Zeichenketten, die zur Initialisierung der JAXP Document-Factory benötigt werden, Ganzzahl-Konstanten für SAP- und AIM-Dokumente sowie Konstanten, die genutzt werden, um anzuzeigen, ob eine Spalte in einer Tabelle optional ist. Die Definition geschieht in üblicher Java-Manier:

```
public static final int SAP = 0; // SAP-Dokumente
public static final int AIM = 1; // AIM-Dokunemte
[...]
```

Da die hier aufgeführten Konstanten in einem Interface definiert sind, ist es möglich, sie in jede beliebige Klasse einzubinden. Würde stattdessen eine Klasse verwendet, wäre dies in Java

nicht möglich, da in dieser Programmiersprache jede Klasse nur jeweils eine andere Klasse als übergeordnete Klasse haben kann. In den hier beschriebenen Klassen sowie in den vorliegenden Klassendiagrammen wird die Einbindung des `Constants`-Interfaces nicht explizit erwähnt.

Die Klasse `StdUtil` fasst allgemein verwendete Methoden zusammen. Dies betrifft derzeit zum einen die folgende einfache Methode, die abprüft, ob ein Index-Wert innerhalb der übergebenen Liste von Objekten liegt:

```
public static final boolean inBounds(Object[] array, int index) {
  if (array == null || array.length <= index || index < 0) {
    return false;
  }
  return true;
}
```

Zum anderen ist die ebenfalls neu implementierte Methode `strArrayCat` enthalten, welche die in einem Feld von String-Feldern gespeicherten Zeichenketten in einem einzigen String-Feld zusammenfasst.

Der einheitlichen Verwaltung von einfachen Listen dienen die in der Untergruppe `listData` zusammengefassten Klassen. Diese beinhalten jeweils die ebenfalls dort definierte Schnittstelle `IListDataRepository`, mit deren Hilfe eine Liste bearbeitet werden kann:

```
public interface IListDataRepository {
  public ArrayList getList();
  public String[] getStringList();
  public void addEntry(String addEntry);
  public void removeEntry(String removeEntry);
}
```

Für die eigentliche Implementierung stehen derzeit zwei Klassen bereit. Dies sind die Klasse `ArrayListManager`, die intern mit einer Liste des Standardtyps `ArrayList` arbeitet sowie die Klasse `FileListManager`, die Daten aus einer Datei bereithält. Im Konstruktor werden dabei jeweils die initialen Daten übergeben, in Form einer Liste beziehungsweise einer einfachen ASCII-Datei mit durch Zeilenumbrüche voneinander getrennten Einträgen. Auf diese Weise ist es einfach möglich, transparent auf Daten zuzugreifen, ohne sich darum kümmern zu müssen, ob und wie diese gespeichert und verwaltet werden.

Eine weitere sehr einfache Hilfsklasse stellt die Klasse `MessageUtil` dar. Sie besteht im Wesentlichen aus einer einzigen statischen Methode, der eine Zeichenkette übergeben werden kann, an Hand derer sie aus einer Datei eine dazu gehörende zweite Zeichenkette ermittelt, die zurückgegeben wird. Wird in der Datei kein Eintrag gefunden, so wird stattdessen die ursprüngliche Zeichenkette zurückgegeben. Der Sinn des ganzen besteht darin, beschreibende Texte in der Oberfläche nicht fest in den Programmcode zu integrieren, sondern an ihrer Stelle die gerade vorgestellte Methode – mit dem bezeichnenden Namen `getString` – mit einem Schlüsselwort

aufzurufen und die eigentliche Beschreibung aus der erwähnten Datei zu entnehmen. So wird eine Lokalisierung[4] des Programms sehr einfach, da keine Änderungen im Programmcode selbst vorgenommen werden müssen, sondern lediglich die Datei mit den Schlüsselwörtern und Beschreibungen anzupassen ist. Die Verwaltung der Datei sowie der darin enthaltenen Einträge wird von der standardmäßig in Java enthaltenen Klasse `RessourceBundle` übernommen. Abschließend hier noch ein kleiner Auszug aus dieser Datei für die englischsprachige Programmversion:

```
newPrimName=(new primitive)
newMsgName=(new message)
none=(none)
MsgStructElementsTable=Structured message element
MsgBasicElementsTable=Basic message element
select_repository_entry=Select repository entry
```

7.5 XML Datenhaltung

Die Verwaltung der XML-Daten geschieht mit Hilfe dreier Klassen. Es sind dies die Klassen

- `DOMManagement`, in der die gesamte zum Lesen und Schreiben der XML-Daten benötigte Funktionalität enthalten ist, und die in der Lage ist, eine mit den Mitteln von Eclipse darstellbare und modifizierbare Baumstruktur aus dem eingelesenen DOM-Baum zu erzeugen;
- `DOMNodeData`, die einen einzelnen Knoten des DOM-Baumes enthält und Funktionen zur Bearbeitung der darin enthaltenen Daten und Unterelemente bereitstellt;
- `ElementNameGenerator`, mit deren Hilfe sich nach bestimmten Regeln der aktuell zu verwendende Name für ein Element des DOM-Baumes zur Laufzeit ermitteln lässt.

Diese decken zusammen alle Operationen ab, die direkt mit den XML-Daten zu tun haben. Sie kapseln die Funktionalität der verwendeten Elemente, namentlich des DOM in Verbindung mit JAXP, was einen möglichen Austausch dieser Elemente oder den Einsatz neuer Versionen einfach macht. Nachfolgend werden Aufbau und Funktion der drei Klassen näher beschrieben. Auf die Darstellung von Klassendiagrammen wird hierbei verzichtet, da jede der drei Klassen unabhängig von irgendwelchen anderen Klassen existiert.

7.5.1 ElementNameGenerator

Die Klasse `ElementNameGenerator` enthält ausschließlich statische Methoden. Die einzige öffentliche Methode darunter ist die Folgende:

```
public static String createElementName(AbstractTreeElement treeElement);
```

[4] Anpassung an eine Sprache.

Diese Methode ermittelt den aktuell gültigen Namen für das übergebene Element. Hierzu wird zuerst das zugehörige XML-Schema-Element auf einen „addName“ oder „replaceName“ Eintrag hin überprüft. Wird ein solcher gefunden, wird mittels der von der Klasse `DOMNodeData` bereitgestellten Methoden der Name des Zielelements zur Namensbildung herangezogen. Wird kein Eintrag des letzteren Typs gefunden, wird zudem der Name des übergebenen Elements ermittelt. Der zurückgegebene String beinhaltet dann entweder den aus dem „replaceName“ Eintrag ermittelten Namen, oder eine Kombination aus dem eigentlichen Namen des Elements und dem Ergebnis eines „addName“ Eintrags, oder nur den Namen des übergebenen Elements. Die Reihenfolge der aufgezählten Optionen spiegelt dabei die Reihenfolge der Prioritäten wieder. Die erste vorhandene und nicht leere der drei genannten Möglichkeiten wird zurückgegeben.

Unterstützt wird die eben beschriebene Methode durch einige nicht öffentliche Methoden. Diese dienen im Einzelnen

- der Ermittlung der „Annotation“ Elemente im Schema-Dokument:

```
private static void getAppInfoAnnotations(
  Node actNode, ArrayList anno);
```

- der Auswahl der gewünschten Art, beispielsweise „addName“:

```
private static ArrayList getAnnotationContents(
  ArrayList appInfoAnnotations,
  String annotationType);
```

- dem Zusammenbau des textuellen Inhalts eines übergebenen Elements:

```
private static String createAnnotationsName(
  ArrayList appInfoNames,
  AbstractTreeElement rootElement);
```

Die drei Methoden nehmen hierzu wie erwähnt die Methoden aus `DOMNodeData` zur Hilfe. Die Vorgehensweise ist im Ablauf sehr geradlinig, so dass sie für eine genauere Analyse an dieser Stelle nicht interessant ist.

Als besonderes Problem bei der Verwendung dieser Klasse hat sich die zeitliche Abfolge der Änderung des Inhalts eines Elements und der Ermittlung seines neuen Namens erwiesen. Es ist zwingend erforderlich, dass die Informationen zu DOM-Knoten und Schema-Element bereits gesetzt sind, bevor mit Hilfe der vorliegenden Klasse ein neuer Name für ein Element ermittelt wird, da die Methoden dieser Klasse auf die besagten Elemente zur Ermittlung des Namens zurückgreifen. Dies führte in der Praxis zu einigen schwer nachvollziehbaren Problemen, die erst nach eingehender Analyse des Programmablaufs ausgeräumt werden konnten.

7.5.2 DOMNodeData

Die Klasse `DOMNodeData` dient der Kapselung einzelner Knoten des DOM-Baumes sowie der Bereitstellung von Methoden zur Vereinfachung des Umgangs mit diesen Knoten. Diese Methoden greifen direkt auf die von den verschiedenen im DOM definierten Elementen zur Verfügung gestellten Funktionen zu. Anschaulich wird dies an einem frei herausgegriffenen Beispiel:

```
public Node createDOMNode(String string) {
  if (node == null || !(node instanceof Element) || string == null) {
    return null;
  }
  Element el = (Element) node;
  Element newEl = el.getOwnerDocument().createElement(string);
  el.appendChild(newEl);
  return newEl;
}
```

Dieser Methode wird der Name eines neu zu erstellenden Knotens („Node“) übergeben. Zuerst wird überprüft, ob der vom enthaltenden Objekt selbst gekapselte Knoten tatsächlich vorhanden ist und ob er vom Typ `Element` ist, also ein normales Element der Baumstruktur dargestellt und Unterelemente haben kann. Ist auch der übergebene Name ein real existierendes Objekt, so wird ein neues Element kreiert. Dies geschieht im DOM, indem das Dokument, das den gesamten Baum enthält, mittels `getOwnerDocument()` ermittelt wird und für dieses Dokument mittels `createElement(String)` ein neues Element erstellt wird. Dieses neu erstellte Element wird dann mittels `appendChild(Node)` als Unterelement des gekapselten Elements in den Baum eingefügt.

Entsprechend wird auch beim Anlegen, Entfernen und Auslesen von Attributen, Text (Zeichenketten, als ASCII-Wert eines Elements) und dem Namen von Elementen vorgegangen. Letzteres stellt dabei einen Sonderfall dar, da ein Name in der vorliegenden Implementierung einerseits nur ausgelesen, nicht aber geändert werden kann, und andererseits für das Auslesen eines Namens parallel zur normalen Funktion (`getNodeName`), die sich auf das gekapselte Element des Baums bezieht, noch eine statische Funktion zur Verfügung steht, die das Element, für das ein Name ermittelt werden soll, als Übergabeparameter enthält. Diese zweite Methode ist nachfolgend als ein weiteres Beispiel dargestellt:

```
public static String getNodeName(Node node) {
  if (node != null) {
    String name = node.getLocalName();
    if (name == null) {
      name = node.getNodeName();
    }
    return name;
```

```
  }
  return null;
}
```

Wie zu erkennen ist, gibt es im DOM zwei Arten von Namen für ein Element. Diese werden beide nacheinander abgefragt, um in jedem Fall einen Namen ermitteln zu können. Dies muss geschehen, da der „LocalName“ nicht zwingend gesetzt sein muss. Bei den Gründen hierfür handelt es sich um Details des DOM, die hier nicht weiter vertieft werden sollen. Es genügt zu wissen, dass in der Praxis zumeist beide Methoden zur Ermittlung des Namens die selbe Zeichenkette zurückgeben, so denn beide Arten des Namens gesetzt sind.

Wie bereits an den beiden Beispielen zu erkennen ist, ist auch die in dieser Klasse enthaltene Funktionalität sehr einfach und baut im Wesentlichen einfach auf der Funktionalität des DOM auf. Hinzu kommen einige Methoden zum setzen und auslesen von Objekt-Variablen, im Wesentlichen des zum enthaltenen Knoten gehörenden XML-Schema-Elements sowie einer Hash-Tabelle, die alle gefundenen XML-Schema-Elemente enthält. Deren Zustandekommen und der dahinterliegende Sinn werden bei der Besprechung der Klasse `DOMManagement` näher betrachtet.

Als einzige etwas komplexere Methode in der Klasse `DOMNodeData` soll noch die Methode `sortChildNodes(String[] orderedList)` Erwähnung finden. Diese sortiert alle direkten Unterelemente des gekapselten Elements der Reihenfolge der als Liste von Zeichenketten übergebenen Namen entsprechend. Dies wird benötigt, um vor dem Schreiben des DOM-Baums ins XML-Format die korrekte Reihenfolge der Elemente zu gewährleisten, da diese zum Beispiel bei komplexen „Sequence“-Elementen von Bedeutung ist. Die Funktion des Sortieralgorithmus[5] sei hier im Pseudocode wiedergegeben:

```
sortChildNodes (Namensliste) {
  // Das erste Kind wird für den späteren Aufruf von insertBefore()
  // als Marker verwendet.
  // Es wird solange in der Liste nach hinten geschoben, bis sein
  // Name in der Namensliste auftaucht, dann wird es an der
  // aktuellen Stelle belassen und das darauffolgende Element
  // übernimmt die Funktion als "Schiebekind".
  schiebeKind = getFirstChild()

  for("alle Elemente der Namensliste,
    beginnend beim ersten Element") {

    aktuellesKind = schiebeKind
    while(aktuellesKind != null) {
```

[5] Dieser Algorithmus wurde von mir selbst entwickelt. Da allerdings im Bereich der Sortieralgorithmen bereits eine unüberschaubare Zahl von Varianten beschrieben ist, scheint es dennoch wahrscheinlich, dass es sich hierbei tatsächlich um die Nachahmung eines bereits bestehenden Algorithmus handelt.

```
      if("Name von aktuellesKind gleich Name des aktuellen Elements
        in der Namensliste") {
        // Ein Element der gerade gesuchten Art wurde gefunden!

        if(aktuellesKind != schiebeKind) {
          "Verschiebe aktuellesKind an die Position direkt vor
            schiebeKind in der Liste der Unterelemente und nehme
            das Element hinter aktuellesKind als neues aktuellesKind"
        }
        else {
          "ersetze schiebeKind durch das darauf folgende Element"
          "war schiebeKind bereits das letzte Element, beende den
            Sortiervorgang, da alle Elemente entsprechend Namensliste
            geordnet sind"
          aktuellesKind = schiebeKind
        }
      }
      else {
        "ersetze aktuellesKind durch darauf folgendes Element"
      }
    }
  }
}
```

Der Algorithmus wurde in dieser Form implementiert, um insbesondere zwei Rahmenbedingungen gerecht zu werden: zum einen ist es möglich, dass ein in der Namensliste enthaltenes Element in der Liste der zu sortierenden Elemente namentlich mehrmals vorkommt. Für diesen Fall soll keine Veränderung in der Reihenfolge der gleichnamigen Elemente stattfinden, was durch den implementierten Algorithmus gewährleistet wird. Zum anderen steht nur eine eng begrenzte Zahl von Methoden zur Manipulation der Unterelemente eine Knotens im DOM zur Verfügung. Dies sind im Einzelnen Möglichkeiten zur Ermittlung des ersten Elements und des jeweils auf ein Element folgenden Elements sowie zum Vertauschen von Elementen und zum Einfügen eines Elements vor einem anderen Element. Der Algorithmus wurde so angelegt, dass er mit den gegebenen Möglichkeiten ohne Umwege auskommt.

7.5.3 DOMManagement

Die Klasse `DOMManagement` erfüllt drei Hauptaufgaben. Dies sind das Lesen und Schreiben von XML-Daten, die Erstellung einer für die Verwendung in Eclipse geeigneten Baumstruktur aus einem DOM-Baum sowie die Auswertung von XML-Schema-Dokumenten und die Bereitstellung der benötigten darin enthaltenen Daten in einfach zu lesender Form. All dies sind lediglich

Hilfsfunktionen. Die eigentliche Intelligenz, die auch den Umgang mit XML-Daten letztendlich festlegt, ist in den Klassen von Editor und Daten-Repository enthalten.

7.5.3.1 Auswertung der XML-Schemata

Die letztgenannte Aufgabe wird von der Methode `getSchemaInfo(Document schemaDoc)` übernommen. Deren Funktion sei hier – etwas vereinfacht – im Pseudocode wiedergegeben:

```
HashMap getSchemaInfo(schemaDokument) {
  "Finde das Unterelement 'schema' in schemaDokument"
  for("Alle Unterelemente mit Namen 'element' im 'schema' Unterelement") {

    // SchemaElement ist eine Datenstruktur, die Informationen über den
    // XML-Schema-Eintrag für ein einzelnes Element des DOM-Baums
    // aufnehmen kann.
    "Erzeuge neues SchemaElement 'element'"

    // Speichere das ganze Element, um notfalls direkten Zugriff
    // durch eine nachgeordnete Anwendung zu ermöglichen.
    element.schemaNode = "aktuelles Unterelement"

    // Speichere Name und Art des aktuellen Elements.
    element.name = "'name' Attribut des aktuellen Unterelements"
    element.type = "'type' Attribut des aktuellen Unterelements"

    for("Alle als mögliche Unterelemente des aktuell beschriebenen
      XML-Elements angegebenen Element-Referenzen") {

      // Speichere die für das Unterelement im XML-Schema verfügbare
      // Information.
      "Erzeuge neue SchemaElementRef 'elementRef'"

      // Name des als Referenz angegebenen Unterelements.
      elementRef.name = "Wert des Attributs 'ref'"

      // Minimale Anzahl an Unterelementen dieser Art in einem XML-Baum.
      elementRef.minOccurs = "Ganzzahl-Wert des Attributs 'minOccurs'"

      // Maximale Anzahl an Unterelementen dieser Art in einem XML-Baum,
      // -1 zur Kennzeichnung einer unbeschränkten maximalen Anzahl.
      elementRef.maxOccurs = "Ganzzahl-Wert des Attrubuts 'maxOccurs',
```

```
        -1 wenn dieses der Zeichenkette 'unboundet' entspricht"

      "Setze Werte für minOccurs und maxOccurs auf 1, wenn keine
        Angaben darüber im Schema enthalten sind"

      "Speichere 'elementRef' in element.children"
    }

    "Speichere 'element' in zurückzugebender Hash-Tabelle, mit dem Namen
      des Elements als Schlüssel"
  }
  "Gebe die Hash-Tabelle zurück, die alle erstellten 'element'-Einträge
    enthält"
}
```

Die von dieser Methode erzeugte und gefüllte Hash-Tabelle sollte idealerweise alle Informationen enthalten, die für die Arbeit mit den XML-Daten benötigt werden. Dies ist in der vorliegenden Implementierung nur teilweise der Fall. So werden Attributwerte nicht in den Datenstrukturen vom Typ „SchemaElement“ gesetzt. Ebenso fehlt die explizite Berücksichtigung der möglichen Strukturierungsformen für Unterelemente eines XML-Schema-Elements[6]. Beides führt dazu, dass es für die gegenwärtige Implementierung des Editors nicht genügt, ausschließlich auf die hier enthaltenen Informationen zurückzugreifen. Stattdessen sind die fehlenden Informationen fest im Programmcode eingebaut. Damit sind die hier besprochene Methode und deren Arbeit mehr ein mögliches Modell für eine generische Gesamtimplementierung, als ein zentraler Bestandteil der aktuellen Gesamtimplementierung.

Der mit dieser Methode verfolgte Ansatz bietet durchaus einige Vorteile. So kann bei einer konsequenten Umsetzung der Idee, alle benötigten Informationen aus dem XML-Schema-Dokument zu extrahieren, der Editor ohne Änderungen in der Software auf Veränderungen in den Datenstrukturen für SAP- und AIM-Elemente eingehen. Weiterhin ist es nicht notwendig, alle ohnehin in den XML-Schemata enthaltenen Informationen manuell in den Quellcode zu integrieren.

