Automatische Generierung von Softwarekomponenten auf der Basis von Metamodellen und XML

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Gerd Nusser
aus Bad Saulgau

Tübingen
2005

Tag der mündlichen Qualifikation: 16. Februar 2005
Dekan: Prof. Dr. Michael Diehl
1. Berichterstatter: Prof. Dr. Wolfgang Küchlin
2. Berichterstatter: Prof. Dr. Dietmar Kaletta

Meiner Frau, meinen Eltern und meinen Großeltern.

Vorwort

Die vorliegende Arbeit entstand im Rahmen der Forschungsarbeiten am Arbeitsbereich Symbolisches Rechnen der Universität Tübingen und am Institut für Automatisierung in der Produktion der Fachhochschule Reutlingen in Zusammenhang mit der Entwicklung eines virtuellen Labors des Förderprojektes Virtuelle Universität, gefördert vom Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg.

Meinen besonderen Dank möchte ich Herrn Prof. Dr. Wolfgang Küchlin, der diese Arbeit ermöglicht und betreut hat, aussprechen. Bei Herrn Prof. Dr.-Ing. Gerhard Gruhler bedanke ich mich für die langjährige Unterstützung und für die Chance einen Einblick in die Automationstechnologie zu gewinnen. Zudem bedanke ich mich bei Herrn Prof. Dr. Dietmar Kaletta für die Erstellung des zweiten Gutachtens.

Ein spezieller Dank gilt all denen, die mich bei der Durchführung dieser Arbeit unterstützt haben. Zu nennen sind hier vor allem meine Freunde und Kollegen Michael Friedrich, Stefan Müller und Ralf-Dieter Schimkat. Außerdem danke ich Thomas Schulz für die Korrekturen und vielleicht noch mehr für die langjährige Freundschaft.

Bedanken möchte ich mich außerdem bei meinen Eltern, die mich in jeglicher Hinsicht unterstützt haben. Ihre positive Lebenseinstellung und ihre Werte waren stets eine Bereicherung und werden immer vorbildlich für mich sein. Mein besonderer Dank gilt meiner Frau, die stets zu mir gestanden und mir mit ihren motivierenden Worten neue Kraft gegeben hat.

Gerd Nusser

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1

Motivation, Konzept und Aufbau

1.1 Motivation

Organisatorische Infrastrukturen umfassen eine Vielzahl von Systemen, die auf unterschiedlichen Hardware- und Softwareplattformen basieren. Die Hardware reicht von Mainframes, Workstations und Personal-Computern bis hin zu eingebetteten und in jüngster Zeit auch zunehmend mobilen Systemen. Die Software, die auf diesen Systemen zur Ausführung kommt, basiert auf verschiedenen Betriebssystemen und unterschiedlichen Netzwerkarchitekturen, -protokollen und -diensten [Ber96].

Mit der weiten Verbreitung und dem rasanten Wachstum des World Wide Web (WWW) wird die Vielfalt der Systeme noch größer. Das schnelle Wachstum des Internets und die Allgegenwärtigkeit von Softwaresystemen im täglichen Leben [Par95] führte und führt zu einem steigenden Bedarf von verteilten kooperierenden Anwendungen [Bet94] mit dem Ziel der Integration von verteilten heterogenen Systemen.

Im Umfeld der Automation wird ebenfalls ein steigender Bedarf an der Integration von Systemen sichtbar. Das Ziel ist eine Einbindung von Automationssystemen in eine allgemeine Service- (z.B. Pflege und Wartung) und Management-Infrastruktur (z.B. Logistik) auf Basis von modernen Softwarekonzepten und -technologien [NS01]. Dabei sind die vorhandenen Systeme ebenfalls heterogen und unterliegen meist anderen Gesetzmäßigkeiten bzgl. den zur Verfügung stehenden Ressourcen, z.B. mit Blick auf vorhandene Speicher- oder Rechenkapazität.

Allgemein besteht die Notwendigkeit, heterogene Systeme in eine einheitliche Softwareinfrastruktur mit dem Ziel der Interoperabilität zu integrieren. Die Schwierigkeit einer Integration von Systemen wird dabei durch ihre Eigenschaften bestimmt. Nach Belady [BL79], Bernstein [Ber93], Brown [Bro99] und Yellin [Yel01], besitzen Systeme die folgenden Eigenschaften:

- Systeme sind komplex.
- Systeme verändern sich.

Systeme bestehen aus einer Vielzahl von heterogenen Komponenten, die auf unterschiedlichen Rechnern mit unterschiedlichen Betriebssystemen zur Ausführung kommen. Die Vielzahl der Komponenten und die Tatsache, dass heterogene Systeme, auch über Systemgrenzen hinweg, integriert werden, tragen zur inhärenten Komplexität von Systemen bei. Die Komplexität wird zwar durch die Verwendung von bestimmten Technologien, wie z.B. von Middleware- und Komponentensystemen reduziert, jedoch bergen diese Systeme ebenfalls wieder eine gewisse Komplexität in sich.

Gemäß Belady [BL79] unterliegen alle großen und weit verbreiteten Systeme dem „Phänomen der kontinuierlichen Veränderung“, d.h. sie sind Gegenstand eines fortwährenden Prozesses von „Korrekturen, Verbesserungen, Anpassungen und Diversifikationen, da es meistens kostengünstiger ist, vorhandene Software zu modifizieren, als sie neu zu entwickeln“. Falls neue Systeme in eine vorhandene Infrastruktur integriert werden sollen, ist die zur Verfügung stehende Zeit zudem oft so kurz, dass die Entwicklung eines neuen Systems keine sinnvolle Alternative darstellt [Emm00].

Die Veränderungen betreffen nicht nur die Systeme selbst, sondern auch die Technologien, mit denen die Systeme realisiert und integriert werden. Entwickler sehen sich ständig mit neuen Technologien konfrontiert, die es zu beherrschen und anzuwenden gilt. Ein aktuelles Beispiel hierfür sind Web Services [Wor02] zur Integration von Diensten in das WWW.

Die Verwendung von Middleware- und Komponententechnologien trägt wesentlich zur Integration von Systemen bei. Middleware stellt die notwendige Funktionalität zur verteilten Kommunikation zur Verfügung. Der Schwerpunkt von Komponentensystemen liegt in der Integration von Komponenten, oftmals unter Verwendung einer bestimmten Middleware-Technologie.

Obwohl Middleware zweifellos viele Probleme lösen kann, stellt sie kein „Allheilmittel“ dar [Ber93]. Sie unterliegt, wie alle Systeme, dem Gesetz der kontinuierlichen Veränderung, da immer wieder neue Anforderungen an Systeme gestellt werden. Es wurden bereits zahlreiche Middleware-Technologien entwickelt, die erfolgreich eingesetzt werden. Der entfernte Prozeduraufruf RPC [BN84], die Common Object Request Broker Architecture CORBA [Obj00], das verteilte Komponenten-Modell DCOM [Red97] oder die Java Remote Method Invocation RMI [Sun04] gelten, neben zahlreichen anderen Technologien, als Schlüsseltechnologien bei der Realisierung von verteilten Systemen. Diese nicht vollständige Aufzählung von existierenden Middlewaretechnologien und -konzepten macht deutlich, dass auch in Zukunft immer wieder neue Technologien und Konzepte entwickelt werden und keine einzelne von ihnen allen Anforderungen gerecht werden wird. Dabei fällt die Entscheidung, welche Middlewaretechnologie eingesetzt wird, aufgrund der großen Vielfalt an angebotener Middleware, mit teilweise nur sehr kleinen Unterschieden, sehr schwer. Sobald eine bestimmte Architektur für ein System ausgewählt ist, ist es extrem teuer, wenn nicht gar unmöglich, die getroffene Wahl rückgängig zu machen und eine andere Middlewarearchitektur einzusetzen [Emm00, Pur94]. Weiterhin gilt, dass moderne Middlewareinfrastrukturen zwar die Integration von Anwendungen fördern, „Kompatibilität aber eher die Ausnahme, als die Norm“ [Sch03] darstellt.

Die Gesetzmäßigkeiten von Systemen allgemein und die der Middleware im speziellen

gelten gleichermaßen auch für Komponentensysteme. Thomason stellt bei der Einordnung von Komponentensystemen in einen 3-dimensionalen Raum, bestimmt durch die Dimensionen Verteiltheit, Modularität, Plattform- und Sprachunabhängigkeit, fest, dass „entgegen der populären Meinung sowohl CORBA als auch Java das Nirvana [1,1,1][1]“ verfehlen [Tho00].

Ein wichtiger Punkt bei der Entwicklung von Systemen ist die Wiederverwendung von existierender Software [Kru92, Sam97]. Dabei fließen einmal getätigte Investitionen möglichst oft in einen Entwicklungsprozess ein, so dass eine kürzere Entwicklungszeit und eine höhere Qualität neuer Software erreicht werden kann. Die Wiederverwendung von Software ist u.a. ein Ziel von objektorientierten und komponentenbasierten Systemen. Während objektorientierte Sprachen ihr Versprechen bzgl. der Wiederverwendung von Software gemäß Kiely [Kie98] nicht erfüllen konnten und laut Udell [Ude94] sogar gescheitert sind, werden komponentenbasierte Systeme als die Systeme der Zukunft gesehen.

1.2 Konzept

In dieser Arbeit wird eine Vorgehensweise und eine Plattform zur automatischen Erzeugung von Softwarekomponenten zur Integration von heterogenen Softwaresystemen vorgestellt. Eine Komponente ist dabei eine Softwareeinheit mit einer definierten Schnittstelle (s. Abschnitt 2.3.1.1). Die erzeugten Komponenten dienen als Mediatoren zwischen unterschiedlichen Infrastrukturen bzw. Technologien.

Ein wichtiger Aspekt des in dieser Arbeit realisierten Systems ist die Wiederverwendung existierender Software bzw. existierender Softwarekomponenten. Der Komplexität und Evolution von Middleware- und Komponentensystemen wird insofern Rechnung getragen, als die notwendige Software zur Integration von Systemen automatisch generiert wird und das notwendige Wissen dem Benutzer gegenüber nahezu transparent ist.

Um dem Trend hinsichtlich einer allgemeinen Infrastruktur von kooperierenden Diensten und Systemen zu genügen, ist es entscheidend, von Details bzgl. unterschiedlicher Hard- und Software zu abstrahieren. Das Ziel ist eine Vernetzung von heterogenen Systemen basierend auf einheitlichen Standards zur transparenten Nutzung von verteilten und lokalen Diensten [SNB00].

Das in der vorliegenden Arbeit realisierte System namens MAX (Metadata Architecture based on XML) umfasst sowohl das Konzept, als auch die Architektur und Plattform zur Metaisierung, Repräsentation und Generierung von Softwarekomponenten. Konzeptionell werden Metamodelle unterschiedlicher Systeme, teilweise unter Verwendung bestimmter Middleware, in ein allgemeines Metamodell abgebildet und dieses allgemeine Metamodell zur Generierung von Softwarekomponenten verwendet. Abb. 1.1 zeigt eine schematische Übersicht des Gesamtkonzepts von MAX.

Die MAX zugrundeliegende Idee ist, die Schnittstelle einer Quell- bzw. Ursprungskomponente mittels Reflexion bzw. Introspektion automatisch zu extrahieren, im Zuge der Abstraktion in ein allgemeines Metamodell abzubilden und auf Basis dieses, auch se-

[1] d.h. verteilt, modular, plattform- und sprachunabhängig

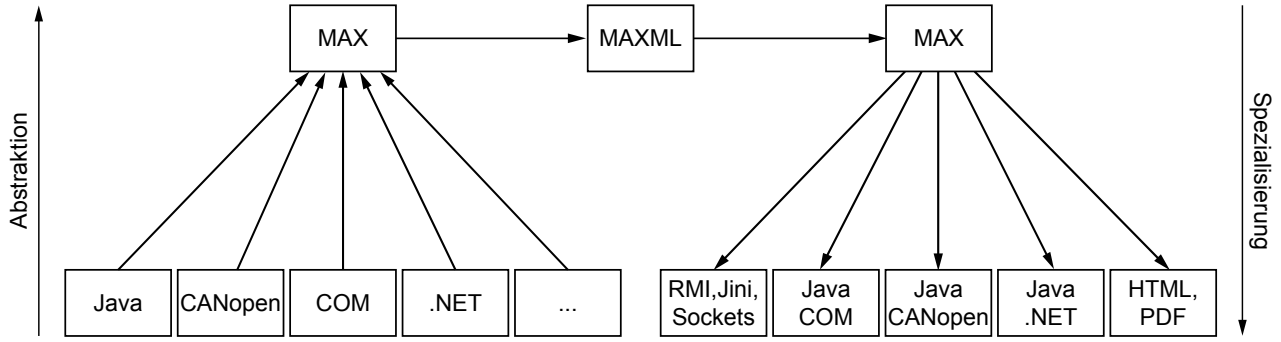


Abbildung 1.1: Schematische Übersicht von MAX

rialisierten, Metamodells Softwarekomponenten zur Integration der Quellkomponente durch Spezialisierung weitestgehend automatisch zu erzeugen. Ohne der Definition eines Metamodells in Abschnitt 3.1.6 vorwegzugreifen, ist ein Metamodell ein Modell, das die Elemente, die im Rahmen einer Modellierung verwendet werden können, beschreibt. Bei der objektorientierten Programmierung wird beispielsweise ein Gegenstand der Realität durch eine Klasse repräsentiert bzw. modelliert. Eine Klasse wird im Metamodell durch eine Metaklasse repräsentiert und ist die Beschreibung der Klasse selbst.

Die Architektur von MAX basiert auf einem verallgemeinerten Metamodell zur schnittstellenbasierten Repräsentation von Komponenten, der Serialisierung dieser Repräsentation unter Verwendung der *eXtended Markup Language* (*XML*) [Wor98b] und der automatischen Codegenerierung.

Basierend auf der Fähigkeit einer Komponente, Informationen über sich selbst zu liefern, wird eine allgemeine Beschreibung dieser Komponente erzeugt und zur automatischen Generierung von Code benutzt. Der erzeugte Code dient als Mediator zwischen der ursprünglichen Komponente und einer bestimmten Technologie. Dabei steht die Verwendung und Wiederverwendung von existierenden Komponenten im Vordergrund.

Die vorgestellte Architektur unterstützt prinzipiell sämtliche Programmiersprachen, die über eine prozedurale, eine objektorientierte oder eine zu der prozeduralen bzw. objektorientierten Schnittstelle analoge Schnittstelle verfügen. Eine Voraussetzung für die Einbindung eines Systems in MAX ist, dass ein System bzw. eine Architektur entweder direkt, d.h. aus der jeweiligen Programmiersprache, oder indirekt, d.h. über eine wie auch immer geartete Schnittstelle in Java integriert werden kann. Aktuell werden von MAX die Software-Infrastrukturen Java, COM, .NET und CANopen unterstützt. Die Quellkomponenten können entsprechend ihres Typs in unterschiedliche Systeme und in unterschiedliche Middleware-Technologien integriert werden. Hauptsächlich wird die Integration von Systemen in Java, in die verteilten Middlewaretechnologien RMI[Sun04], Jini[Sun99a], in Ansätzen .NET Remoting[Mic01a], Web Services [Wor02] und CORBA [Obj00] und in die Middlewaretechnologie JNI [Lia99] unterstützt. Zur Dokumentation von Schnittstellen stehen die Dokumentationsformate HTML und PDF zur Verfügung.

Durch eine automatische Extraktion der Schnittstelle von Komponenten, eine plattform- und sprachunabhängigen Beschreibung dieser Schnittstelle mit XML und eine automa-

tische Generierung von Softwarekomponenten zur Kapselung der ursprünglichen Komponenten, ergeben sich die folgenden Vorteile [NG00]:

1. Existierende Komponenten, die über keine Schnittstelle zur verteilten Kommunikation verfügen, können einfach in eine verteilte Umgebung integriert werden. MAX unterstützt die Integration von Java-, COM-, .NET- und CANopen-Komponenten in unterschiedliche Middlewareinfrastrukturen.

2. Der Integrationsprozess ist unabhängig von einer bestimmten Middleware-Infrastruktur. Es wird davon ausgegangen, dass die zu integrierenden Komponenten zwar von einer bestimmten System- bzw. Softwarearchitektur abhängig sind, aber nicht von einer bestimmten Middleware. Demzufolge können unterschiedliche Middlewarekomponenten für unterschiedliche Middlewareinfrastrukturen und jeweilige Architekturen generiert werden.

3. Komponenten aus unterschiedlichen Systemen können, falls eine entsprechende Schnittstelle vorhanden ist, einfach integriert werden. Dabei werden die notwendigen Mediatoren automatisch generiert. MAX unterstützt beispielsweise die Integration von CANopen, COM und .NET in Java.

4. Durch eine allgemeine, plattform- und sprachunabhängige, schnittstellenbasierte Repräsentation von Komponenten in XML können Komponenten aus unterschiedlichen Systemen verarbeitet werden. XML ist standardisiert, plattform- und sprachunabhängig, offen und im Hinblick auf zukünftige Entwicklungen erweiterbar. Zudem können XML-Dokumente einfach übertragen und mit Standard-Software, z.B. XML-Bibliotheken und -Werkzeugen, verarbeitet werden.

5. Bereits erzeugte und serialisierte Beschreibungen können zur Erzeugung von unterschiedlichen Komponenten und Dokumentation beliebig oft verwendet und somit wiederverwendet werden. Außerdem ist es möglich, diese persistenten Beschreibungen nach bestimmten Kriterien zu durchsuchen bzw. auf bestimmte Eigenschaften hin zu untersuchen.

6. Komponenten können so zur Laufzeit generiert werden, dass die Plattform eine Basis für dynamische Umgebungen, z.B. für mobile Anwendungen, darstellt [FNK02]. Mobile Anwendungen, z.B. Software-Agenten, haben somit die Möglichkeit, bestimmte Komponenten zu suchen, woraufhin die notwendigen Komponenten zur Kommunikation mit den Agenten dynamisch und ohne Benutzerinteraktion generiert werden können [FTNK04].

7. Eine automatische Generierung von Beschreibungen basierend auf dem Konzept der Reflexion und eine automatische Generierung von Softwarekomponenten ist effizienter und weniger fehleranfällig als eine manuelle Erzeugung, Implementierung und Integration. Dies führt zu einer wesentlichen Verkürzung der Entwicklungszeit und zu einer höheren Qualität sowohl von lokalen als auch von verteilten Applikationen. Zudem wird von einem Entwickler ein geringeres Maß an Spezialwissen vorausgesetzt, so dass er sich auf seine eigentliche Kernkompetenz, die Anwendungsprogrammierung, konzentrieren kann.

1.3 Aufbau

Die vorliegende Arbeit befasst sich mit sehr unterschiedlichen Konzepten der Informatik. Die Konzepte und theoretischen Grundlagen, welche die Realisierung des Systems betreffen bzw. benötigt werden, werden dem jeweiligen Kapitel vorangestellt, so dass auf ein allgemeines einführendes Kapitel mit allen Grundlagen verzichtet wird. Im Einzelnen besitzt die Dissertation folgenden Aufbau.

Kapitel 2 enthält eine allgemeine Einführung in Systeme und diskutiert die Problematik von Systemen und Programmiersprachen hinsichtlich deren Integration in unterschiedliche Infrastrukturen. Ein wesentlicher Aspekt ist dabei die Wiederverwendung existierender Software. Als Beispiele für die Integration von Systemen werden unterschiedliche Middleware- und Komponentensysteme vorgestellt. Zudem enthält dieses Kapitel eine allgemeine Einführung in XML, CAN und CANopen.

Kapitel 3 gibt zunächst einen Überblick über das Konzept der Reflexion und eine Einführung in Metamodelle. Zur Abbildung von Komponenten in das entsprechende Metamodell, der sogenannten Metaisierung, finden reflektive Eigenschaften der jeweiligen Technologie Verwendung. Das realisierte System MAX unterstützt eine automatische Metaisierung von Java-, COM-, .NET-, CANopen- und ansatzweise von CORBA-Komponenten.

Kapitel 4 stellt auf Basis der in Kapitel 3 vorgestellten Metamodelle der unterschiedlichen Programmiersprachen und Technologien ein allgemeines Metamodell und dessen Serialisierung in XML vor. Das allgemeine Modell bietet eine schnittstellenbasierte Abstraktion von Komponenten und abstrahiert von spezifischen Eigenschaften einer bestimmten Technologie. Das serialisierte Metamodell dient als Zwischenformat zur Speicherung und zum Austausch des allgemeinen Metamodells. Zur Serialisierung und Deserialisierung des allgemeinen Metamodells steht eine komfortable Schnittstelle zur Verfügung, so dass Modelle importiert und exportiert werden können.

Kapitel 5 zeigt, wie das serialisierte Metamodell zur automatischen Generierung von Software verwendet wird. Auf Basis von Schnittstellenbeschreibungen werden Komponenten in unterschiedliche Infrastrukturen integriert. Entsprechend der Beschreibung einer Quellkomponente werden die notwendigen Komponenten zur Interaktion bzw. Integration mit einer bestimmten Technologie automatisch erzeugt und die Quellkomponente gekapselt. Das System unterstützt entsprechend der Quellkomponente die Generierung von Komponenten zur Integration von COM, .NET und CANopen in Java, die Generierung von Middlewarekomponenten für RMI, Jini, .NET, die Generierung von Webservices und die Generierung von Komponentenbeschreibungen im PDF- und HTML-Format.

Kapitel 6 erörtert das Gesamtkonzept und die Architektur des realisierten Systems, wobei die einzelnen Systemkomponenten und deren Interaktion im Detail erläutert werden. Die Möglichkeiten des Systems werden am Beispiel von mehreren, mitunter auch theoretischen Fallstudien erörtert. Die Fallstudien umfassen u.a. die

Integration von CANopen-Systemen in Jini und die Interaktion von CANopen-Systemen mit einer Java3D-Simulation eines Roboters.

Kapitel 7 enthält einen Zusammenfassung der Arbeit, einen Ausblick auf mögliche Erweiterungen und eine Auflistung der veröffentlichten Publikationen.

2

Grundlagen

In diesem Kapitel werden in Abschnitt 2.1 zunächst Systeme im Allgemeinen und in Abschnitt 2.3 Middleware- und Komponentensysteme im Besonderen eingeführt und hinsichtlich ihrer Eigenschaften und Qualitätskriterien betrachtet.

Das Problem, umfangreiche und zuverlässige Systeme kostengünstig zu erstellen, war bereits Gegenstand der NATO Software Engineering Konferenz 1968 [Kru92]. Seit diesem Zeitpunkt existiert der Begriff der Softwarekrise, deren Bewältigung man u.a. in der Wiederverwendung von Software sah [Boe87, Sta84].

In Abschnitt 2.2 wird auf die Wiederverwendung von Software näher eingegangen. Zudem werden unterschiedliche Konzepte und Techniken bezüglich der Wiederverwendung erörtert. Während die objektorientierte Programmierung lange Zeit als die Zukunftstechnologie für die Wiederverwendung von Software gesehen wurde [PD93], sind Kiely [Kie98] und Udell [Ude94] der Ansicht, dass die Zukunft von komponentenbasierten Systemen bestimmt wird. Ein Ziel komponentenbasierter Software ist die Verwendung und Wiederverwendung von Komponenten, die in unterschiedlichen Sprachen entwickelt und auf unterschiedlichen Plattformen lauffähig sind. Dabei nimmt die Technik zur Verbindung von Komponenten (engl. „glue“ bzw. Middleware) eine zentrale Rolle ein [BBB+99]. Komponentenbasierte Systeme werden in Abschnitt 2.3 näher erläutert.

Die Ergebnisse und Konsequenzen, die sich aus der Betrachtung der einzelnen Systeme für die in dieser Arbeit realisierten Konzepte ergeben, werden – nach einer Einführung in die technologischen Grundlagen zu XML, CAN und CANopen – am Ende des Kapitels nochmals zusammengefasst.

2.1 Systeme

Entsprechend Abschnitt 1.1 umfassen organisatorische Infrastrukturen eine Vielzahl von Systemen, die auf unterschiedlichen Hardware- und Softwareplattformen basieren. Die Realisierung von Systemen für diese Plattformen hängt maßgeblich von den zur Verfügung stehenden Technologien und Konzepten ab. Im Rahmen dieser Arbeit werden

sowohl Software- als auch Hardwaresysteme betrachtet, wobei sich die Unterscheidung der Hardware auf die verfügbaren Ressourcen (z.B. Speicher) und nicht auf die Art der Hardware (z.B. Prozessortyp) bezieht.

2.1.1 Definitionen

Sämtliche Ressourcen einer organisatorischen Einheit bilden ein Computersystem (bzw. Compusystem [Yel01]). Ein Computersystem, im Folgenden als *System* bezeichnet, ist ein rechnerbasiertes System zur Beantwortung von Fragen bzw. Ausführung von Aktionen innerhalb einer bestimmten Domäne [Mae87]. Ein System umfasst sämtliche Hardware- und Softwarekomponenten und die Middleware, die die Komponenten miteinander verbindet. Eine Komponente ist in diesem Zusammenhang eine identifizierbare Softwareeinheit mit einer definierten Schnittstelle zur Interaktion. Auf das Thema Middleware wird in Abschnitt 2.3.2 noch genauer eingegangen.

2.1.2 Eigenschaften

Gemäß Abschnitt 1.1 sind Systeme komplex und unterliegen dem Phänomen der kontinuierlichen Veränderung. Die Komplexität von Systemen betrifft die Sprachen und Technologien, die zur Realisierung eines Systems Verwendung finden. Moderne Systeme bestehen aus hunderten von heterogenen Komponenten auf zehn, hundert oder tausend unterschiedlichen Rechnern mit unterschiedlichen Betriebssystemen. Da ständig neue Anforderungen und Bedürfnisse an Systeme gestellt werden, unterliegen sie dem Phänomen der kontinuierlichen Veränderung. Systeme müssen in der Regel ständig korrigiert, verbessert, angepasst und erweitert werden, „da es kostengünstiger ist, bestehende Systeme zu erweitern, anstatt sie neu zu entwickeln“ [BL79].

Die Integration von neuen Anwendungsdomänen ist u.U. von vollständig anderen Parametern bestimmt. So unterliegen mobile oder eingebettete Systeme üblicherweise anderen Bedingungen als Desktop-Systeme. Eingebettete Systeme erfüllen aufgrund der zur Verfügung stehenden Ressourcen nur minimale Anforderungen, so dass beispielsweise nicht vorausgesetzt werden kann, dass ein eingebettetes System über eine Java 1.1 kompatible Laufzeitumgebung verfügt. Mobile Systeme benötigen zudem eine flexible Anpassung an sich verändernde Verbindungen [KC00] und an eine sich verändernde Umgebung.

Die kontinuierliche Veränderung betrifft nicht nur die Systeme selbst, sondern auch die Software, die zur Realisierung von Systemen benutzt wird. Software durchläuft bestimmte Entwicklungszyklen und wird, vor allem in ihren Anfängen, häufig modifiziert. Bei der objektorientierten bzw. prozeduralen Programmierung wird die Schnittstelle durch die Methoden bzw. Prozeduren bestimmt. Die Änderungen einer Schnittstelle beziehen sich sowohl auf die Namen, als auch auf die Signatur von Methoden bzw. Prozeduren. Bei der Signatur kann sich der Rückgabetyp, die Anzahl der Parameter, deren Reihenfolge oder deren Typ verändern. Potentielle Vertreter dieser Kategorie von Software sind z.B. Java [GJSB00] oder die Java-Plattform Jini [Sun99a]. Im günstigsten Fall wird die veraltete Schnittstelle nach wie vor unterstützt und bei der Über-

setzung von Quellcode auf die geänderte Schnittstelle durch den Compiler hingewiesen. Im schlechtesten Fall führt die Übersetzung zu einem Fehler. Neue Software-Versionen oder Anwendungen können zudem neue Eigenschaften in Form von zusätzlichen Methoden, also zusätzlicher Funktionalität besitzen. Die neuen Versionen der Microsoft Office-Anwendungen Word 2000, Excel 2000, usw. bieten beispielsweise mehr Funktionalität als deren Vorgänger. Die ursprüngliche Schnittstelle zur Interaktion mit den Anwendungen kann zwar weiter genutzt werden, entspricht aber natürlich nicht mehr dem aktuellen Stand der Software.

Die Evolution von Software ist ein „natürlicher“ Prozess, da es immer wieder neue Erkenntnisse, neue Erfahrungen und neue Anforderungen gibt, die in bestehende Software integriert werden. Für den Erfolg und die Akzeptanz eines Systems, einer Architektur, einer Technologie oder einer Anwendung ist jedoch ein gewisses Maß an Beständigkeit notwendig.

2.1.3 Qualitätskriterien

Zur Bewältigung der Komplexität und der Fähigkeit zur Anpassung, bedarf es Software, die bestimmten Qualitätskriterien genügt. Laut Bernstein [Ber93] kann die Qualität von Systemen, Anwendungen und Middleware anhand folgender Kriterien, den sog. allgemein gültigen Attributen (engl. *pervasive attributes*), beurteilt werden:

- Die **Verwendbarkeit** eines Systems wird durch das Maß bestimmt, in wie weit ein System einen Benutzer bei der Erledigung von dessen Arbeit effizient unterstützt, wobei ein Benutzer ein Administrator, ein Entwickler oder ein Endbenutzer sein kann.

- Durch die **Verteiltheit** eines Systems besteht die Möglichkeit, auf Dienste über ein Netzwerk zuzugreifen. Die Verteiltheit sollte dabei für einen Benutzer möglichst transparent sein.

- Die **Integrierbarkeit** beschreibt die Fähigkeit von Komponenten, miteinander zu interagieren. Integrierbarkeit umfasst die *Interoperabilität* und die *Uniformität*. Die Interoperabilität beschreibt die Fähigkeit von Komponenten, effektiv zusammen zu arbeiten und zu kommunizieren. Die Uniformität beinhaltet die Beständigkeit und die Einheitlichkeit von Komponenten.

- Durch die **Konformität zu Standards** wird die Portabilität und die Interoperabilität von Komponenten und Anwendungen gewährleistet. Ein Standard hat üblicherweise positive Auswirkungen auf die Kosten, da durch eine weite Verbreitung große Stückzahlen erreicht werden und der Preis durch mehrere konkurrierende Hersteller bestimmt wird. Ein Punkt, der auch bei der Erweiterbarkeit von Systemen zum Tragen kommt, ist deren Offenheit; offen im Sinne von Standards und offen im Sinne von verfügbaren Schnittstellen. Ein offener Standard bildet die Basis für eine breite Akzeptanz einer Technologie.

- Die **Erweiterbarkeit** beschreibt die Möglichkeit, ein vorhandenes System an neue Anforderungen anzupassen. Es sollte immer möglich sein, ein bestehendes System zu erweitern, ohne dessen Funktionalität negativ zu beeinflussen. Offene Schnittstellen erlauben eine Integration von Funktionalität und somit eine Erweiterung eines Systems.

- Mit einer ständig wachsenden globalen Interaktion ist die **Internationalisierbarkeit**, d.h. die Anpassung an unterschiedliche Sprachen, ein wichtiger Faktor. Die Internationalisierung einer Software sollte ohne großen Aufwand möglich sein.

- Die **Verwaltbarkeit** beinhaltet die Möglichkeit zur Konfiguration, Diagnose, Kontrolle und Wartung eines Systems. Sämtliche Ressourcen sollten innerhalb des Systems auf ähnliche Weise verwaltbar sein.

- Die **Leistungsfähigkeit** ist die Fähigkeit eines Systems, innerhalb einer bestimmten Zeitspanne korrekte Ergebnisse zu liefern. Die Leistungsfähigkeit wird durch die Anwortzeiten, die Effizienz, den Durchsatz und die Nutzung von Systemen bestimmt.

- Die Übertragbarkeit von Software auf anderen Plattformen wird als **Portabilität** bezeichnet. Bei der Realisierung von portablen Systemen müssen deshalb plattformunabhängige Schnittstellen entwickelt und plattformabhängige Komponenten isoliert werden.

- Die **Zuverlässigkeit** wird durch die Zeit bestimmt, während der ein System verfügbar ist und die erwarteten Ergebnisse liefert. Von einigen Systemen wird z.B. erwartet, dass sie 24 Stunden am Tag und 365 Tage im Jahr zur Verfügung stehen.

- Die **Skalierbarkeit** beschreibt die Fähigkeit eines Systems sowohl kleine, als auch große Probleme bewältigen zu können.

- Der Schutz von Information vor unbefugtem Zugriff wird durch die **Sicherheit** eines Systems bestimmt. Dies betrifft sowohl die Authentifizierung als auch die Verschlüsselung von Information.

Diese Kriterien betreffen nicht nur Systeme als Ganzes, sondern auch die einzelnen Systemkomponenten. D.h., die Qualität eines Systems wird maßgeblich von der Qualität der betreffenden Komponenten bestimmt. Die vorgestellten Architekturen und Technologien, sowie das in dieser Arbeit realisierte System, werden entsprechend ihrem Typ anhand einiger dieser Qualitätskriterien beurteilt.

2.2 Softwarewiederverwendung

Die Wiederverwendung von Hardwarekomponenten ist im Gegensatz zur Wiederverwendung von Software eine Selbstverständlichkeit. Es scheint, als funktionierten die Konzepte der Wiederverwendung bei Hardware wesentlich besser als bei Software. Der

Grund hierfür ist wiederum in der Komplexität von Software zu suchen. Hardwarekomponenten besitzen eine definierte Schnittstelle, die normalerweise nicht verändert werden kann. Die Schnittstelle von Software scheint wesentlich variabler zu sein, dies ist aber für die Integrierbarkeit von Software von entscheidender Bedeutung.

2.2.1 Definition

Der Begriff der Wiederverwendung beschreibt den Prozess, Probleme zu lösen und die Lösung auf gleichartige Probleme anzuwenden. Dabei ist die Wiederverwendbarkeit ein Maß für die Schwierigkeit, vorhandene bzw. bereits gewonnene Ergebnisse bei anderen Aufgabenstellungen nutzbringend zu verwenden. Mit der Wiederverwendung soll gewährleistet werden, dass aus bereits getätigten Investitionen und erworbenem Wissen der größtmögliche Nutzen gezogen wird.

Die Softwarewiederverwendung verfolgt die genannten Ziele im Bereich der Softwaretechnik. Es existieren zahlreiche Definitionen des Begriffs der Softwarewiederverwendung. In dieser Arbeit wird die allgemeine Definition von Krueger [Kru92] herangezogen:

> „Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.“

Krueger beschreibt mit dieser Definition eine softwarezentrierte Ansicht der Wiederverwendung. Zwar werden in dieser Arbeit auch Beschreibungen bzw. Spezifikationen wiederverwendet, diese sind jedoch eng an Softwarekomponenten gekoppelt, so dass die Definition keine Einschränkung darstellt.

2.2.2 Eigenschaften

Die Wiederverwendung von Software wirkt sich laut Sametinger positiv auf die Kosten, die Produktivität, die Qualität und die Zuverlässigkeit von Systemen aus [Sam97].

- **Kosten** Sowohl die Kosten der Implementierung als auch die Kosten für die Wartung eines Systems können durch die Wiederverwendung von Software gesenkt werden. Bei der Implementierung von Systemen werden oftmals viele Teile redundant entwickelt. Falls diese Teile in Form von wiederbenutzbaren Komponenten bereitgestellt werden, können die Entwicklungskosten wesentlich gesenkt werden. Durch die Verwendung von bereits getesteten Komponenten kann davon ausgegangen werden, dass ein System weniger gewartet werden muss und somit die Kosten minimiert werden.

- **Produktivität** Durch die Verwendung von bereits existierender Software muss weniger Code implementiert werden. Vorhandene Software kann in eine bestimmte Umgebung integriert werden, ohne dass sie neu entwickelt werden muss. Die notwendige Zeit zur Anpassung von bestehender Software ist geringer als eine

Neuentwicklung und hängt in der Regel von der Strategie selbst bzw. den zur Verfügung stehenden Werkzeugen ab. Sametinger gibt zu bedenken, dass die Produktivität durch einen gewissen Einarbeitungsaufwand und die Implementierung von wiederverwendbaren Komponenten zwar kurzzeitig abnimmt, dieser zusätzliche Aufwand auf längere Zeit jedoch kompensiert wird.

- **Qualität** Die mehrmalige Verwendung von Komponenten führt zu einer höheren Qualität einzelner Komponenten und damit zu einer höheren Qualität eines Gesamtsystems. Eine Voraussetzung hierfür ist, dass die Komponenten entsprechend verwaltet und gewartet werden.

- **Zuverlässigkeit** Die Verwendung von bereits getesteten Komponenten führt zu zuverlässigeren Gesamtsystemen. Die mehrmalige Verwendung von Komponenten in unterschiedlichen Systemen erhöht zudem die Chance, etwaige vorhandene Fehler zu entdecken.

2.2.3 Taxonomie für Softwarewiederverwendung

Die verschiedenen Gesichtspunkte der Wiederverwendung lassen sich gemäß Prieto-Díaz [PD93] in die in Tabelle 2.1 dargestellte Taxonomie einordnen.

Wesen	Bereich	Verfahren	Technik	Kapselung	Produkt
Ideen, Konzepte	vertikal	geplant, systematisch	zusammen-setzend	black box, unverändert	Quellcode
Artefakte, Komponenten	horizontal	ad hoc, unsystematisch spontan	generativ	white-box, verändert	Entwurf
Prozesse, Fähigkeiten					Spezifikation
					Objekte
					Text
					Architekturen

Tabelle 2.1: Taxonomie der Wiederverwendung nach Prieto-Díaz [PD93]

Bei dem *Wesen* der Wiederverwendung unterscheidet man, was wiederverwendet wird. Neben der traditionellen Wiederverwendung von Komponenten, z.B. einer Funktionsbibliothek, können auch Ideen und Konzepte, z.B. Design Patterns, wiederverwendet werden. Ebenfalls wiederverwendbar sind Prozesse und Fähigkeiten, die während der Entwicklung von Softwaresystemen entstehen.

Abhängig davon, in welchem *Bereich* Artefakte wiederverwendet werden, unterschiedet man zwischen vertikaler und horizontaler Wiederverwendung [Eil97]. Während bei der vertikalen Wiederverwendung Artefakte in derselben Domäne bzw. in demselben

Anwendungsbereich wiederverwendet werden, ist die horizontale Wiederverwendung anwendungsunabhängig.

Das *Verfahren* beschreibt, wie die Wiederverwendung in den Softwareentwicklungsprozess eingebunden ist. Falls eine bestimmte Wiederverwendungsstrategie inhärenter Bestandteil der Softwareentwicklung ist, so findet eine systematische Wiederverwendung statt. Bei einer eher zufälligen und meist individuellen Verwendung von existierenden Artefakten spricht man von einer unsystematischen bzw. spontanen Wiederverwendung. Das Kopieren und Einfügen von existierendem Code bzw. Codeblöcken ist beispielsweise eine typische und sehr häufige Form der spontanen Softwarewiederverwendung.

Bei der *Technik* unterscheidet man zwischen der Wiederverwendung durch Komposition und der generativen Wiederverwendung. Bei der Wiederverwendung durch Komposition werden existierende Komponenten als Bausteine für neue Systeme benutzt. Durch die Kombination von einfachen Komponenten entstehen neue Komponenten. Die generative Wiederverwendung beschreibt einen Prozess, bei dem Artefakte automatisch, meist auf Basis von Spezifikationen, generiert werden. Dieser Ansatz ist sehr mächtig [Eil97] und verspricht den größten potentiellen Nutzen [PD93]. In Abschnitt 2.2.5.2 wird auf die generative Wiederverwendung eingegangen.

Bei der *black-box*-Wiederverwendung werden Komponenten bzw. Artefakte unverändert wiederverwendet. Die Kommunikation bzw. Interaktion mit einer Komponente wird über deren Schnittstelle definiert. Durch diese Art der Kapselung wird die Qualität und die Verlässlichkeit des resultierenden Systems und der benutzten Komponenten gesteigert. Bei der *white-box*-Wiederverwendung ist eine Anpassung bzw. Modifikation möglich, so dass z.B. interne Attribute einer Komponenten geändert werden können. Ein typisches Beispiel für diese Art der Wiederverwendung ist das Konzept der Vererbung in objektorientierten Programmiersprachen, bei der Attribute bzw. Methoden überschrieben werden.

Die Wiederverwendung ist nicht allein auf die Wiederverwendung von Code beschränkt, sondern umfasst vielmehr alle *Produkte* bzw. Artefakte, die bei der Entwicklung von Software erzeugt werden. Zu diesen Artefakten zählen u.a. Dokumente, Spezifikationen, Entwürfe und jegliche Informationen, die ein Entwickler benötigt [Fre83].

Eine ausführliche Diskussion der Taxonomie wird u.a. in [Sam97] geführt. Beispiele für die unterschiedlichen Facetten der Wiederverwendung finden sich in [PD93]. [Eil97] enthält weitere Beispiele für die horizontale und die vertikale Wiederverwendung.

Die Wiederverwendungsaspekte des in der vorliegenden Arbeit realisierten Systems werden zu einem späteren Zeitpunkt (s. Abschnitt 6.4) in die vorgestellte Taxonomie eingeordnet. In den Abschnitten 2.2.5 und 2.2.6 werden die Techniken bzw. die Produkte der Softwarewiederverwendung genauer erörtert.

2.2.4 Dimensionen der Wiederverwendung

Krueger [Kru92] identifiziert die vier Dimensionen Abstraktion, Selektion, Spezialisierung und Integration, die bei jeder Wiederverwendungsstrategie zum Tragen kommen und im Folgenden zusammengefasst dargestellt werden.

Die *Abstraktion* spielt bei der Softwarewiederverwendung eine zentrale Rolle und ist ein essentielles Merkmal jeder Wiederverwendungstechnik [Kru92]. Eine Abstraktion eines Artefakts ist eine Beschreibung, welche die unwichtigen Details verbirgt und die wichtigen Details hervorhebt. Jede Abstraktionsebene besteht aus Spezifikation und Realisierung bzw. Implementierung, wobei eine Spezifikation einer unteren Ebene eine Implementierung der nächst höheren Ebene darstellt. In Bild 2.1 sind zwei Abstraktionen L und M dargestellt. Die Repräsentationen Rep 1, Rep 2 und Rep 3 sind drei Repräsentationen desselben Artefakts, wobei Rep 1 die detaillierteste Repräsentation widerspiegelt. Die Spezifikation von Rep 2 in der Abstraktionsschicht L ist zugleich die Realisierung von Rep 2 in der Abstraktionschicht M.

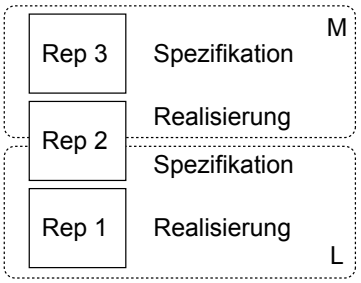


Abbildung 2.1: 2-Schichten Abstraktion nach Krueger [Kru92]

Eine Klasse in einer objektorientierten Sprache ist z.B. eine Abstraktion bzw. Repräsentation eines realen Artefakts. Sie beschreibt gleichartige Objekte, abstrahiert aber von der konkreten Ausbildung. Als intuitives Maß zur Bewertung von Abstraktionen dient der Begriff der *kognitiven Distanz*. Sie beschreibt den Abstand zwischen den Abstraktionen, die ein Softwareentwickler benutzt, und den Abstraktionen, die ihm durch eine bestimmte Technik angeboten werden [Eil97]. Sie steht damit in direkter Beziehung zum intellektuellen Aufwand, der erbracht werden muss, um eine Abstraktion in eine nächste zu überführen bzw. zu transformieren. Die kognitive Distanz kann durch die Verwendung von High-Level-Abstraktionen und die Automatisierung von Transformationen reduziert werden. Laut Eckel [Eck00] bieten alle Programmiersprachen Abstraktionen, so dass argumentiert werden könne, so Eckel weiter, dass die Komplexität des zu lösenden Problems in direkter Beziehung zur Qualität der Abstraktion steht. Im Bereich der Programmiersprachen beschreibt die kognitive Distanz die intellektuelle Leistung zur Abbildung eines Problems auf die Lösung desselben mit einem bestimmten System bzw. einer bestimmten Sprache, oder anders formuliert, zur Modellierung eines Problems auf Basis einer bestimmten Technologie. Dabei erledigt der Compiler die automatische Transformation der Spezifikation in die Realisierung. Objektorientierte Programmiersprachen bieten, beispielsweise im Vergleich zu Assemblersprachen, einen sehr hohen Grad an Abstraktion und besitzen eine geringe kognitive Distanz, da der Problembereich (das reale Problem) dem Lösungsbereich (System bzw. Sprache mit dem das Problem gelöst wird) sehr nahe liegt.

Die meisten Wiederverwendungstechniken bieten einen Mechanismus zur *Selektion* von wiederbenutzbaren Artefakten. Die Selektion umfasst die Lokalisierung, den Vergleich

und die Auswahl von Artefakten aus einer Sammlung von Artefakten. Bei einer unsystematischen Wiederverwendungstechnik beruht die Selektion auf dem Wissen und auf der Erfahrung eines Entwicklers.

Bei vielen Wiederverwendungstechniken werden allgemeine Eigenschaften von Artefakten identifiziert und in einem einzelnen allgemeinen bzw. generischen Artefakt zusammengefasst. Bei der *Spezialisierung* wird dieses allgemeine Artefakt, z.B. durch Parametrisierung, angepasst und verfeinert.

Die *Integration* beschreibt die Einbettung von wiederbenutzbaren Artefakten in ein bestimmtes System mit einem sogenannten Integrationsframework. Ein Entwickler benutzt diese Art von Framework zur Kombination von Artefakten. Damit Artefakte kombiniert werden können, ist es notwendig, dass sie über eine entsprechende Schnittstelle zur Kommunikation und Interaktion verfügen.

2.2.5 Techniken der Wiederverwendung

Prinzipiell werden zusammensetzende und generative Techniken zur Wiederverwendung unterschieden. Diese Techniken werden in [Lin99] auch als komponentenbasierte bzw. prozessbasierte Ansätze bezeichnet, die im Folgenden näher betrachtet werden. Eine ausführliche Diskussion der Ansätze findet sich in [Sam97].

2.2.5.1 Wiederverwendung durch Komposition

Bei der Wiederverwendung durch Komposition werden vorhandene Softwarekomponenten zu neuen Komponenten zusammengesetzt. Ein typischer Vertreter dieses Ansatzes ist die objektorientierte Programmierung, bei der vorhandene Klassen bzw. Objekte durch Aggregation, Komposition und Vererbung zu neuen Klassen oder Systemen zusammengesetzt werden.

Sametinger macht den Erfolg dieser Vorgehensweise von der Verfügbarkeit von qualitativ hochwertigen Komponenten, korrekten Klassifizierungs- und Suchmechanismen, ausreichender Dokumentation der Komponenten und der Möglichkeit zur flexiblen Kombination und Anpassung von Komponenten an spezielle Bedürfnisse abhängig [Sam97]. Bei Komponenten kann es sich beispielsweise um Funktionen, Module oder auch Klassen handeln, die meist in Form von Bibliotheken, z.B. Funktions- oder Klassenbibliotheken, vorliegen.

2.2.5.2 Generative Wiederverwendung

Die generative Wiederverwendung basiert eher auf der Wiederverwendung eines Generierungsprozesses als auf der Wiederverwendung von Komponenten [Sam97]. Dabei werden wiederverwendbare Artefakte, häufig in der Form von Spezifikationen, zur automatischen Generierung von neuen Artefakten benutzt. Obwohl der Fokus der generativen Programmierung in der Literatur auf der Generierung von kompletten Systemen liegt, gibt es doch einige Parallelen mit der in dieser Arbeit angewandten Vorgehensweise.

Ein *Generator* ist ein Programm, das als Eingabe eine Spezifikation erhält und als Ausgabe ein Produkt liefert, das dieser Spezifikation genügt [Ste98]. Im Gegensatz zu *Anwendungsgeneratoren*, die aus Spezifikationen komplette Anwendungen erzeugen (siehe dazu z.B. [Cle88]), wird in dieser Arbeit die Generierung von Softwarekomponenten und Dokumentation realisiert. Die Funktionsweise eines Generators ist schematisch in Abb. 2.2 dargestellt.

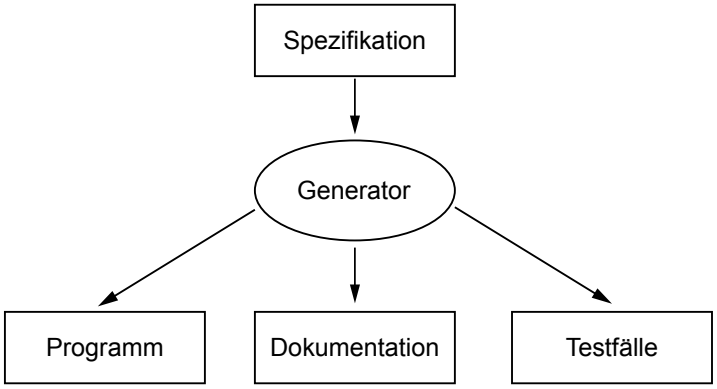


Abbildung 2.2: Funktionsweise eines Generators gemäß Cleaveland [Cle88]

Aus einer Spezifikation werden mittels eines Generators unterschiedliche Artefakte, wie z.B. Dokumentation, Quelltext und Testfälle, erzeugt. Nach Krueger [Kru92] ist der Einsatz von (Anwendungs-) Generatoren dann sinnvoll, falls mindestens einer der folgenden Faktoren zutrifft:

- es werden viele gleichartige Systeme entwickelt,
- ein System wird häufig modifiziert oder
- es werden während der Entwicklung viele Prototypen benötigt.

Betrachtet man die automatische Generierung von Komponenten, so kann diese Liste noch um einen Faktor erweitert werden. Wie in Kapitel 5 ersichtlich wird, sind evtl. sehr viele Komponenten in einen bestimmten (Generierungs-) Prozess involviert. Eine manuelle Verarbeitung ist in diesem Fall unter ökonomischen Gesichtspunkten nicht vertretbar.

2.2.6 Produkte der Wiederverwendung

Gemäß Abschnitt 2.2.3 können unterschiedliche Produkte wiederverwendet werden. Die Produkte umfassen u.a. Quellcode, Algorithmen, Datenstrukturen, Bibliotheken, Architekturen, Frameworks, Patterns, Anwendungen und Dokumentationen. Einige dieser Produkte werden im Hinblick auf das realisierte System erläutert.

Das Zusammenführen von *Quellcode* kann als eine primitive Form der Komposition aufgefasst werden. Die Integration von existierendem Quellcode in einen neuen Kontext

ist zwar unsystematisch, kann aber dennoch äußerst effektiv sein [Kru92]. Eine unmittelbare Wiederverwendung ist oft nicht möglich, sondern erfordert eine Adaption des Codes [Lin99]. Das Kopieren von Code ist eine sehr häufige Form der Softwarewiederverwendung.

Die klassische Methode zur Wiederverwendung von domänenspezifischem Wissen sind *Bibliotheken* [vDKV00]. McIlRoy propagierte bereits 1968 in [McI68] eine Bibliothek von wiederbenutzbaren Komponenten, die nach bestimmten Kriterien angepasst werden können, wobei McIlroy unter einer Komponente eine Prozedur einer prozeduralen Sprache verstand. Generell umfassen Softwarebibliotheken programmiersprachliche Konstrukte, wie beispielsweise Module, Klassen, Funktionen und Methoden. Diese Konstrukte stellen häufig benötigte Funktionalität zur Verfügung und erlauben eine einfache Wiederverwendung dieser Funktionalität. Softwarebibliotheken liegen in Form von Quelltext oder in Form von übersetztem Programmcode vor. Bibliotheken werden unter anderem in C, C++, Perl und Java unterstützt. Bei objektorientierten Programmiersprachen werden Bibliotheken als Klassenbibliotheken bezeichnet. Eine Klassenbibliothek ist eine „organisierte Softwarewaresammlung, aus der ein Entwickler nach Bedarf Einheiten verwendet“ [Eis95] und dient nach Balzert [Bal98] der „Wiederverwendung im Kleinen“.

Ein *Framework* ist nach Johnson und Foote [JF88] eine Menge von Klassen, die einen abstrakten Entwurf zur Lösung einer ganzen Familie von ähnlichen Problemen beinhalten. Frameworks umfassen abstrakte und konkrete Klassen, die erweitert und verfeinert werden [Tai93]. Sie weisen zwar einen hohen Grad an Wiederverwendung auf. Allerdings sind Komponenten eines Frameworks meist sehr eng an das Framework selbst gekoppelt, da sie auf die Funktionalität von anderen Komponenten des Frameworks angewiesen sind. Zusätzlich gilt, dass die Entwicklung von Frameworks ein gewisses Maß an Wissen und Erfahrung voraussetzt.

Ein *Entwurfsmuster* bzw. *Design Pattern* „beschreibt ein bestimmtes, in einem bestimmten Kontext immer wiederkehrendes Problem sowie ein generisches Schema zur Lösung dieses Problems“ und „basiert auf langjähriger praktischer Erfahrung im Entwurf von Software-Systemen“ [Bus95]. Entwurfsmuster erfassen die Absicht hinter dem Design eines Softwaresystems und geben „eine bewährte generische Lösung für ein immer wiederkehrendes Entwurfsproblem an, das in bestimmten Situationen auftritt“ [Bal98]. Die Wiederverwendung bezieht sich bei Design Patterns ausschließlich auf den Entwurf. Inzwischen unterstützen bereits einige Werkzeuge für den Softwareentwurf, wie beispielsweise Together [Tog04], die automatische Generierung von Klassen auf Basis von Patterns. Beispiele für Design Patterns finden sich in der Literatur z.B. bei Coad [Coa92], Gamma et al. [GHJV95] und Pree [Pre95]. In dieser Arbeit wird bei der Integration und Komposition von Komponenten ebenfalls, wie in Abschnitt 5.2.1.1 beschrieben, auf bewährte Design Patterns zurückgegriffen.

Programmiersprachen verfügen ebenfalls über bestimmte Mechanismen zur Wiederverwendung von Software, wie z.B. die Verwendung der bereits erwähnten Bibliotheken. In objektorientierten Programmiersprachen ist die Wiederverwendung von *Objekten* inhärenter Bestandteil der Sprache. Typische Beispiele hierfür sind die Konzepte der Vererbung oder der Aggregation. Die objektorientierte Programmierung besitzt ein enormes

Potential bei der Entwicklung von Software und wurde lange Zeit als die Zukunftstechnologie für die Wiederverwendung von Software gesehen [PD93]. Im Allgemeinen gilt jedoch, dass Objekte nur in derselben Programmiersprache wiederverwendet und nicht einfach in andere Sprachen integriert werden können.

Ein weiteres, sehr wichtiges Beispiel der Wiederverwendung ist die Implementierung von *Standards* [Ber96]. Gemäß den Qualitätskriterien aus Abschnitt 2.1.3 bildet ein offener Standard die Basis für eine breite Akzeptanz einer Technologie. Ein Problem bzgl. Standards bringt Tanenbaum [Tan96] zur Sprache, welches an dieser Stelle nicht kommentiert wird:

> „The nice thing about standards is that there are so many to choose from“.

2.3 Komponenten- und Middlewaresysteme

Dieser Abschnitt umfasst neben der Definition, den Anforderungen an und Eigenschaften von Komponenten- und Middlewaresystemen umfangreiche Beispiele für die Anbindung von Systemen an unterschiedliche Middleware-Infrastrukturen. Anhand der Beispiele soll gezeigt werden, wo die Schwierigkeiten bzw. Probleme bzgl. der vorgestellten Middlewaretechnologien liegen. Der Fokus liegt dabei auf der Middleware, welche die Komponenten miteinander verbindet, und nicht auf den Komponenten selbst.

2.3.1 Komponentenbasierte Softwareentwicklung (CBSE)

Die komponentenbasierte Softwareentwicklung (CBSE, Component Based Software Engineering bzw. CBSD, Component Based Software Development) beschäftigt sich mit der Entwicklung von Systemen auf Basis von wiederbenutzbaren Artefakten – den Komponenten – und der Entwicklung von Komponenten [ICS02]. Die Motivation für CBSE liegt laut Kozaczynski und Booch u.a. in der wachsenden Notwendigkeit zur Interoperabilität von unabhängig voneinander entwickelter Software [KB98]. Somit ist das Ziel von CBSE die Verwendung und Wiederverwendung von Code, unabhängig davon, in welcher Programmiersprache und auf welcher Plattform dieser Code entwickelt wurde [BB00]. Eine Eigenschaft von vielen komponentenbasierten Systemen ist die Verteilung und Nutzung von Komponenten mittels Middleware, die ein inhärenter Bestandteil von verteilten Komponentensystemen ist.

2.3.1.1 Definitionen

In der Literatur existieren zahlreiche unterschiedliche Definitionen einer Komponente. Während eine Komponente in Abschnitt 1.2 als eine Softwareeinheit mit einer definierten Schnittstelle beschrieben wurde, soll an dieser Stelle eine konkretere Definition gegeben werden. Sametinger definiert eine Komponente als „ein in sich geschlossenes, eindeutig identifizierbares Artefakt, das eine bestimmte Funktionalität, eine definierte Schnittstelle und entsprechende Dokumentation besitzt“ [Sam97].

Eine Komponente sollte eine in sich geschlossene Einheit sein, damit sie bei der Wiederverwendung von keiner anderen Komponente abhängig ist. Sollten dennoch Abhängigkeiten vorhanden sein, so sollten diese klar ersichtlich und dokumentiert sein. Damit eine Komponente eindeutig identifizierbar ist, darf sie nicht über mehrere Standorte verteilt oder sogar mit anderen Artefakten vermischt sein. Eine Komponente zeichnet sich u.a. dadurch aus, dass sie eine definierte Schnittstelle zur Kommunikation besitzt. Diese Schnittstelle kann von anderen Programmen bzw. Artefakten aufgerufen und benutzt werden. Die Schnittstelle nimmt – analog zur objektorientierten Programmierung – bei der komponentenbasierten Softwareentwicklung einen hohen Stellenwert ein, da sie die eigentliche Möglichkeit zur Interaktion mit der Komponente zur Verfügung stellt. Die Art und Weise wie mit der Schnittstelle kommuniziert wird spielt dabei nur eine untergeordnete Rolle. Die Schnittstelle kann prozedural, objektorientiert, nachrichten- oder ereignisbasiert sein, um nur einige Möglichkeiten zu nennen. Die Schnittstelle und die Komponente selbst müssen ausreichend dokumentiert sein, so dass eine adäquate Wiederverwendung einfach möglich ist.

2.3.1.2 Anforderungen

Kiely geht in [Kie98] davon aus, dass Komponentensysteme erfolgreich sind, wenn sie eine standardisierte Infrastruktur unterstützen. Zu dieser Infrastruktur zählen einheitliche Beschreibungen, standardisierte Schnittstellen und Repositories. Einheitliche Beschreibungen und standardisierte Schnittstellen garantieren einen hohen Grad an Kompatibilität und Integrierbarkeit. Repositories dienen zur Speicherung von Beschreibungen und Komponenten und bieten so eine Möglichkeit zur strukturellen Suche nach denselben.

2.3.1.3 Eigenschaften

Inwiefern unterscheiden sich nun die Komponententechnologie und die Objekttechnologie (OT) voneinander? Die OT besitzt im Vergleich zu anderen Technologien einen hohen Grad an Abstraktion. Zudem können Objekte gekapselt werden, so dass der Zugriff nur über die Schnittstelle eines Objektes möglich ist. Darüber hinaus besitzt die OT weitere Merkmale, die durch die komponentenbasierte Softwareentwicklung gefordert werden. Brown und Wallnau führen in [BW98] u.a. zwei Gründe an, warum die Objekt- und die Komponententechnologie nicht übereinstimmen. Zum einen bietet OT nicht den notwendigen Grad an Abstraktion und zum anderen lassen sich Komponenten auch ohne OT realisieren, so dass die Autoren zu dem Schluss kommen, dass „die Objekttechnologie für die komponentenbasierte Softwareentwicklung weder notwendig noch ausreichend“ ist.

Nach Thomason [Tho00] kann eine Komponente einen unterschiedlichen Grad an Verteiltheit, Modularität, Plattform- und Sprachunabhängigkeit besitzen und somit in einen 3-dimensionalen Raum, den *Komponentenraum*, abgebildet werden.

In dem Komponentenraum aus Abb. 2.3 befinden sich monolithische Systeme an der Koordinate [0,0,0]. Das bedeutet, dass monolithische Systeme weder verteilt, noch modular, noch plattform- und sprachabhängig sind. Thomason stellt fest, dass weder COR-

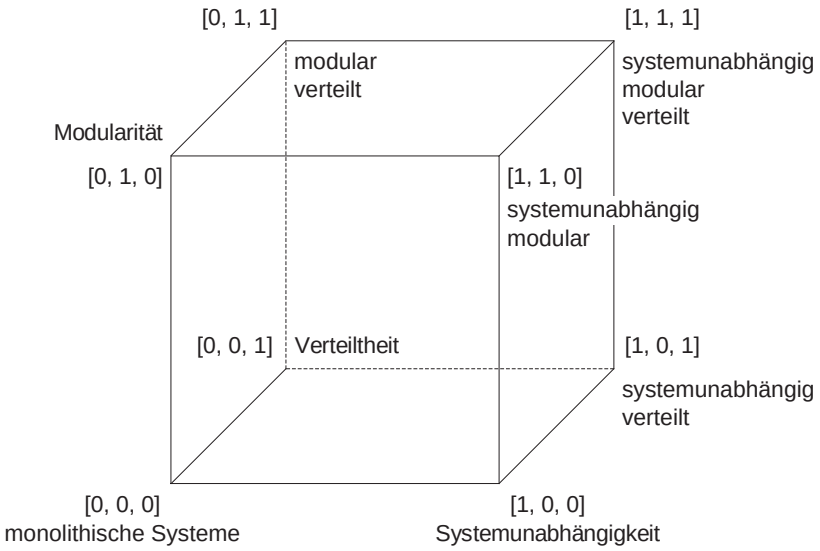


Abbildung 2.3: Komponentenraum gemäß Thomason

BA noch Java eine optimale Lösung darstellen. CORBA ist verteilt und sprachunabhängig, aber die zugrunde liegenden Komponenten sind oft plattformabhängig. Java-Komponenten sind zwar für die Verwendung auf unterschiedlichen Plattformen (Cross-Plattform) gedacht, aber im allgemeinen sprachspezifisch. „Jedoch würde durch eine Kapselung einer plattformunabhängigen Sprache, wie Java, durch eine sprachunabhängige Middleware, wie CORBA, eine Komponente entstehen, die den [1,1,1]-Status verdient“ [Tho00]. Diese Kernaussage bildet die Basis für das in dieser Arbeit realisierte System. Erwähnenswert ist dabei, dass dieses System nicht auf Java und CORBA beschränkt ist, wie in Kapitel 5 detailliert erörtert wird.

2.3.1.4 Komponentensysteme

Die drei führenden Kräfte in Bezug auf Komponentensysteme sind Microsoft mit dem Distributed Component Object Model (COM/DCOM) [Ses97], Sun Microsystems mit den JavaBeans [Ham97] bzw. den Enterprise JavaBeans [Sun03a] und der Java Enterprise Edition (J2EE) [Sha03] und die Object Management Group (OMG) mit der Object Management Architecture (OMA) [Obj01b]. Jede dieser Technologien bietet gewisse Techniken zur Implementierung und zur Integration von Komponenten.

In jüngster Zeit werden Komponentensysteme bzw. deren zugrunde liegenden Konzepte auch in der Automationsindustrie eingesetzt. Dabei geht es nicht in erster Linie um den Einsatz von Komponentensystemen, sondern primär um die Verwendung moderner Sprachen, Systeme und Konzepte. Das Ziel ist eine Verschmelzung der klassischen Automation mit modernen Konzepten der Informatik. Damit eröffnet sich der Automationsindustrie ein breites Spektrum an neuen Anwendungen, wie der Wartung, Fehleranalyse und Steuerung von Systemen via Internet oder allgemeiner die Einbettung von Systemen in eine moderne Service- und Management-Infrastruktur [NS01, NG00, NBGK01].

Beim Einsatz von Komponentensystemen in der Automation geht es, im Gegensatz zur Verwendung in großen und komplexen Systemen, weniger um die Beherrschung der Komplexität, als vielmehr um die Abstraktion von Low-level-Komponenten (Geräte) und -Kommunikation (z.B. CAN). Typische Vertreter dieser Kategorie von Software sind OPC [OPC03a], JFPC [Sie04] und Jini [Wal99]. OPC (OLE for Process Control) bildet das Konzept von OLE (Object Linking and Embedding) auf das Gebiet der Automation ab und bietet über eine standardisierte Schnittstelle einen einfachen Zugriff auf beliebige Geräte. Die Java-Technologie JFPC (Java For Process Control) realisiert den echtzeitfähigen Zugriff in Java auf interne und externe Komponenten eines Systems namens SICOMP [Sie03]. Bei JFPC werden sämtliche Geräte und deren Funktionalität über einen einheitlichen Namensraum adressiert. JFPC verfolgt somit eine ähnliches Konzept wie OPC. Jini bietet gemäß Venners „die objektorientierte Schnittstelle der Zukunft“ [Ven99]. In Abschnitt 2.3.3.3 wird detailliert auf Jini eingegangen.

2.3.2 Middleware

Middleware kann wesentlich zur Integration von Systemen beitragen. Sie bietet die notwendigen Abstraktionen und Techniken zur Verbindung von Systemen. Das in dieser Arbeit realisierte System MAX generiert auf Basis einer Schnittstellenbeschreibung u.a. unterschiedliche Middlewarekomponenten zur Integration von teilweise sehr unterschiedlichen Systemen. Deshalb wird an dieser Stelle eine ausführliche Einführung in Middlewaresysteme gegeben. Abschnitt 2.3.3 enthält konkrete Beispiele für die Anbindung von Komponenten an unterschiedliche Middleware und zeigt Probleme und Schwierigkeiten hinsichtlich der Anbindung auf.

2.3.2.1 Definitionen

Zur Realisierung von verteilten Systemen basierend auf heterogenen Plattformen bedarf es verteilter Dienste mit standardisierten Programmierschnittstellen (APIs) und standardisierten Kommunikationsprotokollen. Diese Dienste werden als Middleware bezeichnet, da sie sich zwischen den Plattformen und den Applikationen befinden. Middleware löst das Problem der Heterogenität und ermöglicht die Kommunikation und Koordination von verteilten Komponenten [Emm00]. Allgemein ist Middleware eine Software, die unabhängige Anwendungen miteinander verbindet [INT04] und befindet sich nach Yellin [Yel01] entweder in der Mitte zwischen verteilten Komponenten oder zwischen einer Applikation und den Diensten eines Netzwerkes. Im ersten Fall spricht man von einer horizontalen, im zweiten Fall von einer vertikalen Komposition einer Anwendung mit den Diensten eines Netzwerkes.

Eine Middleware abstrahiert allgemeine Eigenschaften einer Anwendungsdomäne und präsentiert eine einheitliche Schnittstelle, die es auf einfache Art und Weise erlaubt, Anwendungen in eine Domäne zu integrieren. Eine Kommunikationsmiddleware abstrahiert allgemeine Eigenschaften von verteilten Anwendungen und präsentiert eine einheitliche Schnittstelle zur verteilten Kommunikation. Dabei ist es wünschenswert, dass die Schnittstelle standardisiert ist und standardisierte Protokolle unterstützt. Durch eine

Abstraktion auf Basis von High-level-Schnittstellen wird die Entwicklung von Programmen wesentlich erleichtert, da von der Komplexität bzgl. Netzwerken und Protokollen abstrahiert wird.

