EINE PLATTFORM ZUR VISUALISIERUNG VON EINANDER ABHÄNGIGER PRODUKTDATEN IM PRODUKTLEBENSLAUF

von

Dipl.-Ing. Martin Richter aus Karlsruhe

Fakultät für Maschinenbau
Universität Karlsruhe (TH)
zur Erlangung des akademischen Grades

eines Doktors der Ingenieurwissenschaften

vorgelegte Dissertation

Hauptreferent: o. Prof. em. Dr.-Ing. Prof.E.h. Dr.h.c. H. Grabowski

Korreferent: o. Prof. Dr.-Ing. C. Weber

Tag der mündlichen Prüfung: 4. Februar 2005

20. November 2005

ABSTRACT

Produkte werden immer komplexer, Käufer immer anspruchsvoller, Güter immer billiger. In diesem Spannungsdreieck haben die Anbieter wesentliche Herausforderungen zu meistern: Zum einen wird der Lebenslauf von Produkten kontinuierlich kürzer, womit auch die Zeit, mit einem neuen Produkt Geld zu verdienen, immer kürzer wird. Zweitens werden mehr als 80 Prozent der gesamten Produktkosten bereits in der Entwicklung festgelegt. Bei der Entwicklung neuer Produkte muss daher nicht nur die gewünschte Produktfunktion und -qualität sichergestellt werden, erfolgskritisch ist auch die Berücksichtigung von Kostentreibern nachgelagerter Prozesse wie Beschaffung, Logistik, Produktion oder Wartung. Ziel des Product Lifecycle Managements (PLM) ist es Produkte entlang ihres Lebenslaufs informationstechnisch zu erfassen. Das „digitale Modell" begleitet das Produkt sozusagen entlang seines Lebenslaufs, wobei die Verknüpfungen zwischen der eigentlichen Gestaltungsphase und sehr frühen Konstruktionsphasen, sowie den der Konstruktion nachgelagerte Phasen, etwa der Fertigung, von besonderem Interesse sind. Der Konstrukteur braucht ein leistungsfähiges Werkzeug, Abhängigkeiten dieser Art zu visualisieren, um sie zu begreifen. Die vorliegende Arbeit stellt einen Ansatz vor, Informationen entlang des Produktlebenslaufs mit der Produktgeometrie in Bezug zu setzen, und die Zusammenhänge zu visualisieren. Sie beschreibt dabei einen Ansatz wie die Informationen zusammengeführt werden können ohne auf ein allumfassendes monolithisches Produktmodell angewiesen zu sein und wie die Informationen in einem zweiten Schritt anwendungsbezogen, immersiv visualisiert werden.

Viele der Ergebnisse dieser Arbeit sind in den zwei Jahren meiner Tätigkeit als Projektleiter und Bearbeiter des internationalen Verbundprojektes (ITEA[1] + BmBF) „3D Workbench" entstanden und haben einen ausgeprägten Bezug zur industriellen Praxis. Nicht nur für mich sondern auch für das Institut (RPK), das traditionell seine Wurzeln in der Konstruktionsmethodik hat war dies eine Herausforderung, da die Kompetenz auf diesem Gebiet erst erarbeitet werden musste. Der Ausführliche „Stand der Technik" dieser Arbeit soll weiteren Arbeiten als Grundlage und Ausgangspunkt dienen um der neuen Präsentationstechnik „VR" den Einzug in die Methodik der Produktentwicklung zu ebnen. Die Nachhaltigkeit der Ergebnisse des Projektes wurden in einem „Spin-Off" aus dem Institut heraus, der **incentrix GmbH**, gesichert, der die, in dieser Arbeit vorgestellte, komponentenbasierte Entwicklungsplattform weiterentwickelt. Diese Plattform soll in Zukunft ermöglichen Software innerhalb von Industrieprojekten schnell und effizient zu entwickeln und hat daher auch strategische Bedeutung. „Reusability" von Softwarekomponenten

[1] ITEA Information Technology for European Advancement (EUREKA Cluster Programme)

werden das in der Zukunft ermöglichen, und viele weitere Komponenten die nahtlos in dieses Framework passen werden die Anwendungsfunktionalität komplettieren.

Karlsruhe, den 10.11.2004

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

DANKSAGUNGEN/WIDMUNGEN

Die vorliegende Arbeit entstand neben meiner fünfjährigen Tätigkeit als wissenschaftlicher Mitarbeiter und Projektleiter am Institut für Rechneranwendung in Planung und Konstruktion der Universität Fridericiana Karlsruhe (TH).

Herrn o. Prof. em. Dr.-Ing. Prof. e.h. Dr. h.c. H. Grabowski, dem ehemaligen Leiter des Instituts gilt mein besonderer Dank für die wissenschaftliche Betreuung, die konstruktive Kritik und das mir entgegengebrachte Vertrauen während meiner Institutszeit.

Bei Frau Prof. Dr. Dr.-Ing. J. Ovtcharova, der jetzigen Leiterin des Instituts, möchte ich mich für die Unterstützung bei der Gründung des Spin-Off aus der Universität heraus bedanken.

Bei meinen beiden Mitgesellschaftern Oliver Klaar und Jochen Kögel möchte ich mich dafür bedanken, dass Sie mir oft „den Rücken" freigehalten haben um diese Arbeit fertig zu stellen.

Allen Kollegen, Mitarbeitern und Ehemaligen des Instituts möchte ich für die stets gute und angenehme Zusammenarbeit danken.

Ebenfalls danken möchte ich meinen langjährigen studentischen Mitarbeitern Salah Edine Benamira, Hristina Borissova, Marco Ochs und Micaela Wünsche für deren überdurchschnittliches Engagement, die ausgezeichnete Zusammenarbeit und das freundschaftliche Verhältnis.

Mein ganz besonderer Dank gilt auch meiner Lebenspartnerin F. Schmidt, die mich in dieser Zeit hat „ertragen" müssen, für Ihre Unterstützung bei der Durchsicht des Manuskriptes.

Danken möchte ich auch meinen Eltern die mich von Beginn meines Studiums an unterstützt haben und mir diese Arbeit erst möglich machten.

Karlsruhe, im August 2004

Martin Richter

ABKÜRZUNGSVERZEICHNIS

API	**A**pplication **P**rogramming **I**nterface
B-REP	**B**oundary **Rep**resentation
BMBF	**B**undes**m**inisterium für **B**ildung und **F**orschung
BNF	**B**ackus-**N**aur-**F**orm
CAD	**C**omputer **A**ided **D**esign
CAE	**C**omputer **A**ided **E**ngineering
CAM	**C**omputer **A**ided **M**anufacturing
CAVE	**C**ave **A**utomatic **V**irtual **E**nvironment
CAVEE	**CAVE E**ngineering **E**nvironment
CFD	**C**omputational **F**luid **D**ynamics, numerische Strömungssimulation
CORBA	**C**ommon **O**bject **R**equester **B**rooker **A**rchitecture
DOM	**D**ocument **O**bject **M**odel
DTD	**D**ata **T**ag **D**efinition
DMU	**D**igital **M**ock **U**p, digitaler Zusammenbau
FEM	**F**inite **E**lemente **M**ethode
GML	**G**eneral **M**ark up **L**anguage
HTML	**H**ypertext **M**ark up **L**anguage
IDL	**I**nterface **D**efinition **L**anguage
IGES	**I**nitial **G**raphics **E**xchange **S**pecification
ISO	**I**nternational **S**tandardization **O**rganisation
ITEA	**I**nformation **T**echnology for **E**uropean **A**dvancement, EUREKA Cluster
KMU	**K**lein- und **M**ittelständische **U**nternehmen
MDA	**M**odel **D**riven **A**rchitecture
MFGDTF	**M**anu**f**acturin**g** **D**omain **T**ask **F**orce

MOM	**M**essage **O**riented **M**iddleware
NURBS	**N**on-**U**niform **R**ational **B**-**S**plines
OEM	**O**riginal **E**quipment **M**anufacturer, Originalhersteller
OMA	**O**bject **M**anagement **A**rchitecture
OMG	**O**bject **M**anagement **G**roup
PDM	**P**roduct **D**ata **M**anagement
PID	**P**ersistent **ID**
PLM	**P**roduct **L**ifecycle **M**anagement
STEP	**St**andard for the **E**xchange of **P**roduct Model Data
RPC	**R**emote **P**rocedure **C**all
SFB	**S**onder**f**orschungs**b**ereich
SGML	**S**tandard **G**eneralized **M**ark up **L**anguage
UMEO	**U**nified **M**odel of **E**ngineering **O**bjects
ULEO	**U**niversal **L**inking of **E**ngineering **O**bjects
VDA-FS	**V**erein **d**eutscher **A**utomobilbauer **F**lächen**s**chnittstelle
VE	**V**irtual **E**ngineering
VR	**V**irtuelle **R**ealität
WSDL	**W**eb service **D**efinition **L**anguage
XML	e**x**tended **M**ark up **L**anguage

GLOSSAR

B-Rep Topologisch Geometrisches Strukturmodell (eines Volumenkörpers) [2] [3] [4] [5]

CAVE Umgebung für immersive Virtual Reality Simulationen. Entwickelt von der Universität Illinois und zuerst vorgestellt auf der SIGGRAPH 1992 (siehe 2.4.1.4)

Feature Ein Feature ist ein informationstechnisches Element, das Bereiche von besonderem (technischen) Interesse (nicht ausschließlich Geometrie) eines Produktes darstellt, wobei das Feature eine spezifische Sichtweise auf die Produktbeschreibung repräsentiert, die mit bestimmten Eigenschaftsklassen und bestimmten Phasen des Produktlebenslaufs im Zusammenhang steht. [6] [7]

IGES Neutrales Austauschformat für 2D- und 3D-Produktdaten

Immersion Der Begriff der Immersion beschreibt im Zusammenhang der virtuellen Realität das „Eintauchen" in eine künstliche Welt. Je höher der Grad der Immersion desto ausgeprägter ist die Erfahrung eines Anwenders, sich in einer virtuellen Welt zu befinden, und interaktiv mit Objekten aus dieser künstlichen Welt zu interagieren.

STEP Unter Federführung der ISO entwickelte Schnittstelle, die den Austausch von geometrischen, technologischen und administrativen Produktdaten standardisieren soll. Zurzeit werden die Bereiche Mechanik, Elektronik, Schiffsbau und Bauwesen durch diese Norm unterstützt.

UML Unified Modelling Language [8]

Kapitel 1

EINLEITUNG

Produkte, die heutzutage den Markt erobern, sind durch eine hohe Komplexität, hohe Erwartungen an die Qualität und immer kürzere Produktentwicklungszeiten charakterisiert. Firmen, die diesen Forderungen nicht nachhaltig Rechnung tragen, werden in Zukunft durch wachsenden Druck der Konkurrenz Wettbewerbsnachteile auf dem Markt haben. Neben der Rationalisierung des Produktentstehungsprozesses ist in allen Bereichen der Produktentwicklung der neue Trend zur Produktindividualisierung zu spüren, der zu einer explosionsartigen Zunahme der Variantenvielfalt führt. Der Konstrukteur sieht sich einer stetig wachsenden Flut von Informationen, bei gleichzeitig schrumpfenden Zeiten für die Entwicklungsprozesse, gegenüber. Es ist einzusehen, dass die leistungsfähige Visualisierung von Produktdaten ein entscheidender Faktor ist, der die Wettbewerbsfähigkeit von Unternehmen z.B. des Automobil-, Luft und Raumfahrt, Maschinen- und Anlagenbaus bestimmt. Da informationsverdichtende Visualisierungen aller Daten entlang des Produktlebenslaufs, die Entscheidungsgrundlage für die Entscheidungsträger eines Unternehmens und die Beteiligten eines Projektes sind.

1.1 Situation in der industriellen Praxis

Seit Mitte der 80er Jahre expandiert der CAD-Markt unaufhaltsam, da die Integration von CAD-Systemen in den ganzheitlichen Produktentwicklungsprozess und die damit einhergehende Datenverwaltung immer wichtiger werden. 3D-CAD-Systeme sind wesentliche Basis für die Realisierung der virtuellen Produktentwicklung in den Unternehmen. 3D-Konstruktion ist in der modernen Produktentwicklung eine zwingende Notwendigkeit geworden: Anspruchsvollere Konstruktionsaufgaben müssen immer schneller, genauer, leichter und effektiver bewältigt werden [9]. Gleichzeitig zeichnet sich jedoch ab, dass einige wenige Systemanbieter durch den Aufbau von proprietären Grenzen, versuchen Marktanteile zu sichern, was auf Kosten der Interoperabilität geschieht. Abbildung 1 zeigt die gegenwärtige Verteilung der CAD-Systeme auf dem Markt im Bereich Automotive. Produktentwicklungssysteme sind Insellösungen die nur lose durch die Möglichkeit des Datenaustausches miteinander verbunden sind. Zum Beispiel im Fall der CAD-Systeme wird man nicht müde den Kunden zu versprechen, dass ein Datenaustausch via Neutralfile Formaten (STEP[2], IGES[3], VDA-FS[4]) verlustfrei möglich ist, verschweigt jedoch elegant, dass dieser

[2] STEP

[3] IGES

[4] VDA-FS Verein deutscher Automobilhersteller - Flächenschnittstelle

Austausch sich nur auf reine Geometrie beschränkt. Weder Feature-Informationen noch andere Meta-Informationen entlang des Produktenstehungsprozesses lassen sich effektiv austauschen oder gar integriert visualisieren. Um dieser Tendenz entgegenzuwirken, ist es gängige Praxis, dass z.B. in der Automobilindustrie Zulieferfirmen genötigt werden, dieselben Produktentwicklungssysteme mit einer identischen Produktversion zur Verfügung zu haben, um den Datenaustausch zwischen Zulieferer und OEM zu gewährleisten. Resultat dieser Vorgehensweise ist, dass Zulieferfirmen alle gängigen Systeme am Markt in der eigenen Firma zur Verfügung haben müssen, nur um die Option auf einen Auftrag zu haben. D.h. zu den hohen Anschaffungskosten kommt eine relativ niedrige Systemauslastung. Oftmals reichen die Margen aus Produktionsaufträgen nicht aus, um diese Infrastruktur mit den dazugehörigen Mitarbeitern weiter zu unterstützen. Die Konsequenz aus diesem Zustand ist, dass Zulieferfirmen oft Aufträge nicht annehmen können, weil ihnen die zugehörige Plattform nicht zu Verfügung steht. Abbildung 2 zeigt dies exemplarisch für den Bereich der CAD-Systeme

Doch nicht nur für Zulieferfirmen stellt diese Tendenz in der CAD-Landschaft, einschließlich der damit verbundenen Produktentwicklungswerkzeuge, eine ernstzunehmende Bedrohung dar, klein- und mittelständische Softwareunternehmen sind ebenfalls bedroht. Trotz innovativer Ideen stellen die hohen Anschaffungskosten der CAD-Systeme (Entwickler-Lizenzkosten) eine hohe Markteintrittsbarriere dar. Das Entwickeln von eigenständigen Methoden und Werkzeugen ist nicht mehr möglich, da alles auf das jeweilige CAD-System zugeschnitten werden muss.

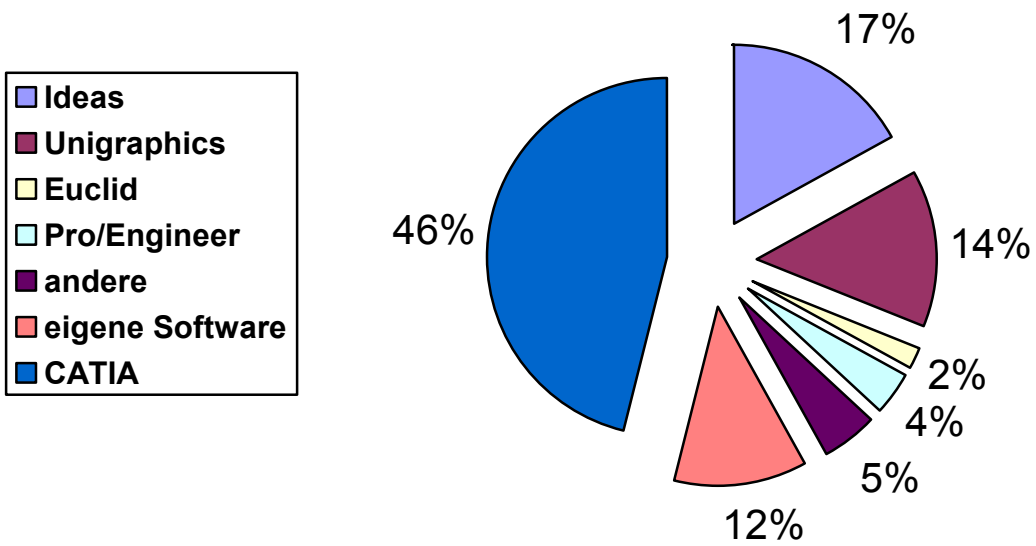


Abbildung 1: Verteilung der CAD-Systeme am Markt[5]

[5] Quelle: IBM (gültig für Bereich Automotive, Stand 2004)

Abbildung 1 gibt eine Übersicht über die Verteilung der CAD-Systeme (Stand 2004). Zwei Hersteller teilen sich mehr als 50 % des Marktvolumens im Bereich Automotive.

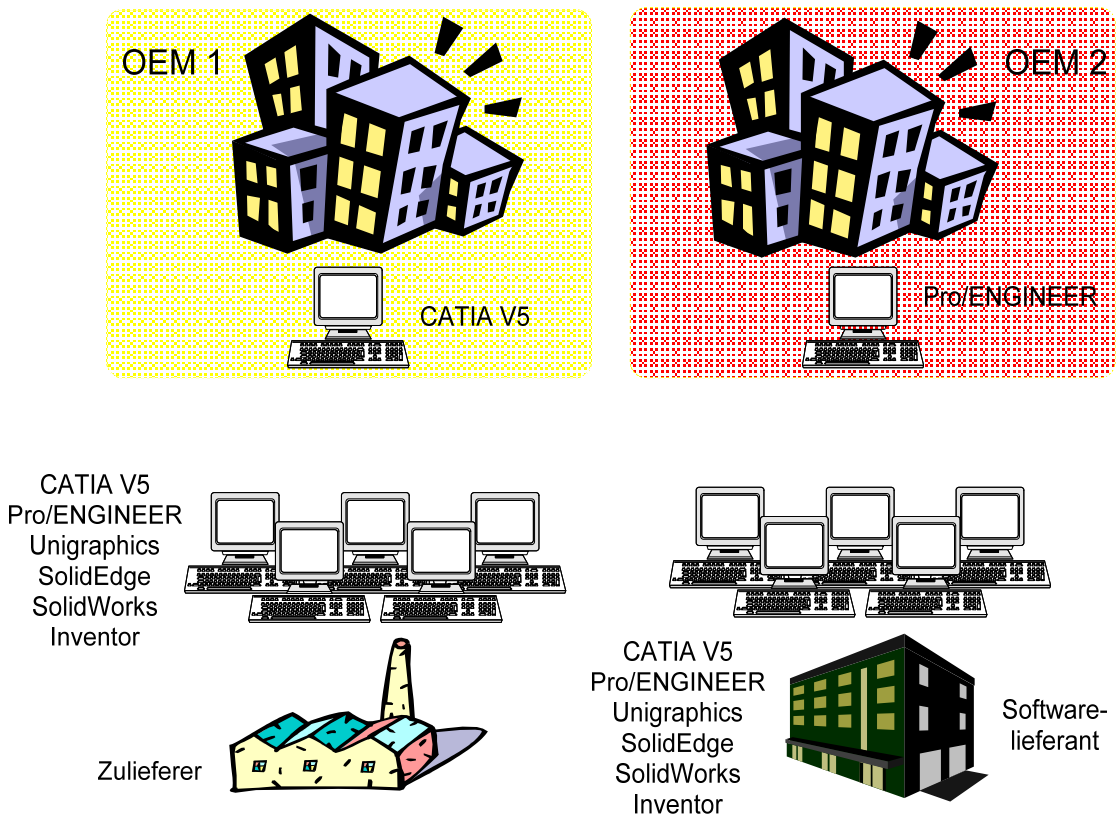


Abbildung 2: OEM-Zulieferer Problematik

1.2 Motivation und Nutzenpotenzial

Die Integration der Entwicklungswerkzeuge entlang des Produktentwicklungsprozesses in heterogenen Systemlandschaften, das konsequente Verzichten auf physische Prototypen während der Produktentwicklung zugunsten eines integrativen serviceorientierten Produktdatenmodells und das Bereitstellen von leistungsfähigen Werkzeugen zur Datenexploration bieten großes Potenzial, Produktentwicklungszeiten zu verkürzen [10]. Erschwerend wirkt sich hierbei jedoch die Tatsache aus, dass Produkte heutzutage neben den mechanischen Komponenten in der Regel Komponenten aus anderen Domänen enthalten, wie z.B. elektrische oder Software-Komponenten. D. h. bei der Entwicklung von komplexen Produkten werden in der Regel interdisziplinäre Design-Teams kooperativ tätig sein, wobei verschiedene domänenabhängige Taxonomien[6] zum Einsatz kommen.

[6] Verwendung von unterschiedlichen Begrifflichkeiten und Darstellungen je nach Anwendungsfall und fachlichem Hintergrund

Die intelligente und problembezogene Visualisierung, d.h. die die Darstellung eines abstrakten Sachverhaltes mit optischen Mitteln, von Produktdaten besitzt demzufolge einen hohen Stellenwert im kooperativen Produktenstehungsprozess.

Doch es gibt einige ernst zu nehmende Hürden auf dem Weg zu einer integrierten Visualisierung von Produktdaten:

- Die Produktdaten sind über das ganze Unternehmen in den unterschiedlichen Produktentwicklungswerkzeugen mit unterschiedlichen Formaten verstreut.

- Eine explosionsartige Zunahme der Daten und der Datenkomplexität gekoppelt mit der Notwendigkeit, diese Daten in immer kürzerer Zeit zu begreifen und weiterzuverarbeiten.

- Die Produktdaten müssen für Anwender mit unterschiedlichem Hintergrund, unterschiedlichem technischen Sachwissen und unterschiedlichen Bedürfnissen verständlich sein.

- Die Daten müssen sicher, leicht verfügbar und verlässlich sein.

- Die Kosten bei der Bereitstellung der Daten müssen sich in einem vernünftigen Rahmen bewegen.

- Knappere Budgets für Informationsvisualisierung an sich und die Forderung nach Portierbarkeit führen zu dem Trend, Standards einzusetzen.

Damit die verfügbaren Produktdaten, besonders aufwändige 3D-CAD-Daten, effektiv genutzt werden können, müssen mehr Mitarbeiter innerhalb des Unternehmens diese Daten zur Entscheidungsfindung nutzen. Dies trägt zur Innovation bei, verbessert die Kommunikation und reduziert die aufgrund von Missverständnissen entstandenen Fehler. Die meisten Personen, denen die 3D-Daten von Nutzen wären, haben jedoch keinen Zugriff auf diese Daten. Sie können die Daten nicht abrufen, da sie nicht über die notwendige Software verfügen.

Das Diagramm in Abbildung 3 veranschaulicht den Ernst dieses Problems. Aus dem Diagramm wird ersichtlich, dass nur ein sehr kleiner Prozentsatz (4 %) der Personen mit Zugriff auf CAD-Systeme in der Lage sind, 3D-Designmodelle von Produkten abzurufen, zu verändern und zu analysieren. Die Mehrheit der Mitarbeiter in Fertigungsunternehmen hat keinerlei Zugriff auf diese Daten.

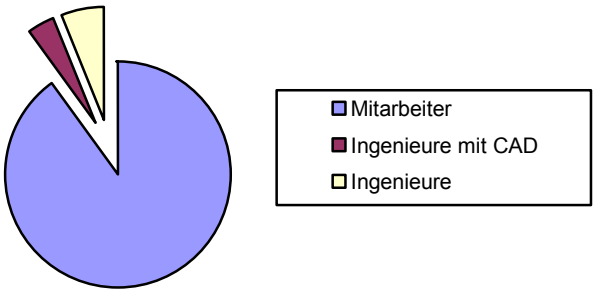


Abbildung 3: Verfügbarkeit von CAD-Daten innerhalb eines Unternehmens (Quelle: US Bureau of Labor Statistics, 1997)

1.3 Zielsetzung der Arbeit

Zielsetzung dieser Arbeit ist es, Mitarbeitern eines Unternehmens Informationen und deren Zusammenhänge aus Produktentwicklungssystemen entlang des Produktlebenslaufs einfach und problembezogen zugänglich zu machen. Um diese Zielsetzung zu erreichen, wird ein Konzept entwickelt wie die Informationen aus den Produktentwicklungssystemen:

- **standardisiert in einem Framework zur Verfügung gestellt werden**.

  Die reale Situation in der industriellen Praxis heute ist dadurch gekennzeichnet, dass die produktbeschreibenden Datenbestände in einer Vielzahl unterschiedlicher IT-Systeme (CAx-Systeme) erzeugt und gespeichert werden. Diese IT-Systeme sind in der Regel nicht oder nur unzureichend miteinander verbunden. Im Rahmen dieser Arbeit wird ein Konzept erarbeitet, wie ein serviceorientiertes Framework aufgebaut sein muss, damit die Informationen aus den einzelnen IT-Systemen über standardisierte Methodenschnittstellen zur Verfügung gestellt werden können.

- **über Relationen miteinander in Beziehung gesetzt werden**.

  Informationsverdichtende Visualisierungen für bestimmte Teilbereiche der Produktentwicklung wie z.B. Bauteilbelastungen bei der FEM-Analyse sind seit Jahren Stand der Technik. Domänenübergreifende Visualisierungen oder Visualisierungen von Produktdaten aus frühen Konstruktionsphasen wie z. B. Anforderungen sind bisher nicht verfügbar. Um die Beziehungen der Informationen aus den Produktentwicklungswerkzeugen abzubilden, wird in der Arbeit ein Ansatz vorgestellt, Beziehungen zwischen den Informationen, insbesondere Produktgestalt, herzustellen.

- **problembezogen, immersiv visualisiert werden**.

  Kooperative Produktentwicklung bedingt, dass Entwickler aus verschiedenen Domänen auf den produktbeschreibenden Datenbeständen gemeinsam arbeiten. Visualisierungen dieser Produktdaten müssen demnach die gewohnte Symbolik der Entwickler benutzen und anpassbar auf das jeweilige Problem sein. In dieser Arbeit wird ein flexibles Konzept zur Informationsvisualisierung mittels eines Mapping Prozesses von Informationen auf grafische Metaphern entwickelt, der es erlaubt, Informationen problembezogen zu visualisieren.

Für den Prozess der Visualisierung von Informationen zusammen mit der zugehörigen Bauteilgeometrie sind derzeit keine Ansätze zur Rechnerunterstützung verfügbar. Allgemeingültige Methoden zur Informationsvisualisierung werden im „Stand der Technik" unter dem Gesichtspunkt der Anwendbarkeit im Produktentwicklungsprozess untersucht.

Die Realisierung des Lösungskonzeptes erfolgt in Form der Implementierung eines Softwareprototyps. Anhand von zwei Beispielen aus dem industriellen Umfeld wird die Tragfähigkeit dieses Konzeptes verifiziert. Für die beiden Beispiele aus dem Bereich der Anforderungsmodellierung und der Gestaltfeatures wurde ein kompletter Satz an grafischen Metaphern entwickelt.

Kapitel 2

GRUNDLAGEN UND ANALYSE BESTEHENDER ANSÄTZE

Menschen gewinnen einen Eindruck über die sie umgebene Welt, indem sie alle Ihre Sinne einsetzen. Zwar sind die optischen Fähigkeiten eines Menschen am weitesten ausgeprägt, um aber ein Objekt in all seinen Eigenschaften zu erfassen, ist der Einsatz aller Sinne nötig. Viele Jahre schaffte der Modellbau bzw. der Prototypenbau die Grundlage für solche beurteilenden Prozesse in der Technik. Die konstruierten technischen Objekte waren in der Regel als physische Prototypen[7] verfügbar und konnten so visuell und haptisch beurteilt werden. Produkteigenschaften konnten sehr realitätsnah beurteilt werden. Diese Vorgehensweise hatte jedoch zwei entscheidende Nachteile. Zum einen war sie trotz der guten Integration von Rapid Prototyping Werkzeugen in den rechnergestützten Konstruktionsprozess relativ teuer und zeitaufwändig. Zum anderen ist es nicht möglich, die Verknüpfung von Informationen aus den frühen Konstruktionsphasen (Anforderungsdefinition, Funktionsmodellierung, Prinzipmodellierung) mit der Bauteilgestalt zu verknüpfen und dem Betrachter die Zusammenhänge zu verdeutlichen. Doch gerade die Kenntnis dieser Zusammenhänge, und deren Visualisierung, führt erst zur vollen Entfaltung der Vorteile eines rechnergestützten methodischen Konstruktionsprozesses.