Jedoch stieß diese Vorgehensweise in der praktischen Umsetzung schon recht bald an Grenzen. So ist zwar tatsächlich alle benötigte Information in den Schemata enthalten, jedoch ist zu deren Auswertung und Bereitstellung in geeigneter Form ein nicht unerheblicher Aufwand notwendig. Das Hauptproblem, das sich hier stellte, liegt in der Verknüpfung von im Editor dargestellten Daten mit den dahinterstehenden Strukturinformation aus einem XML-Schema. Die Entscheidung darüber, in welcher Form Daten grafisch präsentiert werden sollen, erwies sich als zur Programmlaufzeit äußerst komplex und ohne die Bereitstellung zumindest eines gewissen Maßes an Hilfsinformationen in spezifischer und benutzerfreundlicher Weise überhaupt nicht praktika-

[6] Dies betrifft im Wesentlichen die Reihenfolge der Unterelemente, es sind hierbei die Schlüsselwörter „Sequence“, „Choice“ und „All“ erlaubt. Nähere Informationen dazu sind im Kapitel „Grundlagen“ nachzulesen.

bel. Da somit also ohnehin eine Nachbildung der in den SAP/AIM XML-Schemata enthaltenen Strukturen zur Festlegung von deren grafischer Präsentation notwendig ist, sind auch alle anderen Informationen (mögliche Unterelemente, deren Reihenfolge, Attribute, mögliche Optionalität) derzeit in diesen Strukturen nachgebildet. Dies ist nicht zwingend erforderlich, vereinfacht aber die Gesamtstruktur des Programms und verhindert eine Vermischung verschiedenartiger Informationsquellen. Ohnehin war eine Entscheidung für die vorliegende Vorgehensweise zur Einhaltung des vorgegebenen Zeitrahmens unumgänglich.

Der in diesem Abschnitt vorgestellte Ansatz findet also aus den obengenannten Gründen nur in der Baumstruktur-Darstellung der XML-Daten Verwendung, da hierfür die „generisch“ ermittelten Daten ausreichen. Mehr dazu und zur tatsächlichen Implementierung der hier grob umrissenen Strategien im Abschnitt „Editor“ in diesem Kapitel.

7.5.3.2 Lesen und Schreiben von XML-Daten

Das Lesen von XML-Daten aus Dateien sowie die Generierung des darauf basierenden DOM-Baumes erfordert bei Verwendung von JAXP nur sehr beschränkten Programmieraufwand. Ähnlich trivial gestaltet sich auch das Schreiben des DOM-Baums in eine XML-Datei, wenn auch hierzu von JAXP direkt keine geeignete Funktionalität bereitgestellt wird. Die Generierung der Baumstruktur für die Verwendung durch den Editor geschieht dagegen gänzlich manuell. Die ersten beiden Teilaspekte werden hier nur anhand von Code-Ausschnitten in ihrer prinzipiellen Funktionsweise vorgestellt, wohingegen die vorliegende Lösung für die letztgenannte Aufgabe im Detail beschrieben wird, da sie zugleich die Struktur des im Editor verwendeten Baums verdeutlicht.

Das Erzeugen eines DOM-Baums mit Hilfe von JAXP verläuft in den folgenden Schritten:

```
// Benötigte Konstanten.
static final String JAXP_SCHEMA_LANGUAGE =
  "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
static final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";

// Erzeuge eine "DocumentBuilderFactory", von der später der
// eigentliche "DocumentBuilder" angefordert werden kann.
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

// Verwende Namespaces.
factory.setNamespaceAware(true);

// Verwende DTD/XML-Schema zur Gültigkeitsüberprüfung.
factory.setValidating(true);
```

```
// Versuche, XML-Schemata zu verwenden.
// Da dies in JRE 1.4.1 nicht implementiert ist, wird dieser
// Versuch - zum Zeitpunkt der Erstellung dieses Dokuments -
// normalerweise scheitern.
try {
  factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
} catch (IllegalArgumentException x) {
  // Ignoriere die fehlende Unterstützung.
}

// Erzeuge einen entsprechend der vorgenommenen Angaben konfigurierten
// "DocumentBuilder", der die tatsächliche Arbeit der Erzeugung der
// DOM-Baumstruktur übernimmt.
DocumentBuilder builder = factory.newDocumentBuilder();

// Lege eine Routine zur Behandlung möglicher Fehler
// beim Parsen der XML-Daten fest.
DOMErrorHandler errorHandler = new DOMErrorHandler();
builder.setErrorHandler(errorHandler);

// Parse die angegebene Datei und kreiere daraus einen DOM-Baum.
Document domTree = db.parse(new File("DiesIstEineDateiMitXMLDaten.xml"));
```

In diesem Beispiel werden die Daten aus dem Dokument „DiesIstEineDateiMitXMLDaten.xml„ ausgelesen und daraus ein DOM-Baum erzeugt, dessen Wurzelelement in dem Objekt „domTree“ enthalten ist. Die Verwendung von JAXP ist also tatsächlich sehr unkompliziert und bedarf keiner weiteren Erläuterung. Das Konzept der Verwendung einer „Factory“ allerdings ist interessant. Es bietet den Vorteil, dass es genügt, einmal ein Objekt einer „Factory“-Klasse zu konfigurieren, um in der Folge daraus beliebig viele entsprechend konfigurierte Objekte für die tatsächliche Aufgabe – in diesem Falle also Parser – anfordern zu können. Veranschaulicht ist diese Vorgehensweise beispielsweise bei [21] oder im Internet unter [19].

Noch einfacher gestaltet sich das Zurückschreiben des DOM-Baums in eine XML-Datei. Hierfür wird die im Paket von JAXP enthaltene Klasse `Transformer` verwendet. Diese Klasse dient ganz allgemein der Umformung von XML-Daten. Unter anderem ist es damit auch möglich, XML-Daten aus dem Format eines DOM-Baums ins ASCII-Format umzuwandeln. Dies geschieht wie folgt:

```
// Erzeuge aufgrund des DOM-Baums ein Objekt der Klasse DOMSource,
// das vom Transformer als Grundlage der Umformung verwendet wird.
DOMSource domSource = new DOMSource(domTree);
```

```
// Erzeuge ein Objekt, das zur Aufnahme der umgeformten Daten geeignet ist.
ByteArrayOutputStream stream = new ByteArrayOutputStream();
StreamResult sr = new StreamResult(stream);

try {
  // Fordere ein Objekt vom Typ "TransformerFactory" an.
  TransformerFactory factory = TransformerFactory.newInstance();

  // Fordere von dieser "Factory" gemäß der beim Aufbau des DOM-Baums
  // besprochenen Vorgehensweise ein Objekt der eigentlichen Umformer-
  // Klasse an.
  Transformer trans = factory.newTransformer();

  // Setze einige spezielle Eigenschaften für die gewünschte
  // Transformation.
  trans.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
  trans.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "12");
  trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, dtdPfad);
  trans.setOutputProperty(OutputKeys.INDENT, "yes");

  // Transformiere den in "domSource" enthaltenen Baum in eine
  // ASCII-Zeichenkette, die letztendlich im in "sr" enthaltenen
  // "OutputStream"-Objekt gespeichert wird.
  trans.transform(domSource, sr);
} catch (Exception e) {
  // Fehler, Umwandlung war nicht erfolgreich!
  System.exit(1);
}

// Gebe die soeben ins XML-Format umgewandelten Daten aus.
System.out.println(stream.toString());
```

Ohne auf Details wie die verschiedenen möglichen Parameter oder die der Umformung zugrundeliegende Funktionalität näher einzugehen, wird doch an diesem – leicht abgewandelten – Auszug aus dem Quellcode deutlich, wie das Umwandeln eines DOM-Baums in das XML-Format funktioniert. Die erzeugte Zeichenkette kann dabei direkt in eine Datei geschrieben oder aber auch beliebig weiterverarbeitet werden. Letzteres ist beispielsweise für die XML-Ansicht im Editor wichtig und für die Überprüfung des aktuellen DOM-Baumes auf Gültigkeit. Auch hierzu mehr im Abschnitt über die Funktionsweise des Editors. Über verschiedene Möglichkeiten zur Erzeugung von XML-Code mit Hilfe von Java informiert [23].

7.5.3.3 Bereitstellung der Daten für den Editor

SWT und Eclipse definieren mehrere eigene Formate für die Abbildung von Knoten in einer Baumstruktur. Diese in einer eigenen Implementierung zu verwenden, bietet den Vorteil, dass die bereits vorhandenen Routinen zur Bearbeitung und grafischen Darstellung des Baums genutzt werden können. Daher stellt die hier besprochene Klasse eine Möglichkeit zur Einbettung des DOM-Baums in eine Eclipse-spezifische Baumstruktur bereit. Dies geschieht mit Hilfe der Methode

```
public void getTree(Node rootNode, HashMap schemaInfo,
  TreeNodeElement parent);
```

Diese Methode bildet einen Baum, der als Knoten Objekte des Typs `TreeNodeElement` verwendet und als Endpunkte Objekte des Typs `TreeLeafElement`, beide bereits vorgestellt im Abschnitt über allgemeine Datenstrukturen. Die Vorgehensweise hierbei wird nachfolgend im Pseudocode vorgestellt:

```
getTree(rootNode, schemaInfo, parent) {

  if ("Der aktuellen Knoten 'rootNode' is ein leerer Textknoten.") {
    return;
  }
  "Bilde den zu verwendenden Namen des Knotens 'rootNode' mit Hilfe der
    Methode 'createElementName' aus der Klasse 'ElementNameGenerator'."

  if ("'rootNode' hat keine Unterelemente oder Attribute") {
    "Füge durch Aufruf der Methode 'createDisplayLeaf' im Objekt
      'parent' ein neues Blatt (End-Element) 'neuElement' zur Liste
      der Unterelemente dieses Objekts hinzu. Als Argument wird der
      Methode der soeben generierte Name übergeben."
  }
  else {
    "Füge durch Aufruf der Methode 'createDisplayNode' des Objekts
      'parent' einen neuen Knoten 'neuElement' zur Liste der
      Unterelemente dieses Objekts hinzu. Als Argument wird der Methode
      der soeben generierte Name übergeben."

    for("Alle Attribute des DOM-Elements 'rootNode'") {
      // Rekursiver Aufruf von 'getTree', um die Baumstruktur um
      // Attribute des aktuellen Elements zu erweitern.
      getTree("Aktueller Attribut-Knoten von 'rootNode'",
        schemaInfo, neuElement);
```

```
    }

    for("Alle Unterelemente des DOM-Elements 'rootNode'") {
      // Rekursiver Aufruf von 'getTree', um die Baumstruktur mit dem
      // aktuellen Unterelement fortzusetzen.
      getTree("Aktueller Unterelement-Knoten von 'rootNode'",
        schemaInfo, neuElement);
    }
  }

  // Setze im neuen Eclipse-Konten beziehungsweise Blatt eine Referenz
  // auf den zugrundeliegenden DOM-Knoten.
  "Rufe die Methode 'setDOMNode' des neuen Eclipse-Knotens oder Blatts
    'neuElement' mit dem Argument 'rootNode' auf."

  // Entscheide, ob der aktuelle Knoten in grafischen Bäumen für den
  // Benutzer sichtbar dargestellt werden soll oder nicht.
  "Rufe die Methode 'setDisplayNode' in 'neuElement' auf.
    Wenn das aktuelle Baum-Element kein Element-Knoten oder Text-Knoten
    (Daten) ist, gebe als Argument 'false' an, ansonsten 'true'."
}
```

Zum Verständnis der Funktionsweise muss man wissen, dass Attribute wie auch alle sonstige zur Verfügung gestellte Information im DOM-Baum als eigene Knoten (beziehungsweise Blätter) abgebildet werden. Damit ist die einfache rekursive Vorgehensweise beim traversieren des DOM-Baums klar ersichtlich. Nach Beendigung dieses Vorgangs existiert eine vollständige Abbildung des DOM-Baums im Eclipse-Format. Einzige Ausnahme sind nicht gebrauchte leere Text-(Daten-)Elemente, die ignoriert werden. Ergänzend zur bereits vorhandenen Information wird für jedes Element des neuen Baums festgehalten, ob es für den Benutzer sichtbar sein soll. Ansonsten ist in dieser Methode keine weitere interessante Funktionalität enthalten.

7.5.3.4 Sonstige Methoden

Zusätzlich zu den bereits angesprochenen befinden sich in der Klasse `DOMManagement` noch einige weitere Methoden. Hierbei handelt es sich um Hilfsfunktionen oder Funktionen, welche die Anwendung der Klasse vereinfachen.

Die Methode `public Node getRootNode(Document doc);` sucht, ausgehend vom übergebenen Wurzelelement, nach dem ersten Unterelement im Baum, das tatsächlich ein XML-Element abbildet, und gibt dieses zurück.

Die Methode `public TreeNodeElement getTree(IFile xmlFile);` erstellt zuerst durch Aufruf der Methode `readDOM` ein DOM-Dokument. Mittels Aufruf der normalen `getTree`-Methode

mit dem DOM-Dokument, einem Dummy-Element anstelle der XML-Schema-Information, und einem neu erzeugten Wurzelelement für den Eclipse-Baum als Parameter wird sodann der für die Weiterverarbeitung verwendete Baum erzeugt. Dieser wird als Ergebnis von der Methode zurückgegeben. Damit erzeugt diese Methode direkt eine einsetzbare Baumstruktur aus der angegebenen Datei. Sie demonstriert zugleich die Zusammenhänge zwischen den entsprechenden bereits besprochenen Methoden. Allerdings ist diese Vereinfachung nur dort nutzbar, wo keine Schema-Informationen und keine spezielle Behandlung des Zwischenergebnisses benötigt werden.

Schließlich existieren noch verschiedene Versionen der Methode `getValidInfo`, die jeweils die beim parsen von XML-Daten gefundenen Fehler in Form einer mittels Zeilenumbrüchen formatierten Zeichenkette zurückgeben. Die Variante ohne Aufrufparameter gibt hierbei einfach die beim letzten Parsen gefundenen Fehler zurück. Eine andere Variante, der ein DOM-Dokument Objekt sowie der Pfad zu diesem Objekt übergeben werden, wandelt das übergebene Objekt zuerst unter Verwendung der Methode `writeDOMString` in eine Zeichenkette im XML-Format um, um diese dann mittels `readDOM` wieder als Baum einzulesen und die dabei gefundenen Fehler zurückzugeben. Es existieren geeignetere und spezifischere Methoden, einen DOM-Baum auf Fehler zu überprüfen. Eine solche wurde beispielsweise bei der Entwicklung des ersten Prototypen für dieses Projekt angewandt. Jedoch ist es mit den ohnehin bereits existierenden Methoden zum Lesen und Schreiben und in Anbetracht des gesteckten Zeitrahmens und der damit verbundenen Prioritäten einfacher, in der hier beschriebenen Weise vorzugehen. Die zwangsläufig aus diesem Umweg resultierende reduzierte Performance ist in der Praxis nicht von Belang, wie nachfolgende Tests gezeigt haben.

Bei dem bereits immer wieder erwähnten `ErrorHandler` handelt es sich um eine private Klasse, von der ein Objekt vor dem Parsen von XML-Daten der hierfür eingesetzten Instanz der Klasse `DocumentBuilder` bekannt gemacht wird. An dieses Objekt meldet der Parser dann auftretende Warnungen und Fehler. Er tut dies – je nach Schwere des Problems – durch Aufruf der in einem implementierten Interface definierten Methoden `warning`, `error` und `fatalError`. Diese Methoden haben als Argument eine Ausnahme vom Typ `SAXParseException` als Argument, die die Art des Fehlers sowie die Stelle, an der dieser auftritt, näher beschreibt. Die hier vorliegende Implementierung der privaten Klasse wirft im Falle eines fatalen Fehlers („fatalError“) einen eigenen Fehler, da beim Auftreten von Fehlern dieser Art eine sinnvolle Weiterarbeit des Parsers nicht gewährleistet ist. Fehler der anderen beiden Typen werden lediglich intern in einer Liste von Zeichenketten gespeichert. Diese Liste kann von außen mittels der Methode `public String getLogDataString();` abgerufen werden. Hierbei werden die einzelnen Zeichenketten zu einer einzigen zusammengefügt, deren Elemente durch Zeilenumbruch voneinander getrennt sind.

7.6 Daten-Repository

Das Daten-Repository, bis zu dieser Stelle bereits einige Male erwähnt, dient der Indexierung der im XML-Format vorliegenden Daten der SAP- und AIM-Dokumente. Hierzu wird für jedes

relevante Element in allen beteiligten Dokumenten ein Eintrag in einer internen Datenbank angelegt. Relevante Elemente sind alle die Bestandteile der neuen SAP/AIM-Definition, auf die dem Benutzer an irgendeiner Stelle im Editor die Erstellung eines Verweises möglich sein soll. Im Einzelnen sind dies die Elemente „Primitive“, „PrimStructElem“, „PrimBasicElem“, „Message“, „MsgStructElem“, „MsgBasicElem“, „Constant“ und „Values“. Des weiteren ist mit Hilfe der so erstellten Datenbank auch die Ermittlung und Darstellung von Abhängigkeiten zwischen verschiedenen Elementen und Dokumenten möglich. Diese Funktionalität ist im Repository bereits implementiert, wird aber derzeit im Editor nicht verwendet und harrt noch der Zeit, in der sie einmal Verwendung finden wird.

Ein Datenbankeintrag besteht aus dem Namen eines Elements (Element „Name“ oder „Alias“) , dem Namen des es enthaltenden SAP/AIM-Dokuments (Element „DocName“), dem zum Element gehörigen Kommentar (Element „Comment“) sowie – wo sinnvoll – dem Datentypen des Elements (Element „Type“). Gespeichert wird diese Information in einer internen Struktur, die in serialisierter Form auch in eine Datei geschrieben wird, um die Notwendigkeit eines vollständigen Neuaufbaus bei jedem neuen Laden des Programms zu vermeiden. Hierzu wird die in Java bereits enthaltene Funktionalität zum Serialisieren von Objekten genutzt.

Das hier vorgestellte Daten-Repository funktioniert unabhängig vom eigentlichen Editor. Es greift lediglich auf die bereits vorgestellten Methoden zur Arbeit mit XML-Daten und der zugehörigen Baumstruktur zurück.

Zwischen den Klassen, die das Daten-Repository ausmachen, existieren keinerlei hierarchische Abhängigkeiten, wodurch auch hier wieder auf die Darstellung von Klassendiagrammen verzichtet werden kann. Im Folgenden werden die vier Klassen, die das Repository ausmachen, näher vorgestellt. Die hierbei verwendete Baumstruktur ist dabei, soweit nicht anders erwähnt, immer die für Eclipse erstellte Struktur, die den ursprünglichen DOM-Baum kapselt. Dies gilt auch für alle weiteren Teile dieses Kapitels, soweit nicht ausdrücklich anders erwähnt. Der Einfachheit halber wird diese Eclipse-Baumstruktur hier häufig auch schlicht als „Baum“ bezeichnet. Für die Elemente des Baums wird die Bezeichnung „Knoten“ verwendet. Ebenso werden die Bezeichnungen „Repository“, „Daten-Repository“ und „Datenbank“ gleichberechtigt nebeneinander verwendet, um das hier vorgestellte Gesamtsystem aus gespeicherten Daten und Routinen zu deren Verwaltung zu bezeichnen.