Wegen der großen Bedeutung von standardisierten Schnittstellen für die Portabilität und von standardisierten Protokollen für die Interoperabilität, war und ist Middleware Gegenstand zahlreicher Standardisierungsbestrebungen, z.B. durch die International Standardization Organisation (ISO), die ANSI, die Open Software Foundation (OSF) oder die Object Management Group (OMG). Beispielsweise hätte das Internet oder das World Wide Web ohne die Definition der standardisierten Internet-Protokolle sicherlich eine geringere Verbreitung gefunden, als dies heute der Fall ist.

Middleware wird meistens in Zusammenhang mit verteilter Kommunikation gesehen. In dieser Arbeit erhält der Begriff Middleware eine allgemeinere Bedeutung und umfasst sowohl Software zur verteilten als auch zur lokalen Kommunikation. In diesem Sinne werden Softwarebrücken (Bridges, z.B. eine Java/COM-Bridge [Wat98, Int01b]) zur Anbindung von unterschiedlichen Technologien ebenfalls als Middleware bezeichnet.

2.3.2.2 Eigenschaften

Emmerich stellt in [Emm00] fest, dass verschiedene kommerzielle Trends zu einem steigenden Bedarf an verteilten Systemen führen und gibt dafür verschiedene Gründe an. Erstens nimmt die Anzahl der Fusionen von Unternehmen stetig zu. Da die unterschiedlichen Abteilungen ihren Kunden einheitliche Dienste anbieten wollen, müssen die unterschiedlichen Systeme der Abteilungen dazu integriert werden. Aus zeitlichen Gründen stellt die Entwicklung eines neuen Systems oft keine Alternative dar, so dass stattdessen vorhandene Systemkomponenten in ein verteiltes System integriert werden müssen.

Zweitens nimmt die verfügbare Zeit, neue Dienste anzubieten, stetig ab. Die Integration von neuen Diensten kann nur realisiert werden, wenn unabhängige Komponenten benutzt und integriert werden, anstatt neue Komponenten zu entwickeln. Zuletzt bietet das Internet zusätzliche Möglichkeiten, neue Produkte und Dienste einer großen Anzahl an potentiellen Käufern anzubieten.

Bernstein [Ber96] geht davon aus, dass Middleware wesentlich zur Erfüllung der in Abschnitt 2.1.3 genannten Qualitätskriterien beitragen kann, gibt allerdings zu bedenken, dass Middleware zwar viele Probleme lösen kann, jedoch kein „Allheilmittel“ darstellt [Ber93]. Middleware wird für ein immer breiter werdendes Spektrum an Applikationen eingesetzt. Yellin sieht hierin einen Paradigmen-Wechsel von der „Mikro-Programmierung“ hin zur „Makro-Programmierung“ [Yel01]. Während die Mikro-Programmierung die Implementierung von Applikationen in einem homogenen Umfeld beschreibt, bezeichnet die Makro-Programmierung die Konstruktion von Applikationen mittels Integration existierender verteilter Logik in einem oftmals heterogenen Umfeld. Neue Applikationen wiederum haben typischerweise Anforderungen, die von existierenden Middleware-Technologien nicht unterstützt werden und gehorchen in der Regel dem „Parkinson'schen Gesetz“, so dass ihre Ausmaße immer die Grenzen der Technologie erreichen [BL79, Tic79].

Hinsichtlich den Eigenschaften von Systemen gemäß Abschnitt 2.1.2 gilt auch für

Middleware, dass sie komplex ist und dem Gesetz der permanenten Veränderung unterliegt. Für Bernstein ist die Komplexität von Middleware auf längere Dauer unhaltbar und die Mannigfaltigkeit von Middleware bereits zu groß [Ber93, Ber96]. Brown bezeichnet „Middleware – den Eckpfeiler für verteilte Systeme – als die komplexeste und verwirrendste Technologie“ und ist der Ansicht, dass es nahezu unmöglich ist, mit den zahlreichen neuen Technologien Schritt zu halten [Bro99]. Diese Aussagen müssen insofern relativiert werden, als Middleware inzwischen integraler Bestandteil von vielen Sprachen und Systemen ist.

2.3.2.3 Middlewaresysteme

Gemäß Abschnitt 2.3.2.1 war und ist Middleware Gegenstand zahlreicher Standardisierungsbestrebungen. Es wurden bereits zahlreiche Middleware-Technologien entwickelt, die erfolgreich im kommerziellen Umfeld eingesetzt werden. An dieser Stelle soll ein grober Überblick über Systeme bzw. Technologien zur verteilten Kommunikation gegeben werden.

Zur *streambasierten Kommunikationen* zwischen zwei Systemen werden üblicherweise sog. Sockets [Ste90] unterstützt. Analog zur Vorgehensweise zum Schreiben in bzw. Lesen von einer Datei im Dateisystem werden nach dem Verbindungsaufbau eines Clients mit einem Server Daten über einen bidirektionalen Kommunikationskanal transportiert. Der Transport der Daten ist für einen Entwickler zwar transparent, das Empfangen bzw. Senden und die Verarbeitung der Daten müssen jedoch implementiert werden. Demzufolge bieten Sockets einen sehr geringen Grad an Abstraktion. Auf manchen Systemen, z.B. eingebetteten Systemen, steht oftmals keine Alternative zur Socket-Kommunikation zur Verfügung.

Prozedurorientierte Middleware bietet die Möglichkeit, entfernte Prozeduren mittels Remote Procedure Calls (RPC) [BN84] aufzurufen. Zur Realisierung einer RPC-basierten Anwendung wird auf Basis einer plattformunabhängigen Schnittstellenbeschreibung wie XDR (eXternal Data Representation) der notwendige Code zur Kommunikation, die sog. Stubs und Skeletons, automatisch generiert. Für einen Entwickler ist der Aufruf einer entfernten Prozedur nahezu transparent, da die Details der Kommunikation verborgen sind. In jüngster Zeit wird der entfernte Prozeduraufruf auf Basis von XML und HTTP propagiert. SOAP [Wor00] bzw. Web Services [Wor02] entwickeln sich aktuell zum Standard für die verteilte Kommunikation im World Wide Web. Analog zu RPC können Prozeduren auf entfernten Rechnern aufgerufen werden, wobei die Daten im XML-Format via HTTP übertragen werden. Analog zu SOAP bietet XML-OPC [OPC03b] eine Möglichkeit zum entfernten Zugriff auf Geräte.

Das objektorientierte Pendant zu entfernten Prozeduraufrufen sind entfernte Methodenaufrufe. *Objektorientierte Middleware* bietet die Möglichkeit zur verteilten Kommunikation von Objekten. Die bekanntesten Vertreter dieser Kategorie von Middleware sind Implementierungen der Common Request Broker Architecture (CORBA) [Obj00], der Java Remote Method Invocation (RMI) [Sun04] und von .NET Remoting [Mic01a]. Neben der reinen Kommunikation bieten objektorientierte Middlewarearchitekturen noch eine Reihe von Diensten an, z.B. Transaktions- oder Sicherheitsmechanismen.

Die genannten Technologien bieten einen unterschiedlichen Grad an Abstraktion und Unterstützung bei der Kommunikation. Mit Ausnahme des Internet auf Basis der Internet-Protokollfamilie hat sich bis zum heutigen Zeitpunkt kein weltweiter Standard zur verteilten Kommunikation etablieren können. Der Erfolg von Web Services scheint vielversprechend zu sein, ist aber noch nicht eindeutig zu bewerten. Bei der Integration von Legacy-Systemen ist CORBA nach wie vor führend. CORBA findet eine weite Unterstützung und steht für viele Plattformen und Sprachen zur Verfügung.

2.3.3 Middleware-Anwendungsbeispiele

An den folgenden Anwendungsbeispielen sollen die Vorgehensweise zur Realisierung von verteilten Systemen mit unterschiedlichen Techniken bzw. Technologien und die Schwierigkeiten bzw. Probleme dieser Technologien aufgezeigt werden. Am Beispiel der Klasse *Calc* aus Listing 2.1 wird demonstriert, wie diese Klasse in die Middlewareinfrastrukturen CORBA, RMI, Jini, .NET Remoting und Web Services integriert werden kann.

```
package calc;

public class Calc {
  public double add(double arg1, double arg2) {return arg1+arg2;}
  public double sub(double arg1, double arg2) {return arg1-arg2;}
  public double mul(double arg1, double arg2) {return arg1*arg2;}
  public double div(double arg1, double arg2) {return arg1/arg2;}
}
```

Listing 2.1: Die Klasse Calc in Java

Die Klasse *Calc* besitzt vier Methoden mit den vier Grundrechenarten zur Addition, Subtraktion, Multiplikation und Division von zwei Argumenten vom einfachen Datentyp *double*.

2.3.3.1 CORBA

CORBA ist eine Spezifikation zur verteilten Kommunikation von Objekten. Implementierungen von CORBA stehen für unterschiedliche Sprachen und Systeme zur Verfügung. Konzeptionell basiert die Kommunikation von Objekten auf dem entfernten Methodenaufruf.

Zur Integration eines Objektes in CORBA wird zunächst die Schnittstelle des Objektes auf Basis der *Interface Definition Language* (*IDL*), wie in Listing 2.2 dargestellt, definiert. Die IDL-Definition beschreibt die Schnittstelle *CalcServerI* mit den vier erwähnten Methoden. Zusätzlich enthält die IDL-Beschreibung einer Methodensignatur sog. *Richtungsattribute* (*in*, *out*, *inout*) für die einzelnen Parameter.

```
module corba
{
  module calc
  {
    interface CalcServerI
    {
      double add(in double arg1, in double arg2);
      ...
    };
```

```
10    };
  };
```

Listing 2.2: Definition der CORBA-Schnittstelle in IDL

Auf Basis dieser IDL-Definition werden in der Java-Implementierung von CORBA die notwendigen Klassen für CORBA automatisch mit einem IDL-Compiler[1] erzeugt. In nachfolgendem Beispiel wird die Erzeugung von Klassen auf Basis der sog. ImplBase[2] gezeigt. Der IDL-Compiler erzeugt unter Angabe bestimmter Parameter die notwendigen Stubs und Skeletons für die verteilte Kommunikation in CORBA und führt direkt eine Abbildung der Datentypen durch. Eine Implementierung der Schnittstelle *CalcServerI* wird von der erzeugten Klasse *_CalcServerImplBase* abgeleitet. Die Klasse *CalcServer* in Listing 2.3 enthält neben der Implementierung der Schnittstelle die notwendigen Anweisungen zur Registrierung des Objekts beim Namensdienst von CORBA. Die Registrierung kann auch außerhalb der Klasse realisiert werden, so dass die Import-Anweisungen (Zeilen 3-5) nicht benötigt werden.

```
  package corba.calc;

  import org.omg.CosNaming.*;
  import org.omg.CosNaming.NamingContextPackage.*;
5 import org.omg.CORBA.*;

  public class CalcServer
  extends _CalcServerIImplBase
  {
10  // Implementierung der Methoden
    public double add(double arg1, double arg2) {return arg1+arg2;}
    ...

    public static void main(String args[])
15  {
      try {
        ORB orb = ORB.init(args, null);
        CalcServer calcRef = new CalcServer();
        orb.connect(calcRef);
20
        org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

25      NameComponent nc = new NameComponent("CalcServer", "");
        NameComponent path[] = {nc};
        ncRef.rebind(path, calcRef);
      } catch(Exception e) {
        System.err.println(e.getMessage());
30    }
    }
  }
```

Listing 2.3: Implementierung der CORBA-Schnittstelle in Java

CORBA ist sehr mächtig und für die Integration von heterogenen (Legacy-) Systemen sehr gut geeignet, da CORBA für viele Sprachen und Systeme zur Verfügung steht. Die system- und sprachunabhängige Beschreibung auf Basis von IDL ermöglicht die Abbildung in unterschiedliche Sprachen. Durch die automatische Generierung von Code und

[1] In Java 2 z.B. idlj
[2] idlj -oldImplBase

der damit verbundenen Kapselung der Kommunikation in Stubs und Skeletons ist die eigentliche Kommunikation von Objekten für den Benutzer weitestgehend transparent. Von einem Entwickler wird die Kenntnis von IDL vorausgesetzt.

Aus Beispiel 2.3 wird ersichtlich, dass nur eine geringe Bindung zwischen Klasse und Middleware besteht. Betrachtet man die Realisierungen von CORBA im Umfeld der eingebetteten Systeme, so ist CORBA oft zu mächtig und umfangreich. Für manche Systeme bzw. Sprachen steht keine Implementierung von CORBA zur Verfügung. Implementierungen von CORBA für bestimmte Kommunikationsmedien und -protokolle wie CAN [KJH+00] sind mitunter nur schwer realisierbar und haben teilweise nur wenig mit CORBA gemein [BK02].

2.3.3.2 RMI

Die *Remote Method Invocation* (*RMI*) ist eine Erweiterung des traditionellen entfernten Prozeduraufrufs in Java. Java bietet seit Java 1.1 mit RMI Java-Objekten die Möglichkeit, Methoden von anderen Objekten über Prozess- und Systemgrenzen hinweg aufzurufen. RMI unterstützt dabei die Standardkonzepte von verteilten Architekturen, z.B. einen Namensdienst und entfernte Methodenaufrufe. Darüber hinaus ermöglicht RMI im Vergleich zum klassischen entfernten Prozedur- oder Methodenaufruf, wie bei RPC und CORBA, die Übertragung von Objekten mittels Serialisierung.

Jedes RMI-Objekt, das über RMI zur Verfügung steht, definiert eine Schnittstelle, auf die von entfernten Rechnern zugegriffen werden kann. Im Vergleich zu den bereits genannten Technologien ist keine externe Schnittstellendefinition (z.B. IDL) notwendig, da dies indirekt über das Konzept einer expliziten Schnittstelle in Form eines Java-Interfaces realisiert ist.

Zur Realisierung eines RMI-Objektes müssen ein Interface, das von *Remote*[3], und eine Klasse, die von *RemoteObject*[4] bzw. einer von *RemoteObject* abgeleiteten Klasse abgeleitet ist und das Interface implementiert, realisiert werden. Listing 2.4 enthält die Remote-Schnittstelle *CalcServerI*. Bei der Definition der Schnittstelle wird vorausgesetzt, dass sämtliche RMI-Methoden die Exception *RemoteException* definieren.

```
public interface CalcServerI
extends java.rmi.Remote
{
  double add(double arg1, double arg2) throws java.rmi.RemoteException;
  ...
}
```

Listing 2.4: Definition der Java RMI Remote Schnittstelle

Die Klasse *CalcServer* in Listing 2.5 ist von *UnicastRemoteObject*[5] abgeleitet und implementiert das angegebene Interface. Während die Angabe von *RemoteException* beim Konstruktor zwingend ist, ist sie bei den implementierten Methoden seit Java 2 optional. Die Registrierung des Objektes ist in der *main*-Methode realisiert. Die notwendigen Klassen

[3] java.rmi.Remote
[4] java.rmi.RemoteObject
[5] java.rmi.server.UnicastRemoteObject

(Stubs und Skeletons) zur verteilten Kommunikation werden mit dem RMI-Compiler[6] automatisch erzeugt.

```
package rmi.calc;

public class CalcServer
extends java.rmi.server.UnicastRemoteObject
implements CalcServerI
{
  public CalcServer() throws java.rmi.RemoteException {}

  // Implementierung der Methoden
  ...

  static void main(String[] args) {
    int port = 1001;
    try {
      CalcServer cs = new CalcServer();
      java.rmi.Naming.rebind("//localhost:" + port + "/CalcServer", cs);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 2.5: Implementierung der Java RMI Remote Schnittstelle

Aus dem gegebenen Beispiel wird ersichtlich, dass die Middleware Bestandteil des Codes ist (Zeile 4 u. 7). Die Folge ist eine Abhängigkeit der Implementierung von der Middleware. Soll die gleiche Funktionalität mit einer anderen Middleware angeboten werden, so müssen die Klassen im günstigsten Fall angepasst und im schlechtesten Fall, falls z.B. kein Quelltext mehr vorliegt, vollständig neu entwickelt werden. Darüber hinaus gilt, dass die Einbindung von RMI einen bedeutenden Programmieraufwand und die Implementierung von zusätzlichem Code mit sich bringt [Lyo02].

Seit geraumer Zeit existieren auch für eingebettete Systeme Java-Implementierungen. Diese unterliegen jedoch häufig Einschränkungen hinsichtlich der vorhandenen Rechenkapazität und vor allem des zur Verfügung stehenden Speichers. Das System TINI [Dal04] bietet beispielsweise eine eingebettete JVM. Das System verfügt über 512 kB bzw. über 1 MB Speicher und unterschiedliche Peripherie wie z.B. Ethernet, CAN und serielle Anschlüsse. Die JVM ist weitestgehend kompatibel zu Java 1.1, unterstützt jedoch u.a. kein RMI. Das Echtzeit-System *aJile* [aJi04] auf Basis eines Java-Prozessors ist kompatibel zur *Java 2 Micro Edition* (*J2ME*) Spezifikation [Sun02a]. Für J2ME ist mit dem J2ME *RMI Optional Package* [Sun03b] erst seit kurzem eine optionale Bibliothek verfügbar.

Zur einfacheren Implementierung von RMI existiert u.a. die Software *Transparent RMI* (*TRMI*) [Sou03]. Bei TRMI wird nicht vorausgesetzt, dass ein Interface definiert wird. Lopes stellt mit dem Framework *D* resp. Java-Framework *DJ* in [Lop97] ein System zur Separation von Implementierung und verteilter Kommunikation vor.

[6] rmic

2.3.3.3 Jini

Das wichtigste Konzept von Jini sind verteilte Dienste, die über ein Netzwerk miteinander kommunizieren. Ein Dienst bzw. Service ist entweder eine Software, eine Hardware oder eine Kombination aus Soft- und Hardware. Jini ist plattformunabhängig und verfolgt konzeptionell die Idee einer spontanen Vernetzung von Diensten.

Die Jini-Implementierung von Sun [Sun03c] basiert größtenteils auf RMI. Demzufolge werden Jini-Services analog zu Abschnitt 2.3.3.2 entwickelt. Die Schnittstelle eines Jini-Services ist, wie die Implementierung der Schnittstelle, identisch zu 2.4 bzw. 2.5. Die Unterschiede liegen, wie aus Listing 2.6 ersichtlich wird, hauptsächlich in der Registrierung des Dienstes beim Namensdienst und der Verwendung von Zusatzfunktionalität (z.B. Implementierung der Methode *serviceIDNotify*) von Jini.

```
package jini.calc;

import java.rmi.*;
import java.rmi.server.*;
import net.jini.core.lookup.*;
import net.jini.core.entry.*;
import net.jini.discovery.*;
import net.jini.lookup.*;
import net.jini.lookup.entry.*;

public class CalcServer
extends UnicastRemoteObject
implements CalcServerI, ServiceIDListener
{
  public CalcServer() throws RemoteException {}

  // Implementierung der Methoden
  ...

  public void serviceIDNotify(ServiceID id) {
    System.out.println("CalcServer.serviceIDNotify: id = " + id);
  }

  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      CalcServer cs = new CalcServer();
      LookupDiscoveryManager mgr =
        new LookupDiscoveryManager(null, null, null);
      JoinManager manager = new JoinManager(cs, null, cs, mgr, null);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 2.6: Implementierung eines Jini-Services

In dem Beispiel wurde auf Attribute (z.B. Adresse und Lokalisierung des Dienstes), die bei der Registrierung an die Klasse *JoinManager* übergeben werden können, und auf zahlreiche andere Möglichkeiten, die in Jini zur Verfügung stehen, verzichtet.

Die Jini-Implementierung von Sun Microsystems benötigt aufgrund des RMI Activation Frameworks und einigen anderen verwendeten Klassen als Minimum die *Java Version 2 Standard Edition* (*J2SE*). JMatos [Psi01], eine Jini-Implementierung für eingebettete Systeme, benötigt keine Unterstützung von RMI seitens der JVM und steht u.a. für TINI

zur Verfügung.

Die Integration von eingebetteten Systemen in Jini kann teilweise nur mit bestimmten Techniken, z.B. mittels Proxies, realisiert werden [NG00]. Durch die Abhängigkeit von Jini zu TCP und UDP ist es teilweise schwierig, wenn nicht unmöglich, Jini auf anderen Kommunikationsmedien mit anderen Protokollen einzusetzen. Beveridge kommt in [BK02] bei der Integration des Systems RoSES [NK00] zur Realisierung von robusten und flexiblen verteilten eingebetteten Systemen zu dem Schluss, dass eine Implementierung von Jini auf CAN zwar realisierbar, aber nur mit einem hohen Aufwand und einem umfangreichen Reengineering möglich ist. In Abschnitt 6.3.3.2 werden mehrere Möglichkeiten zur Integration von Systemen in Jini erörtert und ein konkretes Beispiel für die Integration von CANopen-Modulen in Jini mittels Proxies vorgestellt.

2.3.3.4 .NET Remoting

Analog zu RMI bietet .NET das sog. *.NET Remoting*. Mit .NET Remoting ist es möglich, mit .NET-Objekten über Prozess- und Systemgrenzen hinweg zu interagieren. Konzeptionell basiert diese Technologie ebenfalls auf dem entfernten Methodenaufruf. Zur Trennung von Client und Server empfiehlt es sich, analog zu RMI eine Schnittstelle wie in Listing 2.7 zu deklarieren. Die Definition einer Schnittstelle ist in .NET, im Gegensatz zu RMI , nicht zwingend notwendig.

```
namespace calc
{
  public interface CalcServerI
  {
    double add(double arg1, double arg2);
    ...
  }
}
```

Listing 2.7: Definition der .NET Remoting-Schnittstelle

Die C#-Klasse *CalcServer* in Listing 2.8 ist von der Klasse *MarshalByRefObject* abgeleitet und implementiert das angegebene Interface. In der *Main*-Methode wird beispielhaft eine Instanz der Klasse beim Namensdienst von .NET registriert.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace calc
{
  public class CalcServer : System.MarshalByRefObject, CalcServerI
  {
    public CalcServer() { }

    // Implementierung der Methoden
    ...

    static void Main(string[] args) {
      CalcServer cs = new CalcServer();
      int port = 1001;

      try {
        TcpServerChannel channel = new TcpServerChannel(port);
```

```
        ChannelServices.RegisterChannel(channel);
        RemotingConfiguration.RegisterWellKnownServiceType(
          typeof(CalcServer), "CalcServer", WellKnownObjectMode.SingleCall
        );
      } catch(Exception e) {
        Console.WriteLine(e.Message);
      }
    }
  }
}
```

Listing 2.8: Implementierung der .NET Remoting-Schnittstelle

Im Gegensatz zu RMI findet sich in der Schnittstelle kein Hinweis auf die verwendete Middlewaretechnologie, so dass diese unabhängig von einer bestimmten Middleware ist. Die Remote-Klasse, die die Schnittstelle implementiert, muss jedoch von MarshalByRefObject bzw. von einer von MarshalByRefObject abgeleiteten Klasse abgeleitet sein[7].

2.3.3.5 Web Services

Web Services entwickeln sich aktuell zum Standard für verteilte Applikationen im Internet und bieten u.a. einen entfernten Prozeduraufruf auf Basis von XML und dem Hypertext Transfer Protokoll (HTTP). Web Services basieren auf der Idee des XML-RPC, der von Microsoft unter der Bezeichnung *SOAP* (*Simple Object Access Protocol*) weiterentwickelt wurde. Zur Beschreibung eines Web Services dient die *Web Services Description Language* (*WSDL*) [Wor01a]. WSDL ist eine XML-Grammatik zur Spezifikation der Schnittstelle eines Web Services. Web Services sind inzwischen vom *World Wide Web Consortium* (*W*^{3}*C*) standardisiert. Im Folgenden soll die Realisierung von Web Services in Java und C# an Beispielen betrachtet werden.

Java-Web Services Idealerweise besteht der eigentliche Web Service in Java nur aus der eigentlichen Klasse gemäß Listing 2.1 und besitzt keinerlei Verbindung mit der Middleware. Es handelt sich um eine Java-Klasse, die in einen Applikationsserver wie Tomcat [Apa04c] mittels eines Servlets integriert wird. Die Interaktion mit der Klasse und die Kommunikation mit einem Client auf Basis von SOAP wird durch das Servlet realisiert. Ein Java-Client, der mittels SOAP auf einen Web Service zugreift, ist in Listing 2.9 dargestellt. Im Beispiel realisiert die Methode *calc* den entfernten Prozeduraufruf auf Basis von SOAP. Dabei werden die aufzurufende Methode bzw. deren Namen und die Parameter in einer SOAP-Nachricht verpackt und per HTTP an den Server übertragen. Der Server entpackt die Anfrage, führt die entsprechende Methode aus und schickt das Ergebnis, ebenfalls in Form einer SOAP-Nachricht, an den Client zurück.

```
package jws.calc;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
```

[7] Für Marshal-by-reference Objekte wird ein Proxy erzeugt, der von Clients zum entfernten Zugriff auf ein Objekt benutzt wird.

```
public class CalcClient
{
  private double calc(String op, double arg1, double arg2)
    throws SOAPException, MalformedURLException
  {
    Call call = new Call();
    call.setTargetObjectURI("urn:examples:calcserver");
    call.setMethodName(op);
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

    Vector params = new Vector();
    Parameter p1 = new Parameter("arg1",double.class,new Double(arg1),null);
    Parameter p2 = new Parameter("arg2",double.class,new Double(arg2),null);
    params.addElement(p1);
    params.addElement(p2);
    call.setParams(params);

    URL url = new URL("http://localhost:8070/soap/servlet/rpcrouter");
    Response resp = call.invoke(url, "");

    if (resp.generatedFault()) {
      Fault fault = resp.getFault();
      System.err.println("Generated fault: " + fault);
      return Double.NaN;
    } else {
      Parameter result = resp.getReturnValue();
      return ((Double) result.getValue()).doubleValue();
    }
  }
}
```

Listing 2.9: Java-Web Service Client

Bei der Realisierung von Web Services in Java besteht keinerlei Bindung zwischen der Middleware und dem Serverobjekt. Die Bindungen finden sich in diesem Fall nur beim Client. Zur Konstruktion, zum Verschicken und zur Auswertung von Nachrichten sind mehrere Schritte notwendig, die zwar teilweise durch generische Methoden gekapselt werden können, aber dennoch implementiert werden müssen.

Für eingebettete Systeme gelten wiederum spezielle Bedingungen. Web Services basieren auf XML und HTTP[8]. Demzufolge müssen Systeme, die Web Services direkt benutzen resp. anbieten, XML und HTTP unterstützen. Für diese Technologien gilt, dass entsprechende Bibliotheken verfügbar sind und bestimmte Anforderungen hinsichtlich der zur Verfügung stehenden Rechenkapazität bzw. des zur Verfügung stehenden Speichers gestellt werden. Mit MinML-RPC [Wil01b] steht mit ca. 50 KB eine Java-Implementierung von XML-RPC für das eingebettete System TINI zur Verfügung. eSOAP [EXO02a] ist eine Implementierung von SOAP für das eingebettete System TpICU [EXO02b] und ist kompatibel zur SOAP-Spezifikation 1.1. Das Java Web Services Developer Pack (Java WSDP) enthält die o.g. Schlüsseltechnologien zur Realisierung von Web Services basierend auf Java [Sun02b] für Standard-PCs. Das Paket umfasst u.a. Technologien zum XML-basierten Nachrichtenaustausch, zur Verarbeitung von XML und zur XML-basierten RPC-Kommunikation.

[8] Anstatt HTTP können auch andere Protokolle, z.B. das Simple Mail Transfer Protocol (SMTP) [Pos82] oder das File Transfer Protocol (FTP) [PR85], verwendet werden.

.NET Web Services Mit C# bzw. einem der .NET-Dialekte ist ein Web Service ebenfalls sehr leicht realisierbar, wobei eine Klasse, die als Web Service angeboten werden soll, bestimmte Eigenschaften erfüllen muss. Die Klasse muss, wie in Listing 2.10 dargestellt, zum einen von der Klasse *WebService* abgeleitet sein. Zum anderen muss jede Methode, die als entfernte Methode zur Verfügung stehen soll, mit dem Attribut *WebMethod* gekennzeichnet werden. Zur Integration des Web Services in den *Internet Information Server* (*IIS*) findet sich in der ersten Zeile eine Anweisung zur Identifikation der Programmiersprache und der entsprechenden Klasse. Diese Anweisung wird zur Laufzeit gelesen, der entsprechende Code übersetzt und die angeforderte Methode ausgeführt.

```
<%@ WebService Language="C#" Class="calc.CalcServer" Debug=true %>

using System;
using System.Web.Services;

namespace calc
{
  public class CalcServer : WebService
  {
    [WebMethod]
    public double add(double arg1, double arg2) {return arg1+arg2;}
    ...
  }
}
```

Listing 2.10: .NET-Web Service

Aus Listing 2.10 ist ersichtlich, dass wiederum eine enge Bindung zwischen der Funktionalität und der Middlewarearchitektur besteht. Darüber hinaus gilt für Web Services in .NET, dass sie sehr stark von dem Integrationsframework (IIS, .NET-Framework) abhängig sind. Eine Betrachtung von .NET-Web Services auf eingebetteten Systemen scheidet deshalb à priori aus.

2.3.3.6 Integrationsmiddleware

Im Rahmen dieser Arbeit wird Middleware nicht nur als Technologie zur verteilten Kommunikation, sondern auch als Technologie zur lokalen Kommunikation betrachtet, die eine Basis für die Integration von unterschiedlichen Systemen bietet. Ein Beispiel für eine solche Middleware ist das *Java Native Interface* (*JNI*) [Lia99]. JNI bietet die Möglichkeit, von Java aus auf betriebssystemabhängige Bibliotheken (z.B. Windows Dynamic Link Libraries, DLLs) zuzugreifen und damit die Möglichkeit zur Integration von C/C++-Systemen. Dies ist z.B. dann notwendig, wenn in Java Hardware angebunden werden soll.

Bei der Realisierung einer JNI-Schnittstelle wird üblicherweise zunächst eine Java-Klasse deklariert. Die sog. nativen Methoden werden mit dem Schlüsselwort *native* gekennzeichnet. Der JNI-Compiler generiert aus der übersetzten Klasse eine Header-Datei für eine Systembibliothek in C bzw. C++, in der die nativen Methoden deklariert sind. Auf Basis dieser Header-Datei wird die Bibliothek implementiert und in den Java-Code eingebettet. Listing 2.11 enthält ein Beispiel mit nativen Methoden. Die zugehörige Bibliothek wird mit der Anweisung *System.loadLibrary* geladen.

```
package jni.calc;

public class Calc
{
  public double add(double a, double b) {return _add(a, b);}
  ...

  public native double _add(double a, double b);
  ...

  static {
    System.loadLibrary("Calc");
  }
}
```

Listing 2.11: Definition eines Java Native Interface

Die Bibliothek enthält u.a. die in Listing 2.12 dargestellte Implementierung einer nativen Methode. Die Signatur kann aus der erzeugten Header-Datei übernommen werden.

```
JNIEXPORT jdouble JNICALL Java_jni_calc_CalcServer__1add
  (JNIEnv *env, jobject obj, jdouble arg1, jdouble arg2)
{
  return (arg1+arg2);
}
```

Listing 2.12: Implementierung einer JNI-Methode

Die Realisierung einer JNI-Schnittstelle beginnt üblicherweise mit der Deklaration der nativen Schnittstelle in Java und setzt sich bis zur Implementierung der Bibliothek fort. Falls bereits eine Bibliothek, z.B. mit mathematischen Funktionen, existiert, so hat die beschriebene Vorgehensweise trotzdem Gültigkeit. Darüber hinaus ist die eigentliche Funktionalität der Bibliothek durch die Signaturen und die verwendeten Datentypen sehr eng an JNI gebunden. Dies ist notwendig, da die Java-Datentypen auf C++-Datentypen wechselseitig abgebildet werden müssen.

Die Anbindung von Bibliotheken kann durch die Verwendung von generischen Paketen, z.B. JNI Shared Stubs [Sun98], vereinfacht werden. Ein Shared Stub ist eine einzige native Methode, die beliebige C-Funktionen einer Betriebssystembibliothek aufrufen kann. Damit kann auf die Generierung von Header-Dateien und Implementierung der Methoden verzichtet werden. Zur vollständigen Abbildung einer Bibliothek in Java ist es jedoch sinnvoll, entsprechende Java-Methoden zur Kapselung der nativen Methoden, wie in Listing 2.11 gezeigt, zu implementieren. Die Schnittstelle wird dadurch mit mehr Semantik angereichert, als dies bei generischen Methoden der Fall ist.

2.3.3.7 COM/DCOM

COM/DCOM ist zusammen mit den Technologien ActiveX und OLE eine der führenden Technologien zur komponentenbasierten Softwareentwicklung. Auf die technischen Details von COM wird in Abschnitt 3.3 noch etwas näher eingegangen. Es sei jedoch an dieser Stelle bereits erwähnt, dass COM nicht plattformunabhängig ist [Ses98] und entsprechende Mechanismen notwendig sind, um auf COM-Schnittstellen und -Klassen aus Java zuzugreifen. Die verfügbaren Schnittstellen benutzen dazu entweder eine JNI- oder eine protokollbasierte Kommunikationsschnittstelle. Die JNI-basierten Schnittstel-

len basieren auf einem Mechanismus analog zu Abschnitt 2.3.3.6. Die protokollbasierten Schnittstellen basieren auf der RPC-Funktionalität von COM/DCOM.

Es stehen eine Vielzahl von unterschiedlichen Produkten zur Anbindung von COM in Java wie z.B. JIntegra [Int01b], Jacob [Adl99] oder JCom [Wat98] zur Verfügung. Teilweise ist die Anbindung von COM und ActiveX integraler Bestandteil von bestimmten Technologien wie z.B. bei JavaBeans [Ham97]. In dieser Arbeit findet eine modifizierte Version der frei verfügbaren Java/COM-Bridge JCom [Wat98] Verwendung. Bei JCom handelt es sich um eine JNI-basierte Middleware zum Zugriff auf COM-Komponenten und -Typbibliotheken gemäß Abschnitt 3.3.1. Die modifizierte Java/COM-Middleware, im Folgenden auch als Java2COM (s. Abschnitt 3.3.3) bezeichnet, umfasst einige für diese Arbeit notwendige Anpassungen und Erweiterungen.

Zum Zugriff auf COM-Komponenten auf Basis der Java2COM-Middleware werden eine Schnittstelle gemäß Listing 2.13 und eine Implementierung dieser Schnittstelle gemäß Listing 2.14 realisiert. Bei den gezeigten Beispielen handelt es sich bereits um automatisch generierten Code, der eine COM-Komponente in Java auf generische Weise kapselt. Das gezeigte Beispiel müsste nicht identisch, aber in ähnlicher Form, manuell implementiert werden. Eine COM-Schnittstelle wird auf eine Java-Schnittstelle abgebildet, wobei jede Methode die Ausnahme Java2ComException[9] definiert.

```
package com.calc;

public interface _Calc
{
  String IID = "{EBA45A9E-182F-36FB-91B0-14542DD6BC1B}";
  String ProgID = "com.calc.Calc";

  public double add(double arg1, double arg2) throws java2com.Java2ComEx...;
  ...
}
```

Listing 2.13: Definition einer COM-Schnittstelle

Die Klasse *_CalcDelegate* ist von *java2com.IDispatch* abgeleitet und implementiert in den einzelnen Methoden den Zugriff auf die COM-Komponenten mittels der Methode *invoke()* (Zeile 19) mit entsprechenden Parametern.

```
package com.alc;

public class _CalcDelegate
extends java2com.IDispatch
implements _Calc
{
  public _CalcDelegate(java2com.ReleaseManager rm)
    throws java2com.Java2ComException
  {
    super(rm, _Calc.ProgID);
  }

  public double add(double arg1, double arg2)
    throws java2com.Java2ComException
  {
    double _pVal[] = { 0.0d };
    Object _pRetVal[] = { _pVal };
```

[9] java2com.Java2ComException

```
    Object _pDispParams[] = { new Double(arg1), new Double(arg2) };
    Object rc = invoke("add", java2com.IDispatch.INVOKE_FUNC, _pDispParams);
    convert(rc, _pRetVal);
    return (double)_pVal[0];
  }