2.1 Der Konstruktionsprozess

Abbildung 4 beschreibt die grundlegenden Schritte des Konstruktionsprozesses und deren Arbeitsergebnisse. Bei der Klärung der Aufgabenstellung wird eine Menge von Anforderungen an das Produkt aufgenommen. Erst in neuester Zeit sind Werkzeuge verfügbar, die Anforderungen in eine rechnerverarbeitbare Form überführen können [11]. Ausgehend von der Anforderungsliste wird die Gesamtfunktion des Produktes festgelegt und im nächsten Schritt, der Funktionsfindung, nach dem „Teile-und-Herrsche"-Prinzip in eine hierarchische Funktionsstruktur zerlegt. Die Prinziperarbeitung ordnet den Teilfunktionen der Funktionsstruktur physikalische Wirkprinzipien zu, die die Funktion erfüllen. Dieser Arbeitsschritt erweist sich in der Praxis als außerordentlich kompliziert, da das Zuordnen von physikalischen Wirkprinzipien in der Regel starke Rückwirkungen auf die Funktionsstruktur hat. So entsteht z.B. durch die Selektion des physikalischen Wirkprinzips „Reibung" zur Realisierung einer negativen Beschleunigung bei einer Bremse, unweigerlich ein neues Element „Wärme ableiten" in der Funktionsstruktur. Der ganze Konstruktionsprozess ist „in sich" ein iterativer Prozess, besonders jedoch beim Übergang von der Funktionsstruktur zur Wirkprinzipstruktur. Ausgehend von den physikalischen Wirkprinzipien wird

[7] Ausnahmen bildeten Großinvestitionsgüter wie z.B. Papiermaschinen oder auch Chemieanlagen

ein Grobentwurf der Bauteilgestalt angefertigt, der in nachfolgenden Arbeitsschritten detailliert wird. Die Anforderungsmodellierung zusammen mit der Funktions- und Prinzipmodellierung werden als frühe Konstruktionsphasen bezeichnet. [12] [13-16] [17] [18] [19] [20] [21, 22] [23] [24] [25]

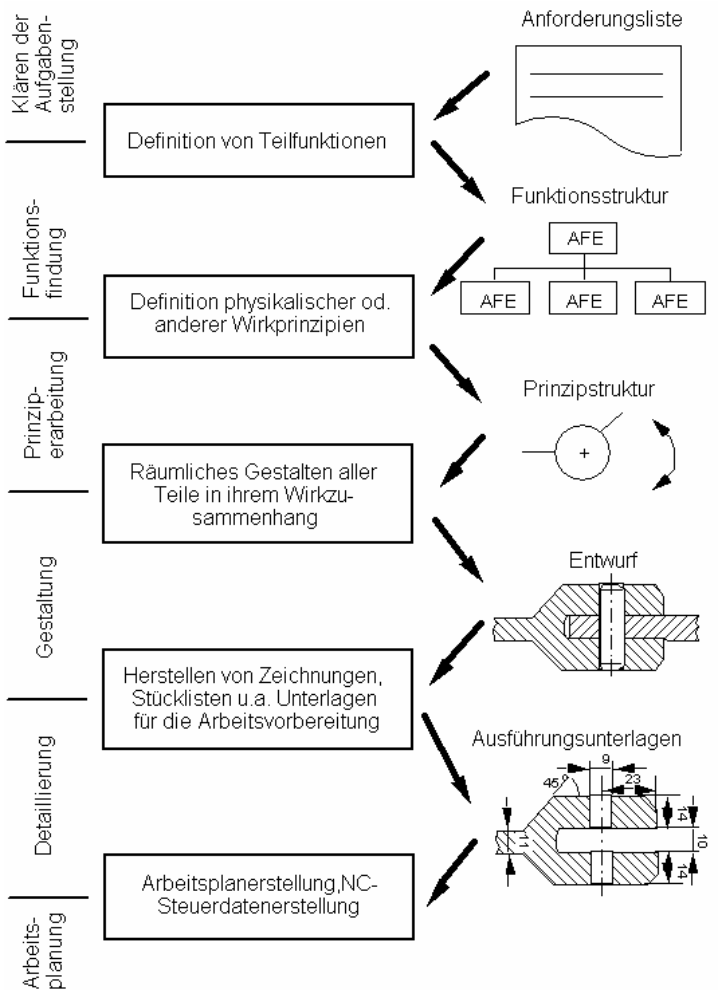


Abbildung 4: Konstruktionsphasen (Arbeitsergebnisse) (Quelle: [26])

2.2 Der Produktlebenslauf

Der Produktlebenslauf beschreibt den Weg eines Produktes von der Konstruktion bis zum Ausscheiden aus dem Markt (Recycling). Der Konstruktionsprozess ist dabei ein vitaler Teil des Produktlebenslaufs, da er bestimmend für viele Bereiche ist. Viele Informationen aus nachfolgenden Bereichen des Produktlebenslaufs werden aus den Konstruktionsdaten gewonnen (NC-Daten, Stücklisten, Betriebsmittel- und Methoden, …). Umgekehrt haben wiederum diese nachfolgenden Bereiche Einfluss auf die Konstruktion. Man spricht hier von den sog. Gerechtigkeiten wie z.B. Fertigungsgerecht, Prüfgerecht, Recyclinggerecht, usw.

Da ein virtuelles Produkt ein möglichst komplettes Abbild eines realen Produktes sein soll, beschreibt es also nicht nur die Gestalt (CAD-Daten), sondern umfasst auch weit reichende Informationen zur internen Struktur, zum Verhalten, zur Fertigung usw. Von der ersten Planung über die Konstruktion bis hin zur Null-Serie werden sämtliche Daten des Produkts durchgehend digital abgebildet und stehen damit über Standort- und Unternehmensgrenzen hinweg zur Verfügung.

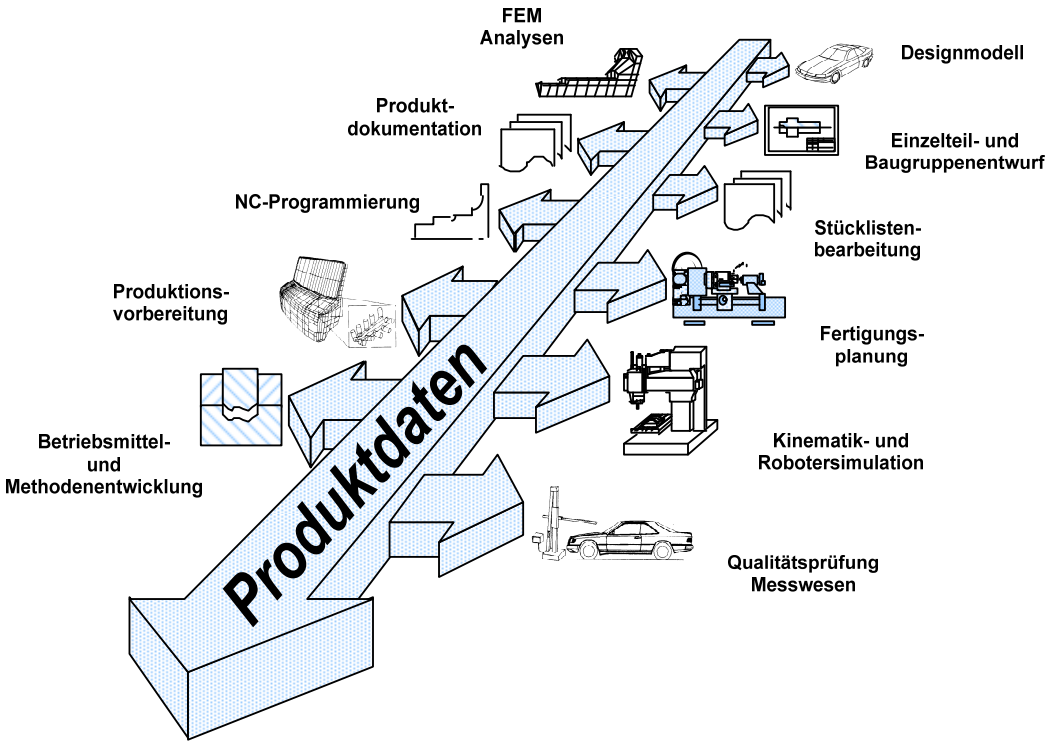


Abbildung 5: Der Produktlebenslauf (Quelle: [26])

2.3 Produktmodelle

Die Existenz jedes Produktes ist durch mehrere unterschiedliche Stufen gekennzeichnet – die so genannten Produktlebensphasen. Ein Bestreben in diesem Bereich ist die möglichst umfangreiche und tief greifende Rechnerunterstützung der Prozesse. [27] In den einzelnen Phasen des Produktlebenslaufs werden große Mengen an prozessspezifischen Informationen verarbeitet. Diese Informationen sind oft heterogen und stammen aus unterschiedlichen Quellen.

Die grundsätzliche Idee der Rechnerunterstützung ist die Verarbeitung dieser Informationen, wobei man unter Verarbeitung das Erfassen, Strukturieren, Bearbeiten und Visualisieren der Informationen versteht.

Für die Realisierung dieses Konzeptes wurde der Begriff des Produktmodells eingeführt. Als Produktmodell wird die Abbildung eines realen Produktes auf ein rechnerinternes Modell verstanden. [28]

Die existierenden Typen von Produktmodellen können folgendermaßen unterteilt werden [29]:

- geometrische Produktmodelle
- featureorientierte Produktmodelle
- strukturorientierte Produktmodelle
- wissensbasierte Produktmodelle
- gekoppelte Produktmodelle
- integrierte Produktmodelle.

Integrierte Produktmodelle werden als Ergebnis der Integration von geometrischen, featureorientierten, strukturorientierten und wissensbasierten Produktmodellen betrachtet [30]. Unter Integration versteht man mehr als eine einfache Vernetzung der einzelnen Modelle – das integrierte Modell stellt eine einheitliche Modellstruktur, bestehend aus Partialmodellen, dar (Abbildung 6).

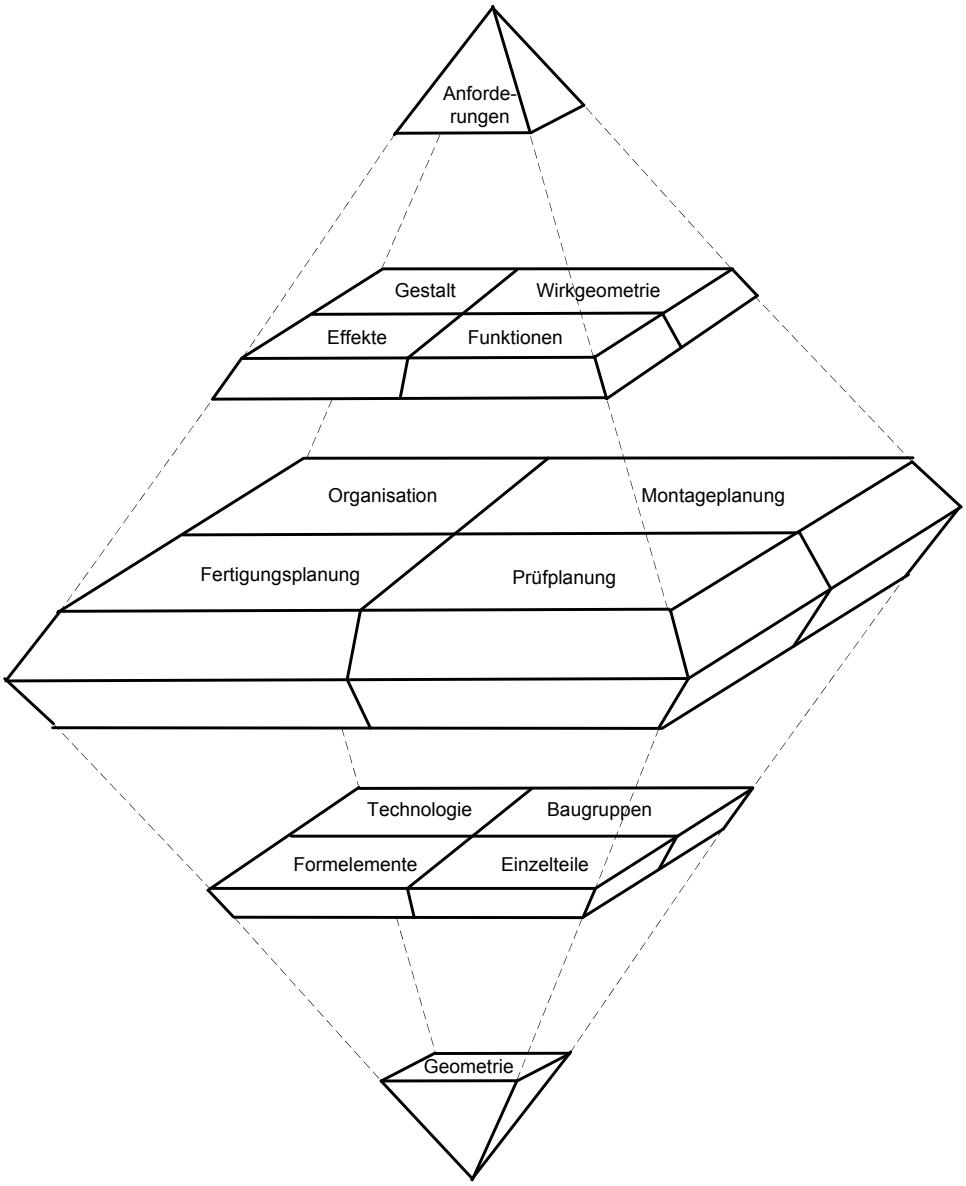


Abbildung 6: Partialmodelle eines integrierten Produktmodells
(Quelle:[31])

Bei diesem Verfahren können neben geometrisch-topologischen auch z.B. auch technologische und funktionelle Informationen beschrieben werden, wodurch semantisch höhere Modellierungsebenen unterstützt werden können [32].

Die Integration von Produktinformationen verläuft auf zwei Ebenen, der horizontalen und vertikalen Integration. Die horizontale Integration muss die Prozesse umfassen, die für eine der Produktlebensphasen kennzeichnend sind, z.B. Problem definieren, Lösung darstellen, Lösung bewerten u. a.; wegen der Heterogenität der Informationen in jeder Produktlebensphase ist eine spezifische Sicht des Produktmodells vorausgesetzt. Mehrere nicht einheitliche Abbildungen der Produktinformationen bezüglich einer der Produktlebensphasen und die schnell wachsende

Komplexität der Zusammenhänge führen zu einem Bedarf an Erweiterung und Optimierung der Strukturen von Produktmodellen. Die vertikale Integration muss alle Produktlebensphasen miteinander verknüpfen – von Produktplanung bis Recycling. Die Relationen zwischen Produktinformationen, die auf beiden Ebenen verteilt sind, bauen eine Vernetzung in der Produktstruktur auf. Dabei werden einmal die Zusammenhänge zwischen den Prozessen einer Phase und das andere Mal Zusammenhänge zwischen den einzelnen Phasen abgebildet. Beispielsweise besteht ein zunehmender Bedarf an möglichst frühzeitiger Berücksichtigung von Produktanforderungen aus den späteren Produktlebensphasen. Auf diese Weise entstehen Zusammenhänge zwischen Produktanforderungen, die zu den Frühkonstruktionsphasen gehören, und Produktinformationen aus den späteren Phasen.

Die graphische Darstellung solcher strukturellen Zusammenhänge wäre eine wichtige Unterstützung des Produktentstehungsprozesses. Neuere Darstellungstechniken erlauben nicht nur die Visualisierung von größeren Informationsmengen, sondern auch von vielfältigen Zusammenhängen. Eine Anwendung dieser Darstellungstechniken in Zusammenhang mit Produktmodellen ist jedoch nicht Stand der Technik.

Die Abbildung von Informationszusammenhängen ist eng mit deren Struktur verbunden wie z.B. hierarchische Strukturen und logische Zusammenhänge. Weiter muss die Darstellung der Produktdatenstruktur bestimmten Anforderungen entsprechen, wie z.B. die Erweiterbarkeit und Kombinierbarkeit von einzelnen Elementen. Die Produktinformationen sind z.B. in Datenbanken oder beliebigen Informationsspeichern enthalten und der Zugriff wird meistens mittels eines Suchprozesses realisiert. Die zurückgegebenen Daten als Ergebnis der Abfrage kann man als Teilmenge aller verfügbaren Informationen betrachten. Inwieweit das Ergebnis den Anforderungen entspricht, kann die Qualität der Ergebnisse bestimmen. Bei Bedarf wird der Suchprozess wieder mit anderen Randbedingungen durchgeführt. Gerade bei diesem iterativen Prozess hat die Strukturierung der Informationen und die Visualisierung der Ergebnisse eine wesentliche Rolle, da die richtig zusammenhängenden Informationen einerseits die Suchprozesse unterstützen, andererseits gewährleisten können, dass keine Informationen, die einen Bezug auf das Ziel haben, unbeachtet bleiben. Nur mittels einer graphischen Darstellung können zusammenhängende Informationsmengen, die bei einem Produktlebenslauf erzeugt werden, nachfolgend übersichtlich visualisiert werden. Die zielgerichtete Darstellung von Ergebnissen einer Informationsabfrage ist die Basis für qualitative Beurteilung und korrekte Betrachtung von Produktinformationen.

Nach Rude [31] unterscheidet man grundsätzlich zwischen integrierten und gekoppelten Produktmodellen.

2.3.1 Gekoppelte Produktmodelle

Jedes Produktentwicklungswerkzeug erzeugt entlang des Produktentwicklungsprozesses seine eigene modellhafte Abbildung des Produktes, welches man Partialmodell nennt. Findet die Unterscheidung aufgrund der Konstruktionsphasen statt, so spricht man hierbei auch von Phasenmodellen (vgl. Rude[31]). Diese Partialmodelle haben eine lose Kopplung über Modelltransformationen, so kann z.B. aus einem 3D-CAD-Modell ein vernetztes Finite-Elemente-Modell abgeleitet werden. Solche Modelltransformationen sind, so sie existieren, nicht unbedingt bidirektional. Wird ein lebensphasenorientiertes Produktmodell aus diesen Partialmodellen aufgebaut sind $n-1$ Transformationsprozesse notwendig.

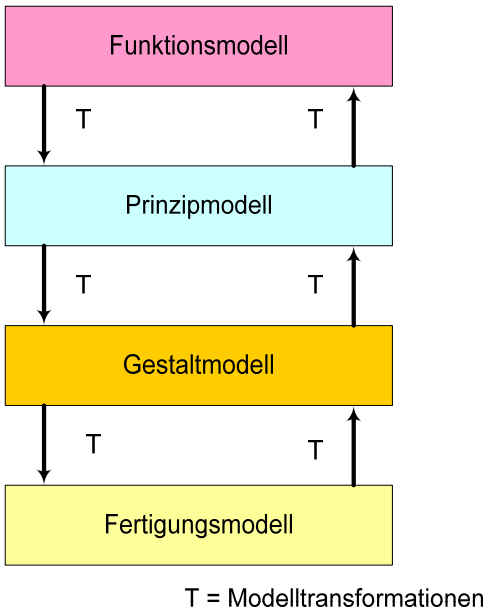


Abbildung 7: Integriertes Produktmodell auf Basis von Modell-Transformation (Quelle: Rude[31])

2.3.2 Integrierte Produktmodelle

Integriert man die Modelltransformationen z.B. in Form von Constraints in das eigentliche Produktmodell, so erreicht man ein integriertes Produktmodell auf Basis von Modellkohärenz. Diese Modelle erreichen eine höhere Integrationstiefe und Redundanzfreiheit.

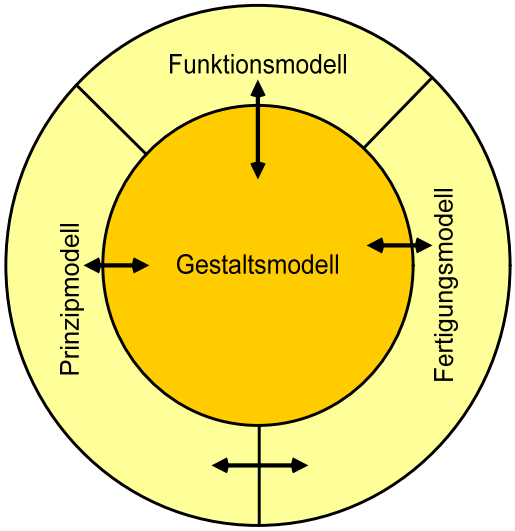


Abbildung 8: Integriertes Produktmodell auf Basis von Modellkohärenz
(Quelle: Rude[31])

2.3.3 Produktmodelldaten in STEP

Der in der Praxis am weitesten fortgeschrittene Ansatz, Produktdaten homogen in einem Modell abzubilden, z.B. zum Zwecke des Datenaustausches, ist STEP[8] (ISO 10303). STEP ist primär eine Methodik, anwendungsspezifische Produktdatenmodelle (Application Protocols) unter Verwendung von Grundbausteinen (Integrated Resources) nach definierten Regeln und genormten Methoden zu beschreiben. Diese anwendungsspezifischen (Partial-)Modelle durchlaufen dabei eine fest vorgegebene Reihe von Bearbeitungs- bzw. Freigabestati, wobei die oberste Stufe die so genannten Anwendungsprotokolle (Application Protocols) bilden. Sie spezifizieren jene Datenmodelle, die die Grundlage für physische Implementierungen eines STEP Prozessors sind. Jede STEP-Implementierung (z.B. ein Datenaustauschprozessor) ist die Implementierung eines Anwendungsprotokolls. Es befinden sich aktuell sehr viele Anwendungsprotokolle in Bearbeitung. Zu den wenigen, die bisher den Reifegrad eines Internationalen Standards erreicht haben, gehören:

- AP203 (Configuration Controlled Design)
- AP212 (Electrotechnical Design and Installation)
- AP214 (Core Data for Automotive Mechanical Design and Processes)

STEP definiert aber auch die Beschreibungssprachen (EXPRESS, EXPRESS-G, EXPRESS-X, …), die verwendet werden, um die Datenmodelle zu spezifizieren. Die Datenmodellierungssprache (Beschreibungsmethode) EXPRESS und die graphische Variante EXPRESS-G ist der erste

[8] STandard for the Exchange of Product model data

internationale Standard zur Spezifikation von Datenmodellen. Mit ihrem objektorientierten Ansatz können

- Objekte (Entities) mit Eigenschaften (Attributes)
- Vererbungsregeln (Inheritance)
- Integritätsbedingungen (Rules, lokal für Objekte und global für alle Objektausprägungen)
- Objektklassen (Schemata)
- Beziehungen zwischen Objektklassen (Schema-Interoperability)

abgebildet werden. EXRESS-X wird verwendet, um zwei existierende Datenmodelle semantisch ineinander abzubilden. EXPRESS ist, analog IDL, eine Spezifikationssprache (keine Programmiersprache) zur logischen Beschreibung von Informationsmodellen. Ein in EXPRESS beschriebenes Informationsmodell kann mit einer konzeptionellen Schemabeschreibung für Datenbanken verglichen werden. Es ist somit noch unabhängig von einer speziellen Implementierung, liegt jedoch schon in rechnerverarbeitbarer Form vor. Damit ist es möglich, ein solches logisches Modell (Spezifikation) mit Hilfe von Softwarewerkzeugen in verschiedene Anwendungssysteme abzubilden. Abbildung 9 zeigt die Baukastenstruktur von STEP mit der Integrated Resources als Kern der Struktur und den Anwendungsprotokollen (AP) für die entsprechenden Domänen.

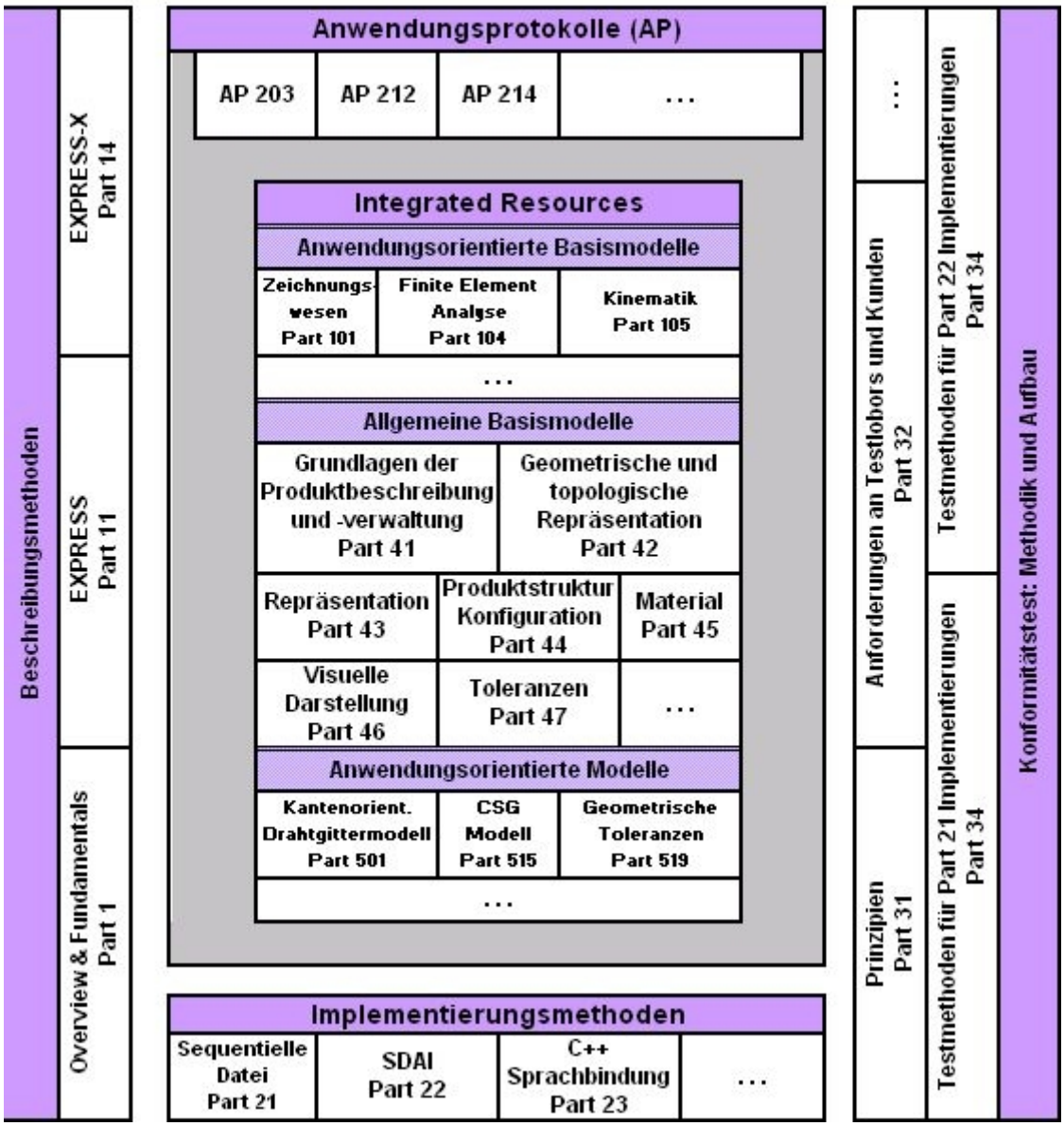


Abbildung 9: Baukastenstruktur von STEP (Quelle: ProSTEP Verein)

Die Problematik bei der Abbildung von Produktdaten in ein STEP Physical File, z.B. mit der Zielsetzung des Austauschs der Daten, liegt in der Interpretierbarkeit der Modelle. Ein Datenaustauschprozessor ist eine softwaretechnische Implementierung eines Anwendungsprotokolls für z.B. ein bestimmtes CAD-System. Es gibt mehre Möglichkeiten, ein konkretes Objekt eines CAD-Systems beim Export in ein STEP Physical File auszuleiten, jedoch mag der importierende Prozessor dieses Objekt in einer ganz anderen Ausprägung erwarten. Selbst die „best practices", eine Sammlung von Handlungsanweisungen, um den Interpretationsspielraum einzuschränken, kann dies in der Praxis nicht verhindern.[33]

2.4 Visuelle Präsentationstechniken

Präsentationstechniken dienen in der Regel dazu eine bestimmte Zielgruppe mit Sachaussagen zu versorgen. Hierzu sind informationsverdichtende visuelle Präsentationstechniken besonders geeignet. Der Begriff der Immersion beschreibt in diesem Zusammenhang das „Eintauchen“ in eine künstliche Welt. Je höher der Grad der Immersion desto ausgeprägter ist die Erfahrung eines Anwenders, sich in einer virtuellen Welt zu befinden, und interaktiv mit Objekten aus dieser künstlichen Welt zu interagieren.

2.4.1 Grundlagen der immersiven Visualisierung

"Mittels der Informationsvisualisierung werden leistungsfähige Darstellungs- und Interaktionstechniken für umfangreiche Datenmengen zur Verfügung gestellt. Dadurch werden Eigenschaften von konkreten oder abstrakten Objekten auf graphische Weise veranschaulicht, indem Abbilder der zugrunde liegenden Daten erzeugt werden." [32]
Im Gegensatz zu physikalischen Daten wie Strömungsdaten etc., verfügen die Objekte der Informationsvisualisierung über keinen räumlichen Bezug, was bedeutet, dass hinsichtlich der Darstellung dieser Objekte eine Konkretisierung erfolgen muss.