7.6.1 Datenstrukturen

Für die Arbeit mit dem Repository existieren zwei Varianten von Datenstrukturen. In den beiden Vertretern der ersteren Variante, `RepositoryEntry` und `RepositoryDocEntry`, werden die eigentlichen Daten des Repository gespeichert. Die andere Variante, mit den Vertretern `AbstractRepTreeElement`, `RepRootNode`, `RepDocTypeNode`, `RepDocNode`, `RepElTypeNode` und `RepElementNode` dient direkt der Darstellung von Repository-Daten in einer Baumstruktur. Die Reihenfolge in dieser Aufzählung entspricht dabei der Hierarchie der Baumstruktur, wobei die erstgenannte Klasse die grundlegende Funktionalität enthält. Alle anderen zu dieser Struktur

gehörenden Klassen sind von dieser Klasse abgeleitet. Anschaulich wird diese Hierarchie bei der praktischen Anwendung der Baumstruktur, wie in Abbildung 7.4 auf der folgenden Seite wiedergegeben. Der interne Aufbau der Klassen ähnelt stark dem der von `AbstractTreeElement` abgeleiteten Klassen, so dass an dieser Stelle auf eine erneute detaillierte Beschreibung verzichtet werden soll. Der einzige signifikante Unterschied besteht darin, dass hier die Hierarchie zwischen den einzelnen Elementen von vorn herein wie beschrieben festgelegt ist, wohingegen ein Baum aus Knoten der Klasse `TreeNodeElement` beliebig tief verschachtelt sein kann und die Hierarchie der beteiligten Elemente untereinander dabei völlig frei ist.

Der eigentliche Zweck der Strukturen zur Datenspeicherung wird auf einfache Weise erfüllt. Für die Klasse „RepositoryEntry“, die einen einzelnen Eintrag für das Repository enthält, sieht dies so aus:

```
public String name = null;
public int type = UNKNOWN;
public ArrayList references = new ArrayList();
public String comment = null;
public String dataType = null;

boolean defined = false;
RepositoryDocEntry parentDoc = null;

public void addReference(RepositoryEntry reference) {
  references.add(reference);
}
public void removeReference(RepositoryEntry reference) {
  references.remove(reference);
}
public RepositoryEntry[] getReferences() {
  return (RepositoryEntry[]) references.toArray(
    new RepositoryEntry[references.size()]);
}
public void addReferences(RepositoryEntry[] repositoryEntries) {
  for (int i = 0; i < repositoryEntries.length; i++) {
    addReference(repositoryEntries[i]);
  }
}
public RepositoryDocEntry getParentDoc() {
  return parentDoc;
}
```

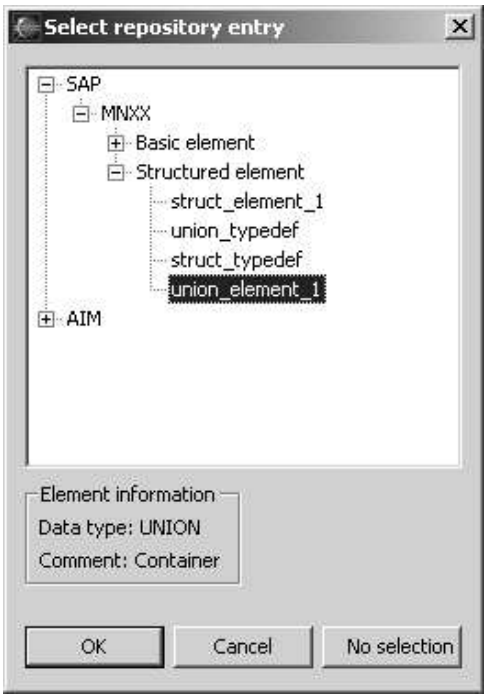

Abb. 7.4: Repository Baumstruktur-Anzeige

Es handelt sich also im Grunde um einen reinen Datenspeicher mit einigen zusätzlichen Funktionen zum einfacheren Zugriff. Die Felder „name“, „comment“ und „dataType“ beinhalten dabei die entsprechenden Werte des abgebildeten Elements. Die „references“-Liste nimmt alle Elemente auf, die das Element referenzieren, also darauf verweisen, und das „type“-Feld sagt aus, um welche Art von Element es sich handelt. Dabei wird zwischen den eingangs erwähnten Elementen unterschieden, denen jeweils eine Integer-Konstante zur Identifikation zugewiesen wird. Diese Konstanten sind bitweise miteinander verknüpfbar. Der „defined“-Wert sagt aus, ob eine Definition für das Element gefunden wurde, oder lediglich Verweise darauf. Und in „parentDoc“ ist ein Verweis auf die Struktur enthalten, die Informationen über das Dokument enthält, in dem der Eintrag definiert ist.

Diese Struktur ist vom Typ `RepositoryDocEntry`. Sie enthält einiges mehr an Funktionalität. Die hier gespeicherten Daten sind:

```
// Name der Datei, in der das hier behandelte Dokument enthalten ist.
public String fileName = null;
// Pfad, an dem die Datei zu finden ist.
public String filePath = null;
// Name des Dokuments, wie er in der XML-Struktur festgelegt ist.
public String docName = null;
// Art des Dokuments, SAP oder MSG.
public int docType = UNKNOWN;
```

```
// Von Eclipse verwalteter Zahlenwert,
// der einmalig ist für jede Version einer Datei.
public long modificationStamp = 0L;
// Zeitpunkt der letzten Änderung.
public long lastModified = 0L;
// Angabe, ob das Dokument tats"achlich existiert, oder nur an anderer
// Stelle auf darin angeblich enthaltene Elemente verwiesen wird.
public boolean defined = true;
// Liste der im Dokument enthaltenen Elemente
private ArrayList repositoryEntryList = new ArrayList();
// Liste aller Dokumente gleichen Namens, einschließlich des
// aktuellen Dokuments selbst.
private ArrayList allFileDocEntries = new ArrayList();
// Liste aller Repository-Einträge aus allen Dokumenten, die den Namen
// des aktuellen Dokuments haben.
private RepositoryEntry[] allFilesRepositoryEntries = null;
// Merker, der besagt, ob die obige Liste verwendbar ist,
// oder neu aufgebaut werden muss.
private boolean allFilesRepConsistent = false;
```

Um diese Daten zu warten und von außen darauf zuzugreifen, existiert eine Reihe von Hilfsfunktionen, deren interne Beschaffenheit zumeist ausgesprochen uninteressant ist. Wo erforderlich, wird jeweils bei der ersten Verwendung einer dieser Funktionen eine kurze Erklärung nachgeliefert.

Auch die Klasse `RepositoryEntry` enthält einige zusätzliche Hilfsfunktionen. Dabei handelt es sich um statische Funktionen zur Ermittlung des Namens eines Elements, des Namens des die Daten zur Definition enthaltenden Unterelements, und des Namens des das Element enthaltenden Abschnitts, jeweils aufgrund einer der bereits erwähnten Integer-Konstanten, welche die verschiedenen Arten von Elementen repräsentieren.

7.6.2 RepositoryDataUtil

Die Klasse `RepositoryDataUtil` enthält ausschließlich statische Hilfsfunktionen. Von außen zugänglich sind hiervon die Methoden `getSectionElements` sowie `getTreeElement`.

Die erstere Methode sucht im übergebenen Baum nach dem „Section“-Element, dessen Name mit übergeben wurde. Aus diesem werden dann alle Unterelemente, die den als drittes Argument übergebenen Namen tragen, mittels der in der Klasse `TreeNodeElement` enthaltenen Methoden ausgelesen und zurückgegeben. Es können also mit Hilfe dieser Methode beispielsweise alle Elemente vom Typ „Primitive“ aus dem Abschnitt „PrimitivesSection“ ausgelesen werden.

Mit der letztgenannten Methode lässt sich der zu einem Repository-Eintrag gehörende Knoten im übergebenen Baum ermitteln. Hierfür werden im Baum der Reihe nach die passenden Elemen-

te für Dokumenttyp („SAP“ oder „MSG“), Abschnitt („Section“-Element, beispielsweise „PrimitivesSection“ oder „PrimStructElementsSection“), Element-Typ („Primitive“, „Constant“,...), und schließlich für den Namen des gesuchten Elements ermittelt. Am Ende der Suche sollte genau ein Element gefunden worden sein, da Elemente gleicher Art *und* gleichen Namens nicht vorgesehen sind. Wird an einem Punkt in der Suche kein passendes Unterelement gefunden, wird das bis dorthin gefundene Element zurückgegeben, ansonsten das letztendlich gefundene Element.

Die einzige benötigte private Methode, `getElementName`, ermittelt den Namen eines Elements. Hierzu wird zuerst der Name des Unterelements bestimmt, in dem der Name des Elements gespeichert ist. Dies wird benötigt, da das entsprechende Unterelement für „Constant“-Elemente „Alias“ heißt, für alle anderen Elemente dagegen „Name“[7]. Diese Aufgabe wird mit Hilfe einer einfachen „switch/case“-Struktur erledigt. Darauf folgend wird aus dem übergebenen Knoten der Wert des ermittelten Elements ausgelesen, der als der Name des übergebenen Elements zurückgegeben wird.

7.6.3 RepositoryManager

Diese Klasse enthält die eigentliche „Intelligenz“ des Daten-Repository. Sie ist nach dem Konzept des „Singleton“ implementiert. Dies bedeutet hier, dass Objekte der Klasse mit Hilfe einer statischen Methode angefordert werden. Intern werden die dabei erzeugten Instanzen in einer Liste verwaltet, und, wo möglich, bereits angelegte Instanz-Objekte als Antwort auf Anforderungen zurückgegeben. Damit kann gewährleistet werden, dass nur jeweils eine Instanz des Repository auf einen bestimmten Datenbestand und damit verbunden auf eine bestimmte Datei mit serialisierten Repository-Einträgen zugreift. Die Trennung zwischen verschiedenen Instanzen wird anhand des zu bearbeitenden Projekts vorgenommen. Da in der gegenwärtigen Implementierung für ein Repository immer alle in jeweils einem speziellen Projekt enthaltenen SAP- und AIM-Dateien berücksichtigt werden, stellt dies eine klare und unmissverständliche Trennungslinie dar.

Die erwähnte statische Methode sieht wie folgt aus:

```
public static RepositoryManager getInstance(IProject repositoryProject) {
  // Bei diesem Projekt handelt es sich um ein Eclipse-Projekt, wird es
  // nicht mitübergeben, so ist eine Zuordnung des Repository nicht möglich.
  if (repositoryProject == null) {
    return null;
  }
  // Da in dieser Version der Methode keine Datei für die gespeicherte
  // Form der Repository-Daten angegeben wird, wird ein Standard-
  // Dateiname angenommen.
  IFile repositoryFile = repositoryProject.getFile("repository.srf");
```

[7] Dies hat „historische“ Gründe – ja, auch ein solch junges Projekt hat bereits eine Vergangenheit aufzuweisen...

```
  // Durchsuche die statische Liste der bereits existierenden
  // Instanzierungen der Klasse nach Varianten, die das selbe Projekt
  // und die selbe Datei verwenden.
  ListIterator li = instances.listIterator();
  while (li.hasNext()) {
    RepositoryManager manager = (RepositoryManager) li.next();
    if (manager.repositoryProject == repositoryProject
      && filesEqual(manager.repositoryFile, repositoryFile)) {

      // Gebe die gefundene, passende Instanz zurück.
      return manager;
    }
  }

  // Erzeuge und initialisiere ein neues Instanz-Objekt der Klassen,
  // da unter den bereits existierenden kein passendes gefunden wurde.
  // Beim Erzeugen wird zugleich eine Liste aller für den Inhalt des
  // Repository herangezogenen Dateien im zu verwendenden Projekt
  // erstellt.
  RepositoryManager manager =
    new RepositoryManager(repositoryProject, repositoryFile);
  initNewInstance(manager);
  return manager;
}
```

Zum Verständnis der Funktionsweise der hier vorgestellten Klasse ist es sinnvoll, dem Ablauf der Ereignisse bei der Initialisierung einer neuen Instanz zu folgen. Der Ausgangspunkt hierfür ist, wie in der obigen Methode zu sehen, die Methode `initNewInstance(RepositoryManager manager);`. Diese veranlasst das übergebene `RepositoryManager`-Objekt zu den folgenden Aktionen:

1. `readRepositoryFile()` – Der Versuch, eine bereits existierende abgespeicherte Datenbank zu verwenden, durch Rückgängigmachen der Serialisierung mit Hilfe von Java-Standard-Klassen[8];

2. `registerRessourceListener()` – Die Installation einer Routine in den zum aktuellen Projekt gehörenden Eclipse-Workspace, die aktiviert wird, wenn sich eine im überwachten Projekt enthaltene Datei verändert[9] , um daraufhin die entsprechenden Einträge im Repository zu aktualisieren;

[8] ObjectInputStream, etc.

[9] Die Registrierung geschieht im Kern mittels der Anweisung `repositoryProject.getWorkspace() .addResourceChangeListener(new IResourceChangeListener() [...]);`

3. `updateRepositoryData()` – Die Aktualisierung beziehungsweise vollständige Neuerstellung der Daten in der Datenbank.

Nach dieser Initialisierung wird das neue Objekt zur statischen Liste der Instanzen hinzugefügt. Die eigentliche Wartung der Daten geschieht zu großen Teilen im Kontext der letztgenannten Aktion, bei der daher die Betrachtung des Geschehens fortgesetzt wird. In der Methode `updateRepositoryData` wird für jede Datei aus der Liste zu berücksichtigender Dateien die Methode `updateData(IFile file);` aufgerufen. Nachdem dies für alle Dateien geschehen ist, werden die in den Dateien gefundenen und zwischenzeitlich in einer Liste zwischengespeicherten Element-Deklarationen mit Hilfe der Methode `updateDeclarations(ArrayListdocDeclarations)` in der Datenbank verarbeitet[10] und dann die Datenbank wieder in die hierfür vorgesehene Datei geschrieben.

Der Weg durch die wesentlichen Funktionalitäten des RepositoryManagers führt uns weiter zur Methode `updateData`, die den Hauptteil der zur Pflege der Datenbank notwendigen Arbeit verrichtet. Deren Funktionsweise ist im Folgenden im Pseudocode wiedergegeben:

```
ArrayList updateData(xmlDocFile) {
  "Wenn die übergebene Datei nicht im Dateisystem vorhanden ist,
    entferne den zugehörigen Eintrag aus der Datenbank."

  RepositoryDocEntry docEntry = "Der in der Datenbank existierende
    Eintrag zur übergebenen Datei 'xmlDocFile'."

  if("Gespeicherte Daten und Datei haben den selben Wert für
    'modificationStamp'") {
    // Datei ist in der Datenbank bereits
    // auf dem neuesten Stand enthalten.
    return null;
  }

  // Version der Datei ist nicht die in der Datenbank enthaltene.
  "Entferne 'docEntry' aus der Datenbank und lege es als neuen
    und leeren Eintrag wieder an."

  // Der Eintrag resultiert direkt aus einer Datei,
  // ist also in jedem Fall tatsächlich definiert.
  docEntry.defined = true;
```

[10] Wird ein Element dabei als bereits in einem Dokument definiert erkannt, wird lediglich in der zum Element gehörigen RepositoryEntry-Struktur die Stelle der Verwendung vermerkt, ansonsten wird ein neues, als nicht definiert (defined = false) gekennzeichnetes Element angelegt.

```
"Setze in 'docEntry' die Variablen 'modificationStamp', 'fileName',
  'filePath' und 'lastModified' auf die für die bearbeitete Datei
  gültigen Werte."

TreeNodeElement docTree = "Erstelle mit Hilfe der Methode 'getTree'
  aus der Klasse 'DOMManagement' aus der aktuellen Datei eine
  Baumstruktur der XML-Daten."

TreeNodeElement treeRoot = "Erstes tatsächliches Datenelement im Baum."
docEntry.docType = "Im in 'treeRoot' enthaltenen Element des Baums zu
  findende Art des Dokuments: 'SAP' oder 'MSG'."
docEntry.docName = "Im Element 'DocInfoSection' zu findender Name des
  Dokuments."

if(docEntry.docType == "SAP-Dokument") {
  "Füge Einträge für die 'Primitive'-Elemente in 'docEntry' hinzu."
  "Füge Einträge für die 'PrimStructElem'-Elemente in 'docEntry' hinzu."
  "Füge Einträge für die 'PrimBasicElem'-Elemente in 'docEntry' hinzu."
}

if(docEntry.docType == "AIM-Dokument") {
  "Füge Einträge für die 'Message'-Elemente in 'docEntry' hinzu."
  "Füge Einträge für die 'MsgStructElem'-Elemente in 'docEntry' hinzu."
  "Füge Einträge für die 'MsgBasicElem'-Elemente in 'docEntry' hinzu."
}

"Füge Einträge für die 'Constant'-Elemente in 'docEntry' hinzu."
"Füge Einträge für die 'Values'-Elemente in 'docEntry' hinzu."

if("Ein Eintrag des in 'docEntry.docName' gespeicherten Namens
  existiert bereits in der Datenbank") {
  "Informiere den Benutzer über die doppelte Vergabe des Namens."
  "Füge den Inhalt von 'docEntry' in geeigneter Weise zum Inhalt
    des bereits existierenden Eintrags hinzu."
}
else {
  "Füge 'docEntry' zur Liste der Dokument-Einträge für diese
    Instanz des RepositoryManager hinzu."
}
```

```
  return "Alle gefundenen Referenzierungen auf Elemente."
}
```

In dieser Weise werden die relevanten Daten verarbeitet, die in einer XML-Datei gefunden werden.

Damit ist zur Bereitstellung der für das Repository benötigten Daten alles wesentliche gesagt. Wie nun aber können die gesammelten Daten tatsächlich verwendet werden? Dazu steht die sehr einfache Methode

```
public RepositoryEntry getRepositoryEntry(String name, String docName);
```

zur Verfügung, die aufgrund eines ihr übergebenen Namens und Dokument-Namens einen dazu passenden Eintrag aus der Datenbank ermittelt und zurückgibt. Hierzu wird zunächst der passende Dokument-Eintrag, ein Objekt der Klasse „RepositoryDocEntry“, in der internen Liste gesucht, und in diesem dann mittels der von dieser Klasse bereitgestellten Methode `findAllFilesRepositoryEntry` das passende Element, das dann zurückgegeben wird.

Damit ist die wesentliche in der Klasse `RepositoryManager` enthaltene Funktionalität erklärt. Es bleiben noch zwei Methoden, `getRepositoryJumpManager` und `getRepositoryTreeManager`, die jeweils zur gerade verwendeten Instanz des Repository passende Objekte der Klassen `RepositoryJumpManager` beziehungsweise `RepositoryTreeManager` zurückgeben. Hierzu mehr in den folgenden beiden Abschnitten.

7.6.4 RepositoryJumpManager

Die Klasse `RepositoryJumpManager` stellt ein Bindeglied zwischen Repository und Editor dar. Eine geeignete Instanz dieser Klasse kann von einem Objekt der Klasse `RepositoryManager` angefordert werden, wie im vorangegangenen Abschnitt dargelegt. Die Klasse stellt eine einzige öffentliche Methode bereit. Diese sorgt dafür, dass das zum ihr als Argument übergebenen Repository-Eintrag gehörende XML-Element in einem bestehenden oder gegebenenfalls neu zu öffnenden Editor[11] angezeigt wird. Im Quellcode sieht dies wie folgt aus:

```
public void jump(RepositoryEntry entry) {
  if (entry != null) {
    // Finde das Dokument, das den übergebenen Eintrag enthält.
    RepositoryDocEntry doc = entry.getParentDoc();
    if (doc != null && doc.filePath != null) {
      // Öffne in dem durch die in 'editorID' enthaltene Zeichenkette
      // festgelegten Editor die XML-Datei, in welcher der übergebene
      // Eintrag enthalten ist.
      IEditorPart editor =
```

[11] Das Wort „Editor“ bezieht sich hier auf einen Editor im Sinne des Eclipse-Konzepts mit in die Plattform eingebetteten Editoren („Editor“) und Ansichten („View“).

```
        openEditor(
          page,
          repository.getFile(doc.filePath),
          editorID);
      if (editor != null) {
        // Teile dem soeben geöffneten Editor mit, welches XML-Element
        // angezeigt und selektiert werden soll.
        setEditorDisplayItem(editor, entry);
      }
    }
  }
}
```

Die aufgerufene Methode `openEditor` besteht dabei im Kern nur aus einem einzigen Statement, das den Aufruf der gleichnamigen Methode in einem Objekt mit dem Interface `org.eclipse.ui.IWorkbenchPage` enthält. Dieses Objekt wird direkt von der Eclipse-Umgebung angefordert. Die zu öffnende Datei und die den nachgefragten Editor identifizierende Zeichenkette werden der Methode als Argumente übergeben. Dabei wird nur dann ein neuer Editor geöffnet, wenn für die entsprechende Datei noch kein Editor geöffnet ist. In jedem Fall wird der betroffene Editor aktiviert und im entsprechenden Fenster in der Eclipse-Plattform im Vordergrund angezeigt.