  ...
}
```

Listing 2.14: Implementierung der COM-Schnittstelle

Dieses Beispiel demonstriert, dass eine variable Anbindung mit sehr großem Programmieraufwand verbunden ist. Während das gezeigte Beispiel die Anbindung einer COM-Komponente mit nur einer COM-Klasse mit einer Methode realisiert, können bestimmte COM-Komponenten mehrere 100 Klassen und diese wiederum hunderte von Methoden besitzen. Eine manuelle Implementierung ist in zuletzt genanntem Fall nicht sinnvoll. Deshalb verfolgen bereits einige der oben genannten Produkte, z.B. JIntegra, Jacob wie auch MAX, die automatische Generierung von Java-Klassen zur Kapselung von COM-Komponenten. Während JIntegra direkt Java-Klassen erzeugt, wird bei MAX und Jacob ein Zwischenformat auf Basis von XML zur Speicherung von Typinformationen (s. Abschnitt 4.2.3) benutzt.

2.4 XML & Co.

Die eXtensible Markup Language (XML) entwickelt sich zum universellen Format für strukturierte Dokumente und Daten. Mit der Spezifikation von XML als internationaler Standard für den plattform-, sprach- und anwendungsunabhängigen Informationsaustausch haben sich das WWW und die Organisation von Daten grundlegend verändert. XML ist inzwischen integraler Bestandteil vieler moderner Informationssysteme zur Speicherung und zum Austausch von Informationen. XML ist eine Metasprache, d.h. eine Sprache für Sprachen, die eine Vielzahl von domänenspezifischen Sprachen und industriellen Markup-Sprachen hervorgebracht hat. XML umfasst eine Reihe von Standards und Technologien, welche dem Informationsaustausch über das WWW und der Entwicklung von Kommunikationsinfrastrukturen neue Möglichkeiten eröffnen [Coy02].

In diesem Abschnitt soll eine Einführung in Markup-Sprachen, XML und ein Überblick zu XML verwandten Technologien und Standards wie z.B. XSL und XML Inclusions, die in dieser Arbeit Verwendung finden, gegeben werden.

2.4.1 Definitionen

2.4.1.1 Markup und Markup-Sprachen

Bei *Markup*[10] handelt es sich gemäß ISO [Int86] um Informationen, die einem Dokument hinzugefügt werden, um dessen Bedeutungsgehalt zu erweitern. Ein Dokument mit Markup enthält somit die eigentlichen Daten und die Daten über diese Daten, den Markup. Diese zusätzlichen Daten werden auch als Metadaten bezeichnet.

[10] zu deutsch Textauszeichnung

Eine *Markup-Sprache* definiert, welcher Markup zulässig und notwendig ist, wie dieser Markup von der eigentlichen Information unterschieden werden kann und was er bedeutet. Eine *Markup-Sprache* umfasst eine Menge von Zeichen und eine Reihe von Regeln, wie diese Zeichen zu einem Dokument hinzugefügt werden, um somit die Struktur oder die Darstellung eines Dokuments zu bestimmen.

2.4.1.2 Arten von Markup

In der Literatur wird prinzipiell zwischen *generischem* und *visuellem* Markup unterschieden [Bry88]. Beim generischen (auch generalisierten oder strukturierten) Markup wird ein Dokument um semantische Information angereichert, die die einzelnen Teile eines Dokumentes nach logischen Gesichtspunkten beschreibt. Zu den generischen Markup-Sprachen, auch Markup-Metasprachen genannt, zählen die die *Standardized General Markup Language* (*SGML*) [Int86] und die *eXtensible Markup Language* (XML) [Wor98b].

Im Gegensatz zum generischen Markup ergänzt das visuelle (auch spezifische oder darstellungsorientierte) Markup den Text um Formatierungsanweisungen, die das Aussehen, also die Formatierung und das Layout, eines Dokumentes festlegen. So definiert beispielsweise HTML die Struktur und das Layout von Web-Dokumenten durch eine Reihe von Tags[11] und Attributen [INT04].

2.4.2 Die eXtensible Markup Language (XML)

XML selbst ist keine Markup-Sprache, sondern eine Metasprache zur Spezifikation von anderen Sprachen. Sie wird als erweiterbar bezeichnet, da sie kein fixes Format wie HTML definiert, sondern durch die Definition von eigenen Tags erweitert werden kann. Analog zu SGML dient XML als Metasprache zur Definition von Dokumenttypen.

2.4.2.1 XML-Dokumente

Ein XML-Dokument enthält ausgezeichnete Daten, deren Struktur durch eine Dokumenttyp-Definition (DTD) beschrieben wird. Jedes XML-Dokument besteht somit aus den beiden Teilen Definition und Instanz. Eine DTD definiert die Elemente und Attribute, die in einem XML-Dokument zulässig sind. Sie ist dann notwendig, wenn ein Dokument auf seine Gültigkeit geprüft werden soll.

2.4.2.2 Das Document Object Model

Das Document Object Model (DOM) ist eine Abbildung eines XML-Dokuments in eine hierarchische Datenstruktur. Zur Verarbeitung dieser Struktur stehen unterschiedliche Möglichkeiten zur Verfügung. Zum einen kann über das sog. Simple API for XML (SAX) ein XML-Dokument seriell verarbeitet werden. Dabei wird ein XML-Dokument während des Einlesens untersucht und beim Auftreten bestimmter Tags eine entsprechende Methode aufgerufen, welche die Daten verarbeitet. Bei DOM wird das komplette XML-

[11] Bezeichnung für eine Markierung in einer Seitenbeschreibungssprache.

Dokument in eine hierarchische Datenstruktur im Speicher geladen. Die Knoten des Baumes können über eine komfortable Programmierschnittstelle adressiert und verarbeitet werden. Im Rahmen dieser Arbeit wird ausschließlich DOM verwendet.

2.4.3 Die eXtensible Style Sheet Language (XSL)

Die eXtensible Style Sheet Language [Wor03] ist eine XML-Anwendung zur Transformation von XML-Dokumenten. XSL basiert auf der Document Style Semantics and Specification Language (DSSSL) [Int96] und benutzt einige Elemente von Cascading Style Sheets (CSS) [Wor98a]. XSL kann dazu benutzt werden, um eine Struktur auf ein Dokument anzuwenden oder die Elemente vollständig umzuordnen. Mit XSL ist es beispielsweise möglich, ein XML-Dokument mit einer bestimmten Struktur in ein HTML-Dokument, ein PDF-Dokument oder in ein anderes Format mit einer vollständig anderen Struktur zu konvertieren. Beispiele für die Konvertierung von XML-Dokumenten nach HTML und PDF finden sich in Abschnitt 5.6.

XSL Style-Sheets definieren, wie ein strukturiertes Dokument präsentiert werden soll. XSL umfasst die XML Path Language (XPath), XSL Transformations (XSLT) und XSL Formatting Objects (XSL FO).

2.4.3.1 XPath

XPath wird dazu benutzt, bestimmte Teile von XML-Dokumenten zu adressieren bzw. zu identifizieren. XPath definiert eine eigene, nicht XML-basierte Syntax und eine Reihe von Funktionen zur Navigation in einem XML-Dokument. Die Syntax ermöglicht die Beschreibung von Pfaden zu bestimmten Teilen eines Dokuments. Diese Pfade entsprechen dem Weg von einem Knoten zu einem anderen Knoten eines Baumes. Somit definiert XPath eine Möglichkeit zur Selektion von Knoten in diesem Baum.

2.4.3.2 XSLT

Die XML Style Sheet Language Transformation, kurz XSLT, ist eine Sprache, um XML-Dokumente in andere XML-Dokument zu überführen. Dabei wird durch bestimmte Regeln angegeben, welche Daten des Quelldokuments in das Zieldokument übernommen werden. XSLT dient hauptsächlich der Filterung und Aufbereitung von bestehenden XML-Dokumenten. Die Transformationen werden von sog. XSL-Prozessoren wie z.B. Xalan [Apa04f], XT oder Saxon [Kay04] realisiert. Xalan findet z.B. in dem XML Publikationssystem Cocoon [Apa04e] (s. Abschnitt 5.6.1.1) Verwendung.

2.4.3.3 XSL FO

Bei der Formatierung von XML-Daten mittels *XSL Formatting Objects* (*XSL FO*) werden die Ergebnisse einer XML-Transformation von einer Beschreibung in eine Repräsentation gemäß Abb. 2.4 überführt.

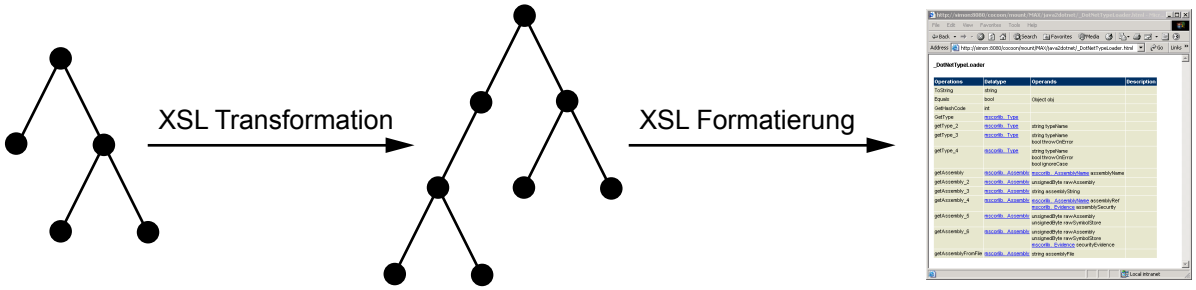


Abbildung 2.4: XSLT-Transformation und Formatierung

XSL FO-Dokumente sind XML-Dokumente, die ein bestimmtes Vokabular zur Seitenformatierung von Dokumenten (Block, Tabelle, Liste, usw.) spezifizieren. Beispielsweise können mit XSLT Daten eines XML-Dokuments extrahiert und um Formatierungsoptionen erweitert werden. Diese Formatierungsoptionen werden dann durch ein Formatierungswerkzeug wie z.B. FOP [Apa04d] in ein PDF-Dokument konvertiert. Diese Vorgehensweise wird bei der Generierung von PDF-Dokumenten in Abschnitt 5.6.1.3 angewandt.

2.4.4 XML Inclusions und XML Linking Language

XML Inclusions (*XInclude*) und die *XML Linking Language* (*XLink*) bieten eine Möglichkeit zur Referenzierung von externen XML-Dokumenten.

2.4.4.1 XInclude

XInclude [Wor01b] bietet eine Möglichkeit, ein logisch zusammengehöriges XML-Dokument in einzelne Teildokumente zu untergliedern. Dabei werden die Teildokumente per Referenz in ein Dokument eingefügt. Diese Vorgehensweise bietet sich dann an, wenn ein einzelnes Dokument zu groß und damit zu unübersichtlich wird. Die Referenzen werden üblicherweise beim Einlesen eines Dokumentes aufgelöst und durch den Inhalt des referenzierten Dokumentes ersetzt, so dass die Trennung in Teildokumente bei der Verarbeitung verborgen bleibt.

2.4.4.2 XLink

Die Möglichkeit, Ressourcen über Links miteinander zu verbinden, hat zu einem großen Teil zum Erfolg des Web beigetragen. Links in HTML, z.B. die Tags ”a” oder ”img”, sind zur Verlinkung von XML-Dokumenten nicht ausreichend. XLink erlaubt die Deklaration von einfachen unidirektionalen Links und komplexen Link-Strukturen, die es ermöglichen, Verbindungen zwischen mehreren Ressourcen herzustellen und die einzelnen Links mit Metadaten anzureichern.

2.4.5 Eigenschaften von XML

XML stellt eine bedeutende, wenn auch keine neue Entwicklung in der Informationstechnologie dar und entwickelt sich zum Standard für den Austausch von strukturierten Daten. XML-APIs sind für viele Programmiersprachen und Plattformen verfügbar, unterstützen die XML-Spezifikationen aber teilweise nicht im vollen Umfang. Es existieren Implementierungen, die aufgrund von Ressourcen-Einschränkungen nicht die vollständige Spezifikation, sondern nur Teile davon implementieren. Beispielsweise existiert für das embedded System TINI [Dal04] zwar eine XML-Implementierung namens MinML [Wil01a], diese unterstützt aber nicht den vollen Umfang von XML. Die Unterstützung von XInclude stellt bei XSL-Prozessoren eher die Ausnahme als die Regel dar.

2.5 CAN/CANopen

MAX unterstützt neben Java, COM/DCOM und .NET auch die Verarbeitung von CANopen-Geräteprofilen und die automatische Generierung von Softwarekomponenten zur Kommunikation mit CANopen-Geräten in Java.

2.5.1 Einführung

Analog zur Kommunikation in Rechnersystemen existieren im Bereich der Automation ebenfalls eine Vielzahl von Kommunikationsnetzen und -protokollen. Eine grundsätzliche Entwicklung stellen jedoch Feldbusse dar. Ein Feldbus ist ein serieller Bus, der mehrere Komponenten miteinander verbindet. Anstatt einer aufwendigen Verkabelung von Komponenten mit einer Zentrale (Steuerung) werden mehrere, mit einer Kommunikationsschnittstelle ausgerüstete, Komponenten in Reihe geschaltet. Die Kommunikationsschnittstelle umfasst sowohl Hard- als auch Software. Als Konsequenz verfügen die einzelnen Geräte über mehr Funktionalität („Intelligenz“) und können einfach hinzugefügt, entfernt und ausgetauscht werden.

2.5.2 CAN

Eine Feldbus-Technologie, die sich u.a. aus Kostengründen im Automations- und Automotivebereich durchgesetzt hat, ist das *Controller Area Network* (CAN) [Int93]. CAN spezifiziert die unteren zwei Schichten des Open Systems Interconnection Schichtenmodells (OSI) der ISO, die physikalische und die Verbindungsschicht.

2.5.3 CANopen

Basierend auf CAN existieren unterschiedliche Applikationsprotokolle (Schicht 7 des ISO/OSI-Schichtenmodells). Die am weitesten verbreiteten Protokolle sind CANopen [CAN00a] und DeviceNet [Ope04]. CANopen bietet mit dem Konzept von sog. Kommunikationsprofilen eine objektorientierte Sichtweise auf CANopen-Komponenten. Die Kommunikation zwischen einem Master und den Slaves spezifiziert sog. *Service Data*

Objects (*SDOs*) und *Process Data Objects* (*PDOs*). Während SDOs hauptsächlich zur Parametrierung verwendet werden, werden PDOs für den Echtzeitdatentransfer im Produktionsbetrieb benutzt. Detaillierte Informationen und Hinweise auf weiterführende Literatur finden sich auf der Webseite der Organisation CAN in Automation (CiA) [CAN03].

2.5.3.1 CANopen-Geräteprofile

Die CiA spezifiziert Geräteprofile für unterschiedliche Gerätetypen, wobei ein Profil die Mindestfunktionalität eines bestimmten Gerätetyps definiert. Das Geräteprofil für generische Ein-/Ausgabe-Module DS401 [CAN99] enthält z.B. die Mindestfunktionalität für digitale und analoge Ein-/Ausgabe-Module. Ein Geräteprofil liegt außer als Spezifikation noch in Form eines elektronischen Datenblatts vor, dem sog. *EDS* (*Electronic Data Sheet*). Im Normalfall liegt ein EDS als Datei vor, falls jedoch ein Gerät über genügend Speicher verfügt, so kann dieses EDS auch in einem Gerät gespeichert sein. Auf das EDS wird in Abschnitt 3.5.1 noch detailliert eingegangen. Im Allgemeinen definieren Geräteprofile die Schnittstelle von CANopen-Komponenten. Auf Basis dieser Profile werden die notwendigen Daten, die zur Kommunikation mit CANopen-Komponenten notwendig sind, gemäß Abschnitt 3.5.1 extrahiert und zur automatischen Generierung von Softwarekomponenten gemäß Abschnitt 5.5.1 herangezogen.

2.5.3.2 Java/CANopen-Middleware

Zum Zugriff auf CAN- bzw. CANopen-Systeme in Java steht eine komfortable Schnittstelle zur Verfügung. Die Java/CANopen-Middleware ermöglicht das Versenden und Empfangen von CAN- und CANopen-Nachrichten in Java. Die Middleware umfasst die Pakete *java2can* und *java2canopen*, wobei letzteres Paket auf ersterem basiert. CAN-Nachrichten werden durch die Klasse *CANFrame* repräsentiert. Zur CANopen-Kommunikation stehen u.a. die Klassen *SDO* und *PDO* zur Verfügung. Neben diesen Klassen stehen weitere Klassen zum Management, z.B. Netzwerkmanagement (NMT), von CAN- und CANopen-Systemen zur Verfügung.

Die Programmiersprache Java eignet sich zur Kapselung dieser Systeme sehr gut, da CANopen ebenfalls einen objektorientierten Ansatz verfolgt. Um von Java auf CAN-Hardware zugreifen zu können, ist es jedoch notwendig mit dem CAN-Softwaretreiber bzw. der Bibliothek, die den Treiber kapselt, zu kommunizieren. Auf der Ebene des Betriebssystems kann auf CAN bzw. auf die CAN-Hardware durch eine herstellerspezifische API zugegriffen werden. Um diese API unter Java zu nutzen, wird sie in einer Bibliothek gekapselt. Aufgrund des Sicherheitskonzeptes von Java ist dies nur über JNI realisierbar. JNI stellt dazu eine Möglichkeit zur Einbindung von nativen Funktionen in Form von Betriebssystembibliotheken zur Verfügung. Eine Übersicht der einzelnen Schichten zur CAN/CANopen-Kommunikation in Java ist in Abb. 2.5 dargestellt.

Abhängig vom System bzw. der zur Verfügung stehenden API basiert die Java/CANopen-Middleware entweder auf Java- oder auf C bzw. C++-APIs. Das eingebettete System TINI bietet beispielsweise eine Java-API zum Versenden und Empfangen von CAN-Nachrichten, so dass diese API direkt in Java genutzt werden kann. Auf PCs

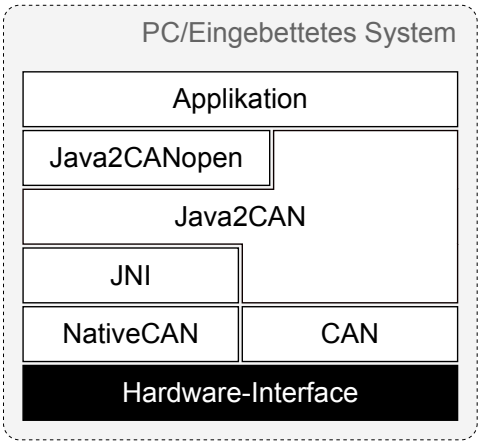


Abbildung 2.5: Die Java/CANopen-Middleware

und dem Industrie-PC SICOMP werden hardware-spezifische CAN-Bibliotheken (z.B. vcand32.dll[12] oder com167.dll[13]) über JNI gekapselt.

Die Java2CAN-Schicht ist die Basis für die Java2CANopen-Schicht und bildet eine Abstraktionsschicht zur CAN-Kommunikation. Im Gegensatz zu [GNBK99, BNGK99, BN00, BNKG01] wird auf ein Management von CAN-Nachrichten in den unteren Kommunikationsschichten (NativeCAN) aus Gründen der Portabilität und im Hinblick auf die Integration von weiteren Systemen verzichtet. Zur asynchronen Verarbeitung von Nachrichten steht ein asynchroner Benachrichtigungsmechanismus zur Verfügung [NGK02]. Dabei können sich Objekte für bestimmte Ereignisse registrieren und werden bei Eintritt dieser Ereignisse benachrichtigt. Bezogen auf CAN bedeutet dies, dass sich Objekte für das Eintreffen bestimmter CAN-Nachrichten registrieren und beim Eintreffen dieser Nachrichten informiert werden. Der asynchrone Benachrichtigungsmechanismus ist u.a. zur Beobachtung von Sensoren sehr wertvoll, da Daten nur bei einer Zustandsänderung bearbeitet bzw. übertragen werden müssen und somit die notwendige Rechenleistung und Übertragungsbandbreite minimiert werden können [NBGK01].

2.6 Zusammenfassung

Die Wiederverwendung von Artefakten stellt ein wichtiges Kriterium für die Realisierung von kostengünstigen und zuverlässigen Systemen dar. Dabei spielt die Art und Weise, wie Artefakte wiederbenutzt werden, nur eine untergeordnete Rolle. Die Wiederverwendung von existierenden Komponenten stellt aufgrund der verwendeten Programmiersprachen oder Betriebssystemen oft ein Problem dar [GAO95].

Während die objektorientierte Softwareentwicklung für die Wiederverwendung zwar förderlich, aber keineswegs ausreichend ist, werden Komponentensysteme als die Systeme der Zukunft betrachtet. Die Einteilung von unterschiedlichen Komponententechnologien in den Komponentenraum von Thomason [Tho00] mit den Dimensionen Verteilt-

[12] zur Anbindung der CAN-Hardware von Vector [Vec03]
[13] zur Anbindung der SICOMP Baugruppe COM167 von Siemens [Sie03]

heit, Modularität, Plattform- und Sprachunabhängigkeit gemäß Abschnitt 2.3.1.3 verdeutlicht jedoch, dass einige Architekturen bzw. Systeme keine optimale Lösung darstellen. Für Komponentensysteme, die auf Microsoft-Technologien beruhen, gilt beispielsweise, dass sie nicht ohne weiteres auf andere Plattformen portiert werden können [Ses98].

Middleware kann wesentlich zur Integration von Systemen beitragen. Die unterschiedlichen Technologien besitzen gewisse Vor- und Nachteile. Die Beispiele in Abschnitt 2.3.3 haben gezeigt, dass die Anwendungslogik häufig sehr eng mit einer bestimmten Technologie verknüpft ist, so dass eine Anwendung nur mit erheblichem Aufwand oder, ohne die zur Verfügung stehenden Quellen, überhaupt nicht in eine andere Technologie integriert werden kann. Teilweise ist eine Integration von unterschiedlichen Technologien nur schwer realisierbar, da eine entsprechende Schnittstelle zur einfachen Integration fehlt. Aus den genannten Gründen fällt die Entscheidung, welche Technologie zur Realisierung eines Systems eingesetzt wird, sehr schwer und ist meist nicht mit vertretbarem Aufwand umkehrbar. Die Situation wird dadurch verschärft, dass jedes System dem Gesetz der kontinuierlichen Veränderung unterliegt und zukünftige Anforderungen kaum abzuschätzen sind. Eine sich daraus ergebende Schlussfolgerung ist, dass keine Middlewaretechnologie allen Anforderungen gerecht und von allen Plattformen und Sprachen gleichermaßen unterstützt wird.

In dieser Arbeit erfolgt die Softwarewiederverwendung durch Kapselung von existierenden Komponenten in neue, automatisch generierte Softwarekomponenten. Neben den existierenden Komponenten können die ebenfalls automatisch generierten Schnittstellendefinitionen dieser Komponenten beliebig oft und zur Generierung von unterschiedlichen Artefakten wiederverwendet werden. In MAX liegt der Schwerpunkt auf der Abstraktion, der Spezialisierung und der Integration und weniger auf der Selektion von Softwarekomponenten. Die Abstraktion wird zur Generalisierung von Komponenten und zur Generierung von neuen Komponenten genutzt, wobei die neuen Komponenten bei der automatischen Generierung spezialisiert und damit in unterschiedliche (Middleware-) Systeme integriert werden.

3

Metaisierung von Komponenten

Ein zentrales Konzept von MAX ist die automatische Bestimmung der Schnittstelle von Komponenten. Die Idee ist, die Schnittstelle mittels Reflexion bzw. Introspektion gemäß Abschnitt 3.1.1 automatisch zu extrahieren und in ein Metamodell gemäß Abschnitt 3.1.6 abzubilden. In den Abschnitten 3.2 ff. werden die Metamodelle von Java, COM/-DCOM, .NET und CANopen betrachtet. Die Ergebnisse und Konsequenzen, die sich für die weitere Vorgehensweise im Zusammenhang mit MAX ergeben, werden am Ende des Kapitels zusammengefasst.

3.1 Metamodelle und Reflexion

Technologien auf Basis von Reflexion und Metaarchitekturen haben inzwischen ein Stadium erreicht, bei dem sie zur Lösung von Problemen in unterschiedlichen Bereichen eingesetzt werden: Programmiersprachen, Betriebssysteme, Datenbanken, Softwareentwicklung, Middlewaresysteme oder Web-basierte Anwendungen [CCL01]. Bevor auf Metamodelle im allgemeinen und auf die Metamodelle der einzelnen Programmiersprachen bzw. Technologien im besonderen eingegangen wird, sollen zunächst einige Begrifflichkeiten bezüglich des Konzepts der Reflexion definiert, charakterisiert und erläutert werden.

3.1.1 Reflexion

Reflexion ist ein weitreichendes Konzept, das in vielen unterschiedlichen wissenschaftlichen Bereichen und vor allem im Bereich der Informatik untersucht wurde [DM95]. Der Begriff der Reflexion geht ursprünglich auf die Bereiche Philosophie und Logik zurück. Im Bereich der künstlichen Intelligenz ist Reflexion das Ziel, ein „intelligentes Verhalten“ zu entwickeln. Nachdem Reflexion als faszinierende, aber eher komplexe Technologie betrachtet wurde, stellte und stellt Reflexion ein wichtiges Konzept beim Design von Architekturen und Sprachen [Fer89] und bei der Entwicklung von offenen Systemen [Foo92] dar.

Das Konzept der *Reflexion* geht im Umfeld der Informatik auf Smith [Smi82] zurück und beschreibt nach Maes [Mae87] die Fähigkeit eines Systems „Aussagen über sich selbst zu treffen“ und „sich selbst zu verändern“. Diese zwei Aspekte werden als *Introspektion*[1] und *Interferenz* bezeichnet. Introspektion ist die Fähigkeit, Objekte eines Systems zu inspizieren, jedoch nicht zu verändern [Foo92]. Interferenz ist die Fähigkeit eines Programmes, den eigenen Ausführungstatus oder die eigene Interpretation zur Laufzeit zu verändern. Beide Aspekte benötigen dazu einen Mechanismus, der es erlaubt, implizite Eigenschaften eines System zu vergegenständlichen. Dieser Prozess wird als Reifikation[2] bezeichnet [BGW93].

3.1.2 Reflektive Architektur

Eine Programmiersprache besitzt laut Maes [Mae87] eine *reflektive Architektur*, falls sie Reflexion als integralen Bestandteil bzw. als fundamentales Konzept ansieht und Werkzeuge zur Handhabung von Reflexion bietet. Formal bietet eine reflektive Architektur eine Technik, die Entitäten einer Domäne D_n, genannt Basissystem, in eine andere Domäne D_{n+1}, bezeichnet als Metasystem, zu überführen. Jede Domäne kann dabei als Basisdomäne für eine höhere Schicht, oder als Metadomäne für eine niedrigere Schicht, mit Ausnahme der Domäne D_0, dienen [Fer89]. Die einzelnen Schichten bilden einen sog. reflektiven Turm [Fer89, Smi82] von Interpretern. Dabei wird ein Programm A von einem Programm B, dem Metaprogramm, ausgeführt. Dieses Metaprogramm kann wiederum durch ein Programm, dem Meta-Metaprogramm, ausgeführt werden usw. Üblicherweise werden die Metaprogramme alle in derselben Sprache bzw. durch dieselbe Struktur repräsentiert.

3.1.3 Reflektive Eigenschaften

Eine Architektur oder ein System besitzt *reflektive Eigenschaften*, falls eine Untermenge der oben genannten Fähigkeiten gegeben ist. Die Programmiersprache Java unterstützt mit der *Java Reflection API* zwar Reflexion, diese ist jedoch weitestgehend auf die Introspektion beschränkt. Die Möglichkeit, den Ausführungsstatus eines Programmes zur Laufzeit zu verändern sind auf die Instantiierung neuer Objekte, das Schreiben und Lesen von Werten oder den Aufruf von Methoden beschränkt [Chi00].

3.1.4 Reflektive Systeme, Sprachen und Anwendungen

Ein *reflektives System* oder Programm enthält Strukturen, die ein System bzw. Programm repräsentieren. Diese Strukturen werden als Selbst-Repräsentation bezeichnet [Mae87]. Dabei ist entscheidend, dass ein System zu jedem Zeitpunkt eine korrekte Repräsentation von sich selbst besitzt und diese Repräsentation konsistent ist, d.h. sie entspricht dem aktuellen Status des Systems. Falls ein System eine korrekte und konsistente Repräsentation besitzt, so besteht eine kausale Verbindung zwischen dem System und dessen Reprä-

[1] Introspektion = „Hineinsehen“, Selbstbeobachtung
[2] Reifikation = Vergegenständlichung, Konkretisierung

sentation. Die Existenz einer kausalen Verbindung ist eine notwendige Eigenschaft von reflektiven Systemen. Die Domäne eines reflektiven Systems bilden die Struktur und die Aktionen des Systems selbst [GK97a]. Eine *reflektive Programmiersprache* ist eine Programmmiersprache, die Reflexion unterstützt und, analog zur reflektiven Architektur (s. Abschnitt 3.1.2), Reflexion als integralen Bestandteil einer Programmiersprache ansieht und somit Schnittstellen zur Handhabung von Reflexion bietet. *Reflektive Anwendungen* sind Anwendungen, die sich zur Realisierung einer bestimmten Aufgabe der Reflexion bedienen.

Foote [Foo92] sieht in der Vereinigung von Objekten und Reflexion ein „enormes Potential und die Möglichkeit, wirklich offene Programmiersprachen und Systeme zu realisieren“. Dabei ändere sich „die Art, wie Programmiersprachen und Systeme organisiert, implementiert und benutzt werden, dramatisch“ [Foo92].

3.1.5 Charakterisierung der Reflexion

3.1.5.1 Strukturelle Reflexion vs. Verhaltensreflexion

Nach Ferber [Fer89] gibt es zwei Typen von Reflexion, die strukturelle Reflexion und die Verhaltensreflexion[3]. Erstere setzt voraus, dass die Sprache eine Reifikation des momentan ausgeführten Programmes und der abstrakten Datentypen unterstützt (z.B. RTTI[4]). Letztere verlangt eine Reifikation der Semantik und der Daten, die zur Ausführung eines Programmes benutzt werden. „Strukturelle Reflexion vergegenständlicht strukturelle Aspekte eines Programmes wie Datentypen, Methodennamen oder Vererbungsbeziehungen“ [GK97a]. Bei der Verhaltensreflexion werden Operationen des Laufzeitsystems reifiziert. In objektorientierten Sprachen, z.B. in Java und C#, zählen zu dieser Art von Operationen Aufrufe von Methoden oder Operationen zur Instantiierung von Objekten und zum Zugriff auf Variablen.

3.1.5.2 Direkte vs. indirekte Reflexion

Im Kontrast zu Abschnitt 3.1.2 gibt es Systeme, die keine direkte Schnittstelle zur Reflexion bieten, aber dennoch reflektive Eigenschaften besitzen bzw. bei denen sich indirekt ein ähnliches Konzept realisieren lässt. In Rahmen dieser Arbeit wird deshalb zwischen *direkter* und *indirekter Reflexion* unterschieden. Erstere genügt der Definition in Abschnitt 3.1.2, bei letzterer sind reflektive Eigenschaften realisierbar, aber nicht inhärenter Bestandteil eines Systems.

3.1.5.3 Dynamische vs. statische Reflexion

Zur Unterscheidung, welches Artefakt zur Reflexion bzw. genauer zur Introspektion herangezogen wird, wird in dieser Arbeit zwischen *dynamischer* und *statischer Reflexion* unterschieden. Während bei der dynamischen Reflexion die Introspektion zur Laufzeit,

[3] computational reflection
[4] Run-Time Type Identification

also z.B. Zugriff auf ein Laufzeit-Objekt, stattfindet, wird bei der statischen Reflexion auf statische Artefakte, wie z.B. ein elektronisches Dokument (z.B. EDS, s. Abschnitt 2.5.3.1), zur Gewinnung von Metainformationen zugegriffen.

3.1.5.4 Reflexion zur Übersetzungs-, Lade- oder Laufzeit

Ein Programm durchläuft die Phasen Übersetzung (compile time), Binden (link time), Laden (load time) und Ausführung (run time). Während jeder dieser Phasen kann das Konzept der Reflexion angewendet werden (s. [GK97a]).

- Bei der *Reflexion zur Übersetzungszeit* wird ein Programm während der Übersetzung durch ein Metaprogramm kontrolliert und modifiziert [GK97a]. Da in dieser Phase der gesamte Quelltext vorliegt, kann auf sämtliche Abschnitte eines Programmes Einfluss genommen werden. Diese Art der Reflexion besitzt den Nachteil, dass die Reflexion ausschließlich zur Übersetzungszeit durchgeführt werden kann und der Quelltext eines Programmes vorliegen muss. In einer späteren Phase kann keine Reflexion mehr durchgeführt werden. In diesem Sinne erfüllt die Reflexion zur Übersetzungszeit nicht die Anforderungen der Reflexion nach Definition 3.1.
- Eine *Reflexion zur Ladezeit* wird während des Ladens und Bindens eines Programmes durchgeführt. Analog zur Reflexion zur Übersetzungszeit verändert ein Metaprogramm den Lade- und Linkprozess. In dieser Phase kann Reflexion z.B. zur Adaption eines Programmes an die aktuelle Umgebung oder zur dynamischen Komposition von Programmen genutzt werden.
- Die *Reflexion zur Laufzeit* ist sehr flexibel, da Reflexion während der Ausführung eines Programmes genutzt werden kann, um Auskunft über den aktuellen Zustand eines Programmes zu erhalten und die Ausführung zur Laufzeit zu verändern.

3.1.5.5 Beispiele reflektiver Systeme, Sprachen, Anwendungen

Das Konzept der Reflexion ist in unterschiedlichen Programmiersprachen realisiert. Die prozedurale Reflexion [Smi84] beschreibt die Reflexion in einer seriellen Programmiersprache und fand erstmals in der Programmiersprache 3-Lisp, einem Dialekt von Lisp, Verwendung. Weitere Anwendungen wurden in logikbasierten, regelbasierten und erstmals in einer von Maes entwickelten reflektiven objektorientierten Architektur namens 3-KRS realisiert [Mae87].

Viele moderne Programmiersprachen wie Java oder die .NET-Dialekte (z.B. C#) besitzen reflektive Eigenschaften. In diesen Sprachen wird Reflexion beispielsweise zur Serialisierung und Deserialisierung von Objekten zur persistenten Speicherung oder zum Transport von Objekten über ein Kommunikationsnetzwerk benutzt. In Java basieren beispielsweise RMI oder die Objektserialisierung auf der Java Reflection API.

Die Programmiersprache C++ bietet mit der Typidentifikation zur Laufzeit (RTTI, s. Abschnitt 3.1.5.1) eine Möglichkeit zur strukturellen Reflexion. Die Komponentenarchitektur COM/DCOM erlaubt auf Basis der Schnittstellen *IUnknown* und *IDispatch* strukturelle Reflexion und eine eingeschränkte Verhaltensreflexion.

Zur automatischen Code-Generierung werden bei COM/DCOM sog. Typbibliotheken herangezogen. Diese dienen als Basis zur Generierung von Software zur Kapselung von COM-Komponenten. Dabei können die Typbibliotheken mit Werkzeugen visualisiert und zur weiteren Verarbeitung genutzt werden.

In Java, CORBA und RPC werden mittels einer dynamischen (in Java mit *rmic*) bzw. statischen (CORBA, RPC) strukturellen Reflexion Metadaten zur Generierung von Kommunikationsmiddleware (Stubs und Skeletons) herangezogen. Bei .NET kann mittels Attributen definiert werden, welche Methoden eines Objektes als Web Service zur Verfügung stehen sollen. Auf Basis dieser Information wird der Code zur Kommunikation über SOAP zur Laufzeit automatisch generiert, übersetzt und zur Ausführung gebracht.

3.1.6 Metamodell

Ein Modell bietet eine vereinfachende und abstrahierende Darstellung von Artefakten. Ein Metamodell ist ein Modell, welches die Konzepte einer Modellierungstechnik abbildet und die Modellierungselemente, die im Rahmen eines Modellierungsansatzes verwendet werden können, beschreibt [BG03, Völ03]. Viele moderne Programmiersprachen und -konzepte beinhalten die Definition eines Metamodells, wobei jede Programmiersprache ihr eigenes Metamodell definiert und auf die speziellen Eigenschaften des jeweiligen Systems angepasst ist.

Ein *Metasystem* ist ein System, dessen Domäne ein anderes System, ein Basissystem, ist [Fer89]. Objekte der Metaebene, kurz *Metaobjekte*, sind Objekte, die eine Anwendung des Basissystems (der Basisebene) definieren, implementieren oder auf andere Weise an der Ausführung einer Anwendung beteiligt sind [Foo92]. In objektorientierten Sprachen – basierend auf Klassen und Instanzen – kann die Klasse eines Objektes als Metaobjekt betrachtet werden [Fer89]. Eine *Metaklasse* ist die Repräsentation einer Klasse einer Programmiersprache und enthält Informationen der Klasse, die sie repräsentiert. In Java ist beispielsweise die Klasse *java.lang.Class* die Metaklasse aller Klassen.

Die einzelnen Metaebenen, auch Abstraktionsebenen, bilden zusammen eine *Metadaten-Architektur* gemäß Abb. 3.1. Prinzipiell kann eine solche Architektur analog zu Abschnitt 3.1.2 beliebig viele Ebenen umfassen.

3.1.7 Metaisierung

Der Prozess der Abbildung von Objekten bzw. Komponenten in ein Metamodell wird in dieser Arbeit als *Metaisierung* oder *Abstraktion* definiert. Die Metaisierung beschreibt die Extraktion der Schnittstelle eines Objekts bzw. einer Komponente und die Abbildung dieser Schnittstelle in ein Metamodell. Das Gegenteil der Metaisierung – also die Abbildung von Objekten des Metamodells in konkrete Objekte – wird als *Konkretisierung* oder *Spezialisierung* bezeichnet. Die genannten Prozesse sind in Abb. 1.1 systematisch dargestellt.

Wie bereits eingangs erwähnt, stellt die Reflexion bzw. Introspektion einen wesentlichen Aspekt dieser Arbeit dar. Mittels Reflexion werden die Metadaten einer Komponente extrahiert. Die Extraktion konzentriert sich dabei im wesentlichen auf die Extraktion der

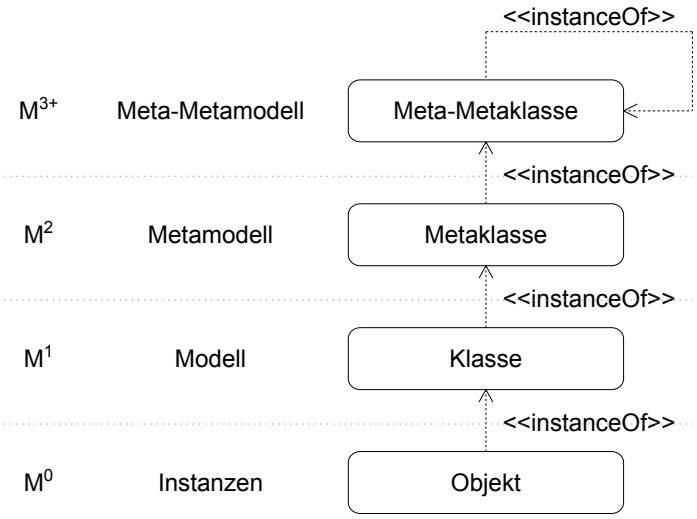


Abbildung 3.1: 4-Schichten Metadaten-Architektur

Schnittstelle zur Kommunikation mit der entsprechenden Komponente bzw. der Daten, die zur automatischen Generierung einer Schnittstelle genutzt werden können. Die Metaisierung von Komponenten bzw. Artefakten basiert auf der direkten oder indirekten, statischen oder dynamischen strukturellen Reflexion und beschränkt sich damit in erster Linie auf die Introspektion von Artefakten.

Das in dieser Arbeit realisierte System MAX unterstützt die Metaisierung von Java, .NET, COM/DCOM, CANopen und verschiedenen Verzeichnisdiensten. Jedes dieser Systeme besitzt zwar reflektive Eigenschaften, unterstützt aber einen unterschiedlichen Grad an Reflexion. So stehen unterschiedliche Schnittstellen und Konzepte zur Durchführung der Reflexion zur Verfügung. Während Java, .NET und COM/DCOM eine komfortable Schnittstelle zur Reflexion bieten, basiert die Reflexion von CANopen und den Verzeichnisdiensten im wesentlichen auf der Extraktion von Name/Wert-Paaren. Im folgenden werden die genannten Systeme hinsichtlich ihrer reflektiven Eigenschaften betrachtet und die Realisierung zur Extraktion der Metadaten beschrieben.

3.2 Metaisierung von Java-Komponenten

Ein Java-Objekt ist im Prinzip system- und maschinenunabhängiger Byte-Code, der entweder aus dem lokalen Dateisystem oder über ein Netzwerk geladen und durch eine Java Virtual Machine (JVM) ausgeführt wird. Das Konzept der Reflexion ist ein wichtiger Bestandteil von Java. Während sich in der Version 1.0 die Reflexion auf die Reifikation von Klassen beschränkte, wie z.B. die Reflexion des Namens, der Superklasse und der implementierten Schnittstellen, ist Reflexion seit der Version 1.1 expliziter Bestandteil von Java in Form der *Java Reflection API* im Paket *java.lang.reflect*. Das Paket umfasst u.a. Klassen zur Reifikation von Konstruktoren, Methoden, Feldern und den zugehörigen Modifizierern.

Java bietet mit der Java Reflection API eine sehr komfortable Möglichkeit, auf die Me-

tadaten von Klassen und Objekten zuzugreifen und unterstützt somit Reflexion auf der Ebene der Programmiersprache. Die einzelnen Klassen und die Beziehungen der Klassen zueinander werden im folgenden an dem Metamodell von Java erläutert.

3.2.1 Das Metamodell von Java

Das Metamodell von Java umfasst die Klassen *java.lang.Object*, *java.lang.Class* und Klassen der Reflection API. Die Beziehungen bzw. Abhängigkeiten der Klassen zueinander sind im Klassendiagramm in Abb. 3.2 dargestellt.

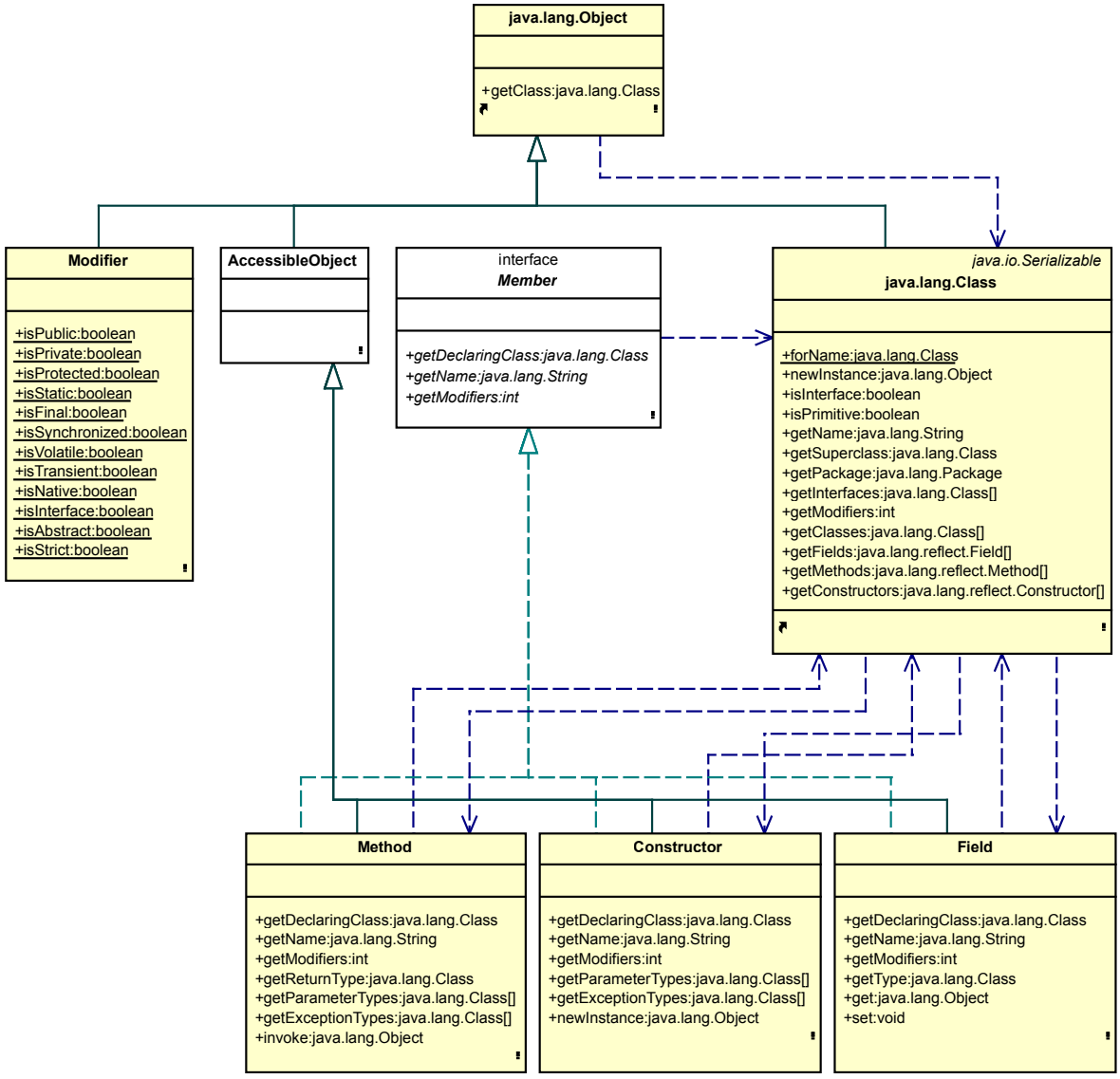


Abbildung 3.2: Das Metamodell von Java

Die Klasse *java.lang.Object* ist die Wurzel der Klassen-Hierarchie, von der jede Klasse in Java abgeleitet ist. Die Klasse definiert neben einigen anderen Methoden eine Methode namens *getClass()*, die ein Objekt des Typs *Class* zurückliefert und als Einstiegspunkt zur

Extraktion von Metadaten dient. Instanzen der Klasse *Class* repräsentieren Klassen, abstrakte Klassen und Schnittstellen (Interfaces). Die Schnittstelle der Klasse *Class* bietet Methoden zur Extraktion von Metadaten der Klasse oder des Interface selbst, wie z.B. des Namens, des Namensraum, der Superklasse, der implementierten Schnittstellen und die von einer Klasse deklarierten Konstruktoren, Methoden und Variablen. Die Konstruktoren, Methoden und Variablen werden durch die folgenden Klassen repräsentiert bzw. reifiziert:

- *java.lang.reflect.Constructor*
- *java.lang.reflect.Method*
- *java.lang.reflect.Field*

Diese Klassen implementieren die Schnittstelle *java.lang.reflect.Member* und damit die von *Member* deklarierten Methoden zur Bestimmung des Namens und zur Bestimmung der Modifizierer (z.B. *public* oder *private*).

Die Klasse *java.lang.reflect.Constructor* kapselt Informationen über einen Konstruktor einer Klasse. Die Klasse bietet neben den von Member deklarierten Methoden, Methoden zur Extraktion der Parameter- und der Ausnahme-Typen. Zum Aufruf eines Konstruktors mit entsprechenden Parametern dient die Methode *newInstance()*.

Eine Methode einer Klasse oder einer Schnittstelle wird durch die Klasse *java.lang.reflect.Method* repräsentiert. Diese Klasse implementiert die von Member deklarierten Methoden und bietet analog zum Konstruktor Methoden zur Bestimmung der Signatur und zur Bestimmung des Rückgabetyps. Methoden, die durch ein Objekt vom Typ *Method* gekapselt sind, können mittels *invoke()* aufgerufen werden.

Instanzen der Klasse *java.lang.reflect.Field* bieten Informationen über und Zugriff auf Member-Variablen einer Klasse. Da sämtliche, auch einfache Datentypen in Java durch eine Klasse repräsentiert werden, kann zur Bestimmung des Datentyps die Methode *getType()* genutzt werden, die ein Objekt vom Typ *Class* zurück liefert. Zum dynamischen Zugriff auf eine Variable definiert die Klasse *Field* für jeden Datentyp jeweils die Methoden *get()* und *set()*. Beispielsweise kann der Wert eine Variable des Datentyps *int* mit *getInt()* gelesen und mit *setInt()* gesetzt werden.

Die Klasse *java.lang.reflect.Modifier* bietet statische Methoden zur Bestimmung der Modifizierer einer Klasse oder eines Members (Konstruktoren, Methoden, Variablen). Mittels dieser Klasse kann z.B. bestimmt werden, ob es sich bei einer Klasse um eine Klasse, eine abstrakte Klasse oder eine Schnittstelle handelt. Die Methode *getModifiers()* der Klasse *Class* liefert den Modifizierer, codiert als Integer-Wert, zurück.

Mit der Java Reflection API steht eine komfortable Schnittstelle zur Identifikation und Extraktion von Metadaten von Klassen und Objekten zur Laufzeit zur Verfügung. Die Metadaten spezifizieren die Schnittstelle, die zur Interaktion mit einer Klasse notwendig ist. Im folgenden werden die konkreten Schritte zur Extraktion der Metadaten erläutert.

3.2.2 Extraktion der Metadaten

Zur Extraktion der Metadaten einer Klasse benötigt das System den vollständig qualifizierten Namen einer Klasse und deren Lokation. Der Name einer Klasse besteht aus dem Namensraum, in dem die Klasse definiert ist, und dem Namen der Klasse. Die Lokation einer Klasse wird über einen *URL* (*Uniform Resource Locator*) definiert.

MAX unterstützt das Laden von einfachen Klassen, d.h. Klassen mit der Endung *.class* im Klassenpfad und von Klassen, die in einem Java-Archiv gespeichert sind. Als Java-Archiv werden sowohl ZIP-Archive (.zip) als auch JAR-Archive (.jar) unterstützt. Durch die Verwendung einer URL können die Klassen entweder vom lokalen Dateisystem oder von einem entfernten Rechner geladen werden.

Die zu metaisierende Klasse wird unter Angabe des vollständig qualifizierten Namens von der angegebenen Lokation geladen. Die Methode *Class.forName(string className)* liefert ein Objekt des Datentyps *Class* zurück. Dieses Objekt wird zur Extraktion der Metadaten der gegebenen Klasse benutzt.

Zur Bestimmung der Schnittstelle einer Klasse werden die Metadaten dieser Klasse und die Member dieser Klasse mittels der Java Reflexion API extrahiert. Zu den Membern einer Klasse zählen sämtliche Konstruktoren, Methoden und Variablen. Die folgenden Metadaten einer Klasse werden mit den angegebenen Methoden der Klasse *Class* extrahiert:

- Name: *getName()*
- Namensraum: *getPackage()*
- Konstruktoren: *getConstructors()*, bzw. *getDeclaredConstructors()*
- Methoden: *getMethods()*, bzw. *getDeclaredMethods()*
- Variablen: *getFields()*, bzw. *getDeclaredFields()*
- Schnittstellen: *getInterfaces()*
- Superklasse: *getSuperclass()*

Der Unterschied zwischen den Methoden zur Bestimmung der Member, z.B. *getDeclaredMethods()* und *getMethods()* ist, dass erstere alle von einer Klasse deklarierten Member und letztere alle von einer Klasse und zusätzlich deren Superklassen liefert. Im Folgenden werden diese Methoden als äquivalent betrachtet, da nur die prinzipielle Vorgehensweise zur Extraktion von Metainformation einer Klasse verdeutlicht werden soll. Die nachfolgenden Informationen zur Reflexion von Java können weitestgehend der Java -Dokumentation ab Version 1.2 [Sun02c] entnommen werden.

- Extraktion des Namens: Die Methode *getName()* liefert eine Zeichenkette zurück, die den vollständigen Namen, d.h. den Namen des Paketes, also den Namensraum, und den Namen der Klasse (z.B. *java.lang.String*), enthält.

- Extraktion des Namensraums: Der Namensraum wird mit der Methode *getPackage()* bestimmt. Diese Methode liefert eine Instanz der Klasse *Package* zurück, die Methoden zur Extraktion von Informationen bzgl. eines Paketes spezifiziert. Die Informationen umfassen u.a. den Namen, den Hersteller und die Version eines Paketes.

- Extraktion der Konstruktoren: Die von einer Klasse deklarierten Konstruktoren werden mittels der Methode *Class.getConstructors()* ermittelt. Diese Methode liefert ein Array mit den verfügbaren Konstruktoren zurück. Die einzelnen Einträge dieses Array besitzen den Datentyp *Constructor*. Mittels Methoden dieser Klasse werden der Name, die Parameter- (*getParameterTypes()*) und die Ausnahme-Typen (*getExceptionTypes()*) des Konstruktors bestimmt. Die Methoden zur Bestimmung der Parameter- und Ausnahme-Typen liefern wiederum ein Objekt des Datentyps *Class* zurück, das zur Bestimmung der Parameter- und Ausnahmen-Datentypen benutzt wird.

- Extraktion der Methoden: Die Methoden einer Klasse werden mit der Methode *getMethods()* bestimmt. Diese Methode liefert ein Feld von Objekten des Datentyps Method zurück. Der Name, die Parameter- und Ausnahme-Typen werden analog zum Konstruktor extrahiert. Die Methode *getReturnType()* liefert wiederum ein Objekt des Datentyps *Class* zurück, das zur Bestimmung des Rückgabetyps der Methode benutzt wird.

- Extraktion der Parameter: Zur Extraktion der Parameter von Konstruktoren und Methoden kann jeweils die Methode *getParameterTypes()* benutzt werden. Diese Methode liefert ein Feld von Objekten des Typs *Class* zurück, die zur Bestimmung der Metadaten der einzelnen Parameter verwendet werden können. Da Parameter in der internen Repräsentation von Java keine Namen besitzen, werden sie in MAX durch ein ”p” gefolgt von einer fortlaufenden Nummer gekennzeichnet.

- Extraktion der Variablen: Die Member-Variablen werden mit der Methode *getFields()* extrahiert. Besitzt eine Member-Variable einen einfachen Datentyp, so wird außer dem Datentyp auch der Wert dieser Variable ausgelesen, ansonsten wird nur der Datentyp ermittelt.

- Extraktion der implementierten Schnittstellen: Falls eine Klasse eine oder mehrere Schnittstellen implementiert, so kann diese Information mit der Methode *getInterfaces()* extrahiert werden. Diese Methode liefert ein Feld mit Objekten des Datentyps *Class* zurück.

- Extraktion der Superklasse: Zur Extraktion der Superklasse dient die Methode *getSuperclass()*. Falls eine Klasse das Wurzelobjekt, ein Interface, einen primitiven Datentyp oder *void* repräsentiert, wird der Wert *null* zurückgegeben, d.h., dass die entsprechende Klasse keine Schnittstellen implementiert.

Für alle Member, d.h. alle Konstruktoren, Methoden und Variablen werden zudem mittels *getModifiers()* die zugehörigen Modifizierer bestimmt. Bei der Extraktion einer Klasse ist es möglich und wahrscheinlich, dass externe Datentypen bzw. externe Klassen

referenziert werden. Beim Laden einer Klasse von einem entfernten Rechner wird deshalb sichergestellt, dass die referenzierten Klassen automatisch nachgeladen werden. Dies ist notwendig, da die Klassen zur Reflexion lokal vorhanden sein müssen. Mit dem beschriebenen Mechanismus werden die Metadaten, die zur Kommunikation mit einer Klasse notwendig sind, also deren Schnittstelle, extrahiert.

3.2.3 Beispiel

Zur Erläuterung der oben beschriebenen Vorgehensweise soll die Reflexion von Java-Objekten an dem in Listing 2.1 vorgestellten Beispiel der Klasse *Calc* demonstriert werden. Eine Metaisierung dieser Klasse liefert die in Tabelle 3.1 dargestellten Metadaten.

	Namensraum	Name	Modifizierer	Parametertyp	Rückgabetyp
Klasse	calc	Calc	public class		
Konstruktoren		Calc	public		
Methoden		add sub ...	public public ...	double, double double, double ...	double double ...

Tabelle 3.1: Metadaten der Java-Klasse *Calc*

Falls in Java kein Konstruktor angegeben wird, existiert zumindest immer – analog zum angegeben Beispiel – der Standardkonstruktor mit einer leeren Argumentliste.

3.3 Metaisierung von COM-Komponenten

Es wurde bereits erwähnt, dass COM/DCOM ebenfalls eine Möglichkeit zur strukturellen Reflexion von Komponenten bietet. Die strukturelle Reflexion bei COM basiert auf den sog. Typbibliotheken, die Typinformationen von COM-Schnittstellen und -Klassen enthalten. Innerhalb einer Typbibliothek sind üblicherweise mehrere, logisch zusammengehörige Typen deklariert.

3.3.1 Typbibliothek und Typinformation

Eine Typbibliothek (.tlb) ist eine binäre Datei, die in einem Microsoft-spezifischen Format COM-Klassen und -Schnittstellen beschreibt [Loo01]. Prinzipiell handelt es sich bei einer COM-Typbibliothek um eine binäre IDL-Datei, die eine Beschreibung einer Komponentenschnittstelle mit sämtlichen Methoden, deren Parameter und Rückgabetypen enthält. Optional enthält eine Typbibliothek Informationen zu den einzelnen Methoden, z.B. eine Beschreibung.

Durch die Verwendung einer Typbibliothek ist es anderen Anwendungen und Werkzeugen möglich, auf die in einer Typbibliothek definierten Objekte zuzugreifen. Typbibliotheken können in Form einer separaten Datei (.tlb), als Ressource innerhalb einer Dynamic Link Library (.dll) oder innerhalb einer ausführbaren Datei (.exe) vorliegen. Falls

eine Bibliothek eine oder mehrere Typbibliotheken enthält, so handelt es sich um eine sog. Objektbibliothek (.olb).

Eine COM-Schnittstelle wird mittels der Microsoft Interface Decription Language (MIDL) und einem optionalen sog. Application Configuration File (ACF) definiert. Die MIDL spezifiziert, wie Daten zwischen einem Client und einem Server ausgetauscht werden. Eine Typbibliothek kann automatisch aus den genannten Definitionen mit dem sog. MIDL Compiler erzeugt werden und folgende Informationen enthalten:

- Information über Datentypen (z.B. Aliase (aliases), Aufzählungen (enumerations), Strukturen (structures) oder Mengen (unions)).
- Beschreibungen von einem oder mehreren Objekten (z.B. Modul (module), Interface (interface), Dispatch-Schnittstelle (dispinterface) oder COM-Klasse (component object class (coclass)). Diese Beschreibungen werden allgemein als Typinformation *typeinfo* bezeichnet.
- Referenzen auf Typbeschreibungen aus anderen Typbibliotheken.

Eine Typbibliothek enthält sämtliche Metadaten einer COM-Komponente und definiert somit die Schnittstelle, die zur Interaktion mit einer COM-Komponenten zur Verfügung steht.

3.3.2 Das Metamodell von COM

Die COM-Schnittstellen und -Klassen und deren Beziehung zueinander bilden in ihrer Gesamtheit das in Abb. 3.3 dargestellte Metamodell.

Jede COM-Klasse leitet sich direkt oder indirekt von *IUnknown* ab und implementiert die von *IUnknown* definierte Schnittstelle. Eine Implementierung dieser Schnittstelle realisiert die Verwaltung der Lebensdauer (*AddRef()*, *Release()*) und die Navigation (*QueryInterface()*) zwischen den unterschiedlichen Schnittstellen eines COM-Objekts.

Die Klasse *IDispatch* ist ebenfalls von *IUnknown* abgeleitet und bietet Mechanismen, um zur Laufzeit auf Methoden und Variablen eines Objektes zuzugreifen. Im einzelnen stellt *IDispatch* die folgenden Methoden zur Verfügung:

- *GetTypeInfoCount()* stellt fest, ob Typinformationen für ein Objekt zur Verfügung stehen.
- *GetTypeInfo()* liefert die zu einem Objekt gehörige Typinformation zurück.
- *GetIDsOfNames()* ermittelt die Dispatch-ID einer Methode oder einer Variablen.
- *Invoke()* dient zum Aufruf einer Methode bzw. zum Zugriff auf eine Variable.

Eine Typbibliothek kann mit den COM-Schnittstellen *ITypeLib* und *ITypeInfo* analysiert werden, wobei *ITypeLib* als Einstiegspunkt zum Zugriff auf Informationen einer Typbibliothek dient. Die Methoden von ITypeLib erlauben es einer Anwendung, auf individuelle Informationen einer Typbibliothek, genauer auf die in einer Typbibliothek definierten

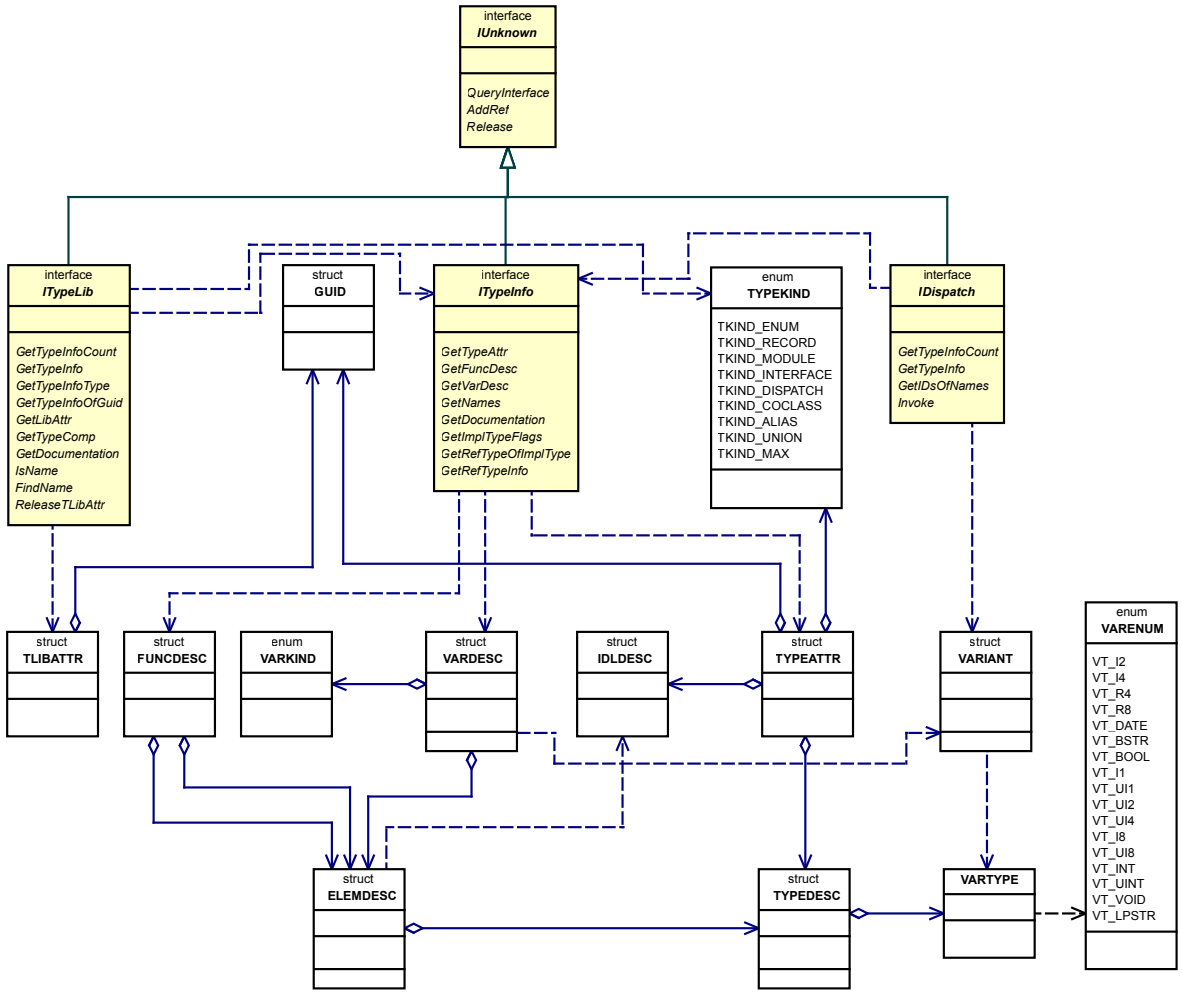


Abbildung 3.3: Das Metamodell von COM

Objekte, zuzugreifen. Die Methode *GetTypeInfoCount()* gibt die Anzahl der verfügbaren Typinformationen einer Typbibliothek zurück. Spezifische Typinformationen können mit den folgenden Methoden ermittelt werden:

- *GetTypeInfo()* bzw. *GetTypeInfoOfGuid()* erhalten als Parameter einen Index bzw. einen Identifier und liefern eine Referenz auf die zugehörige Typinformation zurück.
- *GetTypeInfoType()* ermittelt den Typ eines Objektes an einem gegebenen Index.
- *FindName()* findet die Typinformation bzw. -informationen zu einem bzw. mehreren Objekten mit einem bestimmten Namen.
- *IsName()* stellt fest, ob ein Objekt mit einem bestimmten Namen innerhalb einer Typbibliothek definiert ist.
- *GetLibAttr()* liefert in der Struktur *TLIBATTR* Informationen über eine Typbibliothek selbst, wie z.B. einen Identifier, die Versionsnummer oder die Zielplattform.

Die Schnittstelle *ITypeInfo* ist die umfangreichste Schnittstelle[5] in COM und erlaubt den Zugriff auf die Typinformationen von COM-Objekten.

- *GetTypeAttr()* liefert eine Struktur vom Typ *TYPEATTR* zurück. Diese Struktur enthält Informationen über einen bestimmten Typ. Sie enthält beispielsweise die Anzahl der Variablen und Methoden, die von einem Typ definiert werden, dessen Identifier (*GUID*) und die Versionsnummer. Eine Referenz auf den Aufzählungstyp *TYPEKIND* enthält die Information, ob es sich um einen Aufzählungstyp (*TKIND_ENUM*), eine Struktur (*TKIND_RECORD*), ein Modul mit statischen Funktionen (*TKIND_MODULE*), eine Schnittstelle (*TKIND_INTERFACE*), eine Dispatch-Schnittstelle (*TKIND_DISPATCH*), eine Klasse (*TKIND_COCLASS*), eine Vereinigung (*TKIND_UNION*) oder einen Alias (*TKIND_ALIAS*) für einen bereits definierten Typ handelt. Entsprechend der Art eines Typs kann auf weitere Informationen zugegriffen werden.

- *GetFuncDesc()* gibt unter Angabe eines Index einen *Funktionsdeskriptor* vom Typ *FUNCDESC* zurück und enthält Informationen über eine bestimmte Funktion. Diese Struktur umfasst u.a. einen Identifier vom Typ *MEMBERID*, die Aufrufkonvention, die Anzahl der Parameter und einen Zeiger auf den ersten Parameter. Der Rückgabetyp einer Funktion kann durch eine Referenz auf einen Elementdeskriptor (s. *ELEMDESC*) bestimmt werden. Mittels dem Identifier kann der Funktionsname oder die zu einer Funktion gehörende Dokumentation extrahiert werden.

- *GetVarDesc()* erhält als Parameter einen Index und liefert eine Datenstruktur vom Typ *VARDESC*, einen sog. *Variablendeskriptor* zurück. Ein Variablendeskriptor kapselt Informationen über eine bestimmte Variable. Ein Identifier vom Typ *MEMBERID* identifiziert eine Variable eindeutig und kann u.a. dazu benutzt werden, den Namen einer Variable zu extrahieren. Entsprechend des Typs einer Variable (s. *VARKIND*) enthält der Deskriptor entweder einen Verweis auf einen *VARIANT* oder auf eine Instanz der Variablen. Die Elemente eines Aufzählungstyps werden ebenfalls als Variablen betrachtet.

- *GetNames()* liefert den Namen einer Variablen, einer Methode oder die Namen der Parameter. Diese Methode erhält als Parameter wiederum einen Index, der die Variable bzw. Methode innerhalb der Bibliothek eindeutig identifiziert.

- *GetDocumentation()* gibt entweder eine zu einer Variable oder Methode gehörige Beschreibung oder den Pfad und Namen einer Datei, in der die zugehörige Beschreibung gespeichert ist, inkl. eines Identifiers zur Lokalisierung der Beschreibung innerhalb dieser Datei, zurück.

Der Typ einer Klasse, einer Methode (Rückgabetyp), eines Parameters oder einer Variablen kann durch einen Typdeskriptor vom Datentyp *TYPEDESC* bestimmt werden. Der Deskriptor enthält die Kennung eines Datentyps (*VARTYPE*) und bildet somit einen indirekten Verweis auf den Aufzählungstyp *VARENUM*, der eine Liste aller elementarer COM-Datentypen enthält.

[5] Das Klassendiagramm enthält nur eine Untermenge der verfügbaren Methoden.

Ein Elementdeskriptor wird durch die Datenstruktur *ELEMDESC* repräsentiert und enthält außer einer Referenz auf einen Typdeskriptor (s.o.) noch Richtungsattribute oder zusätzliche Informationen zu einem Parameter. Ein Elementdeskriptor realisiert über den Typdeskriptor den Zugriff auf Informationen eines Parameter-, Variablen- oder Rückgabetyps.

Ein IDL-Deskriptor vom Typ *IDLDESC* enthält die Richtungsattribute (in, out) der Parameter einer Methode. Die Datenstruktur *GUID* kapselt den global eindeutigen Bezeichner und zerlegt diesen in einzelne Teilbereiche[6].

Der Datentyp *VARIANT* enthält, entsprechend dem COM-Datentyp, entweder einen einfachen Wert oder eine Referenz auf einen Wert.

3.3.3 Die Java2COM-Middleware

Das in dieser Arbeit realisierte System ist weitestgehend in Java implementiert. Um die bestehende Infrastruktur zu benutzen und in Java auf COM-Objekte zugreifen zu können, wird deshalb ein Mechanismus benötigt, der die COM-Schnittstelle durch eine entsprechende Java-Schnittstelle kapselt.

Wie bereits in Abschnitt 2.3.3.7 erwähnt, wird in der vorliegenden Arbeit eine modifizierte Version der Java/COM-Brücke JCom [Wat98] verwendet. Die Java2COM-Middleware bietet eine Reihe von Klassen, die über JNI auf die Windows-COM-Schnittstelle zugreifen. Dabei werden in Java entsprechende Pendants zu Objekten der COM-Schnittstelle bereitgestellt. Die Brücke gehört zur Klasse der Dispatch-Produkte und definiert u.a. die Klassen *IDispatch*, *ITypeLib*, *ITypeInfo* und *IUnknown*. Darüber hinaus stehen ein Großteil der unter Abschnitt 3.3.2 aufgeführten Datenstrukturen in Form von inneren Klassen zur Verfügung. Das zugehörige Klassendiagramm ist in Abb. 3.4 dargestellt.

3.3.4 Extraktion der Metadaten

Zur Extraktion von COM-Komponenten in Java wird eine Instanz eines COM-Objektes oder eine entsprechende Typbibliothek, in der die Metadaten gespeichert sind, benötigt. Das System unterstützt entweder die Instantiierung eines COM-Objektes durch Angabe von dessen Namen oder das Laden von Typbibliotheken aus dem Dateisystem bzw. über die Windows Registry.

Eine Instanz eines COM-Objekts vom Typ *IDispatch* kann durch die Angabe eines Klassenidentifikators (CLSID) bzw. durch einen sog. *Programmatic Identifier* (*ProgID*) erzeugt werden. Eine Typbibliothek wird entweder direkt aus dem Dateisystem oder durch Angabe eines global eindeutigen Bezeichners (*GUID*) indirekt über die Windows Registry geladen. Beim Laden der Bibliothek wird ein Objekt vom Typ *ITypeLib* zurückgegeben.

Beide Klassen, *IDispatch* und *ITypeLib*, bieten die oben beschriebene Methode *GetTypeInfo()*, die unter Angabe eines Index eine Referenz auf die entsprechende Typinformation

[6]Der Grund dieser Zerlegung ist, dass nicht alle Systeme einen 128-Bit Integer-Wert unterstützen.

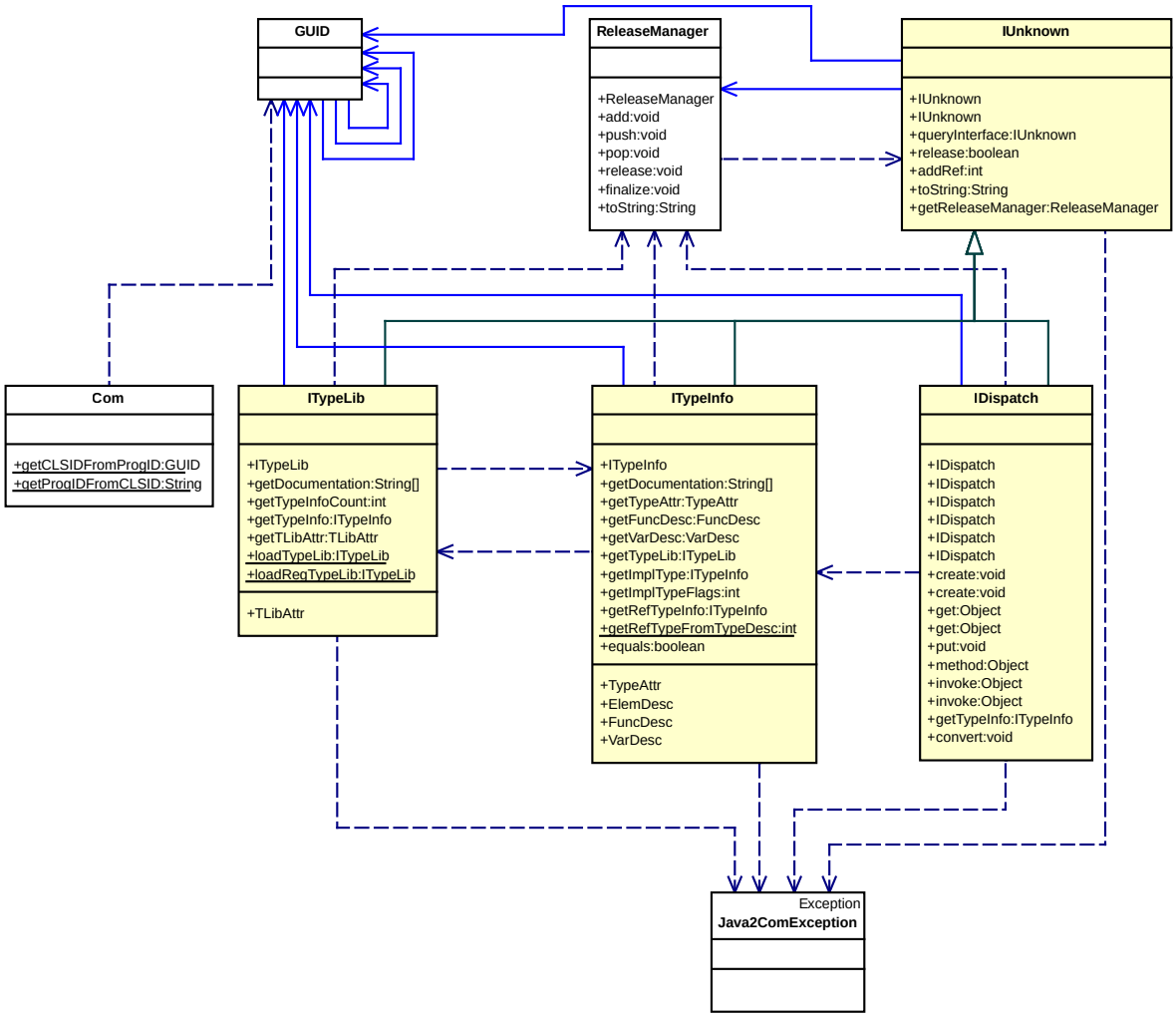


Abbildung 3.4: Klassendiagramm der Java2COM-Middleware

vom Typ *ITypeInfo* zurückliefert. Entsprechend des COM-Typs werden sämtliche Informationen mit den unter Abschnitt 3.3.2 vorgestellten Methoden extrahiert.

Üblicherweise definieren COM-Komponenten eine Vielzahl von Klassen und Schnittstellen. Beispielsweise umfasst die Typbibliothek von Excel ca. 550 Typen. Zudem werden ca. 200 Klassen und Schnittstellen aus externen Bibliotheken referenziert. Soll nur ein bestimmter Typ bzw. dessen Schnittstelle extrahiert werden, so benötigt das System den vollständig qualifizierten Namen des Typs.

Die Metaisierung von COM-Komponenten ist aufgrund der unterschiedlichen COM-Typen und der Verschachtelungstiefe der Strukturen und der Beziehungen der Klassen und Strukturen zueinander im Vergleich zur Metaisierung von Java-Klassen wesentlich umfangreicher und komplexer.

3.3.5 Beispiel

Als Beispiel zur Reflexion in COM wird eine COM-Komponente zum Laden von .NET-Typen vorgestellt. Die COM-Komponente *DotNetTypeLoader* in Listing 3.1 wird zur Reflexion von .NET-Komponenten gemäß Abschnitt 3.4 benötigt. Die Komponente ist in C# implementiert und definiert u.a. die Methode *getType()*, die als Parameter den vollständig qualifizierten Namen eines .NET-Datentyps erhält und ein Objekt vom Typ *System.Type* zurück gibt.

```
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyKeyFile("java2dotnet.snk")]

namespace java2dotnet {

  [ClassInterface(ClassInterfaceType.AutoDual)]
  public class DotNetTypeLoader {

    public DotNetTypeLoader() { }

    // Type.GetType methods
    public Type getType(string typeName) {
      return System.Type.GetType(typeName);
    }

    public Assembly getAssemblyFromFile(string assemblyFile) {
      return Assembly.LoadFrom(assemblyFile);
    }
    ...
  }
}
```

Listing 3.1: Die COM-Komponente DotNetTypeLoader

Aus der gegebenen Klasse wird mit entsprechenden Werkzeugen eine COM-Komponente erzeugt und im System registriert. Die resultierende Komponente definiert u.a. die COM-Klasse *DotNetTypeLoader* und die Schnittstelle *_DotNetTypeLoader*. Listing 3.2 zeigt einen Ausschnitt aus der erzeugten Typbibliothek.

```
library java2dotnet {

  importlib("mscorlib.tlb");
  importlib("stdole2.tlb");

  coclass DotNetTypeLoader {
    [default] interface _DotNetLoader;
    interface _Object;
  };

  interface _DotNetTypeLoader : IDispatch {
    [id(0x60020004), custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, getType)]
    HRESULT getType_2( [in] BSTR typeName,
                       [out, retval] _Type** pRetVal);

    HRESULT getAssemblyFromFile( [in] BSTR assemblyFile,
                                 [out, retval] _Assembly** pRetVal);
    ...
  }
}
```

Listing 3.2: Die Typbibliothek des DotNetTypeLoader

Die Komponente importiert in den Zeilen 3-4 die externen Bibliotheken *mscorlib* und *stdole*. Erstere umfasst die Typen zur Kapselung des .NET-Frameworks in COM. Aus dem Beispiel ist ersichtlich, dass die Methode *getType()* in der Typbibliothek unter dem Namen *getType_2* gespeichert ist. Der Parameter *typeName* ist mit dem Richtungsattribut *[in]* versehen und repräsentiert einen sog. binären String (*BSTR*). Der Rückgabetyp *pRetVal* ist mit dem Richtungsattribut *[out, retval]* gekennzeichnet. Die Methode liefert einen Zeiger auf einen Zeiger vom Typ *_Type* zurück.

Die Metaisierung liefert die in Tabelle 3.2 dargestellten Informationen, wobei es sich nur um einen Auszug der gewonnenen Daten handelt. In der Typbibliothek werden die Rückgabetypen als Parameter mit entsprechendem Richtungsattribut (*retval*) gespeichert. Die Rückgabetypen der dargestellten Methoden sind *_Type* und *_Assembly*. Dabei handelt es sich um Typen aus der importierten .NET-Typbibliothek *mscorlib.tlb* gemäß Listing 3.2, wobei die referenzierten Typen soweit wie nötig analysiert werden (z.B. Extraktion des Namens und des Namensraums).

	Namensraum	Name	Parametertyp	Rückgabetyp
Klasse	java2dotnet	DotNetTypeLoader		
Schnittstelle	java2dotnet	_DotNetTypeLoader		
Methoden		getType_2	BSTR	mscorlib._Type
		getAssemblyFromFile	BSTR	mscorlib._Assembly
		...	...	...

Tabelle 3.2: Meta-Informationen der COM-Komponente DotNetTypeLoader

Mittels der Java2COM-Middleware kann auf auf nahezu sämtliche Metadaten einer COM-Komponente aus Java zugegriffen werden. Bei der Reflexion einer Komponente werden die in der Typbibliothek definierten Typen identifiziert und die zugehörigen Metadaten entsprechend ihres Typs extrahiert.

3.4 Metaisierung von .NET-Komponenten

Analog zu Java bietet .NET resp. eine .NET-Sprache wie beispielsweise C# eine Schnittstelle zur Reflexion von .NET-Komponenten. Mit der .NET Reflection API steht eine komfortable Schnittstelle zur Extraktion von Metadaten von Klassen und Objekten zur Verfügung. Die konzeptionellen und funktionellen Unterschiede zwischen Java und .NET bzgl. Reflexion sind marginal.

3.4.1 Das Metamodell von .NET

Die .NET-Schnittstelle zur Reflexion ist im Paket *System.Reflection* definiert und enthält die in Abb. 3.5 dargestellten Klassen. Analog zu Java und COM bilden diese Klassen ein Metamodell.

In .NET ist jede Klasse von *System.Object* abgeleitet. Diese Klasse ist somit die Superklasse aller Klassen und stellt die Wurzel der Klassenhierarchie dar. Zur Bestimmung des

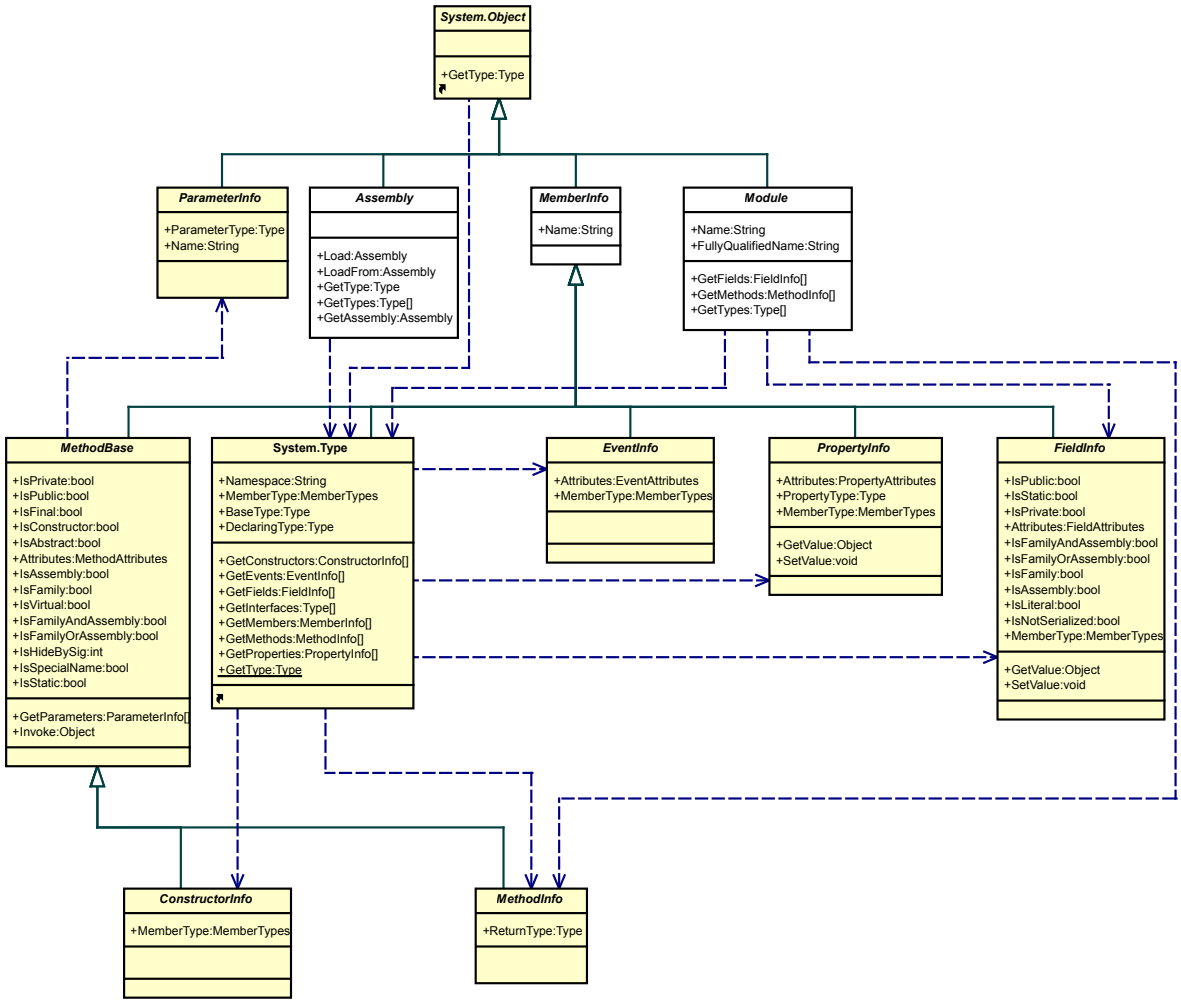


Abbildung 3.5: Das Metamodell von .NET

Datentyps einer Klasse bzw. eines Objekts dient die Methode *GetType()*, die ein Objekt vom Typ System.Type zurück gibt.

Die Klasse *System.Type* ist das .NET-Pendant zu *java.lang.Class* und stellt die notwendigen Methoden zur Extraktion von Metadaten einer Klasse in .NET zur Verfügung. Die Klasse bietet u.a. Methoden zur Extraktion der Konstruktoren (*GetConstructors()*), der Methoden (*GetMethods()*) und der Variablen (*GetFields()*). Die Konstruktoren, Methoden, Parameter und Variablen werden in .NET durch die Klassen *ConstructorInfo*, *MethodInfo*, *ParameterInfo* und *FieldInfo* repräsentiert.

Die Klassen *ConstructorInfo* und *MethodInfo* sind von der abstrakten Klasse *MethodBase* abgeleitet, die ihrerseits von der abstrakten Klasse *MemberInfo* abgeleitet ist. Die Klasse *MemberInfo* erlaubt über die Variable *Name* die Bestimmung des Namens eines Members.

Ein Konstruktor wird durch die Klasse *ConstructorInfo* gekapselt. Diese Klasse definiert Methoden und Properties zur Bestimmung des Namens und der Parameter. Zur Ausführung des Konstruktors, der durch diese Klasse repräsentiert wird, dient die Methode *Invoke()*. Die einzelnen Parameter des Konstruktors sind in der Klasse *ParameterInfo* (s.u.)

gekapselt.
Über der Klasse *MethodInfo* kann auf die Metadaten einer Methode zugegriffen werden. Analog zum Konstruktor bietet diese Klasse Methoden und Properties um den Namen und die Parameter zu extrahieren. Der Datentyp des Rückgabewertes wird mit der Methode *GetType()* bestimmt.

Die Klasse *ParameterInfo* leitet sich direkt von *System.Object* ab und repräsentiert einen Parameter eines Konstruktors oder einer Methode. Mit der Methode *GetType()* kann der Datentyp und mit dem Attribut *Name* der Name eines Parameters bestimmt werden.

Die Metadaten einer Member-Variable werden durch die Klasse *FieldInfo* repräsentiert. Diese Klasse umfasst Properties und Methoden zur Bestimmung des Namens und des Datentyps einer Variable. Der Wert einer Variable kann mit den Methoden *GetValue()* bzw. *SetValue()* gelesen bzw. geschrieben werden.

Im Gegensatz zu Java definiert .NET keine separate Klasse zur Bestimmung des Modifiers. Der Modifier kann über eine Reihe von Properties, z.B. *IsPublic* oder *IsStatic*, des jeweiligen Typs bestimmt werden.

3.4.2 Reflexion von .NET-Komponenten

Aus rein praktischen Gesichtspunkten ist eine Reflexion von .NET-Komponenten in einer von .NET unterstützten Sprache, z.B. in C#, einfach zu realisieren, da die vorgestellte Schnittstelle innerhalb von .NET verwendet und zur Metaisierung benutzt werden kann. Dabei ist allerdings zu bedenken, dass MAX weitestgehend in Java implementiert ist und die bestehende Infrastruktur soweit als möglich (wieder-)verwendet werden soll. Demzufolge stellt sich die Frage, wie die Reflexion von .NET-Komponenten in das aktuelle System zu integrieren ist bzw. welche Funktionalität in welcher Sprache implementiert wird. Es besteht offensichtlich ein Integrationsproblem zwischen Java und .NET. Analog zu COM/DCOM stellt sich das Problem einer Middleware, die es ermöglicht, von Java auf .NET zuzugreifen. Zur Reflexion von .NET-Objekten in Java bestehen folgende Möglichkeiten:

1. Implementierung der gesamten Infrastruktur auf Basis von .NET
2. Einsatz einer Middleware zum Zugriff auf .NET-Komponenten in Java
   (a) Verwendung einer verfügbaren Middleware
   (b) Manuelle Implementierung der Middleware
   (c) Automatische Generierung der Middleware

Eine Implementierung des Gesamtsystems auf Basis von .NET ist sehr aufwendig. Der Aufwand bezieht sich sowohl auf die Implementierung, als auch auf die Wartung der Software. Zur Implementierung der gesamten Infrastruktur in .NET mittels einer von .NET unterstützten Sprache, wie z.B. C#, können nur bedingt Werkzeuge zur automatischen Umsetzung des bestehenden Java-Codes, z.B. J2CS [Azt02], verwendet werden. Eine parallele Entwicklung und Wartung eines Java- und eines .NET-Systems ist nicht

gewollt und ökonomisch nicht vertretbar, da bei jeder Änderung des Systems alle vorhandenen Quellen auf den aktuellen Stand gebracht werden müssen.

Gegen den Einsatz einer Java-Middleware zum Zugriff auf .NET-Komponenten sprechen vor allem deren Verfügbarkeit und deren Kosten. Aktuell[7] wird nur eine Lösung namens JaNET [Int01a] angeboten[8].

Eine manuelle Implementierung der Middleware scheint aus ökonomischen Gesichtspunkten heraus ebenfalls nicht sinnvoll zu sein. Darüber hinaus ist das Ziel dieser Arbeit die Realisierung eines Systems zur Lösung von Integrationsproblemen. Deshalb liegt es nahe, MAX zur Lösung des Integrationsproblems zu benutzen.

Aus den genannten Gründen wird eine automatische Generierung der Middleware favorisiert. Die automatische Generierung der Java/.NET-Middleware mittels Reflexion und Repräsentation ist eine Beispiel-Anwendung für und in MAX, um die Flexibilität des Systems unter Beweis zu stellen. In Abschnitt 5.4.2 wird die automatische Generierung der Middleware zur Integration von .NET in Java vorgestellt.

3.4.3 Metaisierung des .NET-Frameworks

.NET bietet eine COM-Schnittstelle, die, analog zur Metaisierung von COM-Komponenten in Abschnitt 3.3, zur Metaisierung von .NET-Klassen genutzt werden kann. Die Typbibliothek, die mit dem .NET-Framework ausgeliefert wird, enthält Informationen über nahezu sämtliche .NET-Objekte. Das .NET-Framework besteht aus ca. 1400 Klassen, wobei für jede Klasse entsprechend ihres Typs eine oder mehrere Java-Klassen generiert werden. Zur Extraktion dieser Information wird wiederum die bereits beschriebene Java2COM-Middleware eingesetzt. Der Prozess der Metaisierung wird analog zur Metaisierung von COM-Objekten durchgeführt.

Die extrahierte Information wird gemäß Abschnitt 5.4.2 zur automatischen Generierung einer Java/.NET-Middleware benutzt. Die generierte Java-Middleware basiert auf der Java2COM-Bridge und bietet für jede .NET-Klasse eine entsprechende Java-Klasse. Damit besteht die Möglichkeit, aus Java auf .NET-Objekte und -Typen und insbesondere auf deren Metadaten zuzugreifen.

Einige Methoden sind nicht über die COM-Schnittstelle verfügbar. Zu diesen Methoden zählen z.B. die statischen Methoden *GetType(string typeName)* der Klasse *System.Type* zum Laden eines Typs und *LoadFrom(string assemblyFile)* der Klasse *System.Reflection.-Assembly* zum Laden eines Assemblies. Das Laden von .NET-Typen und -Assemblies ist wie oben beschrieben als COM-Komponente in C# gemäß Listing 3.1 implementiert und kann somit analog zu Abschnitt 3.3 metaisiert werden. Zur Verwendung der COM-Komponente in Java werden die entsprechenden Java-Klassen gemäß Abschnitt 5.4.1 wiederum automatisch generiert.

Auf Basis der generierten Klassen kann die Metaisierung von .NET-Komponenten in Java stattfinden, wobei die existierende Infrastruktur (wieder-)verwendet wird. Die Vorteile dieser Vorgehensweise sind offensichtlich. Die automatische Generierung der Klassen er-

[7] Stand Jan. 2004

[8] Die Existenz weiterer verfügbarer Middleware entzieht sich der Kenntnis des Autors.

fordert einen minimalen Zeitaufwand. Eine zu Java analoge Implementierung ist nicht notwendig und aus ökonomischen Gesichtspunkten daher jeder anderen der unter Abschnitt 3.4.2 beschriebenen Möglichkeiten vorzuziehen.

Listing 3.3 zeigt einen Ausschnitt des Java-Experten zur Metaisierung von .NET-Komponenten.

```
public class DotNetReflectionExpert
extends ReflectionExpert
{
  private ReleaseManager rm = null;
  private DotNetTypeLoader dotnetTypeLoader = null;

  public DotNetReflectionExpert() {
    try {
      rm = new ReleaseManager();
      dotnetTypeLoader = new DotNetTypeLoader(rm);
    } catch (Java2ComException j2ce) {
      j2ce.printStackTrace();
    }
  }

  public void reflectAssembly(String assemblyName)
  {
    _Assembly assembly = dotnetTypeLoader.getAssemblyFromFile(assemblyName);
    _Type[] types = assembly.GetTypes();
    for (int n = 0; n < types.length; n++)
      reflectType(types[n]);
  }

  public void reflectType(String typeName)
  {
    _Type type = dotnetTypeLoader.getType_2(typeName);
    reflectType(type);
  }

  public void reflectType(_Type type)
  {
    ...
  }
}
```

Listing 3.3: Beispiel zur Metaisierung von .NET-Komponenten

Der Experte greift dabei über die generierten Klassen auf die .NET-Objekte zu. Im Beispiel wird die COM-Komponente DotNetTypeLoader im Konstruktor instantiiert (Zeilen 5-12). In den Methoden *reflectAssembly()* (Zeile 14) und *reflectType()* (Zeile 22) wird ein .NET-Assembly bzw. -Typ geladen und zur weiteren Verarbeitung an die Methode *reflectType()* (Zeile 28) zur Extraktion der Metadaten übergeben.

Die Extraktion der Metadaten geschieht analog zu Abschnitt 3.2, so dass auf eine ausführliche Beschreibung und ein Beispiel verzichtet wird.

3.5 Metaisierung von CANopen-Komponenten

Gemäß Abschnitt 2.5 besitzt CANopen ebenfalls reflektive Eigenschaften. CANopen bietet die Möglichkeit, die an einen Bus angeschlossenen CANopen-Module zur Laufzeit zu identifizieren und Modul-interne Daten abzufragen. Im Gegensatz zur Reflexion bei Java und .NET werden zusätzlich externe Metadaten zur Reflexion herangezogen.

Die Begriffe Gerät, Knoten, Modul und Komponente werden im Folgenden als äquivalent betrachtet. Alle Begriffe beschreiben eine in sich geschlossene Einheit mit einer definierten Schnittstelle zum Austausch von Daten bzw. zur Ausführung von Aktionen.

3.5.1 CANopen-Metadaten

Die CANopen-Metadaten liegen meist in Form von sog. elektronischen Datenblättern (electronic data sheet, EDS) gemäß der CANopen-Spezifikation [CAN00b] vor. Elektronische Datenblätter sind standardisierte Beschreibungen von CANopen-Geräteprofilen in textueller und maschinenlesbarer Form und werden als Datei mit dem jeweiligen Gerät ausgeliefert. Der Zweck dieser Dateien ist die Verwendung von Werkzeugen, so dass bestimmte Aufgaben, wie z.B. die Konfiguration von CANopen-Geräten, automatisch durchgeführt werden können. Falls kein EDS zu einem Gerät existiert, kann ein Standard-EDS für einen bestimmten Gerätetyp (z.B. E/A-Modul [CAN99], Antrieb [CAN98], usw.) benutzt werden.

Ein EDS enthält im Gegensatz zu einer Geräte-Konfigurationsdatei (Device Configuration File, DCF) nur konfigurationsunabhängige Daten und demzufolge keine aktuellen Werte, z.B. den Modul-Identifier. Ein Datenblatt beschreibt neben der geräte- und herstellerspezifischen Funktionalität die Kommunikationsobjekte, die von einem Gerät unterstützt werden und ist gemäß [CAN00b] in die folgenden funktionalen Blöcke unterteilt:

- **Dateiinformation** beschreibt die Datei selbst (z.B. Datum und Uhrzeit der Erzeugung).
- **Geräteinformation** enthält Angaben über den Hersteller und über den Produkt-Typ.
- **Objektverzeichnis** umfasst ein Menge von Objekteinträgen, die von dem Gerät implementiert sind.

Im Rahmen dieser Arbeit ist hauptsächlich das Objektverzeichnis von Interesse, da dieser Block die Metadaten hinsichtlich der Funktionalität eines Gerätes beinhaltet. Das Objektverzeichnis ist untergliedert in Datentypen (*StandardDataTypes*), in obligatorische Objekte (*MandatoryObjects*), in optionale Objekte (*OptionalObjects*) und herstellerspezifische Objekte (*ManufacturerObjects*). Listing 3.4 zeigt einen Ausschnitt des elektronischen Datenblatts des digitalen Ein-/Ausgabe-Moduls BK5100.