Bestehende Darstellungsverfahren können aufgrund ihrer meist zweidimensionalen Beschränktheit nur Ausschnitte von Informationsmengen zeigen. Hier sind neue Ansätze gefragt, die die umfangreichen Informationsmengen des Produktlebenslaufes im Gesamtzusammenhang darstellen können und zusätzlich in der Lage sind, auf Informationen aus den verteilten Systemen des Produktentwicklungsprozess zugreifen zu können.

"Verfahren der Informationsvisualisierung bieten zur Darstellung großer und komplexer Informationsmengen spezielle Techniken. Die Suche nach Informationen und der nachfolgende Zugriff darauf werden durch das Prinzip der Navigation auf der visualisierten Informationsmenge ermöglicht. Der Benutzer interagiert mit der Darstellung, indem er durch die Darstellung navigiert, d.h. sich darin bewegt und die Darstellungsobjekte untersucht. Hierzu erzeugt die Informationsvisualisierung (symbolische) Darstellungen der Informationsmenge, worin anwendungs- bzw. problembezogene Sachverhalte bezüglich der zugrunde liegenden Daten enthalten sind." [32]

Bei Miteinbeziehung der dritten Dimension lassen sich bei bestimmten symbolischen Darstellungsverfahren auf sehr kleinem Raum große Informationsmengen visualisieren. Ziel von Renner [32] ist die Konzeption einer Integration der Informationsvisualisierung in den Produktentwicklungs- und Konstruktionsprozess, wodurch *"eine verbesserte Unterstützung konstruktiver Tätigkeiten beim methodischen Vorgehen im Konstruktionsprozess erreicht und eine interaktive Darstellungsalternative zur Analyse struktureller Zusammenhänge umfangreicher Produktmodelle zur Verfügung gestellt werden soll."* Renner beschreibt zwar die Verknüpfung von Daten aus dem

Produktentwicklungsprozess auf einer Metaebene analog zu einem PDM System. Eine Methode für eine feingranulare Verknüpfung von Metainformationen mit effektiver Bauteilgeometrie wird jedoch erst mit dem in dieser Arbeit beschriebenen Ansatz möglich.

Durch die hohe Komplexität der Datenzusammenhänge ist das unmittelbare Erkennen von wichtigen Eigenschaften nicht möglich. Die Visualisierung muss also hierzu als Vermittler dienen, um für ein besseres Verständnis der komplexen Zusammenhänge zu sorgen.

Das zweidimensionale Diagramm ist beispielsweise eine bekannte Technik zur visuellen Umsetzung technischer Daten. Hier wird der Zusammenhang zweier Größen über Linien oder Punkte veranschaulicht. Wird die Datenmenge größer, sollen z.B. mehrere Fälle (Linien) in das Diagramm eingezeichnet werden, bedient man sich der Variation der Linienform oder Farbe oder erweitert im Falle einer Abhängigkeit durch eine weitere Variable das Diagramm um eine dritte Achse. Es wird hier klar, dass bei einer Zunahme der Komplexität neue Visualisierungstechniken von Nöten sind. Möglichkeiten der Dimensionserweiterung gehen von Interaktionstechniken bis hin zur statischen und dynamischen zeitvariierten Darstellung. [32]

"Das Visualisierungsziel beschreibt, welche Arten von Informationen mit Hilfe der visuellen Analyse aus der graphischen Repräsentation der Daten extrahiert werden sollen". [32]

Grundsätzlich gilt sowohl bei der speziellen graphischen Umsetzung von Informationen aus dem Konstruktionsprozess als auch bei der Visualisierung allgemeiner komplexer Zusammenhänge die gleiche Zielsetzung:

- Erkennung einer **großen Anzahl von Objekten** und deren Eigenschaften im **Gesamtzusammenhang** [32]

- Erkennung der durch die **strukturellen Zusammenhänge** bedingten Wechselwirkungen zwischen den verschiedenen Objekten und Eigenschaften [32]

In Anlehnung an Keller [34] lassen sich für das Anwendungsgebiet von Produktmodellen die folgenden allgemeinen Visualisierungsziele verwenden:

- **Identifizieren**: Festlegung der Eigenschaften, durch die ein Objekt klar erkennbar ist
- **Kategorisieren**: Erkennen von Gemeinsamkeiten zwischen Objekten
- **Gruppieren**: in Gruppen zusammenfügen und Zusammenhänge erkennen

- **Assoziieren**: Herstellung einer gedanklichen Verbindung, um Zusammenhänge zu erkennen
- **Lokalisieren**: Finden durch gerichtete Navigation

Die DIN 66234 [35] [36] [37, 38] bestimmt die Ziele und Besonderheiten der Informationsvisualisierung im Zusammenhang mit graphischen Darstellungsformen:

- Deutliche **Datenreduktion** und Platzersparnis bei qualitativen Informationen
- **Hohe Informationsdichte** bei schneller Erfassbarkeit und sicherer Interpretierbarkeit
- **Erleichterung beim Abschätzen und Vergleichen von Größen** sowie beim Erkennen von Anomalien und Fehlern
- Möglichkeit zur Darstellung zeitlicher oder räumlicher Verläufe
- Erleichterung beim Erkennen von Zyklen und Tendenzen
- Erleichterung beim **Erkennen räumlicher und funktionaler Beziehungen**
- Erzeugung bildhafter Symbole und daraus folgendes schnelles Unterscheiden, Klassifizieren und wieder Erkennen von Bildschirminhalten

Eine Menge von Objekten und ihre Relationen werden in einem so genannten Raum abgebildet. Im Bezug auf den Einsatz werden die Begriffe Datenraum, Informationsraum oder Darstellungsraum definiert. Beispielsweise die Produktdaten, die ein Produktmodell aufbauen, werden in einem Datenmodell abgebildet. Dementsprechend sind die Informationen (Daten + Semantik) Objekte des Informationsraums und die graphischen Repräsentationen der Informationen werden in einem Darstellungsraum abgebildet.

Alle Daten in Bezug zu bestimmten Sachverhalten bilden eine Datenmenge. Werden die Beziehungen dieser Daten zueinander berücksichtigt (aufgrund logischer Funktionen oder semantischer Ähnlichkeiten), ergeben die Zusammenfassung der Daten und deren Beziehungen den so genannten Datenraum. Ähnlicherweise werden die Begriffe für Informationsraum und Darstellungsraum definiert [32].

Man kann vier Ansätze für den Aufbau von Darstellungsräumen unterscheiden:

- **Benediktine-Ansatz** – die Attribute der Darstellungsobjekte werden in innere Dimensionen (die Semantik dargestellt über Hilfsmittel wie Farbe, Form und Größe) und äußere Dimensionen (die Position allgemein in einem n-dimensionalen Raum) unterteilt. Bei

diesem Ansatz generiert der Benutzer ein Schema, das das gewünschte Mapping von Attributen der Datenobjekte zu den inneren und äußeren Dimensionen der Darstellungsobjekte enthält. Beispielsweise wird das Attribut „Ordnung des Produktdatenobjekts“ als äußere Dimension auf der X-Achse abgebildet und der „Typ des Produktdatenobjekts“ durch die Form des Darstellungsobjekts als innere Dimension abgebildet. Als Ansatzmuster wird das Q-PIT-System aufgeführt (vgl. Abbildung 10).

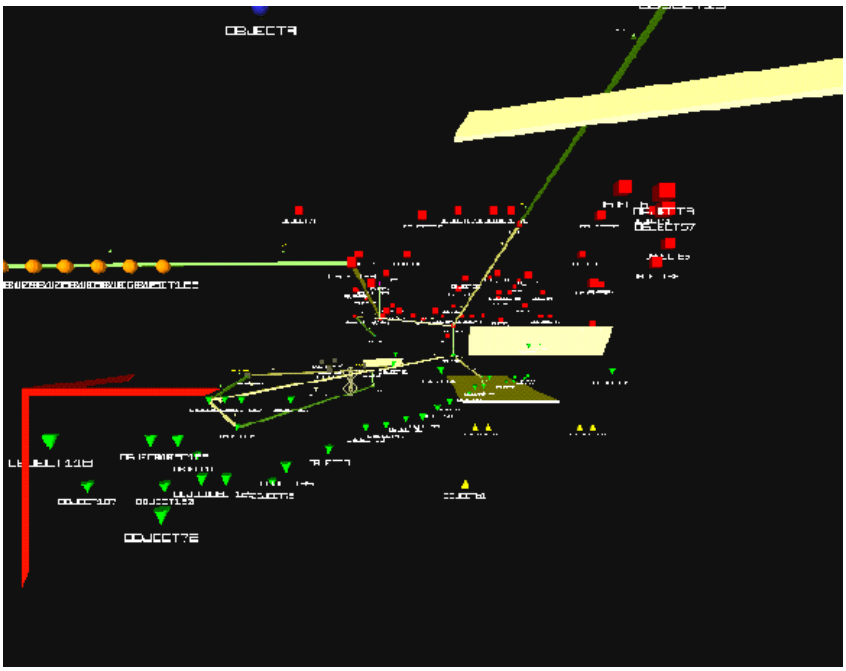


Abbildung 10: Q-PIT[9] Visualisierung unter LEADS [39]

- **Statistische Ansätze** werden für die Strukturierung und Gruppierung von Datenbeständen (oft Dokumente) aufgrund semantischer Ähnlichkeit angewendet. Statistische Systeme finden Einsatz bei der Produktentwicklung, beispielsweise bei der Klassifikation. Systeme, die auf dem statistischen Ansatz basieren sind VR-VIBE und BEAD [39]. Abbildung 11 zeigt eine hierarische Anordnung von klassifizierten Dokumenten (Bücher einer Bibliothek) in der VR-Umgebung DIVE.

[9] PIT (engl. Populated Information Terrain)

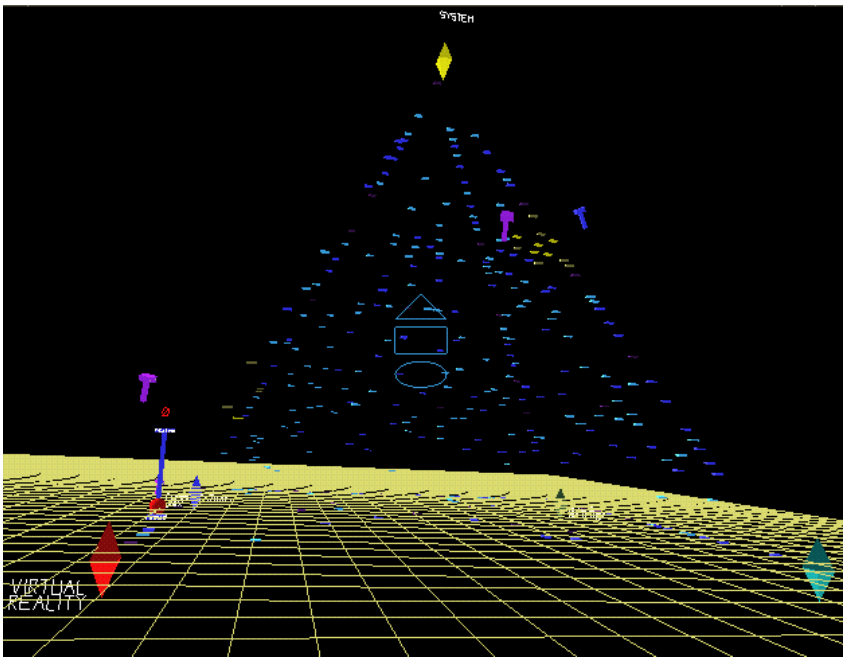


Abbildung 11: VR-VIBE Visualisierung unter DIVE [39]

- **Hyperstrukturen** – Unterstützung einer dreidimensionalen Darstellung von Daten mit Schwerpunkt auf deren Beziehungen. Solche Systeme basieren auf Entität-Beziehungs-Modellen, Hyper-Media und Netzwerk-Informations-Retrieval-Systeme wie Gopher, ARCHIE, WAIS und World Wide Web. Diese Visualisierungstechnik wird vorwiegend für Informationsdarstellung bei Netzwerk-Anwendungen verwendet. Vertreter dieses Ansatzes sind Techniken wie Kegelbäume, fish-eye, perspektive Wall [39]. Abbildung 12 zeigt die Visualisierung einer hierarchischen Hyperstruktur in einer VR-Umgebung.

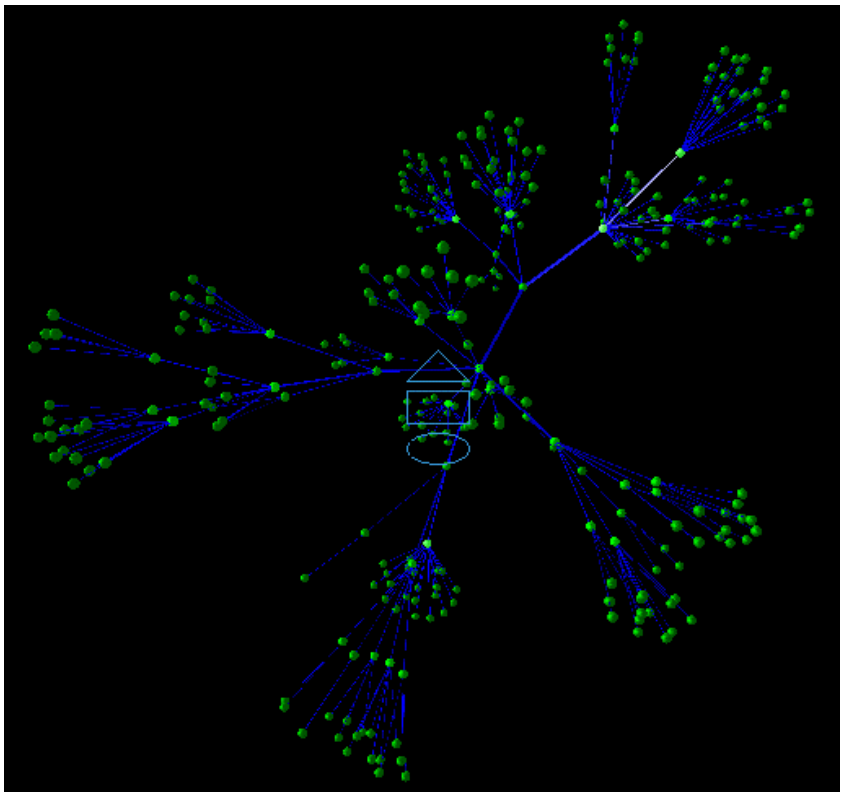


Abbildung 12: Visualisierung mit hierarchischer Hyperstruktur [39]

- **Benutzerspezifische Ansätze** – basierend auf wirklichkeitsabbildenden Metaphern und VR-Techniken, wie fly-through, z.B. für Landkarten. Ein weiterer Ansatz ist das ad-hoc Konzept, das eine spezielle benutzerspezifische Lösung für die Gestaltung des Darstellungsraums anbietet. Die Anwendung von graphischen Metaphern (Symbolen) stellt die Möglichkeit bereit, mehrere Informationen über ein Symbol kompakt zu beschreiben, da die graphischen Metaphern die Eigenschaft haben, die Information zu kodieren. [39]

Diverse prototypische Visualisierungssysteme bieten eine systematische Vorgehensweise zur Erzeugung von Visualisierungsanwendungen an, es sind jedoch keine Systeme oder Anwendungen bekannt, die systematisch auf die Aspekte des Produktlebenslaufs eingehen. [32]

Ein weiterer Begriff in diesem Umfeld ist der *Visualisierungsprozess*. Er beschreibt das methodische Vorgehen beim Abbilden des Datenraums auf den Darstellungsraum, umfasst jedoch nicht die genaue graphische Darstellung der Information. Der Visualisierungsprozess wird auch als Datenfluss-Model bezeichnet.

Die diversen Visualisierungstechniken beschreiben unterschiedliche Ansätze für die visuelle Repräsentation von Daten. Die Ergebnisse sind stark beeinflusst durch die Datentypen und die Strukturen des Datenraums. Die meistgeeigneten Visualisierungstechniken für die Darstellung von

strukturierten Zusammenhängen der Produktmodelle sind die graphbasierten dreidimensionalen Darstellungstechniken.

Eine Teillösung von Renner [32] für die spezifische Informationsvisualisierung im Bereich des Entwicklungs- und Konstruktionsprozess betrachtet die Struktur des Datenraums als „anwendungsbezogene, hierarchische Ordnungsstruktur".

Der Informationsraum beinhaltet nach Renner [32] aufgabenrelevante Informationen über ein Produkt. Die Zusammenfassung und Strukturierung der Informationen dient als Aufbereitung für die Abbildung im Darstellungsraum. Für die Repräsentation von Produktinformationen, die auf die Produktentwicklungs- und Konstruktionsphase bezogen sind, ist ein allgemeingültiges Darstellungsmodell entwickelt worden. Mit diesem Darstellungsmodell werden unterschiedliche Darstellungsobjekte mit bestimmten Darstellungseigenschaften (Farbe, Form) definiert. Die Darstellungsobjekte werden in statische und dynamische unterteilt. Die statischen Darstellungsobjekte werden durch Symbole dargestellt (Glyphen, Metaphern). Die dynamischen Darstellungsobjekte ermöglichen die Navigation im 3D-Visualisierungsraum – Translation, Rotation, Blickrichtung und Skalierung sind die Operationen, die bereitgestellt werden.

2.4.1.1 Visualisierungssysteme

Das Referenzmodell zur methodischen Unterstützung der Visualisierung der Informationen von Produktmodellen nach De Ferrari und Robertson [40], welches sich aus den folgenden sechs Komponenten zusammensetzt, wurde in dieser Form noch nicht systematisch auf die Visualisierung im Produktentwicklungsprozess bezogen.

- **Datenmodell**: Daten und deren Struktur
- **Visualisierungsspezifikation**: benutzerdefinierte Systemvorgaben und benutzerdefinierte Visualisierungsziele.
- **Visualisierungsrepräsentation**: formale Beschreibung der Visualisierungsobjekte und Eigenschaften
- **Visualisierungsausgabe**: Merkmale der technischen Ausgabeeinheiten
- **Abbildungsprozess**: Abbildung der Daten auf die Visualisierungsrepräsentation unter Berücksichtigung der Visualisierungsspezifikation
- **Interaktionskomponente**: Interaktionen auf den Daten sowie Interaktionen auf dem Visualisierungsprozess

Bestehende Visualisierungssysteme, die sich auf das o. g. Referenzmodell stützen, haben ihren Ursprung in der Aufbereitung allgemeiner Dokumentmengen und sind nicht im Speziellen auf die Darstellung von Daten aus dem Produktenstehungsprozess ausgelegt. Folgende Anwendungen bieten einen Zugriff auf komplexe Dokumentstrukturen und können in geeigneter Form zur Visualisierung der Zusammenhänge von Produktmodellen eingesetzt werden:

Visualisierungstechniken primär zum Informationsretrieval:

- **Kegelbäume**: (Cone Tree) 3D-Erweiterung der bekannten 2D-Baumstrukturen durch ein räumliches, gerichtetes Aufspannen von Kegelbäumen beginnend bei den Knoten. Durch selektives Ein- und Ausblenden oder Hervorheben von Teilstrukturen ist ein Navigieren und ein gezieltes Auslesen gewünschter Elemente möglich. [41]

- **Fractal Tree**: Vergleichbar mit den o. g. Kegelbäumen, mit dem Unterschied, dass die Länge der Verästelung mit der Tiefe der Hierarchieebene exponentiell abnimmt. Rein theoretisch können beliebig viele Informationen in einem endlichem Raum dargestellt werden. Der Nachteil der Verkleinerung der Kegel und damit der Verlust der Übersichtlichkeit werden in Kauf genommen. [41]

- **Hyperbolic Tree**: Baumstruktur, die sich nicht in eine Richtung, sondern konzentrisch ausbreitet. Wird ein Unterelement angewählt, wandert dieses ins Zentrum des Kreises. [41]

- **Perspective Wall**: Darstellung linearer Informationsstrukturen an einer dreigeteilten virtuellen Wand. Während auf den äußeren Flächen der Gesamtkontext in Form von "Kacheln" ersichtlich ist, wird das gerade ausgewählte Element auf die zentrale Wand projiziert. [41]

- **Stretch Tools**: 2D-Umsetzung einer komplexen Netztopologie. Durch Vergrößern interessanter Bereiche ist es ähnlich einer Landkartendarstellung möglich, verborgene Details offen zu legen. [41]

- **Dokument Lens**: Außer der Darstellung eines Hauptdokumentes senkrecht zur Benutzeroberfläche, sind weitere Dokumente der Suchauswahl zu sehen, die perspektivisch nach hinten abkippen. Hierdurch ist die Sicht zwar verzerrt, jedoch wird dadurch erreicht, dass auf kleinerer Fläche das komplette Dokument zu sehen ist. Durch Anwählen einer Nebenseite dieser abgestumpften Pyramide wird diese zum Hauptdokument. [41]

- **Info Crystal**: 2D-Darstellung von Ergebnissen komplexer boolescher Anfragen mit der Möglichkeit sowohl qualitative als auch quantitative Aussagen über die Resultatmenge treffen zu können. [41]

- **Information Cube**: Instrument zur Beherrschung sehr großer hierarchischer Strukturen. Die Ebenen der Hierarchie werden durch ineinander geschachtelte, transparente Quader veranschaulicht, wobei sich die Würfel nicht überschneiden. Durch Rotieren und Zoomen ist es dem Benutzer möglich, in immer kleinere Bereiche vorzudringen. [41]

- **LyberWorld**: prototypisches Retrievalsystem, welches 3D-Visualisierungstechniken wie Kegelbäume oder Relevanzkugeln benutzt. [41]

Visualisierungstechniken primär zur Informationsexploration:

- **Tree Map**: 2D-Darstellung einer großen hierarchischen Datenmenge durch den rekursiven "Slice-and-Dice" Mechanismus. Hierbei wird der Bildschirm, den Ebenen des Baumes entsprechend, in vertikale und horizontale Regionen aufgeteilt, wobei tiefere Schichten innerhalb ihres Vorgängers angeordnet sind. Vorteil dieses Systems ist die Möglichkeit Operationen, wie Sortieren, Kopieren etc. leicht anwenden zu können, dafür aber muss eine überladene Darstellung in Kauf genommen werden. [41]

- **BEAD**[10]: 3D-Darstellung von Ähnlichkeitsbeziehungen über Wortvergleiche in Dokumenten. Dreiecksförmige Körper repräsentieren symbolisch die Dokumente und gruppieren sich je nach Thematik oder Suchauswahl zueinander und bilden somit Cluster, die wiederum Rückschlüsse auf den gesamten Datenbestand zulassen. [41] [42] [43]

- **VisDB**[11]: 2D-Visualisierungssystem, mit dem sehr große Datenmengen nach Relevanzkriterien geordnet werden können. Jedes Dokument wird lediglich als ein Pixel dargestellt und gibt durch geeignete Positionierung und Farbwahl Aufschluss über die Übereinstimmung der Suchanfrage. Aufgereiht werden die Pixel spiralförmig, wobei die Relevanz nach außen hin abnimmt und die Farbe der Pixel dabei dunkler wird. [41]

- **Table Lens**: 2D-Darstellung von relationalen Datensätzen in Tabellenform. Die Spaltenbreiten und Zeilenhöhe sind variierbar und lassen somit Fokussierungen von

[10] BEAD bezeichnet ein Programmsystem in Anlehnung an ein Glasperlenspiel.

[11] VisDB (engl. Visualization Database)

Bereichen zu. Außerdem helfen graphische und symbolische Elemente, wie Balken, Farbe oder Schattierungen dem Benutzer, Beziehungen zu erkennen und interessierende Bereiche näher zu betrachten. [41]

- **IVEE (Information Visualization and Exploration Environment)**: Umfangreiche Umsetzung verschiedener Darstellungsmethoden, wie TreeMap oder ConeTree. Die Benutzeroberfläche stellt sowohl die Suchanfragespezifikationen als auch die Resultate dar. [41]

2.4.1.2 Projektionstechniken

In den letzten Jahren haben sich stereoskopische Projektionssysteme gegenüber den klassischen Datenhelmen (Head-Mounted Devices) als Displaysysteme für VR-Umgebungen durchgesetzt. Sie finden, insbesondere in Verbindung mit geeigneten Interaktionstechniken, beim praktischen Einsatz die derzeit größte Benutzerakzeptanz. Projektionstechniken, welche in der virtuellen Realität ihre Anwendung finden, unterscheiden sich zum einen in der Art der Projektion und zum anderen in der Anzahl der Seiten, die für solch eine Projektion genutzt werden. Es ist hinlänglich bekannt, dass räumliches Sehen zustande kommt, indem beide Augen unterschiedliche Bilder an das Gehirn liefern. Sollen computergenerierte Bilder zu einem räumlichen Eindruck führen, so muss ein Rechner eine Darstellung je einmal für das rechte und das linke Auge berechnen. Die beiden Ansichten müssen dann dem jeweils "richtigen" Auge zugeführt werden. Verschiedene technische Lösungen wurden im Laufe der Zeit entwickelt, um dies zu bewerkstelligen.

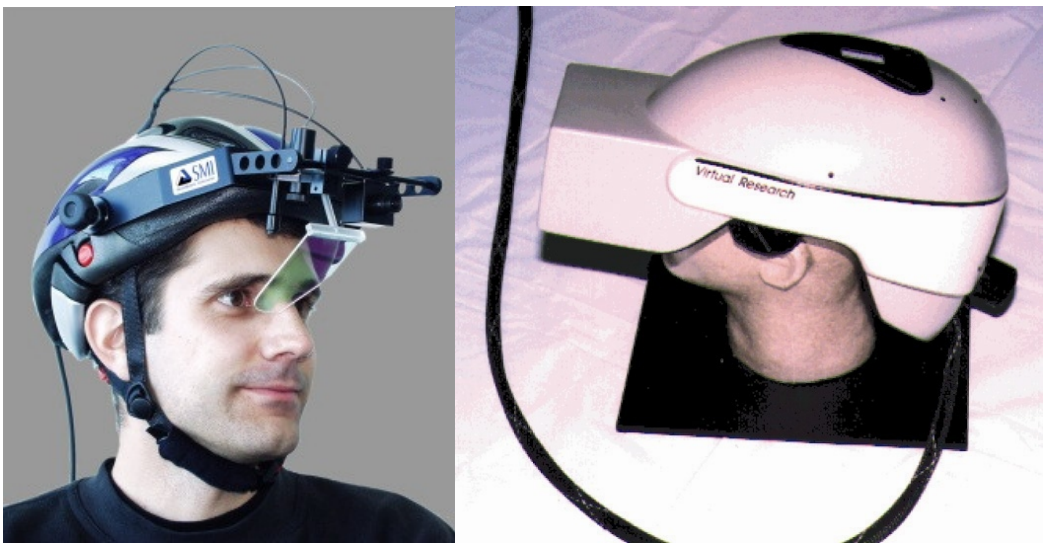


Abbildung 13: Head Mounted Device[12]

[12] Quelle: SMI – Sensomotoric Instruments (http://www.smi.de/)

Head mounted devices sind die Geräte der Anfangsphase von Virtual Reality. Sie bestehen aus zwei Farbdisplays, die sich innerhalb des Helmes befinden. Das Bild jedes Displays ist für genau ein Auge des Anwenders sichtbar. Der Benutzer wird durch den Helm von der Außenwelt vollkommen isoliert und mittels der Displays, die unterschiedliche zusammenhängende Bilder bringen, in die Illusion einer 3D-Welt geführt. Stereo-Kopfhörer fügen den Bildern die notwendige akustische Komponente hinzu. Am Helm angeschlossene Tracking- und Ortungsgeräte (siehe die weiteren Abschnitte) liefern dem System Informationen über den Aufenthaltsort des Benutzers im virtuellen 3D-Raum. Nachteilig ist das relativ geringe Sichtfeld, das durch solche Geräte erreichbar ist. Die 3D-Illusion ist zwar vorhanden, doch der Benutzer kann sich nicht ohne Einschränkungen durch die virtuelle Welt bewegen.

2.4.1.3 Projektionsarten

Prinzipiell unterscheidet man bei den Projektionssystemen Aktiv- und Passiv-Stereo-Projektionen.

Aktiv-Stereo Projektion

Bei einer Aktiv-Stereo Projektion [44] zeigt ein Projektor in schneller Folge abwechselnd die beiden Bilder für das rechte bzw. linke Auge. Der Betrachter trägt hierbei eine Brille (Shutter-Brille) die mit der gleichen Frequenz jeweils das rechte oder linke Auge abdunkelt. Wichtig ist hierbei, dass der Projektor und die Shutter-Brille so synchronisiert sind, dass jedes Auge nur das für es vorgesehene Bild sehen kann. Auch muss die Bildfrequenz sehr hoch sein (mindestens 120 KHz), um die Augen nicht zu stark zu belasten. Ein Beispiel zeigt Abbildung 14.