Die zweite aufgerufene Methode, `setEditorDisplayItem`, fällt ähnlich unkompliziert aus – sie prüft, ob das ihr übergebene Eclipse-„IEditorPart“-Objekt vom Typ „SAPEditor“, also des SAP/AIM-Editors, ist, und ruft, wenn dem so ist, in dem Objekt die Methode `setTreeSelection` auf, mit dem ebenfalls der Methode übergebenen Repository-Eintrag als Argument. Diese Methode sorgt dafür, dass im Editor – und damit in allen Grafikmodulen – das entsprechende Element ausgewählt und angezeigt wird.

7.6.5 RepositoryTreeManager

Auch die Klasse `RepositoryTreeManager` birgt keine weiteren Überraschungen. Zum aktuellen Repository passende Objekte dieser Klasse können ebenfalls durch die Klasse `RepositoryManager` angefordert werden. Sie dienen der Verwaltung eines zur Anzeige bestimmten Baums der Repository-Elemente. Im vom Repository-Manager aufgerufenen Konstruktor der Klasse werden zwei Argumente an neue Objekte übergeben. Hierbei handelt es sich zum einen um eine Liste von `RepositoryDocEntry` Objekten und zum anderen um eine `long`-Ganzzahl, welche die im Baum darzustellenden Typen von XML-Elementen in bitweise verknüpfter Form enthält. Der Konstruktor erzeugt ein Objekt des Typs `RepRootNode`, dem die beiden Parameter als Argumente übergeben werden. Dieses Objekt ist das Wurzelelement des anzuzeigenden Baums. Alle in der Hierarchie vorkommenden Unterelemente, bis hinab zum Element der `RepElementNode`, werden jeweils erst auf Anforderung von dem ihnen in der Hierarchie jeweils übergeordneten Element erzeugt.

Auf das somit in jedem Objekt dieser Klasse von Begin an enthaltene Wurzelelement für den in Eclipse darstellbaren Baum von Repository-Elemente kann mittels der Methode `getRepositoryTree` jederzeit zugegriffen werden. Mittels der Methode `getTreeNode` kann, aufgrund der Namen eines Dokuments und darin enthaltenen Elements oder aufgrund eines `RepositoryEntry` Objekts ein einzelner Element-Knoten aus dem Baum herausgegriffen werden. Dieser kann damit beispielsweise in einem mit Hilfe der Eclipse-Umgebung dargestellten Baum markiert und sichtbar gemacht werden.

7.7 Editor

Der Editor stellt den Kern des hier vorgestellten Projekts dar. Er ermöglicht dem Anwender, die im XML-Format vorliegenden SAP- und AIM-Daten mit Hilfe einer grafischen Oberfläche zu bearbeiten. Der Editor baut dabei auf der Eclipse-Plattform auf und nutzt die von Eclipse bereitgestellte Funktionalität. Des weiteren greift der Editor auf die bereits vorgestellten Programmmodule zurück.

Die Benutzerschnittstelle des Editors besteht, wie bereits bei der Einführung in die Bedienung des Systems ausgeführt, derzeit aus zwei Teilen. Dies ist zum einen eine Einheit, die auf dem „View“-Konzept von Eclipse[12] aufbaut. In diesem View wird die Struktur des gerade aktiven XML-Dokuments in Form eines Baums angezeigt. Damit erfüllt der View die Funktion eines „Outline-View“[13], und tatsächlich baut er auch auf dem bereits in der Eclipse-Plattform integrierten Outline-View auf. Im anderen Teil der Benutzerschnittstelle, der auf dem Eclipse Editor-Konzept aufbaut, kann das selbe XML-Dokument mittels speziell auf die SAP/AIM-Daten zugeschnittener Masken vom Anwender bearbeitet werden. Im Folgenden werden die beiden Teile zumeist einfach als „Editor“ beziehungsweise „View“ bezeichnet.

Editor und View sind eng miteinander verknüpft. Wechselt das aktive Element im Editor, so wird auch in der Baumstruktur des Views das entsprechende neue Element sichtbar gemacht und markiert. Und ändert der Benutzer das im View ausgewählte Element, so wird im Editor das entsprechende Element zur Bearbeitung dargestellt. Im Folgenden wird zunächst die interne Funktionsweise des entwickelten Views erklärt, gefolgt von einer Darstellung der Mechanismen, welche die eben beschriebene enge Verknüpfung von Editor und View bewirken. Ist all dies geklärt, folgt – endlich – eine ausführliche Beschreibung des Editors und seiner Funktion.

7.7.1 Eclipse-View

Der hier entwickelte View, dargestellt in Abbildung 7.5 auf der nächsten Seite, ist im Kern eine Erweiterung des normalen Outline-Views der Eclipse-Plattform. Den zentralen Teil des Views stellt die Klasse `SapeContentOutlinePage` dar, die von der Klasse `ContentOutlinePage` abgeleitet ist.

[12] Näheres hierzu ist bei der Beschreibung der Eclipse-Plattform unter [13] zu finden.

[13] Ein Fenster, in dem die Struktur der im gerade aktiven Editor bearbeiteten Daten angezeigt wird, etwas vereinfacht ausgedrückt.

Im Konstruktor wird dieser abgeleiteten Klasse ein Objekt vom Typ `IAdaptable` übergeben, das in der bereits beschriebenen Weise die für die Baumstruktur benötigten Informationen verfügbar macht.

Die Initialisierung des Views geschieht mittels der Methode `createControl`. Diese in der übergeordneten Klasse bereits implementierte Methode wird in der abgeleiteten Klasse überschrieben. Sie hat den folgenden Aufbau:

```
public void createControl(Composite parent) {
  super.createControl(parent);

  // Erstelle und initialisiere ein Objekt zur Anzeige der Baumstruktur.
  // Als die zu verwendenden Daten beinhaltend wird dabei das
  // im Konstruktor übergebene IAdaptable-Objekt übergeben.
  TreeViewer viewer = getTreeViewer();
  [...]
}
```

`getTreeViewer` ist ebenfalls eine Methode aus der Klasse `ContentOutlinePage`. Sie liefert das in Objekten der Klasse bereits vorhandene Objekt vom Typ `TreeViewer`, das zur Anzeige der gewünschten Daten genutzt werden kann. Da bis auf die tatsächlich im Baum angezeigten Daten keine Änderung in der Funktionalität des Outline-Views gewünscht ist, ist keine weitere Anpassung notwendig.

7.7.2 Interne Kommunikation

In dem soeben vorgestellten View und im eigentlichen Editor soll immer jeweils das gleiche Element angezeigt werden. Um dies zu erreichen, wird ein System benötigt, mit dessen Hilfe Nachrichten zur Synchronisierung ausgetauscht werden können. Zum einen muss es damit möglich sein, die Änderung der aktuellen Selektion mitzuteilen, zum anderen, die Veränderung der dargestellten Daten oder der Struktur des Baumes selbst bekanntzumachen.

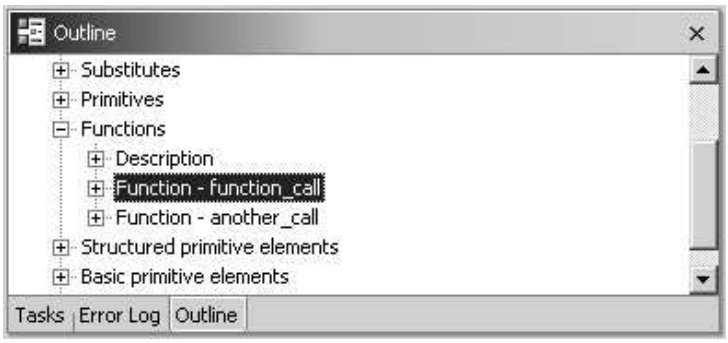

Abb. 7.5: Fenster des Outline-View

Um das Gesamtkonzept offen zu gestalten, und so eine spätere Integration weiterer Views zu erleichtern, wurde entschieden, zu diesem Zwecke sogenannte „Events“ einzusetzen. Dabei handelt es sich um Datenpakete – genauer: Objekte –, die von einer einzelnen Quelle aus an verschiedene Empfänger versandt werden. Diese müssen sich zuvor bei der Quelle in geeigneter Weise registrieren. Da die zugrundeliegende Technik recht interessant ist, wird sie im Folgenden etwas eingehender betrachtet.

Implementiert wurde das hierauf basierende Benachrichtigungskonzept in den drei Klassen `EventClient`, `EventServer` und `EventTransmitter`, wobei die letztgenannte Klasse eine Hilfsklasse für die beiden anderen darstellt und einen Großteil der Funktionalität beinhaltet. Sie implementiert die Interfaces `ISelectionProvider`, `ITreeContentChangeNotifier`, `ISelectionChangedListener` und `ITreeContentChangeListener`, wie aus Abbildung 7.6 zu entnehmen ist. Damit kann diese Klasse Events beider Arten sowohl empfangen als auch versenden. Tatsächlich leitet sie in der Regel die empfangenen Events direkt weiter, wie bereits aus dem Namen zu erschließen ist.

Beschrieben werden soll die Funktion des darauf basierenden Systems am Beispiel der neuen Auswahl eines Elements im Outline-View. Das grundsätzliche Geschehen hierbei sowie die beteiligten Objekte sind in Abbildung 7.7 auf Seite 91 zu erkennen. Die Klasse `EventTransmitter` wird von einer solchen Veränderung – über den Umweg der Klasse `EventClient` – durch Aufruf der folgenden Methode in Kenntnis gesetzt:

```
public void setSelection(SelectionChangedEvent event) {
  if (event != null) {
    this.selection = event.getSelection();
  }
  fireSelectionChanged(event);
}
```

Die erhaltene Nachricht wird also sogleich weitergeleitet. An Hand der hierfür aufgerufenen Methode `fireSelectionChanged` kann zugleich gezeigt werden, wie die Benachrichtigung der

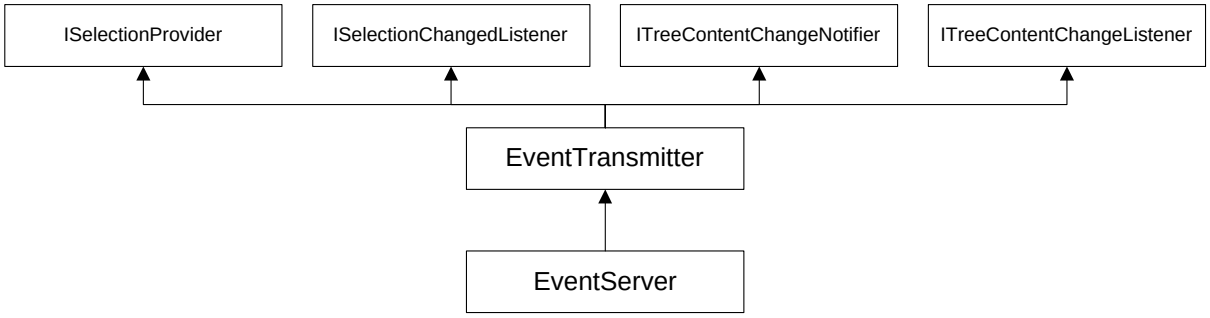

Abb. 7.6: Klassendiagramm: EventServer

registrierten Empfänger geschieht. Es werden dabei alle bei der aktuellen Instanz der Klasse `EventTransmitter` registrierten Empfänger werden von der Veränderung in Kenntnis gesetzt:

```
private void fireSelectionChanged(SelectionChangedEvent event) {
  Object[] listeners = selectionListeners.getListeners();
  for (int i = 0; i < listeners.length; i++) {
    ((ISelectionChangedListener) listeners[i]).selectionChanged(event);
  }
}
```

Das Objekt „selectionListeners“ ist vom Typ `ListenerList`, der speziell zur Aufnahme von Event-Empfängern vorgesehen ist. Als Empfänger registrieren kann man sich durch Aufruf der folgenden Methode:

```
public void addSelectionChangedListener(ISelectionChangedListener arg0) {
  selectionListeners.add(arg0);
}
```

Die Klasse tritt als Empfänger einer Nachricht in Aktion, wenn im Konstruktor ein Objekt vom Typ `EventServer` übergeben wird. Bei diesem erfolgt dann in folgender Weise eine Registrierung:

```
server.addContentChangeListener(this);
server.addSelectionChangedListener(this);
```

Sendet das Server-Objekt dann in der eben dargelegten Weise ein Event, so wird wie gehabt die Methode `selectionChanged` aufgerufen. In der `EventTransmitter`-Klasse bewirkt dieser Aufruf, dass eine gleichnamige Methode mit dem aus dem Event extrahierten Parameter `AbstractTreeElement` aufgerufen wird. Als zweiter Parameter wird ein Wahrheitswert übergeben, der aussagt, ob das empfangene Event ursprünglich von eben dem nun empfangenden `EventTransmitter`-Objekt gesendet wurde. Dies dient der Vorbeugung eines unendlichen Hin- und Hersendens, wenn mehrere entsprechende Objekte miteinander gekoppelt sind. Eine solche Koppelung geschieht tatsächlich, da in beiden anderen Klassen auf die Klasse `EventTransmitter` zurückgegriffen wird. Um in einem realen Objekt in geeigneter Weise auf das Eintreffen eines Events reagieren zu können, muss diese Methode überschrieben werden.

Die Klasse `EventServer` ist ein direkter Abkömmling der Klasse `EventTransmitter`. Die eben vorgestellte `selectionChanged` -Methode sowie die analog dazu existierende `contentChanged`-Methode sind überschrieben und leiten eingehende Events gleich wieder nach außen weiter, so diese von einem anderen als dem Server-Objekt ausgehen. Im Falle der Auswahl sieht dies folgendermaßen aus:

```
protected void selectionChanged(
  SelectionChangedEvent event,
  boolean selfInitiated) {
```

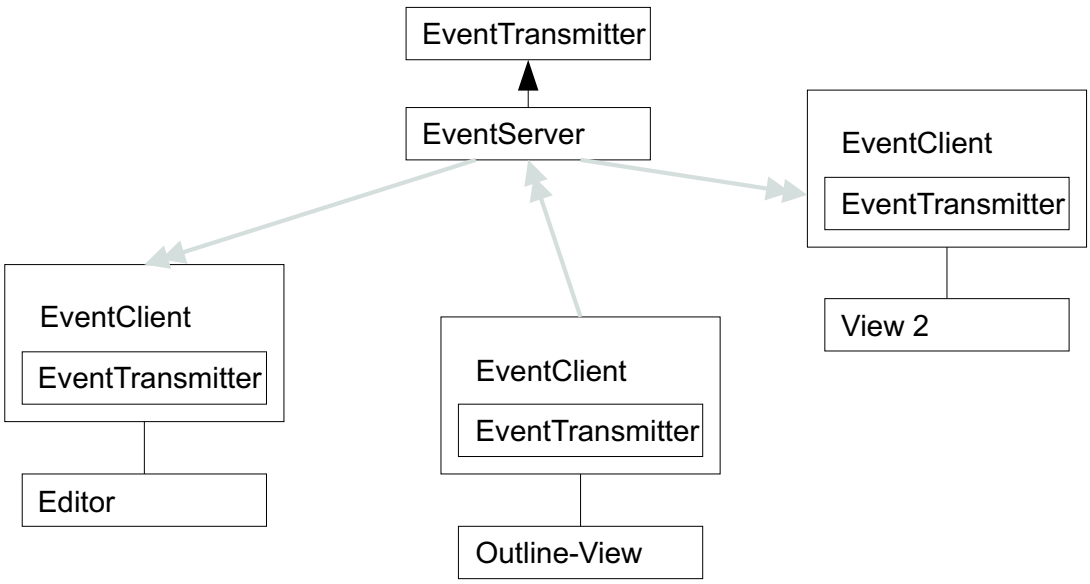

Abb. 7.7: Verarbeitung eines vom Outline-View initiierten Events in der Event-Architektur

```
  if (!selfInitiated) {
    setSelection(event);
  }
}
```

Der `EventServer` sitzt also wie die Spinne im Netz und leitet alle eingehenden Nachrichten einfach an sämtliche Empfänger weiter.

Bei diesen Empfängern handelt es sich um Objekte der Klasse `EventClient`. Diese sollten diejenigen unter den erhaltenen Nachrichten, die sie selbst an den Server gesendet haben, tunlichst ignorieren. Alle anderen Nachrichten sollten dagegen direkt nach außen weitergegeben werden. Um diese Funktionalität zu erreichen, wird auf ein privates Objekt der Klasse `EventTransmitter` zurückgegriffen. Dieses wird im Konstruktor mit dem darin übergebenen Objekt vom Typ `EventServer` initialisiert. Die beiden Methoden zur Benachrichtigung beim Eintreffen eines Events werden überschrieben und so neu implementiert, dass von ihnen jeweils eine gleichnamige, wiederum zu überschreibende Methode aufgerufen wird, falls das Event nicht ursprünglich vom aktuellen Client gesendet wurde. Die `SetSelection`-Methode beziehungsweise ihr Äquivalent für die Benachrichtigung bei Veränderung sind in der Klasse `EventClient` so implementiert, dass sie jeweils die entsprechende Methode im `EventTransmitter`-Objekt aufrufen. Zudem setzen sie noch eine Variable, die, wenn auf „true“ gesetzt, bewirkt, dass empfangene Events immer weitergegeben werden, auch wenn sie vom Objekt selbst gesendet wurden.

Mit den Klassen `EventClient` und `EventServer` ist somit eine umfassende Struktur von Benachrichtigungen möglich. Dazu wird an zentraler Stelle ein Objekt der letzteren Klasse in-

stalliert, das jedem irgendwo verwendeten Objekt der anderen Klasse bei dessen Initialisierung übergeben wird. Diese Objekte können in beliebigen Programmteilen platziert werden, und sind, über das zentrale `EventServer`-Objekt, in der Lage, Events an alle anderen beteiligten Objekte zu versenden. Derzeit wird dieser Mechanismus allerdings nur vom im vorhergehenden Abschnitt besprochenen Outline-View sowie von den Elementen des Editors genutzt. Um allerdings beispielsweise einen weiteren View – zum Beispiel zur Anzeige von Referenzen auf Elemente – zum Projekt hinzuzufügen, der über aktuelle Elemente informiert sein muss, genügt es, in diesem ein Objekt der `EventClient`-Klasse zu halten, um ihn in das Gesamtsystem einzugliedern. So ist die geforderte einfache Erweiterbarkeit um zusätzliche Programmmodule für die Zukunft gewährleistet.