```
[MandatoryObjects]

[1000]
SubNumber=0
ParameterName=device type
ObjectType=0x07
DataType=0x0007
AccessType=ro
LowLimit=
HighLimit=
DefaultValue=0x00030191
PDOMapping=0
```

```
[6200]
SubNumber=9
ParameterName=digital 8 bit output blocks

[6200sub0]
ParameterName=number of output blocks
ObjectType=0x0008
DataType=0x05
AccessType=ro
LowLimit=
HighLimit=
DefaultValue=
PDOMapping=0

[6200sub1]
ParameterName=1.digital 8 bit output block
ObjectType=0x0008
DataType=0x05
AccessType=rw
LowLimit=
HighLimit=
DefaultValue=
PDOMapping=1
```

Listing 3.4: Auszug aus einem elektronischen Datenblatt

Das Beispiel enthält die Sektion der obligatorischen Objekte und zeigt mehrere Objekteinträge. Ein Objekteintrag beginnt mit einem Bezeichner in eckigen Klammern und reicht bis zum nächsten Eintrag. Wie aus dem Beispiel ersichtlich wird, umfasst jeder Eintrag eine Menge von Name/Wert-Paaren. Falls ein Objekt über weitere Objekte, die sog. Subobjekte, verfügt (z.B. [6200]), so wird die Anzahl dieser Subobjekte spezifiziert (SubNumber=9). Ein Subobjekt (z.B. [6200sub0] oder [6200sub1]) ist wiederum mit eckigen Klammern gekennzeichnet und enthält im Bezeichner den entsprechenden Subindex. Aus diesen Informationen werden die Metadaten zur Beschreibung von CANopen-Geräten bzw. deren Schnittstelle extrahiert.

3.5.2 Aufbereitung der externen Metadaten mit EDS2XML

Zur weiteren Verarbeitung eines elektronischen Datenblatts werden EDS-Dateien mittels eines Werkzeugs namens *EDS2XML* automatisch aufbereitet. *EDS2XML* ist in Java implementiert und erfüllt prinzipiell zwei Aufgaben:

- Konvertierung eines EDS nach XML
- Konvertierung bzw. Aufbereitung der Daten eines EDS

Eine Konvertierung nach XML hat den Vorteil, dass ein EDS wesentlich einfacher zu handhaben ist. Die in einem EDS zwar inhärent vorhandene, aber auf den ersten Blick nicht sichtbare hierarchische Datenstruktur, also die Struktur, die z.B. durch Objekte und deren Subobjekte vorgegeben wird, kann adäquat in eine hierarchische XML-Struktur abgebildet werden.

Die Konvertierung von Daten umfasst hauptsächlich die Umsetzung von CANopen-Objekttypen und -Datentypen auf entsprechende symbolische Namen gemäß Tabelle B.2 bzw. B.3 und die Änderungen des Wertes des Attributs *ParameterName*. Als Beispiel

werden, entsprechend der CANopen-Spezifikation, der Objecttyp *0x07* auf *Var* und der Datentyp *0x0007* auf *Unsigned32* abgebildet. Bei der Umsetzung des Namens werden sämtliche Sonderzeichen und Leerzeichen entfernt. Der Parametername 'number of output blocks' wird z.B. auf 'NumberOfOutputBlocks' abgebildet[9]. Der ursprüngliche Wert wird in einem neuen Attribut namens *Description* gespeichert. Diese Transformationen erleichtern zum einen die Lesbarkeit der Metadaten und zum anderen die automatische Generierung von Code. Listing 3.5 zeigt das in Listing 3.4 abgebildete EDS nach der Konvertierung nach XML.

```
<MandatoryObjects>
  <Object Index="0x1000">
    <SubNumber>0</SubNumber>
    <Description>device type</Description>
    <ParameterName>DeviceType</ParameterName>
    <ObjectType>Var</ObjectType>
    <DataType>Unsigned32</DataType>
    <AccessType>ro</AccessType>
    <LowLimit></LowLimit>
    <HighLimit></HighLimit>
    <DefaultValue>0x00030191</DefaultValue>
    <PDOMapping>0</PDOMapping>
  </Object>
  <Object Index="0x6200">
    <SubNumber>9</SubNumber>
    <Description>digital 8 bit output blocks</Description>
    <ParameterName>Digital8BitOutputBlocks</ParameterName>
    <SubObjects>
      <SubObject Index="0x6200" Subindex="0x0">
        <Description>number of outputs blocks</Description>
        <ParameterName>NumberOfOutputBlocks</ParameterName>
        <ObjectType>Array</ObjectType>
        <DataType>Unsigned8</DataType>
        <AccessType>ro</AccessType>
        <LowLimit></LowLimit>
        <HighLimit></HighLimit>
        <DefaultValue></DefaultValue>
        <PDOMapping>0</PDOMapping>
      </SubObject>
      <SubObject Index="0x6200" Subindex="0x1">
        <Description>1.digital 8 bit output block</Description>
        <ParameterName>1Digital8BitOutputBlock</ParameterName>
        <ObjectType>Array</ObjectType>
        <DataType>Unsigned8</DataType>
        <AccessType>rw</AccessType>
        <LowLimit></LowLimit>
        <HighLimit></HighLimit>
        <DefaultValue></DefaultValue>
        <PDOMapping>1</PDOMapping>
      </SubObject>
      ...
    </SubObjects>
  </Object>
  ...
</MandatoryObjects>
```

Listing 3.5: Elektronisches Datenblatt in XML

[9] Aus Gründen der Lesbarkeit wird der Buchstabe, der auf ein Leerzeichen folgt, in einen Großbuchstaben konvertiert.

3.5.3 Das Metamodell von CANopen

CANopen definiert im Gegensatz zu den bisher vorgestellten Systemen kein explizites Metamodell. Auf Basis der CANopen-Spezifikationen und den externen Metadaten kann jedoch ein Metamodell gemäß Abb. 3.6 konstruiert werden. Das implizite Metamodell basiert auf den CANopen-Spezifikationen DS301[CAN00a] und DS306[CAN00b].

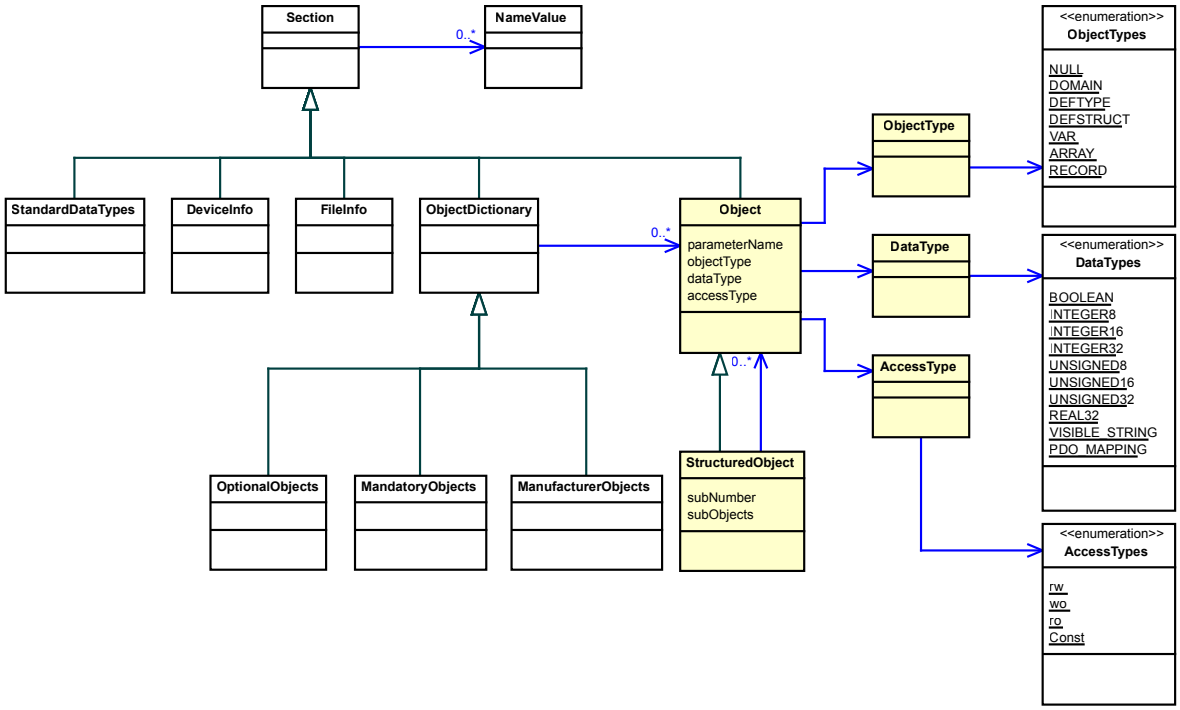


Abbildung 3.6: Das Metamodell von CANopen

Ein EDS enthält mehrere Sektionen (*Section*), die jeweils eine Reihe von zusammengehörenden Einträgen umfassen, wobei Sektionen und Einträge das folgende Format besitzen:

```
[Section name]
key=value
```

Zu den standardisierten Sektionen gehören u.a. *FileInfo*, *DeviceInfo*, *StandardDataTypes* und die Gruppe der Objektverzeichnisse (*MandatoryObjects*, *OptionalObjects*, *ManufacturerObjects*). Da im Kontext dieser Arbeit vor allem die Objektverzeichnisse von Interesse sind, werden die erst genannten Sektionen an dieser Stelle nicht näher betrachtet.

Ein Objektverzeichnis umfasst eine Menge von Objekteinträgen vom Typ *Object*. Falls es sich bei einem Objekt um ein sog. strukturiertes Objekt (*StructuredObject*) handelt, so wird jedes Subobjekt in einer eigenen Sektion beschrieben. Jedes Objekt besitzt u.a. einen Objekttyp, einen Datentyp und einen Zugriffstyp.

Der Objekttyp (*ObjectType*) definiert den Typ eines CANopen-Objekts. Der Typ eines Eintrag kann z.B. ein einfacher Datentyp (*VAR*), ein Feld (*ARRAY*) oder ein Record (*RE-*

CORD) sein. Sämtliche, von CANopen spezifizierten, Objekttypen gemäß [CAN00a] finden sich in Tabelle B.2.

Die Klasse *DataType* enthält einen Index auf einen Datentyp und repräsentiert den Datentyp eines Objektes. Tabelle B.3 enthält eine Auflistung der einfachen und komplexen Datentypen in CANopen[10].

Auf ein Objekt kann entweder nur lesend (*ro*), nur schreibend (*wo*) oder schreibend und lesend (*rw*), zugegriffen werden. Die Zugriffsattribute *rwr* bzw. *rww* beziehen sich auf das Lese-/Schreibrecht bei Prozesseingabe bzw. -ausgabe. *Const* repräsentiert einen konstanten Wert. Sämtliche Zugriffsrechte werden durch die Klasse *AccessType* repräsentiert und sind in *AccessTypes* kodiert.

3.5.4 Die CANopen-Reflection API

Zur Reflexion von CANopen-Komponenten gemäß Abschnitt 3.5.5 wurde im Rahmen dieser Arbeit eine komfortable Java-Schnittstelle realisiert, welche die Details der Reflexion verbirgt. Die API kapselt die CANopen-Kommunikation und den Zugriff auf die externen Metadaten.

Wie aus Abb. 3.7 ersichtlich wird, basiert die *CANopen Reflection API* auf der CANopen-Middleware aus Abschnitt 2.5.3.2 zum Management von CAN- und CANopen-Systemen.

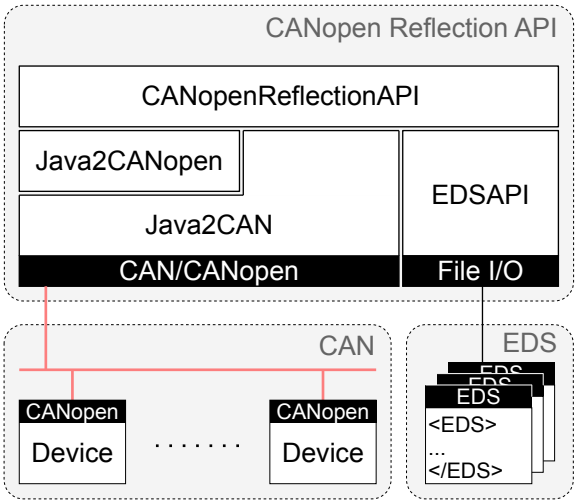


Abbildung 3.7: CANopen Reflection API

Zur Identifikation aller an einen CAN angekoppelten Komponenten dient die Klasse *CANopenChannel*. Diese Klasse bietet u.a. eine Methode namens *getCANopenDevices()*, die eine Liste der verfügbaren CANopen-Komponenten zurückliefert. Dabei werden, wie in Abschnitt 3.5.5.1 erläutert, CANopen-Nachrichten erzeugt und für jede gefundene Komponente ein Objekt vom Typ *CANopenDevice* erzeugt.

Die Klasse *CANopenDevice* repräsentiert eine CANopen-Komponente. Sie bietet Methoden zur Bestimmung des Identifiers (*getDeviceID()*), zur Extraktion des Typs (*getDevi-*

[10] Optional können weitere komplexe Datentypen definiert werden

ceType()), des Namens (*getDeviceName()*) und der von diesem Gerät implementierten Objekte (*getCANopenObjects()*). Zur Realisierung dieser Funktionalität greift diese Klasse parallel sowohl auf das reale Gerät, als auch auf die externen XML-Metadaten zu.

Ein von einer Komponente implementiertes Objekt des Objektverzeichnisses wird durch die Klasse *CANopenObject* repräsentiert. Ein Objekt vom Typ *CANopenObject* bietet zum einen Methoden zum Zugriff auf die statischen externen Metadaten und zum anderen auf den Wert eines Objekteintrags einer Komponente. Zur Extraktion der statischen Informationen stehen u.a. die Methoden *getIndex()*, *getSubindex()*, *getParameterName()*, *getObjectType()*, *getDataType()*, *getAccessType()* und *getDescription()* zur Verfügung. Ein schreibender Zugriff auf die externen Metadaten ist nicht möglich.

Entsprechend des Zugriffs-Typs eines Objektes stehen die Methoden *getValue()* zum Lesen und *setValue()* zum Schreiben des Wertes eines Objekteintrags zur Verfügung. Diese Methoden greifen per Nachrichtenaustausch direkt auf eine reale Komponente zu. Die Methode *getValue()* liefert eine Folge von Bytes zurück, die entsprechend des Datentyps eines Objektes konvertiert werden.

3.5.5 Extraktion der Metadaten

Die Reflexion von CANopen-Komponenten umfasst die Identifikation der vorhandenen Komponenten an einem Bus und die Identifikation und Extraktion der Metadaten der einzelnen Komponenten. Bei der Extraktion der Metadaten werden die identifizierten Komponenten genauer untersucht. Dabei sind die Metadaten der einzelnen Komponenten teilweise in der Komponente selbst oder in einer externen Beschreibung gespeichert. Zur Reflexion einer Komponente werden beide Datenquellen herangezogen.

Die Reflexion von CANopen-Komponenten ist auf Basis von zwei Konzepten realisiert. Zur Extraktion von Metadaten werden zum einen CANopen-Nachrichten gesendet und empfangen und zum anderen externe Metadaten genutzt. Demzufolge werden sowohl geräte-interne, als auch -externe Daten zur Extraktion von Metadaten herangezogen. Interne Daten werden durch den Austausch von Nachrichten ermittelt. Das Versenden von Nachrichten dient primär der Identifikation aller Module, die an einem Bus angeschlossen sind, und der Identifikation der einzelnen Module.

Die Extraktion von CANopen-Metadaten umfasst folglich

- die Extraktion aller Typen gemäß Abschnitt 3.5.5.1,
- die Extraktion des Typs gemäß Abschnitt 3.5.5.2 und
- die Extraktion der Schnittstelle gemäß Abschnitt 3.5.5.3.

3.5.5.1 Identifikation aller CANopen-Komponenten

Da an einem Bus mehrere CANopen-Knoten angeschlossen sein können, werden in einem ersten Schritt alle verfügbaren Komponenten identifiziert. Gemäß der CANopen-Spezifikation unterstützt jedes CANopen-Modul den Parameter *DeviceType*. Dieser Parameter befindet sich im Objektverzeichnis an der Adresse mit dem Index 0x1000 und

dem Sub-Index 0x0. Folglich wird jedes Modul, das an den Bus angekoppelt ist, bei einer entsprechenden Anfrage-Nachricht den Wert an dieser Adresse zurückliefern.

Zum Auffinden aller Module werden entsprechende Anfrage-Nachrichten (SDO) mit unterschiedlichen Identifiern zwischen 1 und 127 (max. Anzahl der Knoten) erzeugt und sequentiell auf den Bus übertragen. Falls ein Modul mit dem angegebenen Identifier vorhanden ist, schickt das Modul eine Antwort-Nachricht zurück. Falls innerhalb einer bestimmten Zeitspanne (SDO Timeout) keine Antwort empfangen wird, so ist kein Modul mit dem angegebenen Identifier vorhanden. Diese Vorgehensweise erlaubt das Auffinden sämtlicher CANopen-Module.

3.5.5.2 Identifikation einer CANopen-Komponente

Bei der Identifikation einer Komponente werden der Typ und der Name eines Gerätes extrahiert, die mittels CANopen-Nachrichten direkt aus dem jeweiligen Gerät ausgelesen werden. Die Antwort-Nachricht auf die gesendete Anfrage-Nachricht nach dem Gerätetyp hat das in Abb. 3.8 dargestellte Format.

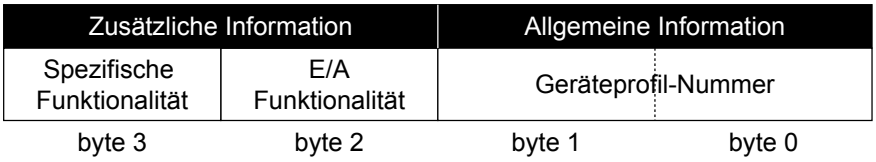


Abbildung 3.8: CANopen Geräteyp-Eintrag

Der Wert des Gerätetyps besteht aus vier Bytes und enthält die Geräteprofil-Nummer (Device Profile Number, DPN) und zusätzliche Informationen. Die DPN spezifiziert ein bestimmtes Geräteprofil gemäß Tabelle 3.3 und bestimmt folglich den Typ eines Moduls.

Geräteprofil-Nummer	Gerätetyp
DS 401	Generische Ein-/Ausgabe-Module
DS 402	Elektrische Antriebe und Antriebssteuerungen
DS 404	Sensoren und Regler
DS 405	Programmierbare Steuerungen
DS ...	...

Tabelle 3.3: Geräteprofil-Nummern und Gerätetypen

Besitzt die DPN beispielsweise den Wert 401 (0x191), so ist das Modul ein generisches Ein-/Ausgabe-Modul gemäß Spezifikation DS-401[CAN99]. Entsprechend der DPN können nun die zusätzlichen Informationen (2. bzw. 3. Byte) eines Moduls interpretiert bzw. extrahiert werden. Die Bedeutung dieser Information wird in dem jeweiligen Geräteprofil definiert und spezifiziert die spezifische Funktionalität einer CANopen-Komponente. Mittels der spezifischen Funktionalität kann beispielsweise bestimmt werden, ob es sich bei einem generischen E/A-Modul um ein digitales oder analoges Eingabe-Modul,

Ausgabe-Modul oder eine Kombination aus beiden handelt. Die E/A-Funktionalität von generischen E/A-Modulen kann z.B. entsprechend Tabelle 3.4 bestimmt werden.

Gerätetyp	Wert
Digital Input	0x1
Digital Output	0x2
Analog Input	0x4
Analog Output	0x8

Tabelle 3.4: Ein-/Ausgabe-Funktionalität von generischen E/A–Modulen

Obwohl es sich bei dem Geräte-Namen um ein optionales Objekt handelt, unterstützen die meisten CANopen-Module den Objekteintrag mit dem Index 0x1008, Subindex 0x0. Der Name ist als ASCII-Zeichenkette gespeichert und wird, falls die Länge der Zeichenkette max. 4 Bytes beträgt, mit einer einzigen Nachricht, ansonsten mit mehreren Nachrichten übertragen. Analog zur Extraktion des Geräteyps wird eine entsprechende Anfrage-Nachricht gesendet und der Wert an dem angegebenen Index ermittelt. Die Antwort liefert den Hersteller-spezifischen Namen eines Gerätes. Das Modul DIOC711 liefert z.B. den Wert DX11 zurück[11].

3.5.5.3 Extraktion der Schnittstelle

Die Schnittstelle eines CANopen-Moduls wird durch dessen Objektverzeichnis bestimmt. Demzufolge bestimmen die Objekte die Operationen, die zur Interaktion mit einer CANopen-Komponente zur Verfügung stehen. Die Operationen eines Gerätes können wiederum als Äquivalent zu den Methoden eines Objektes gesehen werden. Mit den Operationen kann analog zu den Methoden der interne Status einer Komponente geändert werden.

Die geräteinternen Daten, also der Gerätetyp und -namen, erlauben die Referenzierung von externen Metadaten. Der in Abschnitt 3.5.5.2 identifizierte Typ und Name dienen dabei als Verweis auf das externe EDS bzw. dessen XML-Pendant. Eine Voraussetzung hierfür ist, dass aufgrund des Namens ein entsprechender Dateiname konstruiert und die entsprechende Datei geöffnet werden kann. Führt diese Vorgehensweise zu keinem Ergebnis, so wird das für den jeweiligen Gerätetyp standardisierte Geräteprofil benutzt.

Zur Extraktion der Schnittstelle der angeschlossenen CANopen-Modulen werden für jedes gemäß Abschnitt 3.5.5.1 identifizierte CANopen-Modul die entsprechenden externen Metadaten geladen und jeder Objekteintrag (*Object* und *Subobject*) des Objektverzeichnisses in das vorgestellte CANopen-Metamodell abgebildet. Die Reflexion von CANopen-Komponenten wird abschließend an einem Beispiel erörtert.

[11] Der Grund, dass nur 4 Bytes übertragen werden, ist, dass der Wert mit einer einzigen Nachricht (sog. expedited data transfer) übertragen werden kann. Zur Übertragung von größeren Datenmengen werden mehrere Nachrichten ausgetauscht.

3.5.6 Beispiel

Die aktuelle Konfiguration umfasst die in Tabelle 3.5 aufgeführten CANopen-Module. Alle Module gehören zur Klasse der E/A-Module und erfüllen die CANopen-Spezifikation für generische E/A-Module DS-401 [CAN99]. Für jedes Modul stehen externe Metadaten in Form von XML-Dateien, die zuvor aus den entsprechenden EDS-Dateien generiert wurden, zur Verfügung. Die Vorgehensweise für andere Geräteklassen gemäß Abschnitt 2.5.3.1 ist analog.

ID	Name	Typ	Zusätzliche Information
12	DX11	Digitales Ein-/Ausgabe-Modul	0x3
14	BK5100	Digitales Ein-/Ausgabe-Modul	0x3
16	AI11	Analoges Eingabe-Modul	0x4
18	AO11	Analoges Ausgabe-Modul	0x8

Tabelle 3.5: CANopen-Hardware-Konfiguration

Zunächst werden mittels *CANopenChannel.getCANopenDevices()* sämtliche an den Bus angeschlossenen Geräte ermittelt. Diese Methode liefert ein Feld von Objekten vom Typ *CANopenDevice* zurück. Die Methoden *CANopenDevice.getDeviceID()*, *CANopenDevice.getDeviceName()* und *CANopenDevice.getDeviceType()* liefern den Identifier, den Namen und den Typ des entsprechenden Moduls. Die Auswertung des jeweiligen Gerätetyps gemäß dem Geräteprofil für generische E/A-Module ergibt, dass es sich bei den identifizierten Modulen um zwei digitale Ein-/Ausgabe-Module, ein analoges Eingabe- und ein analoges Ausgabe-Modul handelt.

Danach wird für jede Komponente die Schnittstelle bestimmt. Die Methode *CANopenDevice.getCANopenObjects()* liefert sämtliche Objekte des Objektverzeichnisses der einzelnen Module zurück. Intern verarbeitet diese Methode, die zu einem gegebenen Modul gehörige XML-EDS. Auf Grundlage des Namens wird die entsprechende XML-Datei geöffnet. Falls keine Datei mit dem gegebenen Namen existiert, wird das Standard-EDS für den jeweiligen Gerätetyp benutzt.

	Namensraum	Name	Datentyp
Typ	can0	BK5100	
Objekt		Digital8BitOutputBlocks	
Subobjekt		NumberOfOutputBlocks	Unsigned8
		1Digital8BitOutputBlock	Unsigned8
		...	...

Tabelle 3.6: Meta-Informationen der CANopen-Komponente BK5100

Das Ergebnis der Metaisierung von CANopen-Komponenten ist in Tabelle 3.6 dargestellt, wobei die Tabelle eine vereinfachte Darstellung der Parameternamen enthält. Im Hinblick auf die automatische Code-Generierung ist eine eindeutige Namensgebung zwingend notwendig. Da es möglich ist, dass gleiche Parameternamen innerhalb eines EDS vorkommen, wird zu deren Unterscheidung der hierarchische Name eines Objektes benutzt. D.h., ein Subobjekt wie bspw. *NumberOfOutputBlocks*, erhält zusätzlich das Präfix

des Objektnamens, also z.B. *Digital8BitOutputBlocks*. Damit lautet der korrekte Namen im Beispiel *Digital8BitOutputBlocksNumberOfOutputBlocks*. Sollte dieser Namen immer noch nicht eindeutig sein, so wird eine fortlaufende Nummer an den Namen angehängt.

3.6 Zusammenfassung

Das Konzept der Reflexion bzw. die Nutzung von reflektiven Eigenschaften bieten die notwendigen Mechanismen zur Metaisierung, also zur Abbildung von Komponenten in ein entsprechendes Metamodell. Die reflektiven Eigenschaften der vorgestellten Systeme bzw. Sprachen erlauben dabei eine automatische Extraktion der Schnittstelle von Komponenten.

Jede der vorgestellten Programmiersprachen bzw. Technologien unterstützt entweder explizit oder implizit ein spezielles Metamodell, das auf die jeweilige Technologie angepasst ist. Während sich die Reflexion in Java und .NET aufgrund einer entsprechenden API zur Reflexion bzw. der Verfügbarkeit eines entsprechenden Metamodells sehr einfach gestaltet, ist die Reflexion von COM- und CANopen-Komponenten wesentlich umfangreicher.

In Java und .NET werden Komponenten direkt in ein entsprechendes Metamodell abgebildet. Die Metaisierung von .NET-Komponenten ist ebenfalls in Java realisiert und basiert auf einer automatisch generierten Java/.NET-Middleware gemäß Abschnitt 5.4.2. Zur Metaisierung von COM-Komponenten in Java wird eine entsprechende Java/COM-Middleware benutzt, welche die Metadaten einer COM-Komponente in Java zur Verfügung stellt. Bei der Metaisierung von CANopen-Komponenten werden die an einem Bus verfügbaren Komponenten und deren Typen dynamisch bestimmt und die zugehörigen Metadaten aus externen Gerätespezifikationen extrahiert.

Bei allen Systemen wird eine objektorientierte Abstraktion von Komponenten angestrebt, d.h., dass sämtliche identifizierbare Typen auf Objekte mit einer entsprechenden objektorientierten Schnittstelle abgebildet werden. Dies legt nahe, von den einzelnen speziellen Metamodellen zu abstrahieren und ein einziges allgemeines Metamodell für alle Technologien zu verwenden. Das allgemeine Metamodell wird im folgenden Kapitel vorgestellt.

4

MAX und MAXML

Aus Kapitel 3 ist ersichtlich, dass jedes System bzw. jede Technologie ein eigenes Metamodell definiert. Im Hinblick auf die automatische Generierung von Softwarekomponenten und der Verwendung einer einzigen Infrastruktur zur Verwaltung von Metamodellen, wird in diesem Kapitel ein allgemeines Metamodell entwickelt. Zur plattform- und sprachunabhängigen Speicherung dieses Modells wird eine Serialisierung desselben auf Basis von XML vorgestellt.

4.1 Ein allgemeines Metamodell

Entsprechend den Anforderungen bzgl. Softwarewiederverwendung gemäß Abschnitt 2.2.4 wird von speziellen Eigenschaften abstrahiert, indem systemspezifische Details der einzelnen Sprachen bzw. Systeme verborgen und allgemeine Eigenschaften hervorgehoben werden.

4.1.1 Das Metamodell von MAX

Das allgemeine Metamodell in MAX abstrahiert von speziellen Eigenschaften einer Programmiersprache bzw. Technologie und stellt eine Verallgemeinerung sämtlicher vorgestellter Metamodelle dar. Dabei liegt der Schwerpunkt auf einer objektorientierten Abstraktion von abstrakten Datentypen. Prinzipiell können die folgenden Entitäten identifiziert werden:

- Typ
- Feld
- Operation
- Operand
- Ausnahme

- Benachrichtigung

Bei den identifizierten Entitäten wird soweit wie möglich auf sprachspezifische Bezeichner verzichtet, um Assoziationen zu einer bestimmten Programmiersprache zu vermeiden. Deshalb wird z.B. anstatt Konstruktor, Methode oder Funktion der Begriff Operation verwendet. Dennoch gilt, dass das Metamodell einen objektorientierten Ansatz verfolgt und daher sämtliche Typen bzw. Komponenten in ein objektorientiertes Modell abgebildet werden. Das Klassendiagramm des Metamodells von MAX ist in Abb. 4.1 dargestellt.

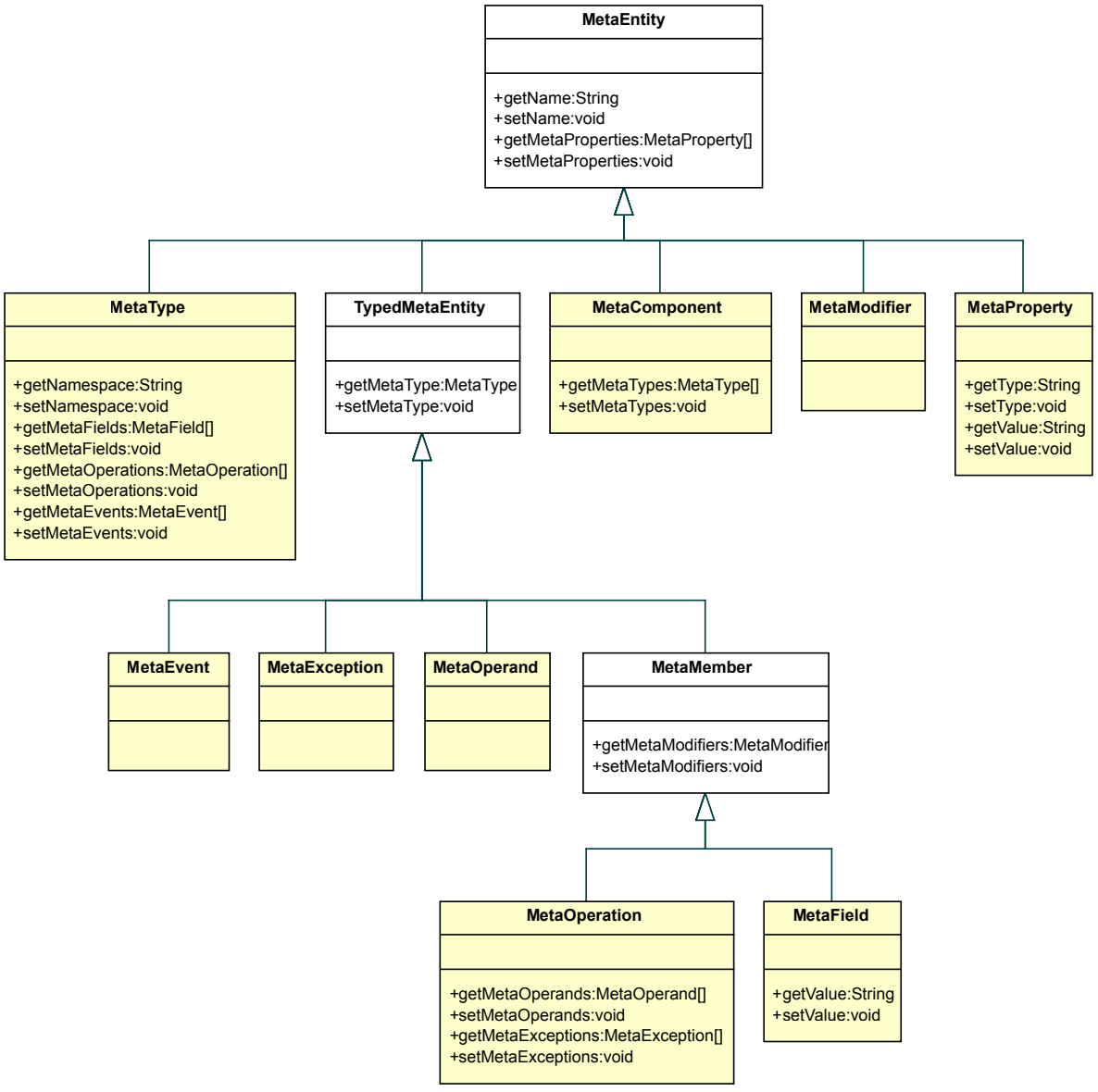


Abbildung 4.1: Das Metamodell von MAX

Die Klasse *MetaEntity* stellt die Wurzel des Metamodells dar. Sämtliche Klassen des Metamodells sind von *MetaEntity* abgeleitet. Eine *MetaEntity* besitzt einen Namen und optional eine oder mehrere Eigenschaften vom Typ *MetaProperty*. Die Klasse bietet folgende Methoden:

- *getName()* bzw. *setName()* dienen zum Lesen bzw. Setzen des Namens.
- *getMetaProperties()* bzw. *setMetaProperties()* dienen zur Verwaltung der Eigenschaften.

Die Klasse *MetaProperty* definiert ein Tripel, bestehend aus Name, Typ und Wert. Der Name identifiziert eine Eigenschaft, der Typ und Wert beinhalten den Datentyp und den Wert einer Eigenschaft, wobei die einzelnen Werte jeweils durch eine Zeichenkette repräsentiert werden. Der Klasse *MetaProperty* kommt eine herausragende Wertigkeit in dem Metamodell zu. Sämtliche sprach- oder technologiespezifischen Eigenschaften bzw. Attribute werden durch diese Klasse gekapselt. Die Klasse wird zumeist bei der Konkretisierung bzw. Spezialisierung einer bestimmten Technologie benötigt. Zum Zugriff auf die genannten Attribute dienen die Methoden *getName()* und *setName()*, *getType()* bzw. *setType()* und *getValue()* bzw. *setValue()*.

Die Klasse *MetaType* repräsentiert entweder einen einfachen oder einen komplexen Datentyp. Ein einfacher Typ ist die Repräsentation eines einfachen Datentyps der jeweiligen Programmiersprache bzw. Technologie. Ein abstrakter Datentyp, eine Klasse oder eine Hardware- bzw. Softwarekomponente wird durch einen komplexen Typ repräsentiert. Ein komplexer Datentyp besitzt einen Namen und einen Namensraum, wobei innerhalb eines Namensraums ein Name eindeutig sein muss. Ein komplexer Typ kann eines oder mehrere Objekte vom Typ *MetaField*, *MetaOperation* und *MetaEvent* umfassen. Die Klasse *MetaType* besitzt die folgenden Methoden:

- *getNamespace()* liefert den Namensraum, in dem ein Typ definiert ist, zurück.
- *getMetaFields()* bzw. *setMetaFields()* dienen zur Verwaltung der Felder eines abstrakten Typs.
- *getMetaOperations()* bzw. *setMetaOperations()* dienen zur Verwaltung der Operationen.
- *getMetaEvents()*, *setMetaEvents()* dienen zur Verwaltung der Benachrichtigungen.

Die Klasse *MetaMember* beschreibt ein Feld, d.h. eine Variable oder Konstante, eine Operation oder eine Benachrichtigung eines komplexen Typs. Sie ist die Basisklasse von *MetaField*, *MetaOperation* und *MetaEvent*. Ein Member besitzt einen Typ und optional einen oder mehrere Modifizierer. Zur Verwaltung der Modifizierer dienen die Methoden *getMetaModifiers()* bzw. *setMetaModifiers()*.

Ein *MetaField* kapselt eine Variable oder eine Konstante eines komplexen Typs und besitzt zumindest einen Typ und durch die Vererbung von *MetaEntity* einen Namen, der die Variable oder Konstante eindeutig identifiziert. Falls ein *MetaField* eine Konstante beschreibt, so besitzt ein *MetaField* zusätzlich einen vordefinierten Wert, der mit den Methoden *getValue()* gelesen und *setValue()* gesetzt werden kann. Die Klasse *MetaField* ist von der Klasse *MetaMember* abgeleitet und besitzt somit die von *MetaMember* deklarierten Methoden.

Ein Konstruktor, eine Methode oder eine Prozedur wird durch die Klasse *MetaOperation* repräsentiert. Eine Operation besitzt eine Signatur, einen Namen und spezifiziert optional ein oder mehrere Ausnahmen. Die Signatur umfasst den Rückgabetyp einer Operation und optional eine Menge von Operanden. Ein Konstruktor einer Klasse ist eine spezielle Operation, deren Rückgabetyp den Wert `null` besitzt. Die Klasse *MetaOperation* ist von *MetaMember* abgeleitet und bietet, zusätzlich zu den von *MetaMember* definierten Methoden, folgende Methoden:

- getOperands() bzw. setOperands() dient zur Verwaltung der Operanden.
- getExceptions() bzw. setExceptions() dient zur Verwaltung der Ausnahmen.

Ein Operand einer Operation oder eines Konstruktors wird durch die Klasse *MetaOperand* repräsentiert. Ein *MetaOperand* besitzt einen Namen und einen Typ. Sprachspezifische Konstrukte wie z.B. die IDL-Attribute von COM werden nicht direkt durch einen Operanden, sondern durch zusätzliche Eigenschaften definiert. Der Typ eines Operanden kann mit den Methoden *getMetaType()* bzw. *setMetaType()* verwaltet werden.

Eine Benachrichtigung einer Komponente bzw. eines Objektes, also ein Event, wird durch die Klasse *MetaEvent* repräsentiert. Ein *MetaEvent* besitzt einen Namen und einen Typ. Benachrichtigungen werden im Rahmen dieser Arbeit nur bedingt verarbeitet, werden aber im Hinblick auf zukünftige Erweiterungen dennoch berücksichtigt.

Ein Modifizierer wird durch die Klasse *MetaModifier* repräsentiert. Ein Modifier umfasst durch die Vererbungsbeziehung einen Namen und eine Menge von Eigenschaften.

Als Container für Typen dient die Klasse *MetaComponent*. Eine Instanz dieser Klasse umfasst ein oder mehrere Objekte vom Typ *MetaType*. Die Klasse ist vor allem zur logischen Zusammenfassung von Typen gedacht. So werden beispielsweise die Typen, die in einem Java-Paket oder von einer COM-Komponente definiert sind, in einer *MetaComponent* zusammengefasst. Zusätzliche Informationen können der Klasse mittels Objekten vom Typ *MetaProperty* hinzugefügt werden.

Zur Verwaltung, z.B. zum Hinzufügen bzw. Löschen einzelner Properties, Felder, Operationen, Benachrichtigungen, Modifizierer und Typen, existieren weitere Methoden, die als Parameter einen Index oder ein Objekt vom jeweiligen Typ erwarten. Beispielsweise kann mit *add-* bzw. *removeMetaOperation()* eine einzelne Operation hinzugefügt bzw. gelöscht werden. Aus Gründen der Übersichtlichkeit wurde auf die Darstellung dieser Methoden im Klassendiagramm in Abb. 4.1 verzichtet.

4.1.2 Abbildung der Metamodelle in das allgemeine Metamodell

In den folgenden Abschnitten werden die Abbildungen der einzelnen Metamodelle aus Kapitel 3 in das allgemeine Metamodell betrachtet.

4.1.2.1 Abbildung des Java-Metamodells

Bei der Abbildung des Java-Metamodells gemäß Abschnitt 3.2.1 in das allgemeine Metamodell werden (abstrakte) Klassen und Schnittstellen durch ein Objekt vom Typ *MetaTy-*

pe repräsentiert. Mehrere logisch zusammengehörende (abstrakte) Klassen bzw. Schnittstellen, z.B. ein Paket, werden in einer Instanz vom Typ *MetaComponent* zusammengefasst.

Ein Konstruktor oder eine Methode mit zugehörigem Rückgabetyp und zugehörigen Parametern wird auf eine Instanz vom Typ *MetaOperation* mit entsprechender Signatur abgebildet. Falls es sich bei einem Rückgabetyp oder einem Operanden um ein ein- oder mehrdimensionales Feld handelt, so wird dem *MetaType* ein entsprechendes *MetaProperty* mit der Bezeichnung `ReferenceType` hinzugefügt. Dieses *MetaProperty* erhält wiederum ein *MetaProperty*, das die Dimension des Feldes definiert.

Die Abbildung von Java nach MAX ist vergleichsweise einfach, da in beiden Fällen ein objektorientiertes Metamodell mit zueinander ähnlichen Klassen vorliegt.

4.1.2.2 Abbildung des COM-Metamodells

Zur Abbildung des COM-Metamodells gemäß Abschnitt 3.3.2 werden sämtliche abstrakte Datentypen auf die Klasse *MetaType* abgebildet. Zu den abstrakten Datentypen zählen die durch *TYPEKIND* spezifizierten COM-Typen. Der eigentliche Datentyp wird im Hinblick auf die Spezialisierung durch eine Klasse vom Typ *MetaProperty* gekapselt. Der Datentyp *TKIND_DISPATCH* wird z.B. durch das Tripel `[name="TYPEKIND", type="int", value="TKIND_DISPATCH"]` repräsentiert.

Entsprechend des Datentyps besitzt ein Typ eine Menge von Operationen und Feldern. Diese werden auf Objekte des Typs *MetaOperation* und *MetaField* abgebildet. Die Richtungsattribute von Parametern einer Operation werden in einem zugehörigen *MetaProperty* gespeichert. Falls es sich bei einem Rückgabetyp einer Operation um eine Referenz auf einen abstrakten Datentyp handelt, so wird der Name und der Namensraum dieses Typs aufgelöst und durch ein Objekt vom Typ *MetaType* gekapselt.

Die ursprünglichen COM-Datentypen werden als Objekte vom Typ *MetaProperty* mit dem Namen "IDLType" gespeichert. Bei einfachen COM-Datentypen sind dies die in Tabelle B.1 aufgeführten Datentypen. Bei komplexen, in COM auch als benutzerdefiniert bezeichneten Datentypen wird der entsprechende Datentyp nach Name und Namensraum aufgelöst. Das COM-Objekt *Word.Application* bietet beispielsweise die Methode *Documents()*, die eine Referenz auf einen benutzerdefinierten Typ namens *Word.Documents* zurückliefert. Dieser Typ wird durch das Tripel `[name="IDLType", type="string", value="VT_PTR+VT_USERDEFINED(3004cf4):Documents"]` repräsentiert. Falls eine COM-Klasse eine oder mehrere Schnittstellen implementiert, wird jede implementierte Schnittstelle durch ein *MetaProperty* repräsentiert.

Durch die Verwendung von Eigenschaften wird bei der Abbildung des COM-Metamodells von sämtlichen COM-spezifischen Details abstrahiert, so dass nur noch die Schnittstelle einer COM-Komponente in Form von abstrakten Datentypen mit entsprechenden Operationen explizit sichtbar ist. Durch die beschriebene Vorgehensweise ist die Abbildung des COM-Metamodells wesentlich komplexer, als dies in Java bzw. .NET der Fall ist.

4.1.2.3 Abbildung des .NET-Metamodells

Die Abbildung von .NET-Klassen und -Schnittstellen ist analog zu Abschnitt 4.1.2.1 aufgrund der Ähnlichkeit der Modelle sehr einfach. Schnittstellen und (abstrakte) Klassen werden durch die Klasse *MetaType* repräsentiert. Felder werden auf Instanzen der Klasse *MetaField*, Konstruktoren und Methoden auf die Klasse *MetaOperation* und deren Parameter auf *MetaOperand* abgebildet. Der Rückgabetyp einer Methode wird wiederum durch die Klasse *MetaType* repräsentiert. Logisch zusammengehörige Typen (Assemblies) werden analog zu Java-Paketen in einem Objekt vom Typ *MetaComponent* zusammengefasst.

4.1.2.4 Abbildung des CANopen-Metamodells

Im Folgenden werden die Begriffe Gerät, Knoten, Modul und Komponente, analog zu Abschnitt 3.5, als äquivalent betrachtet. Hintergrund ist, dass eine CANopen-Komponente prinzipiell als abstrakter Datentyp mit entsprechenden Operationen betrachtet werden kann. Demzufolge wird eine CANopen-Komponente auf ein Objekt vom Typ *MetaType* abgebildet.

Die Operationen, die für ein Modul zur Verfügung stehen, ergeben sich aus dem zugehörigen elektronischen Datenblatt, also der Menge der Objekte des Objektverzeichnisses. Dabei werden für jedes Objekt entsprechend des Zugriffsattributs eine oder mehrere Operationen mit dem entsprechenden Parameternamen zzgl. des Schlüsselwortes *get* oder *set* konstruiert. Der Rückgabetyp einer Operation entspricht dem Datentyp des entsprechenden Objekteintrags.

Für jedes Objekt im Objektverzeichnis wird entsprechend des Zugriffstyps (r, w, rw) eine oder mehrere Operationen bzw. Objekte vom Typ *MetaOperation* erzeugt. Abhängig vom Zugriffstyp erhalten die Operationsnamen dabei das Präfix *get* oder *set*. Der Parameter- bzw. Rückgabetyp wird aus dem Attribut *DataType* gemäß 3.5.1 extrahiert. Aus Gründen der Eindeutigkeit besteht der Operationsname analog zu Abschnitt 3.5.6 aus dem Namen des Objektes und dem des Subobjektes.

Das digitale Ein-/Ausgabemodul DIOC711 besitzt beispielsweise einen Objekteintrag namens `WriteOutputByte` mit dem Subindex `NrOutputModules` und dem Subindex `Output1`, wobei beide Einträge den CANopen-Datentyp `Unsigned8` besitzen. Der erste Eintrag kann nur gelesen und der zweite nur geschrieben werden. Für diese Einträge werden eine Operation namens *getWriteOutputByteNrOutputModules()* mit einem Rückgabetyp vom Typ `Unsigned8` und eine Operation *setWriteOutputByteOutput1()* mit einem Parameter vom Typ `Unsigned8` konstruiert. Die Operationen können in einer prozeduralen bzw. objektorientierten Programmiersprache wie folgt dargestellt werden:

```
Unsigned8 getWriteOutputByteNrOutputModules();
void setWriteOutputByteOutput1(Unsigned8 param);
```

Falls bei der Konstruktion von Operationsnamen ein Namenskonflikt entsteht, d.h. es existieren mehrere Operationen mit demselben Namen und derselben Signatur, so erhalten die Operationsnamen zusätzlich eine fortlaufende Nummer.

Die Werte für den Index und Subindex eines Objekteintrags werden jeweils durch die Klasse *MetaProperty* repräsentiert. Der oben genannte Eintrag `Output1` umfasst die folgenden Tripel für Index bzw. Subindex:

- `[name="index", type="int", value="0x6200"]`
- `[name="subindex", type="int", value="0x1"]`

Die Beschreibung eines Objekteintrags wird ebenfalls als Objekt vom Typ *MetaProperty* in einem Tripel `[name="description", type="string", value="number of output blocks"]` gespeichert.

Zur logischen Gruppierung von mehreren, an einem Bus identifzierten CANopen-Komponenten dient die Klasse *MetaComponent*.

4.1.3 Bemerkung

Das vorgestellte allgemeine Metamodell bietet, mit Ausnahme von CANopen, keinen höheren Abstraktionsgrad, sondern eher eine Verallgemeinerung der beschriebenen Metamodelle mit einer schnittstellenbasierten Abstraktion von Soft- und Hardwarekomponenten. Die kognitive Distanz zwischen der jeweiligen Technologie und dem Metamodell ist aufgrund der objektorientierten Modellierung von Komponenten minimal.

Das Metamodell von MAX bietet eine Vereinheitlichung der Metamodelle von Java, COM, .NET und CANopen. Die CANopen-Komponenten werden analog zu abstrakten Datentypen auf komplexe Typen mit entsprechenden Operationen abgebildet. Dabei wird ebenfalls eine objektorientierte Sichtweise auf die Komponenten realisiert.

4.2 Serialisierung des allgemeinen Metamodells nach XML

Mit dem allgemeinen Metamodell liegt eine temporäre, im Speicher gehaltene Abstraktion von Soft- und Hardwarekomponenten vor. Im Hinblick auf die Generierung von Softwarekomponenten und Dokumentation gemäß Kapitel 5 wird im Folgenden eine Serialisierung des Modells auf Basis von XML vorgestellt.

4.2.1 Ziele

Die Serialisierung des Metamodells verfolgt die Ziele Persistenz, Übertragung und Wiederverwendung. Bereits metaisierte Komponenten sollen dauerhaft gespeichert werden, so dass sie über ein Netzwerk übertragen werden können. Zudem erlaubt die Speicherung die strukturelle Suche und Analyse von metaisierten Komponenten. Das Ziel einer persistenten Speicherung ist, dass die Beschreibungen möglichst oft und zu unterschiedlichen Zwecken wiederverwendet werden können.

4.2.2 Anforderungen

Im Rahmen dieser Arbeit werden an die serialisierten Beschreibungen die folgenden Anforderungen gestellt:

- Plattformunabhängigkeit: Die Plattformunabhängigkeit betrifft weniger die Beschreibungen selbst als vielmehr die Werkzeuge und APIs, die zur Verarbeitung der Beschreibung notwendig sind.
- Sprachunabhängigkeit: Die Beschreibung von Komponenten bzw. deren Schnittstellen sollte möglichst sprachunabhängig sein, so dass die gleichen Werkzeuge zur Verarbeitung eingesetzt werden können.
- Generalität: Die verwendete Beschreibung sollte möglichst unabhängig von einer bestimmten Technologie sein, so dass sie für ein breites Spektrum an Anwendungen Gültigkeit hat.
- Effizienz: Die Beschreibung sollte möglichst schnell verarbeitet werden können.
- Einfachheit: Die Beschreibung sollte einfach lesbar sein, so dass eine geringe kognitive Distanz besteht.

XML ist aus Gründen der Einfachheit, Flexibilität und breiten Unterstützung bzgl. den zur Verfügung stehenden Werkzeugen als Format zur Beschreibung und zum Austausch von Metamodellen prädestiniert. Für (fast) jede Plattform existieren entsprechende Tools und APIs zur Verarbeitung von XML-Dokumenten.

4.2.3 Serialisierung des Metamodells

Das allgemeine Metamodell wird als Menge von Objekten im Speicher gehalten. Zur Serialisierung dieses Modells werden diese Objekte bzw. deren Attribute in ein entsprechendes XML-Dokument abgebildet, wobei sich die Abbildung im wesentlichen direkt aus dem vorgestellten Metamodell herleiten lässt.

4.2.4 MAXML

Die MAX Markup Language, kurz MAXML, ist eine Markup-Sprache zur Beschreibung von Metamodellen auf Basis von XML. Die MAXML-DTD definiert die zulässigen Elemente und Attribute von MAXML.

4.2.4.1 MAXML-DTD

Da sich jedes Metaobjekt entweder direkt oder indirekt von *MetaEntity* ableitet, besitzt jedes Element einen Namen und eine Menge von Elementen vom Typ *MetaProperty*. Die zu einer *MetaEntity* gehörenden Eigenschaften werden durch ein Element namens *MetaProperties* gekapselt. Das Element *MetaProperties* in Listing 4.1 umfasst dabei eines oder

mehrere Elemente vom Typ *MetaProperty*. Die zusätzliche Ebene, die durch das Element *MetaProperties* entsteht, ist zwar nicht notwendig, trägt aber zur Übersichtlichkeit des XML-Dokuments bei[1]. Das Element *MetaProperty* besitzt die Attribute *name*, *type* und *value*. Optional enthält eine *MetaProperty* ein Element vom Typ *MetaProperties*. Durch dieses Konstrukt kann ein hierarchischer Baum von Eigenschaften konstruiert werden.

```
<!ELEMENT MetaProperties (MetaProperty+)>
<!ELEMENT MetaProperty (MetaProperties?)>
<!ATTLIST MetaProperty
  name CDATA #REQUIRED
  type CDATA #REQUIRED
  value CDATA #REQUIRED
>
```

Listing 4.1: Die XML-Elemente *MetaProperties* und *MetaProperty*

Eine Instanz vom Typ *MetaType* wird auf das in Listing 4.2 dargestellte XML-Element abgebildet. Ein *MetaType* besitzt einen Namen und optional einen Namensraum, eine Menge von Eigenschaften (*MetaProperties*), Variablen (*MetaFields*), Operationen (*MetaOperations*) und Beachrichtigungen (*MetaEvents*).

```
<!ELEMENT MetaType (MetaProperties?, MetaFields?, MetaOperations?,
                    MetaEvents?)>
<!ATTLIST MetaType
  name CDATA #REQUIRED
  namespace CDATA #IMPLIED
>
```

Listing 4.2: Das XML-Element *MetaType*

Das Element *MetaFields* kapselt eines oder mehrere Felder vom Type *MetaField* gemäß Listing 4.3. Letzteres besitzt einen Namen, einen Typ und optional eine Menge von Eigenschaften und einen Wert.

```
<!ELEMENT MetaFields (MetaField+)>
<!ELEMENT MetaField (MetaType, MetaProperties?, MetaValue?)>
<!ATTLIST MetaField
  name CDATA #REQUIRED
>
```

Listing 4.3: Die XML-Elemente *MetaFields* und *MetaField*

Das Element *MetaOperations* umfasst eines oder mehrere Elemente vom Typ *MetaOperation*. Nach Listing 4.4 umfasst eine Operation in XML einen Namen, einen (Rückgabe-) Typ und optional eine Menge von Argumenten, Ausnahmen und Eigenschaften.

```
<!ELEMENT MetaOperations (MetaOperation+)>
<!ELEMENT MetaOperation (MetaType, MetaProperties?, MetaOperands?,
                         MetaExceptions?)>
<!ATTLIST MetaOperation
  name CDATA #REQUIRED
>
```

Listing 4.4: Die XML-Elemente *MetaOperations* und *MetaOperation*

Falls eine Operation eine oder mehrere Argumente definiert, so werden diese jeweils auf das Element *MetaOperand* abgebildet. Ein Operand besitzt gemäß Listing 4.5 einen Namen, einen Typ und evtl. eine Menge von Eigenschaften.

[1] Gleiches gilt für die Elemente *MetaOperations*, *MetaOperands*, *MetaEvents* und *MetaExceptions*

```
<!ELEMENT MetaOperands (MetaOperand+)>
<!ELEMENT MetaOperand (MetaType, MetaProperties?)>
<!ATTLIST MetaOperand
  name CDATA #REQUIRED
>
```

Listing 4.5: Die XML-Elemente *MetaOperands* und *MetaOperand*

Definiert eine Operation eine oder mehrere Ausnahmen, so werden diese durch das Element *MetaException* repräsentiert. Eine Ausnahme besitzt nach Listing 4.6 zumindest einen Namen und einen Typ.

```
<!ELEMENT MetaExceptions (MetaException+)>
<!ELEMENT MetaException (MetaType, MetaProperties?)>
<!ATTLIST MetaException
  name CDATA #REQUIRED
>
```

Listing 4.6: Die XML-Elemente *MetaExceptions* und *MetaException*

Definiert ein Typ eine oder mehrere Benachrichtigungen, so werden diese durch das Element *MetaEvent* gemäß Listing 4.7 repräsentiert. Eine Benachrichtigung besitzt einen Typ und einen Namen.

```
<!ELEMENT MetaEvents (MetaEvent+)>
<!ELEMENT MetaEvent (MetaType, MetaProperties?)>
<!ATTLIST MetaEvent
  name CDATA #REQUIRED
>
```

Listing 4.7: Die XML-Elemente *MetaEvents* und *MetaEvent*

Das Element *MetaComponent* stellt die Wurzel eines XML-Dokumentes dar. Die Wurzel umfasst die optionalen Elemente *MetaProperties* und *MetaTypes*.

Eine Komponente kann eine Vielzahl von Typen umfassen. Eine Abbildung dieser Typen in einem einzigen XML-Dokument ist deshalb sehr unübersichtlich. Aus diesem Grund kann die Repräsentation optional in mehrere Teildokumente untergliedert werden. Jede Instanz vom Typ *MetaType* wird dabei durch ein XML-Dokument repräsentiert. Die einzelnen Dokumente werden durch XInclude in ein Hauptdokument integriert, so dass das Element *MetaTypes* um ein zusätzliche Element gemäß Listing 4.8 für jeden deklarierten Typ erweitert wird.

```
<!ELEMENT MetaTypes ((xi:include+) | (MetaType+))>
<!ELEMENT xi:include EMPTY>
<!ATTLIST xi:include
  href CDATA #REQUIRED
>
```

Listing 4.8: Referenzierung von *MetaTypes* mittels XInclude

Die vollständige MAX-DTD kann Abschnitt A.1 entnommen werden.

4.2.4.2 Repräsentation von Datentypen

Einfache Datentypen der jeweiligen Programmiersprache werden im Hinblick auf zukünftige Erweiterungen (siehe dazu Abschnitt 7.2) auf die in Abb. B.1 dargestellten

XML Schema-Datentypen abgebildet. Die ursprünglichen Datentypen werden zusätzlich in einem *MetaProperty*-Element gespeichert, da sie bei der Spezialisierung bzw. Konkretisierung, also zur Generierung von Komponenten, benötigt werden. Beim Attribut *type* des *MetaProperty*-Elements finden ebenfalls XML Schema-Datentypen Verwendung.

4.2.5 Beispiel

Die Serialisierung eines Metamodells in MAX soll am Beispiel einer COM-Komponente verdeutlicht werden. Das Beispiel zeigt die Serialisierung der metaisierten COM-Komponente *DotNetTypeLoader* zum Laden von .NET-Typen gemäß Abschnitt 3.3.5.

Die COM-Komponente *java2dotnet.DotNetTypeLoader* besteht gemäß der Typbibliothek in Listing 3.2 aus einer COM-Klasse (*DotNetTypeLoader*) und einer COM-Schnittstelle (*_DotNetTypeLoader*), die von der COM-Klasse implementiert wird.

Das Basis-Dokument in Listing 4.9 referenziert per XInclude die in der Typbibliothek definierte Klasse und Dispatch-Schnittstelle. Eine Referenz auf ein externes XML-Dokument wird bei der XInclude-Anweisung in dem Attribut `href` übergeben.

```
<MetaComponent name="java2dotnet" xmlns:xi="http://www.w3.org/2001/XInclude">
  <MetaProperties>
    ...
  </MetaProperties>
  <MetaTypes>
    <xi:include href="java2dotnet/DotNetTypeLoader.xml"/>
    <xi:include href="java2dotnet/_DotNetTypeLoader.xml"/>
  </MetaTypes>
</MetaComponent>
```

Listing 4.9: Die COM-Komponente java2dotnet

Die COM-Klasse *DotNetTypeLoader* ist gemäß Listing 4.10 vom Typ *TKIND_COCLASS* und implementiert die Dispatch-Schnittstelle *_DotNetTypeLoader*.

```
<MetaType name="DotNetTypeLoader" namespace"java2dotnet">
  <MetaProperties>
    <MetaProperty name="ITypeInfo.TYPEKIND"
                  type="int"
                  value="TKIND_COCLASS"/>
    <MetaProperty name="ImplementationType"
                  type="string"
                  value="java2dotnet._DotNetTypeLoader">
      <MetaProperties>
        <MetaProperty name="ImplementationTypeFlags"
                      type="int"
                      value="1"/>
      </MetaProperties>
    </MetaProperty>
  </MetaProperties>
</MetaType>
```

Listing 4.10: Die COM-Klasse DotNetTypeLoader in XML

Die COM-Komponente *DotNetTypeLoader* ist unter dem Namen bzw. der ProgID *java2dotnet.DotNetTypeLoader* in der Windows Registry registriert und besitzt gemäß Listing 4.11 eine Dispatch-Schnittstelle namens *DotNetTypeLoader*. Die Schnittstelle umfasst u.a. die Funktion (wFlags=INVOKE_FUNC) *getAssemblyFromFile()*, die als Operan-

den einen Wert vom Typ *string* erwartet und ein Objekt vom Typ *mscorlib._Assembly* zurück gibt.