Abbildung 14: Crystal Eyes[45]

Passiv-Stereo Projektion

Im Gegensatz zur Aktiv-Stereo Projektion werden bei der Passiv-Stereo Projektion von zwei Projektoren unterschiedliche Bilder auf dieselbe Fläche projiziert. Damit jedes Auge nur das für es vorgesehen Bild sehen kann sind die Projektoren mit Filtern ausgestattet. Hierbei kommen Polarisationsfilter sowie frequenzselektive Filter (INFITEC) zum Einsatz. Der Betrachter trägt ebenfalls einen entsprechenden Filter. In beiden Fällen sind spezielle Brillen beim Benutzer zur Erstellung der 3D-Illusion notwendig. Die Abbildung 15 zeigt ein solches spezielles Augengläser-paar.

Abbildung 15: INFITEC Brille[46]

2.4.1.4 Anordnung der Projektionsflächen

Die Projektionssysteme reichen von einer einzelnen stereoskopischen Projektionsfläche, der so genannten Powerwall und der Immersadesk / Workbench, über Systeme wie der zweiseitigen Responsive Workbench bis hin zur CAVEE, die sich aus einer Anzahl einzelner Projektionsflächen zusammensetzt, und zur Cybersphäre.

Immersadesk/ Powerwall

Immersadesk/Workbench ist eine schreibtischgroße, auf 45° geneigte Projektionsfläche. Der Benutzer kann meistens mit Datenhandschuhen in die virtuelle 3D-Welt eingreifen (siehe Abbildung 16). Die Powerwall ist eine große, teildurchsichtige Leinwand, auf die von hinten das Bild projiziert wird (siehe Abbildung 17). (Für weniger anspruchsvolle Arbeiten können auch 3D-Monitore genutzt werden, doch ihr Einsatz ist wegen der beschränkten Möglichkeiten der Eingabegeräte relativ gering).

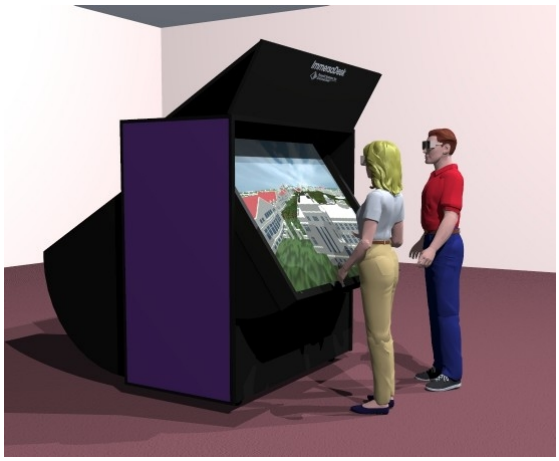


Abbildung 16: Immersadesk [47]

Abbildung 17: Powerwall [47]

Holobench / Responsive Workbench

Wie in Abbildung 18 erkennbar, ist eine Holobench ein Gerät mit zwei Projektionsleinwänden, die waagerecht und senkrecht angeordnet ein "L" bilden. Hinter den Bildschirmen befinden sich leistungsfähige Projektoren, die die 3D-Bilder erzeugen. Wie im Falle der Powerwall lassen die großen Projektionsflächen die Bilder realistisch wirken. Dazu kommt noch die Berücksichtigung der Bewegungsparallaxe. Wenn sich ein Betrachter bewegt, nimmt er eine Veränderung der relativen Position von beobachteten Objekten zueinander wahr. Besonders gut zu sehen ist dieser Effekt bei Objekten, die sich teilweise verdecken. Will man diese Art von Wahrnehmung vortäuschen, müssen die computergenerierten Bilder an den Betrachterstandpunkt angepasst werden. Dazu werden Kopfposition und Blickrichtung durch ein sog. Tracking-System erfasst und

an die Anwendung gemeldet. Liefert die Anwendung dann bei einer Positionsveränderung schnell genug eine neue, korrigierte Ansicht des Szenarios, gewinnt die Darstellung enorm an Realitätsnähe. Selbst ohne Stereo-Sehen (das manchen Menschen aus physiologischen Gründen gar nicht möglich ist) entsteht so ein räumlicher Eindruck.

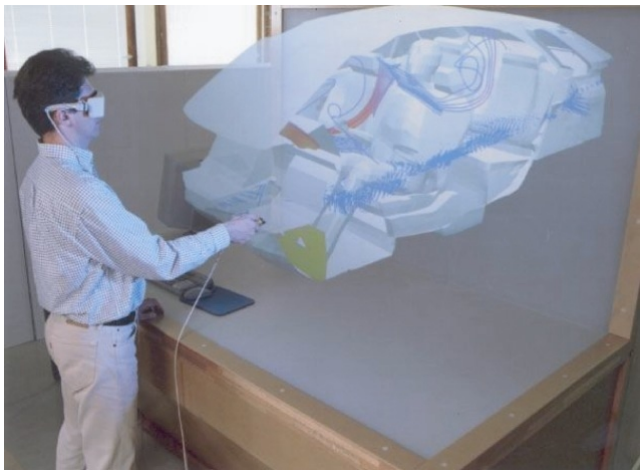


Abbildung 18: Holobench / Responsive Workbench

CAVEE

Das Hinzufügen weiterer Projektionsflächen erzeugt eine CAVEE (=CAVE Engineering Environment). CAVEE gibt es in Bauformen mit drei bis sechs Seiten (siehe Abbildung 19 und Abbildung 20).

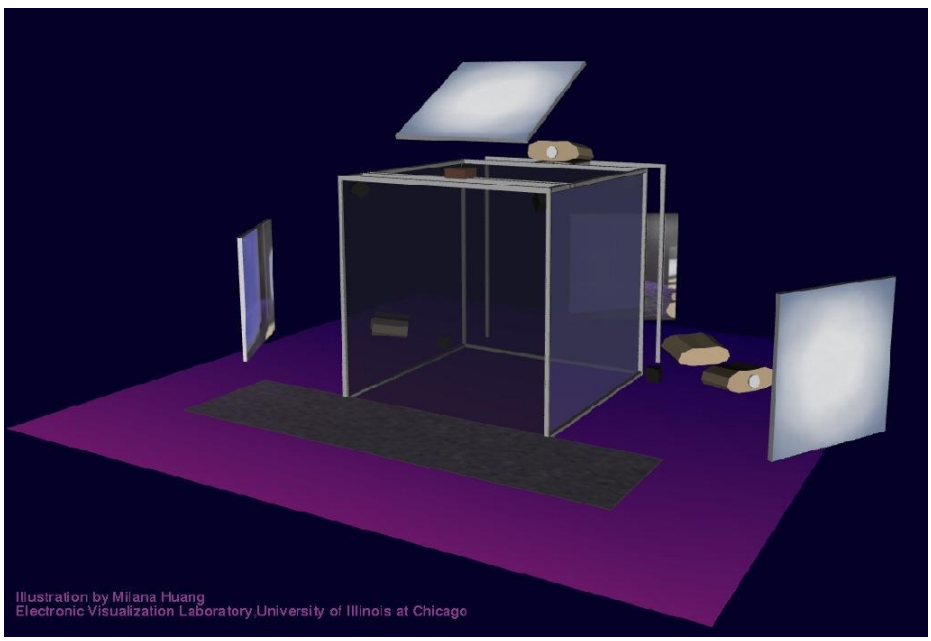


Abbildung 19: CAVEE mit vier Seiten [48]

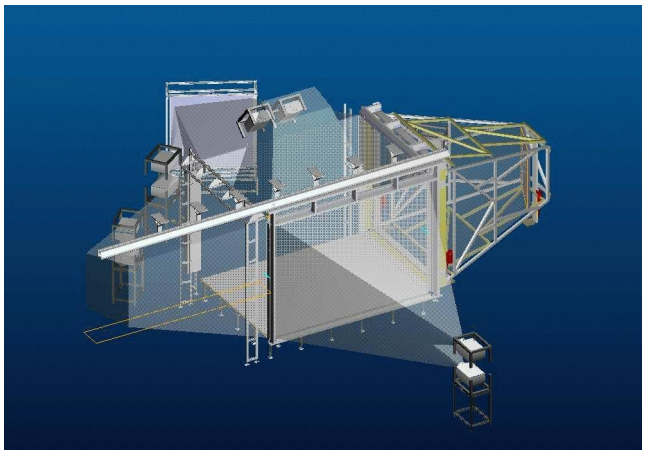


Abbildung 20: CAVEE (Aufbau) [48]

Cybersphere

Alle bisher vorgestellten Systeme haben den entscheidenden Nachteil, dass die Bewegungsfreiheit des Betrachters innerhalb des Projektionssystems durch die natürlichen Grenzen der Projektions-einrichtung oder des Tracking-Systems beschränkt ist. Eine Möglichkeit, diese Restriktion zu umgehen, ist das Verwenden einer großen hohlen durchscheinenden Kugel aus opakem Glas. Die Kugel ist auf einem Ring beweglich gelagert. Der Betrachter kann durch eine spezielle Luke in das Innere der Kugel gelangen und seine Bewegungen versetzen die Kugel in Rotation. Die Bilder werden hierbei auf die Außenfläche der Kugel projiziert und können vom Betrachter innerhalb der Kugel wie bei einer Rückprojektion wahrgenommen werden (siehe Abbildung 21, Abbildung 22). Die Abbildung zeigt ein auf der Kugel entstehendes Bild (der 3D-Effekt ist allerdings auf dieser Art von Foto schlecht erkennbar).

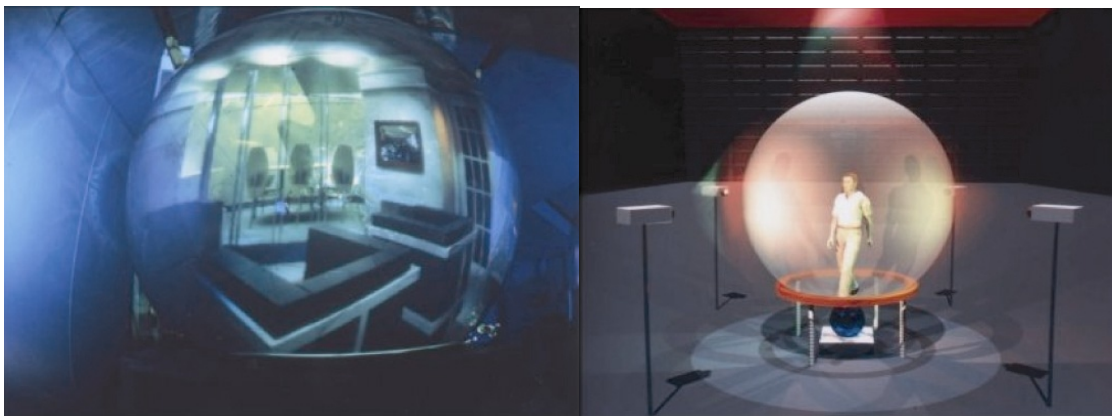


Abbildung 21: Cybersphere [49]

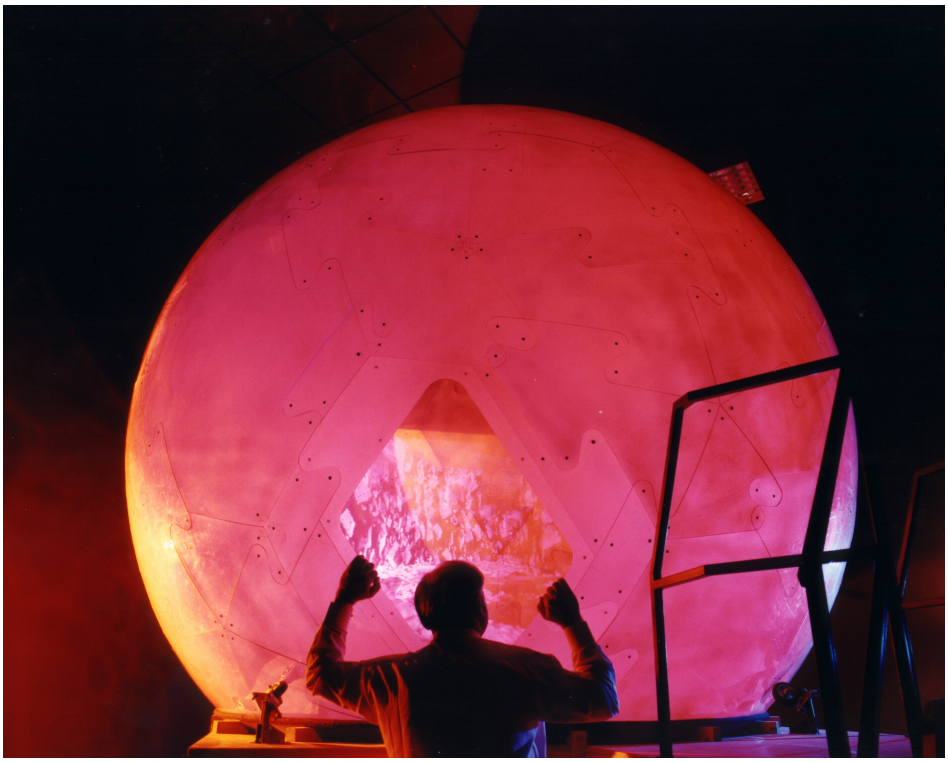


Abbildung 22 Prototyp einer Cybersphere [49]

All diese Ausgabegeräte erfordern, dass die vom Computer erzeugten Bilder schnell genug an eine Änderung der Darstellungsparameter, des Beobachterstandpunktes und der Position der dargestellten Objekte angepasst werden können. Neben einer dafür geeigneten Anwendungssoftware ist die Unterstützung durch eine leistungsfähige Grafikhardware notwendig.

2.4.1.5 Benutzungsschnittstellen

Virtual-Reality-Systeme verbinden räumliche Darstellung und verschiedene Möglichkeiten der effektiven räumlichen Interaktion. So erlauben sie die Simulation echten räumlichen Arbeitens in einer dreidimensionalen Umgebung und erleichtern das Verständnis komplexer drei- und mehrdimensionaler Strukturen und den Umgang mit diesen wesentlich. Eine Schnittstelle besteht aus drei grundlegenden Komponenten: der Eingabe, der Ausgabe und einer Übersetzungskomponente, die die Eingabe in eine Ausgabe übersetzt. Die Ausgabe erfolgt meist visuell über stereoskopische Displaysysteme, die eine räumliche Darstellung unterstützen (siehe voriger Abschnitt). Die visuelle Ausgabe lässt sich anwendungsabhängig zum Beispiel durch akustische, taktile oder haptische Komponenten erweitern. Die Eingabe wird durch Geräte repräsentiert, die eine räumliche Steuerung ermöglichen, wie zum Beispiel die bekannten Datenhandschuhe, Zeigestifte oder die Cubic Mouse.

Die Verbindung zwischen Ein- und Ausgabe erfolgt im Allgemeinen durch einen vielschichtigen Prozess, der meist in ein Anwendungsprogramm eingebettet ist.

Eingabegeräte und die Verbindung zwischen Ein- und Ausgabe

Während sich für Desktopsysteme die Maus und für 3D-CAD-Anwendungen die Space Mouse etabliert haben und dazu noch eine Vielzahl Alternativen existieren, gibt es für VR-Anwendungen weniger Möglichkeiten. Datenhandschuhe, Space Balls und Zeigestifte bestimmen hier das Bild und bieten standardmäßig sechs Freiheitsgrade an. Die Cubic Mouse stellt eine neue Alternative dar, die 12 Freiheitsgrade zur Navigation und intuitiven, objektbezogenen Manipulation in einem Gerät vereint. Die wichtigsten Bestandteile der Eingabegeräte sind Sensoren. Dies können einfache Sensoren, wie Schalter, Tasten oder Schieberegler sein oder komplexere, wie Kameras, Kraft-/ Momenten-Sensoren oder Beschleunigungsmesser. Ein allen VR-Anwendungen gemeinsames Eingabegerät ist das Tracking- oder Ortungssystem, das die Messung der sechs räumlichen Freiheitsgrade (Position und Orientierung) ermöglicht. Heute werden hauptsächlich elektromagnetische Systeme eingesetzt, die aus einem Sender und mehreren daumengroßen Empfängersensoren bestehen. Die in den Empfängern enthaltenen winzigen Antennen messen die Stärke des vom Sender erzeugten elektromagnetischen Feldes. Aus den Messungen lässt sich die Position und Orientierung eines Empfängers im Raum relativ zu dem Sender bestimmen. Diese Ortungssensoren werden zum Beispiel für die Erfassung der Kopfposition der Betrachter eingesetzt. Durch eine Interaktionstechnik wird die Manipulation eines Eingabegerätes in die Manipulation der Anwendung umgesetzt. Man unterscheidet bei den Interaktionstechniken zwischen Navigation, Manipulation und Systemsteuerung. Durch die Navigation bewegen sich Benutzer in einer virtuellen Umgebung. Durch Manipulation werden Objekte bewegt oder deren Eigenschaften verändert. Unter Systemsteuerung versteht man die Veränderung des Zustandes des Systems, wie zum Beispiel die Auswahl aus einem Menü oder das Setzen einer Hintergrundfarbe.

Räumliche Interaktionstechniken erfordern auch räumliche Eingabegeräte. Diese enthalten dann häufig Ortungssensoren, um die Erfassung und Manipulation räumlicher Freiheitsgrade zu ermöglichen. Bisher werden hauptsächlich räumlich arbeitende Zeigestifte, Datenhandschuhe und Space-Balls eingesetzt. Die sechs verfügbaren Freiheitsgrade dieser Geräte erlauben jedoch oft nicht den gewünschten intuitiven Umgang mit den virtuellen Objekten. (Auch dadurch zeigt sich, dass im Bereich der räumlichen Eingabegeräte zurzeit noch erheblicher Forschungsbedarf besteht.)

Die beiden nachfolgenden Abbildungen (Abbildung 23 und Abbildung 24) zeigen einen Datenhandschuh (CyberGlove) und einen Space-Ball.

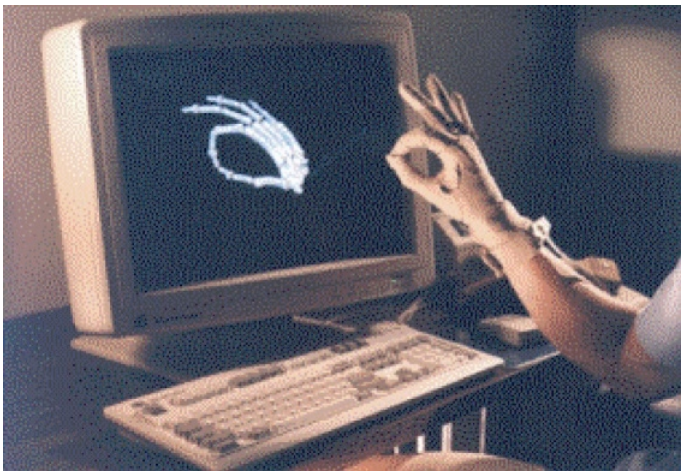


Abbildung 23: CyberGlove [50]

Abbildung 24: Space-Ball [51]

CubicMouse

Einen viel versprechenden Ansatz stellt die CubicMouse dar (siehe Abbildung 25). Dieses neuartige Eingabegerät besteht aus einem würfelförmigen Gehäuse, das einen Ortungssensor enthält und damit die sechs Freiheitsgrade der "klassischen" Eingabegeräte erfüllt. Zusätzlich führen durch das Gehäuse drei verschieb- und drehbar gelagerte Stäbe, die als physikalische Repräsentationen der x-, y-, und z-Achse des Anwendungskoordinatensystems zu betrachten sind und virtuelle Schnittebenen durch das Objekt erstellen. Die CubicMouse verfügt insgesamt über zwölf manipulierbare Freiheitsgrade, sechs Freiheitsgrade werden durch den Ortungssensor bereitgestellt und drei mal zwei Freiheitsgrade durch die dreh- und verschiebbaren Stäbe. Damit befindet sich das dreidimensionale Koordinatensystem in der Hand des Benutzers und erlaubt so einen einfachen und effektiven Umgang mit räumlichen Modellen.

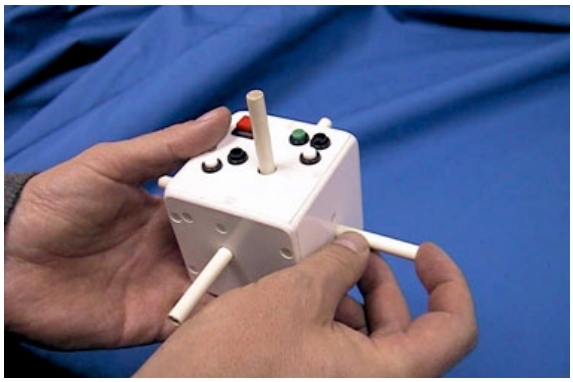


Abbildung 25: CubicMouse [52]

Abbildung 26 zeigt den Einsatz der CubicMouse zum Betrachten eines Fahrzeugmodells. Durch den Ortungssensor können die Orientierung der CubicMouse und die Orientierung des Modells immer in Übereinstimmung gehalten werden. Dreht oder bewegt der Benutzer die CubicMouse, so dreht bzw. bewegt sich das virtuelle Fahrzeug entsprechend mit. Das Verschieben oder Drehen der Stäbe erlaubt die Steuerung eines oder mehrerer Objekte relativ zum Fahrzeugmodell.

Abbildung 26: Anwendung der CubicMouse im industriellen Einsatz [52]

Abbildung 26 zeigt ein Beispiel, bei dem die drei Stäbe drei achsenorientierte Schnittebenen steuern: die Cubic Mouse steuert drei orthogonale Schnittebenen, die Verschiebung eines Stabes steuert die Position der dazu orthogonalen Schnittebene, und die Drehung eines Stabes nimmt eine Feinpositionierung vor (Modell BMW).

Erfahrungen mit den CubicMouse Prototypen haben ergeben, dass Benutzer sehr schnell die Arbeitsweise des Eingabegerätes verstehen und es blind (ohne den Blick von der Anwendung zu nehmen) bedienen können. Die kubische Form des Eingabegerätes lässt sich sehr gut generisch einsetzen. Es wird allerdings empfohlen, dass die Datensätze nur ein implizites Koordinatensystem haben, so dass man die Achsen der CubicMouse den Achsen des Koordinatensystems zuordnen

kann. Bisher wurden erfolgreich Experimente mit Datensätzen aus der Medizin, der Automobilindustrie und der Geologie durchgeführt. Bei Bedarf kann statt des würfelförmigen Gehäuses die Mouse zum Beispiel ein Gehäuse in Form eines Fahrzeugs oder eines Teiles des Fahrzeugs haben.

Systemsteuerung

Eine in vielen VR-Systemen vernachlässigte Komponente ist die Systemsteuerung. Häufig werden lediglich Menüs im dreidimensionalen Raum aufgeklappt und es wird versucht, diese mit einem Zeigestift und einem virtuellen Laserstrahl zu bedienen. Dies frustriert oft die Benutzer, da Menüelemente vielfach zu klein, wegen mangelnder Auflösung unlesbar, wegen niedriger Bildwiederholrate und fehlender taktiler Rückkopplung schwer zu treffen oder von anderen Objekten verdeckt sind. Diese Probleme lassen sich elegant lösen, wenn der Teil der Schnittstelle auf ein anderes Medium ausgelagert wird. Abbildung 27 zeigt eine geeignete Lösung: die Menüs werden mit einem Stift auf PDAs oder PCs mit sehr geringem Gewicht (zwischen 200 und 1000 Gramm), die über eine drahtlose Schnittstelle mit der Applikation verbunden sind, bedient.

Abbildung 27: Systemsteuerung mittels PDA

Die Akustik

Akustik wird heute in VR-Anwendungen hauptsächlich als Ausgabemedium eingesetzt. Neben der akustischen Rückkopplung für bestimmte Ereignisse bietet sich die Sonifizierung von Daten als zusätzliche Ausgabemöglichkeit an und erweist sich insbesondere für zeitabhängige Datenreihen als geeignetes Medium. Einige VR-Systeme unterstützen Spracheingabe zur Steuerung der Anwen-

dung. Dies kann sehr effektiv sein, erfordert von Benutzern jedoch das Erlernen bestimmter Befehlssequenzen. Insbesondere beim kollaborativen Arbeiten treten immer wieder Probleme auf, da die Unterscheidung zwischen normaler Unterhaltung und dem Befehlsmodus oft schwer zu treffen ist. Weitere Formen der akustischen Eingabe sind denkbar und werden im künstlerischen Bereich bereits erprobt.

Kraftrückkopplungsgeräte

Kraftrückkopplungssysteme bieten die Möglichkeit, Benutzer mit haptischen und taktilen Rückkopplungen zu versorgen. Diese Systeme lassen sich so programmieren, dass virtuelle Oberflächen ertastet und deren Struktur gefühlt werden kann. Gleichzeitig können damit die vom Benutzer ausgeübten Kräfte erfasst und die Position und Orientierung der Hand des Benutzers festgestellt werden. Allerdings verfügen diese Geräte über eine eingeschränkte Reichweite und verlangen sehr hohe Updateraten im Bereich von 500 bis 1000Hz um ruckfrei zu arbeiten, was sehr hohe Anforderungen an die verwendeten Algorithmen und Hardware stellt. Abbildung 28 zeigt das Kraftrückkopplungsgerät (SensAble PHANToM) mit sechs Freiheitsgraden.

Abbildung 28: SensAble PHANToM Force Feedback Device [53]

Freiheitsgrade

Eingabegeräte mit mehr als sechs Freiheitsgraden wurden bislang kaum erforscht. Die Cubic Mouse (siehe den vorherigen Abschnitt) stellt eine Innovation in diesem Bereich dar. Grundsätzlich erwartet man bei einer Steigerung der Anzahl manipulierbarer Freiheitsgrade auch eine Steigerung der Komplexität für die Handhabung. Bringt man allerdings die Freiheitsgrade in den richtigen Zusammenhang und beschränkt die Anzahl der gleichzeitig zu manipulierenden Freiheitsgrade, dann lassen sich zusätzliche Freiheitsgrade so einsetzen, dass sie andere Freiheitsgrade unterstützen

und diese intuitiver eingesetzt werden können. Hierzu kommt die VR/AR Initiative des BMBF zur rechten Zeit. Im Rahmen des vom Ministerium geförderten Projektes VR Interaktionsbaukasten soll ein System aus Hard- und Softwarekomponenten entstehen, das ein schnelles Zusammenbauen (Rapid Prototyping) komplexer Eingabegeräte mit den passenden Interaktionstechniken erlaubt. Getreu dem Motto "Mehr Freiheitsgrade statt weniger" soll auch hier mit einer erhöhten Anzahl von Freiheitsgraden experimentiert werden. Aber auch bei diesen neuen Interaktionstechniken werden die zusätzlichen Freiheitsgrade nicht alle gleichzeitig zum Einsatz kommen, sondern in sinnvoller Beziehung zueinander das jeweilige VR-System um Aktionsmöglichkeiten erweitern und ein intuitiveres effektiveres Arbeiten in der virtuellen Realität erlauben.

2.4.1.6 Visualisierung von 3D Geometrie

Moderne Hardware (Grafikkarten) zur Ausgabe von grafischen Informationen (z.B. OpenGL) sind hochoptimierte Prozessoren zur Darstellung von Dreiecken. Um nun computergrafische Daten in industriellen VR/VE-Szenarien, bestehend aus ebenen Flächen mit einem Polygonzug als berandende Kurve oder Freiformflächen in Echtzeit darzustellen, werden diese durch Tessellierung in primitive ebene Flächen (meist Dreiecke) zerlegt. Dies erlaubt eine effektive Darstellung. Die Problematik bei dieser Vorgehensweise ist, wenn man solche Echtzeit-Visualisierungssysteme im Produktenstehungsprozess verwendet, dass tessellierte Flächen nach einer Gestaltmodifikation im virtuellen Raum nicht mehr in topologisch-geometrische Strukturmodelle eines CAD-Systems zurückgeführt werden können (vgl. Abbildung 31). Die gleiche Problematik tritt z.B. auch bei der Strukturoptimierung im FEM-Bereich auf. Optimierte Bauteile, bestehend aus Voxeln, können dort auch nicht mehr in Volumenmodelle zurückgeführt werden. Prinzipiell hat man zwei Möglichkeiten ein Universum (Computergrafik-Szene) mit vorhandenen Grafiksubsystemen aufzubauen. Man unterscheidet hierbei die *zeichnungsorientierten Systeme* und die *objektorientierten Systeme*. So genannte Szenen-Graph Systeme gehören zu Klasse der objektorientierten Systeme und werden im folgenden Abschnitt näher erläutert da sie sich gut zur Darstellung von 3D Geometrie eignen.