7.7.3 Eclipse-Editor

Der Editor ist das zentrale Element dieses Projekts. Er stellt zugleich die Schnittstelle zum Benutzer und zur zugrundeliegenden Eclipse-Plattform wie auch die Schaltstelle für den Einsatz aller anderen – großenteils bereits vorgestellten – Bestandteile des Projekts dar. Die grafische Oberfläche des Editors ist noch einmal in Abbildung 7.8 auf der folgenden Seite wiedergegeben.

Die Verbindung zur Eclipse-Plattform wird mit Hilfe der im Projekt enthaltenen XML-Datei „plugin.xml“ hergestellt. Der Inhalt dieser Datei für das hier vorgestellte Projekt ist im Folgenden wiedergegeben:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="sape09alpha1"
  name="Sape v0.9alpha1 plug-in"
  version="1.0.0"
  provider-name=""
  class="sape09alpha1.SapePlugin">
  <runtime>
    <library name="sape09alpha1.jar"/>
  </runtime>
  <requires>
    <import plugin="org.eclipse.core.resources"/>
    <import plugin="org.eclipse.ui"/>
  </requires>
  <extension
      point="org.eclipse.ui.editors">
    <editor
      name="Sape v0.9alpha1"
      icon="icons/sample.gif"
      extensions="xml"
```

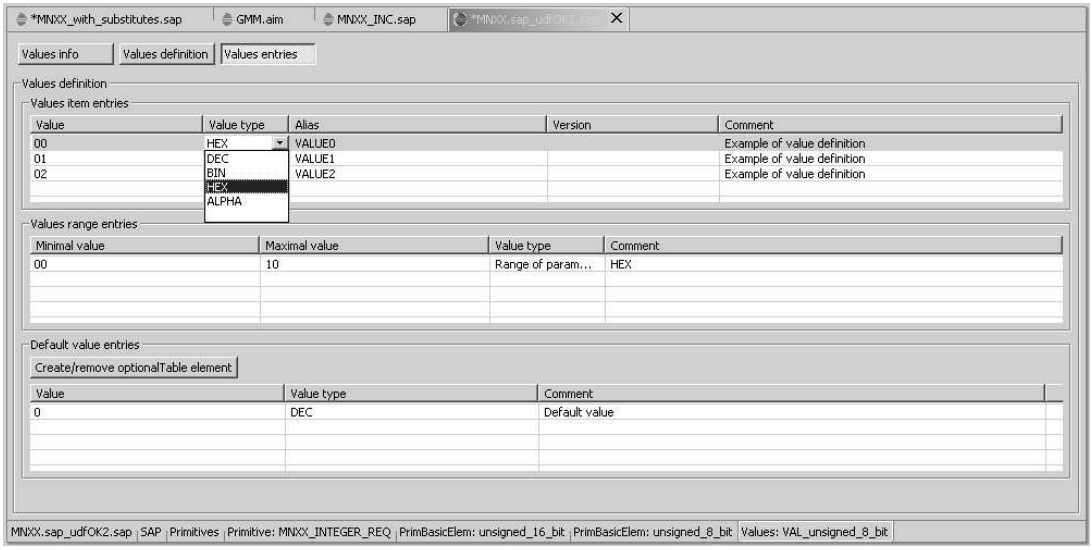

Abb. 7.8: Editor-Fenster

```
      contributorClass="sape09alpha1.editors.SAPEditorContributor"
      class="sape09alpha1.editors.SAPEditor"
      id="sape09alpha1.editors.SAPEditor">
    </editor>
  </extension>
</plugin>
```

Zuerst wird das Plugin mit Namen und einigen weiteren Daten definiert. Dem folgen Abschnitte zum Namen des kompilierten Programmmoduls sowie der benötigten externen Module. Zuletzt wird der implementierte Editor bekannt gemacht. Dies geschieht in Form eines entsprechenden „extension point“. Die tatsächlich von der Plattform zu instanziierende Klasse wird hierin als Wert des Attributs „class“ angegeben. Wird, zum Beispiel nach dem Klick auf eine entsprechende Datei im „Navigator“-View das Öffnen eines neuen Editor-Fensters notwendig, so schaut die Eclipse-Plattform an dieser Stelle nach und erstellt ein neues Objekt der angegebenen Klasse. Für tiefergehende Informationen zu dem Mechanismus der „extensions“ sowie zu Sinn und Funktionsweise der „plugin.xml“-Dateien sei einmal mehr an die in Eclipse bei [13] integrierte Online-Hilfe verwiesen.

Die hier benannte Klasse `SAPEditor` ist von der Klasse `MultiPageXmlEditor` abgeleitet, so dass zunächst deren Funktionalität beschrieben wird. Die Klasse `MultiPageXmlEditor` ist selbst von der in Eclipse enthaltenen Klasse `MultiPageEditorPart` abgeleitet, welche die grundlegende Funktionalität eines Eclipse-Editors bereitstellt. In diesem Falle handelt es sich dabei um einen Editor mit mehreren Seiten, auf denen wiederum andere Editoren oder sonstige Grafikelemente eingebunden werden können. Auf die verschiedenen Seiten kann mittels Karteireitern am unteren

Rand des Editor-Fensters zugegriffen werden, wie bei der Beschreibung der Handhabung des Editors bereits erklärt.

Die Klasse `MultiPageXmlEditor` erstellt zunächst nur eine einzige Seite. Diese enthält einen eingebetteten Editor, der folgendermaßen generiert und in die Seite eingebunden wird:

```
TextEditor editor = new TextEditor();
int index = addPage(editor, getEditorInput());
setPageText(index, editor.getTitle());
```

Dies geschieht in der Methode `createPages`, die von der Eclipse-Plattform aufgerufen wird, wenn der Editor zum ersten mal tatsächlich dargestellt werden soll. Es wird damit schlicht der Inhalt der zu öffnenden Datei in einem neuen Standard-Eclipse-Editor dargestellt. Weiterhin beinhaltet die Klasse hauptsächlich Methoden, die Anforderungen der Eclipse-Umgebung direkt an den eingebetteten Editor weiterleiten, beispielsweise `doSave`, das aufgerufen wird, wenn der Anwender auf den „save“-Button klickt.

Sehr viel komplexer gestaltet sich die abgeleitete Klasse `SAPEditor`, welche die tatsächliche Arbeit verrichtet. Sie kümmert sich um die Darstellung und Verwaltung der Seiten, welche die Eingabemasken für einzelne Elemente enthalten sowie um die Handhabung aller anderen bis hierher beschriebenen Elemente. Hierzu sind zunächst einige Vorarbeiten notwendig.

7.7.3.1 Initialisierungsphase

Im Konstruktor der Klasse werden drei Objekte angelegt. Darunter ein Objekt vom Typ `EventClient`, dessen `selectionChanged`-Methode dergestalt überschrieben ist, dass sie die Methode `showElement` in der Klasse selbst aufruft. Das zweite Objekt vom Typ `DOMManagement` ist für die Verwaltung der XML-Daten zuständig. Das dritte neu angelegte Objekt ist vom Typ `TreeManager`, und enthält Methoden zum Ermitteln des Wurzelelements einer SAP/AIM-Struktur aus einem Baum und dessen Typ (SAP oder AIM) sowie eine Methode `normalize`, die einen Baum von `AbstractTreeElement`-Elementen inhaltlich konsistenter macht, indem beispielsweise leere optionale Elemente daraus entfernt werden.

Nach dem Konstruktor wird die erneut überschriebene Methode `createPages` abgearbeitet. Zuerst wird hierbei die Implementierung der Methode aus der übergeordneten Klasse aufgerufen. Danach wird das darzustellende XML-Dokument mit den Funktionen der `DOMManagement`-Klasse eingelesen und verarbeitet, wie an entsprechender Stelle bereits beschrieben. Auf diese Weise werden eine Baumstruktur erstellt, das betroffene Schema-Dokument ausgewertet sowie Wurzelelement und Dokument-Typ ermittelt. Mit der neu erstellten Baumstruktur und dem spätestens jetzt erstellten `EventServer`-Objekt als Argumente wird daraufhin ein Objekt vom Typ `SapeContentOutlinePage` erzeugt. Dieses Objekt wird auf Anfrage durch das bereits bekannte `IAdaptable`-Interface, das auch von der Klasse `SAPEditor` implementiert wird, an die Eclipse-Umgebung weitergegeben. Auf diesem Wege wird der selbst erstellte Outline-View quasi „von außen“ abgefragt und kommt dann ganz normal zum Einsatz.

Im nächsten Schritt wird die private Methode `setSelection` mit dem soeben ermittelten Wurzelelement des Baums als Argument aufgerufen, die ihrerseits wiederum die gleichnamige Methode im `eventClient`-Objekt aufruft, um die neue Auswahl allen angeschlossenen Einheiten bekannt zu machen. Weiterhin ruft diese Methode die weiter unten genauer beschriebene Methode `showElement` auf, um das gewünschte Element auch im Editor selbst sichtbar zu machen.

Als vorletzter Schritt in der Initialisierung wird eine Instanz der `RepositoryManager`-Klasse angefordert. Dies geschieht bereits hier beim Start des Editors, da der Aufbau der zugehörigen Datenbank einige Zeit in Anspruch nehmen kann, was mitten im Programmablauf störend wirken würde. Zum Abschluss der `createPages`-Methode werden die im geladenen XML-Dokument gefundenen syntaktischen Fehler in einem Popup-Fenster angezeigt, so denn tatsächlich welche gefunden wurden. Damit ist der Editor initialisiert, und die Startseite mit Informationen über mögliche zu bearbeitende Elemente des geladenen SAP- beziehungsweise AIM-Dokuments auf dem Bildschirm dargestellt.

7.7.3.2 Verwaltung von Seiten im Editor

Die innerhalb des Editors geöffneten Seiten werden in einer internen Liste festgehalten. Diese Liste enthält für jede Seite ein Objekt der Klasse `PageData`, das vier Elemente enthält. Dabei handelt es sich um je ein Objekt der Typen

- `Control`, in dem das Wurzelelement der in der Seite verwendeten SWT-Grafikelemente gespeichert wird;
- `AbstractTreeElement`, in dem das Wurzelelement des Teils der XML-Baumstruktur gespeichert ist, dessen Daten auf der Seite dargestellt sind und bearbeitet werden können;
- `IRootPageElement`, das für die tatsächliche Darstellung und Funktion der Seite verantwortlich ist.

Ergänzend ist eine Wahrheitsvariable „needsSave“ enthalten, die angibt, ob in der Seite Veränderungen vorgenommen wurden, die noch nicht gespeichert sind.

Es existieren eine Reihe von Methoden, um auf einzelne `PageData`-Objekte aus der Liste gezielt zuzugreifen. Diese durchsuchen die Liste anhand eines der drei aufgelisteten Elemente und geben das Objekt zurück, welches das entsprechende Element enthält. Zur Arbeit mit den verschiedenen Seiten gibt es noch einige weitere Hilfsfunktionen. Mittels der Methode `createNewPageComposite` wird von der Eclipse-Umgebung ein Wurzelelement für eine neue Seite angefordert, das mittels der Methode `setPageComposite` tatsächlich als neue Seite im Editor etabliert werden kann. Dies kann, bestimmt durch einen entsprechenden Übergabeparameter, entweder in der gegenwärtig aktiven Seite erfolgen, oder in einer neuen Seite. Um ersteres zu erreichen, wird die aktive Seite mittels der Methode `closePage` geschlossen, und das `Control`-Wurzelelement der Seite im Editor neu gesetzt. Da anhand dieser Methode schön die Interaktion zwischen Eclipse und dem Plugin zu sehen ist, sei sie hier im Quellcode wiedergegeben:

```
public void setPageComposite(
  Composite customPage,
  String name,
  boolean inActivePage) {
  int index;

  // Füge das Composite-Element in die aktive Seite ein.
  if (inActivePage) {
    index = getActivePage();
    closePage(getControl(index));
    setControl(index, customPage);
  }
  // Füge das Composite-Element in eine neue Seite ein.
  else {
    index = addPage(customPage);
  }
  // Bestimme den anzuzeigenden Namen der Seite.
  setPageText(
    index,
    name != null ? name : MessageUtil.getString("no_name"));

  // Aktiviere Menüpunkte und Buttons für diese Seite.
  installActions(customPage);
}
```

Die aufgerufenen Methoden `getActivePage`, `setControl`, `addPage` und `setPageText` sind dabei Methoden der übergeordneten Klasse `MultiPageEditorPart`. Die Methode `closePage` sei hier exemplarisch ebenfalls noch einmal im Quellcode aufgeführt:

```
public void closePage(Control pageControl) {
  // Finde die interne Information zur zu schließenden Seite.
  PageData data = getPageData(pageControl);
  if (data == null) {
    // Seite nicht gefunden, weg hier..
    return;
  }
  // Wenn die Seite noch gespeichert werden muss, teile dies der für
  // die Verwaltung der Seite zuständigen Einheit mit.
  if (data.needsSave && data.pageElement != null) {
    data.pageElement.writePageElement();
  }
```

```
  if (data.control != null) {
    // Teile der Eclipse-Umgebung mit, dass die Grafikelemente der
    // Seite nicht mehr gebraucht werden und "entsorgt" werden müssen.
    data.control.dispose();
  }
  // Entferne die Seite aus der internen Liste.
  pageDataList.remove(data);
}
```

Zur Vervollständigung der Werkzeuge zur Arbeit mit den Editor-Seiten existieren noch die Methoden `setActivePage` und `removeActivePage`. Erstere macht das übergebene `Control`-Element zur aktiven und damit im Editor dargestellten Seite, letzteres entfernt die aktive Seite durch Aufruf der Methode `removePage` und macht die davor befindliche Seite zur aktiven Seite. Einzige Ausnahme hierbei ist die erste Seite, welche die textuelle Darstellung der XML-Daten beinhaltet und nie geschlossen wird. Zum Entfernen einer Seite wird die bekannte Methode `closePage` aufgerufen, gefolgt vom Aufruf der Methode `removePage` in der übergeordneten Klasse, die tatsächlich die Seite selbst und deren Karteireiter aus dem Editor entfernt. Das Entfernen der gerade aktiven Seite wird, wie an entsprechender Stelle beschrieben, vom Benutzer gezielt durch entsprechende Aktionen ausgelöst.

Um den Wechsel der aktiven Seite durch den Benutzer verfolgen zu können, ist die Methode `pageChange` der Klasse `MultiPageEditorPart` überschrieben. Wechselt der Anwender auf die erste Seite, wird die Methode `updateEditorInput` aufgerufen, die mittels der entsprechenden Methoden aus der Klasse `DOMManagement` den aktuellen DOM-Baum in die XML-Darstellung umwandelt und das Ergebnis dieser Umwandlung im dort eingebundenen Editor als Text darstellt. Für jede andere Seite wird ebenfalls die Neudarstellung deren Inhalts veranlasst, um Veränderungen der zugrundeliegenden Daten zu berücksichtigen, die möglicherweise seit der letzten Darstellung der Seite stattgefunden haben.

Diese Neudarstellung geschieht für Editor-Seiten, die der Bearbeitung der Daten dienen, durch den Aufruf der Methode `setSelection`. Diese Methode teilt mittels des `EventClient` nach außen die neue Auswahl mit und veranlasst die Darstellung der Seite im Editor durch Aufruf der Methode `showElement`. Je nach Zweck ihres Aufrufs wird der Methode neben dem die darzustellenden Daten beinhaltenden Element der Baumstruktur vom Typ `AbstractTreeElement` mitgeteilt, ob das darzustellende Element in der gerade aktiven oder in einer neuen Seite gezeigt werden soll. Das Geschehen innerhalb der Methode sowie in der nur von dieser Methode aufgerufenen Hilfs-Methode `openPage` ist im Folgenden im Pseudocode veranschaulicht:

```
showElement(
  AbstractTreeElement darzustellendesElement,
  boolean inAktiverSeiteDarstellen) {

  if("mittels 'getPageData(treeElement)' bereits im Editor vorhandene
```

```
    Seite gefunden") {

    "Mache die gefundene Seite zur aktiven Seite."
    inAktuellerSeiteDarstellen = true;
  }

  neueSeite = createNewPageComposite();

  zuVerwendendeKlasse = findCustomElementClass(
    darzustellendesElement.getNodeName());
  if("geeignete Klasse gefunden") {
    seitenverwaltung = zuVerwendendeKlasse.newInstance();
  }
  else {
    // Zeige das Element mit Hilfe der Standard-Anzeige-Klasse an.
    seitenverwaltung = "neues Objekt der Klasse 'MiniTree'."
  }
  "Veranlasse das neue Objekt 'seitenverwaltung' zur Erzeugung der
    zum aktuellen Knoten des Baums passenden Editor-Seite."

  // Die erste Seite muß unangetastet bleiben.
  if("Aktive Seite ist die erste Seite (die den XML-Text enthält)") {
    inAktuellerSeiteDarstellen = false;
  }

  "Rufe 'setPageComposite' auf, so dass 'neueSeite' in der aktuell
    dargestellten Seite angezeigt wird, wenn 'inAktuellerSeiteDarstellen'
    den Wert 'true' enthält, ansonsten so, dass die Seite als eine
    komplett neu erzeugte Seite im Editor dargestellt wird."

  "Mache die aktuelle Seite zur aktiven Seite,
    wenn sie dies nicht bereits ist."