```
<MetaType name="_DotNetTypeLoader" namespace="java2dotnet">
  <MetaProperties>
    <MetaProperty name="ProgID"
                  type="string"
                  value="java2dotnet.DotNetTypeLoader" />
    <MetaProperty name="ITypeInfo.TYPEKIND"
                  type="int"
                  value="TKIND_DISPATCH" />
  </MetaProperties>
  <MetaOperations>
    <MetaOperation name="getAssemblyFromFile">
      <MetaType name="_Assembly" namespace="mscorlib">
        <MetaProperties>
          <MetaProperty name="IDLType"
                        type="string"
                        value="VT_PTR+VT_USERDEFINED(3000025):_Assembly" />
        </MetaProperties>
      </MetaType>
      <MetaProperties>
        <MetaProperty name="wFlags" type="string" value="INVOKE_FUNC"/>
      </MetaProperties>
      <MetaOperands>
        <MetaOperand name="assemblyFile">
          <MetaType name="string">
            <MetaProperties>
              <MetaProperty name="IDLType" type="string" value="VT_BSTR"/>
              <MetaProperty name="IDLValue" type="int" value="1"/>
            </MetaProperties>
          </MetaType>
        </MetaOperand>
      </MetaOperands>
    </MetaOperation>
    ...
  </MetaOperations>
</MetaType>
```

Listing 4.11: Das COM-Dispatch-Interface _DotNetTypeLoader in XML

Mit den gezeigten XML-Dokumenten liegt eine persistente Repräsentation der COM-Komponente *java2dotnet.DotNetTypeLoader* vor. Die Dokumente enthalten sämtliche Informationen, die zur Wiederherstellung des speicherbasierten Metamodells notwendig sind, so dass auf eine erneute Metaisierung verzichtet werden kann. Die Dokumente können beliebig oft und für unterschiedliche Prozesse, auf die in Kapitel 5 eingegangen wird, wiederverwendet werden.

4.2.6 Serialisierung und Deserialisierung

Zur Erzeugung von serialisierten Metamodellen stehen prinzipiell zwei Möglichkeiten zur Verfügung. Eine Möglichkeit ist die automatische Serialisierung von speicherbasierten Metamodellen mittels eines Programmes bzw. einer API, die andere ist die manuelle Erzeugung von Dokumenten mit Hilfe von Werkzeugen wie XML-Editoren.

4.2.6.1 Automatische (De-)Serialisierung auf Basis des Metamodells

Zur automatischen (De-)Serialisierung von speicherbasierten Metamodellen steht eine komfortable Java-API in Form eines Java-Pakets zur Verfügung. Das Paket umfasst prinzipiell die zwei Klassen *MAX2MAXML* und *MAXML2MAX* .

Die Klasse *MAX2MAXML* erhält als Eingabe eine Referenz auf ein speicherbasiertes Metamodell und einen Pfad, an dem die XML-Dokumente gespeichert werden sollen. Die Serialisierung selbst kann über Parameter beeinflusst werden. Wahlweise können ein einzelnes XML-Dokument oder mehrere XML-Dokumente erzeugt werden. Das einzelne XML-Dokument enthält sämtliche Typen, die von einer Komponente definiert werden. Im anderen Fall wird für jeden Typ ein XML-Dokument erzeugt und im Dateisystem gespeichert, wobei, wie in Abschnitt 4.2.4.1 erwähnt, ein Hauptdokument existiert, das per XInclude Verweise auf die einzelnen Dokumente enthält.

Zur Deserialisierung von MAXML-Dokumenten steht die Klasse *MAXML2MAX* zur Verfügung. Bei der Deserialisierung werden ein bzw. mehrere persistente XML-Dokumente in ein speicherbasiertes Metamodell eingelesen. Die Klasse erhält als Eingabe den vollständig qualifizierten Namen eines XML-Dokumentes und liefert eine Referenz auf ein speicherbasiertes Metamodell zurück. Falls mehrere XML-Dokumente durch ein Hauptdokument referenziert werden, werden die Referenzen aufgelöst und in das Hauptdokument integriert. Dies geschieht vollkommen transparent durch den verwendeten XML-Parser Xerces [Apa04g], der ab Version 2.5.0 die Verwendung von XInclude unterstützt.

Bei einigen Technologien, z.B. CANopen und in Ansätzen CORBA, können existierende Dokumente zur Metaisierung herangezogen werden. Die Werkzeuge CANopen2MAXML und CORBA2MAXML importieren Quelldokumente im EDS- bzw. IDL-Format, erzeugen ein temporäres speicherbasiertes Metamodell und exportieren dieses Modell nach XML.

4.2.7 Manuelle Serialisierung

Viele Technologien besitzen keine oder nur minimale reflektive Eigenschaften und folglich auch keine Möglichkeit zur automatischen Konstruktion von speicherbasierten Metamodellen. Beispielsweise können zwar die von einer Bibliothek des Windows Betriebssystems exportierten Operationen, nicht jedoch deren Operanden bestimmt werden[2]. Zur Integration von Technologien bzw. Systemen steht in den meisten Fällen eine Schnittstelle (API) und ein zugehörige Beschreibung zur Verfügung. Üblicherweise werden diese Beschreibungen in Form von Dokumentation oder Bibliotheken geliefert. Eine Dokumentation kann meistens nicht automatisch verarbeitet werden. Bibliotheken können zwar automatisch verarbeitet werden, besitzen jedoch ein binäres Format, so dass entsprechende Werkzeuge zur Verarbeitung notwendig sind.

Aufgrund der Einfachheit des allgemeinen Metamodells und von XML, kann eine Beschreibung einer Schnittstelle bzw. einer Komponente mit einem beliebigen (XML-) Editor manuell erstellt werden[3]. Die Erstellung eines serialisierten Metamodells mit einem

[2]Eine Möglichkeit zur Extraktion der Operanden mit vertretbarem Aufwand entzieht sich der Kenntnis des Autors.

[3]Auf die Realisierung eines speziellen Werkzeuges zur Erstellung von serialisierten Metamodellen wur-

Editor wird als *manuelle Serialisierung* bezeichnet.

Im Rahmen dieser Arbeit wurde auf Grundlage einer Dokumentation, eine Schnittstellenbeschreibung für eine PC-Komponente (digital, analoge E/A-Karte) erstellt und für die automatische Generierung einer JNI-Schnittstelle gemäß Abschnitt 5.4.3 zur Integration dieser Komponente in die Java-Laufzeitumgebung gemäß Abschnitt 2.3.3.6 benutzt. Das realisierte System wird in Abschnitt 6.3.2 im Detail vorgestellt.

4.2.8 Bemerkungen

Es ist offensichtlich, dass den eigentlichen Daten viele Metadaten hinzugefügt werden. Eine Reduzierung der Metadaten ist z.B. durch die Verwendung von kürzeren Namen und eine Referenzierung von redundanten Daten möglich. Da viele Datentypen mehrfach in einer Komponente verwendet werden, kann anstatt der Deklaration des Typs durch Namen und Namensraum ein Identifikator benutzt werden, der eine Referenz auf die Deklaration des Typs enthält, wie dies bei XMI [Obj02] der Fall ist. Die Referenzen werden erst bei der Verarbeitung von XML-Dokumenten aufgelöst. Da der Schwerpunkt bei der Serialisierung des Metamodells in dieser Arbeit hauptsächlich auf der Lesbarkeit der Beschreibungen liegt, wird auf eine Datenreduktion verzichtet.

4.3 Zusammenfassung

Das allgemeine Metamodell von MAX bietet eine Verallgemeinerung der in den Abschnitten 3.2 ff. vorgestellten Metamodelle. Dabei bleibt der Abstraktionsgrad bis auf die Modellierung von CANopen-Komponenten erhalten. Sprach- bzw. modellspezifische Eigenschaften werden durch die Verwendung von Attributen externalisiert, so dass Komponenten aus unterschiedlichen Sprachen und Technologien dieselbe Repräsentation besitzen. Die Verallgemeinerung der Metamodelle und Abbildung der einzelnen Metamodelle in ein allgemeines Metamodell bietet den Vorteil, dass dieselben Werkzeuge zur Verarbeitung des Metamodells herangezogen werden können und von sprachspezifischen Eigenschaften abstrahiert wird.

Durch die Serialisierung des allgemeinen Metamodelles nach XML liegt eine persistente Repräsentation des Modells vor, das beliebig oft und zu unterschiedlichen Zwecken wiederverwendet werden kann. Die generierten XML-Beschreibungen sind einfach lesbar und einfach, auch manuell, weiter zu verarbeiten. Zur Serialisierung und Deserialisierung von Metamodellen steht eine einfache Schnittstelle zur Verfügung, so dass sich ein Entwickler nicht um die Details der Serialisierung bzw. Deserialisierung kümmern muss. Die XML-Beschreibungen können manuell erzeugt und editiert werden. Darüberhinaus können die Beschreibungen beliebig oft und zu unterschiedlichen Zwecken wiederverwendet werden. Im Rahmen dieser Arbeit werden die Beschreibungen zur Generierung von unterschiedlichen Softwarekomponenten und Dokumentationen wiederverwendet.

---

de verzichtet, da eine Vielzahl von allgemeinen Werkzeugen zur Verfügung steht.

5

Generierung von Softwarekomponenten

Die in einem ersten Schritt metaisierten (s. Kapitel 3) und in einem zweiten Schritt in ein allgemeines Metamodell bzw. dessen XML-Repräsentation abgebildeten Komponenten (s. Kapitel 4) werden nun in einem dritten Schritt zur automatischen Generierung von Softwarekomponenten und Dokumentation benutzt. Im Mittelpunkt steht dabei die Integration von Komponenten in unterschiedliche (Middleware-) Infrastrukturen.

Mit der automatischen Codegenerierung steht ein mächtiges Werkzeug zur Integration der unterschiedlichen Komponenten in verschiedene Infrastrukturen zur Verfügung. Die generelle Repräsentation von Komponenten auf Basis eines allgemeinen Metamodells erlaubt eine automatische Verarbeitung der Schnittstelle und somit eine automatische Generierung von Software. Dabei werden die ursprünglichen Objekte unverändert übernommen und durch entsprechende generierte Objekte gekapselt.

Aufgrund der plattform- und sprachunabhängigen Repräsentation (s. Abschnitt 4.1.1) kann unterschiedlicher Quelltext erzeugt werden, sofern die Quellkomponente bzw. das Quellobjekt in der erzeugten Sprache referenziert und integriert werden kann.

Das Wissen um die Generierung von entsprechenden Softwarekomponenten findet sich in unterschiedlichen sogenannten Softwareexperten wieder. MAX umfasst Softwareexperten zur Generierung von RMI-, Jini-, JNI-, COM-, CANopen- und Web Services-Komponenten und Dokumenten im HTML- und PDF-Format.

Als Basis für die Generierung von Softwarekomponenten dienen das in Kapitel 4 vorgestellte speicherbasierte allgemeine Metamodell bzw. dessen XML-Repräsentation.

5.1 Einführung

Eine automatische Generierung von Code ist Bestandteil vieler Technologien und wird u.a. bei RPC, CORBA, RMI und COM/DCOM angewandt. Dabei basiert die Generierung bei diesen Technologien entweder auf externen (RPC, CORBA, COM/DCOM) oder auf internen (RMI) Metadaten. Bei den Technologien zum entfernten Prozeduraufruf werden auf Basis von Schnittstellenbeschreibungen die Dateien bzw. Klassen, Stubs und

Skeletons zur verteilten Kommunikation automatisch erzeugt. Die Stubs und Skeletons kapseln die Kommunikation zwischen einem Client und einem Server und den eigentlichen Aufruf der Prozedur des Quellobjektes. Während bei RPC und CORBA externe Beschreibungen benutzt werden, wird bei Java RMI die Schnittstelle per Reflexion extrahiert. Bei COM stehen unterschiedliche Werkzeuge zur automatischen Codegenerierung unter Verwendung von Typbibliotheken zur Verfügung. Basierend auf Typinformationen können COM-Klassen in unterschiedlichen Sprachen, z.B. Visual C++ und Visual Basic, gekapselt werden.

5.2 Techniken zur Generierung von Komponenten

Bei der Generierung von neuen Komponenten werden die ursprünglichen Komponenten in die Neuen integriert. Diese Integration kann mit verschiedenen Konzepten und unterschiedlichen Techniken realisiert werden. Letztendlich geht es immer um die Kapselung von Quell- durch Zielkomponenten, wobei letztere über eine gewisse Zusatzfunktionalität, z.B. die Möglichkeit zur entfernten Kommunikation, verfügen.

5.2.1 Kapselung von Komponenten und Objekten

Die Kapselung einer Komponente ist ein genereller Ansatz, um einerseits Funktionalität zu einer Komponente hinzuzufügen, andererseits diese dabei unverändert zu lassen [WS02]. Eine Möglichkeit zur Kapselung von Komponenten bieten objektorientierte Sprachen mit den Konzepten der Aggregation, Komposition und Delegation. MAX benutzt diese Konzepte auf Basis von bewährten Entwurfsmustern.

Laut Gamma et. al. [GHJV95], Lieberman [Lie86] und Johnson [JZ91] ist die „Komposition durch Delegation in Bezug auf die Wiederverwendung so mächtig wie das Konzept der Vererbung“. Delegation sei „ein extremes Beispiel von Objektkomposition“ und zeige „dass es immer möglich ist, Vererbung durch Komposition als Mechanismus zur Wiederverwendung zu ersetzen“ [GHJV95]. Details zu den genannten Konzepten finden sich in [Boo94] bzw. [GHJV95].

5.2.1.1 Patterns zur Kapselung von Komponenten

Die Kapselung von Komponenten bzw. Klassen ist Gegenstand von einigen Entwurfsmustern. Zu diesen Mustern zählen u.a. das Proxy- bzw. Surrogate-Pattern und das Adapter- bzw. Wrapper-Pattern. Die Technik zur Kapselung eines Objektes durch ein anderes Objekt basiert prinzipiell auf dem Proxy-Pattern. Das Proxy-Pattern definiert gemäß Abb. 5.1 einen Stellvertreter (Platzhalter, Ersatz) für ein anderes Objekt, um den Zugriff auf dieses Objekt zu kontrollieren.

Ein Proxy besitzt eine Referenz auf ein reales Objekt, eine zu diesem Objekt identische Schnittstelle und kontrolliert den Zugriff auf das reale Objekt. Zudem ist ein Proxy u.U. für die Erzeugung und Löschung eines Objektes verantwortlich. Das Adapter- bzw. Wrapper-Pattern haben Ähnlichkeiten zum Proxy-Pattern, besitzen aber im Unterschied

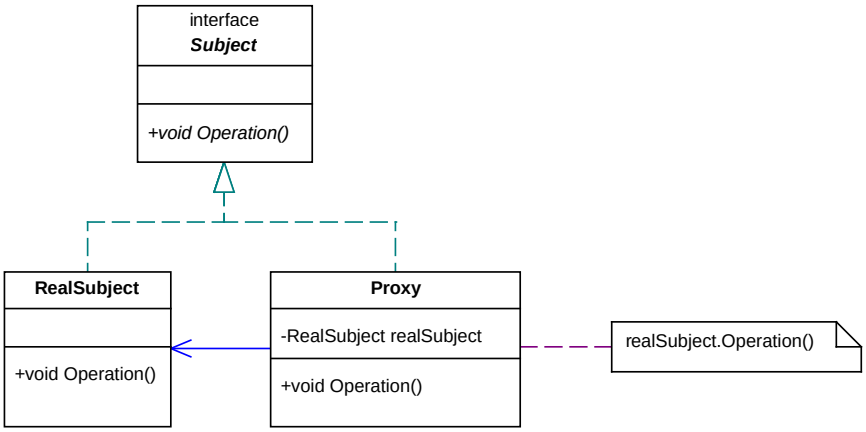


Abbildung 5.1: Die Struktur des Proxy-Pattern gemäß Gamma et al. [GHJV95]

zum Proxy eine vom realen Objekt unterschiedliche Schnittstelle. Das Proxy-Pattern ist laut Gamma et al. [GHJV95] in den folgenden Fällen anwendbar:

- Ein *entfernter Proxy* bietet eine lokale Repräsentation eines Objekts in einem anderen Adressraum.
- Ein *virtueller Proxy* erzeugt ein Objekt bei Bedarf.
- Ein *geschützter Proxy* kontrolliert den Zugriff auf das Originalobjekt.
- Ein *smarter Proxy* führt bei einem Zugriff auf ein Objekt zusätzliche Aktionen aus.

Bei der automatischen Generierung von Komponenten werden die ursprünglichen Komponenten bzw. Objekte durch neue Komponenten gekapselt. Dabei wird die Schnittstelle einer Quellkomponente durch eine entsprechende Schnittstelle einer Zielkomponente abgebildet und die Quell- in die Zielkomponente eingebunden. Auf mögliche Realisierungen von zusätzlichen, nicht-funktionalen Eigenschaften wird in Abschnitt 5.7 näher eingegangen.

5.2.2 Vererbung und Mehrfachvererbung

Einige objektorientierte Programmiersprachen wie z.B. Eiffel, C++, u.a. unterstützen das Prinzip der Mehrfachvererbung von Klassen, d.h. eine Klasse kann von mehreren Klassen abgeleitet sein. Im Gegensatz dazu unterstützen z.B. Smalltalk, Java oder C# nur eine Einfachvererbung. COM unterstützt zwar keine Vererbung, realisiert aber ein ähnliches Konzept über die Komposition von Schnittstellen. Bei der Kapselung von Sprachen mit Mehrfachvererbung durch eine Programmiersprache ohne Vererbung bzw. ohne Mehrfachvererbung besteht die Schwierigkeit, die genannten Konzepte adäquat zu modellieren.

Das Prinzip der Mehrfachvererbung kann durch programmiersprachliche Konstrukte simuliert werden. In Java ist es beispielsweise möglich, Mehrfachvererbung durch die Verwendung von Schnittstellen zu simulieren [TKH99, BLV01]. Die Simulation der Mehrfachvererbung kommt bei der Kapselung von COM-Komponenten in Java zum Tragen.

5.2.3 Bemerkung

Die Kapselung existierender Software mittels Proxies bzw. Wrappern bringt gewisse Nachteile mit sich. Zum einen sind ein oder mehrere zusätzliche Indirektionsschritte zur Ausführung einer Aktion notwendig und führen somit zu einem schlechteren Laufzeitverhalten. Zum anderen tragen die Indirektionsschritte bzw. zusätzlicher Code zu einer gewissen Komplexität bei. Wrapper geniessen aus diesen Gründen ein relativ schlechtes Ansehen in der Computerindustrie [Ear99]. Tatsache ist, dass das Einfügen von zusätzlichem Code einen gewissen Overhead erzeugt. Dies kann aber aufgrund der heutzutage zur Verfügung stehenden Rechenleistung und dem zur Verfügung stehenden Speicher nahezu vernachlässigt werden[1]. Außerdem kann die Benutzung von Wrappern maßgeblich zur Wiederverwendung von Software beitragen. In vorliegender Arbeit wird darüber hinaus davon ausgegangen, dass generierter Code nicht manuell bearbeitet werden muss, so dass ein Entwickler diesen Code im Regelfall nicht analysieren muss.

5.3 Generierung von Middlewarekomponenten in MAX

MAX unterstützt die Generierung von Proxies zur Integration von Quellkomponenten in verteilte Middlewareinfrastrukturen. Die unter Abschnitt 2.3.3 beschriebenen Vorgehensweisen werden hierzu automatisch durchgeführt. Grundsätzlich wird zunächst die XML-Spezifikation einer Quellkomponente mittels *MAXML2MAX* in das speicherbasierte allgemeine Metamodell übertragen. Dies gilt nicht nur für die Generierung von Middleware- sondern auch für die Generierung von Technologie- bzw. CANopen-Softwarekomponenten.

5.3.1 Generierung von RMI-Komponenten

Das Java-Paket *java.rmi* stellt die notwendigen Schnittstellen und Klassen zur Kommunikation mit RMI zur Verfügung. Dieses Paket respektive die notwendigen Klassen aus diesem Paket werden von den generierten Java-Klassen importiert bzw. entsprechend referenziert.

5.3.1.1 Generierung von Java RMI-Klassen und -Schnittstellen

Die Klasse *MetaType* repräsentiert einen Typ auf den entfernt zugegriffen werden soll. Zur Realisierung eines RMI-Objektes wird eine Remote-Schnittstelle und eine Implementierung dieser Schnittstelle benötigt. Der Quelltyp wird gemäß dem Proxy-Pattern aus Abschnitt 5.2.1.1 durch einen Proxy gekapselt.

[1] Diese Aussage gilt nicht unbedingt für eingebettete Systeme

Zunächst wird ein Java RMI Remote-Interface entsprechend Listing 5.1 deklariert, welches den Namen eines Typs zuzüglich der Kennung *ServiceI* erhält. Dabei wird für jede Operation eines Typs eine entsprechende Methode mit zugehörigem Typ (*optype*) und zugehörigen Parametern (*operands*), bestehend aus Parametertyp und -name, zu dem Interface hinzugefügt.

```
public interface <typename>ServiceI
extends Remote
{
   public <optype> <opname>(<[operands]>) throws java.rmi.RemoteException;
}
```

Listing 5.1: Muster einer Java RMI Remote-Schnittstelle

Die Implementierung der Schnittstelle in Listing 5.2 erhält den Namen des Typs mit der Kennung *Service*. Diese Klasse wird gemäß RMI-Spezifikation von *UnicastRemoteObject* abgeleitet und implementiert die zugehörige Schnittstelle (Zeilen 1-3).

Damit ein Quellobjekt benutzt werden kann, wird im Konstruktor wahlweise ein Quellobjekt übergeben oder eine Instanz eines Quellobjektes erzeugt. Im ersten Fall wird ein Konstruktor benötigt, der als Parameter eine Referenz auf das Quellobjekt erhält. Im zweiten Fall werden ein oder mehrere zu den Konstruktoren des Quellobjektes identische Konstruktoren erzeugt. In beiden Fällen wird eine Referenz auf das Quellobjekt als globale Variable zu der erzeugten Klasse hinzugefügt.

```
public class <typename>Service
extends java.rmi.server.UnicastRemoteObject
implements <typename>ServiceI
{
  private <typename> obj = null;

  public <typename>Service(<typename> obj)
    throws java.rmi.RemoteException
  {
    this.obj = obj;
  }

  public <optype> <opname>(<[operands]>)
    throws java.rmi.RemoteException;
  {
    return (obj.<opname>());
  }

  public <typename> getRealSubject()
  {
    return this.obj;
  }
}
```

Listing 5.2: Muster einer Java RMI Remote-Klasse

Für jede Methode des Quelltyps bzw. der bereits erzeugten Remote-Schnittstelle wird eine entsprechende Implementierung der Methode generiert. Dabei wird in den Methodenrumpf eine entsprechende Anweisung zur Ausführung der jeweiligen Methode des Quellobjektes eingefügt[2].

[2]In dem Beispiel wurde auf die Darstellung der Ausnahmebehandlung verzichtet

Die bisherige Verfahrensweise funktioniert nur bei einfachen serialisierbaren Datentypen. Bei komplexen, nicht serialisierbaren Datentypen wird für jeden Typ ein entsprechendes Remote-Interface und die entsprechende Implementierung analog zu oben generiert. Eine Methode mit einem komplexen Rückgabetyp ist in Listing 5.3 dargestellt. Ein komplexer Typ wird demnach durch seinen Proxy repräsentiert. Analog dazu werden für komplexe Parametertypen ebenfalls entsprechende Remote-Schnittstellen und -Klassen erzeugt. Beim Aufruf einer entsprechenden Methode eines Quellobjekts wird eine Referenz auf das reale (lokale) Quellobjekt übergeben (s. Listing 5.2 Zeilen 19-22). Abschnitt 5.3.1.3 enthält weitere Beispiele für die Behandlung von komplexen Datentypen.

```
public <optype>ServiceI <opname>([<operands>])
throws java.rmi.RemoteException
{
  <optype>ServiceI ret = null;
  try {
    ret = (<optype>ServiceI) (new <optype>Service(obj.opname([<operands>]));
  } catch (Exception e) {
    e.printStackTrace();
  }
  return ret;
}
```

Listing 5.3: Beispiel einer generierten Java RMI Remote-Methode mit komplexem Rückgabetyp

Optional wird eine beispielhafte Methode zur Registrierung eines entfernten Objektes hinzugefügt. Die Methode *register()* in Listing 5.4 enthält Anweisungen zur Installation eines sog. Security-Managers, zur Instantiierung eines RMI-Objekts[3] und zur Registrierung dieses Objekts bei der RMI-Registry.

```
public void register()
{
  System.setSecurityManager(new java.rmi.RMISecurityManager());
  try {
    <typename>Service svc = new <typename>Service();
    Naming.rebind("//localhost:2005/<typename>Service", svc);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listing 5.4: Beispiel einer generierten Java RMI-Registrierung

5.3.1.2 Bemerkungen

Bei der automatischen Generierung von Komponenten können Namenskonflikte und Schwierigkeiten bzgl. der Ausnahmebehandlung auftreten.

Namenskonflikte treten auf, falls ein Quellobjekt eine Methode definiert, die bereits durch *UnicastRemoteObject* definiert wird, aber eine andere Signatur besitzt. Da die erzeugte Klasse von *UnicastRemoteObject* abgeleitet ist, können somit manche Methoden nicht überschrieben werden. Die Java-Klasse *String* definiert z.B. eine Methode *toString()*. Diese Methode wird in den erzeugten Klassen mit *RemoteException* deklariert.

[3] In der Regel muss die Instantiierung manuell angepasst werden.

Bei der Übersetzung der Klassen tritt ein Fehler auf, da die Methode *toString()* von *UnicastRemoteObject* bzw. *RemoteObject* eine andere Signatur besitzt und somit nicht überschrieben werden kann.

Zur Auflösung bzw. Vermeidung von Namenskonflikten wird jede Methode mit dem Präfix *svc_* versehen. Bei der Definition des Proxy-Pattern wurde darauf hingewiesen, dass ein Proxy eine identische Schnittstelle wie das Quellobjekt besitzt. Dies ist zwar in diesem Fall nicht mehr gegeben, es besteht jedoch die Möglichkeit, nur diejenigen Methoden umzubenennen, bei denen Namenskonflikte auftreten.

Bei der Behandlung von Ausnahmen ist die Reihenfolge der Behandlung zu beachten. Bei der Reflexion bzw. Extraktion der Ausnahmen wird keine definierte Reihenfolge eingehalten. In Java ist es jedoch entscheidend, dass Ausnahmen, die von anderen Ausnahmen abgeleitet sind, zuerst abgefangen werden. Deshalb wird die Klassenhierarchie schon bei der Metaisierung untersucht und gemäß der Vererbungshierarchie sortiert.

5.3.1.3 Beispiel einer generierten RMI-Komponente

In Abschnitt 3.3.5 wurde eine COM-Komponente zum Laden von .NET-Typen vorgestellt. Wie später in Abschnitt 5.4.1.2 ersichtlich wird, ist diese Komponente in Java gekapselt. Die automatisch generierte Java-Komponente soll ihrerseits wiederum per RMI zur Verfügung stehen. Das Beispiel verdeutlicht zudem die Vorgehensweise bei nicht serialisierbaren Datentypen.

Die Java-Klasse *DotNetTypeLoader* kapselt die COM-Komponente *DotNetTypeLoader* und soll von einem entfernten Client benutzt werden können. Wie oben beschrieben, wird ein Java-Interface mit den entsprechenden Methoden gemäß Listing 5.5 generiert.

```
public interface DotNetTypeLoaderServiceI
extends java.rmi.Remote
{
  public _TypeServiceI svc_getType_2(String typeName) throws ...;
  public _AssemblyServiceI svc_getAssemblyFromFile(String assemblyFile) ...;
  ...
}
```

Listing 5.5: Beispiel einer generierten Java RMI-Schnittstelle

Die Implementierung der Schnittstelle, die Klasse *DotNetTypeLoaderService*, ist in Listing 5.6 dargestellt. Das Quellobjekt *DotNetTypeLoader* besitzt eine Methode namens *getType_2*, die unter Angabe eines Namens ein Objekt von *mscorlib._Type* liefert. Da es sich hierbei um ein nicht serialisierbares Objekt handelt, wird für diese Klasse ebenfalls eine entsprechende Schnittstelle namens *_TypeServiceI* und die zugehörige Implementierung namens *_TypeService* generiert. Die Klasse *_TypeService* erhält im Konstruktor ein Quellobjekt vom Typ *mscorlib._Type*.

```
public class DotNetTypeLoaderService
extends java.rmi.server.UnicastRemoteObject
implements DotNetTypeLoaderServiceI
{
  private java2dotnet.DotNetTypeLoader obj;

  public DotNetTypeLoaderService(java2dotnet.DotNetTypeLoader obj)
    throws java.rmi.RemoteException
```

```
  {
    this.obj = obj;
  }

  public _TypeServiceI svc_getType_2(String typeName)
    throws java.rmi.RemoteException
  {
    _TypeServiceI ret = null;
    try {
      ret = (_TypeServiceI) (new _TypeService(obj.getType_2(typeName)));
    } catch (Exception e) {
      e.printStackTrace();
    }
    return ret;
  }
  ...
}
```

Listing 5.6: Beispiel einer generierten Java RMI-Komponente

Handelt es sich bei einem Rückgabetyp einer Methode um ein Feld von nicht serialisierbaren Objekten, so wird entsprechend Listing 5.7 für jedes Quellobjekt ein Proxy instantiiert. Das Listing zeigt die Methode *svc_GetMethods_2()* der Klasse *_TypeService* zur Bestimmung der von einem Typ deklarierten Methoden.

```
public _MethodInfoServiceI[] svc_GetMethods_2()
throws java.rmi.RemoteException
{
  _MethodInfoServiceI[] ret = null;
  mscorlib._MethodInfo[] tmp = null;
  try {
    tmp = obj.GetMethods_2();
    ret = new _MethodInfoServiceI[tmp.length];
    for (int n = 0; n < tmp.length; n++)
      ret[n] = ((_MethodInfoServiceI) new _MethodInfoService(tmp[n]));
  } catch (Exception e) {
    e.printStackTrace();
  }
  return ret;
}
```

Listing 5.7: Beispiel einer generierten Java RMI-Komponente

5.3.1.4 Konfiguration von RMI-Komponenten

Bei der Generierung können externe Metadaten zur Parametrierung einer Komponente hinzugezogen werden. Somit können variable Bestandteile, wie z.B. der verwendete Security-Manager oder der Namen des RMI-Objektes, der zur Registrierung beim Namensdienst verwendet wird, konfiguriert werden. Die externen Konfigurationsdaten liegen, wie in Listing 5.8 gezeigt, ebenfalls als XML-Dokument vor.

```
<?xml version="1.0" encoding="UTF-8"?>
<Service name="RMI">
 <MetaProperties>
  <MetaProperty name="Manager" type="string" value="RMISecurityManager" />
  <MetaProperty name="Name" type="string" value="DotNetTypeLoaderService" />
  <MetaProperty name="Host" type="string" value="localhost" />
  <MetaProperty name="Port" type="int" value="2005" />
 </MetaProperties>
</Service>
```

Listing 5.8: Java RMI-Konfigurationsdaten

5.3.2 Generierung von Jini-Komponenten

Jini ist ein Framework zur Implementierung von verteilten Diensten basierend auf Java und bietet eine Plattform zur verteilten Kommunikation. Die zugrunde liegende Realisierung von Jini ist in der aktuellen Implementierung von der Firma Sun Microsystems weitestgehend von RMI abhängig. In seiner einfachsten Form besitzt ein Jini-Dienst die gleiche Struktur wie ein entferntes RMI-Objekt.

5.3.2.1 Generierung von Jini-Klassen und -Schnittstellen

Die Generierung von Jini-Komponenten unterscheidet sich von der Generierung von RMI-Komponenten nur durch die Referenzierung von zusätzlichen Klassen und dem Vorgang der Registrierung eines Objektes. Die Schnittstelle und deren Implementierung wird analog zu RMI erzeugt. Der Hauptunterschied liegt in der Art und Weise der Registrierung von Objekten bei dem Verzeichnisdienst von Jini, dem sogenannten *Jini Lookup Service*.

Mit MAX lassen sich einfache Jini-Dienste bzw. die Grundfunktionalität zur Interaktion mit dem Jini-Framework erzeugen. Abschnitt 6.3.3.2 enthält ein beispielhaftes Szenario für die Integration von in Java gekapselten CANopen-Komponenten in Jini.

5.3.2.2 Konfiguration von Jini-Komponenten

Analog zur Parametrierung von RMI-Komponenten können Jini-Komponenten durch Angabe einer Konfigurationsdatei parametriert werden. Ein Beispiel für eine solche Konfigurationsdatei findet sich im Anhang in Listing A.3.

5.3.3 Generierung von .NET Remoting-Komponenten

Analog zu RMI und Jini werden bei der Generierung von .NET Remoting-Komponenten auf Basis des speicherbasierten Metamodells eine Schnittstelle und deren Implementierung in C# generiert. Zudem wird eine Referenz auf das Ursprungsobjekt in Form einer globalen Variable in die Implementierung eingefügt. Die einzelnen Methoden des Urprungsobjekts werden wiederum analog zum Proxy-Pattern durch entsprechende Methoden gekapselt. Zudem wird eine Registrierungsmethode mit der Registrierung eines Objekts beim .NET-Namensdienst eingefügt.

5.3.4 Generierung von Socket-Komponenten

Gemäß Abschnitt 2.3.2.3 steht bei manchen Systemen oftmals keine Alternative zur Socket-Kommunikation zur Verfügung. Dies betrifft vor allem eingebettete Systeme mit eingeschränkter Prozessorleistung und geringer Speicherkapazität. Diese Systeme bzw. die Komponenten auf diesen Systemen (z.B. CANopen-Module) sollen jedoch ebenfalls nahezu transparent in eine Service- und Management-Infrastruktur integrierbar werden. Aus diesem Grund unterstützt MAX die Generierung von Komponenten, die mittels

Socket-Kommunikation einfache entfernte Prozeduraufrufe realisieren. Die dafür notwendige Client- und Server-Funktionalität wird von MAX ebenfalls automatisch generiert. Die generierten Socket-Komponenten kommen beispielsweise bei der Integration von CANopen-Komponenten eines eingebetteten Systems in ein Jini-Framework zum Tragen (s. Abschnitt 6.3.3.2)[4].

5.3.4.1 MAXRPC

Die Generierung von sog. Socket-Komponenten basiert auf der Idee, einen einfachen entfernten Prozedur- bzw. Methodenaufruf mittels eines einfachen Serialisierungsprotokolls zu ermöglichen. Die hierfür notwendige Funktionalität ist im Java-Paket *MAXRPC* realisiert. Zum Aufruf von entfernten Methoden werden der Name und Datentyp einer Operation und die zugehörigen Operanden und deren Datentyp an eine Methode übergeben. Diese Methode serialisiert den Namen, die Operanden und zugehörigen Datentypen mittels eines einfachen Serialisierungsprotokolls in eine Folge von Bytes und schickt diese über einen zuvor aufgebauten Kommunikationskanal zum Server. Der Server deserialisiert die empfangen Bytes und führt die entsprechende Operation mit den angegebenen Operanden aus und schickt das Ergebnis der Operation an den Client zurück. Das von MAX verwendete Serialisierungprotokoll ist proprietär und unterstützt aktuell nur eine Serialisierung von einfachen Datentypen und nur eine minimale Fehlerbehandlung.

Der realisierte entfernte Operationsaufruf entspricht im Kern prinzipiell den bisher vorgestellten objektorientierten Prozedur- bzw. Methodenaufrufen, genügt jedoch minimalen Systemanforderungen.

5.3.4.2 Generierung von Java-Klassen zur Socket-Kommunikation

Zur Kapselung eines entfernten Typs über das oben vorgestellte Verfahren werden entsprechende Client- und Server-Klassen generiert. Die generierten Klassen werden dabei von der abstrakten Klasse *RemoteType* abgeleitet. Diese Klasse umfasst u.a. Methoden zum Aufbau einer Socket-Verbindung, zum Aufruf einer entfernten Methode (*callRemoteOperation()*) und eine abstrakte Methode namens *executeOperation()*.

Auf beiden Seiten (Client und Server) werden für jede Operation eines gegebenen Typs eine Methode deklariert, die entsprechende Anweisungen zur Serialisierung und zum Aufruf bzw. zur Deserialisierung und Ausführung einer Operation enthalten. Die abstrakte Methode *executeOperation()* wird dabei von den generierten Klassen implementiert. Innerhalb dieser Methode wird die entsprechende Operation bestimmt (Dispatching) und ausgeführt. Da nicht vorausgesetzt werden kann, dass ein Zielsystem Reflexion unterstützt, ist das Dispatching über einen einfachen Namensvergleich realisiert.

[4]Die CANopen-Komponenten werden durch Java-Komponenten auf einem eingebetteten System gekapselt. Die Java-Komponenten werden wiederum durch Socket-Komponenten gekapselt.

5.3.4.3 Beispiel einer generierten Socket-Komponente

Das folgende Beispiel setzt voraus, dass eine CANopen-Komponente gemäß Abschnitt 5.5.1 in Java gekapselt ist. Der generierte Java/CANopen-Proxy namens *Node14* liegt als Java-Klasse bzw. nach der Metaisierung als deren Metamodell vor. Diese Klasse wiederum wird ihrerseits in eine Client/Server-Komponente gekapselt.

Die Server-Klasse *Node14Service* in Listing 5.9 ist von *RemoteType* abgeleitet und implementiert die abstrakte Dispatch-Methode *executeOperation()*. Im Konstruktor wird eine Referenz auf den gekapselten Typ und der Port zur Socket-Kommunikation übergeben. Innerhalb der Dispatch-Methode wird die aufzurufende Operation ermittelt und ausgeführt. Das Beispiel enthält eine Methode[5] zum Setzen der Ausgänge eines digitalen CANopen-Moduls[6].

```
public class Node14Service
extends RemoteType
{
  private can0.Node14 obj = null;

  public Node14Service(can0.Node14 obj, int port)
  {
    super(port);
    this.obj = obj;
  }

  public Object executeOperation(String opname, Object[] operands)
  {
    ...
    if (opname.equalsIgnoreCase("setDigital8BitOutputBlock_0"))
      return setDigital8BitOutputBlock_0(operands);
    ...

    return null;
  }

  public Object setDigital8BitOutputBlock_0(Object[] operands)
  {
    byte p0 = ((Byte)operands[0]).byteValue();
    obj.setDigital8BitOutputBlock_0(p0);
    return null;
  }

  ...
}
```

Listing 5.9: Beispiel einer generierten Server-Klasse (Ausschnitt)

Die Client-Klasse erhält im Konstruktor den Namen und Port eines Servers, also eines entfernten Typs. Mittels der Methode *callRemoteOperation()* werden die angegebene Operation mitsamt dem Rückgabetyp und den Parametern serialisert und zum Server geschickt.

```
public class Node14Client
extends RemoteType
{
  public Node14Client(String host, int port)
```

[5] Der vollständige Name lautet *setDigital8BitOutputBlocks1Digital8BitOutputBlock_0*.

[6] Aufgrund der generischen Methode *executeOperation()* unterscheidet sich die Schnittstelle des Servers von der des gekapselten Typs insofern, als dass der Rückgabetyp und die Parameter vom Typ *Object* sind.

```
  {
    super(host, port);
  }

  public void setDigital8BitOutputBlock_0(byte p0)
  {
    Object[] operands = new Object[1];
    operands[0] = new Byte(p0);
    callRemoteOperation("setDigital8BitOutputBlock_0",
                        RemoteType.VOID,
                        operands);
  ...
}
```

Listing 5.10: Beispiel einer generierten Client-Klasse (Ausschnitt)

Bis auf die Instantiierung von Client und Server ist die Benutzung eines entfernten Typs für einen Benutzer transparent. Das gezeigte Beispiel ist Teil der Integration von CANopen-Komponenten in Jini gemäß Abschnitt 6.3.3.2.

5.3.5 Generierung von Web Services

Da sich die Generierung von Java und .NET Web Services nur marginal von der bereits dargestellten Codegenerierung unterscheidet, wird die Vorgehensweise im Folgenden nur skizziert.

5.3.5.1 Generierung von Java Web Services

Gemäß Abschnitt 2.3.3.5 muss eine Java-Anwendung, die einen Web Service benutzen möchte, in eine gegebene Web Service-Infrastruktur integriert werden. Dazu ist es notwendig, die entfernten Prozedur- bzw. Methodenaufrufe entsprechend zu kapseln. Das in Listing 2.9 gezeigte Beispiel eines Methodenaufrufs wird hierzu automatisch generiert. Für jede Methode wird eine entsprechende Methode im Client definiert. Ein Methodenrumpf enthält die Erzeugung einer SOAP-Nachricht, in der der Namen der aufzurufenden Methode und die Parameter verpackt und der SOAP-Nachricht hinzugefügt werden. Der Rückgabewert des Web Service wird entsprechend ausgewertet und konvertiert.

5.3.5.2 Generierung von .NET Web Services

In .NET wird jede Methode eines Server-Objekts, die als Web Service zur Verfügung stehen soll, durch eine Methode gekapselt, die mit dem Attribut [WebMethod] gemäß Listing 2.10 gekennzeichnet ist. Im Rumpf einer Methode wird die Methode des Server-Objektes analog zum Proxy-Pattern bzw. zur Generierung von RMI-Komponenten gekapselt und zur Ausführung gebracht.

5.3.6 Zusammenfassung

Die Generierung von Komponenten zur Integration dieser Komponenten in unterschiedliche verteilte Middlewareinfrastrukturen unterscheidet sich bei den einzelnen Techno-

logien nur marginal. Die notwendigen Schritte zur Integration einer Komponente werden von MAX automatisch ausgeführt und können durch entsprechende Konfigurationsdaten dynamisch angepasst werden. Die gezeigten Beispiele demonstrieren die Möglichkeiten zur Integration von Komponenten in verteilte Middlewareinfrastrukturen.

5.4 Generierung von Technologiebrücken

Neben der Generierung von Middlewarekomponenten zur verteilten Kommunikation unterstützt MAX die automatische Generierung von lokalen Middlewarekomponenten, den sog. Technologiebrücken. Mit Technologiebrücken werden Komponenten bezeichnet, die es erlauben, von einer bestimmten Technologie, z.B. von Java, auf Komponenten einer anderen Technologie, z.B. COM oder .NET, oder auf sprachfremde Bibliotheken, z.B. Windows-Bibliotheken, zuzugreifen. Das Ziel ist folglich eine Integration von unterschiedlichen Technologien. Eine notwendige Voraussetzung zur Integration ist die Verfügbarkeit einer Schnittstelle, die den Zugriff auf eine bestimmte Technologie erlaubt. Der Zugriff von Java auf COM- und .NET-Komponenten wird beispielsweise durch die Java2COM-Middleware ermöglicht. Der Zugriff auf Windows-Bibliotheken in Java ist mittels JNI realisiert.

5.4.1 Generierung von Java/COM-Komponenten

Wie bereits in Abschnitt 5.2.2 erwähnt, unterstützt COM zwar keine Mehrfachvererbung, realisiert diese aber durch die Komposition von Schnittstellen. Diese Technik kann in Java gleichermaßen angewendet werden.

Zur Anbindung einer COM-Komponente in Java wird für jeden COM-Typ eine entsprechende Java-Klasse oder -Schnittstelle benötigt, die ein COM-Objekt in Java kapselt. Analog zu der Generierung von verteilten Middlewarekomponenten werden Komponenten zur Kapselung von COM-Objekten auf Basis der XML-Metadaten bzw. des speicherbasierten Metamodells automatisch erzeugt.

5.4.1.1 Generierung von Java-Komponenten

Die Klassen zur Kapselung von COM-Komponenten benutzen ebenfalls die in Abschnitt 3.3.3 vorgestellte Java2COM-Klassenbibliothek. Zur Kommunikation mit der COM-Schnittstelle werden dazu entsprechende Anweisungen in den Java-Code eingefügt.

Aus Abschnitt 3.3.2 ist bekannt, dass COM unterschiedliche Typdeskriptoren definiert. Bei der Generierung von Softwarekomponenten zur Kapselung der COM-Typen wird ein Typdeskriptor gemäß Tabelle 5.1 durch eine Klasse oder eine Schnittstelle in Java repräsentiert.

Eine COM-Dispatch-Schnittstelle (*TKIND_DISPATCH*) wird durch ein Java-Interface und eine Klasse, die von *java2com.IDispatch* abgeleitet ist und das Interface implementiert, repräsentiert. Für jede Methode eines COM-Objektes wird eine entsprechende Methode in Java deklariert. Dabei werden die einfachen Datentypen gemäß Tabelle B.1 in ent-

Typdeskriptor	Java-Konstrukt
Dispinterface (*TKIND_DISPATCH*)	Java-Schnittstelle und -Klasse
Interface (*TKIND_INTERFACE*)	Java-Schnittstelle
CoClass (*TKIND_COCLASS*)	Java-Klasse, die eine oder mehrere Schnittstellen implementiert
Enum (*TKIND_ENUM*)	Java-Klasse bestehend aus statischen Variablen

Tabelle 5.1: Abbildung von COM-Typdeskriptoren in Java-Konstrukte

sprechende Java-Datentypen konvertiert. Gemäß Listing 5.11 umfasst eine generierte Dispatch-Methode die Methode *invoke()*[7] (Zeilen 8-10) zur Ausführung einer Methode eines COM-Objektes. Diese Methode erhält als Parameter den Namen einer Methode oder eines Property, die Art des Aufrufs und ein Feld von Parametern vom Typ *Object*, wobei die Parameter bei Bedarf zuvor in entsprechende Objekte konvertiert werden. Ein einfacher Datentyp `int` wird beispielsweise durch eine Klasse vom Typ *Integer* repräsentiert und in dem Feld gespeichert. Die Methode *invoke()* besitzt einen Rückgabetyp vom Typ *Object*, der entsprechend des Rückgabetyps einer Methode in den erforderlichen Datentyp konvertiert wird (Zeile 11).

```
  public mscorlib._Type getType_2(java.lang.String typeName)
  throws java2com.Java2ComException
  {
    mscorlib._Type   __pVal[] = { null };
5   java.lang.Object __pRetVal[] = { __pVal };
    java.lang.Object __pDispParams[] = { typeName };

    java.lang.Object rc = invoke("getType_2",
                                 java2com.IDispatch.INVOKE_FUNC,
10                               __pDispParams);
    convert(rc, __pRetVal);
    return (mscorlib._Type)__pVal[0];
  }
```

Listing 5.11: Beispiel einer generierten Java-Methode zur Kapselung einer Dispatch-Methode

COM-Klassen (*TKIND_COCLASS*) werden durch Java-Klassen gekapselt. Da eine COM-Klasse eine oder mehrere Schnittstellen implementieren kann, sind diese auch von der Java-Klasse zu implementieren. Für jede zu implementierende Schnittstelle existieren dabei Stellvertreter, sog. Delegates (s. Listing 5.13), die im Rumpf der Klasse deklariert und im Konstruktor instantiiert werden. Die Methoden der implementierten Schnittstellen werden mit der jeweiligen Signatur der Klasse hinzugefügt. Innerhalb einer Methode wird der entsprechende Aufruf des Stellvertreters eingefügt. Falls zwei oder mehrere Schnittstellen identische Methoden besitzen, wird nur eine der Methoden generiert.

Ist ein COM-Objekt ein Aufzählungstyp (*TKIND_ENUM*), so wird eine Klasse mit dem entsprechenden Namen erzeugt und sämtliche Variablen, die in dem COM-Objekt definiert sind, als statische Variablen der Klasse hinzugefügt.

[7] Diese Methode wird von der Basisklasse *java2com.IDispatch* zur Verfügung gestellt.

5.4.1.2 Beispiel einer generierten Java/COM-Komponente

Im Folgenden soll die Generierung von Java-Klassen zur Kapselung der COM-Komponente *DotNetTypeLoader* gemäß Abschnitt 3.3.5 an einem Beispiel verdeutlicht werden.

Die Komponente umfasst eine COM-Klasse und eine COM-Dispatch-Schnittstelle. Bei der Generierung einer Java-Komponente zur Kapselung der COM-Komponente werden eine Java-Schnittstelle und zwei Java-Klassen erzeugt. Die Dispatch-Schnittstelle wird durch die Java-Schnittstelle *_DotNetTypeLoader* gemäß Listing 5.12 und die Java-Klasse *_DotNetTypeLoaderDelegate* gemäß Listing 5.13 repräsentiert.

```
package java2dotnet;

public interface _DotNetTypeLoader
{
  java.lang.String ProgID = "java2dotnet.DotNetTypeLoader";

  public mscorlib._Type getType_2(java.lang.String typeName)
    throws java2com.Java2ComException;

  ...
}
```

Listing 5.12: Beispiel einer generierten Java-Schnittstelle zur Kapselung einer COM-Dispatch-Schnittstelle (Ausschnitt)

```
public class _DotNetTypeLoaderDelegate
extends java2com.IDispatch
implements java2dotnet._DotNetTypeLoader
{
  public _DotNetTypeLoaderDelegate(java2com.ReleaseManager rm)
  throws java2com.Java2ComException
  {
    super(rm, java2dotnet._DotNetTypeLoader.ProgID);
  }

  public mscorlib._Type getType_2(java.lang.String typeName)
  throws java2com.Java2ComException
  {
    ... // call invoke()
  }
}
```

Listing 5.13: Beispiel einer generierten Java-Klasse zur Kapselung einer COM-Dispatch-Schnittstelle (Ausschnitt)

Die COM-Klasse wird durch die Java-Klasse *DotNetTypeLoader* in Listing 5.14 gekapselt. Die Klasse implementiert die Dispatch-Schnittstelle *_DotNetTypeLoader* und erzeugt im Konstruktor eine Instanz auf den entsprechenden Stellvertreter *_DotNetTypeLoaderDelegate*[8].

```
package java2dotnet;

public class DotNetTypeLoader implements _DotNetTypeLoader
{
  private static final String ProgID = "java2dotnet.DotNetTypeLoader";
  private _DotNetTypeLoaderDelegate d__DotNetTypeLoaderDelegate = null;
```

[8] Die Klasse implementiert zusätzlich die Schnittstelle *mscorlib._Object*, da jedes Objekt in .NET von der Klasse *System.Object* abgeleitet ist.

```
  public DotNetTypeLoader(java2com.ReleaseManager rm)
  throws java2com.Java2ComException
  {
    d__DotNetTypeLoaderDelegate = new _DotNetTypeLoaderDelegate(rm, ProgID);
  }

  public mscorlib._Type getType_2(java.lang.String typeName)
  throws java2com.Java2ComException
  {
    return d__DotNetTypeLoaderDelegate.getType_2(typeName);
  }

  ...
}
```

Listing 5.14: Beispiel einer generierten Java-Klasse zur Kapselung einer COM-Klasse (Ausschnitt)

Die erzeugte Java/COM-Komponente ist ein Bestandteil der erzeugten Java/.NET-Middleware zur Metaisierung von .NET-Komponenten und wird zum Laden von .NET-Datentypen und -Assemblies in Java benutzt. Die Verwendung in Java ist Listing 3.3 zu entnehmen. Das Klassendiagramm in Abb. 5.2 verdeutlicht die Beziehungen der von MAX generierten Klassen zueinander.

5.4.2 Generierung von Java/.NET -Komponenten

Die Generierung von Java-Komponenten zur Interaktion mit .NET-Komponenten stellt eine Anwendung von MAX dar. Mittels der durch Reflexion gewonnenen Informationen wird automatisch eine Java/.NET-Technologiekomponente bzw. -Middleware zur Kommunikation mit dem .NET-Framework in Java erzeugt.

5.4.2.1 Generierung von Java-Klassen und -Schnittstellen zur Kapselung des .NET-Frameworks

.NET bietet eine COM-Schnittstelle zur Interaktion mit dem .NET-Framework, wobei die COM-Typen des Frameworks in einer Typbibliothek namens *mscorlib.tlb* definiert sind. Die Typbibliothek umfasst 1386 COM-Typen und referenziert die Standard-OLE-Bibliothek (stdole2.tlb) mit 41 COM-Typen. Diese Typbibliotheken werden gemäß Abschnitt 3.3 vollständig in MAX metaisiert und nach XML serialisiert. Entsprechend der Generierung von Java/COM-Komponenten gemäß Abschnitt 5.4.1 werden in Abhängigkeit des COM-Typs eine oder mehrere Java-Klassen generiert.

Damit wird dem Argument aus Abschnitt 2.2.5.2 Rechnung getragen, dass eine manuelle Implementierung eines Systems unter ökonomischen Gesichtspunkten nicht vertretbar ist.

5.4.2.2 Die Java/.NET-Middleware

Die Java/.NET-Middleware basiert auf der Java2COM-Middleware und umfasst inklusive der Komponente zum Laden von .NET-Typen und -Assemblies insgesamt 2110 Java-

Klassen. Die Middleware umfasst die Java-Pakete *mscorlib* (.NET-Framework), *stdole* (OLE) und *java2dotnet* (*DotNetTypeLoader*).

Für die Metaisierung von .NET-Komponenten ist vor allem die .NET-Klasse *System.Type* von Interesse, die durch die Java-Klasse *mscorlib._Type* repräsentiert wird. Diese Klasse bietet die notwendigen Methoden zur Reflexion bzw. zur Metaisierung von Komponenten gemäß Abschnitt 3.4. Das Klassendiagramm in Abb. 5.2 enthält eine Übersicht über einige Klassen, die für die Metaisierung von .NET-Komponenten in Java benutzt werden.

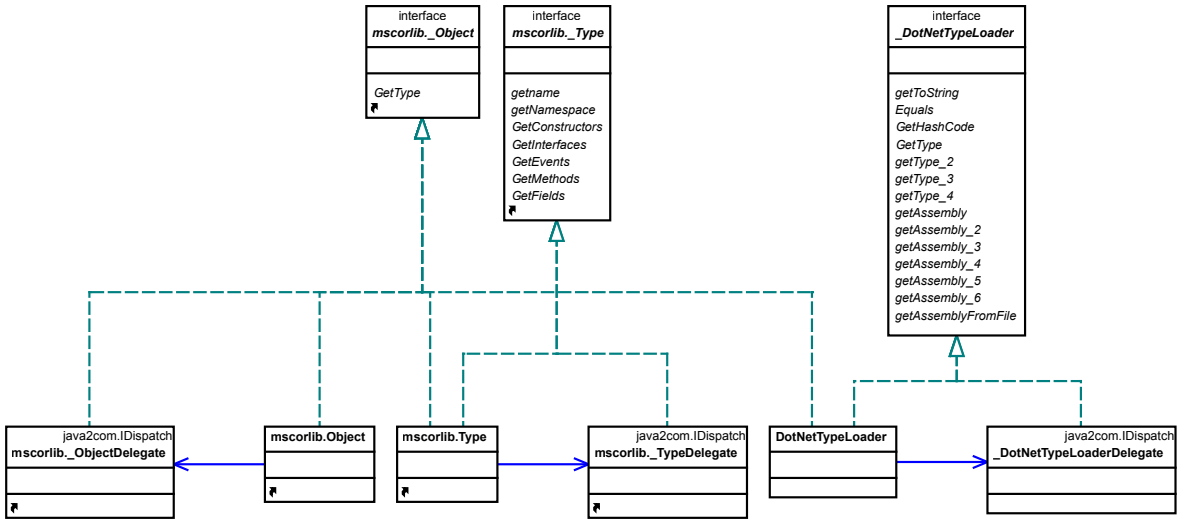


Abbildung 5.2: Klassendiagramm der Java/.NET-Middleware

Die automatische Generierung der Java/.NET-Middleware zeigt die Flexibilität und das Potenzial von MAX auf.

5.4.3 Generierung von JNI-Komponenten

In Abschnitt 2.3.3.6 wurde verdeutlicht, dass für die Anbindung von betriebssystemspezifischen Bibliotheken mehrere Schritte notwendig sind. Auf Basis der XML-Metadaten bzw. des allgemeinen Metamodells werden diese von MAX ebenfalls automatisch ausgeführt. Die Java-Klassen und C++-Dateien zur Integration von Bibliotheken werden mittels einer manuell erstellten Schnittstellenbeschreibung (s. Abschnitt 4.2.7) automatisch generiert.

5.4.3.1 Generierung von JNI-Klassen

In einem ersten Schritt werden eine oder mehrere Java-Klassen mit den entsprechenden Methoden deklariert. Dabei wird für jede Bibliotheksfunktion eine native Methode und eine Methode, welche die native Methode kapselt, hinzugefügt. Die notwendige JNI-Header-Datei für die Bibliothek wird mittels dem Werkzeug *javah* des Java Developer Kit (JDK) aus der Java-Klasse erzeugt.

5.4.3.2 Generierung von C++-Dateien

Die C++-Datei ist das Gegenstück zu den nativen Methoden und kapselt die Aufrufe der Bibliothek. Analog zu Java wird für jede native Methode eine Prozedur mit entsprechender Signatur erzeugt. Eine Prozedur kapselt die entsprechende Funktion der Quellbibliothek. Damit sind alle notwendigen Dateien zur Integration einer Bibliothek realisiert.

5.4.3.3 Beispiel

Analog zur Klasse *Calc* in Listing 2.1 implementiert eine Windows-Bibliothek namens *Calc.lib* die einfachen Grundrechenarten. Die Schnittstelle dieser Bibliothek liegt als (manuell) serialisierte XML-Beschreibung im Dateisystem vor. Aus den Metadaten wird, wie oben beschrieben, eine Java-Klasse gemäß Listing 5.15 erzeugt.

```
package jni.calc;

public class Calc {

  public double add(double a, double b) {
    return _add(a, b);
  }

  public native double _add(double a, double b);
}
```

Listing 5.15: Beispiel einer generierten JNI-Klasse

Die nativen Methoden werden in einer C++-Datei gemäß Listing 5.16 implementiert.

```
JNIEXPORT jdouble JNICALL Java_jni_calc_Calc__1add
  (JNIEnv *env, jobject obj, jdouble a, jdouble b)
{
  return add(a, b);
}
```

Listing 5.16: Beispiel einer generierten C-Funktion

Analog zu beschriebener Vorgehensweise wurden prototypisch Softwarekomponenten zur Kapselung einer digital/analogen PC-Schnittstellenkarte (PCI-20428W) und unterschiedlicher CAN-Hardware (CANCardX bzw. COM167, s. Abschnitt 2.5.3.2) generiert.

5.4.3.4 Bemerkung

Bei der Verwendung von Bibliotheken ist es oftmals der Fall, dass Resultate nicht, wie oben gezeigt, über den Rückgabewert einer Funktion, sondern über einen Argumentzeiger zurückgegeben werden. Die Bibliothek der PCI-20428W bietet z.B. die in Listing 5.17 dargestellten Funktionen *DIORead* und *AIRead* zum Lesen der digitalen bzw. analogen Eingänge. Diese Funktionen erhalten als Argument einen Zeiger zur Speicherung des Resultats an einer bestimmten Adresse[9].

[9]Das Resultat wird in beiden Fällen in einem 16-Bit Wert gespeichert.

```
int DIORead (int slot, ..., unsigned far *data)
int AIRead (int slot, ..., int far *data)
```

Listing 5.17: Schnittstelle der PCI-20428W-Bibliothek (Ausschnitt)

Dieser Fall wird bei der automatischen Codegenerierung aktuell nicht berücksichtigt. Es stehen jedoch Java-Klassen zur Kapselung von einfachen Datentypen zur Verfügung, die das Setzen von Werten in Java aus der nativen Bibliothek über einen sog. JNI-Callback gemäß Listing 5.18 heraus erlauben. Der nativen Methode wird dabei eine Referenz auf das entsprechende Callback-Objekt übergeben.

```
JNIEXPORT jint JNICALL Java_pci20428w__1DIORead
  (JNIEnv *env, ..., jobject objCallback)
{
  unsigned int value = 0;

  DIORead(..., &value);
  jclass clsCallback = env->GetObjectClass(objCallback);
  jmethodID methodSet = env->GetMethodID(clsCallback, "set", "(I)V");
  env->CallVoidMethod(objCallback, methodSet, value);
  ...
}
```

Listing 5.18: Beispiel eines JNI-Callback

Mit zusätzlichen Metadaten könnte diese Funktionalität mit entsprechenden Erweiterungen ebenfalls automatisch generiert werden.

5.5 Generierung von CANopen-Softwarekomponenten

Eine Hauptanwendung von MAX stellt die Generierung von Softwarekomponenten zur Kapselung von CANopen-Geräten dar. Basierend auf dem allgemeinen Metamodell werden dabei die Softwarekomponenten analog zu den vorhergehenden Abschnitten automatisch erzeugt. Die Ziele bei der Generierung von Komponenten zur Steuerung von CANopen-Geräten sind die Integration von Geräten in eine allgemeine Service- und Management-Infrastruktur und die Abstraktion bzgl. der Kommunikation und der Hardware. Durch die Kapselung von CANopen-Modulen in Java-Softwarekomponenten können die Module ebenso einfach benutzt werden wie die Java Standardklassen.

5.5.1 Generierung von Java/CANopen-Komponenten

MAX unterstützt die automatische Generierung von Java-Code zur Ansteuerung von CANopen-Geräten. Aktuell werden PCs mit dem Windows-Betriebssystem, das eingebettete System TINI und das Java-Echtzeitsystem von Siemens unterstützt. Auf den genannten Systemen stehen entsprechende Treiber und Bibliotheken zur CAN-Kommunikation zur Verfügung. Andere Systeme können einfach in die bestehende Plattform eingebunden werden. Die einzige Voraussetzung hierfür ist, dass das System über eine Schnittstelle zur CAN-Kommunikation verfügt.

5.5.1.1 Generierung von Java-Klassen zur Kapselung von CANopen-Komponenten

Die XML-Metadaten bzw. das allgemeine Metamodell zur Repräsentation von CANopen-Modulen enthalten alle notwendigen Informationen zur Kapselung der Module in Software-Proxies. Zur Kapselung und Ansteuerung von CANopen-Modulen wird die bereits vorgestellte CANopen-Klassenbibliothek benutzt. Die Bibliothek umfasst u.a. Klassen, welche die Grundfunktionalität von CANopen-Modulen bereitstellen. Dabei existieren für eine Untermenge der in Tabelle 3.3 aufgeführten Geräteprofile entsprechende Java-Klassen und -Schnittstellen, wie z.B. die Klasse *CANopenDevice* oder die Schnittstellen *DigitalInputDevice* und *DigitalOutputDevice*.

Die Klasse *CANopenProvider* stellt die notwendigen Methoden zum Senden und Empfangen von SDOs, PDOs und NMTs zur Verfügung. Diese Klasse kapselt die (native) CAN-Schnittstelle und bietet eine CAN- und CANopen-Schnittstelle in Java. Zur Konvertierung von Datentypen zwischen CANopen und Java dient die Klasse *DataConverter*.

Im einem ersten Schritt wird für jedes Modul eine Java-Klasse erzeugt. Der Namen der Klasse setzt sich zusammen aus dem Schlüsselwort *Node* und dem Identifier des Gerätes. Die Klasse ist dabei von der Basis-Klasse *CANopenDevice* abgeleitet. Entsprechend des Gerätetyps eines Moduls implementiert diese Klasse eine oder mehrere Schnittstellen, wobei diese dem jeweiligen Geräteprofil entsprechen. Handelt es sich bei einem Modul z.B. um ein digitales E/A-Modul, so implementiert die Klasse die Schnittstellen *DigitalInputDevice* und *DigitalOutputDevice*.

Der erzeugte Konstruktor ruft den Konstruktor der Basisklasse auf. Der Basiskonstruktor erhält als Parameter den Identifier eines Moduls und erzeugt mit dieser ID ein SDO- und ein PDO-Objekt zur CANopen-Kommunikation.

Gemäß den Metadaten wird für jede Funktion bzw. Operation, die ein Gerät zur Verfügung stellt, eine entsprechende Methode in den Quelltext eingefügt. Die Methode kapselt dabei die CANopen-Kommunikation und die Konvertierung des Datentypen. Bei der Generierung einer Methode wird der Name zusammen mit den Parameter- und Rückgabetypen in den Quelltext eingefügt, wobei jede Methode die Ausnahme *CANopenException* deklariert. Innerhalb des Methoden-Rumpfes werden die notwendigen Anweisungen zur CANopen-Kommunikation in den Code eingefügt. Dabei werden die Parametertypen vor dem Versand bzw. die Rückgabetypen nach dem Empfang von CANopen-Nachrichten mit der Klasse *DataConverter* in die erforderlichen Datentypen konvertiert. Die Funktionalität zur CAN/CANopen-Kommunikation (*send()*, *recv()*) ist in der Basis-Klasse *CANopenDevice* implementiert. Die Klasse *DataConverter* bietet für jeden CANopen-Datentyp gemäß Tabelle B.3 eine statische Methode zur Konvertierung eines Byte-Arrays in den entsprechenden Datentyp und umgekehrt.

Eine Besonderheit bei der Erzeugung der Proxies bzw. Wrapper ist die Realisierung einer asynchronen Callback-Funktionalität. CANopen-Module können so parametriert werden, dass im Falle einer Zustandsänderung eine Nachricht verschickt wird. Ändert sich beispielsweise das Eingangssignal eines analogen oder digitalen Eingangs-Moduls, so wird von dem Gerät ein PDO mit dem aktuellen Wert des Eingangs verschickt. Ein Geräte-Proxy stellt ebenfalls diese Funktionalität zur Verfügung. Ein Objekt, das an ei-

ner Zustandsänderung eines Moduls interessiert ist, registriert sich bei einer bestimmten Komponente und implementiert die entsprechende Java-Schnittstelle. Im Falle einer Zustandsänderung wird die registrierte Instanz benachrichtigt. Der Mechanismus zur Registrierung, Verwaltung und Benachrichtigung von Instanzen ist in der Basisklasse und in den entsprechenden Geräteklassen implementiert.

5.5.1.2 Beispiel einer generierten Java/CANopen-Komponente

Listing 5.19 zeigt einen Ausschnitt einer Java/CANopen-Komponente zur Kapselung des digitalen Ein-/Ausgabe-Moduls BK5100. Das Beispiel umfasst den Konstruktor und zwei Methoden. Die Methode *getDeviceType()* liest den Objekteintrag an der Adresse mit dem Index 0x1000, SubIndex 0x0 und gibt den Wert nach der Konvertierung in den geforderten Datentyp zurück. Die zweite Methode schreibt einen als Parameter übergebenen Wert an die Adresse mit dem Index 0x6200, SubIndex 0x1 und setzt damit den Ausgang des ersten Ausgangsblocks (Ausgänge 1-8).

```
package can0;

public class Node14
extends java2canopen.CANopenDevice
implements java2canopen.DigitalInputDevice, java2canopen.DigitalOutputDevice
{
  public Node14()
  {
    super(14);
  }

  public int getDeviceType()
  throws java2canopen.CANopenException
  {
    byte[] data = null;
    data = recv(0x1000, 0x0);
    return (java2canopen.DataConverter.ByteArray2int(data));
  }

  public void setDigital8BitOutputBlocks1Digital8BitOutputBlock_0(byte p0)
  throws java2canopen.CANopenException
  {
    byte[] data = null;
    data = java2canopen.DataConverter.byte2ByteArray(p0);
    send(0x6200, 0x1, data);
    return;
  }

  ...
}
```

Listing 5.19: Beispiel einer generierten CANopen-Komponente (Ausschnitt)

Das Beispiel verdeutlicht, dass mittels der generierten Java-Klasse auf sehr einfache Art und Weise auf ein CANopen-Gerät zugegriffen werden kann. In Abschnitt 6.3.3 wird die Verwendung der generierten Java/CANopen-Komponenten an Beispielen demonstriert.

5.6 Generierung von Dokumenten und Skripten

Neben der Generierung von Software unterstützt MAX die Generierung von Skripten zur automatischen Übersetzung der erzeugten Komponenten und die Generierung von Schnittstellen-Dokumentation auf Basis der XML-Metadaten.

5.6.1 Generierung von Dokumenten

Die serialisierten XML-Metadaten werden zur dynamischen Generierung von HTML-Seiten und PDF-Dokumenten benutzt. Im Allgemeinen können unterschiedliche Sichtweisen auf Basis eines einzigen serialisierten XML-Dokumentes erzeugt werden.

5.6.1.1 Architektur

Eine statische Sicht auf die Geräte kann durch eine XSL-Transformation erzeugt werden. Wie in Abb. 5.3 dargestellt, ist ein XSLT-Prozessor [Wor99] in einen Web-Server integriert und liefert dynamisch erzeugte HTML-Seiten. Ein XSLT-Prozessor erhält als Eingabe ein Dokument basierend auf XSL, extrahiert Informationen aus einem gegebenen XML Dokument und produziert eine Ausgabe, z.B. ein HTML-Dokument.

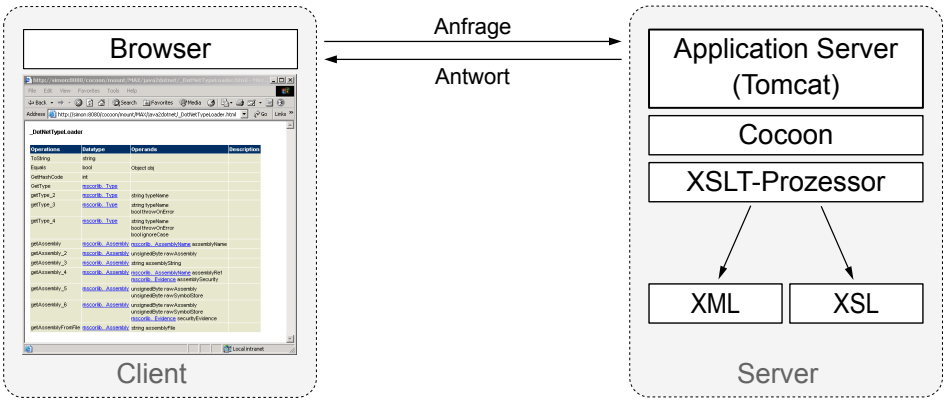


Abbildung 5.3: Architektur zur Visualisierung von Dokumenten

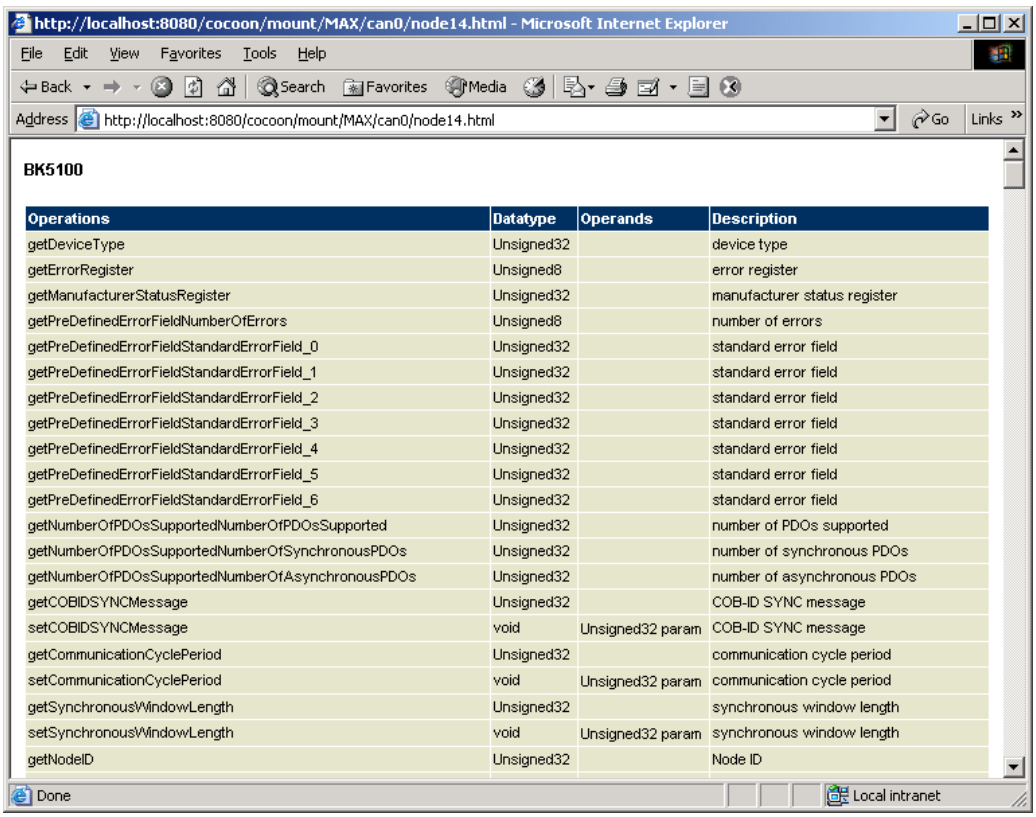
5.6.1.2 Automatische Generierung von HTML-Dokumenten

Die Ähnlichkeit von XML zu HTML legt nahe, die XML-Metadaten direkt zur Verfügung zu stellen, bzw. entsprechende HTML-Seiten für die Anzeige von Metadaten im Browser zu generieren.

Die Generierung von HTML-Seiten basierend auf den bestehenden XML-Metadaten ist mit XSLT realisiert. XSLT ist eine Query-Sprache, die Daten aus XML-Dokumenten extrahiert und entsprechende Anweisungen ausführt. Zur Erzeugung von HTML-Seiten werden demzufolge die notwendigen Metadaten aus den XML-Dokumenten extrahiert und in HTML-Seiten, die mit einem Browser betrachtet werden können, eingebettet. Die

Referenzen auf externe Typen werden als Link in die Seite eingefügt und entsprechend gekennzeichnet.

Abb. 5.4 zeigt eine tabellenorientierte Sichtweise auf ein CANopen-Modul. Dargestellt sind die zur Verfügung stehenden Operationen inkl. der Rückgabe- und Parameter-Typen und einer Beschreibung.