2.4.1.7 Szenen-Graph Systeme

Unter Szenen-Graph basierten System versteht man Softwaresysteme zur Visualisierung, die als bestimmende Datenstruktur eine hierarchische Baumstruktur, bestehend aus Knoten, aufweisen. In der Computergrafik werden sie unter der Klasse der generativen Computergrafik eingeordnet, d.h. sie sind objekt- und nicht zeichnungsorientiert (wie z.B. OpenGL). Die Knoten des Szenen-Graphen können sowohl Geometrieelemente, Transformationen, Beobachtungspunkte (Viewpoint, Camera) aber auch z.B. Lichtquellen repräsentieren. Nachdem der Baum aufgebaut ist, traversiert[13]

[13] Traversieren ist ein gebräuchlicher Begriff für das Durchwandern einer hierarchischen informationstechnischen Datenstruktur von oben nach unten und links nach rechts wobei der komplette Baum besucht wird

eine so genannte Rendering-Engine [54] diesen Baum in zyklischen Abständen und erzeugt daraus, basierend auf einem festgelegten Beobachtungspunkt, das Bild. Hierbei ist ein Qualitätskriterium wie viele Zyklen (Frames) die Rendering-Engine pro Sekunde leisten kann. Die drei bekanntesten Vertreter dieser Szene-Graphen Systeme sind OpenSG, SGI Performer und OpenInventor. Performer wird hier nicht separat betrachtet, weil es OpenSG sehr ähnlich ist.

OpenSG

OpenSG ist ein freies portables Szenen-Graph System das auf der SIGGRAPH 1999 von Dirk Reiners, Allen Bierbaum und Kent Watsen initiiert wurde, nachdem alle Bemühungen von Microsoft, SGI und HP bzgl. des Fahrenheit-Projektes[14] eingestellt wurden. Speziell bei VR-Systemen, die im industriellen Umfeld eingesetzt werden, bestand ein großer Bedarf nach einem Szene-Graph basierten Rendering-Subsystem das direkt auf OpenGL aufsetzt und die Vorteile der bisherigen Systeme wie:

- Portabilität
- Unterstützung von multi-threading[15]
- Unterstützung für mehrfache graphic pipes[16] und cluster
- Erweiterbarkeit und
- Flexibilität

in sich vereint.

[14] vereinheitlichtes 3D Interface von Microsoft und SGI

[15] Ein *thread* ist informationstechnisch eine Menge von Anweisungen. *Multi-threading* bezeichnet die Fähigkeit eines Systems zur „gleichzeitigen" parallelen Ausführung von Anweisungsblöcken auf Single-Prozessor-Maschinen mittels Zeitscheiben-Verfahren bzw. echten parallelen Ausführung auf Multiprozessor-Maschinen

[16] *Graphic pipes* sind logische Einheiten von Grafikprozessoren, es gibt sie in der Ausprägungs das ein Hochleistungsprozessor mehrere graphic pipes besitzt die durch die Software separat angesteuert werden können (PC Grafikkarten), oder in der Ausprägung das eine graphic pipe mehere Prozessoren zusammenfasst (SGI Workstations)

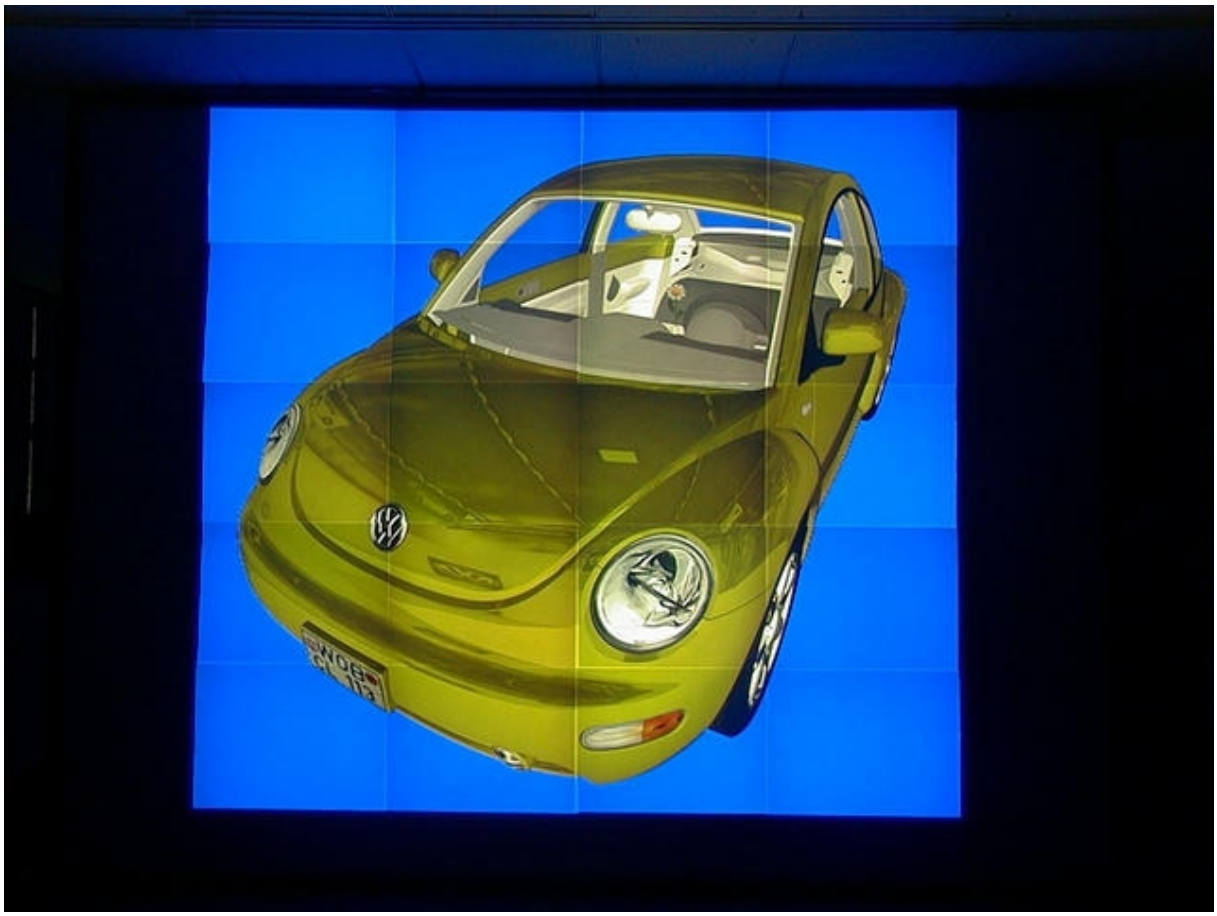


Abbildung 29: VR-Projektionswand mit einer Software basierend auf OpenSG (Quelle: OpenSG Forum http://www.opensg.org)

Prinzipiell unterscheidet OpenSG in zwei verschiedene Knotentypen: `Groups` und `Drawables`. `Groups` strukturieren den Szene-Graphen in logische Einheiten (Sub-Graphen) um z.B. die Auswirkungen von Transformationen einzuschränken. `Drawables` sind üblicherweise Blätter des Szene-Graphen und beinhalten geometrische Objekte, die gerendert werden sollen. OpenSG implementiert einen Knoten des Szene-Graphen so, dass der eigentliche Knoten (`Node`) und sein Inhalt (`NodeCore`) getrennte Strukturen sind. Dies hat, analog den topologisch-geometrischen Strukturmodellen, den Vorteil, dass z.B. Geometrieinformationen (NodeCore) innerhalb des Szene-Graphen nur einmal gespeichert werden müssen, der Knoten an sich jedoch mehrfach verfügbar und somit auch identifizierbar (selektierbar) innerhalb des Szene-Graphen ist.

OpenInventor

Analog zu OpenSG ist das wesentlich ältere OpenInventor Toolkit [55] ebenfalls eine Scene-Graph basierte Cross-Plattform[17] C++ API. Der Szenengraph von OpenInventor ist ein gerichteter

[17] *Cross-Plattform* ist ein Attribut eines Softwaresystems das beschreibt ob dieses System auf mehreren Plattformen gleichermaßen verfügbar ist und unterschiede der Systeme für den Anwender unsichtbar bleiben

azyklischer Graph und besteht zum einen aus Knoten (nodes), die die einzelnen Elemente der Szene und deren Eigenschaften modellieren, und zum anderen aus Gruppen (groups), die die Knoten verknüpfen. Auf den damit definierten Graphen lassen sich speziell implementierte Methoden, so genannte „Actions“ anwenden. Eine typische Action ist etwa das Rendering der Struktur (render action), das Abspeichern (write action) oder das Bestimmen der Größe der Szene (get bounding box action). Das Ausführen einer Action führt im Allgemeinen zu einer Traversierung des Graphen, die tiefenorientiert, also zuerst in die Tiefe, und dann in die Breite durchgeführt wird. Abbildung 30 zeigt einen OpenInventor Szenen-Graph eines Würfels. Das mit eins gekennzeichnete Element stellt den sogenannten Wurzelknoten dar.

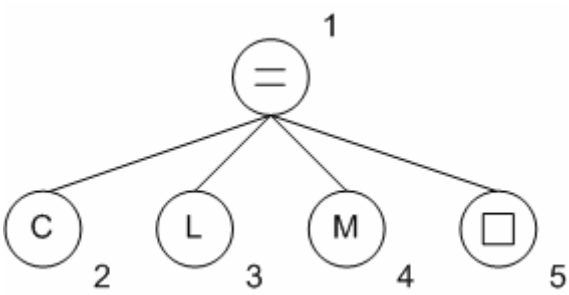


Abbildung 30: OpenInventor Scene-Graph eines Würfels

Für diese Modellierung werden fünf unterschiedliche Knoten benötigt: Links unten zunächst ein Kameraknoten (camera node). Dieser Knoten definiert Kameraposition und Ausrichtung der Kamera, mit der die Szene betrachtet werden soll. Rechts daneben ein Lichtquellenknoten (light node), der Art, Richtung und Stärke einer Beleuchtungsquelle definiert. An dritter Stelle folgt ein Materialknoten (material node). Dieser Knoten enthält mehrere Felder, in die Materialparameter wie etwa Farbe oder Reflektionseigenschaften eingetragen werden können. Rechts unten schließlich befindet sich ein Knoten, der Inventor anweist, einen Würfel in die Szene aufzunehmen. Diese vier Knoten unten werden durch den Gruppenknoten oben zusammengehalten. Die neben den Knoten angebrachten Zahlen beschreiben die Reihenfolge der Traversierung (vgl. Abbildung 30).

Kameras	C	Inventor stellt zwei Kameratypen zu Verfügung. `SoPerspectiveCamera` für die übliche Zentral-perspektive und `SoOrthographicCamera` für eine orthografische Projektion.
Lichtquellen	L	Es gibt im Wesentlichen zwei Arten von Lichtquellen in Inventor: `SoDirectionalLight` definiert den Einfall von parallelen Lichtstrahlen auf die gesamte Szene, ähnlich

		etwa direktem Sonnenlicht; `SoPointLight` definiert eine punktförmige Lichtquelle an einer bestimmten Stelle, die gleichmäßig in alle Richtungen strahlt. Eine Variante davon ist `SoSpotLight`, das vorwiegend in eine Richtung strahlt.
Einfache Grundkörper		Inventor stellt fertige Knoten für Würfel, Kugeln, Zylinder und Kegel zur Verfügung, die jedoch nicht weiter modifiziert werden können und so für ernsthafte Anwendungen ungeeignet sind. Weiterhin können mit `SoText3` Knoten 3D-Buchstaben aus Postscript Zeichensätzen erstellt werden.
Komplexe Körper		Die Modellierung von komplexen Körpern kann mit verschiedenen Methoden geschehen: `SoFaceSet` ermöglicht die Definition von mehreren Flächen über definierende Punkte (Stützpunkte); `SoQuadMesh` stellt ein verformbares quadratisches Gitter zur Verfügung; `SoTriangleStripSet` stellt verformbare Streifen aus Dreiecken zur Verfügung. `SoNurbsSurface` ermöglicht die Definition von gerundeten Körpern mittels NURBS-Kurven. Die Angabe von Eckpunkten und Oberflächennormalen geschieht jeweils über spezielle Knoten `SoCoordinate` und `SoNormal`.
Gruppen		Gruppenknoten sind die elementaren Bausteine, um komplexe Szenegraphen aufzubauen. OpenInventor unterscheidet zwischen `SoSeparator`-Knoten und `SoGroup`-Knoten. Erstere speichern alle Eigenschaften, bevor die darunter liegende Knoten besucht[18] werden und stellen Sie wieder her, wenn die Traversierung abgeschlossen wurde. Dies ist ein bequemer Mechanismus, Eigenschaften lokal für ein Objekt zu definieren. SoGroup Knoten dienen nur zum logischen Gruppieren des Graphen, speichern aber keine Eigen-

[18] man spricht beim „Traversieren" eines hierarchischen Baumes vom „besuchen" von Knotenelementen, d.h. wärhrend des Vorgangs des „Renderns" werden die Knoten besucht und damit die Darstellung des Szene berechnet wobei jeder Knoten seinen Beitrag bringt

		schaften.
Schalter		Um einen Szene-Graphen flexibel verändern zu können, dient der `SoSwitch` Knoten. Er kann wie ein Separator-Knoten verwendet werden, besitzt jedoch ein Feld mit dem Entschieden werden kann, welcher der angeschlossenen Knoten traversiert werden soll. Weitere Optionen ermöglichen es, alle Knoten oder kein Knoten zu traversieren.
Eigenschaften		`SoMaterial` ermöglicht die Angabe von Materialparametern; `SoTransform` definiert eine Transformationsmatrix durch Spezifikation von Position, Größe und Orientierung. `SoDrawStyle` ermöglicht das Umschalten von Wireframe- oder Eckpunktdarstellung. `SoComplexity` ermöglicht es zu bestimmen, wie genau das Rendering zu erfolgen hat, um für subjektiv unwichtigere Objekte weniger Rechenzeit aufzuwenden.
Texturen		Durch Definition eines `SoTExture2`-Knotens lassen sich Objekte mit zweidimensionalen Texturen versehen. Die Texturen können nur Farbe und Transparenz der Objekte modifizieren.

Tabelle 1: Übersicht über die OpenInventor Knotentypen

Java3D

Im Unterschied zu OpenInventor gliedert sich der Szenen-Graph von Java3D in einen Inhaltszweig (content branch) und einen Ansichtszweig (view branch). Der Inhaltszweig enthält hierbei die Definition aller in der Szene enthaltenen Gegenstände (Shape3D Objekte). Der Ansichtszweig enthält alle für die Darstellung der Szene notwendigen Angaben. Die Auftrennung in den Inhaltszweig und den Ansichtszweig findet im `Locale`-Datenelement statt, welches ein hochauflösendes (256-Bit)[19] Koordinatensystem hat. Alle Bestandteile eines Scene-Graph sind In-

[19] Computergrafiksysteme rechnen in Maßzahlen und Einheiten, wenn nun geometrische Entitäten in ein und demselben Universum dargestellt werden deren Einheiten weit auseinander liegen kann es sein, daß die Maßzahlen nach der Umrechnung außerhalb des abbildbaren Bereiches liegen – darum verwendet man Datentypen die einen größeren Bereich abbilden können wie z.B. reine Fließkommazahlen

stanzen der Klasse SceneGraphObject, wobei die Methoden zur Festlegung der Fähigkeiten des Objekts gelesen (z.B. welche Knoten eingehängt sind) und geändert (z.B. Ändern einer Transformation) werden können. Es gibt zwei Arten (Unterklassen) von SceneGraphObject

- Node: Objekte, die (direkt oder als Link) als Knoten in den Scene-Graph eingehängt sind
- NodeComponent: Objekte, die in Knoten als Bestandteile eingetragen werden

Das `ViewPlatform`-Datenelement leistet die Trennung zwischen realer und virtueller Welt. Die endgültige Projektionsmatrix ergibt sich jedoch erst aus weiteren Transformationen. `View` fasst alle zur Projektion des aktuellen Bildes notwendigen Informationen zusammen und bettet sie über einen `Canvas3D` in die GUI[20] ein. [56]

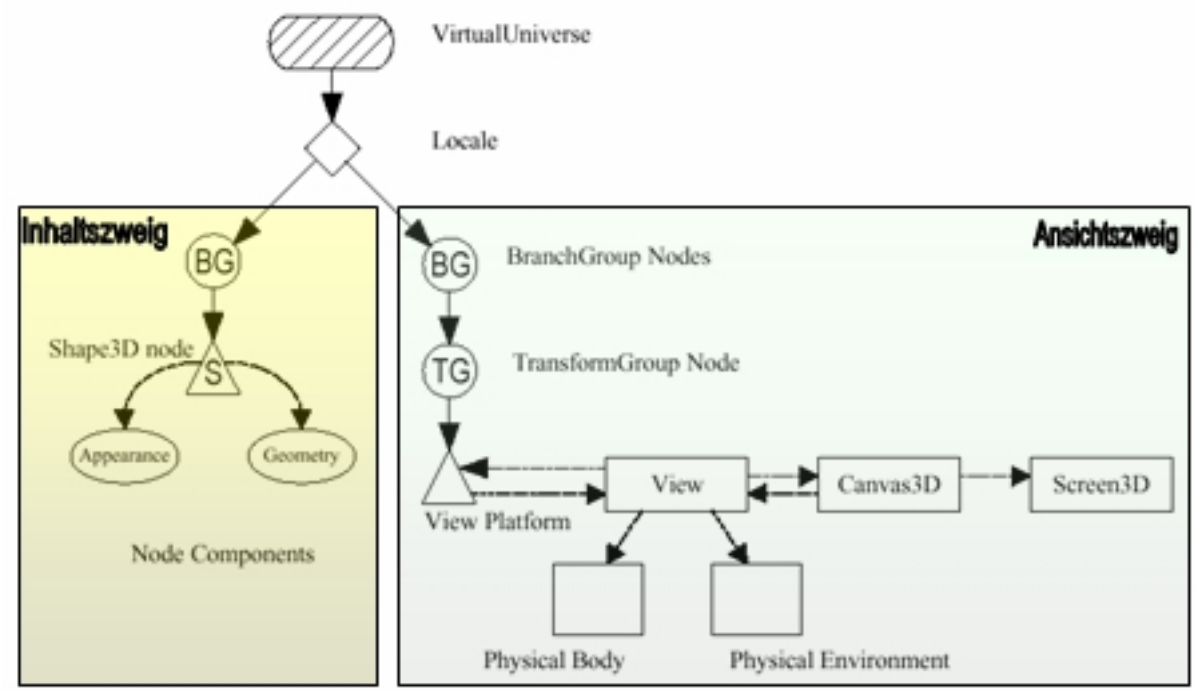


Tabelle 2: Java3D Szenen-Graph

2.5 Virtuelle Realität (VR)

Als Virtuelle Realität (VR) wird die Darstellung und gleichzeitige Wahrnehmung der Wirklichkeit und ihrer physikalischen Eigenschaften in einer in Echtzeit computergenerierten virtuellen Umgebung bezeichnet.

2.5.1 Taxonomie

Die Begriffsbildungen im Umfeld der Virtual Reality (VR) Technologie sind vielfältig. Gebräuchliche Begriffe waren lange Zeit auch Cyberspace, Simulator Technologie, Synthetische

[20] GUI (= Graphical User Interface) grafische Benutzungsschnittstelle

Umgebungen usw., aber erst der Begriff „**V**irtuelle **R**ealität“ erreichte dank der Medien eine gewisse Popularität. Für die einen ist diese Präsentationstechnik untrennbar mit gewissen Ein- / Ausgabegeräten, wie z.B. einer CAVE verknüpft, andere weiten diesen Begriff auf konventionelle Bücher, Filme und sogar die pure Fantasie aus. Allen Definitionen ist jedoch gemeinsam, dass der Betrachter die künstliche Welt von einem Standpunkt aus beobachtet und diese künstliche Welt in Echtzeit auf diese Interaktion reagiert.

Im Rahmen dieser Arbeit referenziere ich mit dem Begriff Virtuelle Realität (VR) jedoch nur computerbasierte Systeme. Die weitaus beste Definition von VR liefert [57].

"Virtual Reality is a way for humans to visualize, manipulate and interact with computers and extremely complex data".

Anwendungsbereiche von VR decken ein großes Spektrum ab und bieten oftmals einen kostengünstigen, flexiblen Ersatz konventioneller Technologien (z. B. Prototypenbau), trotz der hohen Anschaffungskosten der erforderlichen Hardware. Wichtig ist aber hierbei der Fokus auf der Entwicklung neuer Technologien zur Darstellung der Informationen. Eine “Portierung” der 2D-Präsentationstechniken, wie wir sie von herkömmlichen Bildschirmen kennen, kann keinen Vorteil bei der Navigation und Exploration der Informationen bringen und wirkt sich obendrein noch nachteilig auf den subjektiv empfundenen Grad der Immersion aus.

2.5.2 Arten von VR-Systemen

Man unterscheidet folgende VR-Systeme [57]:

- **Fensterbasierte Systeme** (Desktop-VR): Ausgabegerät ist hier ein Computermonitor, der entweder eine 2D-Projektion einer virtuellen Welt darstellt oder mittels der Kombination von einem 3D-Monitor und einem Tracking-System einen dreidimensionalen Eindruck erzeugt.

- **Video Mapping:** Genau wie bei den fensterbasierten Systemen betrachtet hier der Benutzer die Welt durch ein Fenster, mit dem Unterschied, dass er sich selbst in der Szene befindet. Dies wird durch ein Polygonmodell erreicht, auf das in Echtzeit Videodaten des äußeren Erscheinungsbildes des Benutzers projiziert werden.

- **Telepräsenzsysteme:** Diese Kategorie von Systemen beinhaltet sowohl Kommunikationssysteme, die es erlauben, über Weitverkehrsverbindungen Partner eines Meetings virtuell an einen Konferenztisch zu holen, als auch Systeme für minimalinvasive Chirurgie. Gemein-

samkeit dieser Systeme ist die (ferngesteuerte) Verbindung von Sensoren mit den Sinnen eines menschlichen Operators.

- **Mixed- / Augmented Reality:** Diese Systeme sind charakterisiert durch die Überlagerung von physischen (realen) Gegebenheiten mit künstlich berechneten Objekten. Problematisch bei dieser Art von Systemen ist die genaue geografische Lokalisation bzw. Identifikation der physischen Objekte, um sie mit den virtuell erzeugten Objekten zur kombinieren (augmentieren).

- **Immersive Systeme:** Sie stellen die bei weitem komplexeste und teuerste Ausbaustufe von virtuellen Umgebungen dar. Der Benutzer wird hierbei komplett in die künstliche Welt miteinbezogen. Häufigste Implementierung eines solchen immersiven Systems ist eine CAVEE (vgl. 2.4.1.4), aber auch Head Mounted Devices gehören zu dieser Kategorie von Systemen. (Beispiel: Cockpit Simulatoren)

2.6 Kommerzielle VR Softwaresysteme

Die primäre Aufgabe einer Softwareumgebung für VR ist eine Programmierschnittstelle zur Abstraktion der Hardware anzubieten. Hauptsächliche Kriterien für solche Systeme sind Performanz und Flexibilität, wobei diese oftmals in einem direkten Konflikt miteinander stehen. Die Performanz ist dabei im Wesentlichen durch die Performanz der Grafikhardware bestimmt, denn eine hohe Framerate (d.h. effektive Nutzung der Grafikhardware) ist für eine immersive Wahrnehmung zwingend. Flexibilität heißt, dass verschiedene Hard- und Softwarekonfigurationen eingesetzt werden können, denn aufgrund der marktüblichen Preise ist davon auszugehen, dass die zur Verfügung stehende Hardware schon vorhanden ist. Das Gesamtsystem muss portabel sein und keine Limitierungen bzgl. der möglichen Anwendungen implizieren. Die folgende Aufstellung (Tabelle 3) gibt eine Übersicht der zurzeit[21] am Markt verfügbaren Softwaresysteme:

	Funktionalität	**Flexibilität**	**Einfachheit**
TUCAN (AWARON)	• einmal erzeugte 3D-Daten werden in anderen Modulen wieder verwendet. • TUCAN bietet eine integrierte Arbeitsumgebung für Visualisierung, Simulation, Unterhaltung • Keine Integration von benutzerdefinierten grafischen Metaphern • Keine Integration in den Konstruktionsprozess	• es können Daten mit unterschiedlichen Standardformaten importiert werden (z.B. VRML) • kein Export außer VRML möglich	• relativ einfache Anwendung

[21] Stand: Juli 2004 (kein Anspruch auf Vollständigkeit)

RENDERIZER (ModViz aka Siemens Renderizer)	• Zusammenfassung von Programmfunktionen professioneller Simulationsumgebungen für die visuelle Kommunikation für Bauten und Projekte • Strömungssimulation • Keine Integration von benutzerdefinierten grafischen Metaphern • Keine Integration in den Konstruktionsprozess	• in verschiedenen Umgebungen einsetzbar • optimale und stabile Performance • Anpassungen der Software für kundenspezifische Umgebungen. • Installationsaufwand und der Platzbedarf sind gering • Stabil • Gute Performance	• kompliziert zu Installieren • Abhängig vom jeweiligen LINUX Kernel
World Toolkit (Sense 8)	• Bibliothek (Toolkit) zur Implementierung von VR Anwendungen • Verwaltung von Sensoren-Eingaben • Rendering • Laden von Daten-Modelle	• schlechte Performanz • breite VR Hardware Unterstützung • sehr flexibel da nur Toolkit, Anwendung muss zuerst implementiert werden	• komplizierte Einarbeitung
COVISE VR	• unterstützt 3D-Eingabegeräte, immersive stereoskopische Displays und intuitive Interaktion • der Renderer basiert auf SGI Open GL Performer und ist somit für SGI und Linux Systeme verfügbar • Keine Integration von benutzerdefinierten grafischen Metaphern • Keine Integration in den Konstruktionsprozess	• kann auf CAVE, Powerwall, Curved Screen, Workbench, Desktop-Monitor verwendet werden aber mit sehr unterschiedlichem Bedienkomfort. • Nur auf CAVE-Systemen und 3D-Eingabegeräten einsetzbar	
VR Juggler (JVP)	• Aufbau aus Managern, die bestimmte Systemdetails verkapseln und das System kontrollieren und konfigurierten	• plattformunabhängig: eine Anwendung soll in jedem VR-System laufen. Hardware unabhängig, • sehr flexible und effiziente Architektur	• zu umfangreich und komplex
dVS/ dVISE	• unterstützt verteilte Applikationen und Netzwerkzugriffe, importiert CAD Objekte und erzeugt virtuelle Produkt-Repräsentationen • Keine Integration von benutzerdefinierten grafischen Metaphern • Keine Integration in den Konstruktionsprozess	• graphische Skalierbarkeit unabhängig von der Applikation	• kompliziert wegen eigener Programmiersprache, erfordert hohe Spezialisierung
Lightning	• stellt Module zur Beschreibung der virtuellen Umgebung und deren Funktionen zur Verfügung • Keine Integration in den Konstruktionsprozess	• interaktive 3D-Szenarien können erstellt und sehr flexibel und leicht erweitert werden	• sehr einfache Anwendung, auch für Nutzern mit geringen Vorkenntnissen

Virtual Design 2	• OpenGL-basiert, Mehrkörpersimulationen müssen aufwändig durch Matrizen händisch erstellt werden, 3D CAD Import rudimentär • Keine Integration in den Konstruktionsprozess	• flexibel durch spezielle Entwicklerversion die umfangreiche Anpassungen ermöglicht	• einfaches intuitiv zu bedienendes System
SGI- OpenGL Performer	• Anwendungs-Programmierschnittstelle zur Erstellung visueller Simulationen. Deckt die ganze Anforderungs-spannweite komplexer Grafiklösungen ab	• keine Unterstützung für spezielle VR Geräte	• komplex zu programmieren

Tabelle 3: Übersicht über die am Markt Verfügbaren VR-Softwaresysteme

2.7 Virtuelle Realität in der Produktentstehung

Die stetige Verkürzung der Produktentwicklungszyklen, das heißt der zunehmende Zwang in immer kürzeren Zeitabständen innovative Produkte hervorzubringen und dabei die Kosten weiter zu senken, erfordert den Einsatz neuester flexibler Techniken im Produktionsprozess. In vier Anwendungsbereichen ist die VR-Technologie im Moment „Stand der Technik". Es existiert eine unidirektionale Prozesskette zwischen CAD und VR-Anwendungen, um Konstruktionen immersiv einem breiten Publikum, etwa bei Designstudien, zugänglich zu machen. Verteilte virtuelle Umgebungen werden eingesetzt um kooperativ an geographisch verschiedenen Standorten Produktentwicklung zu betreiben, wobei sich die Interaktion mit dem System meistens auf „Redlining"-Funktionalität beschränkt. Konstruktive Änderungen können durch die Unidirektionalität der CAD-VR Prozesskette nicht in CAD-Systeme zurückgeführt werden. VR findet auch Anwendung bei der Visualisierung höher-dimensionaler Daten im CAE-Bereich, wie z.B. bei der Strömungssimulation. Weiterhin ist „Augmented Reality" insbesondere im Service- und Fabrikplanungsbereich im Einsatz. Hierbei werden reale Objekte mit virtuellen Objekten augmentiert, was zur Folge hat, dass der Betrachter eine reale Szene sieht, die mit virtuellen Objekten angereichert ist.