  "Füge die neue Seite zur Liste mit den Daten der im Editor
    dargestellten Seiten hinzu."
}
```

Das einzige bisher unbekannte Element ist hierbei die Art und Weise, wie die für die Handhabung und Darstellung zuständige Klasse ermittelt und verwendet wird. Um herauszufinden, welche Klasse zu verwenden ist, wird der Name des Baum-Knotens herangezogen. Auf Grund dieses

Namens wird in der Methode `findCustomElementClass` nach einer geeigneten Klasse gesucht, von der sodann ein Objekt zur Verwaltung der aktuellen Seite instanziiert wird. Die genannte Methode geht hierfür eine interne Liste aller bekannten entsprechenden Klassen durch und fragt jede dieser Klasse durch Aufruf einer speziellen Methode nach dem Namen des Knotens, dessen Inhalt durch sie bearbeitet werden kann. Hat diese Suche keinen Erfolg, so wird zur Anzeige ein neues Objekt der Klasse `MiniTree` verwendet, das standardmäßig die direkten Unterelemente des Knotens, für den es zuständig ist, in Baumform darstellt.

Damit ist die ungefähre Funktionsweise des Editors und somit der in der Klasse `SAPEditor` implementierten Methoden geklärt. Somit wird es nun Zeit, auch Aufbau und Funktion einer einzelnen Seite des Editors näher zu betrachten. Damit die Objekte, welche die einzelnen Seiten verwalten, ordnungsgemäß funktionieren können, haben sie Zugriff auf das Interface `IPageControl`, welches von der Klasse `SAPEditor` implementiert wird. Diese Schnittstelle beinhaltet unter anderem Methoden zum Zugriff auf geeignete Objekte des `RepositoryManager` und des `FileListManager`, auf die Eclipse-Elemente, in die der Editor eingebettet ist und auf die aktuellen Schema-Daten. Weiterhin kann durch diese Schnittstelle dem Editor mitgeteilt werden, wenn sich ein Element des Baums oder der Baum selbst verändert hat, die aktuelle Auswahl sich geändert hat, oder ein Neuladen der zugehörigen Seite erforderlich ist. Die entsprechenden Methoden in der Klasse `SAPEditor` reagieren hierauf jeweils in geeigneter Weise unter Verwendung der vorgestellten Methoden. Die Interna der Editor-Seiten werden im Folgenden dargestellt.

7.7.3.3 Funktionsweise von Editor-Seiten

Jede der Klassen, welche die Seiten des Editors ausmachen, kann die ihr zugedachte Aufgabe – dem Benutzer die Bearbeitung der Daten im übergebenen Teil des Baums zu ermöglichen – im Prinzip in beliebiger Weise erfüllen. Die einzige für solche Klassen bestehende Vorgabe ist, dass sie das Interface `IRootPageElement` oder eine der in Abbildung 7.9 dargestellten abgeleiteten Varianten implementieren müssen.

In diesen Interfaces sind einige Methodendeklarationen enthalten, die es der kontrollierenden Instanz der Klasse `SAPEditor` ermöglichen, die Erstellung des Seiteninhalts für einen bestimmten

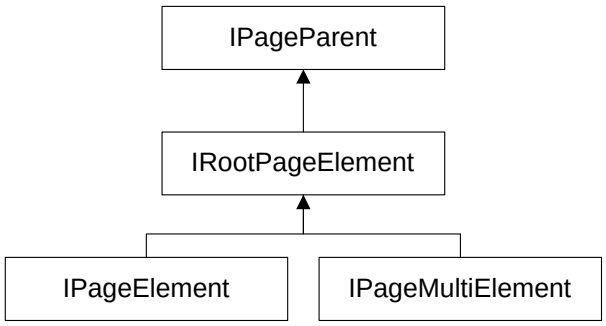

Abb. 7.9: Klassendiagramm: PageElement Interfaces

Knoten im Baum oder – im Falle der abgeleiteten Version `IPageMultiElement` – für eine Reihe von Knoten zu veranlassen (`createPageElement`), oder die in der Seite enthaltenen Daten zurück in die Baumstruktur zu schreiben `writePageElement`. Außerdem ist die bereits angesprochene Methode enthalten, die den Namen des von der Klasse abgebildeten Baumknotens zurückgibt sowie eine Methode, die zu einem übergebenen zur Klasse passenden Element der Baumstruktur den darzustellenden Namen ermittelt und zurückgibt. Neben den beiden bereits angesprochenen Interfaces existiert noch die ebenfalls von `IPageRootElement` abgeleitete Version `IPageElement`, die diesem eine weitere Methode zum Erstellen von Seiten hinzufügt, die sich von der in ersterem Interface enthaltenen dadurch unterscheidet, dass sie geeignete Parameter enthält, um als Unterelement einer Seite genutzt zu werden.

Da die Schnittstelle zum Benutzer einheitlich aussehen und zu bedienen sein soll, ist es zweckmäßig, nicht jede Klasse, die ein Element der XML-Struktur darstellen soll, für sich allein zu entwickeln, sondern möglichst weitgehende allgemeine Strukturen für alle implementierten Klassen zu verwenden. Dies geschieht in der gegenwärtigen Implementierung in mehreren Stufen.

Zunächst sind in drei verschiedenen Klassen die grundlegenden Funktionalitäten der verschiedenen benötigten Arten von Elementen untergebracht. Die einfachste dieser Klassen, `BasicPageElement`, bewirkt lediglich einige grundsätzliche Vereinfachungen für die Implementierung der im Interface `IPageElement` und dessen übergeordneten Interfaces definierten Methoden. Auf dieser Klasse aufbauende Klassen können einzelne normale Elemente auf einer Editor-Seite darstellen. Werden mehr Bildschirmseiten für die zur Darstellung eines Elementes benötigten Grafikelemente gebraucht, so kann mittels Buttons am oberen Fensterrand zwischen mehreren untergeordneten Seiten umgeschaltet werden. Diese Buttons und die dahinterliegende Funktionalität sind in der Klasse `BasicMultiPageElement` unter Rückgriff auf die Methoden des SWT implementiert. Die Klasse erbt die Funktionalität der Klasse `BasicPageElement`. Hinzu kommen noch einige durch ableitende Klassen zu implementierende abstrakte Methoden, die Informationen darüber zurückgeben, wie die verschiedenen untergeordneten Seiten heißen sollen – und damit zugleich, wie viele davon erstellt werden sollen – und welche Seite vorzugsweise anzuzeigen ist. Die dritte Basisklasse in diesem Bereich, `BasicPageMultiElement`, dient der gemeinsamen Darstellung mehrerer gleichartiger Knoten eines Baums auf einer Seite. Hierzu erbt sie die Funktionalität der Klasse `BasicPageElement` und implementiert zusätzlich das Interface `IPageMultiElement`, für dessen Methoden wiederum jeweils vereinfachte Versionen als abstrakte Methoden weitergegeben werden. Die genauen Zusammenhänge zwischen den hier vorgestellten Klassen und Interfaces sind Abbildung 7.10 auf der folgenden Seite zu entnehmen.

Bevor auf die Implementierung der einzelnen Klassen näher eingegangen wird, sei noch angemerkt, dass es problemlos möglich ist, Objekte von verschiedenen dieser Klassen auch ineinander zu schachteln. Hierzu muss ein Objekt, das andere Objekte enthält, lediglich Aufrufe der im implementierten Interface `IRootPageElement` – oder der entsprechenden abgeleiteten Schnittstelle – enthaltenen Methoden an die eingebetteten Objekte in geeigneter Form weiterleiten. So ist sichergestellt, dass beispielsweise eine Aufforderung zum Speichern eines Elements an alle

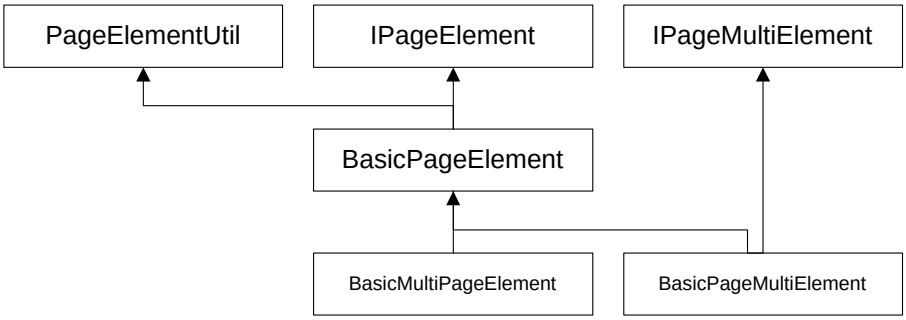

Abb. 7.10: Klassendiagramm: BasicPageElement

in dieser Form enthaltenen Unterelemente weitergegeben wird. Die grafischen Elemente eines Unterelements werden in der Regel einfach in einem diesem zugeordneten Bereich innerhalb des Haupt-Elements platziert. Das Unterelement bemerkt dabei nichts davon, dass es nicht direkt mit dem `SAPEditor` beziehungsweise dessen Interface `IPageControl` zu tun hat. Die Handhabung der hieraus resultierenden Unterschiede im Zugriff auf die Methoden dieser Schnittstelle übernehmen die drei vorgestellten Basisklassen, die von `IRootPageElement` abgeleiteten Interfaces sind zur Handhabung von untergeordneten Elementen bereits ausgelegt. Dies geschieht in der Praxis, indem sich ein Unterelement so lange von Element zu Element durch die Hierarchie aufwärts hangelt, bis es zum alle anderen Bestandteile enthaltenden Objekt kommt, das dann wiederum eine Referenz auf das Interface-Objekt enthält. Der Vorteil dieses geschachtelten Konzepts ist, dass die Abbildung des XML-Baums in den Schablonen des Editors sehr modular gestaltet werden kann, so dass umfangreichere Elemente – beispielsweise zur Darstellung einer Primitive – aus vielen Teilbestandteilen zusammengesetzt werden können, die jeweils in unterschiedlichen Strukturelementen verwendet werden können.

Auf eines der drei vorgestellten grundsätzlichen Elemente greifen alle tatsächlichen Implementierungen zurück. Direkt von den Klassen `BasicPageElement` und `BasicMultiPageElement` abgeleitet sind hauptsächlich jene Elemente, die keine eigenen Text- oder Tabellenelemente definieren, sondern lediglich andere Unterelemente enthalten. Dies trifft auf alle „Section“-Elemente zu, und auch auf die meisten der in der XML-Struktur eine Hierarchiestufe darunter liegenden Elemente, beispielsweise die Elemente `Primitive`, `Message`, `PrimStructElem`, `PrimBasicElem`, um nur eine kleine Auswahl zu nennen. Wie dies in der Praxis aussieht, soll am Beispiel des `Primitive`-Elements erläutert werden.

In der Klasse existiert ein Objekt der Klasse `ChildPagesHandler`, welches die Verwaltung von untergeordneten Seiten übernimmt. Die in den Interfaces definierte `createPageElement`-Methode wird in der Klasse `BasicMultiPageElement`, von der die hier vorgestellte Klasse `Primitive` abgeleitet ist, in eine abstrakte Methode umgeformt, die – etwas vereinfacht und zusammengefasst im Pseudocode dargestellt – wie folgt implementiert ist:

```
protected Composite createPageElementImpl(
```

```
  TreeNodeElement wurzelKnoten,
  Composite enthaltendesGrafikelement,
  int seitenNr) {

  primName = "Ermittle den Namen der Primitive aus den Daten im Baum."

  // Stelle die angeforderte Seite dar. 'BasicMultiPageElement' sorgt
  // auf diese Weise für die Darstellung aller Seiten dieses Elements
  // in geeigneter Form.
  switch (pageNr) {
    case 0 :
    "Erstelle ein Rahmenelement als Unterelement von
      'enthaltendesGrafikelement'."

    // Teile dem in der Klasse vorhandenen Objekt der Klasse
    // 'ChildPagesHandler' eine nach der anderen die zu verwaltenden
    // Unterelement-Seiten mit.
    "Erstelle eine untergeordnete Seite für das 'Description'-
      Unterelement im Baum und stelle diese im neuen Rahmenelement dar."
    "Erstelle eine untergeordnete Seite für die vorhandenen 'PrimUsage'-
      Unterelemente im Baum und stelle diese im neuen Rahmenelement dar."
    "Erstelle eine untergeordnete Seite für die vorhandenen 'History'-
      Unterelemente im Baum und stelle diese im neuen Rahmenelement dar."
    break;

    case 1 :
    "Erstelle ein Rahmenelement als Unterelement von
      'enthaltendesGrafikelement'."
    "Erstelle eine untergeordnete Seite für das 'PrimDef'-Unterelement
      im Baum und stelle diese im neuen Rahmenelement dar."
    "Erstelle eine untergeordnete Seite für die vorhandenen 'PrimItem'-
      Unterelemente im Baum und stelle diese im neuen Rahmenelement dar."
    break;
  }
}
```

Im `ChildPagesHandler`-Objekt wird jede angeforderte Seite gespeichert, und es wird die `createPageElement`-Methode der Seite aufgerufen, um diese tatsächlich aufzubauen. Bekommt die übergeordnete Seite die Anweisung zur Speicherung ihres Inhalts, so genügt der Aufruf der Methode `writePageElements` im `ChildPagesHandler`-Objekt, um dieses zur Speicherung aller Unterele-

mente zu veranlassen. Dies geschieht, in dem in allen in der internen Liste von Seiten enthaltenen Objekten die Methode `writePageElement` aufgerufen wird. Mit diesem Konzept ist es problemlos möglich, beliebig tiefe Verschachtelungen von Elementen zu erzeugen.

Wie jedoch sind nun die Elemente implementiert, die tatsächlich Daten enthalten? Um für die Benutzerschnittstelle ein übersichtliches Gesamtkonzept zu gewährleisten, werden lediglich zwei grundlegende Grafikelemente verwendet: Textfelder und Tabellen. Textfelder sind sehr einfach aufgebaut, wogegen die Verwaltung von Tabellen einen der komplexeren Bereiche des Projekts darstellt. Aufgrund der beschriebenen Möglichkeiten der Verschachtelung genügt es, wenn jede Klasse ein einzelnes Grafikelement enthält. Daher wurden zur einheitlichen Implementierung einige abstrakte Klassen für die beiden Typen von Grafikelementen zusammengestellt, die ein Grundgerüst bieten und von denen alle Klassen, die Daten aus dem Baum enthalten, abgeleitet sind.

Die wenigen Klassen, die auf Textfeldern basieren – beispielsweise die Klasse `Description` –, sind von der Klasse `AbstractTextField` abgeleitet. Objekte dieser Klasse erstellen in der Methode `createPageElementImpl` ein Textfeld, das den Rückgabewert der Methode `createTextString` zum Inhalt hat. Standardmäßig liest diese einfach die im übergebenen Element der Baumstruktur enthaltenen Daten aus. Um andere Daten als darzustellenden Text zu verwenden, oder die Daten vor der Darstellung zu bearbeiten, muss eine abgeleitete Klasse diese Methode überschreiben und in gewünschter Form selbst implementieren. Dasselbe gilt analog für die Methode `writeTextString`, mittels derer beim Speichern des Seiteninhalts der übergebene Text aus dem Datenelement zurück in den Baum geschrieben wird. Damit ist zur Implementierung von Textfeldern im Editor bereits alles Wesentliche gesagt. Die Bearbeitung des Textes selbst wird zur Gänze vom verwendeten SWT `Text`-Grafikelement verwaltet.

Prinzipiell funktioniert die Implementierung von Tabellen genauso – nur ist die Verwaltung hier ungleich komplexer, da innerhalb von Tabellen eine Vielzahl von Operationen möglich ist. Um den Aufbau der entsprechenden Klassen dennoch möglichst einfach zu halten, wurde die notwendige Funktionalität auf mehrere Stufen verteilt. Für die Verwaltung der Tabellen selbst ist die Klasse `AbstractTableViewer` zuständig, die von der Klasse `BasicPageMultiElement` abgeleitet ist. Um den Klassen, die eine solche Tabelle implementieren, die administrative Arbeit abzunehmen, existiert eine weitere von `AbstractTableViewer` abgeleitete Klasse mit dem Namen `AbstractAutoTableViewer`. Für Tabellen, die nicht direkt auf dem übergebenen Element der Baumstruktur aufbauen, sondern nur auf einem Teil von dessen Unterelementen gibt es eine wiederum davon abgeleitete Variante des Namens `AbstractOverviewAutoTableViewer`. Die gegenseitigen Abhängigkeiten dieser drei Klassen sind noch einmal in Abbildung 7.11 auf der nächsten Seite dargestellt, ihre jeweilige Funktion wird nun etwas näher betrachtet.

Die Klasse `AbstractTableViewer` beinhaltet als zentrales Element eine mittels der bereits bekannten Klasse `TableViewer` implementierte Tabelle. Die meisten Methoden der Klasse beziehen sich somit auf die Erstellung und Wartung dieser Tabelle sowie auf die Handhabung von Sonderfällen, wie beispielsweise optionalen Tabellen. Dies bringt viel Programmcode mit sich und

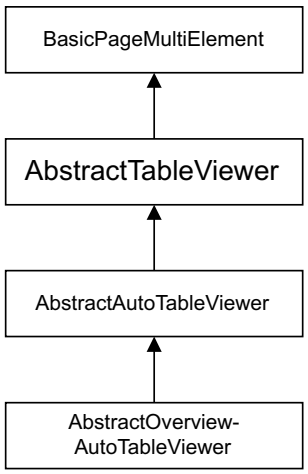

Abb. 7.11: Klassendiagramm: AbstractTableViewer

bedeutet eine Menge Arbeit mit SWT und JFace. Viel Intelligenz oder ausgefeilte Algorithmen abseits der ausgetretenen Pfade der GUI-Programmierung sind dagegen nicht enthalten. Daher soll an dieser Stelle eine Auflistung möglicher Aktionen des Anwenders und der dadurch in Gang gesetzten Mechanismen genug der Beschreibung sein. Wie bereits im entsprechenden Kapitel beschrieben, ist es dem Anwender möglich:

- ein Element der Tabelle zu bearbeiten, was je nach Art des Tabellenelements von verschiedenen Klassen ermöglicht wird, die von der in JFace enthaltenen Klasse `CellEditor` abgeleitet sind und entsprechend die Bearbeitung von Text, die Auswahl eines Elements aus einer Liste, oder die Auswahl eines Elements aus dem Repository ermöglichen;

- Elemente zur Tabelle hinzuzufügen oder daraus zu entfernen, was durch Hinzufügen beziehungsweise Entfernen von entsprechenden Knoten in der Baumstruktur und Neuaufbau des betroffenen Teils der Tabelle erledigt wird;

- durch Doppelklick auf eine Zeile der Tabelle gegebenenfalls zu dem Element zu springen, auf welches das in dieser Zeile dargestellte Element verweist, was eine Instanz der Klasse `RepositoryJumpManager` übernimmt.

Die Klasse `AbstractAutoTableViewer` nimmt, wie bereits angesprochen, den von ihr abgeleiteten Klassen die zur Verwaltung und Synchronisation mit dem die XML-Daten beinhaltenden Baum notwendige Arbeit ab. Die Klasse `AbstractTableViewer` leitet die hierzu notwendige Information lediglich für jede einzelne Tabellenzeile weiter. Dies geschieht für den Aufbau der Tabelle mittels der Methode `createTableEntry`. Die Erstellung eines Eintrags durch diese Methode und die von ihr aufgerufenen Hilfsfunktionen geschieht – wiederum vereinfacht und im Pseudocode dargestellt sowie unter Zuhilfenahme der gleich danach vorgestellten Klasse `TreeStructManager` – wie folgt:

```
protected void createTableEntry(
  AbstractTreeElement wurzelKnoten,
  ParentTableElement wurzelTabellenElement) {

  // Teile dem Objekt der für die Verwaltung der Baumstruktur
  // zuständigen Klasse den zu verwendenden Knoten mit.
  treeStructManager.setRootElement(wurzelKnoten);
  values = "Die in der Baumstruktur unterhalb des 'wurzelKnoten'-Elements
    beheimateten Daten, angefordert durch das 'treeStructManager'-Objekt."

  tabellenZeile = "Neues Tabellenelement mit sovielen Spalten,
    wie für die aktuelle Tabelle benötigt werden."
  for ("Alle Spalten der Tabelle") {
    switch ("Art der Spalte") {
      case "String" :
        "Erzeuge neues 'StringTableElement' mit dem Wert des
          zugehörigen Baum-Elements als Inhalt."
        break;

      case "Auswahlliste" :
        "Erzeuge neues 'ComboBoxTableElement' mit dem Wert des
          zugehörigen Baum-Elements als Inhalt und einem Objekt vom Typ
          'ComboValueManager' zur Verwaltung der möglichen Werte."
        break;

      case "Repository" :
       "Erzeuge neues 'RepositoryTableElement' mit - je nach Tabellen-
         Spalte - Name, Typ oder Kommentar eines Repository-Eintrags,
         der durch Name sowie Name und Typ des enthaltenden Dokuments
         definiert ist sowie dem Repositry-Eintrag selbst,
         der beispielsweise als Sprungziel dienen kann."
       break;
    }
  }

  // Setze weitere Werte in einem zweiten Durchgang.
  for ("Alle Spalten der Tabelle, die ein Repository-Element (Verweis)
    enthalten.") {
```

```
    // Ermögliche die synchorone Behandlung von beispielsweise zwei
    // Spalten mit Informationen über Name und Typ ein und des selben
    // Verweises im Repository.
    "Füge zu jeder Spalte mit Repository-Eintrag Verweise auf alle
      mit ihr zusammenhängenden Spalten der Tabellen-Zeile hinzu."
  }