Operations	Datatype	Operands	Description
getDeviceType	Unsigned32		device type
getErrorRegister	Unsigned8		error register
getManufacturerStatusRegister	Unsigned32		manufacturer status register
getPreDefinedErrorFieldNumberOfErrors	Unsigned8		number of errors
getPreDefinedErrorFieldStandardErrorField_0	Unsigned32		standard error field
getPreDefinedErrorFieldStandardErrorField_1	Unsigned32		standard error field
getPreDefinedErrorFieldStandardErrorField_2	Unsigned32		standard error field
getPreDefinedErrorFieldStandardErrorField_3	Unsigned32		standard error field
getPreDefinedErrorFieldStandardErrorField_4	Unsigned32		standard error field
getPreDefinedErrorFieldStandardErrorField_5	Unsigned32		standard error field
getPreDefinedErrorFieldStandardErrorField_6	Unsigned32		standard error field
getNumberOfPDOsSupportedNumberOfPDOsSupported	Unsigned32		number of PDOs supported
getNumberOfPDOsSupportedNumberOfSynchronousPDOs	Unsigned32		number of synchronous PDOs
getNumberOfPDOsSupportedNumberOfAsynchronousPDOs	Unsigned32		number of asynchronous PDOs
getCOBIDSYNCMessage	Unsigned32		COB-ID SYNC message
setCOBIDSYNCMessage	void	Unsigned32 param	COB-ID SYNC message
getCommunicationCyclePeriod	Unsigned32		communication cycle period
setCommunicationCyclePeriod	void	Unsigned32 param	communication cycle period
getSynchronousWindowLength	Unsigned32		synchronous window length
setSynchronousWindowLength	void	Unsigned32 param	synchronous window length
getNodeID	Unsigned32		Node ID

Abbildung 5.4: Generiertes HTML-Dokument für ein CANopen-Modul

Anstatt HTML-Dokumenten ermöglicht MAX die Erzeugung beliebiger anderer Dokumente, z.B. basierend auf der *Wireless Markup Language* (*WML*). Zur Visualisierung von elektronischen Datenblättern (s. Abschnitt 3.5.1) steht ebenfalls ein XSL-Dokument zur Verfügung.

5.6.1.3 Automatische Generierung von PDF-Dokumenten

PDF-Dokumente werden analog zu HTML-Dokumenten erzeugt. Bei der Transformation und Formatierung von PDF-Dokumenten werden Anweisungen zur Formatierung von Objekten (FOP) [Apa04d] in ein XML-Dokument eingefügt. Eine Anwendung (*FOPSerializer*[10]) interpretiert diese Formatierungsanweisungen und generiert ein PDF-Dokument. Abb. 5.5 zeigt einen Ausschnitt eines generierten PDF-Dokumentes des COM-Objekts *IWebBrowser2* der COM-Komponente *InternetExplorer.Application* (*SHDocVw*).

[10]Der *FOPSerializer* ist Bestandteil von Cocoon.

SHDocVw.IWebBrowser2

Operation	Type	Operands	Description
GoBack	void		Navigates to the previous item in the history list.
GoForward	void		Navigates to the next item in the history list.
GoHome	void		Go home/start page.
GoSearch	void		Go Search Page.
Navigate	void	string URL Object[] Flags Object[] TargetFrameName Object[] PostData Object[] Headers	Navigates to a URL or file.
Refresh	void		Refresh the currently viewed page.
Refresh2	void	Object Level	Refresh the currently viewed page.
Stop	void		Stops opening a file.
Application	java2com.IDispatch		Returns the application automation object if accessible, this automation object otherwise..
Parent	java2com.IDispatch		Returns the automation object of the container/parent if one exists or this automation object.

Abbildung 5.5: Beispiel eines generierten PDF-Dokuments

Analog zu oben werden externe Referenzen durch die in Listing 5.20 gezeigte Anweisung gekennzeichnet und mit einem Link auf ein externes PDF-Dokument versehen.

```
<fo:basic-link color="..." external-destination="/{$namespace}/{$name}.pdf">
```

Listing 5.20: Formatierungsanweisung mit FOP zur Erzeugung eines Verweises

Die Generierung von Dokumenten basierend auf den serialisierten XML-Metadaten ist zur Wiederverwendung von Komponenten sehr hilfreich. Für viele COM-Komponenten existiert beispielsweise keine Dokumentation, so dass die erzeugte Dokumentation die Programmierung dieser Applikationen erheblich erleichtert.

5.6.2 Generierung von Skripten

Im Hinblick auf die Verwendung von MAX in einer dynamischen Umgebung werden zur Übersetzung der erzeugten Softwarekomponenten XML-Dokumente für das Werkzeug ANT [Apa04a] erstellt. ANT ist ein Java-basiertes Werkzeug zur Übersetzung und Archivierung (Paketierung) von Dateien. Ein Beispiel für ein generiertes ANT-Dokument ist in Listing A.4 im Anhang enthalten. Entsprechend der Konfiguration werden zudem soweit als möglich Skripte zum Start der erzeugten Komponenten generiert.

5.7 Diskussion und Bewertung

Eine mehrmalige Anwendung der Metaisierung und Generierung führt zu einer mehrmaligen Kapselung von Systemen. Beispielsweise kann zuerst eine Java-Komponente zur Kapselung einer CANopen- oder COM-Komponente erzeugt werden. Dann wird der gleiche Mechanismus auf die erzeugte Java-Komponente angewandt, um diese wiederum in eine bestimmte Middleware zu integrieren. Diese Verfahrensweise kann beliebig oft, soweit sinnvoll, wiederholt werden. Allerdings ist zu beachten, dass die generierten Komponenten den jeweiligen Anforderungen eines Zielsystems genügen.

Die Generierung von Software muss nicht allein auf die Generierung der oben gezeigten Proxies zur Kapselung von Typen oder von Schnittstellendokumentation beschränkt sein, sondern kann auch Code zur Realisierung von nicht-funktionalen Eigenschaften umfassen. Es ist beispielsweise denkbar, dass Code zur Analyse, zum Debugging, zum Tracing, zum Logging, zur Datenkomprimierung, zur sicheren Datenübertragung, zur Lastbalanzierung oder zur (Client-) Synchronisation hinzugefügt wird. Mit einer Verschlüsselung von Daten könnte z.B. eine sichere Übertragung prozesskritischer Daten analog zu Gruhler et al. [GNBK99] realisiert werden, so dass der Zugriff auf Geräte und Anlagen über Internet weitestgehend geschützt ist.

5.8 Zusammenfassung

Das allgemeine Metamodell von MAX dient als Basis für die automatische Generierung von unterschiedlichen Softwarekomponenten. Dabei werden die Ursprungskomponenten durch entsprechende Stellvertreter, die Proxies, die über eine gewisse Zusatzfunktionalität verfügen, gekapselt. In MAX liegt der Fokus auf der Generierung von (verteilter) Middleware zur Integration von existierenden Komponenten in unterschiedliche Middlewareinfrastrukturen. MAX unterstützt die Integration von Java-Komponenten in RMI, Jini und Sockets, die Integration von COM-, .NET-, CANopen-Komponenten und betriebssystemspezifischen Bibliotheken in Java.

Die Java/.NET-Middleware zur Metaisierung von .NET-Komponenten ist nahezu vollständig automatisch generiert. Die Generierung von Java-Softwarekomponenten zur Kommunikation von CANopen-Komponenten erlaubt die einfache Integration von CANopen-Systemen. Durch die objektorientierte Modellierung und Kapselung von Hardwarekomponenten in Java-Klassen wird von den Details bzgl. der Hardware und der Kommunikation abstrahiert.

Zur automatischen bzw. dynamischen Übersetzung der generierten Komponenten werden entsprechende Konfigurationsdokumente für ANT generiert. Mit der Generierung von Dokumentation im HTML- und PDF-Format liegen entsprechende Schnittstellendokumentationen der Komponenten vor, die bei Bedarf generiert und über das Web zur Verfügung gestellt werden.

Durch die Integration der genannten Systeme in Java lassen sich sämtliche Systeme miteinander kombinieren. So ist es beispielsweise möglich, die generierten Java/CANopen-Komponenten ihrerseits wiederum in eine der genannten Middlewareinfrastrukturen zu integrieren oder mit Java/COM-Komponenten zu kombinieren. Ausführliche Beispiele hierfür finden sich in Kapitel 6.

6

MAX Systemübersicht

6.1 MAX

MAX umfasst gemäß Abb. 6.1 sämtliche bisher vorgestellten Komponenten und integriert diese in eine einheitliche Infrastruktur. Den Kern des Systems bilden die Metaisierung (s. Kapitel 3) und die Generierung (s. Kapitel 5) von Softwarekomponenten, die Konvertierung (De- bzw. Serialisierung) und die Observierung von Ressourcen. Als allgemeines Format dient das Metamodell MAX bzw. dessen XML-Repräsentation MAXML.

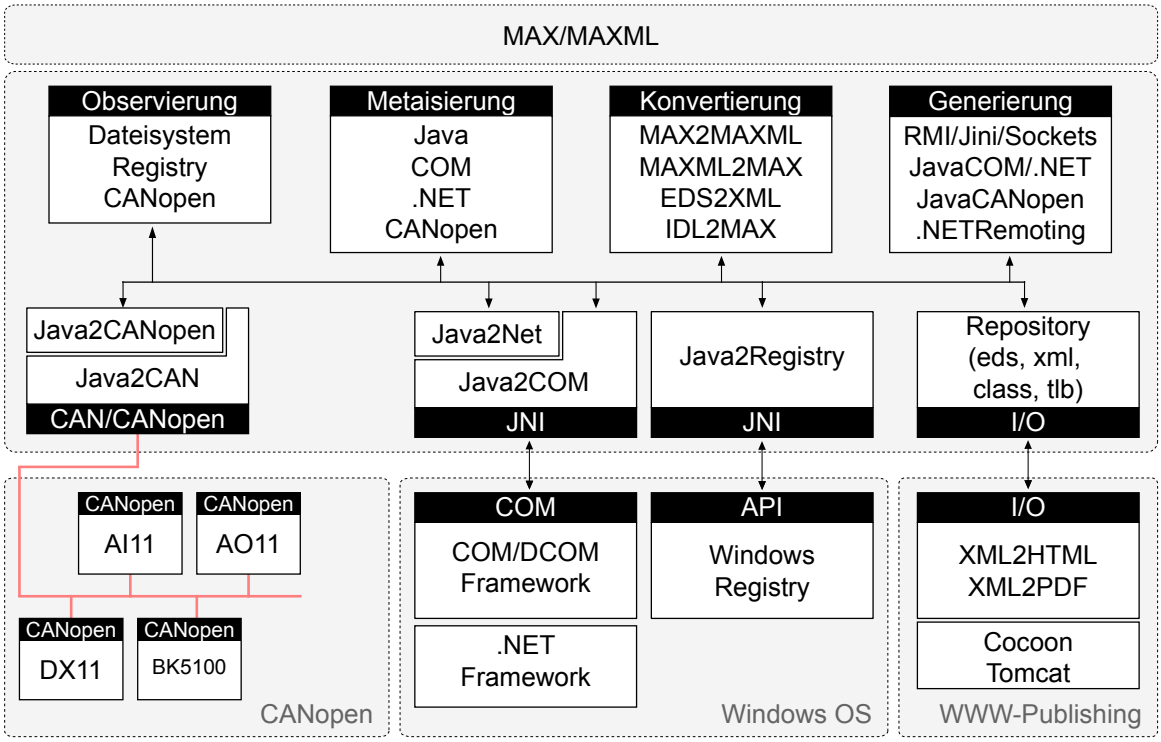


Abbildung 6.1: MAX

6.1.1 Experten

Zur Integration der unterschiedlichen Technologien sind die einzelnen Komponenten logisch in sog. Experten unterteilt. Die Basisklasse sämtlicher Experten ist in dem Klassendiagramm 6.2 dargestellt. Die Klasse *Expert* definiert einige abstrakte Methoden, die von einer abgeleiteten Klasse – einem Domänen-Experten – implementiert werden müssen.

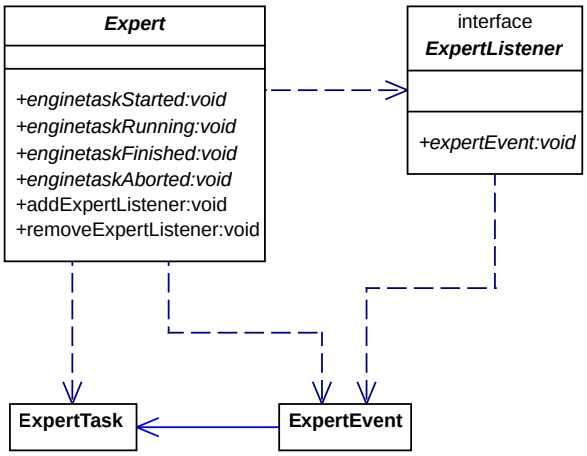


Abbildung 6.2: Die abstrakte Klasse Expert

Zur Beobachtung des aktuellen Ausführungstatus implementiert eine Klasse das Interface *ExpertListener* und registriert sich mit *addExpertListener* bei dem entsprechenden Experten. Ändert sich der Ausführungsstatus eines Experten, so wird die registrierte Instanz durch einen Callback-Mechanismus über den aktuellen Status informiert. Prinzipiell werden die Stati *started*, *running*, *aborted* und *finished* unterschieden, wobei ein einzelner Status bestimmte Zusatzinformationen liefert.

6.1.1.1 Experten zur Metaisierung

Die Experten zur Metaisierung sind, wie in dem Klassendiagramm in Abb. 6.3 dargestellt, von der Klasse ReflectionExpert abgeleitet und implementieren die o.g. Methoden zur Beobachtung des Ausführungsstatus.

Sämtliche Experten erhalten über Parameter Angaben über das zu metaisierende Artefakt. Generell können einzelne Artefakte, ganze Archive oder mehrere Artefakte innerhalb eines Archivs metaisiert werden. Der Java-Experte unterstützt die Metaisierung von Java-Klassen, -Schnittstellen und -Archiven. Der Experte zur Metaisierung von COM-Komponenten erhält als Parameter entweder den Namen einer in der Windows Registry registrierten Komponente oder den vollständigen Pfadnamen einer Typbibliothek. Eine Voraussetzung für die Metaisierung von .NET-Komponenten ist, dass die Klasse oder das Assembly im *Global Assembly Cache* (*GAC*) registriert ist. Zur Metaisierung von CANopen-Komponenten benötigt der CANopen-Experte eine Knoten-ID oder eine Kanal-ID.

Jeder der genannten Experten extrahiert die entsprechenden Metadaten und bildet diese entsprechend Kapitel 3 in das allgemeine Metamodell ab. Falls eine Metaisierung längere

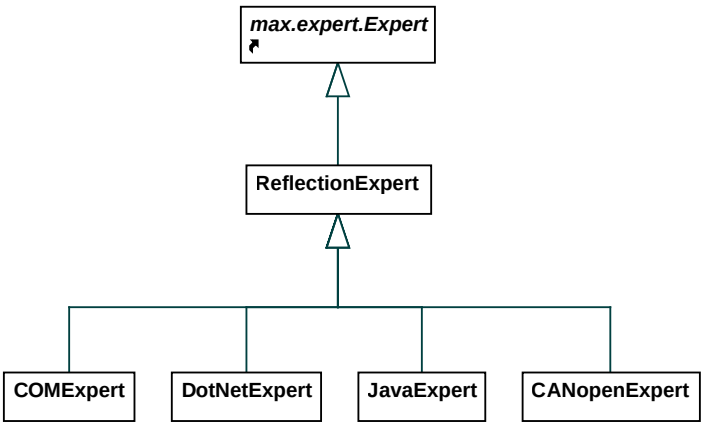


Abbildung 6.3: Die Experten zur Metaisierung

Zeit in Anspruch nimmt, wie die Metaisierung des kompletten .NET-Frameworks, so wird der aktuelle Status über den o.g. Mechanismus propagiert.

Zur Metaisierung von Komponenten über das WWW sind die Experten in eine Client-/Server-Umgebung integriert. Die entsprechenden Experten werden durch ein CGI-Skript, das in den HTTP-Server Apache [Apa04b] eingebettet ist, aktiviert. Das Skript selbst ist in Perl realisiert und erhält als Parameter den Quelltyp und den Namen der zu metaisierenden Komponente. Die folgende URL aktiviert beispielsweise den Experten zur Metaisierung von COM-Komponenten und metaisiert die COM-Komponente Excel.

```
http://127.0.0.1/cgi-bin/max2maxml?type=COM&name=Excel.Application
```

Nach der Metaisierung stehen die generierten XML-Dokumente beispielsweise zur Generierung von HTML- und PDF-Dokumenten gemäß Abschnitt 5.6 zur Verfügung.

6.1.1.2 Experten zur Konvertierung

Die Experten zur Konvertierung sind, wie in Abb. 6.4 gezeigt, von der Klasse *Conversion-Expert* abgeleitet und implementieren wiederum die abstrakten Methoden zur Beobachtung des Ausführungsstatus.

Die speicherbasierten Metamodelle werden durch den Experten zur Serialisierung nach XML konvertiert und im Dateisystem gespeichert. Umgekehrt liest der Experte zur Deserialisierung die persistenten XML-Dokumente und speichert die Metadaten in einem speicherbasierten Metamodell. Zur Konvertierung eines CANopen-EDS bietet MAX das Werkzeug EDS2XML. Der Experte zur Konvertierung von CORBA-IDL Dateien ist mit dem *Java Compiler Compiler* (*JavaCC*) [Col03] realisiert. JavaCC erhält als Eingabe eine Grammatik und erzeugt daraus einen Parser, der in der Lage ist, IDL-Dateien zu analysieren. Dieser Parser wurde im Rahmen dieser Arbeit so erweitert, dass die IDL-Metadaten in das allgemeine speicherbasierte Metamodell übertragen werden.

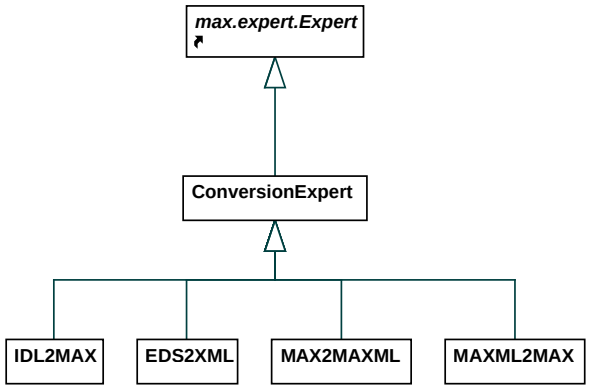


Abbildung 6.4: Die Experten zur Konvertierung

6.1.1.3 Experten zur Generierung

Sämtliche Experten gemäß Abb. 6.5 zur Generierung von Softwarekomponenten sind von der Klasse *GenerationExpert* abgeleitet.

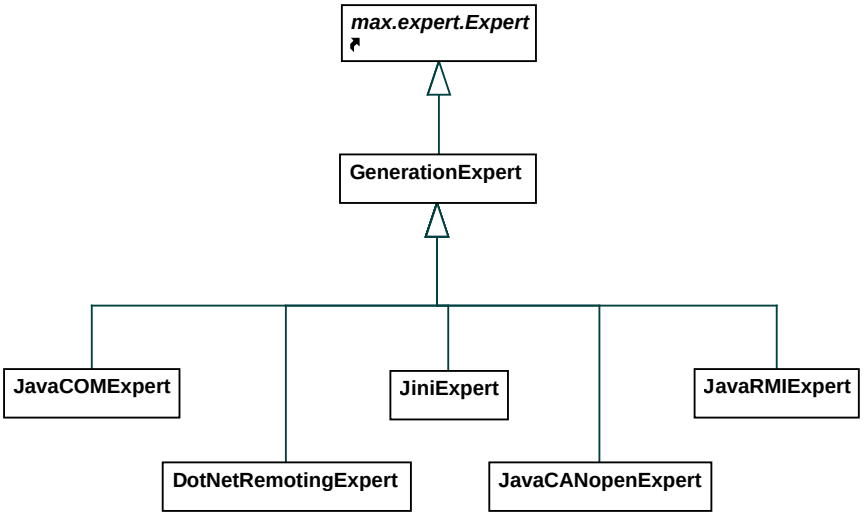


Abbildung 6.5: Die Experten zur Generierung

Die Experten zur Generierung von Softwarekomponenten erhalten als Eingabe ein speicherbasiertes Metamodell und erzeugen gemäß Kapitel 5 Java- und im Falle des .NET Remoting Experten .NET-Klassen. Darüberhinaus erzeugen alle Experten ANT-Skripte zur Übersetzung der erzeugten Dateien gemäß Abschnitt 5.6.2.

6.1.1.4 Experten zur Observierung

Zur Beobachtung von Ressourcen stehen ebenfalls Experten, sog. Observer, zur Verfügung. Dabei stehen in MAX aktuell ein Observer für das Dateisystem, für die Windows Registry und für CANopen zur Verfügung. Die Aufgabe der Observer ist die Beobachtung

von Ressourcen und die asynchrone Benachrichtigung von bei dem Observer registrierten Applikationen. Die Experten zur Observierung werden hauptsächlich zur passiven Metaisierung und Generierung von Komponenten gemäß Abschnitt 6.2.2 benötigt.

Ein Dateisystem-Observer erhält als Parameter ein zu beobachtendes Verzeichnis, das zyklisch und optional rekursiv auf neue oder modifizierte Dateien durchsucht wird. Falls eine Datei zu einem beobachteten Verzeichnis (bzw. einem Unterverzeichnis) hinzugefügt oder eine vorhandene Datei modifiziert wird, werden die Listener, die sich für dieses Ereignis registriert haben, asynchron benachrichtigt.

Die Einträge der Windows Registry können durch einen Registry-Observer beobachtet werden. Falls ein beobachteter Schlüssel geändert wird oder neue Einträge zu diesem Schlüssel hinzugefügt werden, so werden die entsprechenden Listener benachrichtigt. Bei der Installation einer COM-Komponente wird beispielsweise dem Schlüssel HKEY_CLASSES_ROOT ein Eintrag hinzugefügt. Bei einer Observierung dieses Schlüssels werden die Applikationen, die sich für diesen Schlüssel interessieren, über den neuen Eintrag informiert. Der Registry-Observer wird beispielsweise in Friedrich et al. [FNK02] zur Wartung von verteilten Systemen durch mobile Agenten genutzt.

Der CANopen-Observer identifiziert analog zu Abschnitt 3.5.5 zyklisch die an einen Bus angeschlossen CANopen-Komponenten. Falls eine neue Komponente hinzugefügt wird, so werden die Listener, die sich für dieses Ereignis interessieren, benachrichtigt.

6.1.1.5 Die CAN/CANopen API

Sämtliche in den folgenden Abschnitten vorgestellten Demonstrationen und Versuche für CAN bzw. CANopen basieren auf der objektorientierten Java CAN/CANopen API aus Abschnitt 2.5.3.2.

6.1.1.6 Die Windows-Registry API

Die Windows Registry ist der Verzeichnisdient in einem Windows Betriebssystem. In dieser Datenbank sind Einträge gespeichert, die aus Name/Wert-Paaren bestehen. In der Registry sind tausende von Einträgen, die das Windows System bzw. die installierte Software betreffen, gespeichert. Die Registry gibt z.B. Auskunft über die registrierten COM-Objekte. In MAX wird die Registry ebenfalls als Datenquelle benutzt und dient hauptsächlich der Metaisierung von COM-Objekten.

Die im Rahmen dieser Arbeit realisierte Windows Registry Reflection API ist in Java und C++ implementiert. Die Verbindung von Java und C++ ist mit JNI realisiert. Die C++-Realisierung kapselt die Windows-Schnittstelle zum Zugriff auf die Windows Registry und wird ihrerseits in Java gekapselt. Die Java-API bietet eine komfortable Möglichkeit zum Zugriff auf die Daten der Windows Registry. Zudem unterstützt diese API einen asynchronen Benachrichtigungsmechanismus, falls ein Eintrag in die Registry hinzugefügt, aus dieser gelöscht oder verändert wird.

Die Windows Registry besteht aus einer großen Anzahl von sog. Schlüsseln, die hierarchisch angeordnet sind. Ein Schlüssel besitzt einen Namen, einen Wert und eine Reihe von Einträgen. Die Einträge selbst bestehen wiederum aus einem Namen und einem

Wert.

Die Klasse *Registry* kapselt Methoden zum Zugriff auf die Windows Registry. Mit der Methode *openRegistryKey()* wird ein Schlüssel, der durch die Klasse *RegistryKey* repräsentiert wird, geöffnet. Die Klasse *RegistryKey* bietet Methoden zum Lesen und Schreiben von Name/Wert-Paaren. Zur asynchronen Benachrichtigung kann sich eine Instanz bei einem gegebenen Schlüssel registrieren. Falls der Schlüssel bzw. ein Schlüssel unterhalb dieses Schlüssels modifiziert wird, werden die registrierten Instanzen über diese Änderung informiert. Dieser Mechanismus ist auf Basis des Publish/Subscribe-Pattern realisiert. Die Daten eines Name/Wert-Paares können mit der Klasse *NameValuePair* gelesen bzw. geändert werden.

Die Windows Registry Reflection API bietet eine komfortable Möglichkeit zur Extraktion von Metadaten aus der Windows-Registry. Mit dieser API können Einträge gelesen, hinzugefügt, verändert oder auf Modifikationen beobachtet werden.

6.2 Interaktion mit MAX

Zur Interaktion mit dem in dieser Arbeit realisierten System MAX stehen eine interaktive und eine passive Schnittstelle zur Verfügung. Die interaktive Schnittstelle erlaubt den Zugriff auf die Systemfunktionalität über eine graphische Benutzerschnittstelle. Die passive Schnittstelle bietet eine Möglichkeit zur Nutzung des Systems ohne die Interaktion eines Benutzers.

6.2.1 Interaktive Schnittstelle

Die Funktionalität des Systems, d.h. die Reflexion bzw. Metaisierung, die Serialisierung und die Generierung von Komponenten kann über eine interaktive Schnittstelle bedient werden. Die interaktive Schnittstelle ist in die Entwicklungsumgebung Eclipse [IBM04] integriert. Eclipse bietet über das Konzept der sog. *Plugins* eine Möglichkeit, die Entwicklungsumgebung auf eigene Bedürfnisse anzupassen bzw. um eigene Funktionalität zu erweitern.

Eine Möglichkeit der Erweiterung umfasst die Realisierung von Wizards, die immer wiederkehrende Aufgaben weitestgehend automatisieren und dem Bediener eine Möglichkeit zur Parametrierung eines bestimmten Vorgangs über Dialoge ermöglichen. Zur Einbindung von MAX in Eclipse stehen deshalb ein Wizard zur Metaisierung und ein Wizard zur Generierung von Komponenten zur Verfügung, die im Rahmen dieser Arbeit implementiert wurden.

6.2.1.1 Eclipse-Wizard zur Metaisierung von Komponenten

Der Wizard zur Metaisierung von Komponenten besteht aus mehreren Dialogen und führt den Benutzer durch die einzelnen Schritte, die zur Metaisierung einer Komponente erforderlich sind. Zum Starten des Wizard wird in Eclipse über das Menü File > New > Other ein Dialogfeld geöffnet und der Reflection Wizard, wie in Abb. 6.6 dargestellt,

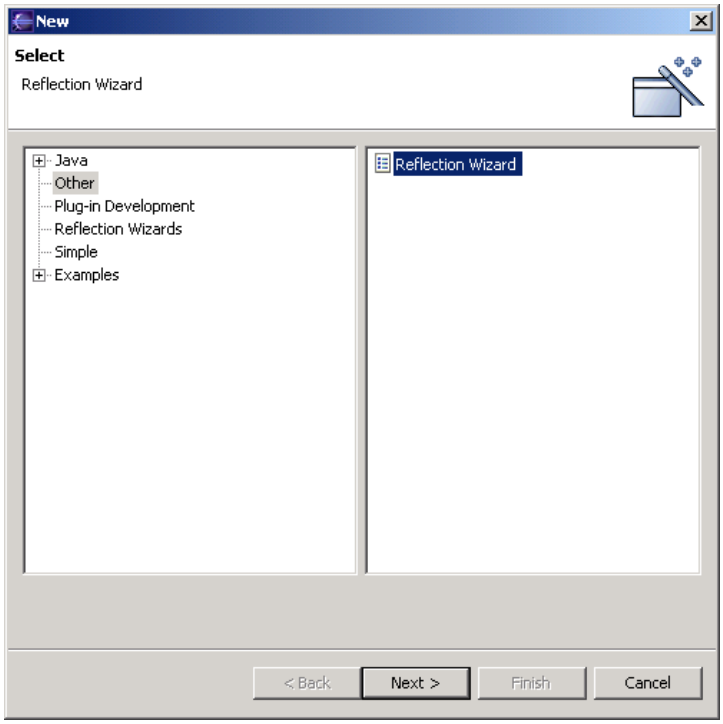


Abbildung 6.6: Start des Reflection Wizard

ausgewählt.

In einem ersten Schritt wird der Typ der Komponente bestimmt. In dem in Abb. 6.7 gezeigten Dialog besteht die Auswahl zwischen Java, COM, .NET, CANopen und IDL.

Entsprechend des gewählten Typs wird in einem zweiten Dialog die Quelle der Komponente bestimmt. Als Beispiel wird im Folgenden die Metaisierung einer COM-Komponente gezeigt[1]. Dazu ist es notwendig, die COM-Komponente *DotNetTypeLoader* und die Komponente, die das Microsoft .NET-Framework repräsentiert, in Java zu kapseln. Die Schnittstellenbeschreibungen des .NET-TypeLoader und der .NET-Datentypen liegen in Form einer Typbibliothek (java2dotnet.tlb bzw. mscorlib.tlb) vor. Diese Typbibliotheken und die von diesen referenzierte Bibliotheken (stdole2.tlb) werden durch MAX metaisiert und nach XML serialisiert.

Nach Auswahl des Typs COM werden die zur Metaisierung in Frage kommenden Quellen aufbereitet. Bei COM können entweder Typbibliotheken aus dem Dateisystem geladen oder Komponenten, die in der Windows Registry registriert sind, metaisiert werden. Demzufolge besteht bei dem in Abb. 6.8 gezeigten Dialog die Auswahl zwischen Typbibliothek und Windows Registry. Im Falle der Typbibliothek wird ein Dateidialog geöffnet, so dass die entsprechende Bibliothek ausgewählt werden kann. Soll eine in der Registry registrierte Komponente metaisiert werden, so kann diese Komponente aus der gezeigten Liste ausgewählt werden.

In einem nächsten Schritt werden die Typen, die durch die Komponente definiert wer-

[1]Wie bereits in den Abschnitten 3.4 und 5.4.2 erläutert, ist der Experte zur Metaisierung von .NET-Komponenten automatisch generiert.

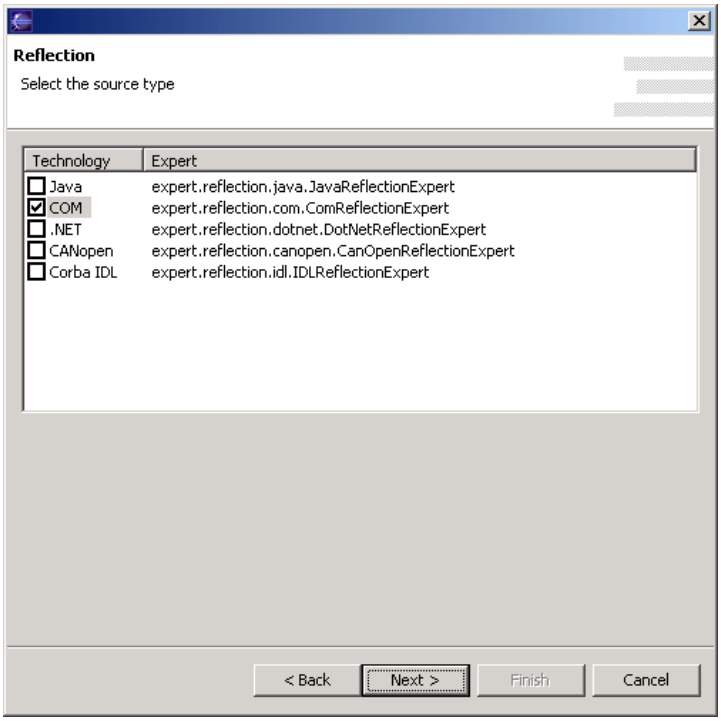


Abbildung 6.7: Bestimmung des Ursprungstyps

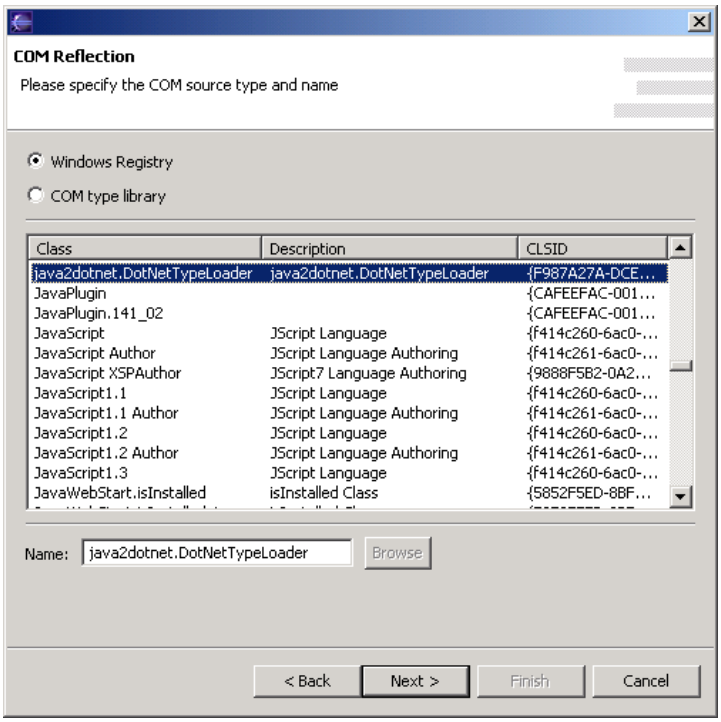


Abbildung 6.8: Auswahl der COM-Komponente

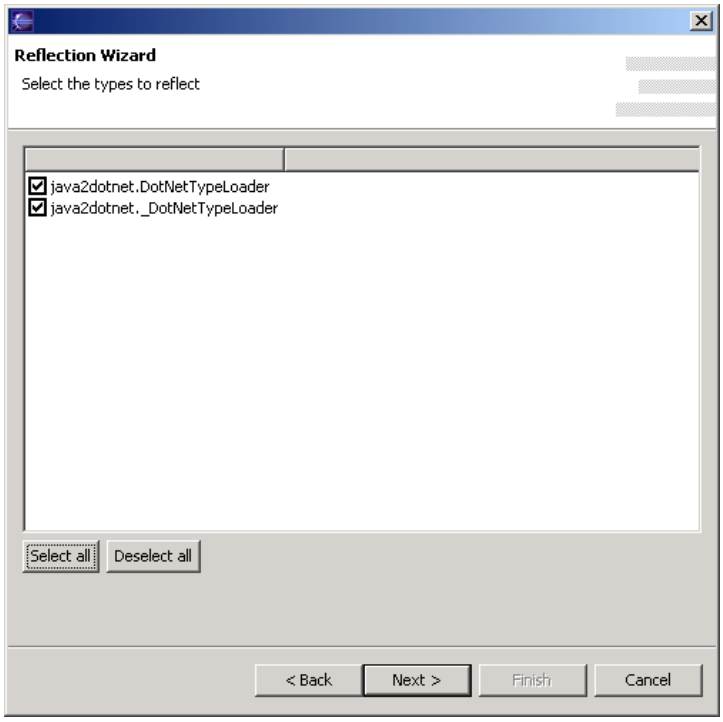


Abbildung 6.9: Anzeige der von einer Komponente definierten Datentypen

den, extrahiert und in einer Liste angezeigt. Die zu metaisierenden Typen können, wie in Abb. 6.9 dargestellt, wahlweise einzeln oder alle selektiert werden.

Zum Abschluss werden die Projekt-Parameter für das zu erstellende Eclipse-Projekt gemäß Abb. 6.10 festgelegt. Hierzu zählen der Namen und das Wurzelverzeichnis des zu erzeugenden Projektes und die Namen der Ordner relativ zu dem Wurzelverzeichnis, in dem die zu erzeugenden Dateien bzw. Klassen gespeichert werden.

Nachdem sämtliche Informationen für die Metaisierung bekannt sind, wird ein Java-Projekt mit jeweils einem Ordner für die XML-Metadaten (z.B. xml), die Quellen (z.B. src) und die übersetzten Klassen (z.B. bin) erzeugt. Die im vorigen Schritt selektierten Typen werden metaisiert, nach XML serialisiert und in dem entsprechenden XML-Ordner abgelegt. Der aktuelle Verarbeitungsschritt wird, wie in Abb. 6.11 gezeigt, protokolliert. Diese Funktionalität ist mit dem Callback-Mechanismus gemäß Abschnitt 6.1.1 realisiert.

Nach erfolgreicher Beendigung der Metaisierung liegt ein Projekt gemäß Abb. 6.12 in Eclipse mit den entsprechenden Ordnern und den XML-Dokumenten, die mittels einem XML-Editor (z.B. XMen) betrachtet und editiert werden können, vor.

Mit dem Eclipse-Reflection-Wizard steht ein komfortables Werkzeug zur Metaisierung von Komponenten mit einer einfachen Benutzerführung zur Verfügung.

6.2.1.2 Eclipse-Wizard zur Generierung von Komponenten

Der Wizard zur Generierung von Komponenten verbirgt sich in einem Popup-Menü, das über den Menü-Baum des Projektes aktiviert wird. In diesem Popup-Menü spezifiziert

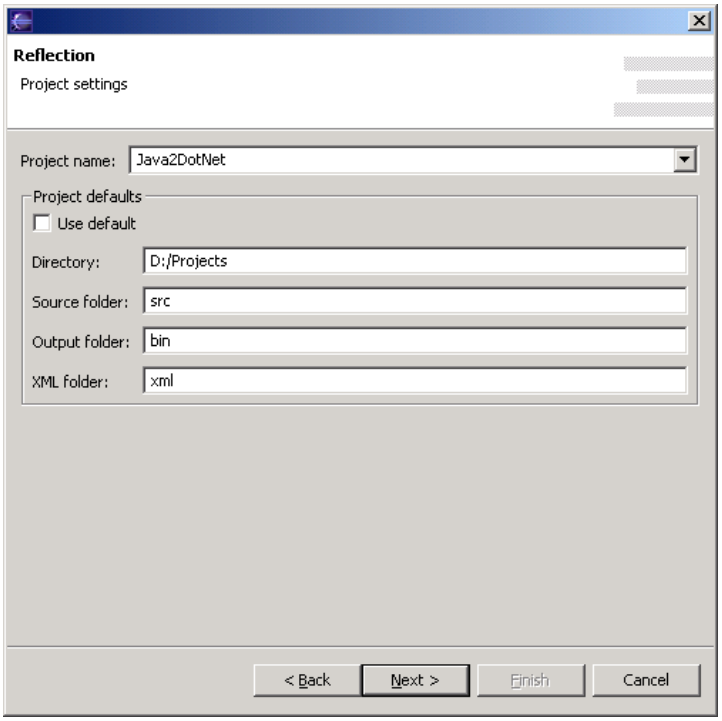


Abbildung 6.10: Konfiguration der Projekt-Einstellungen

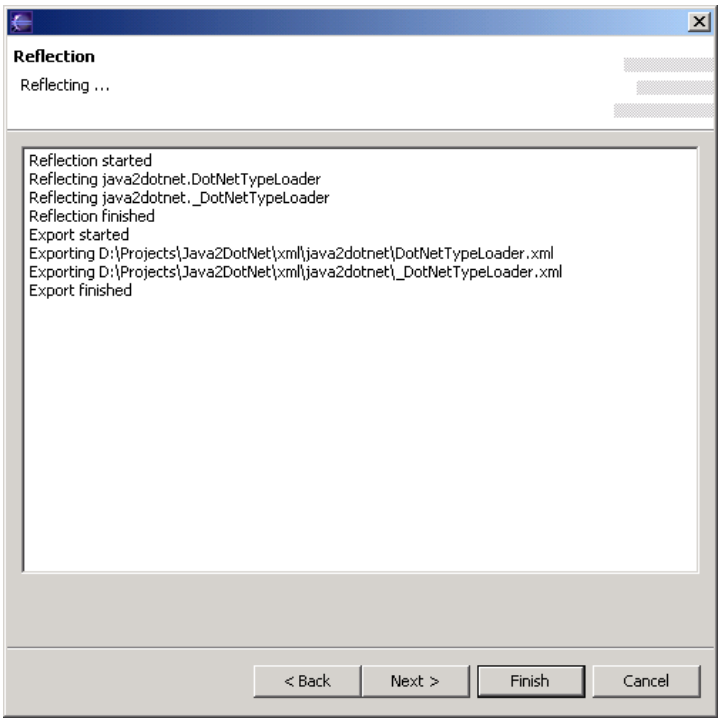


Abbildung 6.11: Protokollierung der Wizard-Aktionen

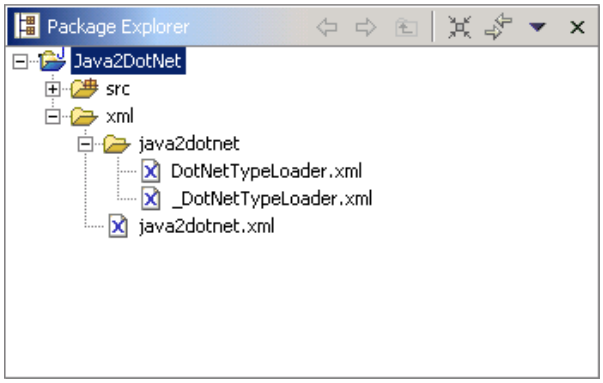


Abbildung 6.12: Erstelltes Projekt in Eclipse

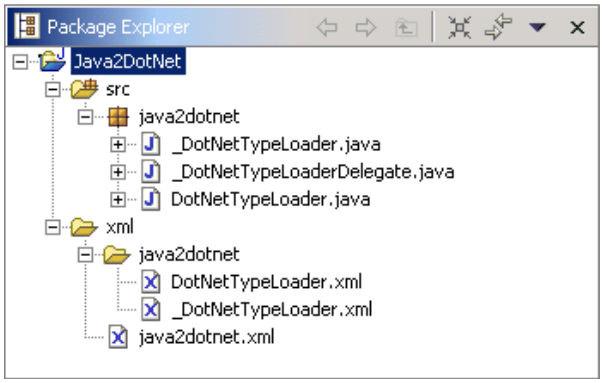


Abbildung 6.13: Projekt mit generierten Java-Klassen

ein Benutzer, welche Komponenten erzeugt werden sollen. Entsprechend des Quelltyps steht dabei die Generierung der in Kapitel 5 vorgestellten Komponenten zur Verfügung. So können z.B. für Java-Komponenten RMI- bzw. Jini-Komponenten oder für eine COM-eine Java-Komponente zur Kapselung der COM-Komponente in Java automatisch generiert werden. Gemäß Abb. 6.13 werden die erzeugten Klassen dabei in dem Ordner für die Quelldateien abgelegt. Mit Eclipse können diese Klassen übersetzt und zur Ausführung gebracht werden. Die notwendigen Einstellungen zur weiteren Verarbeitung werden zwar größtenteils automatisch vorgenommen, evtl. sind jedoch noch Einstellungen, wie. z.B. die Konfiguration von Klassenpfaden, erforderlich.

Durch die Integration in Eclipse kann der generierte Code bearbeitet, übersetzt und zur Ausführung gebracht werden. Darüber hinaus unterstützen Eclipse-spezifische Erweiterungen, beispielsweise die Komplettierung von Code oder die kontextsensitive Hilfe, die Programmierung erheblich.

6.2.2 Passive Schnittstelle

Im Gegensatz zur interaktiven Schnittstelle bietet die passive Schnittstelle eine Metaisierung und Generierung ohne Benutzerinteraktion.

6.2.2.1 Passive Metaisierung

Zur passiven Metaisierung von Komponenten existiert eine Java-Applikation, die sich bei den unter Abschnitt 6.1.1.4 genannten Observern als Listener registriert. Falls ein analog zu 6.1.1.4 genanntes Ereignis auftritt, so wird die hinzugefügte bzw. modifizierte Komponente durch entsprechende Experten metaisiert und serialisiert. Wird z.B. eine COM-Typbibliothek in einem zuvor spezifizierten Verzeichnis, das von einem Datei-Observer beobachtet wird, abgelegt, so wird der entsprechende Experte aktiviert. Die serialisierten Metabeschreibungen werden entsprechend ihres Typs im Dateisystem abgelegt und stehen damit zur weiteren Verarbeitung, z.B. zur passiven Generierung von Softwarekomponenten, zur Verfügung.

6.2.2.2 Passive Generierung

Analog zur passiven Metaisierung von Komponenten, werden gegebene Verzeichnisse (z.B. Java2COM, RMI, Jini, .NET, CANopen) von einem Dateisystem-Observer beobachtet. Wird beispielsweise eine metaisierte und serialisierte Beschreibung einer COM-Komponente in einem bestimmten Verzeichnis (z.B. Java2COM) zur Generierung von Java2COM-Komponenten gespeichert, so werden automatisch die entsprechenden Java-Klassen, die diese COM-Komponente kapseln, generiert und mittels ANT übersetzt.

6.2.3 Fazit

Mit der interaktiven und der passiven Schnittstelle stehen komfortable Mechanismen zur Metaisierung und Generierung von Komponenten zur Verfügung. Während bei der interaktiven Schnittstelle der Benutzer im Vordergrund steht, werden bei der passiven Schnittstelle Komponenten automatisch metaisiert und generiert.

Die passive Schnittstelle eignet sich u.a. für dynamische Umgebungen, wie z.B. für mobile Software-Agenten [FTNK04]. Für diesen Fall soll beispielhaft ein Anwendungsszenario gegeben werden. Angenommen, ein mobiler Agent besitzt den Auftrag ein Word-Dokument in ein PDF-Dokument zu konvertieren. Eine notwendige Voraussetzung für diesen Prozess ist die Verfügbarkeit von Anwendungen, die zum einen Word-Dokumente interpretieren und zum anderen PDF-Dokumente erzeugen können. Ein Agent kann mittels dieser Information eine Laufzeitumgebung ermitteln, die beide Bedingungen erfüllt. Zur Kommunikation mit der entsprechenden Anwendung stehen dem Agenten das COM-Objekt *Word.Application* als Java-Komponente zur Verfügung. Verfügt der Agent über das notwendige 'Wissen', eine Word-Datei zu laden und als PDF-Dokument zu speichern, so kann er mit dem resultierenden PDF-Dokument zu seinem Ausgangspunkt zurückkehren. In Friedrich et al. [FTNK04] wurde MAX zur Generierung von Java-Komponenten zur Kapselung der COM-Komponente Word für die Suche in Word-Dokumenten genutzt. Die erforderlichen Klassen wurden aus Performanzgründen bereits zur Entwicklungszeit generiert.

6.3 Integration von Systemen, Anwendungen und Szenarien mit MAX

Im Folgenden soll die Vorgehensweise zur Integration von Systemen mit MAX anhand einiger Beispiele veranschaulicht werden. Die Beispiele lehnen sich an die Szenarien des Projektes Verbund Virtuelles Labor, Teilprojekt Informatik Virtueller Systeme, an [GKNB99, BKGN00, Gru01, NBGK01, GKBN01]. Im Rahmen dieses Projektes wurden unterschiedliche Geräte und Anlagen an das Internet angebunden, um die Möglichkeiten einer Internet-basierten Lehr- und Lernumgebung zu evaluieren.

6.3.1 Internet-Kamera

Die steuerbare Internet-Kamera EVI-D31 der Firma Sony dient zur Beobachtung von Geräten in einem entfernten Labor. Die Kamera ist an einen Standard-PC mit einer Framegrabber-Karte angeschlossen. Zur Übertragung der Live-Bilder über Internet dient eine kommerzielle Software. Die steuerbare Kamera verfügt über einen Schwenk-/Neigekopf, einen optischen Zoom und eine serielle Schnittstelle. Zur Kommunikation mit der Kamera steht ein serielles Kommunikationsprotokoll namens VISCA zur Verfügung. Ein VISCA-Frame setzt sich, wie in Abb. 6.14 dargestellt, aus mehreren Bytes zusammen. Jede Nachricht umfasst einen Präfix, ein Befehl beliebiger Länge und einen Suffix. Die in Abb. 6.14 dargestellte Nachricht schaltet beispielsweise die Kamera ein.

Präfix	Befehl				Suffix
0x81	0x01	0x04	0x00	0x02	0xFF

Abbildung 6.14: VISCA-Frame

Die im Rahmen dieser Arbeit realisierte Java-Klasse Cam bildet die vollständige Funktionalität der Kamera auf eine entsprechende Java-Klasse ab. Zur Kommunikation mit der seriellen Schnittstelle benutzt die Klasse die Java Communications API, kurz JavaComm. Dieses Paket realisiert den Zugriff auf serielle und parallele Schnittstellen und ist für unterschiedliche Plattformen verfügbar.

Zur Realisierung eines entfernten Zugriff gemäß Abb. 6.15 auf die Kamera-Funktionalität wird die Klasse Cam mit MAX metaisiert und serialisiert.

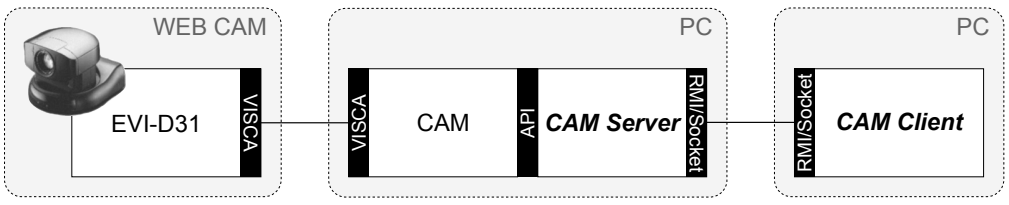


Abbildung 6.15: Steuerbare Internet-Kamera

Auf Basis der serialisierten Metadaten wird die notwendige Schnittstelle zur entfernten Kommunikation via Java-RMI oder Sockets mit MAX automatisch generiert. Eine Synchronisation von Clients zum sequentiellen Zugriff steht bei der generierten Software nicht zur Verfügung, kann jedoch durch eine entsprechende Basisklasse oder zusätzlichen Code realisiert werden.

6.3.2 Steuerung eines kartesischen Portals

Das kartesische Handhabungssystem, besteht aus zwei frei positionierbaren X,Y-Achsen und einer binären Z-Achse mit einem Sauggreifer, wobei sämtliche Achsen pneumatisch geregelt werden. Die Regelung des Systems übernimmt eine programmierbare Steuerung mit der Bezeichnung SPC200.

Die Steuerung verfügt über zwei digitale Ein-/Ausgangs-Module und eine serielle Schnittstelle zum Datenaustausch mit einem Server. Die digitalen Ein-/Ausgänge werden z.B. zur Steuerung der Z-Achse und des Sauggreifers am unteren Ende der Z-Achse benutzt. Zur Steuerung der Achsen werden Ablaufprogramme nach DIN66025 und Steuerungskommandos vom Server über die serielle Schnittstelle zur Steuerung übertragen. Die Ausführung der Programme einschließlich der Regelung der Achsen wird von der Steuerung selbst übernommen. Da die Steuerung keine bahngesteuerte, sondern nur eine punktgesteuerte Bewegung unterstützt, werden nur Positions-Koordinaten an die Steuerung übergeben. Die Bewegungsbahn zwischen zwei Koordinaten kann bis auf einige Parameter, z.B. die Beschleunigung, die Verzögerung und die Positioniergüte, nicht beeinflusst werden. Zur Bestimmung der X-,Y-Koordinaten sind an den Achsen Potentiometer angebracht. Die aktuellen Potentiometerwerte werden permanent an eine analoge E/A-Karte mit der Bezeichnung PCI-20428W übertragen. Damit ist es möglich, die aktuelle Position der Achsen und somit die Bewegungsbahn zu bestimmen und zu protokollieren. Zur Integration der Karte ist eine Windows-Bibliothek und entsprechende Dokumentation verfügbar.

Die Steuerung des Portals, d.h. die Übertragung von Nachrichten über die serielle Schnittstelle an die Steuerung, ist mit einer Java-Klasse namens *SPC200* realisiert. Analog zur steuerbaren Kamera (s. Abschnitt 6.3.1) wird die serielle Schnittstelle über die JavaComm API angesprochen. Zur Bestimmung der X-,Y-Koordinaten ist die E/A-Karte bzw. die entsprechende Windows-Bibliothek mit JNI gekapselt.

In diesem Beispiel sind zwei Integrationsschritte notwendig. Zum einen soll es möglich sein, über Internet auf die Anlage zuzugreifen. Zum anderen soll ein zusätzlicher Programmieraufwand zur Integration der Karte mit JNI vermieden werden. Das resultierende Anwendungsszenario ist in Abb. 6.16 dargestellt.

Gemäß Abschnitt 5.4.3 werden die notwendigen Dateien und Klassen zur Integration der Bibliothek auf Basis einer manuellen Serialisierung analog zu Abschnitt 4.2.7 automatisch generiert. Das Resultat der Generierung ist eine Java-Klasse, die über JNI eine generierte Windows-Bibliothek kapselt. Die generierte Bibliothek kapselt wiederum die Bibliothek zur Kommunikation mit der E/A-Karte. Der entfernte Zugriff auf die Anlage wird analog zu Abschnitt 6.3.1 durch die Metaisierung und Serialisierung der Klasse SPC200 und der Generierung von entsprechender Middleware mittels MAX realisiert.

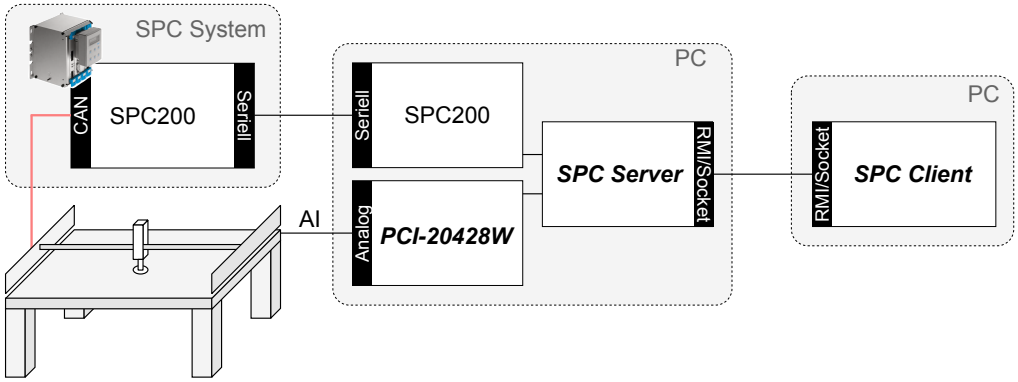


Abbildung 6.16: Steuerbares XY-Portal

6.3.3 Integration von CANopen-Komponenten

Im Folgenden wird die Integration von CANopen-Komponenten in Java und die Integration dieser Java-Komponenten in Jini bzw. in eine Client/Server-Umgebung vorgestellt.

6.3.3.1 Integration in Java

Die Programmierung einer Java-Klasse zur Steuerung eines CANopen-Moduls gestaltet sich sehr einfach, da die Gerätefunktionalität in Form einer Java-Klasse mit entsprechenden Methoden zum Lesen und Setzen von internen Geräteparametern gekapselt ist. Die CANopen-Kommunikation ist vollkommen transparent. Der asynchrone Benachrichtigungsmechnismus erlaubt darüber hinaus eine elegante Möglichkeit, um auf Änderungen eines Gerätestatus zu reagieren.

Listing 6.1 zeigt eine prototypische Anwendung zur Ansteuerung eines CANopen-Moduls in Java. Nach der Erzeugung einer Instanz kann mit den entsprechenden Methoden auf das Modul zugegriffen werden. Die zugrunde liegende CANopen-Kommunikation ist hierbei für einen Benutzer vollkommen transparent. Nach erfolgter Registrierung wird im Falle einer Änderung des digitalen Eingangssignals die Callback-Methode *inputChanged()* aufgerufen.

```
public class CANopenApplication
implements DigitalInputListener
{
  can0.Node14 node;

  public CANopenApplication()
  {
    /* Erzeugung einer Instanz */
    node = new can0.Node14();

    /* Abfrage des Gerätetyps */
    node.getDeviceType();

    /* Registrierung für asynchrone Benachrichtigung */
    node.addDigitalInputListener(this);
  }

  /* callback */
```

```
    public void inputChanged() {
20      ...
    }
}
```

Listing 6.1: Beispiel einer CANopen-Applikation

Mit der Kapselung von CANopen-Komponenten in Java kann sehr einfach auf die entsprechenden Geräte bzw. auf deren Funktionalität zugegriffen werden. Wie in nachfolgendem Abschnitt deutlich wird, besteht darüber hinaus die Möglichkeit, die generierten Java-Komponenten ihrerseits in entsprechende Java-Infrastrukturen einzubetten.

6.3.3.2 Integration in Jini

Das Ziel bei der Integration von CANopen-Komponenten ist die Benutzung von CANopen-Komponenten in einer verteilten Service- und Management-Infrastruktur. Dazu soll zunächst ein Überblick über die Technologie gegeben werden.

Eine Sammlung von Diensten in einem Netzwerk wird als *Jini Community* oder auch *Djinn* bezeichnet. Die Zentrale einer Community ist der *Jini Lookup Service* [Sun99d]. Der Lookup Service selbst ist wiederum ein Jini Service und dient als Repository für Services. Der Jini Lookup Service Browser ist Teil der Jini Implementierung von Sun Microsystems und dient zur Darstellung und Verwaltung der aktuell registrierten Dienste.

Die Jini Discovery und Join Protokolle [Sun99c] erlauben es Services, den Lookup Service zu lokalisieren (discover) und sich zu registrieren (join). Während des Registrierungsprozesses werden ein Service-Objekt bzw. ein Service-Proxy und optional einige Attribute beim Lookup Service gespeichert. Eine Instanz, die einen bestimmten Service benutzen möchte, kontaktiert zuerst einen Lookup Service und fragt sämtliche Dienste an, die bestimmte Eigenschaften besitzen. Falls ein solcher Service existiert, wird das Service-Objekt an die anfragende Instanz übergeben. „Dies gibt Jini the Fähigkeit, ohne Installation eines Treibers, Services und Geräte, zu benutzen“ [Edw99].

Da die Fähigkeit, dynamisch Code zu laden und auszuführen, Teil der Jini Spezifikation ist, wird angenommen, dass jedes Jini-fähige Gerät über Speicher und Rechenleistung verfügt [Sun99a]. Dies impliziert, dass Geräte, die in einer Jini Community partizipieren möchten, ausreichend Speicher und Rechenkapazität zur Verfügung stellen, um eine JVM auszuführen und Java-Klassen zu speichern.

Die *Jini Device Architecture Specification* [Sun99b] beschreibt drei Möglichkeiten, einen Jini Service mit einer bestimmten Hardware zu implementieren.

- Die erste Möglichkeit beschreibt Hardware mit einer JVM mit allen Ressourcen, die notwendig sind, in einer Jini Community zu partizipieren. Diese Möglichkeit resultiert allerdings in hohen Kosten bzgl. Hard- und Software, da ein Mikroprozessor, Speicher und eine entsprechende Implementierung einer JVM vorhanden sein muss.

- Die zweite Möglichkeit betrachtet Geräte mit einer speziellen virtuellen Maschine.

Die Idee ist, nur die Funktionalität zu implementieren, die notwendig ist, um mit dem Lookup Service zu interagieren.

- Eine dritte Möglichkeit erlaubt mehreren Geräten, eine einzelne JVM als Proxy zur Jini Community zu teilen. Bei diesem Ansatz ist eine Gruppe von Geräten mit einem zusätzlichen Gerät physikalisch oder über ein Netzwerk verbunden. Ein solches Gerät wird als *Device Bay* bezeichnet. Eine Device Bay verfügt über einen Prozessor, Speicher und eine JVM. Registriert sich ein Gerät bei der Device Bay, teilt ein Gerät der Bay mit, wo der Java Code gespeichert ist, um das Gerät zu benutzen. Während der Registrierung wird dieser Code zum Lookup Service übertragen. Das Protokoll zur Kommunikation zwischen den Geräten und der Device Bay ist dabei nicht spezifiziert. Mit diesem Ansatz benötigen die Geräte zwar keine zusätzliche Hardware, aber es muss ein zusätzlicher Proxy existieren.

Eine direkte Anbindung der CANopen-Module an Jini stellt aus mehreren Gründen keine Alternative dar. Gemäß Abschnitt 2.3.3.3 ist Jini auf anderen Kommunikationsmedien bzw. -protokollen nur mit erheblichem Aufwand zu realisieren. Zudem verfügen die Module über keinen Prozessor, so dass keine Programme ausgeführt werden können. Deshalb wird der dritte Ansatz, eine Anbindung mit einem Proxy gemäß Nusser et al. [NG00], favorisiert. Bezüglich dieser Anbindung ist jedoch zu bedenken, dass es aus Kostengründen nicht sinnvoll ist, als Proxy einen Standard-PC zu verwenden. Aus diesem Grund kommt das kostengünstige eingebettete System TINI [Dal04] zum Einsatz. Dieses System verfügt über einen Prozessor, Speicher (je nach Ausführung 512 kB bzw. 1 MB) und eine JVM. Das System besitzt u.a. eine Ethernet- und eine CAN-Schnittstelle. Da die eingebettete JVM von TINI selbst kein RMI unterstützt, wird ein weiterer Proxy benötigt, der eine Verbindung zu einer Jini Community herstellen kann. Dieser Proxy ist geographisch ungebunden und kann mehrere TINI-Systeme verwalten. Die Architektur zur Anbindung von CANopen-Modulen an Jini ist in Abb. 6.17 dargestellt.

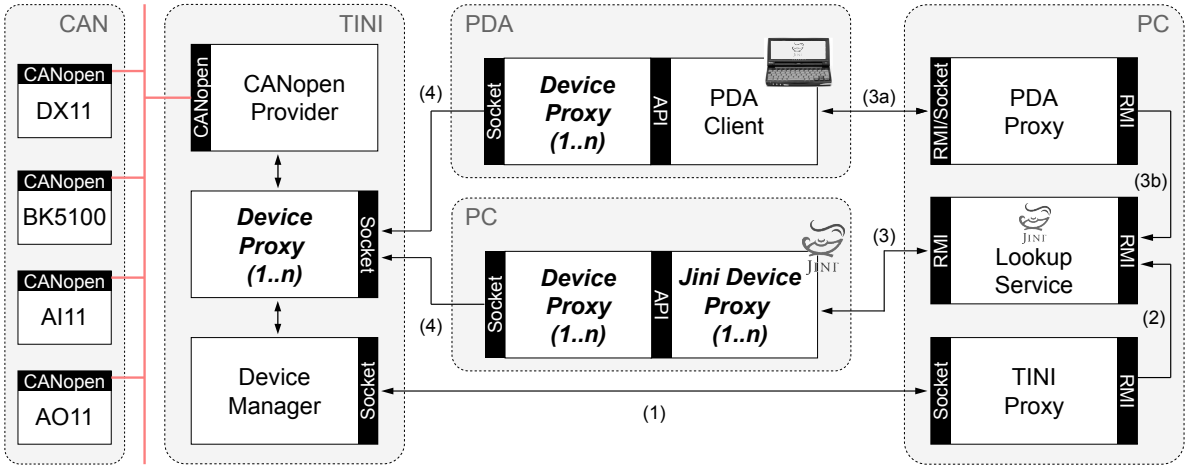


Abbildung 6.17: Anbindung von CANopen-Modulen an Jini

Zur Integration von CANopen-Komponenten in Jini werden entsprechend Abschnitt 5.5.1 Java-Komponenten zur Kapselung der CANopen-Komoponenten in Java und

für diese Java-Komponenten wiederum Jini-Komponenten zur Kapselung der Java-Komponenten in Jini generiert. Das Resultat sind Java-Proxies, die über Jini-Funktionalität verfügen. Abb. 6.18 zeigt den Jini Lookup Browser nach der Registrierung der vorhandenen Geräteproxies.

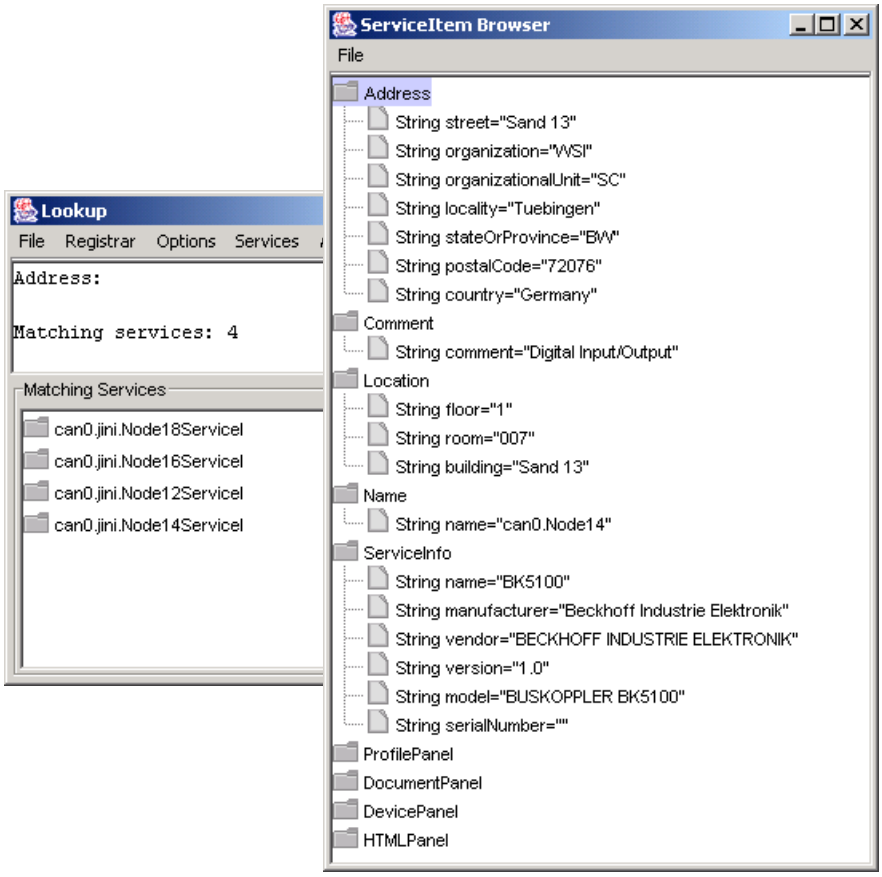


Abbildung 6.18: Jini Geräteproxies

Gemäß Abschnitt 2.3.3.2 unterstützt das eingebettete System TINI bzw. die JVM über kein RMI, so dass nur Sockets zur verteilten Kommunikation zur Verfügung stehen. Aus diesem Grund werden für die Java/CANopen-Komponenten zusätzliche Geräteproxies (Client- und Server) analog zu Abschnitt 5.3.4 generiert und zusammen mit dem Jini-Service-Proxy beim Jini Lookup Service registriert. Ein Jini-fähiger Client fordert den entsprechenden Dienst an und erhält als Antwort ein Jini-Service-Proxy und ein Geräteproxy. Der Geräteproxy dient dabei zur Kommunikation mit einem Geräteproxy auf dem eingebetteten System gemäß Abb. 6.17.

Zusätzlich zum Geräteproxy werden mehrere GUI-Klassen, sog. Panels, registriert, die unterschiedliche Sichtweisen auf ein Gerät bzw. auf die zu einem Gerät zugehörige Informationen bieten. Die Panels *Profile*, *Document*, *HTML* und *Device* dienen zur Anzeige von Geräteprofilen, Schnittstellendokumentation gemäß Abschnitt 5.6, HTML-Dokumenten und zum Zugriff auf CANopen-Komponenten. In Abb. 6.19 ist ein Jini-Client abgebildet, der die vorhandenen Geräte (bzw. -proxies) und beispielhaft das elektronische Datenblatt des CANopen-Moduls BK5100 anzeigt.