Bisher sind jedoch keine Systeme bekannt, die aus den einzelnen Produktentwicklungswerkzeugen Informationen beziehen, um sie dann integriert mit der Produktgestalt zu visualisieren.

2.7.1 CAD-VR Prozesskette

Abbildung 31 stellt die CAD-VR Prozesskette aus heutiger Sicht dar. Beginnend mit einer Idee, die in eine formale Spezifikation überführt wird, werden in der Phase der Modellbildung Fertigungsunterlagen erstellt. Im Ingenieurbereich sind in dieser Phase in der Regel 2D- bzw. 3D CAD-Systeme anzutreffen. In einer Zwischenstufe werden diese exakten Geometriedaten durch

Tessellierung in ein für die Visualisierung performantes Datenmodell gewandelt, welches dann in einem VR-System visualisiert werden kann. Der Nachteil dieser Vorgehensweise ist die Tatsache, dass die tessellierten Flächendaten unter anderem den Bezug zur Original-Freiformfläche verloren haben. Das heißt, dass eine Geometrieänderung, z.B. während eines Design-Reviews, nicht wieder automatisiert in das Ursprungs-CAD-System übernommen werden. Stand der Technik ist, dass das VR-Modell mit Anmerkungen versehen wird und danach die Änderungen in der Konstruktionsabteilung nachgeführt werden. Aus diesem korrigierten Modell wird wieder ein visualisierbares Datenmodell erzeugt.

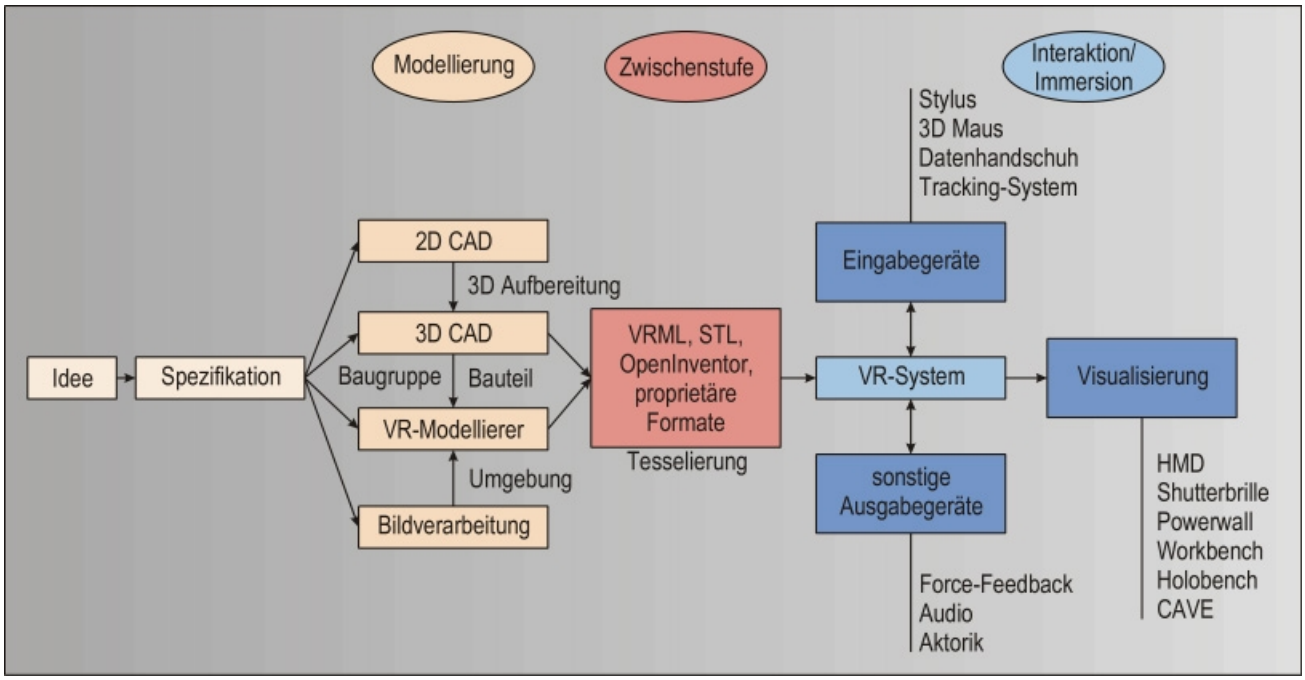


Abbildung 31: CAD-VR Prozesskette

2.7.2 Verteilte Virtuelle Umgebungen

Verteilte Virtuelle Umgebungen unterstützen den kooperativen Produktentwicklungsprozess[22] bei geographisch an unterschiedlichen Plätzen lokalisierten Projektteilnehmern. Ihr primäres Ziel sind Projekt- und Zusammenarbeit in der Produktentwicklung zu unterstützen. Die Funktionalität dieser Produkte ist in der Regel auf folgende Funktionalitäten beschränkt: Zusammenbau, Inspektion, messen, ändern und markup, wobei die momentan erhältliche Software meistens webbasiert ist. Abbildung 32 zeigt den OneSpaceDesigner von CoCreate, bei dem es sich um ein typisches webbasiertes Desktop-Produkt handelt.

[22] Simultaneous Engineering

Bei der kooperativen Produktentwicklung in immersiven VR-basierten Umgebungen kommen meistens Avatare[23] als Modelle für die an der Konferenz beteiligten Projektpartner zum Einsatz. An der ETH Zürich wurde in Rahmen eines Projektes auch die Möglichkeit der Kombination von Videoconferencing in Zusammenhang mit VR Umgebungen getestet. Problematisch war bei diesem Ansatz, dass in einer CAVE in der Regel keine Lichtverhältnisse vorherrschen, wie sie für eine Videokonferenz benötigt werden. Dort hat man dieses Problem gelöst, indem bei der Aktiv-Stereo Projektion ein dritter Shutter-Zyklus eingeführt worden ist, bei dem beide Augen verdunkelt wurden und die VR Umgebung durch einen Blitz erhellt wurde. Durch eine geschickte Wahl der Frequenz konnte ausreichend Licht in der VR-Umgebung zur Verfügung gestellt werden, um Videoconferencing zu ermöglichen, ohne die Immersion zu beeinträchtigen. [58]

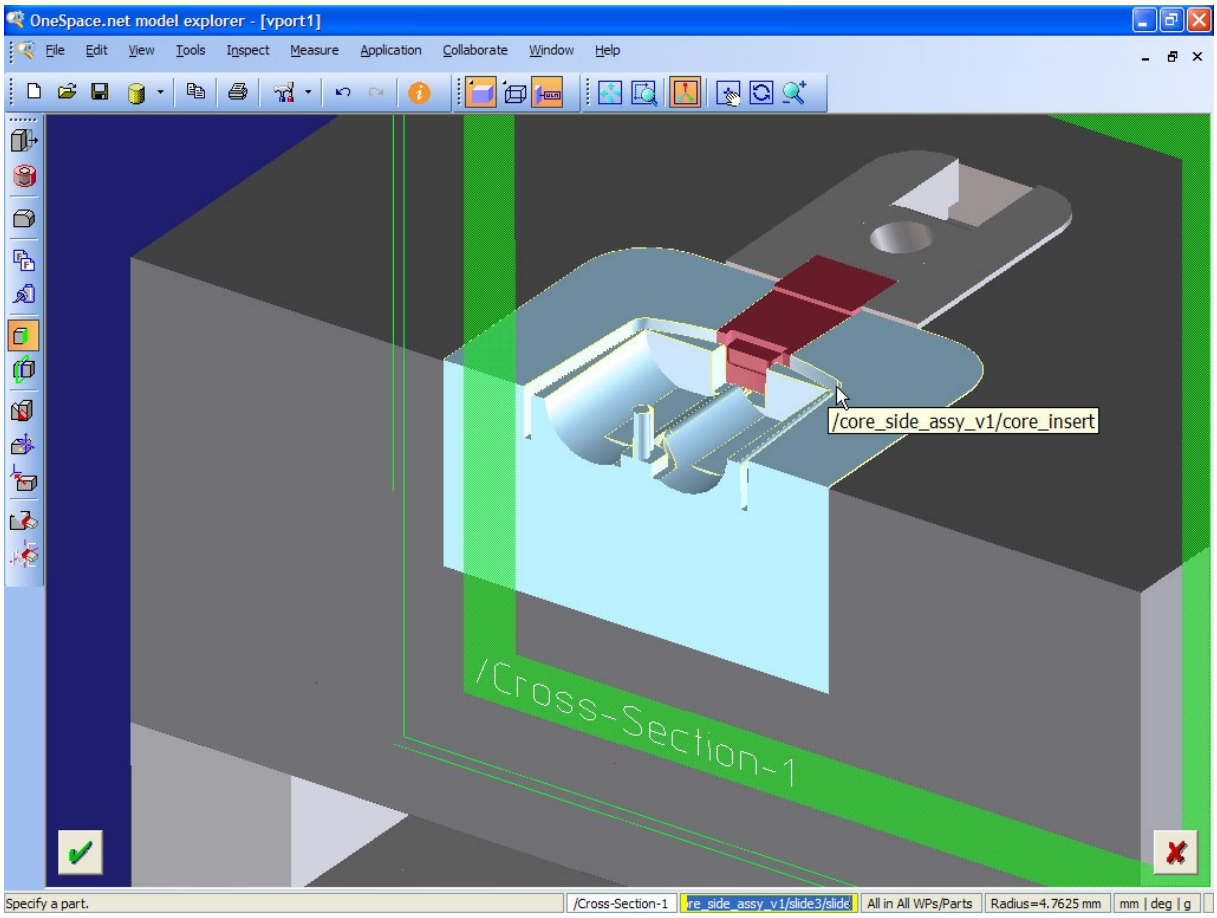


Abbildung 32: OneSpace Designer von CoCreate (Quelle: CoCreate)

2.7.3 Visualisierung höher-dimensionaler Daten

Business Grafiken

[23] Elektronisch generierte Figur, wird dazu benutzt die teilnehmenden Personen zu illustrieren.

Eine der einfachsten und daher auch in den meisten Tabellenkalkulationsprogrammen enthaltenen Methoden sind die Business-Grafik-Methoden wie z.B. Kuchengrafiken, Linien- oder Balkendiagramme. Die Abbildungen leisten gute Dienste, wenn es um das Aufzeigen von Entwicklungen und Trends im Verlaufe der Zeit geht, und sie stellen im Allgemeinen nur moderate Anforderungen an die Rechenleistung. Typische Beschränkungen zeigen sich, sobald man Hierarchien oder komplexe Zusammenhänge darstellen möchte.

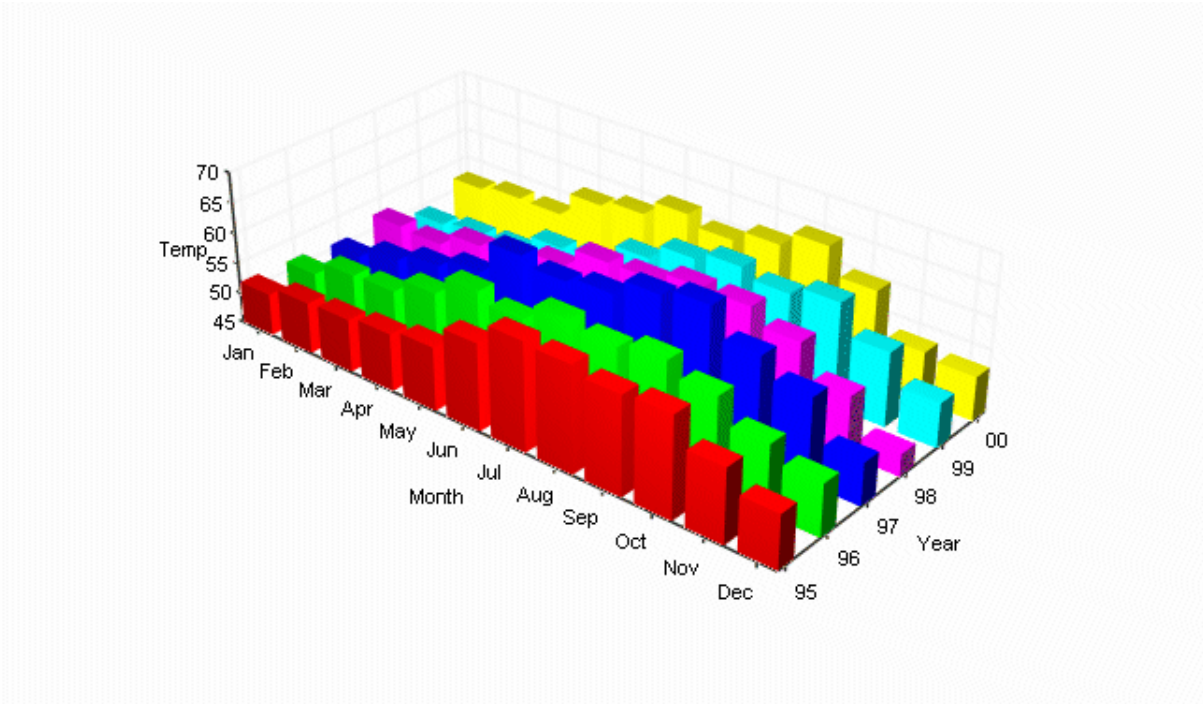


Abbildung 33: Beispiel Businessgrafik

Vorteile	Nachteile
schnell und einfach zu verstehen	schnell überladen bei komplexen Daten
geringe Anforderungen an Rechenleistung	keine schnelle Übersicht möglich
	ungeeignet für mehrdimensionale Daten

Tabelle 4: Vor und Nachteile von Businessgrafiken -

Einfache Höhenfelder

Höhenfelder sind immer noch eine der Standardmethoden für die Visualisierung von regelmäßig gemessenen skalaren Datensätzen. Die Darstellung erfolgt als eine dreidimensionale Abbildung der

Daten, bei der die Farbe die dritte Dimension darstellt. Da Höhenfelder als eine natürliche Erweiterung der zweidimensionalen Business Grafiken angesehen werden können, leiden sie unter denselben Nachteilen. Sie sind v. a. beschränkt, was die Zahl der Dimensionen betrifft.

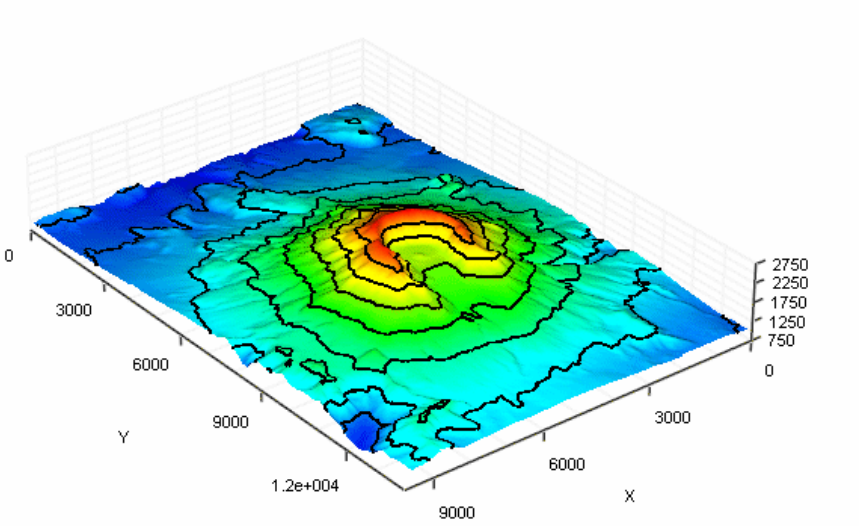


Abbildung 34: Beispiel einfache Höhenfelder

Vorteile	Nachteile
natürliche Erweiterung der Business Grafiken	große Datensets benötigen Filterung
intuitive visuelle Metapher, die eine räumliche Komponente einbringt	beschränkte Anzahl darstellbarer Dimensionen
erlaubt komplexere Datasets darzustellen	benötigt lückenlos und regelmäßig gesammelte Daten
hilft Abweichungen aufzuzeigen	
ermöglicht Interaktivität und Animationen	

Tabelle 5: Vor- und Nachteile von einfachen Höhenfeldern

Triangulations- und Netzdarstellungsmethoden

Im Gegensatz zu den einfachen Höhenfeldern können generische Triangulations- und Netzdarstellungsmethoden auf strukturierten wie auch unstrukturierten Daten angewendet werden.

Falls die Daten unregelmäßig (räumlich oder temporal) gesammelt wurden, kann die Triangulation eine mögliche Methode zur Interpolation sein. Auch falls eine Interpolation nicht benötigt wird, kann sie die Rasterdaten in eine kontinuierliche Form bringen. Umgekehrt können große Datenmengen durch Abbildung in gröbere Netzdarstellungen mit einem akzeptablen Informationsverlust stark reduziert werden, wodurch die zur Darstellung benötigte Rechenleistung ebenfalls stark reduziert wird und gleichzeitig die Möglichkeiten der Interaktivität erhöht werden.

Abbildung 35: Beispiel Triangulations- und Netzdarstellungsmethoden

Vorteile	**Nachteile**
kann mit unregelmäßigen Datensätzen umgehen	große Rechenanforderungen
kann der Datenfilterung und -reduktion dienen	lange Antwortzeiten
vereint Filter- und Abbildungsmethode	
Hierarchien und Detailstufen	
sehr große Datensätze	

Tabelle 6: Vor- und Nachteile von Triangulations- und Netzdarstellungsmethoden

Volumenintegrationsmethoden[24]

In den letzten Jahren hat die Volumenvisualisierung einen immer größeren Stellenwert eingenommen. Das liegt zum einen an der fortschreitenden Technologie, die in verschiedenen Gebieten die Aufzeichnung von Volumendaten ermöglicht, wie zum Beispiel in der Medizin (Computertomographie, Kernspintomographie etc.), der Strömungsmechanik oder der Akustik-simulation im Fahrzeugbau. Zum anderen hat sich die Graphik-Hardware dahingehend weiterentwickelt, dass 3D-Textur-Hardware als Basis für effiziente Volumenvisualisierung wohl bald auch zur Standardfunktionalität von PC-Graphikkarten gehört.

Den wachsenden Einsatzfeldern wurden die diversen Szenengraph-APIs bisher nicht gerecht, da sie lediglich flächige Daten in Form von polygonalen Netzen oder parametrischen Flächen repräsentierten (vgl. Abschnitt Triangulations- und Netzdarstellungsmethoden). Die fehlende Unterstützung durch Programmierbibliotheken erschwerte die Realisierung von Anwendungen zur Volumenvisualisierung, insbesondere da interaktive Volumenvisualisierung, die für eine effiziente Analyse der Volumendaten unbedingt erforderlich ist, in hohem Maße von der Graphikhardware Gebrauch machen muss. Dies erfordert für den Programmierer sehr detaillierte Kenntnisse der Zielplattform und führt zu erheblichen Problemen bei der Portierung solcher Anwendungen. Die bisherigen Methoden waren sehr beschränkt in der Anzahl der Dimensionen. In vielen Anwendungen, speziell im Finanzbereich, treffen wir auf Volumendaten, die auf drei, oder falls die Zeit eine Rolle spielt, auf vier Dimensionen aufsetzen. Falls die Daten skalar oder Vektoren mit geringer Dimension sind, können Volumenintegrationsmethoden sehr effizient sein. Eine Eigenschaft dieser Methoden ist, dass sie gleichzeitig auch die Wiedergabe der Daten in der Form eines Bildes beinhalten. Abbildung 36 zeigt ein Bild eines Volumen-Renderers.

[24] Volume Rendering

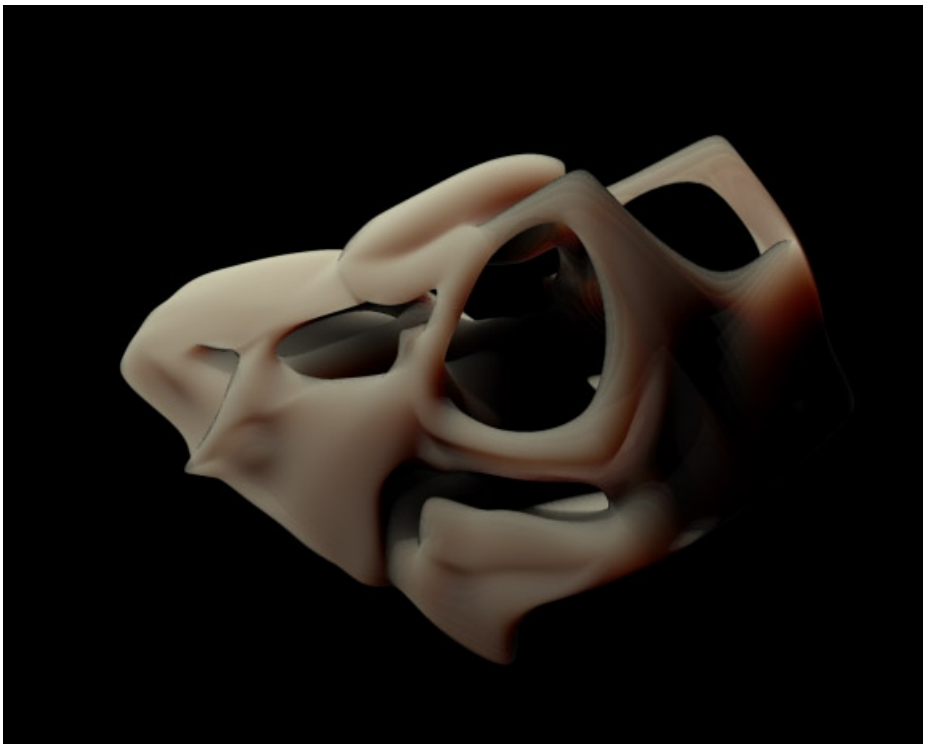


Abbildung 36: Beispiel Volumenintegrationsmethoden [59]

Vorteile	Nachteile
natürliche visuelle Metapher für 3D Datensätze	rechenaufwendig
Einblick in die interne Struktur von 3D Datensätzen	sensibel auf Änderungen der Render-Parameter
	Interpretation eines einfachen Bildes kompliziert

Tabelle 7: Vor- und Nachteile von Volumenintegrationsmethoden

Isolinien und Isooberflächen

In vielen dreidimensionalen Datensätzen ist es effektiver, Isolinien oder Isooberflächen (Abbildung 37), anstelle der ganzen Volumina darzustellen. Isolinien und Isooberflächen heben Werte hervor, die eine Eigenschaft oder einen Wert gemeinsam haben. Auch der moderate Rechenaufwand spricht für die Isolinien/-oberflächen. Aufgrund der immensen Bedeutung von Isooberflächen-Methoden für die Visualisierung wurden in diesem Bereich über die Jahre sehr viele Algorithmen gefunden.

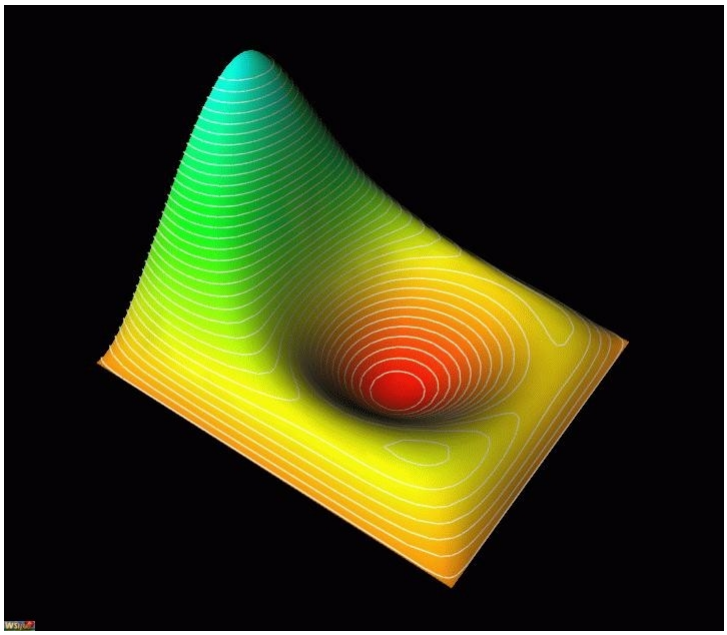


Abbildung 37: Beispiel für Isolinien

Vorteile	Nachteile
kann nicht direkt offensichtliche Eigenschaften der Daten hervorheben	nur für skalare Felder brauchbar
interaktive Analyse durch effiziente Algorithmen möglich	sensibel auf Datenfehler
produziert geometrische Primitiven welche Grafikbeschleuniger rendern	Topologie von Isooberflächen sehr komplex

Tabelle 8: Vor- und Nachteile von Isolinien und Isooberflächen

Stromlinien und Partikelverfolgung

Falls die Ausgangsdaten vektorbasiert (z.B. Geschwindigkeitsfelder) sind, können Stromlinien oder Partikelverfolgungsmethoden große Dienste leisten. Oft kann die direkte Visualisierung durch kleine Pfeile überladen wirken. Falls die Vektorfelder stationär sind, kann die animierte Darstellung von Partikeln weiterhelfen. Stromlinien- und Partikelverfolgungsmethoden können sich als sehr effektiv bei der Visualisierung und Analyse von großen Vektorfeldern erweisen. Abbildung 38 zeigt eine Strömungssimulation mit Stromlinien.

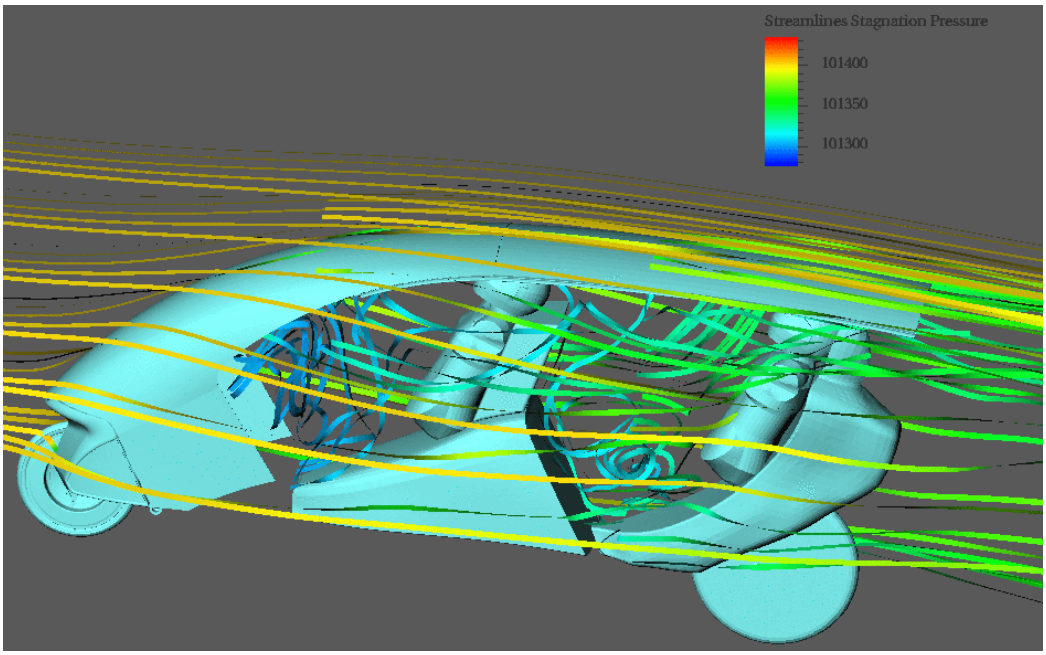


Abbildung 38: Beispiel für Stromlinien

Vorteile	Nachteile
eigenschaftenbasierte Methode, um tieferen Einblick in Vektordaten zu erhalten	benötigt tieferes mathematisches Verständnis
effiziente Reduktion von visueller Komplexität	keine intuitive visuelle Metapher
	sensibel auf Parameterveränderungen

Tabelle 9: Vor- und Nachteile von Stromlinien und Partikelverfolgung

Vektorfeldtopologien

Für visuelles Data Mining und die Analyse von großen Vektorfeldern (Abbildung 39) ist es oft erstrebenswert, die Topologie des Feldes aufzuzeigen, also kritische Punkte oder Linien und ihre Verbindungen hervorzuheben. Der Einfachheit halber können kritische Punkte/Linien als die Zentren von Strudeln innerhalb des Feldes verstanden werden. Die Berechnung dieser Punkte stellt aber mathematisch immer noch eine große Herausforderung dar und braucht ein großes Wissen über die differentiale Geometrie der Vektorfelder. Bekannte Methoden umfassen die Analyse von Eigenvektoren und Eigenwerten und werden immer noch weiter erforscht.