  "Füge die neu erstellte Zeile zu 'wurzelTabellenElement' hinzu."
}
```

Auf diese Weise wird spaltenweise eine Zeile der darzustellenden Tabelle aufgebaut. Zum Verständnis der verwendeten Objekte bedarf es noch einiger Klärungen. Die Klassen `StringTableElement`, `ComboBoxTableElement` und `RepositoryTableElement` sind abgeleitete Versionen der Klasse `AbstractTableElement` aus deren Objekten die mittlerweile wohlbekannte Baumstruktur – wie gehabt mit den Schnittstellen `IAdaptable` und `IWorkbenchAdapter` – erzeugt wird. Die hier aufgebaute Baumstruktur dient der Darstellung der Tabelle und beinhaltet neben den bereits genannten End-Elementen lediglich Elemente der Klasse `ParentTableElement` als Knoten des Baums. Eines dieser Objekte bildet dabei die Wurzel des gesamten Baumes, und dessen Unterelemente sind wiederum die Wurzelelemente der einzelnen Zeilen. Deren Unterelemente sind ausschließlich die drei Typen von End-Elementen. So kann also mittels der sehr flexiblen Datenverwaltungsstruktur von JFace auch die Darstellung von Tabellen verwirklicht werden.

Die Klasse `ComboValueManager` dient der Verwaltung der Werte eines Tabellenelements. Hierzu stellt sie eine Auswahlliste bereit und sorgt für die Aktualisierung der Liste und ihrer grafischen Darstellung, wenn neue Werte hinzukommen. Die Bearbeitung der Werte in den einzelnen Spalten erfolgt allgemein mittels für jede Spalte – nicht aber für jede Zeile einzeln – definierter `CellEditor`-Elemente, die von JFace bereitgestellt werden, und lediglich für Repository-Elemente den Gegebenheiten durch eine selbst definierte abgeleitete Klasse angepasst werden müssen.

Bleibt noch die Erklärung der Klasse `TreeStructManager`. Diese Klasse dient der Verwaltung des Zugriffs auf einfache Elemente der Baumstruktur, welche die darzustellenden XML-Daten enthält. Ein genauerer Blick auf die Funktionsweise der Klasse erfolgt in einem nachfolgenden Unterabschnitt.

In ähnlicher Weise wie beim Erstellen von Tabellenzeilen gehen auch die anderen in der Klasse `AbstractAutoTableViewer` implementierten Methoden vor. Für von dieser Klasse ableitende Klassen bleiben einige wenige Methoden zu implementieren, die im folgenden Beispiel der Klasse `PrimDef` zu sehen sind. Diese Klasse enthält die Implementierung von vier abstrakten Methoden aus den verschiedenen übergeordneten Klassen. Vom Interface `IPageElement` bleibt die Methode, die den Namen der Klasse als Zeichenkette für die Auswahl durch den `SAPEditor` bereitstellt, in diesem Fall also „PrimDef“. Von den abstrakten Methoden der Klasse `AbstractTableViewer` ist lediglich noch `createTableColumns` zu implementieren. Diese Methode erstellt eine Struktur, in der die zum Aufbau der Tabelle notwendigen Informationen über Anzahl und Beschaffenheit der

gewünschten Spalten enthalten sind. Dies betrifft vor allem den Namen der Spaltenüberschrift sowie die Art der Spalte. Die Tabelle für das „PrimDef“-Element enthält also beispielsweise die Definition einer Spalte mit Namen „Name“ und Typ „String“. Andere mögliche Typen sind „Repository“ und Auswahllisten. Die beiden verbliebenen Methoden überschreiben direkt Methoden aus der Klasse `AbstractAutoTableViewer`. Sie teilen Informationen über die Struktur des verwendeten Knotens der `AbstractTreeElement`-Baumstruktur und seiner Unterelemente mit beziehungsweise über die initialen Werte für einen neu zu erzeugenden Knoten.

Auf diese Weise sind etwa 50 verschiedene Klassen aufgebaut. Sie alle basieren auf dem `AbstractAutoTableViewer` und geben den Inhalt einzelner Knoten des Baumes wieder. In keiner dieser Klassen ist irgendwelche weitergehende Funktionalität implementiert, sie dienen nur als Datenspeicher für die Darstellung „ihrer“ jeweiligen Baum-Knoten mit Hilfe der beschriebenen übergeordneten Klassen.

Einige wenige Klassen benötigen die Darstellung einer Teilmenge ihrer Unterelemente in Tabellenform. Dazu zählt beispielsweise die Klasse `PrimitivesTable`, die mit den Baumknoten vom Typ „Primitive“ arbeitet, aber in ihrer Tabelle Daten nur aus dem Unterelement „PrimDef“ darstellt. Objekte diese Klasse werden in bekannter Weise als Unterelemente von Objekten der Klasse `PrimitivesSection` eingebunden, um einige elementare Informationen über Primitiven zu liefern. Mittels einiger Kunstgriffe funktionieren die hier beschriebenen Mechanismen des Seitenaufbaus und -speicherns auch für diesen Sonderfall. Verantwortlich hierfür ist die von der Klasse `AbstractAutoTableViewer` abgeleitete Klasse `AbstractOverviewAutoTableViewer`, die im Grunde genauso verwendet werden kann wie die ihr übergeordnete Klasse.

Ein letzter zu behandelnder Sonderfall ist die Klasse `MiniTree`. Diese stellt das von ihr verwaltete Baum-Element sowie dessen mögliche Unterelemente in Baumform dar. In der Regel werden Objekte dieser Klasse überall dort verwendet, wo für ein Element der Baumstruktur keine speziell zugeordnete Klasse existiert. Eine prominente Ausnahme ist die Startseite dar, die für das „SAP“- beziehungsweise „MSG“-Element im Baum angezeigt wird. Die hierfür zuständigen Klassen erweitern die `MiniTree`-Klasse um die Möglichkeit, mittels eines Buttons noch nicht vorhandene Unterelemente zu erzeugen. Dies ist notwendig, da die meisten der betroffenen Unterelemente – also der verschiedenen Abschnitte des SAP/AIM-Dokuments, beispielsweise „PrimitivesSection“ – als optionale Elemente definiert sind.

7.7.3.4 Weitere Hilfsmodule

Zunächst soll hier die bereits mehrmals angesprochene Klasse `TreeStructManager` etwas genauer unter die Lupe genommen werden. Es handelt sich dabei um eine Klasse, die das Auslesen einer Baumstruktur in eine Liste von Zeichenketten sowie umgekehrt die Umwandlung einer solchen Liste in eine festgelegte Baumstruktur ermöglicht. Hierzu werden Angaben über die Beschaffenheit der Baumstruktur benötigt, die durch eine Nachbildung des Baums mittels von der Klasse `AbstractTreeStructElement` abgeleiteter Methoden geliefert werden. Um die Elemente der „flachen“ Liste von Zeichenketten eindeutig den Elementen der Baumstruktur zuordnen zu

können, sind nur solche Teile eines Baums für diese automatisierte Art der Bearbeitung erlaubt, in denen Unterelemente nur jeweils null- oder einmal vorkommen. Kommt ein Element doch mehrmals vor, wird nur das erste Auftreten berücksichtigt. Möglicherweise im Verlauf der Bearbeitung entstehende leere optionale Elemente werden mit Hilfe der Methode `normalize` der Klasse `TreeManager` später wieder entfernt.

Die Elemente der verwendeten Baumstruktur sind einfacher gestaltet als die mittlerweile vertraute übliche Variante. Da diese Struktur nur innerhalb des `TreeStructManager` verwendet wird, wurde auf die Implementierung irgendwelcher Interfaces verzichtet. Es existieren abgeleitete Elemente für Daten (`TextTreeStructElement`), Attribute (`AttrTreeStructElement`), Auswahl-Elemente (`ChoiceTreeStructElement`) und gewöhnliche Knoten (`NodeTreeStructElement`). Zusätzlich gibt es eine Klasse `TextNodeTreeStructElement`, mit der vollständige Elemente abgeblidet werden können, die lediglich Daten enthalten. Die genauen Zusammenhänge zwischen den beteiligten Klassen sind Abbildung 7.12 zu entnehmen. Jede dieser Klassen implementiert jeweils die Methoden `getValue` und `setValue` oder `getNode`, `setNode` und `getChildren` der abstrakten Klasse `AbstractTreeStructElement`. Diesen Methoden wird jeweils ein Objekt vom Typ `AbstractTreeElement` übergeben, aus dem dann die entsprechenden Informationen zu ziehen sind. Um einem Knoten bekannt zu machen, welches „seine“ Elemente sind, wird – außer im Falle von Daten, wo dies offensichtlich nicht benötigt wird – im Konstruktor eine Zeichenkette übergeben, die beispielsweise den Namen des XML-Elements oder des Attributs enthält. Die Methode `getChildren` liefert nicht die Unterelemente des Knotens aus dem Baum, der die eigentlichen XML-Daten enthält, sondern diejenigen aus dem Baum, der die XML-Struktur nachbildet. Die entsprechenden Unterelement-Knoten werden im Konstruktor der betroffenen Klassen – `NodeTreeStructElement` und `ChoiceTreeStructElement` – mit übergeben. Auf diese Weise ist es dem `TreeStructManager` möglich, sich durch die Baumstruktur zu hangeln. Der Aufbau einer solchen Nachbildung einer Baumstruktur kann beispielsweise wie folgt aussehen:

```
static final NodeTreeStructElement msgDef =
  new NodeTreeStructElement(
    new AbstractTreeStructElement[] {
      new TextNodeTreeStructElement("Name"),
```

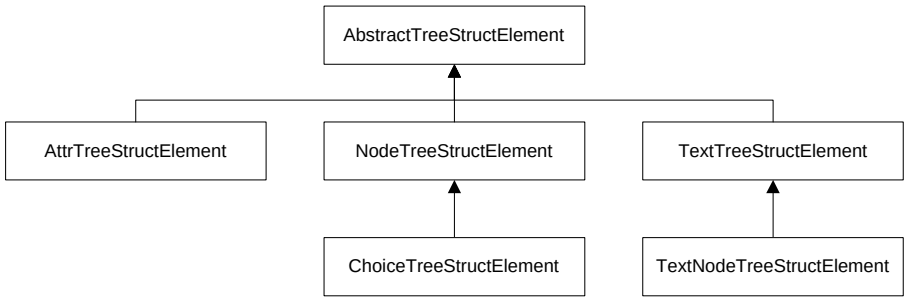

Abb. 7.12: Klassendiagramm: TreeStruct

```
new NodeTreeStructElement(
  "MsgID",
  new AbstractTreeStructElement[] {
    new AttrTreeStructElement("IDType"),
    new AttrTreeStructElement("Direction")}),
new TextNodeTreeStructElement("MsgLenMax"),
new TextNodeTreeStructElement("Version"),
new TextNodeTreeStructElement("Group"),
new TextNodeTreeStructElement("Comment") });
```

In der Praxis sind für alle Baumknoten, die im Editor in Form von Tabellen dargestellt sind, solche Strukturen vorhanden. Die Darstellung von Knoten in je einer Tabellenzeile bedingt bereits, dass jedes Unterelement eines Knotens nur einmal vorkommt, so dass die hier vorgestellte Methode für die Arbeit mit diesem Knoten geeignet ist. Es wäre prinzipiell auch möglich, die benötigten Informationen direkt aus den XML-Schemata zu beziehen. Jedoch wird in der hier vorgestellten Weise die Arbeit mit den Strukturen erheblich vereinfacht, da beispielsweise die Reihenfolge der Elemente fest definiert ist. Eine ausführlichere Diskussion der Vorteile dieser Vorgehensweise fand bereits bei der Vorstellung der generischen Variante der Auswertung von XML-Schemata statt.

In der vorliegenden Implementierung existiert eine Klasse `ElemStructures`, die alle benötigten derartigen Strukturen beinhaltet. Die Zusammenfassung aller dieser Strukturen bringt – gegenüber der Unterbringung in jeweils der Klasse, in der sie gebraucht werden – den Vorzug, dass es, im Gegensatz zum hier gezeigten Beispiel, möglich ist, mehrmals verwendete Elemente – beispielsweise das Element „Name“ – nur jeweils einmal zu definieren und anhand ihres Namens in andere Elemente einzubinden. Dies bringt eine ähnliche Modularität mit sich, wie sie bereits von der Implementierung der grafischen Elemente her bekannt ist. Besonders im Falle von in der Hierarchie weit „oben“ stehenden Elementen wie „Primitive“ oder „Message“ ist der Sinn einer solchen Vorgehensweise sofort einsichtig.

Dasselbe Prinzip wurde auch an einigen anderen Stellen angewandt. So existiert eine Klasse `ElemInitValues` mit den für neue Elemente gewünschten initialen Werten. Diese liegen in von der Klasse `TreeStructManager` verwertbarer Form als Listen von Zeichenketten vor. Weitere zentrale Zusammenfassungen finden sich in den Klassen `TableColumns`, welche die Definitionen für alle anzuzeigenden Tabellenspalten enthält, und `TableColumnsOptional`, die festlegt, welche Elemente einer Tabelle optional sind und somit gegebenenfalls nicht angezeigt werden müssen. Unter den in `TableColumns` enthaltenen Daten finden sich auch die Vorgaben darüber, welches Datenelement einer Zeichenketten-Liste jeweils zu welcher Tabellen-Spalte gehört. Dies und andere enthaltene Daten wird von `AbstractAutoTableViewer` zur Organisation und Anzeige der Daten in den Tabellen genutzt.

Zur Abrundung des Gesagten folgt nun noch eine Pseudocode-Darstellung der Methode `getStringValuesRec` aus der Klasse `TreeStructManager`, die in rekursiver Weise die Liste der Zei-

chenketten aus der Baumstruktur zusammenstellt. Der umgekehrte Vorgang, das Setzen von Werten in der Baumstruktur, erfolgt in analoger Weise.