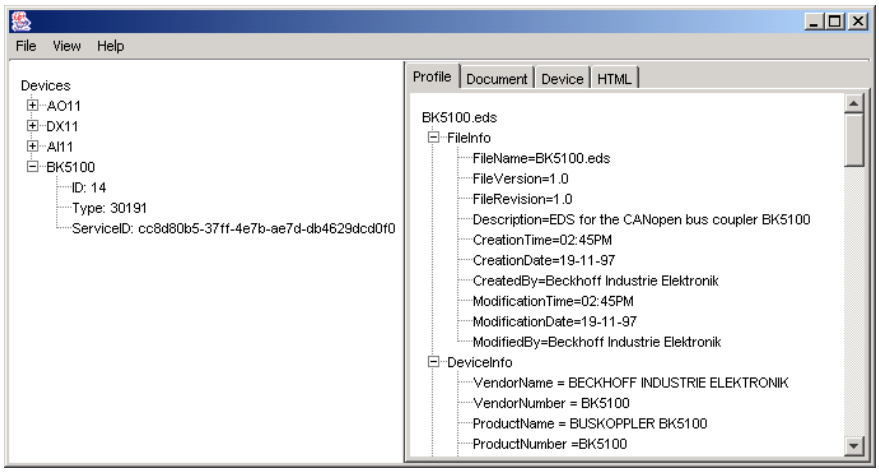


Abbildung 6.19: Jini CANopen-Client

Der Jini CANopen-Client und die zugehörigen Panels sind manuell implementiert. Die Panels werden nach dem Laden des Service-Proxy dynamisch instantiiert. Die den Panels zugrunde liegende Information bzw. Funktionalität wird von entsprechenden Diensten, z.B. HTTP-Server oder EDS-Server, zur Verfügung gestellt.

6.3.4 Kombination von Systemen

Die Integration von unterschiedlichen Systemen in Java erlaubt in der Konsequenz auch eine Kombination von unterschiedlichen Systemen. Dieser Ansatz soll ebenfalls an einem Beispiel erläutert werden. Auf der einen Seite können zur Anbindung von CANopen-Komponenten Java-Komponenten gemäß Abschnitt 6.3.3 generiert werden. Auf der anderen Seite können Windows-Anwendungen durch die Generierung von Java-Komponenten in Java benutzt werden. Zur Kombination dieser Systeme werden einfach beide Systeme in einer gemeinsamen Anwendung verwendet. Ein mögliches Anwendungsszenario ist z.B. die Protokollierung von Eingangssignalen eines analogen CANopen-Moduls und die Übertragung dieser Daten nach Excel gemäß Listing 6.2.

```
public class CANopenExcel
implements AnalogInputListener
{
  Excel.Worksheet sheet = null;
  Excel.Range range = null;
  int row = 0;

  public void readChannel1Changed(int newValue)
  {
    String colA = "A" + row;
    range = sheet.getRange((Object)colA, null);
    range.Select();
    range.setValue(new Integer(newValue));
    row++;
  }
}
```

Listing 6.2: Beispiel einer kombinierten CANopen/COM-Anwendung

In dem gezeigten Ausschnitt werden Analog-Werte durch den vorgestellten Callback-Mechanismus in der Methode *readChannel1Changed()* empfangen (Zeile 8) und in eine selektierte Zelle eines Excel-Datenblatts übertragen (Zeilen 10-13). Dieses Datenblatt wiederum dient gemäß Abb. 6.20 als Grundlage für ein Diagramm.

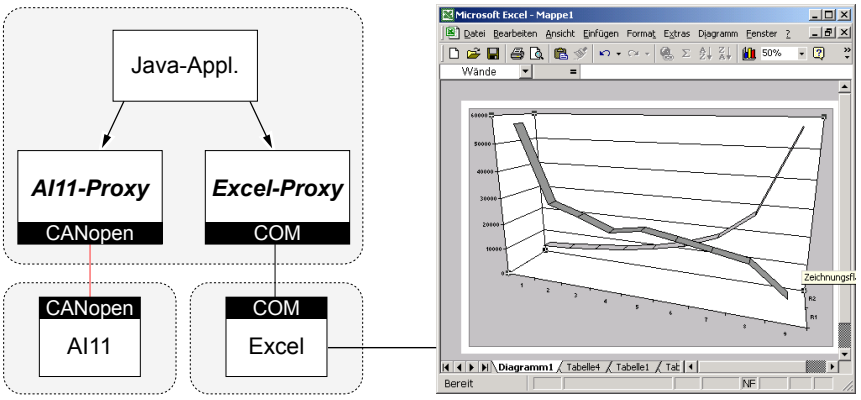


Abbildung 6.20: Kombination von Systemen

Der eingangs motivierten Zielsetzung in Abschnitt 1.1, nämlich der Integration von Automationssystemen in eine allgemeine Service- und Management-Infrastruktur auf Basis von modernen Softwarekonzepten und -technologien, wird u.a. mit der beschriebenen Vorgehensweise Rechnung getragen.

6.3.5 Steuerung eines Roboters

In diesem Abschnitt wird neben der Integration eines Systems ein weiterer Aspekt, nämlich der der XML-basierten Visualisierung, vorgestellt. Das Zielsystem ist ein Roboter, der zum einen über Internet gesteuert und zum anderen mit Java 3D visualisiert wird. Die Visualisierung mit Java 3Dbasiert dabei auf einem XML-basierten Modell des Roboters.

Mit der Einbindung von Automationssystemen in das Internet oder ein Intranet stellt sich das Problem der Visualisierung von entfernten Anlagen. Die Übertragung von Live-Bildern kommt in Folge der zur Verfügung stehenden Bandbreite nicht immer in Betracht. Prinzipiell ist es ausreichend, nur den Status einer entfernten Anlage zu übertragen und entsprechend zu verarbeiten bzw. visualisieren. Der aktuelle Zustand einer Anlage kann, wie in Nusser et al. [NBGK01] erörtert, als Zusatzinformation in ein Standbild eingeblendet werden. Qualitativ hochwertiger und intuitiver ist die Verwendung eines drei dimensionalen Modells, das die aktuellen Daten der Anlage verarbeitet. Diese Vorgehensweise hat den Vorteil, dass das Modell eine sehr gute Visualisierung des Systems bietet, dennoch nur der aktuelle Zustand einer entfernten Anlage übertragen wird. Die Steuerung von entfernten Systemen birgt gewisse Sicherheitsrisiken in sich, auf die hier nicht näher eingegangen werden soll.

6.3.5.1 Java 3D Roboter Visualisierung basierend auf XML

In diesem Abschnitt wird eine Repräsentation eines Scara-Roboters[2] mittels XML und die Umsetzung dieser Beschreibung in ein Java 3D-Modell vorgestellt. Der Vorteil einer Beschreibung mit XML ist, neben den allgemeinen Vorteilen von XML gemäß Abschnitt 2.4.5, die Möglichkeit zur dynamischen Umsetzung eines Modells in eine graphische Visualisierung. Zudem besteht die Möglichkeit, das Modell einfach in die bestehende Infrastruktur zu integrieren.

In der vorliegenden Arbeit werden Roboter berücksichtigt, die durch einen Manipulator gekennzeichnet sind. Primäre Aufgabe von Robotern ist das Greifen und Bearbeiten von Objekten im dreidimensionalen Raum. Um ein Objekt mit einer beliebigen Lage im Raum zu greifen, benötigt ein Roboter sechs Freiheitsgrade. Drei Freiheitsgrade werden zur Translation des Greifers in X-, Y- und Z-Richtung und drei für eine Drehung um die Koordinatenachsen benötigt.

Der Manipulator eines Roboters besitzt einen hierarchischen Aufbau und besteht aus einer Anzahl von *Verbindungsgliedern* (*Link*), die durch *Gelenke* miteinander verbunden sind. Wird ein hierarchisch höher liegendes Gelenk bewegt, so bewegt sich auch der Rest eines Roboters. Gemäß Abb. 6.21 unterscheidet man Rotations- (a und b) und Translationsgelenke (c).

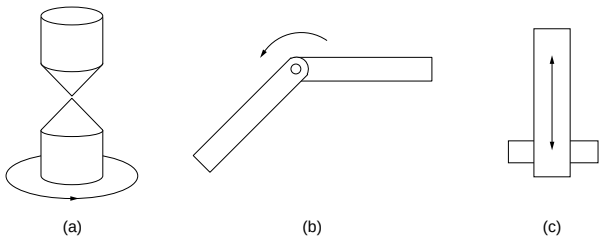


Abbildung 6.21: Gelenktypen nach McKerrow [McK91]

Ein Verbindungsglied verbindet jeweils zwei Gelenke miteinander und wird, falls die Gelenke orthogonal zueinander stehen, als orthogonales Verbindungsglied bezeichnet. Die Verbindungsglieder unterscheiden sich hinsichtlich der Ausrichtung der Gelenke zueinander und werden mittels der sog. DH-Parameter (Denavit-Hartenberg-Paramter) beschrieben. Diese Parameter legen die Translations- und Rotationswerte zwischen dem Koordinatensystem des ersten Gelenkes und dem Koordinatensystems des zweiten Gelenkes fest. Die unterschiedlichen Typen von Verbindungsgliedern sind in Abb. 6.22 dargestellt.

6.3.5.2 Roboter XML-DTD

Ein Roboter besteht aus n Verbindungsgliedern. Ein Verbindungsglied besteht aus zwei Gelenken und einem Verbindungsstück, das die Gelenke miteinander verbindet. Jedes Verbindungsglied besitzt eine *Boundingbox*, die aus n geometrischen Primitiva bestehen

[2] Scara beschreibt eine bestimmte Bauweise eines Roboters.

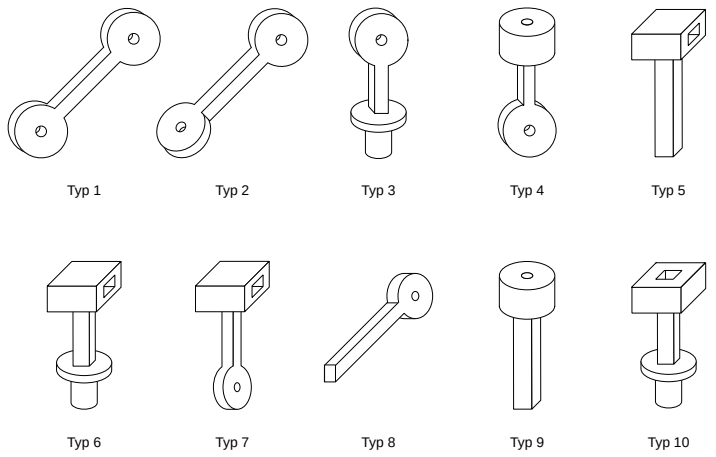


Abbildung 6.22: Verbindungstypen nach McKerrow [McK91]

kann. Diese Elemente bilden die Grundelemente der XML-DTD. Somit definiert die DTD die sechs Elemente *Robot*, *Link*, *Boundingbox* und die graphischen Primitiva *Box*, *Cylinder* und *Sphere*, die im Folgenden erläutert werden. Die vollständige DTD findet sich im Anhang A.5.

Das DTD-Element *Robot* ist das Wurzel- bzw. Containerelement des Roboterprofils. Aufgrund der hierarchischen Struktur enthält dieses Element nur ein einziges Element vom Typ *Link*. Ein Element vom Typ *Link* repräsentiert ein Verbindungsglied und definiert die Attribute *LinkType*, *DegreeMax*, *DegreeMin*, *ZeroPosition* und *zAxis*.

- LinkType definiert den Typ der Verbindung. Grundsätzlich stehen die in Abb. 6.21 aufgeführten Typen zur Verfügung.
- DegreeMax/DegreeMin beschreiben den max. bzw. min. Freiheitsgrad eines Gelenkes. Bei Rotationsgelenken beträgt der max. bzw. min. Freiheitsgrad $180°$ bzw. $-180°$. Die Werte von Translationsgelenken liegen minimal bei 0 bzw. maximal bei der Höhe eines Gelenkes.
- ZeroPosition beschreibt die initiale Position eines Gelenkes.
- zAxis kann entweder den Wert positiv oder negativ annehmen und beschreibt die Richtung, an der ein nachfolgendes Verbindungsglied angebracht ist.

Jedem Verbindungsglied ist eine *Boundingbox* zugeordnet, welche die maximale Ausdehnung eines Verbindungsgliedes spezifiziert. Boundingboxen sind nur für eine sehr grobe Darstellung eines Roboters geeignet. Zur detaillierten Visualisierung kann deshalb eine *Boundingbox* (beliebig) viele geometrische Primitiva umfassen, die zur graphischen Repräsentation von Verbindungsgliedern verwendet werden. Die DTD unterstützt die Primitiva Quader, Zylinder und Kugel, da diese direkt von Java 3D unterstützt werden. Die Primitiva werden durch entsprechende Attribute (z.B. Höhe, Breite, Tiefe, Radius) näher spezifiziert.

6.3.6 Visualisierung des Roboters

Zur Repräsentation von Robotern dient ein XML-Dokument, das der oben aufgeführten DTD genügt. Dieses Dokument wird gemäß Abb. 6.23 manuell aus einer Vorlage erstellt und dient als Bauplan für das Java 3D-Modell des Roboters[3].

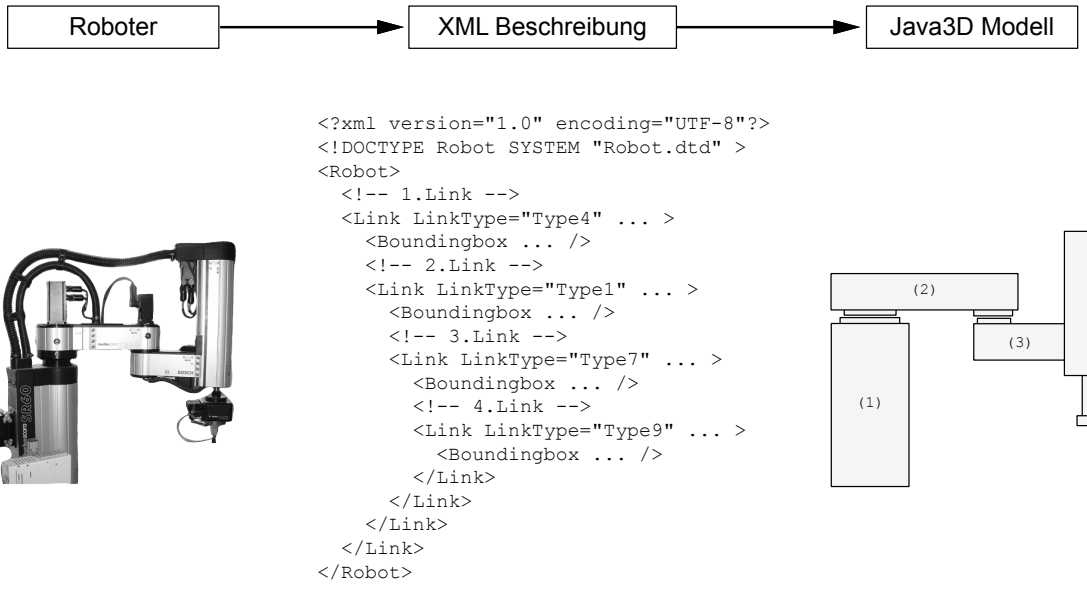


Abbildung 6.23: Abbildung eines Roboters in ein XML-Roboter-Profil

Die Boundingboxen werden in dem erzeugten Java 3D-Modell auf entsprechende Quader abgebildet. Die Quader sind für eine sehr grobe Visualisierung des Roboters geeignet. Für eine detaillierte Ansicht werden zusätzlich die oben genannten graphischen Primitiva benötigt. Zur Visualisierung des Java 3D-Modells dient ein Java-Applet, das zur Laufzeit als Eingabe ein XML-Dokument erhält. Dieses Dokument wird entsprechend interpretiert und dargestellt. Abb. 6.24 zeigt zwei unterschiedliche Profile des Roboters vom Typ Scara 60. Im linken Ausschnitt werden nur die Boundingboxen, im rechten zusätzlich graphische Primitiva zur Visualisierung des Roboters verwendet.

Der visualisierte Roboter ist in einen 3-dimensionalen Raum eingebettet und kann verschoben, skaliert und aus allen Richtungen betrachtet werden. Das Applet unterstützt zudem das Laden von XML-Roboter-Profilen zur Laufzeit und die Steuerung und die Anzeige der einzelnen Verbindungsglieder. Mit der XML-basierten Visualisierung in Java 3D liegt ein Client mit einer komfortablen Benutzerschnittstelle vor.

6.3.7 Integration des Roboters

Während im vorigen Abschnitt die Visualisierung von Robotern basierend auf XML-Profilen im Vordergrund stand, wird in diesem Abschnitt die prototypische Anbindung eines realen Roboters und einer Simulation vorgestellt.

[3]Auf eine detaillierte Beschreibung der Infrastruktur zur Konstruktion des Modells wird in dieser Arbeit verzichtet und auf [Alt99] verwiesen.

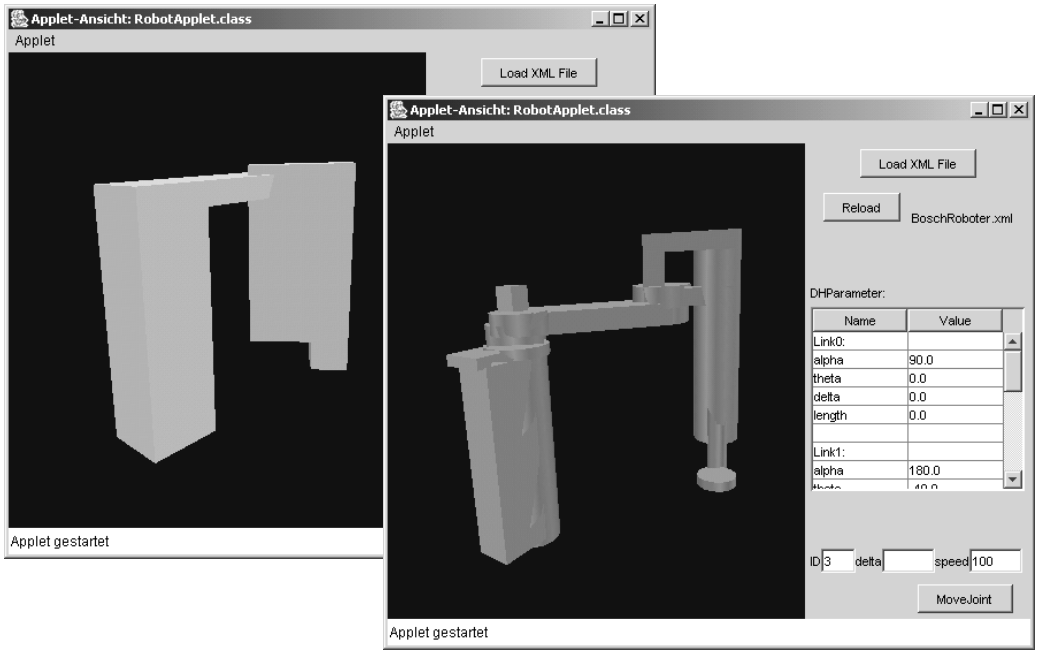


Abbildung 6.24: Java3D-Applet zur Visualisierung und Steuerung von Robotern

Die drei Achsen des Scara 60 werden durch drei CANopen-Antriebe gesteuert. Die Antriebe unterstützen eine absolute und eine relative Positionierung der Achsen, wobei die Positionswerte mit 4 Bytes kodiert sind. Synchronisiert durch eine sog. Sync-Nachricht schicken die Antriebe in regelmäßigen Abständen ihre aktuelle Position und ihren Status. Zur Steuerung der Antriebe werden entsprechende Nachrichten auf den Bus übertragen. Die Steuerung des Roboters in Java ist mittels Java-Klassen und der CANopen-API aus Abschnitt 2.5.3.2 realisiert. Der Scara 60 wird durch die Klasse *Scara* repräsentiert und umfasst drei Antriebe, wobei ein Antrieb durch die Java-Klasse *Drive* gekapselt ist. Die Klasse Scara bietet u.a. die Methoden *moveABS()* und *moveREL()*, die als Parameter einen Identifier, eine absolute bzw. relative Position, eine Beschleunigung und eine Geschwindigkeit erhalten. Die aktuellen Positionswerte werden durch einen Callback-Mechanismus kontinuierlich an die registrierten Klassen weitergereicht. Zur Reduktion des Datenaufkommens kann ein Δ-Wert angegeben werden, so dass nur Positionen propagiert werden, die sich um den gegebenen Wert von der vorigen Position unterscheiden. Neben der Steuerung des realen Roboters gibt es noch die Möglichkeit zur reinen Simulation, auf die hier nicht näher eingegangen wird.

Analog zu den Abschnitten 6.3.1 und 6.3.2 wird die Klasse *Scara* metaisiert und serialisiert. Die Generierung der Middleware wird ebenfalls analog zu den genannten Anwendungen realisiert, so dass eine Möglichkeit zum entfernten Zugriff auf den Roboter zur Verfügung steht. Die Systemarchitektur zur Steuerung des Scara 60 mit Java 3D-Visualisierung ist in Abb. 6.25 dargestellt.

Die vorgestellte Infrastruktur zur Java 3D-Visualisierung von Robotern und automatischen Generierung von Middleware basierend auf XML ist sehr flexibel und komfortabel. Noch flexibler und komfortabler wäre eine Möglichkeit zur direkten Verbindung eines

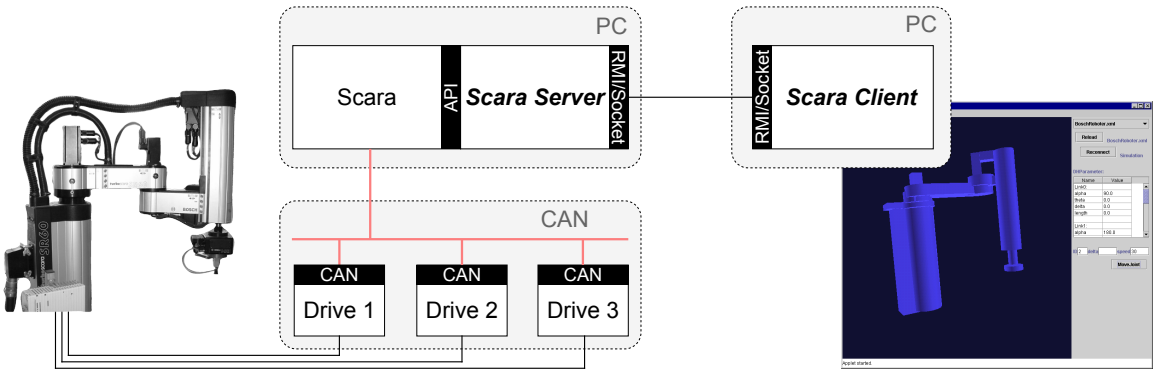


Abbildung 6.25: Scara Roboter

Metamodelles mit einem Robotermodell, so dass die notwendige Logik zur Umsetzung der Schnittstelle in entsprechende Befehle für die Visualisierung automatisch generiert werden könnten. Dieser Ansatz würde z.B. analog zur Vorgehensweise bei Java-Beans eine graphische Programmierung von Systemen ermöglichen.

6.4 Bewertung des Systems

Im Folgenden wird MAX in die Taxonomie der Softwarewiederverwendung gemäß Abschnitt 2.2.3 eingeordnet und an einigen Qualitätskriterien gemäß Abschnitt 2.1.3 beurteilt.

6.4.1 Einordnung in die Taxonomie

In MAX findet eine systematische Wiederverwendung von Artefakten und Komponenten, die in Form von Code und Spezifikationen vorliegen, statt. Dabei werden bestehende Komponenten unverändert übernommen und durch generierte Komponenten gekapselt. Es handelt sich somit um eine Form der generativen black-box Wiederverwendung von Artefakten.

6.4.2 Beurteilung

MAX wird im folgenden an den Attributen Anwendbarkeit, Verteiltheit, Integrierbarkeit, Konformität zu Standards, Erweiterbarkeit, Leistungsfähigkeit, Portabilität und Skalierbarkeit gemäß Abschnitt 2.1.3 beurteilt.

- **Anwendbarkeit:** MAX richtet sich vor allem an Entwickler und unterstützt diesen bei der Integration von Komponenten in (verteilte) Systeme. Der Integrationsprozess wird dabei durch Eclipse-Plugins, Werkzeuge und APIs unterstützt.
- **Verteiltheit:** Zwar kann die Metaisierung von Komponenten und Generierung von Schnittstellendokumentation über das Web initiiert werden, MAX bietet aber an-

sonsten keine Unterstützung für die verteilte Anwendung. Auf sich selbst angewendet könnte MAX jedoch in eine verteilte Middleware integriert werden.

- **Integrierbarkeit:** MAX besteht aus einer Reihe von unabhängigen Komponenten in Form von Klassen bzw. Programmen, wobei nahezu alle Komponenten auf dem allgemeinen Metamodell basieren. Das Modell stellt folglich die Basis für die Integrierbarkeit, also die Interoperabilität und die Uniformität, der Komponenten dar.
- **Konformität zu Standards und Offenheit:** Für die Verarbeitung und den Datenaustausch von Modellen dient ein allgemeines, aber proprietäres Metamodell, so dass MAX die genannte Anforderung nur bedingt erfüllt. Das Format ist zwar proprietär, allerdings stehen ein Import bzw. Export von bzw. nach XML und entsprechende Schnittstellen zur Verarbeitung des allgemeinen Modells zur Verfügung.
- **Erweiterbarkeit:** MAX kann einfach erweitert werden. Zum einen können weitere Systeme und zum anderen weitere Generatoren integriert werden. Ersteres bezieht sich auf die Metaisierung von Komponenten auf Basis des allgemeinen Modells, zweiteres auf die Generierung von Software. Bei der Integration von weiteren Systemen bzw. Technologien ist eine entsprechende Schnittstelle zur Interaktion notwendig.
- **Leistungsfähigkeit:** Die Leistungsfähigkeit hängt stark von den zu metaisierenden bzw. zu generierenden Komponenten ab, wobei vor allem die Anzahl der von einer Komponente definierten Typen ausschlaggebend ist.
- **Portabilität:** MAX ist zwar in Java entwickelt und demzufolge portabel, allerdings sind manche der integrierten Systeme bzw. Bibliotheken, z.B. COM bzw. CANopen, systemabhängig. Systemabhängige Komponenten werden aber durch zusätzliche Schnittstellen gekapselt, so dass es z.B. möglich ist, betriebssystemspezifische Komponenten durch konzeptionell identische Komponenten auszutauschen (z.B. Verwendung von unterschiedlichen CAN-Bibliotheken gemäß Abschnitt 2.5.3.2).
- **Skalierbarkeit:** Die Metadaten einer Komponente bzw. deren XML-Repräsentation (mittels DOM) werden im Hauptspeicher zwischengespeichert, so dass der zur Verfügung stehende Speicher die hauptsächliche Einschränkung bzgl. der Skalierbarkeit darstellt. Eine Optimierung könnte durch die sukzessive Verarbeitung und Serialisierung und Deserialisierung von Typen (mittels SAX) erreicht werden, wobei referenzierte Typen bei Bedarf geladen werden.

6.4.3 Vorteile

Eine systematische Anwendung von MAX führt zu einem hohen Grad an Wiederverwendung. Je unabhängiger Komponenten von einem bestimmten System bzw. einer Systemumgebung sind, desto höher ist der Grad der Wiederverwendbarkeit.

Bestehende Komponenten, die über keine Schnittstelle zur verteilten Kommunikation verfügen, können einfach, wie in Abschnitt 5.3 ausführlich erläutert, in unterschiedliche Middlewaresysteme integriert werden.

Die XML-basierten Komponentenbeschreibungen von MAX sind plattform- und sprachunabhängig und im Hinblick auf zukünftige Entwicklungen erweiterbar. Die generierten XML-Dokumente können einfach übertragen und mit Standardsoftware, z.B. XML-Bibliotheken und -Werkzeugen, verarbeitet werden.

Die Wiederverwendung betrifft nicht allein die Wiederverwendung von Software, sondern auch die Wiederverwendung der generierten Beschreibungen. Die Beschreibungen können beliebig oft und zu unterschiedlichen Zwecken verwendet und somit wiederverwendet werden.

Die in den Abschnitt 6.1.1.4 vorgestellten (Registry-, Dateisystem-, CANopen-) Observer ermöglichen die Metaisierung und Generierung von Komponenten zur Laufzeit, so dass MAX eine Basis für dynamische Umgebungen, beispielsweise für mobile Agenten [FNK02, FTNK04], darstellt.

Eine automatische Metaisierung und Generierung von Komponenten mittels MAX ist effizient und weniger fehleranfällig als eine manuelle Implementierung. Dies führt in der Konsequenz zu einer wesentlichen Verkürzung der Entwicklungszeit und zu einer höheren Qualität von Systemen.

6.4.4 Nachteile

Bei der Metaisierung von Komponenten wird ein Metamodell erzeugt. Gemäß Abschnitt 3.1.4 besteht zwischen diesem Modell und dem System eine kausale Verbindung, falls es korrekt und konsistent ist. In MAX wird dieses Modell zu einem beliebigen Zeitpunkt erstellt und serialisiert. Falls sich das System, also eine der metaisierten Komponenten ändert, so ist das Modell bzw. dessen XML-Repräsentation nicht mehr konsistent. Zur Vermeidung von Inkonsistenzen können bestimmte Eigenschaften, wie das Datum oder die Größe von metaisierten Komponenten, in der entsprechenden XML-Repräsentation gespeichert werden. Optimal wäre jedoch, wenn jede Komponente ihre Metadaten zur Laufzeit liefert.

Der Speicherbedarf zur Zwischenspeicherung von Metamodellen im Hauptspeicher bzw. als XML-Dokument im Dateisystem ist relativ hoch. Die XML-Dokumente von Excel benötigen beispielweise ca. 12 MB an Festplattenspeicher. Der Speicherbedarf kann durch eine Komprimierung der Dateien auf etwa 5% der ursprünglichen Größe reduziert werden. Am Beispiel von Excel entspricht sie damit etwa dem Speicherbedarf der binären Typbibliothek.

Bei den gezeigten Beispielen bzw. Systemen wird der notwendige Code zumeist während der Entwicklungszeit generiert. Durch die Verwendung der in Abschnitt 6.2.2 vorgestellten Werkzeuge, APIs (Observer) und Skripte zur automatischen Metaisierung und Generierung von Komponenten können neue Komponenten auch zur Laufzeit generiert werden. Zur Übersetzung von generiertem Code zur Laufzeit ist allerdings die Verfügbarkeit eines JDK notwendig. Diese Voraussetzung ist bei eingebetteten Systemen zumeist nicht gegeben.

Durch die Kapselung einer Komponente werden ein oder mehrere zusätzliche Indirektionsschritte benötigt. Dieser Nachteil relativiert sich jedoch durch die zusätzliche Funk-

tionalität und den hohen Grad an Wiederverwendung.

Aktuell wird die Software zur Kapselung von Komponenten in Java generiert. Dies hat den Nachteil, dass bei einer Änderung der entsprechende Code editiert und übersetzt werden muss. Im Hinblick auf die Erweiterbarkeit und Flexibilität des Systems wäre die Verwendung von XSL bzw. einer interpretierten Skriptsprache sicherlich sinnvoll.

6.5 Verwandte Forschungsansätze

6.5.1 Reflektive Systeme, Metasysteme, -modelle und -architekturen

Im Folgenden können prinzipiell zwei unterschiedliche Konzepte mit dem Ziel der Integration und der Wiederverwendung von Software bzw. Softwarekomponenten unterschieden werden.

Das erste Konzept umfasst Systeme bzw. Techniken, die zur Integration von Komponenten eine bestimmte Infrastruktur anbieten. Zur Integration von Komponenten in diese Infrastruktur müssen die Komponenten selbst bestimmte Anforderungen erfüllen. So müssen die Komponenten beispielsweise bestimmte Eigenschaften besitzen oder bestimmte Schnittstellen implementieren. Bestehende Komponenten, die diesen Anforderungen nicht genügen, sind nur schwer, wenn überhaupt integrierbar.

Beim zweiten Konzept wird davon ausgegangen, dass bestehende Komponenten nicht geändert werden können, aber aufgrund ihrer Implementierung bestimmte, z.B. reflektive, Eigenschaften besitzen. Diese Eigenschaften werden durch ein System bzw. eine Technik genutzt, um Komponenten in eine entsprechende Infrastruktur zu integrieren. Wie aus den vorangegangenen Kapiteln sichtbar ist, verfolgt MAX das zuletzt genannte Konzept.

Parlavantzas et al. stellen in [PCCB00] mit OpenCOM eine flexible Middlewareplattform auf Basis von Komponenten und Reflexion vor. Sog. *Meta-Interfaces* dienen dabei der Konfiguration, Anpassung und Introspektion von Komponenten zur Laufzeit. Das Komponentenframework selbst hängt aufgrund von binär-kompatiblen Komponenten von keiner bestimmten Programmiersprache ab und es besteht keine Bindung zwischen den Komponenten und dem Framework. Die Kommunikation basiert allerdings auf der Middleware GOPI [Cou98].

Das Framework von Welch et al. [WS02] erlaubt die Realisierung von nicht-funktionalen Eigenschaften in Java zur Laufzeit. Dabei werden auf Basis der Java Reflection API gegebene Komponenten introspeziert und entsprechende Wrapper zur Kapselung von Komponenten zur Laufzeit generiert und übersetzt. Die Benutzung des Frameworks bzw. des zugehörigen Metaobjekt-Protokolls ist auf Java beschränkt und bedarf der Anpassung von Klassen und Anwendungen, z.B. der Ableitung von Klassen von gegebenen Metaklassen oder der Instantiierung eines speziellen Class-Loaders. Zur Übersetzung des Codes zur Laufzeit ist es notwendig, dass ein entsprechendes JDK zur Verfügung steht.

In CentiJ [Lyo02] wird der notwendige Java-Code zur verteilten Kommunikation mit RMI ebenfalls automatisch generiert. Dabei werden auf Basis der Java Reflection API die Schnittstelle von Komponenten introspeziert und automatisch Proxies zur Kommu-

nikation via RMI generiert. Lavers stellt in [Lav01] ebenfalls eine Technik zur automatischen Generierung von RMI-Code vor. LAVA ist eine Erweiterung von Java um Delegation [Kni98, Kni99]. Laut Kniesel ist die Effizienz von aktuellen Implementierungen von LAVA jedoch nicht akzeptabel.

MetaJava bzw. metaXa, ist eine Erweiterung von Java um Reflexion [KG96a, GK97b, Gol97, GK97a, KG97, GK98] gemäß Abschnitt 3.1. Mit MetaJava ist es u.a. möglich, nicht-funktionale Eigenschaften, z.B. entfernte Methodenaufrufe, Migration und Replikation von Objekten [KG96b], Sicherheitsmechanismen oder Synchronisation, mittels Metaobjekten in einer Metaebene zu realisieren. Der Nachteil von MetaJava ist die Notwendigkeit eines erweiterten Java-Interpreters, der MetaJava-Virtual-Machine (MJVM) [KG96a, KG97, GK97a].

In der Vergangenheit wurden für unterschiedliche Domänen unterschiedliche Metamodelle entwickelt. Die bekanntesten Vertreter heutzutage sind die *Meta Object Facility* (*MOF*) der OMG [Obj02] und neuerdings das *Eclipse Core Model Framework* (*EMF*) von IBM [IBM04]. In der Automationsindustrie sieht man ebenfalls die Vorteile der Modellierung.

Die Meta Object Facility [Obj02] ist eine Spezifikation der OMG und bietet eine Sammlung von IDL Schnittstellen für die Verwaltung verteilter Metaobjekte an [Jec01]. Die MOF enthält Repräsentationen für Klassen, Schnittstellen, Typen, Attribute, Operationen, Assoziationen, Generalisierungen, etc. und definiert gemäß Abb. 3.1.6 ein Meta-Metamodell (Ebene M^{3+}). Die OMG benutzt MOF z.B. zur formalen Definition der Unified Modelling Language kurz UML (Ebene M^2) und definiert die Elemente, die im Rahmen von UML-basierter Modellierung verwendet werden können. Zur textuellen Beschreibung, persistenten Speicherung und zum Austausch von UML-Modellen dient XML Metadata Interchange (XMI). Die Ziele von XMI sind die Schaffung eines Industriestandards zur streambasierten Übertragung von einer Vielzahl verschiedener Modelle [Jec04].

Das Ziel der Model Driven Architecture (MDA) ist die Schaffung eines Standards zur Integration von unterschiedlichen Systemen auf Basis von MOF, CWM (Common Warehouse Metamodel) und XMI [Obj01a]. Dabei werden Systeme zunächst unabhängig von einer bestimmten Plattform mit UML modelliert (Plattform Independent Model, PIM), in einem nächsten Schritt auf ein plattformspezifisches UML-Modell (Plattform Specific Model, PSM) abgebildet, das in einem weiteren Schritt zur Generierung von Sourcecode, Konfigurationsdateien und Middleware (z.B. Web Services, CORBA, etc.) benutzt wird. Zur Speicherung der Modelle (PIM und PSM) findet die MOF Verwendung. Die Generierung eines *CORBA Component Model* (CCM) Servers mittels MDA ist beispielhaft in Abb. 6.26 dargestellt.

Bei MDA liegt der Fokus auf der plattformunabhängigen Modellierung von Anwendungen und der automatischen Generierung von plattformspezifischem Code. Analog zu dieser Arbeit werden aus plattformunabhängigen Modellen plattformspezifische Anwendungen automatisch generiert, wobei MDA von bereits in UML modellierten Anwendungen ausgeht.

Das *Eclipse Core Model Framework* (*EMF*) ist „eine effiziente Java-Implementierung einer Untermenge der MOF“ [Ecl02]. Dabei liegt der Fokus, wie in dieser Arbeit, auf der Repräs-

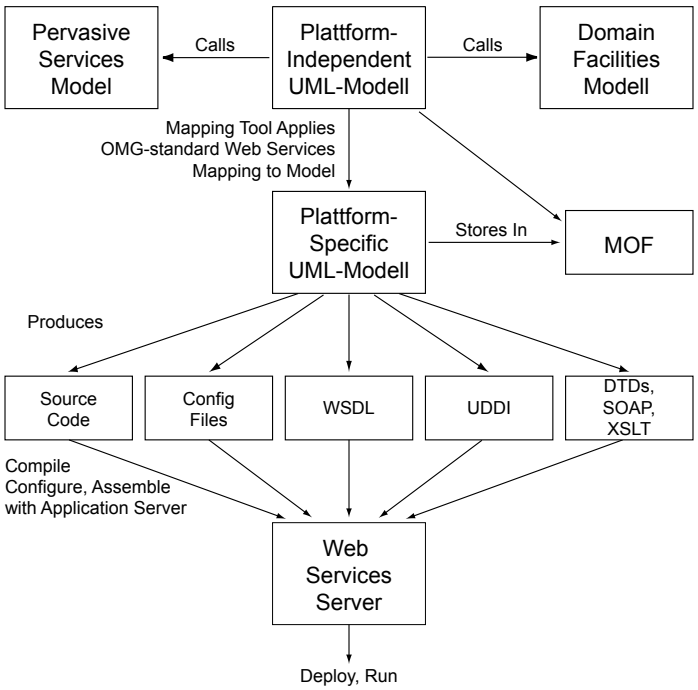


Abbildung 6.26: Generierung eines CCM-Servers mit MDA gemäß OMG [Obj01a]

sentation von Modellen und der automatischen Generierung von Code. EMF unterstützt aktuell die Metaisierung von Java und die Generierung von Java-Code. Das Klassendiagramm des EMF ist in Abb. C.1 dargestellt. In der modernen Softwareentwicklung sind Metamodelle bzw. Modellierungssprachen nicht mehr wegzudenken.

6.5.2 Schnittstellen-, Modul-, Komponenten- und Architekturbeschreibungssprachen

Mit der steigenden Komplexität und Größe von Softwaresystemen (s. a. 1.1) und der damit verbundenen Problematik der Integration von Systemen, wurde bereits in den siebziger Jahren damit begonnen, die Integration von großen Systemen mit Hilfe von formalen Beschreibungen bzw. Beschreibungssprachen zu unterstützen [Ger97]. Diese Sprachen reichen von einfachen Schnittstellenbeschreibungssprachen über Modulbeschreibungssprachen (*Module Interconnection Languages* kurz *MILs* bzw. Komponentenbeschreibungssprachen (*Component Description Languages*) kurz *CDLs* bis hin zu komplexen Architekturbeschreibungssprachen. MILs bieten formale Konstrukte zur Identifikation und Spezifikation von Modulen zur Konstruktion von kompletten Systemen [PDN86]. Das System POLYLITH [Pur94] bietet z.B. eine Infrastruktur zur Integration und Interaktion von Softwarekomponenten aus unterschiedlichen Programmiersprachen in einer heterogenen Systemumgebung, wobei die Anwendungsstruktur durch eine MIL spezifiziert wird. Eine Erweiterung von MILs um die Definition von Kommunikationsprotokollen und semantische Eigenschaften werden als Architekturbeschreibungssprachen (Architecture Description Languages (ADLs) bzw. Architecture Specification Language (ASL)) bezeichnet. In dem Softwaresystem Olan [BR96, BABR96, BBRVD98] dient die

Olan Configuration Language (OCL) zur Konfiguration von Komponenten und deren Konnektoren. Während Schnittstellenbeschreibungssprachen auf die Beschreibung einer einzelnen Komponente beschränkt sind, wird bei ADLs auch die Interaktion von Komponenten beschrieben. ADLs sind durch die Komplexität der Systeme inhärent komplex und meist nicht mit vertretbarem Aufwand zu erstellen. Häufig werden die genannten Beschreibungssprachen wie bei Olan kombiniert, so dass IDL zur Spezifikation von Komponentenschnittstellen und ADL zur Konfiguration von Komponenten und Konnektoren benutzt werden.

Die wohl am weitesten verbreitete Schnittstellenbeschreibungssprache ist IDL (s. Abschnitt 2.3.3.1). IDL ist zwar sehr generell, besitzt aber auch Nachteile. Die Generalität hat zur Folge, dass viele Erweiterungen, z.B für Echtzeit oder Quality of Service, entwickelt werden. Die Tatsache, dass IDL und meist auch die Erweiterung textbasiert sind, hat zur Folge, dass zur Verarbeitung von IDL ein entsprechender Parser, z.B. JavaCC [Col03] mit einer bestimmten Grammatik, notwendig ist. Dem allgemeinen Trend folgend, propagiert Jacobsen in [JK00] deshalb ein Framework zur Abbildung von IDL und IDL-spezifischen Erweiterungen nach XML.

Die *Jade Bird Component Description Language* (*JBCDL*) ist Teil der *Jade Bird Component Library* (*JBCL*), die wiederum auf dem *Jade Bird Component Model* (*JBCOM*) basiert [QJHF97]. JBCOM verfolgt das Ziel der Verwendung von Spezifikationen, Dokumenten, Patterns, Testplänen (indirekte Wiederverwendung) und der Verwendung von Komponenten (direkte Wiederverwendung). Das Komponentenmodell verfolgt dabei die Konzepte des 3C-Modell von Tracz [Tra90]: Concept, Content, Context (Abstrakte Beschreibung einer Komponente, Implementierung der Beschreibung, Abhängigkeit einer Komponente von ihrer Umgebung). JBCDL erlaubt die Beschreibung der Schnittstelle einer Komponente und unterstützt die Komposition, Verifikation und Suche von Komponenten. Die Beschreibungssprache selbst ist textbasiert und definiert mehrere einfache Schlüsselwörter (z.B. parameters, provides, requires, etc.). Zur Verarbeitung der manuell erstellten Komponentenbeschreibungen ist ein entsprechender Parser notwendig. Der Schwerpunkt von JBCDL liegt in der Komposition von Komponenten unter Benutzung von (semi-) automatischen Werkzeugen.

Die *Simple Object Description Language* (*SODL*) [MNA+01] ist eine domänenspezifische Sprache zur automatischen Erzeugung von Schnittstellen. Das Ziel von SODL ist die automatische Generierung von lokalen Objekten, die anstatt entfernter Objekte benutzt werden können. Generierte Objekte werden dabei zu einem Zielpunkt und nach Beendigung einer Aktion wieder an ihren Ursprungsort übertragen (Tier to Tier Object Transport, TTOT). Analog zu dieser Arbeit wird, basierend auf einem SODL-Dokument, ein temporäres SODL-Objektmodell erzeugt, aus dem ein IDL-Dokument und Quelltext in unterschiedlichen Programmiersprachen generiert werden. SODL-Dokumente werden manuell erstellt, sind textbasiert und ähneln IDL.

Die Java Markup Language (JavaML) [Bad00] bietet eine Repräsentation von Java Quelltext basierend auf XML. Die Intention von JavaML ist die Analyse und Transformation von Java Quelltexten mit vorhandenen XML- bzw. SGML-Werkzeugen. Analog zu dieser Arbeit kann dabei zusätzlicher Quelltext, z.B. zur Realisierung von nichtfunktionalen Eigenschaften (z.B. Logging), automatisch generiert werden. JavaML bzw.

die zugehörige DTD beschränkt sich auf die Repräsentation von Java Quelltext und ist, aufgrund des Befehlsumfangs in Java, sehr umfangreich.

Für die Java/COM-Middleware Jacob [Adl99] existiert eine Erweiterung [Lew01], die zur Repräsentation von Typbibliotheken ebenfalls XML benutzt. Aus Listing A.8 ist ersichtlich, dass diese Repräsentation speziell für die Repräsentation von COM-Typen bzw. -Typbibliotheken in XML ausgelegt ist.

Im Umfeld der Automationsindustrie wurden und werden ebenfalls mehrere (domänenspezifische) Beschreibungssprachen entwickelt. Britton et. al stellten bereits in den 80er Jahren in [BPP81] eine prinzipielle (informelle) Methode für das Design von abstrakten Schnittstellen für Gerätekomponenten vor. Unter der Annahme, dass der Austausch bzw. die Modifikation eines Gerätes zwar Auswirkungen auf ein gerätespezifisches Softwaremodul, nicht aber auf den Rest einen Softwaresystems hat, werden spezifische Eigenschaften durch die Verwendung von abstrakten Schnittstellen gekapselt. Die informellen Schnittstellenbeschreibungen enthalten dabei eine Liste von Annahmen, eine Funktionstabelle mit Funktionsnamen und zugehörigen Parametertypen und eine Ereignistabelle.

XML-OPC [OPC03b] transportiert die Ideen von SOAP und Web Services in die Automation auf Basis von OPC. Das Ziel von XML-OPC ist die Kopplung von Automationssystemen an das WWW und die vertikale Integration [HT01]. Die OPC-Foundation versucht damit den Einschränkungen bzgl. COM/DCOM entgegenzuwirken.

6.5.3 Codegenerierung

Die erwähnten Middlewareinfrastrukturen, z.B. RPC, CORBA oder RMI, unterstützen die automatische Generierung von Code. Das Ziel ist, den notwendigen Code zur verteilten Kommunikation automatisch zu generieren und von Details bzgl. der Kommunikation zu abstrahieren. Die Codegenerierung basiert dabei auf einer expliziten (z.B. RPC, CORBA) bzw. impliziten (z.B. RMI) Schnittstellenspezifikation, wobei jede Technologie dabei ihre eigene Spezifikation benutzt. Die oben genannten expliziten Spezifikationen sind mit entsprechenden Parsern zwar leicht zu interpretieren, entsprechen aber, im Gegensatz zu XML-basierten Spezifikationen (z.B. WSDL), nicht mehr dem aktuellen Stand der Forschung. Bei impliziten Spezifikationen wird vorausgesetzt, dass eine entsprechende API zur Extraktion der Schnittstelle vorhanden ist (z.B. Java- bzw. COM-Reflection API gemäß Abschnitt 3.2.1 bzw. 3.3.2).

Die in Abschnitt 2.3.3.7 genannten Java/COM-Produkte generieren ebenfalls automatisch Java-Klassen zur Anbindung von COM-Objekten. Allerdings werden die Komponenten bzw. Klassen in den meisten Fällen direkt auf Basis einer Typbibliothek erzeugt und nicht zwischengespeichert. Dies geschieht zwar aus gutem Grund, da die Schnittstelle bereits in der entsprechenden Typbibliothek spezifiziert ist, hat aber den Nachteil, dass analog zu oben immer eine entsprechende API zur Extraktion der Metadaten verfügbar sein muss. Eine Ausnahme bildet die bereits oben genannte Erweiterung der Java/COM-Brücke Jacob.

Analog zur Vorgehensweise in dieser Arbeit werden von CentiJ [Lyo02] RMI-Proxies zur verteilten Kommunikation mittels Reflexion und automatischer Codegenerierung ebenfalls automatisch erzeugt. CentiJ ist allerdings auf die Reflexion von Java-Komponenten

und die Generierung von RMI-Funktionalität beschränkt. CentiJ benutzt das Konzept der Delegation zur Simulation von Mehrfachvererbung (u.a. Mixin-Pattern).

Entwicklungswerkzeuge zur objektorientierten Programmierung bzw. zum Entwurf unterstützen ebenfalls die automatische Codegenerierung. Zu nennen sind hier vor allem Modellierungswerkzeuge, z.B. Together [Tog04], die aus einer (UML-) Beschreibung Software in unterschiedlichen Sprachen erzeugen. Teilweise bieten diese Werkzeuge die Möglichkeit, UML-Modelle nach XMI zu exportieren und XMI zu importieren. Zwar unterstützen viele Modellierungswerkzeuge das Reverse Engineering, der Einsatz eines solchen ist aber vor allem dann sinnvoll, wenn die Modellierung inhärenter Bestandteil des Entwicklungsprozesses ist, d.h. eine Anwendung von Anfang an modelliert wird.

Insgesamt lässt sich ein wachsendes Interesse an der automatischen Generierung von Software verzeichnen, das sich u.a. durch die steigende Komplexität, der zur Verfügung stehenden Zeit und den Kosten zur Realisierung von Systemen begründen lässt.

7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

MAX bietet eine komfortable und flexible Infrastruktur zur Metaisierung, Serialisierung und automatischen Generierung von Softwarekomponenten. Im Mittelpunkt von MAX stehen ein allgemeines Metamodell und dessen XML-Pendant MAXML. Das erklärte Ziel von MAX ist die einfache Wiederverwendung von bestehenden Komponenten bzw. Klassen durch die Integration dieser Komponenten in eine bestimmte Infrastruktur.

Das allgemeine Metamodell dient zur Abstraktion von spezifischen Systemen bzw. Technologien. Durch die automatische Abbildung von Schnittstellen mittels Reflexion – dem Prozess der Metaisierung – werden unterschiedliche Systeme bzw. deren Metamodelle in ein allgemeines Metamodell abgebildet. Die automatische Reflexion beruht dabei auf systemspezifischen Eigenschaften und Techniken. MAX unterstützt aktuell die Metaisierung von Java, COM, .NET und CANopen-Feldbuskomponenten. Während Java, COM und .NET eine Schnittstelle zur Reflexion bieten, basiert die Reflexion von CANopen-Komponenten weitestgehend auf der Verarbeitung von elektronischen Datenblättern. Innerhalb des Systems ist der Prozess der Metaisierung jedoch transparent. Die Integration von MAX in das Entwicklungswerkzeug Eclipse mittels Plugins unterstützt die Metaisierung und eine direkte Weiterverarbeitung von metaisierten Komponenten.

Die serialisierten Metamodelle werden durch MAXML repräsentiert und zur weiteren Verarbeitung gespeichert. Die XML-Dokumente bilden die Grundlage für die automatische Generierung von Softwarekomponenten und von Dokumentation. Mit MAXML liegt eine plattform- und sprachunabhängige Schnittstellen-Beschreibung der metaisierten Komponenten vor. Die Generierung von HTML- und PDF-Dokumenten basiert direkt auf zuvor serialisierten Dokumenten und ist mittels dem WWW-Publishing-System Cocoon auf Basis einer XML-Transformation realisiert. Zur einfachen Serialisierung und Deserialisierung von MAXML – also zur Abbildung des speicherbasierten Metamodells nach MAXML und umgekehrt – existieren entsprechende Schnittstellen und Werkzeuge. Die serialisierten Metamodelle stehen nach einer Deserialisierung wieder im Speicher zur weiteren Verarbeitung zur Verfügung. Die XML-Dokumente können für unterschied-

liche Zwecke beliebig oft wiederverwendet werden.

Das allgemeine Metamodell bildet die Grundlage für die automatische Generierung von Softwarekomponenten. Dabei werden die metaisierten Komponenten bei der Generierung über entsprechende programmiersprachliche Konstrukte in einen neuen Kontext eingebettet. Abhängig vom Typ einer Quellkomponente können entsprechende Komponenten automatisch generiert und somit in eine bestimmte Infrastruktur integriert werden. MAX unterstützt die Generierung von Middleware zur verteilten und zur lokalen Kommunikation. Die Technologien zur verteilten Kommunikation umfassen Sockets, RMI, Jini, .NET Remoting und Web Services. Die Technologien zur lokalen Kommunikation umfassen COM, .NET und JNI. Wie bereits oben erwähnt, wird zudem die Generierung von Dokumentation unterstützt.

Eine Besonderheit von MAX stellt die nahezu komplett automatisch generierte Middleware zur Integration des .NET-Frameworks in Java und in MAX dar. Mittels der Metaisierung der zugehörigen COM-Komponente des .NET-Frameworks wurden sämtliche .NET-Klassen mittels einer automatischen Generierung von Java-Komponenten zur Kapselung der COM-Komponenten automatisch generiert. Funktionalität, die nicht über COM adressierbar ist, wurde manuell in C# implementiert und wiederum durch Metaisierung und Generierung in Java gekapselt. Die erzeugte Java/.NET-Middleware erlaubt den Zugriff auf .NET-Klassen in Java , so dass die Metaisierung von .NET-Komponenten und -Klassen einfach mit der bestehende Infrastruktur realisiert werden kann. Die Generierung von Softwarekomponenten wird, wie die Metaisierung, durch entsprechende Werkzeuge und Plugins in Eclipse unterstützt, so dass die generierten Komponenten in Eclipse weiterverarbeitet werden können.

Zur passiven Interaktion, also der Interaktion ohne aktive Benutzerbeteiligung, stehen ebenfalls bestimmte Werkzeuge zur Verfügung. Zu nennen sind hier vor allem die Experten zur Observierung von gegebenen Quellen. MAX unterstützt die Observierung des Dateisystems, der Windows Registry und von CANopen-Systemen. Falls eine dieser Quellen modifiziert wird, so werden die oben genannten Prozesse automatisch aktiviert. Mit den Skripten zur automatischen Übersetzung von erzeugten Softwarekomponenten ist die Prozesskette – Metaisierung, Generierung und Übersetzung – vollständig und durchgängig in MAX realisiert. Analog zu oben ist die Metaisierung und Generierung vom Typ der Quellkomponente abhängig. Die passive Interaktion stellt die Grundlage für dynamische Umgebungen dar, in der Softwarekomponenten bei Bedarf metaisiert und generiert werden. Der beschriebene Mechanismus kann z.B. als Grundlage für Agentensysteme gesehen werden. Mit dem vorhandenen Wissen angereichert haben Agenten die Möglichkeit, eine passende Laufzeitumgebung zu suchen, die Generierung einer passenden Schnittstelle zu initiieren und mit einer oder mehreren Komponenten, z.B. COM-Komponenten, in Interaktion zu treten. Ein einfaches Beispiel hierfür ist die Umsetzung eines Word-Dokuments in ein PDF-Dokument, wobei beliebige andere Szenarien vorstellbar sind.

Durch die allgemeine schnittstellenbasierte Spezifikation von Komponenten aus unterschiedlichen Systemen ist die Integration dieser Systeme in verschiedenste Infrastrukturen auf einfache Art und Weise möglich und wird weitestgehend automatisch durchgeführt. Der Abstraktionsgrad zur Abbildung von Komponenten bzw. Klassen in das all-

gemeine Metamodell ist bewusst gering gehalten, so dass das Maß zur Bewertung von Abstraktionen – die kognitive Distanz – minimal ist.

Die Metaisierung von Java-Klassen gestaltet sich aufgrund der Verfügbarkeit einer entsprechenden Reflexions-Schnittstelle sehr einfach. Die nach XML serialisierten Metadaten werden zur Generierung von unterschiedlichen Middlewarekomponenten benutzt. MAX unterstützt die Generierung von Sockets, RMI und Jini. Die Integration der Komponenten in diese Middlewareinfrastrukturen wird hierbei mittels lokalen Stellvertretern – den Proxies –, welche die Quellkomponenten kapseln, realisiert. Die Quellkomponente wird per Aggregation und Delegation in den neuen Kontext eingebettet. Durch die strikte Trennung zwischen Applikationslogik und Middleware bleibt die Quellkomponente von einer bestimmten Middleware unabhängig, so dass sie in einem anderen Kontext wiederverwendet werden kann.

Die Metaisierung von COM-Komponenten auf Basis von Typbibliotheken und Generierung von Java/COM-Softwarekomponenten zur Kapselung der COM-Komponenten erlaubt die Benutzung von Standard-Windows-Anwendungen mit COM-Unterstützung in Java. Die Abbildung von COM-Komponenten in ein allgemeines Metamodell stellt, im Gegensatz zur Integration von COM-Komponenten in Java, eine Neuerung dar. Durch die Abbildung besteht die Möglichkeit, die Infrastruktur von MAX zu verwenden. Damit ist es bspw. möglich, die Schnittstellendokumentation für COM-Komponenten automatisch zu generieren.

Die objektorientierte Modellierung von CANopen-Komponenten und Einbindung dieser Komponenten in Java vereinfacht die Programmierung von CANopen-Systemen erheblich. Die Integration in Java erlaubt darüber hinaus die Einbindung dieser Systeme in moderne Software-Infrastrukturen wie z.B. RMI, Jini oder Web Services. Als Beispiel dient die Integration eines eingebetteten Low-Cost-Systems mit einem CANopen-Subsystem in Jini. Für das Beispiel wurden Java/CANopen-, Sockets- und Jini-Softwarekomponenten generiert und in ein Gesamtsystem integriert. Durch die Integration unterschiedlicher Systeme in ein bestimmtes Zielsystem, im vorliegenden Fall Java, können Systeme einfach kombiniert werden. So erlaubt bspw. die Integration von CANopen in Java und die Integration von COM in Java die Kombination von CANopen und COM. Anhand eines Beispiels wurde demonstriert, wie die Eingangswerte eines analogen CANopen-Moduls in Excel übertragen und dort zur Weiterverarbeitung, also z.B. zur Erstellung eines Diagramms, genutzt werden können.

Die Visualisierung von XML-basierten Roboter-Profilen mittels Java 3D erlaubt die dynamische Visualisierung von Robotern in einem 3-dimensionalen Raum. Die vorgestellte Infrastruktur erzeugt aus einem XML-Dokument eine graphische Repräsentation von hierarchisch angeordneten Manipulatorarmen. Durch die Verwendung von graphischen Primitiva kann eine sehr detaillierte Ansicht eines Roboters erreicht werden. Zur Integration eines realen Roboters wurde server-seitig mit MAX entsprechende Middleware zur verteilten Kommunikation erzeugt, die client-seitig mittels Java 3D visualisiert werden. Java 3Derlaubt dabei die Drehung, Skalierung und Translation des visualisierten Roboters in einem 3-dimensionalen Raum.

7.2 Ausblick

Die Erweiterung der bestehenden Infrastruktur zur Metaisierung um weitere Technologien würde die Möglichkeiten des Systems nochmals vergrößern. Dabei kommen sowohl die Integration von weiteren Programmiersprachen als auch von anderen Systemen in Betracht. Notwendige Bedingung für die Integration einer Programmiersprache ist die Verfügbarkeit einer prozeduralen oder objektorientierten Schnittstelle und einer Middleware zur Integration der Sprache in Java. Eine mögliche Erweiterung ist die Integration von Datenbanken. Moderne Datenbanksysteme erlauben die Extraktion von Metadaten, so dass die vorhandenen Tabellen und deren Attribute bestimmt werden können. Wird eine Tabelle ebenfalls als abstrakter Datentyp gesehen, so könnten hier entsprechende Objekte zur Kommunikation mit einer Datenbank automatisch generiert werden, wobei für das Setzen und Lesen von Attributen entsprechende Methoden erzeugt werden. Als mögliches Anwendungsszenario ist in diesem Fall eine Kombination von Automationssystemen mit Datenbanken analog zur Integration von CANopen und COM gemäß Abschnitt 6.3.4 denkbar.

Gemäß Abschnitt 5.7 könnten mit MAX verschiedene nicht-funktionale Eigenschaften realisiert werden. Zu diesen Eigenschaften zählen beispielsweise Synchronisation, Lastbalancierung, Verschlüsselung, Datenkomprimierung, Protokollierung, Ablaufverfolgung, uvm. Desweiteren könnten die (serialisierten) Metamodelle zum Reverse--Engineering oder zur Realisierung eines Objektbrowsers (s. Abschnitt 5.6.1) oder zur Erstellung von Statistiken genutzt werden. Die Erstellung von Statistiken wäre vor allem in Verbindung mit einer Datenbank interessant.

In Abschnitt 6.2.3 wurde bereits angedeutet, dass die passive Metaisierung und Generierung für dynamische Umgebungen eingesetzt werden können. Konkret hätten Software-Agenten damit die Möglichkeit, das System autonom zu nutzen. Ebenfalls denkbar wäre eine Kombination aus Software-Agenten und CANopen-Komponenten, so dass Agenten zur Steuerung von autonomen Automationssystemen eingesetzt werden könnten.

Eine Kombination aus metaisierten Komponenten und der Java 3D-Visualisierung würde eine graphische Programmierung von Systemen ermöglichen. Die Idee ist, XML-basierte Beschreibungen direkt miteinander zu verknüpfen und die notwendigen Adapter automatisch zu generieren. Bezogen auf die vorgestellte Infrastruktur zur Steuerung und Visualisierung eines entfernten Roboters, würden die Antriebskomponenten mit der zugehörigen Achse verbunden werden. Eine notwendige Umsetzung von Daten (z.B. Umrechnung von Positionswerten) würden von einem entsprechendem Adapter realisiert werden.

Während in der vorliegenden Arbeit eine Metaebene als temporäre Instanz angesehen wird, könnte sie auch ebenso als ein allgemeines Programmier- und Ausführungsmodell betrachtet werden. Falls viele unterschiedliche Programmiersprachen ein einheitliches Metamodell und eine einheitliche Schnittstelle zur Reflexion bieten, könnte eine Programmierung unabhängig von der verwendeten Programmiersprache realisiert werden. Die Operationen einer Metaebene würden in diesem Fall zur Laufzeit auf konkrete Operation einer bestimmten Programmiersprache abgebildet und ausgeführt werden. In der allgemeinen Verwendung einer Metaebene wird daher ein erhebliches Potential gesehen.

Im Rahmen dieser Arbeit sind folgende Publikationen erschienen (in chronologischer Reihenfolge):

1. GRUHLER, GERHARD, WOLFGANG KÜCHLIN, GERD NUSSER und DIETER BÜHLER: *Internet-basiertes Labor für Automatisierungstechnik und Informatik*. In: SCHMID, D. (Herausgeber): *Virtuelles Labor: Ihr Draht in die Zukunft*, Seiten 27–36, Ellwangen, Deutschland, Oktober 1999. R. Wimmer Verlag.

2. GRUHLER, GERHARD, GERD NUSSER, DIETER BÜHLER und WOLFGANG KÜCHLIN: *Teleservice of CAN systems via Internet*. In: *Proceedings of the 6^{th} Intl. CAN Conference (ICC '99)*, Seiten 06/02–06/09, Torino, Italy, November 1999. CAN in Automation (CiA).

3. BÜHLER, DIETER, GERD NUSSER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *A Java Client Server System for Accessing Arbitrary CANopen Fieldbus Devices via the Internet*. South African Computer Journal, (24):239–243, November 1999.

4. GRUHLER, GERHARD und GERD NUSSER: *Der Brückenschlag. Fernzugriff via Internet*. Computer und Automation, (3):58–61, März 2000.

5. BÜHLER, DIETER, WOLFGANG KÜCHLIN, GERHARD GRUHLER und GERD NUSSER: *The Virtual Automation Lab - Web Based Teaching of Automation Engineering Concepts*. In: *Proceedings of the 7^{th} IEEE Intl. Conference on the Engineering of Computer Based Systems (ECBS 2000)*, Seiten 156–164, Edinburgh, Scotland, April 2000. IEEE Computer Society Press.

6. SCHIMKAT, RALF-DIETER, GERD NUSSER und DIETER BÜHLER: *Scalability and Interoperability in Service-Centric Architectures for the Web*. In: *Proceedings of the 11^{th} Intl. Workshop on Network Based Information Systems (NBIS '00)*, Seiten 51–57, Greenwhich, London, Great Britain, September 2000. IEEE Computer Society Press.

7. BÜHLER, DIETER und GERD NUSSER: *The Java CAN API - A Java Gateway to Fieldbus Communication*. In: *Proceedings of the 2000 IEEE Intl. Workshop on Factory Communications Systems (WFCS 2000)*, Porto, Portugal, September 2000. IEEE.

8. NUSSER, GERD und GERHARD GRUHLER: *Dynamic Device Management and Access based on Jini and CAN*. In: *Proceedings of the 7^{th} Intl. CAN Conference (ICC 2000)*, Seiten 4/02–4/09, Amsterdam, Netherlands, Oktober 2000. CAN in Automation (CiA).

9. BÜHLER, DIETER, GERD NUSSER, WOLFGANG KÜCHLIN und GERHARD GRUHLER: *The Java Fieldbus Control Framework – Object-oriented Control of Fieldbus Devices*. In: *Proceedings of the 4^{th} IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, Mai 2001. IEEE Computer Society Press.

10. NUSSER, GERD und RALF-DIETER SCHIMKAT: *Rapid Application Development of Middleware Components by Using XML*. In: *Proceedings of the 12th IEEE Intl. Workshop on Rapid Systems Prototyping (RSP 2001)*, Seiten 116–121, Monterey, CA, USA, Juni 2001. IEEE Computer Society Press.

11. NUSSER, GERD, DIETER BÜHLER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Reality-driven Visualization of Automation Systems via the Internet based on Java and XML*. In: *Proceedings of the 1st IFAC Intl. Conference on Telematics Applications in Automation and Robotics (TA 2001)*, Seiten 407–412, Weingarten, Germany, Juli 2001. Elsevier Science Publishers.

12. GRUHLER, GERHARD, WOLFGANG KÜCHLIN, DIETER BÜHLER und GERD NUSSER: *Internet-Based Lab Assignments in Automation Engineering and Computer Science*. In: SCHILLING, K., H. ROTH und O. ROESCH (Herausgeber): *Proceedings of the Intl. Workshop on Tele-Education in Mechatronics Based on Virtual Laboratories*, Ellwangen, Germany, Juli 2001. R. Wimmer Verlag.

13. NUSSER, GERD, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Automatic Generation of Control Software Components for CAN Devices by Using Java and XML*. In: *Proceedings of the 8th Intl. CAN Conference (ICC 2002)*, Seiten 08/11–08/18, Las Vegas, NV, USA, Februar 2002. CAN in Automation (CiA).