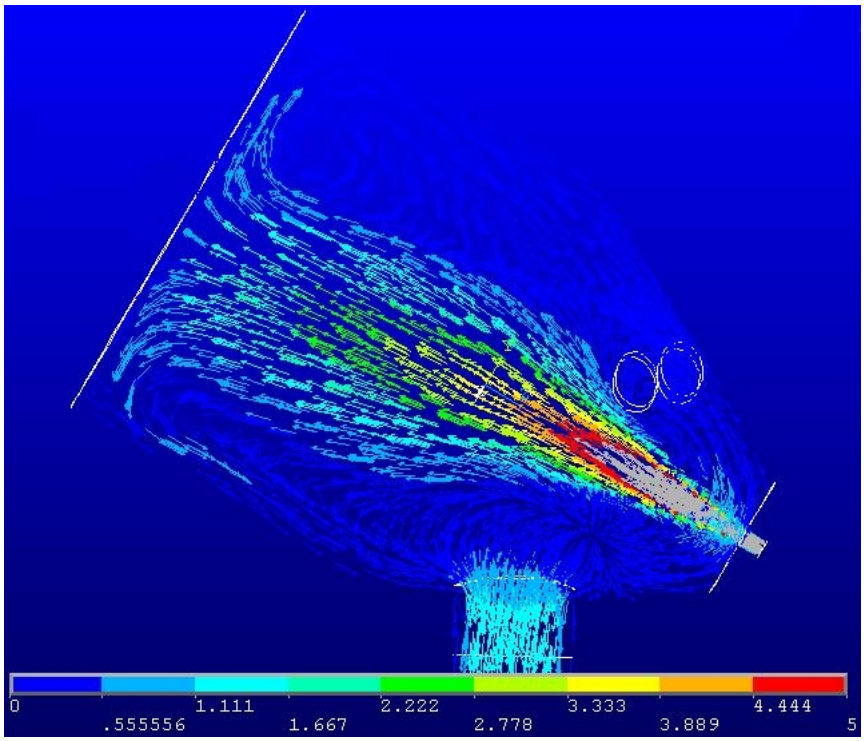


Abbildung 39: Beispiel für Vektorfeldtopologien

Vorteile	Nachteile
eigenschaftenbasierte Methode um tieferen Einblick in Vektordaten zu erhalten	benötigt tieferes mathematisches Verständnis
effiziente Reduktion von visueller Komplexität	keine intuitive visuelle Metapher
	sensibel auf Parameterveränderungen

Tabelle 10: Vor- und Nachteile von Vektorfeldtopologien

2.7.4 Augmented Reality (Mixed Reality)

Im Gegensatz zur Virtuellen Realität (VR) werden in Augmented Reality (AR) Umgebungen reale (physische) und virtuelle Objekte vermischt. Hierbei gibt es zwei Varianten: Bei der ersten Variante wird ein virtuelles Objekt in die reale Welt projiziert. Beispiel hierfür ist die Projektion von Handlungsanweisungen auf Maschinen im Servicebereich. Variante zwei „virtualisiert" die Umgebung mittels Kameras und mischt sie mit virtuellen Objekten. Man spricht in beiden Fällen von einer „Augmentierung" der realen Szene mit rechnergenerierten Objekten. Die zentrale Herausforderung der Augmentierten Realität ist nicht, wie bei der Virtuellen Realität, die performante wirklichkeitsgetreue Darstellung der Szene in Echtzeit mit einer möglichst hohen

Qualität, sondern das Erkennen der gegenwärtigen Position des Betrachters und seiner Blickrichtung bzw. die Erkennung und Augmentierung der Objekte in seinem Blickfeld in Echtzeit. Nur so ist der Computer in der Lage, die erkannte Szene mit virtuellen Objekten zu überlagern. Grundsätzlich bestehen alle Augmented-Reality-Systeme aus zwei Einheiten: Einem Darstellungssystem (Bildgenerator) sowie einem Tracking System, das die Position und Blickrichtung des Benutzers erfasst, um virtuelle Objekte richtig in die reale Umgebung einbetten zu können. [60] Die Anwendungsbereiche von AR sind vielfältig. In der Produktentstehung können z.B. physische Prototypen (etwa bei der Crashsimulation) mit virtuellen Berechnungsergebnissen aus Finite-Elemente-Verfahren überlagert werden. In der Produktion sind Hilfestellungen bei der Montage komplexer Zusammenbauten denkbar und nicht zuletzt können Servicetechniker in einer späteren Phase des Produktlebenslaufs bei Reparaturen und Instandsetzung unterstützt werden. Stand der Technik bei AR-Anwendungen ist die Verwendung eines HMDs (vgl. 2.4.1.2), bei dem das reale Gesichtsfeld mit den virtuellen Informationen angereichter wird, doch viele Systeme entsprechen noch nicht den Anforderungen eines industriellen Umfelds. Beispielsweise können heutige HMDs lediglich 30% des realen Gesichtsfeldes abbilden, was bei vielen Anwendern zu einer Desorientierung führt. Auch im Hinblick auf ergonomische Gesichtspunkte (Tragekomfort, Größe, Gewicht, etc.) ist noch viel Bedarf an Verbesserung.

ARVIKA

Größtes internationales Projekt im Rahmen von AR ist das BMBF Leitprojekt ARVIKA[25]. Zielsetzung von ARVIKA ist die Erforschung von Augmented-Reality-Technologien (AR) zur Unterstützung von Arbeitsprozessen in Entwicklung, Produktion und Service für komplexe technische Produkte und Anlagen.

Die Themenschwerpunkte im Rahmen des ARVIKA Projektes zielen außer auf die Grundlagenforschung vor allem auf die Erprobung von AR-Systemen in der industriellen Entwicklung, der Produktion und Fertigung sowie im Service ab: [60]

- **Unfallsimulation**: Bei der Unfallsimulation werden die zuvor per Computersimulation vorhergesagten Verformungen mit den realen Schäden am Crash-Fahrzeug abgeglichen. Der Entwicklungsingenieur kann dabei sofort entscheiden, ob die Ergebnisse der Simulation mit dem realen Verhalten übereinstimmen.

[25] Augmented Reality for Development, Production and Services

- **Fabrikplanung**: Die virtuelle Fabrikplanung hat zur Zielsetzung, das Risiko für Fehlplanungen und die damit verbundenen Kosten zu minimieren. Der Aufbau eines kompletten digitalen Modells ist jedoch in der Praxis häufig mangels aktueller Dokumentationen zur Layoutplanung schwierig. AR ermöglicht es dem Anwender stattdessen, direkt in der realen Welt zu planen.

- **Designstudien**: Bei Verwendung von AR in Designstudien werden mehrere Varianten und Designdetails eines Themas auf einen physisch vorhandenen Prototyp projiziert.

2.7.5 Leitprojekt integrierte virtuelle Produktentstehung (iViP)

Das Projekt iViP - integrierte virtuelle Produktentstehung – wurde 1989 als Leitprojekt mit 53 Partnern aus den Bereichen Automobilbau, Schienenfahrzeugbau und Maschinenbau initiiert und vom BMBF gefördert [61]. Ziel von iViP ist die über alle Phasen durchgängig, digitale Produktentstehung über Unternehmensgrenzen und -standorte hinweg als wettbewerbssteigernder Faktor für die Zukunft. Abbildung 40 zeigt die iViP Systemarchitektur. Kern des Konzeptes ist der iViP-Client und der „digitale Master". Der iViP-Client ist eine Integrationsplattform für die verschiedenen iViP-Werkzeuge, die sich dort als Plugin anmelden. Um sich direkt zu integrieren, müssen sie einer Spezifikation, ähnlich den Java-Beans, den iViP-Beans genügen.

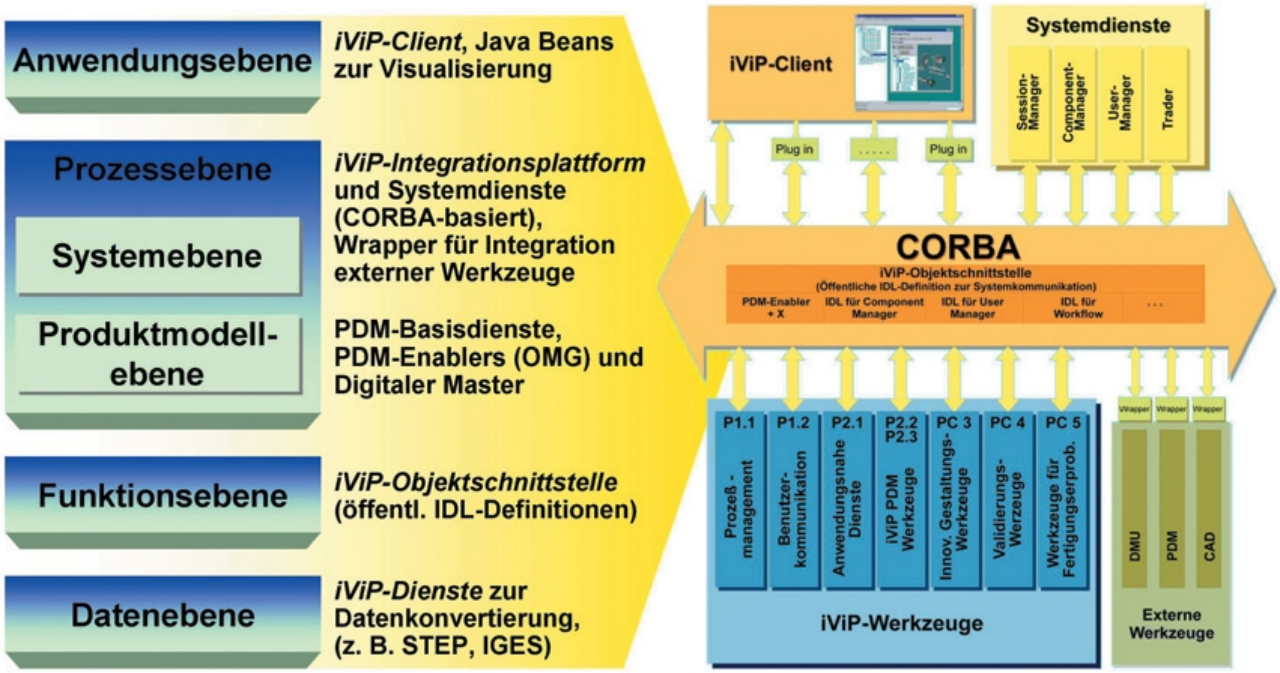


Abbildung 40: iViP Systemarchitektur (Quelle: [61])

Die Kommunikation der iViP-Komponenten basiert auf CORBA als Middleware über die Methodenschnittstelle der PDM-Enabler-Spezifikation. Externe Werkzeuge aus dem Produktentwicklungsprozess, wie CAD- oder DMU-Applikationen, werden über Wrapper-Module an den Kommunikations-Datenbus angebunden (vgl. Abbildung 41). Um einen Zugriff auf CAD-interne Daten zu ermöglichen, sind in der Datenebene Dienste zur Datenkonvertierung in STEP oder IGES aus den jeweiligen CAD-Systemen vorgesehen.

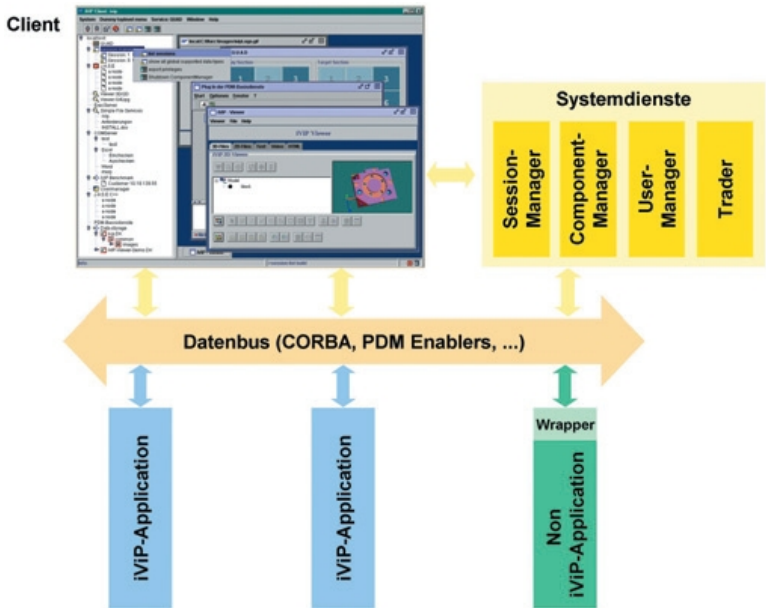


Abbildung 41: Kommunikation zwischen iViP Komponenten (Quelle: [61])

Im Rahmen von iViP wurde zwar die Einsetzbarkeit der CAD-Services-Spezifikation für den Austausch von CAD-Daten diskutiert, letztlich wurde jedoch ein Konstrukt aus den PDM-Enabler und einer TCL-basierten Schnittstelle benutzt, um eine Kopplung mit einem CAD-System zu erreichen. Abbildung 42 zeigt exemplarisch die Architektur der Kopplung des CAD-Systems Pro/ENGINEER an die iViP-Plattform. Die Nachteile dieser Vorgehensweise liegen auf der Hand: die PDM-Enabler Schnittstelle spezifiziert eine Methodenschnittstelle, um verschiedene Systemkomponenten auf „Dokumenten-Ebene“ zu verbinden. Entitäten, die sich in den Dokumenten befinden, sind durch diesen Ansatz nicht erreichbar.

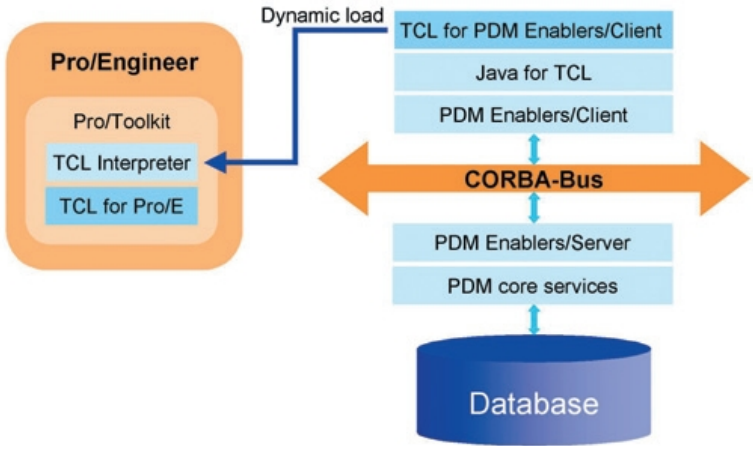


Abbildung 42: Kopplung von Pro/ENGINEER an die iViP Systemarchitektur (Quelle: [61],[62])

„*Der verbindliche Informationsträger, der sämtliche für die Produktentstehung und alle Folgephasen relevanten Daten enthalten soll, wird in iViP als Digitaler Master bezeichnet.*“ (Zitat [61]). Grundlage für die PDM-Enabler war das Konzept des digitalen Masters aus iViP. Jedes PDM-System könnte sich so transparent an die iViP-Plattform anbinden lassen, würde es nicht eines der Hauptmerkmale von PDM-Systemen, die kundenspezifischen Erweiterungen, geben. Sie verhindern eine allgemeine Kopplung von PDM-Systemen über die PDM-Enabler. In iViP wurde diesem Umstand durch eine spezielle Systemkomponente, dem semantischen Mapping, Rechnung getragen. Der Abbildungsprozess (Mapping), der die Daten in das Modell des Austauschformats überträgt, ist dabei hochgradig vom internen Datenmodell des jeweiligen PDM-Systems abhängig. Der digitale Master eines Produktes besteht also, wie bei einem PDM-System, aus einer Ansammlung von Dokumenten aus den am Produktenstehungsprozess beteiligten Systemen.

Ein Datenmodell, welches ein Produkt entlang seines Lebenslaufs beschreibt, muss jedoch zu wesentlich mehr in der Lage sein. Informationen, die *in* den Dokumenten der beteiligten Produktentwicklungswerkzeugen zu finden, sind müssen zugreifbar und - noch viel wichtiger - miteinander verknüpfbar sein. Zusammenfassend betrachtet ist der iViP-Client eine komponenten-basierte Reimplementierung eines PDM-Systems mit einem Plugin-Konzept für weitere Komponenten und der Möglichkeit, Dokumente durch digitale Signaturen zu authentifizieren und zu verschlüsseln, um so Zugriffsberechtigungen bei der verteilten Produktentwicklung zu regeln.

2.8 Middleware

Unter Middleware versteht man eine Menge von wenig spezialisierten ("general-purpose") Diensten, die zwischen der Systemplattform (Hardware + Betriebssystem) und den Anwendungen angesiedelt sind und deren Verteilung unterstützen. Vorhandene Systeme lassen sich mit geeigneter

Middleware ohne tief greifende Reorganisation an andere Systeme ankoppeln. Dies hat einen gewissen Investitionsschutz und die Erhaltung gewachsener Strukturen zu Folge. Weitere Vorteile sind:

- Entwickler wünschen sich Unterstützung für die Entwicklung verteilter Anwendungen. Middleware stellt aus diesem Blickwinkel "höhere" Dienste als reine Betriebssystem-plattformen zur Verfügung (z.B. einen RPC[26]-Mechanismus statt eines Send/Receive-Messaging).

- Entwickler von (verteilten) Anwendungen wollen ihre Produkte aus ökonomischen und softwaretechnischen Gründen so weit wie möglich von der Systemplattform entkoppeln.

- Da es in mehr und mehr Anwendungsfeldern Bedarf für verteilte Lösungen gibt (z.B. weil Daten nicht mehr im systemeigenen Dateisystem, sondern in einer Datenbank abgelegt werden sollen) wird Middleware zunehmend an Bedeutung gewinnen.

Beispiele für Middleware-Architekturen sind:

- "Klassische" (general-purpose) Middleware: OSF-DCE, **CORBA (OMG)**, ANSA, SOAP, Microsoft .NET

- "Moderne" Middleware: Framework-Architekturen, die neben einem API bereits Anwendungen und Tools enthalten (Beispiele: Lotus Notes, Microsoft Office, Link Works (DIGITAL)).

[26] RPC (engl. Remote Procedure Call) Aufruf von Methoden anstatt Kommunikation von Objekten durch Nachrichten

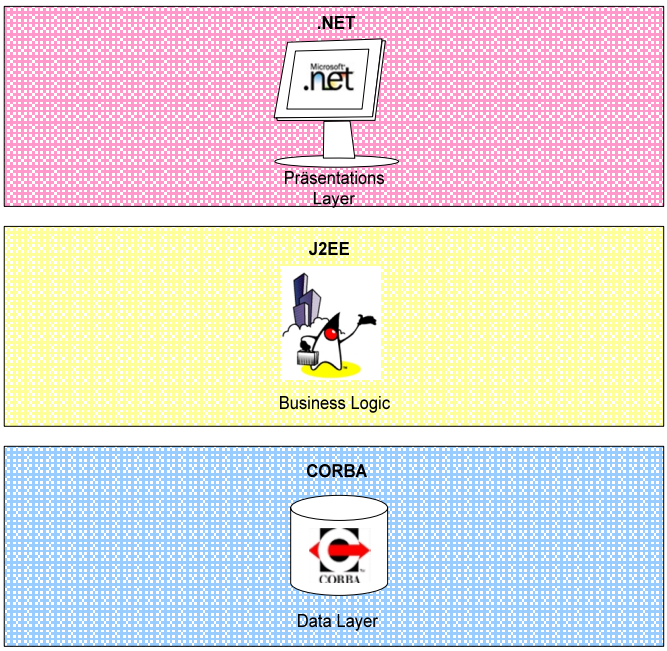


Abbildung 43: Gemische IT Umgebung (Middleware)

2.8.1 CORBA

Die Object Management Group (OMG) ist das weltweit größte Software-Konsortium. Ziel dieses Konsortiums ist, Referenzarchitekturen für verteilte, objektorientierte Systeme zu definieren und zu standardisieren. Die Schwerpunkte liegen hierbei in der Interoperabilität, Wiederverwendbarkeit und Portabilität von Softwarekomponenten. Im Jahre 1989 gegründet, hilft die OMG den Anwendern, ihre Probleme bei integrativen Aufgaben innerhalb der Unternehmen durch Open-Source Standards zu lösen, und stellt den Anbietern neutrale, übertragbare, kompatible und wieder verwendbare Spezifikationen basierend auf Model Driven Architecture (MDA) zur Verfügung. MDA definiert eine Methode, die die Spezifikationen der Systemfunktionalität von der Spezifikation der jeweiligen Implementierung auf einer spezifischen technischen Plattform trennt. Die OMG hat zahlreiche weit verbreitete Standards wie IDL (Interactive Data Language), CORBA (Common Object Request Broker Architecture), Realtime CORBA, GIOP/IIOP (General Inter-ORB Protocol / Internet Inter-ORB Protocol), UML (Unified Modelling Language), MOF (Meta Object Facility), XMI (XML Metadata Interchange) und CWM (Common Warehouse Metamodel), eingeführt.

Corba ist eine Middleware zur Realisierung von verteilten, objektorientierten Anwendungen. Unter Verwendung von Corba wird es einer Applikation ermöglicht, auf einem anderen Rechner liegende Objekte zu referenzieren und mit diesen über Methodenaufrufe zu kommunizieren. Dabei können die entfernt angesprochenen Objekte in einer anderen Programmiersprache implementiert sein, es muss nur ein auf die jeweilige Programmiersprache zugeschnittenes Corba-Produkt existieren.

Diese Programmiersprachenunabhängigkeit wird erreicht durch eine für Corba spezifizierte, deklarative Schnittstellenbeschreibungssprache, die IDL (Interface Definition Language). Über diese IDL werden die auf Server-Seite anzusprechenden Objekte mit ihren Methoden und den dazugehörenden Datenstrukturen definiert. Ein zu dem auf Server-Seite eingesetzten Corba-Produkt gehörender IDL-Compiler generiert aus der IDL-Definition Objekte in der Server-Programmiersprache (die Skeletons). Die Skeletons entpacken einen über das Corba-Protokoll IIOP hereinkommenden Aufruf (Demarshalling, Deserialisierung) und leiten diesen an die zugehörigen Objekt-Implementierungen (Servants) weiter. Auf Client-Seite werden mittels des IDL-Compilers des dort eingesetzten Corba-Produktes entsprechend Objekte in der Client-Programmiersprache erzeugt (die Stubs), welche das Aufbereiten entfernter Methoden-Aufrufe für die IIOP-Kommunikation vornehmen (Marshalling, Serialisierung). Der durch den Client getätigte entfernte Methodenaufruf muss nicht zwingend über statische Stubs erfolgen, er kann auch über einen vom Corba-Produkt angebotenen Mechanismus dynamisch zu Laufzeit zusammengestellt werden (Dynamic Invocation Interface).

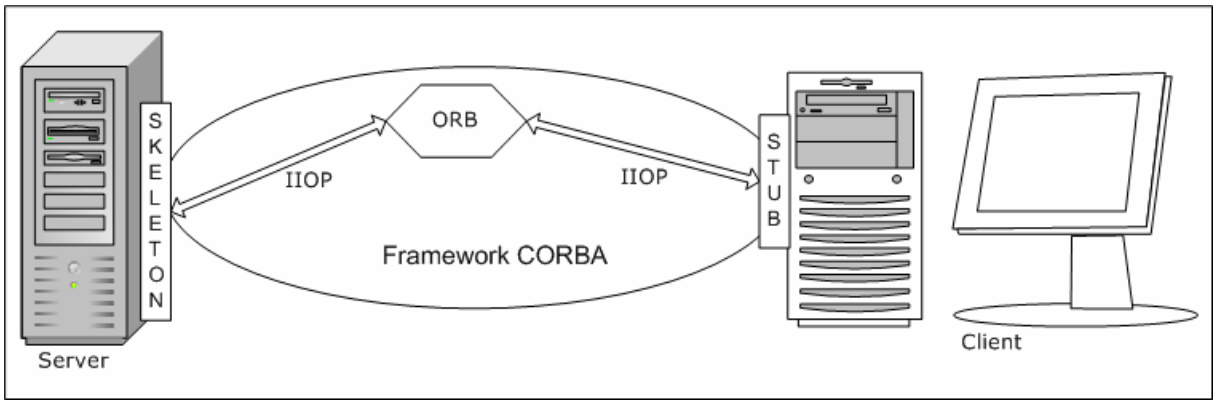


Abbildung 44: Kommunikation über CORBA

Bei der Definition von Methoden kann angegeben werden, ob sie synchron oder asynchron aufzurufen sind. Eine Methodendefinition ohne weitere Angabe bedeutet einen synchronen Aufruf, d.h. der Kontrollfluss kehrt erst wieder zum Client zurück, wenn die Methode serverseitig abgearbeitet und durchlaufen ist. Als One-way deklarierte Methoden ohne Antwort vom Server sind asynchron und blockieren den Kontrollfluss des Clients nicht. In Verbindung mit der Möglichkeit, Callbacks[27] zu nutzen, stehen einem Entwickler alle denkbaren Kommunikationsarten zur Verfügung. Durch diese einheitliche Schnittstellenbeschreibung auf Basis der objektorientierten

[27] Vom Programmierer zu definierende Funktionen die aufgerufen werden wenn sie implementiert wurden. Standardmäßig sind sie als „leere" Funktion implementiert.

IDL bzw. deren Datentypen sowie der Verwendung des spezifizierten Übertragungsprotokolls IIOP[28] wird die betriebssystem- und programmiersprachenunabhängige Kommunikation zwischen verschiedenen Corba-Produkten ermöglicht.

Um die oben beschriebenen Methodenaufrufe durchführen zu können, muss ein Client eine Referenz auf das aufzurufende entfernte Objekt besitzen. Referenzen auf entfernte Objekte können in Corba über den Namingservice geholt werden oder vom Server als Resultat eines Methodenaufrufs geliefert werden (z.B. eine Factory-Methode). Der Namingservice ist ein Dienst, an dem Corba-Serverobjekte zu registrieren sind und damit für Clients referenzierbar werden.

Um dieses programmiersprachenunabhängige Übergeben von Referenzen auf entfernte Objekte zu ermöglichen, wird jedem Skeleton bzw. Stub-Objekt, inklusive der dazugehörenden Servants, eine Corba-interne, interoperable Objekt Referenz (IOR) zugeordnet. Für die korrekte Zuordnung zwischen Stubs und Skeletons für entfernte Methoden-Aufrufe, d.h. der Lokalisierung des aufzurufenden Objektes im Netz und die Vermittlung der zu übertragenden Daten an dieses, ist der Objekt Request Broker (ORB) zuständig. Der ORB nimmt die serialisierten Daten der Stubs entgegen und überträgt (vermittelt) diese dann über IIOP an den ORB, auf dem Server-Rechner, auf dem der Aufruf an das entsprechende Server-Objekt vermittelt wird. Dem Entwickler einer verteilten Anwendung bleibt dieser Kommunikationsmechanismus über den ORB inklusive der Details des Übertragungsprotokolls verborgen, er arbeitet nur auf Basis der definierten Schnittstelle, in Gestalt der generierten Stubs bzw. Skeletons. Als Bindeglied zwischen dem Server-Programm und dem ORB ist ein Objekt-Adapter vorhanden, der für die Erzeugung und Verwaltung der internen Objekt-Referenzen auf den einzelnen Servants zuständig ist. Um ein Server-Objekt beim Namingservice anzumelden, muss für dieses eine interoperable Objekt-Referenz vom Objekt Adapter geholt werden, welche dann zusammen mit dem Server-Objekt beim Namingservice registriert wird. Es sind zwei Objekt-Adapter für Corba spezifiziert, der ältere Basic Object-Adapter (BOA) und der später spezifizierte Portable-Object-Adapter (POA), welcher eine Migration von Server-Programmen auf ein anderes Corba-Produkt erleichtert.

Durch diesen Kommunikations-Mechanismus über den ORB, mit der Möglichkeit, entfernte Referenzen zu Erzeugen und zu übertragen, ist ein wichtiges Merkmal von verteilter objekt-orientierter Middleware erfüllt. So ist es z.B. möglich, Objektreferenzen über eine Factory-Methode zu liefern oder das Observer-Muster über einen Callback-Mechanismus zu realisieren, wodurch die starre Unterscheidung in Client und Server aufgehoben werden kann. Für diesen Callback-Mechanismus erzeugt der Beobachter auf Basis einer entsprechend definierten IDL ein

[28] Internet Inter-Orb Protokoll

Beobachter-Objekt und übergibt dem zu beobachtenden Server eine Referenz auf dieses Objekt über eine entsprechende Methode. Der beobachtete Server kann nun, immer wenn ein entsprechendes Ereignis auftritt, über die Referenz auf das Beobachter-Objekt den Client benachrichtigen (zu den Entwurfsmustern „Beobachter“ und „Fabrik“ siehe [63]).