```
private ArrayList getStringValuesRec(
  AbstractTreeStructElement wurzelStructElement,
  AbstractTreeElement wurzelBaumElement) {
  if ("'wurzelStructElement' hat einen Daten-Wert.") {
    "Füge die Daten zur Liste der zurückzugebenden Zeichenketten hinzu."
  }
  if ("'wurzelStructElement' ist ein Knoten.") {
    for("Alle Unterelemente von 'wurzelStructElement'") {
      "Rufe 'getStringValuesRec' mit dem aktuellen Unterelement von
        'wurzelStructElement' und dem zum aktuellen Knoten gehörenden
        Unterelement von 'wurzelBaumElement' auf und speichere die
        zurückgegebenen Zeichenketten in der Liste der
        zurückzugebenden Zeichenketten."
    }
  }
  "Gebe die geblidete Liste von Zeichenketten zurück."
}
```

Hierbei wurde die korrekte Behandlung von `ChoiceTreeStructElement`-Objekten im Baum der besseren Übersichtlichkeit halber nicht berücksichtigt. Da von den Unterelementen eines XML-„Choice“-Elements nur jeweils eins tatsächlich auftauchen darf, muss diese Bedingung beim Auslesen wie auch bei der Erstellung einer Baumstruktur explizit überwacht werden. Die Funktionsweise der „TreeStruct“-Baumstrukturelemente selbst ist dagegen sehr geradlinig, da sie lediglich die erwähnten Methoden aus `AbstractTreeStructElement` mit Hilfe der zur Verwaltung des XML-Baums allgemein zur Verfügung stehenden Methoden implementieren müssen.

Als letzte der Klassen, die bei der Darstellung und Verwaltung der Daten behilflich sind, sei hier die Klasse `TreeManager` aufgeführt. Neben der bereits kurz beschriebenen Methode `getRootNode` beinhaltet diese die ebenfalls schon mehrmals erwähnte Methode `normalize`. Die eigentliche Hauptaufgabe dieser Methode, das Entfernen von leeren optionalen Knoten im beim übergebenen Knoten beginnenden Dokument, wird von der Methode `normalizeRec` in vertrauter rekursiver Manier erledigt. Mittels der eingebauten Rekursion hangelt sich die Methode durch den übergebenen Baum und entfernt alle Elemente, die leer und optional sind und außerdem keine nicht optionalen oder nicht leeren Unterelemente besitzen. Auf diese Weise wird der Baum „aufgeräumt“ und unnötige leere Zeichenketten in den XML-Daten vermieden. Dies erhöht zum einen die Konsistenz der Daten und erleichtert zum anderen nachfolgenden Programmen die Auswertung. Zum Einsatz kommt die Methode vor jeder Umwandlung der Baumstruktur in das XML-Format. Dies geschieht vornehmlich beim Schreiben der XML-Datei und bei der Aktualisierung der XML-Darstellung im Editor.

8 Zusammenfassung und Ausblick

Im Rahmen der vorliegenden Diplomarbeit wurde der Weg von der Analyse eines bestehenden Prozesses und der Aufdeckung der darin enthaltenen Unzulänglichkeiten hin zur Entwicklung und Fertigstellung einer neuen Lösung beschritten. Die neu entwickelten Bestandteile wurden nahtlos in das vorhandene System integriert. Die wichtigsten Ergebnisse dieses Entwicklungsprozesses sind:

- die Etablierung von XML als dem zur Speicherung der Daten verwendeten Format, und die Entwicklung einer geeigneten Grammatik auf dieser Basis zur Sicherstellung einer robusten, konsistenten und übersichtlichen Form der Datenhaltung;
- die Ermöglichung von Verweisen zwischen Dokumenten und die Bereitstellung komfortabler Mechanismen zur Erstellung und Bearbeitung dieser Verweise mit Hilfe des „Repository“-Managements sowie weitere sinnvolle Hilfen für die Strukturierung der Daten;
- die Einführung eines spezialisierten grafisch orientierten Verarbeitungsprogramms für die betroffenen XML-Dokumente, welches die Erstellung und Wartung der Dokumente in effizienter und übersichtlicher Weise ermöglicht und durch seine Einbindung in die Eclipse-Plattform nahtlos mit anderen Hilfsprogrammen kombiniert werden kann.

Es ist anhand dieser Arbeit unproblematisch möglich, in die Materie einzusteigen, und das Gesamtkonzept des neuen Systems zu verstehen. Auf dieser Grundlage können Planungen für weitere Projekte aufgebaut werden, die ebenfalls XML verwenden, und deren grafische Oberfläche ebenfalls in die Plattform von „Eclipse“ eingebettet ist. So wird zum einen gewährleistet, dass zukünftige Entwicklungen von Software bei der Texas Instruments Berlin AG mit einheitlich zu bedienenden Benutzeroberflächen ausgestattet sind. Zum anderen wird die Interaktion zwischen verschiedenen Modulen erleichtert und der Entwicklungsaufwand zu deren Erstellung durch Verwendung immer derselben Mechanismen reduziert. Auch die benötigte Einarbeitungszeit für Entwickler sowie die Anfälligkeit für Fehler bei der Programmentwicklung wird so reduziert. Ein praktisches Beispiel für ein solches zukünftiges Projekt ist die Erstellung eines Editors für Testfälle. Diese Testfälle simulieren das Hin- und Hersenden von Primitiven im Protokollstack. Dafür greifen sie auf eben jene Strukturen zurück, welche in den SAP-Dokumenten definiert sind. So bietet es sich geradezu an, auch die Daten der Testfälle im XML-Format zu halten und mit den Strukturdaten aus den SAP-Dokumenten zu verknüpfen. Wird die Oberfläche dieses neuen Editors ebenfalls in Eclipse integriert, so kann der Anwender im Idealfall übergangslos gleichzeitig mit dem SAP-/AIM-Editor und dem Testfall-Editor arbeiten. Tatsächlich ist ein entsprechendes

Projekt bei Texas Instruments bereits geplant, und tatsächlich soll dieses auch dem erfolgreichen Beispiel des hier vorgestellten Projekts folgen, und auf XML und Eclipse zurückgreifen.

Mit Abschluß dieser Diplomarbeit stehen die Bestandteile des in ihrem Verlaufe entwickelten Systems in einsetzbare Form zur Verfügung. Jedoch gingen aus den zahlreichen erfolgten Tests und Vorführungen bereits eine Reihe von Wünschen zur Erweiterung und Veränderung hervor. Zudem harren einige bereits geplante Zusatzmodule ihrer tatsächlichen Implementierung. Die wichtigsten Arbeiten im Zusammenhang mit der Weiterentwicklung des bestehenden Systems sind:

- die Entwicklung eines zusätzlichen Views zur Anzeige der – bereits jetzt im Repository verfügbaren – Information darüber, in welchen anderen Dokumenten und Elementen ein gerade ausgewähltes Element der Baumstruktur verwendet wird;
- die inhaltliche Unterstützung des Anwenders bei der Zusammenführung von verschiedenen Versionen eines Dokuments, möglicherweise auch die Eingliederung eines externen Programms zu diesem Zwecke;
- die Verfügbarmachung erweiterter Hilfen zur Semantik der bearbeiteten Daten, beispielsweise bei Versionsnummern und History-Elementen, um dem Anwender die Arbeit zu erleichtern, und gleichzeitig die Konsistenz der Daten zu erhöhen;
- die Durchführung weiterer Tests durch eine größere Zahl von Benutzern.

Vor der breiten Einführung des Systems in den an der Umstellung beteiligten Bereichen der Texas Instruments Berlin AG sind die Behebung einiger weniger neu aufgetauchter Probleme sowie eine gewisse Anpassung der grafischen Oberfläche an aufgekommene Benutzerwünsche geplant. Dies wird, zusammen mit der Vorbereitung der allgemeinen Einführung, nach Abschluss der Diplomarbeit etwa drei Wochen in Anspruch nehmen. Aber auch danach werde ich weiterhin mit diesem – von mir ins Leben gerufenen – Projekt zu tun haben. Ich werde also vermutlich die vorgestellte Liste von Verbesserungen und Erweiterungen selbst implementieren. Daneben werde ich die Wartung des laufenden Systems sowie direkte Hilfestellungen für dessen Anwender mit übernehmen.

Die Arbeit an der vorliegenden Diplomarbeit hat mir bereits viele interessante Erfahrungen und Einblicke ermöglicht. Die eigenständige Entwicklung dieses umfangreichen Projekts und die Evaluierung von in meinem Arbeitsumfeld bisher weitgehend unbekannten Techniken und Methoden haben sich als herausfordernd und lehrreich erwiesen. Ich habe innovative und für mich neue Technologien und Anwendungen kennen gelernt und erfolgreich zum Einsatz gebracht. Zu sehen, dass ich den gebotenen Herausforderungen gewachsen war, und den gestellten Ansprüchen und Erwartungen gerecht werden konnte, war mir Freude und Genugtuung. All dies wird mir bei der Bewältigung künftiger Aufgaben eine wichtige Hilfe und Stütze sein. So blicke ich nun zuversichtlich in die Zukunft und erwarte mit Interesse die Weiterführung meiner Arbeit und alle neuen Herausforderungen, die sich dabei stellen mögen.

Glossar

3GPP	3rd Generation Partnership Project – Zusammenschluss von Unternehmen und Organisation, die bei der Entwicklung und Umsetzung von aktuellen Standards der Telekommunikation eine wesentliche Rolle spielen.
AIM	Air Interface Message – Nachrichten, die zwischen an einer Funkverbindung beteiligten Einheiten ausgetauscht werden. Auch als → MSG abgekürzt.
ASCII	American Standard Code for Information Interchange – Verbreitetes → Datenformat für gewöhnlichen Text im westlichen Alphabet, auch als „Kodierungsformat“ bezeichnet. Soll durch → Unicode ersetzt werden.
Attribut	Im Zusammenhang mit → XML eine Eigenschaft eines → Elements.
AWT	Abstract Windowing Toolkit – Minimalistische Java-Grafikbibliothek von Sun Microsystems. Siehe auch → Swing, SWT.
BNF	Backus Naur Form – Sprache zur Formulierung einer → Grammatik.
Condat AG	Ehemals in Berlin ansässiges Unternehmen, dessen Mobilfunksparte mittlerweile von Texas Instruments aufgekauft wurde und nun unter dem Namen „Texas Instruments Berlin AG“ firmiert.
Datenformat	Nach einer → Grammatik festgelegte Form zur Speicherung von Daten.
dialogbasiert	Interaktion mit dem Benutzer auf der Grundlage von Dialog-Fenstern.
DTD	→ Grammatik für ein in → XML definiertes Format. Ähnlich → BNF. Vorgänger von → XML-Schemata.
Eclipse	Von IBM entwickelte → Plattform, die → Plugin-Programmen allgemeine Funktionalität und eine einheitliche Oberfläche bietet.
Editor	Ein Programm zur Bearbeitung von Daten. Mit einem Texteditor beispielsweise können Texte bearbeitet werden. Im Zusammenhang mit der → Eclipse-Plattform ein Grafikelement, das Informationen aus einer Datei oder einem anderen Datenspeicher darstellt und deren Bearbeitung ermöglicht.
Element	Im Zusammenhang mit → XML ein Objekt, das Daten, → Attribute und andere Elemente enthalten kann.
ETSI	European Telecommunications Standards Institute – eine „non-profit“ Organisation, die sich die Entwicklung von Standards für die Telekommunikation zur Aufgabe gemacht hat.
Event	Ein „Ereignis“, das von einem Programmmodul an andere Module versandt wird, um diese über Veränderungen zu informieren.

Factory Im Java-Klassenkonzept eine Klasse, die sinnvoll konfigurierte Objekte einer anderen Klasse generiert und verwaltet.

freie Software Software, für deren Benutzung kein Entgelt zu entrichten ist und deren Quellcode häufig frei zugänglich ist.

generisch Allgemein verwendbar.

Grammatik Die Definition eines → Datenformats mit Hilfe von Regeln, die zum Beispiel im → BNF-Format vorliegen.

History Verlauf: was bisher geschah.

Implementierung Die Erstellung, Umsetzung, Verwirklichung eines geplanten Projekts.

Instanz Im Zusammenhang mit Java-Klassen ein → Objekt einer → Klasse.

Interface Eine → Schnittstelle.

Interpreter Ein Programm, das den Quelltext von Programmen zur Laufzeit zeilenweise interpretiert und ausführt.

JAXB Java Architecture for XML Binding – Ein Programmpaket, das XML-Daten in Java-spezifischen Objekten abbildet und deren Bearbeitung mit Hilfe von Standardelementen von Java erlaubt.

JAXP Java API for XML Parsing – Ein Programmpaket, das verschiedene Java- → Parser kapselt und so den Zugriff auf diese über eine einheitliche → Schnittstelle erlaubt.

JDE Java Development Environment – integrierte Java Entwicklungsumgebung auf Basis der → Eclipse → Plattform.

Klasse Im Zusammenhang mit Java-Programmierung die Definition einer Einheit, die Daten enthält und Methoden zum Umgang mit diesen Daten. Um deren Funktionalität nutzen zu können, müssen normalerweise zuerst → Objekte daraus erzeugt werden.

Konstante Größe, die in ihrer Form einmal festgelegt wird und fortan unveränderlich ist.

mdf Message Description Format – Einfach automatisiert weiterzuverarbeitendes internes Format für → AIM-Datenstrukturen bei Texas Instruments. Siehe auch → pdf.

MS Word Microsoft Word – Bekanntes Textverarbeitungsprogramm aus dem Hause Microsoft, in vielen verschiedenen Versionen verfügbar und mit den typischen Überraschungseffekten eines WYSIWYG-Programms behaftet.

MSG Message – Historische Bezeichnung für → AIM bei → Condat/ → Texas Instruments.

Objekt Im Zusammenhang mit Java-Programmierung ein nach den für eine bestimmte → Klasse aufgestellten Regeln zur Laufzeit erstelltes Element, das im Rahmen der für die Klasse festgelegten Funktionalität eingesetzt werden kann.

Parser Ein Parser liest nach in einer → Grammatik festgelegten Regeln Daten aus einer Datei und bereitet diese so für nachfolgende Programme auf, dass diese sie einfach weiterverarbeiten können.

pdf Primitive Description Format – Einfach automatisiert weiterzuverarbeitendes internes Format für → Primitive-Datenstrukturen bei Texas Instruments. Siehe auch → mdf.

Perl — Durch einen → Interpreter ausgeführte Scriptsprache, die auf einer Vielzahl von Systemen einsetzbar ist.

Plattform — Basis für Aktivität irgendwelcher Art. Im Falle von → Eclipse eine Grundlage für andere Programme, insbesondere Entwicklungswerkzeuge, die unter anderem verschiedene → Editoren bereitstellt.

Plugin — Im Falle der → Eclipse-Plattform ein Programm, das eine von Eclipse getrennte Entwicklung durchläuft, und dennoch nahtlos in die Plattform eingebunden ist und deren Funktionalität nutzt.

Primitive — Eine Datenstruktur, die Daten oder Steuerinformationen enthält, und die von einem Element eines → Protokollstacks an ein anderes versendet wird. Primitiven sind in einem → SAP definiert und dienen der Kommunikation mit jeweils der Einheit, zu welcher der SAP gehört.

proprietär — Im Alleingang und nicht öffentlich entwickeltes System oder → Datenformat, das häufig rechtlich geschützt und nicht frei zugänglich ist.

Protokollstack — Eine Reihe von idealerweise voneinander unabhängigen Programmmodulen, die gemeinsam die Übertragung von Information über ein irgendwie geartetes Netzwerk ermöglichen.

Pseudocode — Zu Zwecken der Dokumentation von Algorithmen genutzte vereinfachte Form einer Programmiersprache.

Qt — Eine von Trolltech entwickelte Programmierumgebung, deren prominentestes Merkmal die Möglichkeit der Entwicklung von plattformübergreifenden grafischen Benutzerschnittstellen in C++ darstellt. Siehe auch → wxWindows.

Quellcode — Quelltext eines Programmes, also der von Menschen erstellte Text, der mittels spezieller Werkzeuge in ein ausführbares Programm umgewandelt oder direkt ausgeführt wird.

Referenz — → Verweis auf eine andere Stelle. Kann auch die Angabe darüber sein, wo überall ein bestimmtes Element zum Einsatz kommt, welche anderen Elemente also Verweise auf dieses Element beinhalten.

Review — In der Softwareentwicklung die Überprüfung eines Dokuments oder → Quellcodes auf Sinnhaftigkeit durch Personen, die nicht an dessen Entstehung beteiligt sind.

SAP — Service Access Point – Definition einer → Schnittstelle für zumeist → Primitiven in einer Einheit eines → Protokollstacks, um anderen Einheiten dieses Protokollstacks Zugriff auf die von dieser Einheit bereitgestellten Dienste zu ermöglichen.

Schnittstelle — Definierte Methode zum Zugriff auf Daten und Funktionalität, die von der Einheit bereitgestellt werden, in welcher die Schnittstelle eingebaut beziehungsweise implementiert ist. Siehe auch → Interface.

Singleton — Prinzipiell eine → Klasse, die nur ein → Objekt haben kann. Dieses Objekt kann in der Regel von außen abgefragt und verwendet werden. Hier – etwas freier ausgelegt – eine Klasse, die, falls erforderlich, durchaus in mehr als einer Instanz vorhanden sein kann, aber selbst bestimmt, wieviele und welche Objekte von ihr zum Einsatz kommen.

Speichermanagement — Verwaltung des RAM-Speichers. Wird zumeist durch Prozesse im Hintergrund ausgeführt.

Swing Sehr umfangreiche Java-Grafikbibliothek von Sun Microsystems. Siehe auch → AWT, SWT.

SWT Kompakte Java-Grafikbibliothek von IBM. Siehe auch → AWT, Swing.

Tag Markierung für beispielsweise ein Datenelement. Wird in → Textauszeichnungssprachen genutzt.

Texas Instruments Weltweit vertretenes Unternehmen, das unter anderem Soft- und Hardware für Mobiltelefone entwickelt.

Textauszeichnungssprache Eine Sprache, die ein Datenformat begründet, in dem bei den eigentlichen Daten Informationen über diese Daten festgehalten sind, üblicherweise in Form von → Tags.

Toolkette Eine Reihe von Hilfsprogrammen, die aufeinanderfolgend eine bestimmte Aufgabe erfüllen.

Unicode → Datenformat für die Kodierung internationaler Schriftzeichen. Enthält unter anderem alle Zeichen des → ASCII-Formats, aber auch Unterstützung für beispielsweise fernöstliche Sprachen.

Verknüpfung Verbindung eines Elements mit einem anderen. Eine Verknüpfung zeigt auf das Element, mit dem sie verbunden ist. Siehe auch → Referenz, Verweis.

Verweis Ein Element, das auf ein anderes Element verweist, also eine → Referenz zu diesem ist und mit diesem → verknüpft ist.

View Im Zusammenhang mit der → Eclipse-Plattform ein Grafikelement, das Zusatzinformationen zu den im aktiven → Editor dargestellten Daten darstellt. Die Bearbeitung von Daten in einem View geschieht unmittelbar, was bedeutet, daß die veränderten Daten sofort dargestellt und verwendet werden.

W3C World Wide Web Consortium – Eine Vereinigung, die technische Standards für das World Wide Web entwickelt und deren Weiterentwicklung und Verbreitung koordiniert.

Werkzeugleiste Eine Reihe kleiner Bilder (Icons), die, zumeist nebeneinander am oberen Rand eines Fensters angeordnet, den Zugriff auf ihnen zugeordnete Operationen ermöglichen und damit eine Alternative oder Ergänzung zur Verwendung von Menüs bieten.

wxWindows Eine als → freie Software entwickelte Programmierumgebung, deren prominentestes Merkmal die Möglichkeit der Entwicklung von plattformübergreifenden grafischen Benutzerschnittstellen in C und C++ darstellt. Siehe auch → Qt.

XML eXtensible Markup Language – Eine moderne → Textauszeichnungssprache, die eine unkomplizierte Datenhaltung ermöglicht und zu diesem Zwecke heute breite Verwendung findet.

XML-Schema → Grammatik für ein in → XML definiertes Format, die selbst in XML formuliert wird. Nachfolger von → DTDs.

XSLT eXtensible Stylesheet Language for Transformations – Eine Sprache zur automatischen Umwandlung von XML in beliebige Formate.

Literaturverzeichnis

[1] *3GPP*. – URL http://www.3gpp.org. – Zugriffsdatum: 2003-03-27. – Homepage der 3GPP-Organisation

[2] *Abkürzungen im Mobilfunkbereich*. – URL http://www.siemens-mobile.de/mobile-business/CDA/presentation/ap_mb_cda%_presentation_frontdoor/0,2132,85,00.html. – Zugriffsdatum: 2003-03-27. – Siemens AG

[3] *Abkürzungen im Mobilfunkbereich*. – URL http://www.heindl.de/internettipps/glossar.html. – Zugriffsdatum: 2003-03-27. – Heindl Internet

[4] *The Apache XML Project*. – URL http://xml.apache.org. – Zugriffsdatum: 2003-03-30. – Homepage des Apache Projekts

[5] *ETSI*. – URL http://www.etsi.org. – Zugriffsdatum: 2003-03-27. – Homepage der ETSI-Organisation

[6] *Lumrix XML Tools*. – URL http://puvogel.informatik.med.uni-giessen.de/lumrix/#dtd. – Zugriffsdatum: 2003-03-30. – Übersetzung von DTD-Dateien in XML-Schema-Dateien mittels „dtd2xs'

[7] *W3C*. – URL http://www.w3c.org. – Zugriffsdatum: 2003-03-27. – Homepage der W3C-Organisation

[8] MSG Semantic / Texas Instruments. Dezember 2001. – Internes „Technical documentation"-Dokument

[9] Specifying Service Access Points / Texas Instruments. November 2001. – Internes „Technical note"-Dokument

[10] Syntax description for air interface message documents / Texas Instruments. September 2001. – Internes „User guide"-Dokument, Texas Instruments AG

[11] testedit_prjsheet / Texas Instruments. Dezember 2001. – Projekt-Entwurf

[12] *Begriffe aus der Netzwerkwelt*. 2002. – URL http://www.t-lan.de/glossar/glossar.asp. – Zugriffsdatum: 2003-03-30. – Deutsche Telekom Kommunikationsnetze GmbH

[13] Eclipse help system / IBM Corporation und andere. URL http://dev.eclipse.org:8080/help/help.jsp. – Zugriffsdatum: 2003-03-30, 2002. – Programm-Dokumentation. Auch direkt aus dem Eclipse-Programm verfügbar: Menü „Help"->„Help Contents"

[14] Requirements for SAP & MSG editor / Texas Instruments. Februar 2002. – Diskussion über die Anforderungen für Dateiformat und Editor

[15] SAP/MSG Editor / Texas Instruments. März 2002. – Internes „Requirement Analysis“-Dokument

[16] SAP/MSG Editor, XML Input/Output / Texas Instruments. August 2002. – Internes „Low-Level Design“-Dokument. Beschreibung von JAXB und den Möglichkeiten für dessen Einsatz im Editor-Projekt

[17] SAP/MSG Editor, XML Input/Output / Texas Instruments. Oktober 2002. – Internes „Test Specification“-Dokument. Beschreibung der Vorgehensweise zum Test der korrekten Funktionsweise der „early access release“-Version von JAXB

[18] Arciniegas, Fabio: *XML Developer's Guide*. McGraw-Hill Companies, Januar 2001. – ISBN 0072126485

[19] Brand, Thomas ; Habegger, Marc ; Keller, Ramon: XML und JAVA / Hochschule für Technik und Architektur Biel. URL `http://www.hta-bi.bfh.ch/Projects/vsapi/docbkx_html/a679.html`. – Zugriffsdatum: 2003-03-30, 2001. – Projektdokumentation, Anhang

[20] Flanagan, David: *Java in a Nutshell*. O'Reilly, April 2002. – ISBN 0596002831

[21] Harold, Elliotte R.: *Processing XML with Java*. Addison Wesley Longman, November 2002. – URL `http://cafeconleche.org/books/xmljava`. – Zugriffsdatum: 2003-03-30. – Vollständige Onlineausgabe kostenlos verfügbar. – ISBN 0201771861

[22] Harold, Elliotte R. ; Means, W. S. (Hrsg.): *XML in a Nutshell. Deutsche Ausgabe*. O'Reilly, 2003. – ISBN 3897213370

[23] JavaZOOM: Tutorial: XML generation with JAVA. In: *JavaZOOM newsletter* (2002), Juli. – URL `http://www.javazoom.net/services/newsletter/xmlgeneration.html`. – Zugriffsdatum: 2003-03-30

[24] Krüger, Guido: *Handbuch der Java-Programmierung*. Addison-Wesley Longman Verlag GmbH, April 2002. – ISBN 3827319498

[25] Marini, Joe: *The Document Object Model*. Osborne/McGraw-Hill, Juli 2002. – ISBN 0072224363

[26] Tidwell, Doug: *Introduction to XML*. IBM developerWorks, August 2002. – URL `http://www-106.ibm.com/developerworks/edu/x-dw-xmlintro-i.html`. – Zugriffsdatum: 2003-03-30. – Nur online bei IBM verfügbar

[27] Walrath, Kathy ; Campione, Mary: *The JFC Swing Tutorial*. Kap. About the JFC and Swing, Addison-Wesley Professional, Juli 1999. – URL `http://java.sun.com/docs/books/tutorial/uiswing/start/swingIntro.html`. – Zugriffsdatum: 2003-03-30. – Vollständige Onlineausgabe kostenlos verfügbar. – ISBN 0201433214

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Tobias Vogler, geboren am 19. 02. 1978 in Tettnang, ehrenwörtlich,

1. dass ich meine Diplomarbeit mit dem Titel:

   „Datentyp- und Interfaceeditor für Mobilfunkprotokolle auf der Basis von XML“

   bei der Texas Instruments Berlin AG unter Anleitung von Professor Dr. Hedtstück selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als die in der Abhandlung angeführten Hilfen benutzt habe;

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 31. März 2003