14. FRIEDRICH, MICHAEL, GERD NUSSER und WOLFGANG KÜCHLIN: *Maintenance of Distributed Systems with Mobile Agents*. In: *Proceedings of the 18th Intl. Conference on Software Maintenance (ICSM 2002)*, Seiten 659–665, Montreal, Quebec, Canada, Oktober 2002. IEEE.

15. FRIEDRICH, MICHAEL, KIRSTEN TERFLOTH, GERD NUSSER und WOLFGANG KÜCHLIN: *Mobile Agents: A Construction Kit for Mobile Device Applications*. In: *Proceedings of the 5th Intl. Conference on Internet Computing (IC 2004)*, Las Vegas, NV, USA, Juni 2004. (to appear).

A

XML-Dokumente und DTDs

```
<!ELEMENT MetaProperties (MetaProperty+)>
<!ELEMENT MetaProperty (MetaProperties?)>
<!ATTLIST MetaProperty
  name CDATA #REQUIRED
  type CDATA #REQUIRED
  value CDATA #REQUIRED
>
<!ELEMENT MetaType (MetaProperties?, MetaFields?, MetaOperations?, MetaEvents?)>
<!ATTLIST MetaType
  name CDATA #REQUIRED
  namespace CDATA #IMPLIED
>
<!ELEMENT MetaFields (MetaField+)>
<!ELEMENT MetaField (MetaType, MetaProperties?, MetaValue?)>
<!ATTLIST MetaField
  name CDATA #REQUIRED
>
<!ELEMENT MetaOperations (MetaOperation+)>
<!ELEMENT MetaOperation (MetaType, MetaProperties?, MetaOperands?, MetaExceptions?)>
<!ATTLIST MetaOperation
  name CDATA #REQUIRED
>
<!ELEMENT MetaOperands (MetaOperand+)>
<!ELEMENT MetaOperand (MetaType, MetaProperties?)>
<!ATTLIST MetaOperand
  name CDATA #REQUIRED
>
<!ELEMENT MetaExceptions (MetaException+)>
<!ELEMENT MetaException (MetaType, MetaProperties?)>
<!ATTLIST MetaException
  name CDATA #REQUIRED
>
<!ELEMENT MetaEvents (MetaEvent+)>
<!ELEMENT MetaEvent (MetaType, MetaProperties?)>
<!ATTLIST MetaEvent
  name CDATA #REQUIRED
>
<!ELEMENT MetaTypes ((xi:include+) | (MetaType+))>
<!ELEMENT xi:include EMPTY>
<!ATTLIST xi:include
  href CDATA #REQUIRED
>
```

Listing A.1: MAXML-DTD

```
<MetaType name="_DotNetTypeLoader" namespace="java2dotnet">
 <MetaProperties>
  <MetaProperty name="IID"
                type="java2com.GUID"
                value="{0361DAD8-7BCF-32FD-BE01-07D6C8C51727}" />
  <MetaProperty name="ITypeInfo.TYPEKIND"
                type="int"
                value="TKIND_DISPATCH" />
  <MetaProperty name="ProgID"
                type="string"
                value="java2dotnet.DotNetTypeLoader" />
 </MetaProperties>
 <MetaOperations>
  <MetaOperation name="getType_2">
   <MetaType name="_Type" namespace="mscorlib">
    <MetaProperties>
     <MetaProperty name="IDLType"
                   type="string"
                   value="VT_PTR+VT_USERDEFINED(3000019):_Type"/>
    </MetaProperties>
   </MetaType>
   <MetaProperties>
    <MetaProperty name="wFlags" type="string" value="INVOKE_FUNC"/>
   </MetaProperties>
   <MetaOperands>
    <MetaOperand name="typeName">
      <MetaType name="string">
        <MetaProperties>
        <MetaProperty name="IDLType" type="string" value="VT_BSTR"/>
       <MetaProperty name="IDLValue" type="int" value="1"/>
       </MetaProperties>
      </MetaType>
    </MetaOperand>
   </MetaOperands>
  </MetaOperation>
  <MetaOperation name="getAssemblyFromFile">
   <MetaType name="_Assembly" namespace="mscorlib">
    <MetaProperties>
     <MetaProperty name="IDLType"
                   type="string"
                   value="VT_PTR+VT_USERDEFINED(3000025):_Assembly"/>
    </MetaProperties>
   </MetaType>
   <MetaProperties>
    <MetaProperty name="wFlags" type="string" value="INVOKE_FUNC"/>
   </MetaProperties>
   <MetaOperands>
    <MetaOperand name="assemblyFile">
     <MetaType name="string">
      <MetaProperties>
       <MetaProperty name="IDLType" type="string" value="VT_BSTR"/>
       <MetaProperty name="IDLValue" type="int" value="1"/>
      </MetaProperties>
     </MetaType>
    </MetaOperand>
   </MetaOperands>
  </MetaOperation>
   ...
 </MetaOperations>
</MetaType>
```

Listing A.2: Die Klasse _DotNetTypeLoader in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<Service name="Jini" version="1.1">
 <MetaProperties>
  <MetaProperty name="Manager" type="string" value="RMISecurityManager"/>
  <MetaProperty name="Name" type="string" value="DotNetTypeLoaderService"/>
```

```
  <MetaProperty name="Address"
                type="net.jini.lookup.entry.Address"
                value="address">
   <MetaProperties>
    <MetaProperty name="country" type="string" value="Germany"/>
    <MetaProperty name="locality" type="string" value="Tuebingen"/>
    <MetaProperty name="organizaion" type="string" value="WSI" />
    <MetaProperty name="organizationalUnit" type="string" value="SC"/>
    <MetaProperty name="postalCode" type="string" value="72076"/>
    <MetaProperty name="stateOrProvince" type="string" value="BW"/>
    <MetaProperty name="street" type="string" value="Sand 13"/>
   </MetaProperties>
  </MetaProperty>
  <MetaProperty name="Comment"
                type="net.jini.lookup.entry.Comment"
                value="comment">
   <MetaProperties>
    <MetaProperty name="comment" type="string" value="comment"/>
   </MetaProperties>
  </MetaProperty>
  <MetaProperty name="Location"
                type="net.jini.lookup.entry.Location"
                value="comment">
   <MetaProperties>
    <MetaProperty name="building" type="string" value="Sand 13"/>
    <MetaProperty name="floor" type="string" value="1"/>
    <MetaProperty name="room" type="string" value="007"/>
   </MetaProperties>
  </MetaProperty>
  <MetaProperty name="Name"
                type="net.jini.lookup.entry.Name"
                value="name">
   <MetaProperties>
    <MetaProperty name="name" type="string" value="Service description"/>
   </MetaProperties>
  </MetaProperty>
  <MetaProperty name="ServiceInfo"
                type="net.jini.lookup.entry.ServiceInfo"
                value="serviceinfo">
   <MetaProperties>
    <MetaProperty name="manufacturer" type="string" value="WSI"/>
    <MetaProperty name="model" type="string" value="WSI"/>
    <MetaProperty name="name" type="string" value="ServiceName"/>
    <MetaProperty name="serialNumber" type="string" value="42"/>
    <MetaProperty name="vendor" type="string" value="WSI"/>
    <MetaProperty name="version" type="string" value="V0.1"/>
   </MetaProperties>
  </MetaProperty>
 </MetaProperties>
</Service>
```

Listing A.3: Beispiel einer Jini-Konfiguration

Das Build-Dokument in Listing A.4 enthält alle notwendigen Anweisungen zur Übersetzung und Archivierung der COM-Komponente Excel. Da die Komponente externe COM-Bibliotheken (Office, VBIDE) referenziert, werden diese ebenfalls übersetzt und archiviert. Neben Anweisungen zur Übersetzung der Klassenbibliotheken, enthält das Dokument noch Anweisungen zur Generierung der API-Dokumentation mittels dem Werkzeug *javadoc*.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Excel" default="Excel" basedir=".">
  <target name="Excel" >
    <javac srcdir="." destdir="." includes="Excel\**"/>
  </target>
```

```
  <target name="VBIDE">
    <javac srcdir="." destdir="." includes="VBIDE\**"/>
  </target>
  <target name="Office">
    <javac srcdir="." destdir="." includes="Office\**"/>
  </target>
  <target name="jars" depends="Excel">
    <jar jarfile="Excel.jar" basedir="." includes="Excel\**\*.class"/>
    <jar jarfile="VBIDE.jar" basedir="." includes="VBIDE\**\*.class"/>
    <jar jarfile="Office.jar" basedir="." includes="Office\**\*.class"/>
  </target>
  <target name="docs" depends="Excel">
    <mkdir dir="docs\Excel"/>
    <javadoc sourcepath="Excel"
             destdir="docs\Excel"
             sourcefiles="Excel\*.java"/>
    <mkdir dir="docs\VBIDE"/>
    <javadoc sourcepath="VBIDE"
             destdir="docs\VBIDE"
             sourcefiles="VBIDE\*.java"/>
    <mkdir dir="docs\Office"/>
    <javadoc sourcepath="Office"
             destdir="docs\Office"
             sourcefiles="Office\*.java"/>
  </target>
</project>
```

Listing A.4: Beispiel eines ANT XML-Dokuments

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Robot (Link)>
<!ELEMENT Link (Boundingbox, Link?)>
<!ATTLIST Link
  LinkType (Type1 | Type2 | Type3 | Type4 | Type5 |
            Type6 | Type7 | Type8 | Type9 | Type10) #REQUIRED
  DegreeMin CDATA #IMPLIED
  DegreeMax CDATA #IMPLIED
  ZeroPosition CDATA #IMPLIED
  zAxis (positive | negative) "positive"
>
<!ELEMENT Boundingbox (Box | Cylinder | Sphere)*>
<!ATTLIST Boundingbox
  Width CDATA #REQUIRED
  Depth CDATA #REQUIRED
  Offset CDATA #IMPLIED
  HeightJoint0 CDATA #REQUIRED
  HeightLink CDATA #REQUIRED
  HeightJoint1 CDATA #REQUIRED
>
<!ELEMENT Box EMPTY>
<!ATTLIST Box
  Height CDATA #REQUIRED
  Width CDATA #REQUIRED
  Depth CDATA #REQUIRED
  LowerLeftFrontEdgeX CDATA #REQUIRED
  LowerLeftFrontEdgeY CDATA #REQUIRED
  LowerLeftFrontEdgeZ CDATA #REQUIRED
  RotX CDATA #IMPLIED
  RotY CDATA #IMPLIED
  RotZ CDATA #IMPLIED
  Basis (joint0 | link | joint1) "joint0"
>
<!ELEMENT Cylinder EMPTY>
<!ATTLIST Cylinder
  Height CDATA #REQUIRED
  Radius CDATA #REQUIRED
  LowerCircleCenterX CDATA #REQUIRED
  LowerCircleCenterY CDATA #REQUIRED
```

```
  LowerCircleCenterZ CDATA #REQUIRED
  RotX CDATA #IMPLIED
  RotZ CDATA #IMPLIED
  Basis (joint0 | link | joint1) "joint0"
>
<!ELEMENT Sphere EMPTY>
<!ATTLIST Sphere
  Radius CDATA #REQUIRED
  CenterX CDATA #REQUIRED
  CenterY CDATA #REQUIRED
  CenterZ CDATA #REQUIRED
  Basis (joint0 | link | joint1) "joint0"
>
```

Listing A.5: Robot-DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Robot SYSTEM "Robot.dtd">
<Robot>
 <!-- 1.Link -->
 <Link LinkType="Type4">
  <Boundingbox Depth="166"
               HeightLink="589"
               HeightJoint1="60"
               HeightJoint0="0"
               Width="200"/>
  <!-- 2.Link -->
  <Link LinkType="Type1"
        zAxis="negative"
        DegreeMax="140"
        ZeroPosition="0"
        DegreeMin="14 ">
   <Boundingbox Depth="60"
                HeightLink="250"
                HeightJoint1="80"
                HeightJoint0="200"
                Width="166"
                Offset="200"/>
   <!-- 3.Link -->
   <Link LinkType="Type7"
         DegreeMax="150"
         ZeroPosition="0"
         DegreeMin="150">
    <Boundingbox Depth="506"
                 HeightLink="100"
                 HeightJoint1="105"
                 HeightJoint0="80"
                 Width="105"/>
    <!-- 4.Link -->
    <Link LinkType="Type9"
          ZeroPosition="0">
     <Boundingbox Depth="105"
                  HeightLink="180"
                  HeightJoint1="0"
                  HeightJoint0="200"
                  Width="105"/>
    </Link>
   </Link>
  </Link>
 </Link>
</Robot>
```

Listing A.6: XML-Dokument zur grob-granularen Beschreibung des Scara SR60

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Robot SYSTEM "Robot.dtd">
<!-- Bosch SR60-->
```

```
<Robot>
 <!-- 1.Link -->
 <Link LinkType="Type4">
  <Boundingbox Width="200" Depth="166"
               HeightJoint0="0" HeightLink="589" HeightJoint1="60">
   <!-- Geometric primitives -->
   <Box Height="559" Width="160" Depth="110" LowerLeftFrontEdgeX="100"
        LowerLeftFrontEdgeY="0" LowerLeftFrontEdgeZ="-28"
        RotX="0" RotY="0" RotZ="0" Basis="joint0"/>
   <Box Height="30" Width="180" Depth="120" LowerLeftFrontEdgeX="110"
        LowerLeftFrontEdgeY="559" LowerLeftFrontEdgeZ="-23"
        RotX="0" RotY="0" RotZ="0" Basis="joint0"/>
   <Cylinder Height="559" Radius="83" LowerCircleCenterX="100"
             LowerCircleCenterY="0" LowerCircleCenterZ="-83"
             RotX="0" RotZ="0" Basis="joint0"/>
   <Cylinder Height="30" Radius="93" LowerCircleCenterX="100"
             LowerCircleCenterY="559" LowerCircleCenterZ="-83"
             RotX="0" RotZ="0" Basis="joint0"/>
   <!-- Geometric primitives -->
   <Cylinder Height="60" Radius="83" LowerCircleCenterX="100"
             LowerCircleCenterY="0" LowerCircleCenterZ="-83"
             RotX="0" RotZ="0" Basis="joint1"/>
   <Box Height="160" Width="60" Depth="60" LowerLeftFrontEdgeX="70"
        LowerLeftFrontEdgeY="0" LowerLeftFrontEdgeZ="-53"
        RotX="0" RotY="0" RotZ="0" Basis="joint1"/>
  </Boundingbox>
  <!-- 2.Link -->
  <Link LinkType="Type1" DegreeMin="140" DegreeMax="140"
        ZeroPosition="0" zAxis="negative">
   <Boundingbox Width="166" Depth="60" Offset="100"
                HeightJoint0="200" HeightLink="250" HeightJoint1="80">
   <!-- Geometric primitives: 1.Joint-->
   <Cylinder Height="60" Radius="83" LowerCircleCenterX="83"
             LowerCircleCenterY="83" LowerCircleCenterZ="0"
             RotX="90" RotZ="0" Basis="joint0"/>
    <!-- Geometric primitives: Link -->
    <Box Height="400" Width=" 166" Depth="60" LowerLeftFrontEdgeX="0"
         LowerLeftFrontEdgeY="-70" LowerLeftFrontEdgeZ="30"
         RotX="0" RotY="0" RotZ="0" Basis="link"/>
    <!-- Geometric primitives: 2.Joint -->
    <Cylinder Height="60" Radius="83" LowerCircleCenterX="83"
              LowerCircleCenterY="40" LowerCircleCenterZ="0"
              RotX="90" RotZ="0" Basis="joint1"/>
   </Boundingbox>
   <!-- 3.Link -->
   <Link LinkType="Type7" DegreeMin="150" DegreeMax="150"
         ZeroPosition="0">
    <Boundingbox Width="105" Depth="506"
                 HeightJoint0="80" HeightLink="100" HeightJoint1="105">
     <!-- Geometric primitives: 1.Joint -->
     <Cylinder Height="60" Radius="83" LowerCircleCenterX="50"
               LowerCircleCenterY="45" LowerCircleCenterZ="-430"
               RotX="90" RotZ="0" Basis="joint0"/>
     <Box Height="60" Width="60" Depth="200" LowerLeftFrontEdgeX="22"
          LowerLeftFrontEdgeY="25" LowerLeftFrontEdgeZ="-400"
          RotX="0" RotY="0" RotZ="0" Basis="joint0"/>
     <!-- Geometric primitives: Link -->
     <Box Height="200" Width="70" Depth="50" LowerLeftFrontEdgeX="15"
          LowerLeftFrontEdgeY="0" LowerLeftFrontEdgeZ="-400"
          RotX="0" RotY="0" RotZ="0" Basis="link"/>
     <Box Height="200" Width="70" Depth="50" LowerLeftFrontEdgeX="15"
          LowerLeftFrontEdgeY="0" LowerLeftFrontEdgeZ="-550"
          RotX="0" RotY="0" RotZ="0" Basis="link"/>
     <!-- Geometric primitives for: 2.Joint -->
     <Cylinder Height="600" Radius="65" LowerCircleCenterX="50"
               LowerCircleCenterY="-250" LowerCircleCenterZ="-300"
               RotX="90" RotZ="0" Basis="joint1"/>
    </Boundingbox>
```

```
    <!-- 4.Link -->
    <Link LinkType="Type9" ZeroPosition="0">
     <Boundingbox Width="105" Depth="105" HeightJoint0="200"
                  HeightLink="180" HeightJoint1="0">
      <!-- Geometric primitives: 1.Joint -->
      <Cylinder Height="250" Radius="30" LowerCircleCenterX="50"
                LowerCircleCenterY="200" LowerCircleCenterZ="-50"
                RotX="0" RotZ="0" Basis="joint0"/>
      <Cylinder Height="30" Radius="60" LowerCircleCenterX="50"
                LowerCircleCenterY="450" LowerCircleCenterZ="-50"
                RotX="0" RotZ="0" Basis="joint0"/>
     </Boundingbox>
    </Link>
   </Link>
  </Link>
 </Link>
</Robot>
```

Listing A.7: XML-Dokument zur fein-granularen Beschreibung des Scara SR60

```
<?xml version="1.0"?>
<Typelib progid="Outlook.Application" guid="..."
         majorversion="9" minorversion="0" >
  <enum name="OlActionCopyLike">
    <var name="olReply" kind="VAR_CONST" type="int" value="0"/>
    <var name="olReplyAll" kind="VAR_CONST" type="int" value="1"/>
    <var name="olForward" kind="VAR_CONST" type="int" value="2"/>
    <var name="olReplyFolder" kind="VAR_CONST" type="int" value="3"/>
    <var name="olRespond" kind="VAR_CONST" type="int" value="4"/>
  </enum>
  <Dispatch name="ApplicationEvents" iid="..." entryinfo="false">
    <funcdesc name="ItemSend"  dispid="61442">
      <kind value="IDispatch.METHOD"/>
      <return com_type="VT_VOID" type="void"/>
      <param com_type="VT_DISPATCH" type="IDispatch"
             flags="1" name="Item"/>
      <param com_type="VT_PTR+VT_BOOL" type="boolean[]"
             flags="1" name="Cancel"/>
    </funcdesc>
    <funcdesc name="Quit" dispid="61447">
      <kind value="IDispatch.METHOD"/>
      <return com_type="VT_VOID" type="void"/>
    </funcdesc>
  </Dispatch>
  <CoClass name="TaskRequestDeclineItem"  iid="...">
    <typedoc doc="_TaskRequestDeclineItem"/>
    <typedoc doc="ItemEvents"/>
    <typedoc doc="_TaskRequestDeclineItem"/>
  </CoClass>
  <interface name="_IDocSiteControl">
    <funcdesc name="ReadOnly"  dispid="-2147356664">
      <kind value="IDispatch.PROPERTYGET"/>
      <return com_type="VT_HRESULT" type="HRESULT"/>
      <param com_type="VT_PTR+VT_BOOL" type="boolean[]"
             flags="10" name="ReadOnly"/>
    </funcdesc>
    <funcdesc name="ReadOnly"  dispid="-2147356664">
      <kind value="IDispatch.PROPERTYPUT"/>
      <return com_type="VT_HRESULT" type="HRESULT"/>
      <param com_type="VT_BOOL" type="boolean"
             flags="1" name="ReadOnly"/>
    </funcdesc>
  </interface>
</Typelib>
```

Listing A.8: XML-Repräsentation von COM-Typbibliotheken in Jacob aus [Lew01]

B

Datentypen

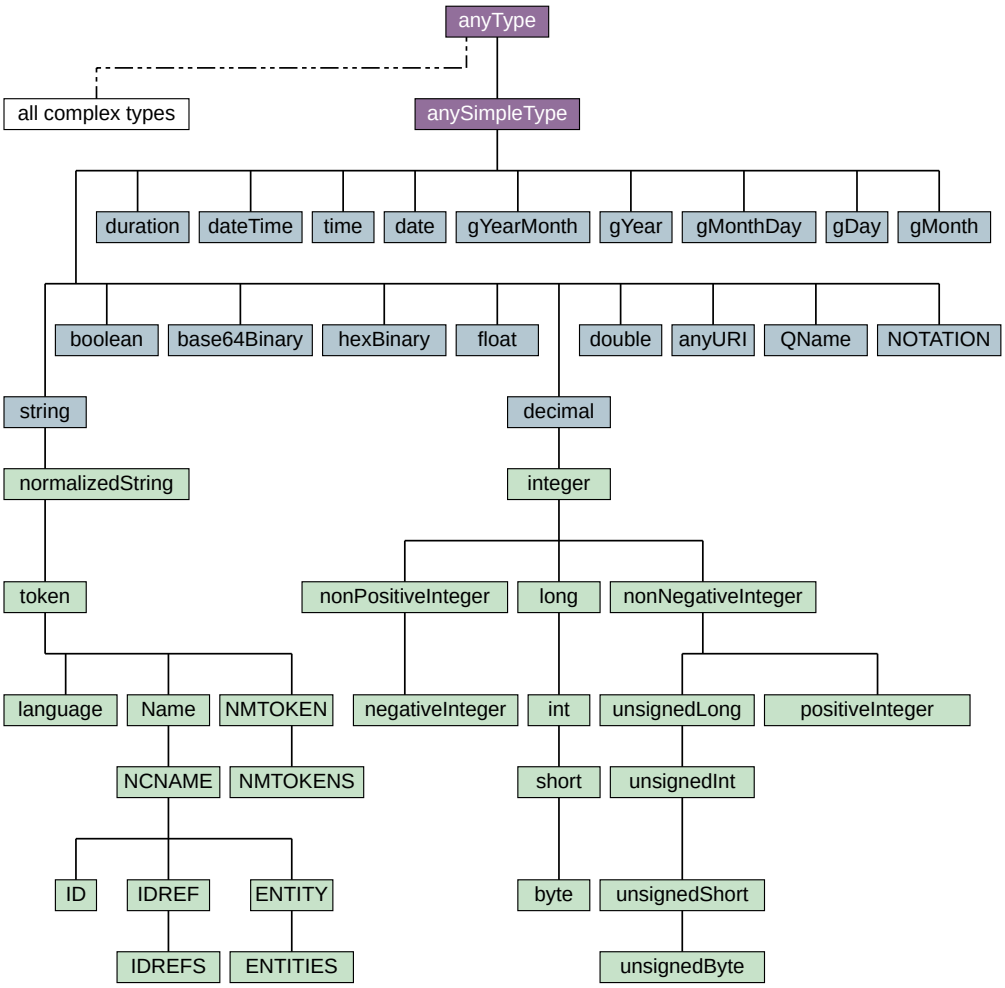


Abbildung B.1: XMLSchema-Datentypen des W^{3}C [Wor01c]

COM-Datentyp	Beschreibung	Java-Datentyp
VT_BOOL	boolean value	boolean
VT_I1	1-byte signed integer	byte
VT_UI1	unsigned char	short
VT_I2	2-byte signed integer	short
VT_UI2	2-byte unsigned integer	int
VT_I4	4-byte signed integer	int
VT_UI4	4-byte unsigned integer	long
VT_I8	8-byte signed integer	long
VT_UI8	8-byte unsigned integer	long
VT_R8	8-byte real	double
VT_BSTR	automation string	java.lang.String
VT_LPSTR	null-terminated string	java.lang.String
VT_DATE	date value	java.util.Date
VT_VOID	C-style void	void
VT_PTR+VT_VOID	C-style void pointer	int
VT_VARIANT	VARIANT far pointer	java.lang.Object
VT_DISPATCH	IDispatch pointer	java2com.IDispatch
VT_UNKNOWN	IUnknown pointer	java2com.IUnknown
VT_HRESULT	HRESULT	long
VT_INT	signed machine integer	int
VT_CARRAY	C style array	byte[]
VT_ERROR	scodes	long
VT_CY	currency value	long

Tabelle B.1: COM-Datentypen nach [Mic01b] und deren Abbildung nach Java

Objecttyp	Beschreibung
Null	Objekt enthält keine Daten
Domain	Objekt mit variabler Anzahl an Daten, z.B. Programm-Code
Deftype	Objekt beschreibt einen Typ, z.B. Boolean, Unsigned16
Defstruct	Objekt definiert eine Struktur, z.B. PDOMapping
Var	Objekt enthält einen Wert eines einfachen Datentyps, z.B. Unsigned8
Array	Objekt mit mehreren Daten des gleichen einfachen Datentyps
Record	Objekt mit mehreren Daten unterschiedlichen einfachen Datentyps

Tabelle B.2: CANopen-Objekttypen gemäß CiA [CAN00a]

Index	Objekt	CANopen-Datentype	Java-Datentyp
0001	DEFTYPE	Boolean	boolean
0002	DEFTYPE	Integer8	byte
0003	DEFTYPE	Integer16	short
0004	DEFTYPE	Integer32	int
0005	DEFTYPE	Unsigned8	short
0006	DEFTYPE	Unsigned16	int
0007	DEFTYPE	Unsigned32	long
0008	DEFTYPE	Real32	float
0009	DEFTYPE	Visible_String	java.lang.String
000A	DEFTYPE	Octet_String	java.lang.String
000B	DEFTYPE	Unicode_String	java.lang.String
000C	DEFTYPE	Time_Of_Day	java.util.Date
000D	DEFTYPE	Time_Difference	long
000E	reserved		
000F	DEFTYPE	Domain	n/a
0010	DEFTYPE	Integer24	int
0011	DEFTYPE	Real64	double
0012	DEFTYPE	Integer40	long
0013	DEFTYPE	Integer48	long
0014	DEFTYPE	Integer56	long
0015	DEFTYPE	Integer64	long
0016	DEFTYPE	Unsigned24	long
0017	reserved		
0018	DEFTYPE	Unsigned40	long
0019	DEFTYPE	Unsigned48	long
001A	DEFTYPE	Unsigned56	long
001B	DEFTYPE	Unsigned64	long
001C-001F	reserved		
0020	DEFSTRUCT	PDO_Communication_Parameter	java2canopen.PDO
0021	DEFSTRUCT	PDO_Mapping	java2canopen.PDOMapping
0022	DEFSTRUCT	SDO_Parameter	java2canopen.SDO
0023	DEFSTRUCT	Identity	long
0024-003F	reserved		

Tabelle B.3: CANopen-Datentypen gemäß CiA [CAN00a] und deren Abbildung auf Java-Datentypen

C

Klassendiagramme

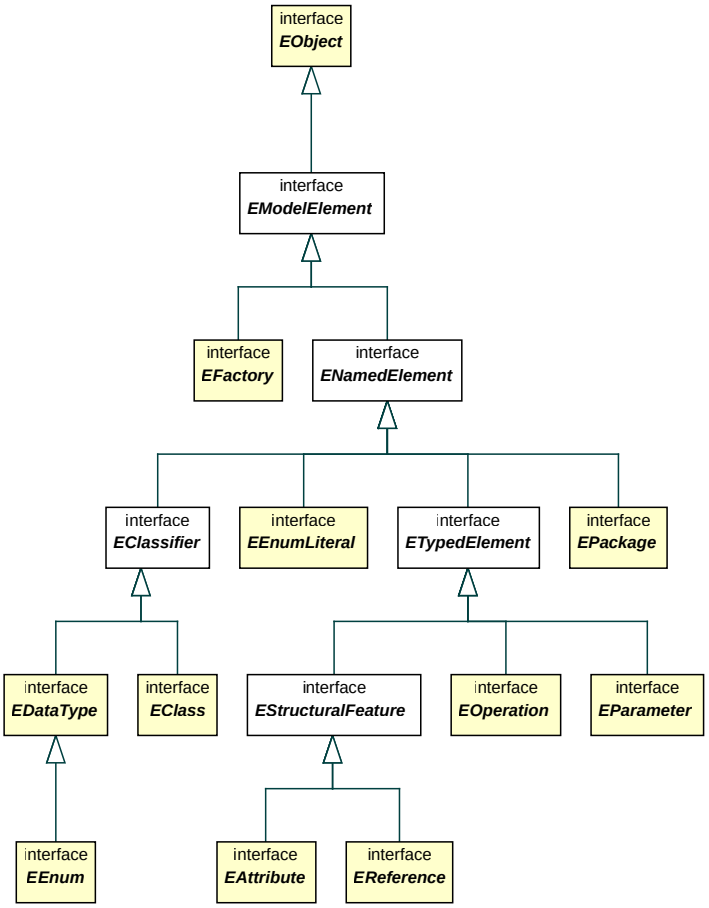


Abbildung C.1: Das Eclipse Core Model Framework (EMF) gemäß IBM [Ecl02]

Literaturverzeichnis

[Adl99] ADLER, DAN: *JACOB – Java/COM Bridge*. http://users.rcn.com/danadler/-jacob/, 1999.

[aJi04] AJILE SYSTEMS, INC., http://www.aJile.com/: *aJile realtime Embedded System*, Juni 2004.

[Alt99] ALT, RENÉ: *Visualisierung und Simulation von Robotern mit XML und Java 3D*. Diplomarbeit, Eberhard-Karls-Universität. Wilhelm-Schickard Institut für Informatik, Symbolisches Rechnen, Tübingen, Germany, August 1999.

[Apa04a] THE APACHE JAKARTA PROJECT, http://ant.apache.org/: *Another Neat Tool (ANT)*, Juni 2004.

[Apa04b] THE APACHE SOFTWARE FOUNDATION, http://httpd.apache.org/: *Apache HTTP Server Project*, Juni 2004.

[Apa04c] THE APACHE SOFTWARE FOUNDATION, http://jakarta.apache.org/tomcat/: *Apache Tomcat*, Juni 2004.

[Apa04d] THE APACHE SOFTWARE FOUNDATION, http://xml.apache.org/fop/: *Formatting Object Processor (FOP)*, März 2004.

[Apa04e] THE APACHE SOFTWARE FOUNDATION, http://cocoon.apache.org/: *The Apache Cocoon Project*, Juni 2004.

[Apa04f] THE APACHE SOFTWARE FOUNDATION, http://xml.apache.org/xalan-j/: *Xalan-Java 2*, 2004.

[Apa04g] THE APACHE SOFTWARE FOUNDATION, http://xml.apache.org/xerces2-j/: *Xerces XML Parser*, 2004.

[Azt02] AZTEC SOFTWARE AND TECHNOLOGY SERVICES LIMITED, http://www.-aztecsoft.com/j2cs: *Java to C# Converter*, 2002.

[BABR96] BELLISSARD, LUC, SLIM BEN ATALLAH, FABIENNE BOYER und MICHEL RIVEILL: *Distributed Application Configuration*. In: *Proceedings of the 16th Intl. Conference on Distributed Computing Systems (ICDCS '96)*, Seiten 579—585, Hong Kong, Mai 1996. IEEE Computer Society Press.

[Bad00] BADROS, GREG J.: *JavaML: A Markup Language for Java Source Code*. Computer Networks, 33:1–6, Juni 2000.

[Bal98] BALZERT, H.: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 1998.

[BB00] BRERETON, PEARL und DAVID BUDGEN: *Component-Based Systems: A Classification of Issues*. IEEE Computer, 33(11):54–62, November 2000.

[BBB+99] BRERETON, PEARL, DAVID BUDGEN, KEITH BENNETT, MALCOLM MUNRO, PAUL LAYZELL, LINDA MACAULAY, DAVID GRIFFITHS und CHARLES STANNETT: *The future of software*. Communications of the ACM, 42(12):78–84, 1999.

[BBRVD98] BELLISSARD, LUC, FABIENNE BOYER, MICHEL RIVEILL und JEAN-YVES VION-DURY: *System Services for Distributed Application Configuration*. In: *Proceedings of the 4th IEEE Intl. Conference on Configurable Distributed Systems (ICCDS'98)*, Annapolis, MD, USA, Mai 1998.

[Ber93] BERNSTEIN, PHILIP A.: *Middleware: An Architecture for Distributed System Services*. Technischer Bericht CRL 93/6, Digital Equipment Corporation, März 1993.

[Ber96] BERNSTEIN, PHILIP A.: *Middleware: A Model for Distributed System Services*. Communications of the ACM, 39(2):86–98, Februar 1996.

[Bet94] BETZ, MARK: *Interoperable objects*. Dr. Dobb's Journal: Software Tools for Professional Programmer, 19(11):18–39, Oktober 1994.

[BG03] BERNSTEIN, ABRAHAM und MARTIN GLINZ: *Informatik II: Systeme, Kommunikation und Modellierung*. Skriptum zur Vorlesung, 2003. http://-www.ifi.unizh.ch/groups/req/.

[BGW93] BOBROW, DANIEL G., RICHARD P. GABRIEL und JON L. WHITE: *CLOS in Context – The Shape of the Design Space*. In: PAEPCKE, A. (Herausgeber): *Object-Oriented Programming – The CLOS Perspective*. MIT Press, 1993.

[BK02] BEVERIDGE, MEREDITH und PHILIP KOOPMAN: *Jini Meets Embedded Control Networking: a case study in portability failure*. In: *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, San Diego, CA, USA, Januar 2002. IEEE.

[BKGN00] BÜHLER, DIETER, WOLFGANG KÜCHLIN, GERHARD GRUHLER und GERD NUSSER: *The Virtual Automation Lab - Web Based Teaching of Automation Engineering Concepts*. In: *Proceedings of the 7th IEEE Intl. Conference on the Engineering of Computer Based Systems (ECBS 2000)*, Seiten 156–164, Edinburgh, Scotland, April 2000. IEEE Computer Society Press.

[BL79] BELADY, L. A. und M. M. LEHMANN: *Characteristics of Large Systems*, Seiten 106–138. MIT Press, Juni 1979.

[BLV01] BETTINI, LORENZO, MICHELE LORETI und BETTI VENNERI: *On Multiple Inheritance in Java*. In: *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) Eastern Europe, Emerging Technologies, Emerging Markets*, Varna, Bulgaria, September 2001.

[BN84] BIRRELL, ANDREW D. und BRUCE JAY NELSON: *Implementing remote procedure calls*. ACM Transactions on Computer Systems (TOCS), 2(1):39–59, 1984.

[BN00] BÜHLER, DIETER und GERD NUSSER: *The Java CAN API - A Java Gateway to Fieldbus Communication*. In: *Proceedings of the 2000 IEEE Intl. Workshop on Factory Communications Systems (WFCS 2000)*, Porto, Portugal, September 2000. IEEE.

[BNGK99] BÜHLER, DIETER, GERD NUSSER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *A Java Client Server System for Accessing Arbitrary CANopen Fieldbus Devices via the Internet*. South African Computer Journal, (24):239–243, November 1999.

[BNKG01] BÜHLER, DIETER, GERD NUSSER, WOLFGANG KÜCHLIN und GERHARD GRUHLER: *The Java Fieldbus Control Framework – Object-oriented Control of Fieldbus Devices*. In: *Proceedings of the 4th IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, Mai 2001. IEEE Computer Society Press.

[Boe87] BOEHM, J. W.: *Improving software productivity*. IEEE Computer, 9(20):43–57, September 1987.

[Boo94] BOOCH, GRADY: *Object-Oriented Analysis and Design With Applications*. Addison-Wesley Publishing Company, Reading Massachusetts, 2. Auflage, 1994.

[BPP81] BRITTON, KATHRYN HENINGER, R. ALAN PARKER und DAVID L. PARNAS: *A Procedure for Designing Abstract Interfaces for Device Interface Modules*. Technischer Bericht Code 7590, Naval Research Laboratory, Washington, D.C. 20375, 1981.

[BR96] BELLISSARD, L. und M. RIVEILL: *From Distributed Objects to Distributed Components: the Olan Approach*. In: *ECOOP '96 Workshop on Putting Distributed Objects to Work*, Juli 1996.

[Bro99] BROWN, ALAN W.: *Mastering the Middleware Muddle*. IEEE Software, 16(4):18–21, Juli/August 1999.

[Bry88] BRYAN, MARTIN: *SGML: An Author's Guide to the Standard Generalized Markup Language*. Addison-Wesley Publishing Company, 1988.

[Bus95] BUSCHMANN, FRANK: *Was ist ein Entwurfsmuster?* Objekt Spektrum, (3):82–84, 1995.

[BW98] BROWN, ALAN W. und KURT C. WALLNAU: *The Current State of CBSE*. IEEE Software, 15(5):37–46, September/Oktober 1998.

[CAN98] CAN IN AUTOMATION (CIA): *Device Profile for Drives and Motion Control. CiA Draft Standard Proposal DSP-402, Version 1.1*, Oktober 1998.

[CAN99] CAN IN AUTOMATION (CIA): *Device Profile for Generic I/O Modules. CiA Draft Standard DS-401, Version 2.0*, Dezember 1999.

[CAN00a] CAN IN AUTOMATION (CIA): *CANopen Application Layer and Communication Profile. CiA Draft Standard DS-301, Version 4.01*, Juni 2000.

[CAN00b] CAN IN AUTOMATION (CIA): *Electronic Data Sheet Specification for CANopen, CiA Draft Standard Proposal 306, Version 1.0*, Mai 2000.

[CAN03] CAN IN AUTOMATION (CIA), http://www.can-cia.de/: *CiA Homepage*, Juni 2003.

[CCL01] CAZZOLA, WALTER, SHIGERU CHIBA und THOMAS LEDOUX: *Reflection and Meta-level Architectures: State of the Art and Future Trends*. Lecture Notes in Computer Science, 1964:1–15, 2001.

[Chi00] CHIBA, SHIGERU: *Load-Time Structural Reflection in Java*. Lecture Notes in Computer Science, 1850:313–336, 2000.

[Cle88] CLEAVELAND, J. CRAIG: *Building Application Generators*. IEEE Software, 5(6):25–33, Juli 1988.

[Coa92] COAD, PETER: *Object-oriented patterns*. Communications of the ACM, 35(9):152–159, 1992.

[Col03] COLLABNET, INC., https://javacc.dev.java.net/: *Java Compiler Compiler™(JavaCC™) – The Java Parser Generator*, 2003.

[Cou98] COULSON, G.: *A Distributed Object Platform Infrastructure for Multimedia Applications*. Computer Communications, 21(9):802–818, Juli 1998.

[Coy02] COYLE, FRANK P.: *XML, Web Services, and the Data Revolution*. Addison-Wesley Publishing Company, 2002.

[Dal04] DALLAS SEMICONDUCTOR, http://www.iButton.com/TINI: *Tiny Internet Interface (TINI)*, Juni 2004.

[DM95] DEMERS, FRANÇOIS-NICOLA und JACQUES MALENFANT: *Reflection in logic, functional and object-oriented programming: a Short Comparative Study*. In: *Proceedings of the Workshop on Reflection and Metalevel Architectures and their Applications in Artificial Intelligence (IJCAI '95)*, Seiten 29–38, Montreal, Canada, August 1995.

[Ear99] EARLES, JOHN: *COOL:Gen® – If you wrap it, they will comm.!* http://www.CBD-HQ.com, 1999.

[Eck00] ECKEL, BRUCE: *Thinking in C++ 2nd Edition*. Prentice Hall, Upper Saddle River, NJ 07458, 2000.

[Ecl02] ECLIPSE, http://www.eclipse.org/emf/: *The Eclipse Modelling Framework (EMF) Overview*, September 2002.

[Edw99] EDWARDS, W. KEITH: *Core Jini*. Sun Microsystems Press, 1999.

[Eil97] EILINGHOFF, CHRISTOPH: *Systematischer Einsatz von Software-Wiederverwendung bei der Entwicklung paralleler Programme*. Technischer Bericht tr-rsfb-97-035, Universität-Gesamthochschule Paderborn, Januar 1997.

[Eis95] EISENECKER, ULRICH W.: *Objekte versus Komponenten – Der Weg zur flinken Software*. iX, (9):164–169, 1995.

[Emm00] EMMERICH, WOLFGANG: *Software Engineering and Middleware: A Roadmap*. In: *Proceedings of the Conference on the Future of Software Engineering (FoSE)*, Seiten 117–129. ACM Press, 2000.

[EXO02a] EXOR INTERNATIONAL INC., http://www.embedding.net/eSOAP/: *Embedded SOAP (eSOAP)*, Juli 2002.

[EXO02b] EXOR INTERNATIONAL INC., http://www.embedding.net/icu/: *TpICU™*, September 2002.

[Fer89] FERBER, J.: *Computational reflection in class based object-oriented languages*. In: *Conference proceedings on Object-oriented programming systems, languages and applications*, Seiten 317–326, New Orleans, Louisiana, United States, 1989. ACM Press.

[FNK02] FRIEDRICH, MICHAEL, GERD NUSSER und WOLFGANG KÜCHLIN: *Maintenance of Distributed Systems with Mobile Agents*. In: *Proceedings of the 18th Intl. Conference on Software Maintenance (ICSM 2002)*, Seiten 659–665, Montreal, Quebec, Canada, Oktober 2002. IEEE.

[Foo92] FOOTE, BRIAN: *Objects, Reflection and Open Languages*. In: *Proceedings of the Workshop on Object-Oriented Reflection and Metalevel Architectures (ECOOP)*, Utrecht, Netherlands, 1992.

[Fre83] FREEMAN, P.: *Reusable software engineering: Concepts and research directions*. In: *Workshop on Reusability in Programming*, Seiten 2–16, Newport, R.I., September 1983. ITT Programming, Stratford, Conn.

[FTNK04] FRIEDRICH, MICHAEL, KIRSTEN TERFLOTH, GERD NUSSER und WOLFGANG KÜCHLIN: *Mobile Agents: A Construction Kit for Mobile Device Applications*. In: *Proceedings of the 5th Intl. Conference on Internet Computing (IC 2004)*, Las Vegas, NV, USA, Juni 2004. (to appear).

[GAO95] GARLAN, DAVID, ROBERT ALLEN und JOHN OCKERBLOOM: *Architectural Mismatch: Why Reuse is so hard*. IEEE Software, 12(6):17–26, November 1995.

[Ger97] GERKEN, MARK: *Software Technology Roadmap: Module Interconnection Languages*. Carnegie Mellon Software Engineering Institute, http://-www.sei.cmu.edu/str/, Januar 1997.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[GJSB00] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java Language Specification*. Addison-Wesley Publishing Company, Boston, Mass., 2. Auflage, Juni 2000.

[GK97a] GOLM, MICHAEL und JÜRGEN KLEINÖDER: *MetaJava*. In: *STJA '97*, Erfurt, Germany, September 1997.

[GK97b] GOLM, MICHAEL und JÜRGEN KLEINÖDER: *MetaJava – A Platform for Adaptable Operating System Mechanisms*. In: *Proceedings of the ECOOP '97 Workshop on Reflective Real-time Object-Oriented Programming and Systems*, Jyväskylä, Finland, Juni 1997.

[GK98] GOLM, MICHAEL und JÜRGEN KLEINÖDER: *metaXa and the Future of Reflection*. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, Vancouver, British Columbia, 1998.

[GKBN01] GRUHLER, GERHARD, WOLFGANG KÜCHLIN, DIETER BÜHLER und GERD NUSSER: *Internet-Based Lab Assignments in Automation Engineering and Computer Science*. In: SCHILLING, K., H. ROTH und O. ROESCH (Herausgeber): *Proceedings of the Intl. Workshop on Tele-Education in Mechatronics Based on Virtual Laboratories*, Ellwangen, Germany, Juli 2001. R. Wimmer Verlag.

[GKNB99] GRUHLER, GERHARD, WOLFGANG KÜCHLIN, GERD NUSSER und DIETER BÜHLER: *Internet-basiertes Labor für Automatisierungstechnik und Informatik*. In: SCHMID, D. (Herausgeber): *Virtuelles Labor: Ihr Draht in die Zukunft*, Seiten 27–36, Ellwangen, Deutschland, Oktober 1999. R. Wimmer Verlag.

[GNBK99] GRUHLER, GERHARD, GERD NUSSER, DIETER BÜHLER und WOLFGANG KÜCHLIN: *Teleservice of CAN systems via Internet*. In: *Proceedings of the 6th Intl. CAN Conference (ICC '99)*, Seiten 06/02–06/09, Torino, Italy, November 1999. CAN in Automation (CiA).

[Gol97] GOLM, MICHAEL: *Design and Implementation of a Meta Architecture for Java*. Diplomarbeit, Friedrich-Alexander-University Erlangen-Nürnberg, Germany. Computer Science Department, Operating Systems – IMMD IV, Januar 1997.

[Gru01] GRUHLER, GERHARD: *Internet-based Remote Access to Automation Systems*. In: *Proceedings of the Intl. Scientific and Technical Conference on Computer Technologies in Science, Education and Industry*, Seiten 199–203, Dnjepropetrovsk, Ucraine, Mai 2001.

[Ham97] HAMILTON, GRAHAM: *JavaBeans^{TM}, Version 1.01*. Sun Microsoft Corporation, http://java.sun.com/beans, 1997.

[HT01] HIMSTEDT, STEFFEN und STEFAN TREBING: *OPC geht online*. iee 46. Jahrgang, Nr. 3, Seiten 88–93, 2001.

[IBM04] IBM CORPORATION, http://www.eclipse.org: *Eclipse Project*, Juni 2004.

[ICS02] IVICA CRNKOVIC, STIG LARSSON und JUDITH STAFFORD: *Composing Systems From Components*. In: IVICA CRNKOVIC, STIG LARSSON und JUDITH STAFFORD (Herausgeber): *Proceedings of the 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems*, Lund University, Lund, Sweden, April 2002. IEEE.

[Int86] INTERNATIONAL ORGANISATION FOR STANDARDIZATION (ISO): *ISO 8879 – Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[Int93] INTERNATIONAL ORGANISATION FOR STANDARDIZATION (ISO): *ISO 11898 – Road Vehicles, Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication*, 1993.

[Int96] INTERNATIONAL ORGANISATION FOR STANDARDIZATION, INTERNATIONAL ELECTROTECHNICAL COMMISSION (ISO/IEC): *ISO/IEC 10179 – Document Style Semantics and Specification Language (DSSSL)*, 1996.

[Int01a] INTRINSIC: *JaNET Homepage (Bridging between Java and .NET)*. http://www.intrinsyc.com/products/bridging/janet.asp, 2001.

[Int01b] INTRINSYC: *J-Integra Homepage (Bridging between Java and COM)*. http://www.intrinsyc.com/products/bridging/jintegra.asp, 2001.

[INT04] INT MEDIA GROUP, INC., http://www.webopedia.com: *Wĕbopēdia*, Juni 2004.

[Jec01] JECKLE, MARIO: *Konzepte der Metamodellierung – Zum Begriff Metamodell*. http://www.jeckle.de/, Dezember 2001.

[Jec04] JECKLE, MARIO: *XML Metadata Interchange (XMI)*. http://www.jeckle.de/-xmi.htm, Juni 2004.

[JF88] JOHNSON, RALPH E. und BRIAN FOOTE: *Designing Reusable Classes*. Journal of Object-Oriented Programming, 1(2):22–35, 1988.

[JK00] JACOBSEN, H.-ARNO und BERND J. KRÄMER: *Modeling Interface Definition Language Extensions*. In: *Proceedings of the 37th Intl. Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37)*, Sydney, Australia, November 2000.

[JZ91] JOHNSON, RALPH E. und JONATHAN ZWEIG: *Delegation in C++*. Journal of Object-Oriented Programming, 4(11):22–35, November 1991.

[Kay04] KAY, MICHAEL H.: *Saxon*. http://saxon.sourceforge.net/, Juli 2004.

[KB98] KOZACZYNSKI, WOJTEK und GRADY BOOCH: *Component-Based Software Engineering*. IEEE Software, 15(5):34–36, September/Oktober 1998.

[KC00] KRAMP, THORSTEN und GEOFF COULSON: *The Design of a Flexible Communications Framework for Next-Generation-Middleware*. In: *Proceedings of the 2nd Intl. Symposium on Distributed Objects and Applications (DOA '00)*, Seite 273, Antwerp, Belgium, September 2000. IEEE.

[KG96a] KLEINÖDER, JÜRGEN und MICHAEL GOLM: *MetaJava: An Efficient Run-Time Meta-Architecture for Java*. In: *Proceedings of the Intl. Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, Seattle, Washington, USA, Oktober 1996. IEEE Computer Society Press.

[KG96b] KLEINÖDER, JÜRGEN und MICHAEL GOLM: *Transparent and Adaptable Object Replication Using a Reflective Java*. Technischer Bericht TR-I4-96-07, Friedrich-Alexander-University Erlangen-Nürnberg, Germany. Computer Science Department, Operating Systems – IMMD IV, September 1996.

[KG97] KLEINÖDER, JÜRGEN und MICHAEL GOLM: *MetaJava – A Platform for Adaptable Operating-System Mechanisms*. In: *Proceedings of the ECOOP '97 Workshop on Object-Orientation and Operating Systems*, Jyväskylä, Finland, Juni 1997.

[Kie98] KIELY, DON: *Are Components the Future of Software?* IEEE Computer, 31(2):10–11, Februar 1998.

[KJH+00] KIMY, KIMOON, GWANGIL JEON, SEONGSOO HONG, TAE-HYUNG KIM und SUNIL KIM: *Integrating Subscription-Based and Connection-Oriented Communications into the Embedded CORBA for the CAN Bus*. In: *4th IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, Washington, D.C., USA, Mai 2000. IEEE.

[Kni98] KNIESEL, GÜNTER: *Delegation for Java: API or Language Extension?* Technischer Bericht IAI-TR-98-5, University of Bonn, Germany, Mai 1998.

[Kni99] KNIESEL, GÜNTER: *Type-Safe Delegation for Run-Time Component Adaptation*. In: GUERRAOUI, R. (Herausgeber): *Proceedings of ECOOP '99*. Springer-Verlag Berlin Heidelberg, 1999.

[Kru92] KRUEGER, CHARLES W.: *Software Reuse*. ACM Computing Surveys (CSUR), 24(2):131–183, 1992.

[Lav01] LAVERS, TIM: *Java Tip 108: Apply RMI autogeneration*. http://www.-javaworld.com/javaworld/javatips/jw-javatip108.html, Februar 2001.

[Lew01] LEWIS, STEVEN: *Modifications to JACOB code for Generated Code*. http://-www.lordjoe.com/Java2Com/JacobProject.doc, März 2001.

[Lia99] LIANG, SHENG: *Java(TM) Native Interface: Programmer's Guide and Specification*. Addison-Wesley, Juni 1999.

[Lie86] LIEBERMAN, HENRY: *Using prototypical objects to implement shared behavior in object-oriented systems*. In: *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, Seiten 214–223, Portland, OR, November 1986.

[Lin99] LINDIG, CHRISTIAN: *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken*. Doktorarbeit, Technische Universität Braunschweig, Braunschweig, Germany, November 1999.

[Loo01] LOOS, PETER: *Go To COM*. Addison-Wesley, 2001.

[Lop97] LOPES, CRISTINA ISABEL VIDEIRA: *D: A Language Framework for Distributed Programming*. Doktorarbeit, Northeastern University, November 1997.

[Lyo02] LYON, DOUGLAS: *CentiJ: An RMI Code Generator*. Journal of Object Technology, 1(5):117–148, November/Dezember 2002.

[Mae87] MAES, PATTIE: *Concepts and Experiments in Computational Reflection*. In: *Proceedings of the OOPSLA-87: Conference on Object-Oriented Programming Systems*, Seiten 147–155, Languages and Applications, Orlando, FL, 1987.

[McI68] MCILROY, M. D.: *Mass produced software components*. In: BUXTON, J.M., P. NAUR und B. RANDELL (Herausgeber): *Software Engineering; Report on Conference by the NATO Science Committee*, Seiten 138–150. NATO Scientific Affairs Division, Brussels, Belgium, Oktober 1968.

[McK91] MCKERROW, P. J.: *Introduction in Robotics*. Addison-Wesley, Sydney, Australia, 1991.

[Mic01a] MICROSOFT CORPORATION, http://msdn.microsoft.com/library/en-us/-dndotnet/html/hawkremoting.asp: *Microsoft .NET Remoting: A Technical Overview*, Juli 2001.

[Mic01b] MICROSOFT CORPORATION: *Platform SDK Release, Automation*, August 2001.

[MNA+01] MERNIK, MARJAN, UROS NOVAK, ENIS AVDICAUSEVIC, MITJA LENIC und VILJEM ZUMER: *Design and Implementation of Simple Object Description Language*. In: *Proceedings of the 16th ACM Symposium on Applied Computing (SAC '01)*, Seiten 590–594, Las Vegas, NV, USA, März 2001. ACM.

[NBGK01] NUSSER, GERD, DIETER BÜHLER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Reality-driven Visualization of Automation Systems via the Internet based on Java and XML*. In: *Proceedings of the 1st IFAC Intl. Conference on Telematics Applications in Automation and Robotics (TA 2001)*, Seiten 407–412, Weingarten, Germany, Juli 2001. Elsevier Science Publishers.

[NG00] NUSSER, GERD und GERHARD GRUHLER: *Dynamic Device Management and Access based on Jini and CAN*. In: *Proceedings of the 7th Intl. CAN Conference (ICC 2000)*, Seiten 4/02–4/09, Amsterdam, Netherlands, Oktober 2000. CAN in Automation (CiA).

[NGK02] NUSSER, GERD, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Automatic Generation of Control Software Components for CAN Devices by Using Java and XML*. In: *Proceedings of the 8th Intl. CAN Conference (ICC 2002)*, Seiten 08/11–08/18, Las Vegas, NV, USA, Februar 2002. CAN in Automation (CiA).

[NK00] NACE, W. und P. KOOPMAN: *A Product Family Architecture Approach to Graceful Degradation*. In: *Proceedings of the Intl. Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, 2000.

[NS01] NUSSER, GERD und RALF-DIETER SCHIMKAT: *Rapid Application Development of Middleware Components by Using XML*. In: *Proceedings of the 12th IEEE Intl. Workshop on Rapid Systems Prototyping (RSP 2001)*, Seiten 116–121, Monterey, CA, USA, Juni 2001. IEEE Computer Society Press.

[Obj00] OBJECT MANAGEMENT GROUP (OMG), http://www.omg.org/: *Common Request Broker Architecture (CORBA)*, August 2000. Revision 2.4.1.

[Obj01a] OBJECT MANAGEMENT GROUP (OMG): *Developing in OMG's Model-Driven Architecture*, November 2001.

[Obj01b] OBJECT MANAGEMENT GROUP (OMG), http://www.omg.org/oma/: *Object Management Architecture (OMA)*, April 2001.

[Obj02] OBJECT MANAGEMENT GROUP (OMG), http://www.omg.org/technology/-documents/formal/mof.htm: *Meta Object Facility (MOF) Specification*, April 2002.

[OPC03a] OPC FOUNDATION, http://www.opcfoundation.org/: *OLE for Process Control (OPC)*, Mai 2003.

[OPC03b] OPC FOUNDATION, http://www.opcfoundation.org/: *XML-OPC*, 2003.

[Ope04] OPEN DEVICENET VENDOR ASSOCIATION (ODVA), http://www.odva.org/: *DeviceNet*, Mai 2004.

[Par95] PARNAS, DAVID L.: *Fighting Complexity*. IEEE Engineering of Complex Computer Systems Newsletter, 2(2), Oktober 1995.

[PCCB00] PARLAVANTZAS, NIKOS, GEOFF COULSON, MIKE CLARKE und GORDON BLAIR: *Towards a Reflective Component-based Middleware Architecture*. In: *On-Line Proceedings of the Workshop on Reflective and Metalevel Architectures at the European Conference on Object-Oriented Programming (ECOOP Workshop 2000)*, Sophia Antipolis and Cannes, France, Juni 2000.

[PD93] PRIETO-DÍAZ, RUBÉN: *Status Report: Software Reusability*. IEEE Software, 10(3):61–66, Mai 1993.

[PDN86] PRIETO-DÍAZ, RUBÉN und JAMES NEIGHBORS: *Module Interconnection Languages*. Journal of Systems and Software, 6(4):307–334, November 1986.

[Pos82] POSTEL, JONATHAN B.: *Simple Mail Transfer Protocol. Internet Standard RFC 821*, August 1982.

[PR85] POSTEL, JONATHAN B. und JOYCE K. REYNOLDS: *File Transfer Protocol. Internet Standard RFC 959*, Oktober 1985.

[Pre95] PREE, WOLFGANG: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[Psi01] PSINAPTIC INC., http://www.psinaptic.com/oem/jmatos.html: *JMatos™ Jini Network Technology for Embedded Processors*, 2001.

[Pur94] PURTILO, JAMES M.: *The POLYLITH Software Bus*. ACM Transactions on Programming Languages and Systems, 16(1):151–174, Januar 1994.

[QJHF97] QIONG, W., C. JICHUAN, M. HONG und Y. FUQING: *JBCDL: an object-oriented component description language*. In: *Proceedings of the 24th Conference on Technology of Object-Oriented Languages and Systems*, Seiten 198–205. IEEE Computer Society Press, September 1997.

[Red97] REDMOND, FRANK E.: *DCOM: Microsoft Distributed Component Object Model*. IDG Books Worldwide, Inc., 1997.

[Sam97] SAMETINGER, JOHANNES: *Software Engineering With Reusable Components*. Springer-Verlag Berlin Heidelberg, 1997.

[Sch03] SCHADER, MARTIN: *Middleware Interoperability of Enterprise Applications (MIEA 03), Call for Paper*, September 2003.

[Ses97] SESSIONS, ROGER: *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley and Sons, Inc., 1997.

[Ses98] SESSIONS, ROGER: *Component-Oriented Middleware for Commerce Systems*. IEEE Software, 15(5):42–43, September/Oktober 1998. Sidebar in The Current State of CBSE by Alan W. Brown and Kurt C. Wallnau.

[Sha03] SHANNON, B.: *Java™ 2 Platform Enterprise Edition Specification*. Sun Microsoft Corporation, http://java.sun.com/j2ee/, September 2003.

[Sie03] SIEMENS AG, http://www.sicomp.de/: *SICOMP*, April 2003.

[Sie04] SIEMENS AG, http://www.ad.siemens.de/sicomp/ftp/jfpc_d.pdf: *SICOMP RTVM - Java Real Time Virtual Machine*, 2004.

[Smi82] SMITH, BRIAN C.: *Reflection and Semantics in a Procedural Language*. Doktorarbeit, Massachusetts Institute of Technology. Laboratory for Computer Science, Januar 1982.

[Smi84] SMITH, BRIAN C.: *Reflection and semantics in LISP*. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Seiten 23–35, Salt Lake City, Utah, United States, 1984.

[SNB00] SCHIMKAT, RALF-DIETER, GERD NUSSER und DIETER BÜHLER: *Scalability and Interoperability in Service-Centric Architectures for the Web*. In: *Proceedings of the 11th Intl. Workshop on Network Based Information Systems (NBIS '00)*, Seiten 51–57, Greenwhich, London, Great Britain, September 2000. IEEE Computer Society Press.

[Sou03] SOURCEFORGE, http://trmi.sourceforge.net/: *Transparent RMI (TRMI)*, Juni 2003.

[Sta84] STANDISH, T. A.: *An essay on software reuse*. IEEE Transactions Software Engineering, 5(10):494–497, September 1984.

[Ste90] STEVENS, W. RICHARD: *UNIX network programming*. Prentice-Hall, Inc., 1990.

[Ste98] STEPHENSON, ALAN: *Supporting Reliable, Systematic Reuse in Embedded Systems*. University of York. Department of Computer Science, Juni 1998.

[Sun98] SUN MICROSOFT CORPORATION, http://java.sun.com/javaone/: *Java Native Interface Technology Programming*, 1998.

[Sun99a] SUN MICROSOFT CORPORATION, http://wwws.sun.com/software/jini/-whitepapers/architecture.html: *Jini Architecture Specification, Revision 1.0*, Januar 1999.

[Sun99b] SUN MICROSOFT CORPORATION, http://wwws.sun.com/software/jini/-specs/jini101specs/devicearch-spec.html: *Jini Device Architecture Specification, Revision 1.0*, Januar 1999.

[Sun99c] SUN MICROSOFT CORPORATION, http://wwws.sun.com/software/jini/-specs/jini101specs/boot-spec.html: *Jini Discovery and Join Specification, Revision 1.0*, Januar 1999.

[Sun99d] SUN MICROSOFT CORPORATION, http://wwws.sun.com/software/jini/-specs/jini10specs/lookup-spec.html: *Jini Lookup Service Specification, Revision 1.0*, Januar 1999.

[Sun02a] SUN MICROSOFT CORPORATION, http://www.javasoft.com/j2me: *Java 2 Micro Edition (J2ME)*, August 2002.

[Sun02b] SUN MICROSOFT CORPORATION, http://java.sun.com/webservices/jwsdp/: *Java Web Services Developer Pack (Java WSDP)*, 2002.

[Sun02c] SUN MICROSOFT CORPORATION, http://java.sun.com/j2se/1.4.1/: *Java™2 SDK, Standard Edition (J2SE)*, 2002.

[Sun03a] SUN MICROSOFT CORPORATION, http://java.sun.com/products/ejb/: *Enterprise Java Beans*, August 2003.

[Sun03b] SUN MICROSOFT CORPORATION, http://java.sun.com/products/rmiop/: *J2ME RMI Optional Package (RMIOP)*, Mai 2003.

[Sun03c] SUN MICROSOFT CORPORATION, http://wwws.sun.com/software/jini/: *Jini Network Technology*, Mai 2003.

[Sun04] SUN MICROSOFT CORPORATION, http://java.sun.com/products/jdk/rmi/: *Java Remote Method Invocation (Java RMI)*, Juni 2004.

[Tai93] TAIVALSAARI, ANTERO: *A Critical View of Inheritance and Reusability in Object-oriented Programming*. Doktorarbeit, University of Jyväskylä, Jyväskylä, Finland, 1993.

[Tan96] TANENBAUM, ANDREW S.: *Computer Networks*. Prentice Hall Ptr, 1996.

[Tho00] THOMASON, STUART: *What is a Component?* IEEE Computer, 33(11):55, November 2000. Sidebar in Component-Based Systems: A Classification of Issues by Pearl Brereton and David Budgen.

[Tic79] TICHY, WALTER F.: *Software Development Control Based on Module Interconnection*. In: *Proceedings of the 4th Intl. Conference on Software Engineering*, Seiten 29–41, Munich, Germany, September 1979. IEEE Computer Society Press.

[TKH99] THIRUNARAYAN, KRISHNAPRASAD, GÜNTER KNIESEL und HARIPRIYAN HAMPAPURAM: *Simulating multiple inheritance and generics in Java*. Computer Languages, 25(4):189–210, 1999.

[Tog04] TOGETHERSOFT, http://www.togethersoft.com: *Together. The Model–Build–-Deploy Platform™*, 2004.

[Tra90] TRACZ, WILL: *Implementation Working Group Summary*. In: BALDO, JAMES (Herausgeber): *Reuse in Practice Workshop Summary*, Seiten 10–19, Alexandria, VA, April 1990. IDA Document D-754, Institute for Defense Analyses.

[Ude94] UDELL, JON: *Componentware*. BYTE Magazine, 19(5):46–56, Mai 1994.

[vDKV00] DEURSEN, ARIE VAN, PAUL KLINT und JOOST VISSER: *Domain-Specific Languages: An Annotated Bibliography*. SIGPLAN Notices, 35(6):26–36, 2000.

[Vec03] VECTOR INFORMATIK GMBH, http://www.vector-informatik.de/: *CAN Interface Hardware CANCardX*, Januar 2003.

[Ven99] VENNERS, BILL: *The Jini vision*. http://www.javaworld.com/jw-08-1999/jw-08-jiniology.html, August 1999.

[Völ03] VÖLTER, MARKUS: *Metamodellbasierte Codegenerierung in Java*. http://www.-voelter.de/data/articles/MetaModelBasedCodeGen.pdf, August 2003.

[Wal99] WALDO, JIM: *The Jini architecture for network-centric computing*. Communications of the ACM, 42(7):76–82, 1999.

[Wat98] WATANABE, YOSHINORI: *JCom (Java-COM Bridge) Homepage*. http://-www11.u-page.so-net.ne.jp/ga2/no-ji/jcom/, 1998.

[Wil01a] WILSON, JOHN: *MinML a minimal XML parser*. http://www.wilson.co.uk/-xml/minml.htm, November 2001.

[Wil01b] WILSON, JOHN: *MinML-RPC a minimal XML-RPC implementation*. http://-www.wilson.co.uk/xml/minmlrpc.htm, Oktober 2001.

[Wor98a] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/1998/-REC-CSS2-19980512/: *Cascading Style Sheets, level 2 (CSS2)*, Mai 1998.

[Wor98b] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/REC-xml/: *Extensible Markup Language (XML) 1.0*, Februar 1998.

[Wor99] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/xslt.html: *XSL Transformations (XSLT)*, November 1999.

[Wor00] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/soap/: *Simple Object Access Protocol (SOAP)*, Mai 2000.

[Wor01a] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/wsdl: *Web Services Description Language (WSDL)*, März 2001.

[Wor01b] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/2001/-WD-xinclude-20010516/: *XML Inclusions (XInclude)*, Mai 2001.

[Wor01c] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/2001/-REC-xmlschema-2-20010502/: *XML Schema Part 2: Datatypes*, Mai 2001.

[Wor02] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/2002/ws/: *Web Services*, November 2002.

[Wor03] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/TR/2003/-WD-xsl11-20031217/: *Extensible Stylesheet Language (XSL)*, Dezember 2003.

[WS02] WELCH, IAN und ROBERT J. STROUD: *Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code*. Journal of Computer Security, 10(4):399–432, Dezember 2002.

[Yel01] YELLIN, DANIEL M.: *Stuck in the Middle: Challenges and Trends in Optimizing Middleware*. In: *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, Seiten 175–180, Snow Bird, Utah, United States, 2001. ACM Press.

Lebens- und Bildungsgang

1976 bis 1980	Grundschule Hummel-Schule Saulgau.
September 1980 bis Mai 1989	Allgemeine Hochschulreife am Störck-Gymnasium Saulgau.
1989 bis 1990	Wehrdienst.
Oktober 1990 bis Februar 1998	Studium der Informatik mit Nebenfach Mathematik an der Eberhard-Karls-Universität Tübingen.
Juli 1998 bis Mai 2003	Wissenschaftlicher Angestellter an der FH Reutlingen in Kooperation mit der Universität Tübingen, Arbeitsbereich Symbolisches Rechnen, im Rahmen des Förderprojektes Virtuelle Universität, gefördert vom Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg.
Juni 2003 bis März 2005	Softwareentwickler bei der Fa. Harman/Becker Automotive Systems GmbH, Filderstadt, Deutschland.
Seit April 2005	Teamleiter bei der Fa. Harman/Becker Automotive Systems GmbH, Filderstadt, Deutschland.