Für die Definition der Schnittstelle über die IDL sind die für die Methodenparameter und Objektattribute benutzbaren Datentypen spezifiziert. Dazu gehören neben einfachen Datentypen wie Integer, Strings oder Enumerations auch zusammensetzbare Datentypen wie Structs, Unions oder Arrays. Zur Ausnahmebehandlung von Methodenaufrufen sind eigene Ausnahmetypen definierbar. Neben diesen Datentypen, die eine statische Typisierung zur Kompilierzeit erzwingen, gibt es in Corba auch die Möglichkeit einer dynamischen Typisierung über den Datentyp „Any“. Ist ein Parameter mit dem „Any-Type“ deklariert worden, kann dieser zur Laufzeit Werte jeden Datentyps annehmen. Das zur Speicherung und zum Austausch von Daten immer häufiger verwendete XML-Format wird von Corba nicht direkt unterstützt. Ein komplettes XML-Dokument kann aber natürlich als String-Parameter übertragen werden.

Um eine interoperable Kommunikation zwischen verschiedenen ORB-Herstellern zu gewährleisten, wurde für Corba das abstrakte General Inter-Orb Protokoll (GIOP) spezifiziert. Für dieses Protokoll ist definiert, wie die entfernten Methoden-Aufrufe inklusive der zugehörigen Datentypen auf ein binäres Format abzubilden sind. GIOP ist unidirektional verbindungsorientiert angelegt, d.h. Anfragen gehen immer vom Client aus, der Server antwortet nur. Zur Ermöglichung der oben erwähnten Callbacks muss auf Client-Seite also ebenfalls ein GIOP-Server laufen. Für die tatsächliche Übertragung wird das abstrakte GIOP auf ein vom Betriebssystem angebotenes Protokoll abgebildet. Die Abbildung von GIOP auf TCP/IP wird Internet Inter-Orb Protokoll (IIOP) genannt. IIOP ist der spezifizierte Standard für die Kommunikation zwischen den einzelnen ORBs. Der Entwickler einer verteilten Anwendung muss keinerlei Kenntnis über das Protokoll haben, er entwickelt ausschließlich auf Basis der mittels der IDL definierten Schnittstelle, weswegen an dieser Stelle keine Details des Protokolls besprochen werden.

Zusätzlich zu der IIOP-Kommunikation über den ORB bzw. der Schnittstellenbeschreibung durch die IDL sind für Corba noch zusätzliche Dienste spezifiziert. Diese Dienste unterstützen die Entwicklung von verteilten Anwendungen, stellen Lösungen für immer wiederkehrende Probleme bereit und erhöhen bei Benutzung dieser Services die Portabilität bzw. die Wiederverwendbarkeit einer Komponente. Hier sollen kurz die wichtigsten Services erwähnt werden, welche direkt die Kommunikation auf der Ebene der verteilten Objekte unterstützen (Objekt-Services):

- **Naming Service**: Über den oben schon erwähnten Naming-Service können Server-Objekte über einen registrierten Namen aufgefunden werden. Der Client wendet sich an diesen zentralen Dienst, bekommt eine Referenz auf das entfernte Server-Objekt und kommuniziert mit diesem ohne zu wissen, auf welchem Rechner sich dieses Server-Objekt befindet (Ortstransparenz).

- **Trading Service**: Der Trading Service dient ebenfalls dem Auffinden von Server-Objekten und bietet das Suchen nach Objekten an, die eine bestimmte Schnittstelle implementieren.

- **Event Service**: Der Event Service bietet eine spezifizierte Lösung für einen Ereignis-Mechanismus. Es können sich mehrere Clients bei einem Server für die Benachrichtigung über ein Ereignis registrieren. Die Benachrichtigung erfolgt über einen einzigen Event-Kanal, d.h. es werden immer alle Clients benachrichtigt.

- **Notification Service**: Da dem Event Service einige wichtige Merkmale fehlen, wurde später der Notification Service spezifiziert. Beim Notification Service können die Benachrichtigungen gefiltert werden, d.h. Nachrichten werden nur an bestimmte Clients verschickt. Zusätzlich kann auch die Güte der Benachrichtigung eingestellt werden (Quality of Service). Güte-Eigenschaften können z.B. eine unterschiedlich priorisierte, garantierte oder eine zeitgesteuerte Benachrichtigung sein.

- **Transaction Service**: Der Transaction Service bietet Unterstützung für verteilte, objektorientierte Transaktionsverwaltung, d.h. mehrere entfernte Methoden-Aufrufe sollen als Einheit entweder alle ausgeführt werden oder gar nicht, falls ein Aufruf fehlerhaft ist.

- **Concurrency Control Service**: Zur Steuerung des Verhaltens bei mehreren gleichzeitig durchgeführten Methoden-Aufrufen an einem Server-Objekt ist der Concurrency Control Service spezifiziert worden.

- **Security Service**: Zur Gewährleistung der Sicherheit einer verteilten Anwendung bietet der Security Service u. a. Mechanismen zur abgesicherten Kommunikation, Authentifikation und Autorisation.

Diese und eine Reihe von weiteren Services werden in Kombination mit den grundlegenden Corba-Spezifikationen (ORB, Objekt Adapter, IDL) als Object Management Architecture (OMA) bezeichnet.

2.8.2 SOAP

SOAP (Simple Object Access Protocol) ist ein Protokollstandard des W3C[29]. Darüber werden Applikationen webfähig, und Kommunikation zwischen verteilten Applikationen und Objekten wird ermöglicht und standardisiert. SOAP ist unabhängig von Betriebssystemen, Programmiersprachen und Objektmodellen, kann verschiedene Plattformen verbinden (z.B. .NET und Java) und ist leichter zu implementieren als CORBA und DCOM.

SOAP stellt einen einfachen und durchsichtigen Mechanismus zum Austausch von strukturierter und typisierter Information zwischen Rechnern in einer verteilten Umgebung zur Verfügung. Dabei soll durch SOAP kein Programmiermodell oder keine implementationsspezifische Semantik vorgegeben werden, sondern ein unkomplizierter Mechanismus, um die interne Semantik einer Anwendung zu beschreiben. Dazu werden ein modulares Paketmodell sowie Mechanismen zum Verschlüsseln von Daten innerhalb von Modulen zur Verfügung gestellt. Dadurch kann SOAP in einem weiten Einsatzfeld gebraucht werden, von einfachen Nachrichtensystemen bis zu Remote Procedure Calls.

SOAP ist ein XML-basiertes Protokoll, das aus drei Teilen besteht:

- der Spezifikation für einen Umschlag (Envelope), der ein Regelwerk definiert, welches beschreibt, was in einer Nachricht enthalten ist, von wem es wie verarbeitet werden soll und ob einzelne Daten optional sind oder enthalten sein müssen

- ein Satz von Kodier- und Ordnungsregeln (Serialization), welcher Instanzen von anwendungsspezifischen Datentypen beschreibt

- eine Konvention, um Remote Procedure Calls und eventuelle Antworten auf diese zu repräsentieren

SOAP-Nachrichten sind im Grunde genommen Übertragungen vom Sender zum Empfänger, die miteinander kombiniert werden können, um Frage-Antwort-Situationen zu modellieren. Eine SOAP-Implementierung kann auf die Charakteristiken des zur Verfügung stehenden Netzwerkes abgestimmt werden, um diese optimal auszunutzen. So kann beispielsweise in Verbindung mit dem HTTP-Protokoll eine Antwort auf eine SOAP- Nachricht als HTTP-Response gesendet und somit die gleiche Verbindung wie die ursprüngliche SOAP-Nachricht benutzt werden. Ganz gleich, an welches Protokoll zur Nachrichtenübermittlung SOAP gebunden ist, verfolgen die Nachrichten

[29] http://www.w3.org/TR/SOAP (W3C == World Wide Web Consortium)

einen sogenannten "Message Path", sodass jede Nachricht an den auf dem Weg liegenden Knotenpunkten verarbeitet werden kann, bevor sie beim eigentlichen Ziel ankommt.

Die Serialisierung und Kodierung von Daten im SOAP basiert auf einem einfachen Typsystem (int, double, …), welches eine Verallgemeinerung der Gemeinsamkeiten, der in bestehenden Programmiersprachen und Datenbanken vorhandenen Typsysteme darstellt. Ein Datentyp gehört entweder den einfachen Datentypen an oder ist ein aus mehreren einfachen Datentypen konstruierter zusammengesetzter Datentyp.

Die Serialisierung von Daten erfolgt im SOAP in zwei Schritten. Zunächst muss aus dem der Anwendung zugrunde liegenden Datenmodell ein XML-Schema konstruiert werden. Danach wird aus diesem Schema und dem aktuellen Datenbestand die eigentliche XML- Datei erzeugt. Bei der Deserialisierung wird dann aus der XML-Instanz unter Kenntnis des zugrunde liegenden Schemas eine Kopie des Datenbestands erstellt. Alle Werte werden durch den Inhalt von Elementen repräsentiert. Dabei muss für jedes nicht-leere Element der Datentyp des enthaltenen Werts auf eine der folgenden Arten angegeben werden:

1. Das Element besitzt das xsi:type-Attribut, durch welches der Typ definiert wird.

2. Das Element selbst befindet sich in einem Array, dessen Typ durch das SOAP-ENC:arrayType-Attribut definiert wurde.

3. Der Name des Elements enthält einen Hinweis auf den Datentyp, der dann aus der zugrunde liegenden Schemadatei entnommen werden kann.

Ein einfacher Wert wird als Charakter Data, also ohne irgendwelche Subelemente, dargestellt. Er muss von einem in der XML-Schema-Spezifikation aufgelisteten Typ sein. Dagegen besteht ein zusammengesetzter Datentyp aus einer Sequenz von Elementen, wobei jeder Teilwert von einem eigenen Unterelement repräsentiert wird. Die SOAP-Spezifikation unterscheidet weiterhin zwischen unabhängigen und eingebetteten Werten. Unabhängige Werte stehen dabei in der obersten Ebene einer Serialisierung, sind also etwa direkte Kindelemente des Elements `<SOAP-ENV:Body>`. Alle anderen Elemente werden als eingebettete Elemente betrachtet.

2.8.3 CORBA vs. SOAP

Das Definieren der Server-Schnittstelle erfolgt bei Corba über die C++ ähnelnde IDL, welche vom Entwickler erlernt werden muss. Middleware für Web-Services (SOAP) bietet üblicherweise den Mechanismus an, eine Schnittstelle direkt in der Web Service implementierenden Programmier-

sprache zu definieren. Zum Entwickeln eines Services ist eine Beschreibung über eine WSDL[30] also nicht zwingend nötig. Diese kann aus der Web-Service-Implementierung generiert werden, so dass ein Entwickler im Idealfall nicht mit WSDL in Berührung kommt, was gegenüber Corba einen geringeren Entwicklungsaufwand bedeutet. Allerdings ist momentan (Herbst 2004) die Interoperabilität zwischen Web-Service-Middleware verschiedener Hersteller nicht immer gewährleistet. Ein Entwickler muss auf eine interoperable Schnittstelle achten bzw. zum Schreiben eines Clients ein Interoperabilitätsproblem verstehen und sich deshalb mit SOAP und den sehr umfangreichen WSDL-Beschreibungen auskennen. Solange dies der Fall ist, ist das Abstraktionsniveau bei Web-Services wesentlich niedriger als bei Corba. Bei Corba sind keinerlei Protokollkenntnisse nötig und die IDL wesentlich einfacher zu verstehen als WSDL.

Ein weiterer Unterschied zwischen einer IDL- und einer WSDL-Beschreibung besteht darin, dass IDL eine reine abstrakte Schnittstellenbeschreibung darstellt, während bei WSDL zusätzlich ein tatsächlich existierender Web-Service inklusive der URL eingetragen werden kann und somit ein Auffinden über einen Namingservice umgangen werden kann.

Als Vorteil von SOAP wird häufig angebracht, dass die über die Leitung geschickten Daten für Menschen lesbar sind und somit das Debugging erleichtert wird. Hierzu ist anzumerken das der Bedarf für das Suchen von Fehlern auf Protokollebene erst dann entsteht, wenn die entsprechende Middleware keine vollständig abstrahierenden APIs und Interoperabilität mit anderen Produkten anbieten kann. In Corba wird die Frage nach Protokoll-Debugging gar nicht gestellt, weil der Bedarf gar nicht existiert.

Die Spezifikationen von Corba und SOAP bzw. WSDL haben eine plattform- und programmiersprachenunabhängigkeit zum Ziel, deren Umsetzung nur durch Interoperabilität zwischen Middleware verschiedener Hersteller zu erreichen ist. Bei Corba gab es anfangs Interoperabilitätsprobleme, die mittlerweile behoben sind. Diese entstanden durch fehlende Spezifikationen u. a. für das Transportprotokoll. Bei der Web Service-Technologie liegt das Augenmerk hauptsächlich auf Protokollebene (SOAP) und der Schnittstellenbeschreibung über WSDL. Die SOAP-Spezifikation weist einige Ungenauigkeiten auf, die den Middleware Herstellern einen gewissen Interpretations-Spielraum bei der Implementierung ließ. Anfänglich dadurch entstandene Interoperabilitätsprobleme scheinen mittlerweile behoben. Ein Problem für die Interoperabilität stellt die Flexibilität von SOAP in Bezug auf die Kommunikationsart und die Datentypen dar. So ist die Voreinstellung mancher Web-Service-Middleware die Benutzung des Document-Styles in Verbindung mit literaler Codierung (z.B. Microsofts .Net), während andere

[30] WSDL = Web-Services Description Language (ist das Äquivalent zu IDL bei CORBA)

Werkzeuge die RPC-Verwendung mit SOAP-Codierung voreingestellt haben (z.B. Apache Axis), was zu unterschiedlicher Serialisierung der Daten in ein SOAP-Dokument führt. Weiterhin führt die Verwendung von nicht in der SOAP-Codierung vorhandenen Datentypen in Verbindung mit RPC zu Problemen.

Verteilte Anwendungen erfordern häufig Mechanismen, um Transaktionen oder eine Sessionverwaltung zu ermöglichen. Für diese Mechanismen existieren bei SOAP keine eindeutigen Spezifikationen, weswegen Lösungen dafür herstellerabhängig und damit nicht interoperabel sind. Ein Objektmodell mit der Möglichkeit, entfernte Referenzen auf verschiedene Instanzen zu verwenden, ist ebenfalls nicht spezifiziert und müsste eigens nachgebildet werden. Diese zusätzlichen Dienste sind in Corba spezifiziert, wodurch Interoperabilität auch in diesem Bereich ermöglicht werden kann.

Ein Aufruf entfernter Methoden über einen RPC[31]-Mechanismus bedeutet zum einen Typsicherheit in Bezug auf die Parameter, zum anderen eine feste Kopplung zwischen Client und Server, da auf eine feste Schnittstelle hin programmiert wird. Die feste Kopplung kann durch Verwendung eines dynamisch typisierbaren Parameter (Any-Type) aufgehoben werden. Diese Möglichkeiten sind sowohl in Corba als auch bei Web Services vorhanden. Die Wahl der geeigneteren Middleware für die Verwendung von RPC ist im Kontext weiterer Vergleiche zu treffen (Performanz, Interoperabilität, Kommunikationsmodelle, Implementierungsaufwand). Eine noch losere Kopplung kann durch das Verschicken eines XML-Dokumentes über SOAP ohne Abbildung auf statisch deklarierte Parameter auf Server-Seite erreicht werden. Hierbei muss kein Typ-Cast eines Any-Typs erfolgen, der exakte Datentyp also gar nicht bekannt sein. Mittels geeigneter XML-Abfragesprachen (XPath, XQuery) können die benötigten Daten aus dem XML-Dokument extrahiert werden. Dadurch kann ein Server beispielsweise verschieden strukturierte Dokumente verarbeiten und muss nicht so häufig angepasst werden. Bei Corba kann natürlich Ähnliches durch Übertragen von XML-Dokumenten als String erreicht werden.

Der Begriff „lose Kopplung“ wird ebenfalls im Bereich von MoM[32]-Produkten verwendet, welche Nachrichtenfilterung und Quality of Service bieten. Insbesondere mit der asynchronen Nutzung dieser Dienste kann eine lose Kopplung von verteilten Systemen erreicht werden. Bei Corba ist diese Möglichkeit über den Notification Service spezifiziert worden. Im Bereich von Web Services gibt es keine derartige Spezifikation, weswegen nur herstellerabhängige Lösungen existieren.

---

[31] RPC (engl. Remote Procedure Call)

[32] MoM (engl. message oriented middleware)

In verteilten Anwendungen auf Basis von Corba können, aufgrund der Möglichkeit, entfernte Objektreferenzen inklusive Callbacks zu nutzen, alle denkbaren Kommunikationsmodelle implementiert werden. Eine strikte Einteilung in Client und Server muss nicht vorgenommen werden. Allerdings werden diese Möglichkeiten durch Firewalls eingeschränkt.

Die Möglichkeit, vom Server initiierte Aufrufe beim Client vorzunehmen, ist zwar in der WSDL-Spezifikation vorgesehen, bei Web-Services steht aber die Verwendung von HTTP als SOAP-Übertragungsprotokoll im Mittelpunkt, weshalb Callbacks ohne HTTP-Server auf Client-Seite nicht möglich sind.

Für die Web-Service-Technologie ist Http als Übertragungsprotokoll favorisiert, da dieses Protokoll nicht von Firewalls blockiert wird. So sollen Softwarekomponenten problemlos weltweit kommunizieren können und ähnliches Potential eröffnen, wie es HTML in Verbindung mit Http für menschlich lesbare Informationen erbracht hat. Diese Möglichkeit wurde mit Corba nicht erreicht, da hier die Probleme mit Firewalls relativ groß sind. An dieser Stelle muss man natürlich nach dem Sinn von Firewalls fragen, die doch unerwünschte Kommunikation verhindern soll. Ein Einbetten von Protokollen in HTTP verschleiert den Unterschied zwischen sicherheitskritischen Kommunikationsdaten und ungefährlichen Informationen. Firewalls könnten diese eingebetteten Protokolle erkennen und herausfiltern. Diese Aufgabe erscheint aber ähnlich schwierig, wie ein standardisiertes Verhalten von Firewalls in Bezug auf das Corba Protokoll IIOP zu erreichen. Vielleicht sollte für die Kommunikation über SOAP/Http ein anderer Standardport als der Port 80 festgelegt werden, um eine strikte Trennung herbeizuführen.

Die Performanz bei der Kommunikation in verteilten Systemen wird durch die zu übertragende Datenmenge und die Geschwindigkeit der Serialisierung/Deserialisierung der Daten beeinflusst. Die Kommunikation über SOAP ist wesentlich unperformanter als die über Corba, da die zu übertragende Datenmenge aufgrund der vielen XML-Tags relativ groß ist. Zusätzlich ist das Parsen der SOAP-Dokumente beim Deserialisieren und das Zusammenstellen derselben beim Serialisieren relativ aufwendig. Ein anderer Performanz-Aspekt ergibt sich in Bezug auf die Anforderungen einer verteilten Anwendung und das verwendete Kommunikationsmodell. Soll z.B. eine Client-Komponente ständig über Zustandsänderungen des Servers benachrichtigt werden, muss diese bei der Verwendung von HTTP als Übertragungsprotokoll ständig beim Server nachfragen (pollen[33]). Bei Corba kann ein Beobachtermuster über Callbacks implementiert werden, so dass der Client bei Zustandsänderungen sofort benachrichtigt werden kann. Das ständige Pollen bei der Verwendung

[33] to poll (engl. Abfragen)

von HTTP hat zur Folge, dass zum einen die Netzwerklast erhöht wird und zum anderen der Client Zustandsänderungen evtl. nicht sofort bemerkt (je nach „Poll-Frequenz“).

2.9 OMG CAD Services

Der Standard CAD-Services wurde von dem Unternehmen EADS Matra Datavision „OpenCASCADE S.A.“ bei der OMG initiiert. Der erste Entwurf der CAD-Services (Version 1.0) wurde im Juni 2001 von den Unternehmen NASA Glenn Research Center, IBM, SDRC, Unigraphics Solutions und OpenCASCADE der Öffentlichkeit vorgestellt. Die CAD Services Spezifikation kam daraufhin erfolgreich durch den OMG FTF (Finalization Task Force) Prozess, und in kurzer Zeit wurde die endgültige Version der Spezifikation der CAD-Services R.1.1 Revision Task Force (RTF) von der OMG MfgDTF (Manufacturing Domain Task Force) zugelassen.

Die CAD-Services stellen einen Interface Standard für Mechanical Computer Aided Design (CAD) dar, der die Kompatibilität von CAD, Computer Aided Manufacturing (CAM) und Computer Aided Engineering (CAE) Tools ermöglicht. Jedes CAD System, für das dieser Adapter implementiert ist, lässt sich über die standardisierte Methodenschnittstelle in der gleichen Art und Weise ansteuern. Informationsverluste wie bei der Übertragung von Geometrie via Neutralfile-formate sind hierbei ausgeschlossen. Es handelt sich um eine direkte Online-Verbindung mit einem CAD-System. Das Ziel ist, den Benutzern eine nahtlose Integration von Design- und Engineering-systemen über eine breite Auswahl von CAD/CAM und CAE Anwendungen durch standardisierte CORBA-Interfaces anzubieten. Diese Interfaces ermöglichen ein verteiltes Produktdesign-Umfeld, das eine Vielfalt von CAx-Systemen enthält. Die aktuelle Version der CAD-Services-Spezifikation ist 1.2 und ist im Juni 2004 veröffentlicht worden. Das Modul besteht aus insgesamt acht Spezifikationsdateien im IDL-Format. Mit Hilfe einer der verfügbaren Implementierung von CORBA, wie ORBexpress, omniORB, Fnorb, ORBacus etc., können diese IDL-Dateien in Hochsprachen wie C++ oder Java übersetzt werden.

Eine ausführliche Betrachtung der CAD Services, die einen wichtigen Baustein für den Zugriff auf die geometrischen Produktdaten darstellen, befindet sich in Anhang E.

2.10 Anforderungs-Entwicklungssystem (SMC)

Ziel des Transferprojekts F6 „Entwicklung eines Anforderungsmodellierers“ [64] war es, die im SFB 346 erarbeiteten Ergebnisse im Bereich der konstruktionsphasenübergreifenden An-forderungsentwicklung in die Praxis zu transferieren. Dazu wurde ein prototypisches Werkzeug zur

Unterstützung der Anforderungsmodellierung entwickelt (SMC[34]) und mit dem kommerziellen System RODON® des Projektpartners R.O.S.E Informatik gekoppelt [65]. RODON ist ein modellbasiertes Werkzeug zur Untersuchung des funktionalen Verhaltens technischer Produkte auf Basis physikalischer Abhängigkeiten zwischen den modellierten Systemkomponenten. Systeme werden in RODON hierarchisch aus Einzelkomponenten und Subsystemen aufgebaut, die miteinander über Eingangs- bzw. Ausgangsschnittstellen verbunden sind. Über geeignete Simulationsmethoden kann das Verhalten des Gesamtsystems vorausgesagt werden.

Der Prototyp erlaubt es, in einem Produktstruktureditor komplexe Produktstrukturen flexibel aufzubauen. Dabei können beliebige Sichten auf ein und dasselbe Produkt parallel erstellt und gepflegt werden.

Die Sichten auf die Produktstruktur dienen als Grundlage für die Anforderungsmodellierung mit SMC [66]. Anforderungen werden dem Benutzer in vordefinierten, hierarchisch strukturierten Anforderungsbibliotheken zur Auswahl angeboten. Aus diesen Anforderungsbibliotheken sucht sich der Benutzer relevante Anforderungen aus und ordnet sie der Produktstruktur des zu entwickelnden Produkts zu. Die Werte von Anforderungen können dabei vorgeprägt sein, beispielsweise wenn gesetzliche Vorschriften für bestimmte Produkteigenschaften existieren.

Die wissensbasierte Komponente von SMC überwacht auf der Grundlage von Wissen über Beziehungen zwischen den Anforderungen den Prozess der Anforderungsmodellierung. Sie gibt dem Benutzer Hinweise über zu beachtende Abhängigkeiten zwischen den modellierten Anforderungen und über fehlende, noch zu spezifizierende Anforderungen. [67]

Die Schnittstelle zu RODON ermöglicht es, Produktstrukturen aus SMC in RODON zu exportieren. Dabei werden die Baugruppen und ihre Komponenten in Subsysteme und Einzelkomponenten des Systemmodells in RODON übersetzt. Aus den Anforderungen der Produktkomponenten generiert SMC Systemeingangs- bzw. –Ausgangsschnittstellen der entsprechenden Systemmodellkomponenten. Das vorausgeprägte Systemmodell kann daraufhin mit den Simulationsmöglichkeiten von RODON auf Konsistenz überprüft und weiter verfeinert werden. Durch die Importfunktionalität von SMC, durch die Systemmodelle aus RODON in SMC eingepflegt werden können, entsteht ein geschlossener Kreislauf zur wissensbasierten, anforderungsgesteuerten Entwicklung von Funktions- und Prinzipstrukturen.

[34] SMC (engl. Specification Modelling Component) Anferderungsmodellierer

2.11 XML (Extensible Markup Language)

Bevor der Begriff XML erklärt werden kann, soll auf die geschichtliche Entwicklung von Auszeichnungssprachen (markup languages) hin zur Notwendigkeit von XML eingegangen werden. Auszeichnungssprachen entwickelten sich bereits zu Zeiten des frühen Buchdruckes, als Autoren ihre Manuskripte mit Markierungen oder Auszeichnungen versahen, um den Setzern damit Anweisungen über die gewünschte Formatierung mitzuteilen. Diese Auszeichnungen bestehen hauptsächlich aus speziellen Zeichen, die sich vom darzustellenden Text abheben, um eine Einflussnahme auf den Inhalt zu unterbinden. Bei der Interpunktion werden ebenfalls Symbole verwendet, die sich deutlich von Buchstaben oder Ziffern unterscheiden. Die erste Standardisierung diesbezüglich wurde im sog. ASCII[35] verwirklicht. Doch wurde der amerikanische Standard nicht konsequent beachtet, was dazu führte, dass heutige Systeme teilweise noch immer nicht in der Lage sind, ohne vorherige Konvertierung Daten auszutauschen.

Zur Abgrenzung von logischen Blöcken innerhalb von Dokumenten wurden im Laufe der Jahre spezielle Zeichen (TAGS) verwendet, jedoch nie standardisiert, was zur Folge hatte, dass es nicht möglich war, Daten und deren Struktur, wie sie z.B. in Datenbanken vorkommen, system-übergreifend zu übertragen. Der Wunsch nach einer einheitlichen Sprache kam auf.

Im Jahre 1969 wurde bei IBM die erste moderne Auszeichnungssprache GML[36] mit dem Ziel entwickelt, Strukturen beliebiger Datenmengen beschreiben zu können. Als Metasprache ist GML in der Lage, andere Sprachen, deren Grammatik und Syntax zu erfassen. 1986 wurde GML von der ISO[37] standardisiert und zu SGML[38] umbenannt.

Folgendes Beispiel soll die SGML-Syntax veranschaulichen: [69]

[35] American Standard Code for Information Interchange

[36] Generalized Markup Language

[37] International Organisation for Standardization

[38] Standard Generalized Markup Language

```
<!DOCTYPE email [
<!ELEMENT email 0 0 ((to & from & date & subject?), text)>
<!ELEMENT text - 0 (para+)>
<!ELEMENT para 0 0 (#PCDATA)>
<!ELEMENT (to, from, date, subject) - 0 (#PCDATA)>
]>
<date>08/20/02
<to>you@yours.com
<from>me@mine.com
<text>Sehr geehrte Damen und Herren, .....
```

Der sehr hohe Komplexitätsgrad der Sprache und der daraus hervorgehenden kostenintensiven Implementierung SGML-gestützter Systeme verhinderte eine Verbreitung dieser nützlichen Technologie.

Das W3C[39] veröffentlichte im Februar 1998 "XML 1.0" (Extensible Markup Language), eine Sprache, die die Möglichkeiten von SGML in einfacherer Form einer breiten Masse zur Verfügung stellte. Nach Anderson [69] bietet XML folgende Vorteile beim Datenaustausch:

- Es ist ein offenes System. Mit XML kann man Daten zwischen Benutzern und Programmen plattformunabhängig austauschen.
- XML ist weitestgehend selbstdokumentierend. Das macht es zu einer geeigneten Lösung für B-2-B[40] und Extranet-Lösungen.
- Man kann Daten auch ohne vorhergehende Koordination der Zeichensätze austauschen.

2.11.1 XML-Syntax

Der Aufbau eines XML Dokumentes unterteilt sich in:

- Prolog (optional)
- Rumpf (zentraler Teil; Hierarchische Anordnung von Elementen)
- Epilog (Kommentare, Anweisungen etc.)

2.11.1.1 Elemente

Elemente bilden die Grundbausteine des oben genannten Rumpfes eines XML-Dokumentes. Sie sind ineinander schachtelbar und ordnen sich demgemäß recht anschaulich in einer Baumstruktur. Start- und End-Tags begrenzen einzelne Elemente. Der Start-Tag besteht aus dem durch spitze

[39] World Wide Web Consortium

[40] B-2-B Business to Business

Klammern eingeschlossen Elementnamen (z.B. <ElementName>). Der Querstrich vor dem Elementnamen unterscheidet den End-Tag vom Start-Tag (</ElementName>). Zwischen den TAGS können Werte, Inhalte oder weitere Elemente auftauchen.

Hier ein einfaches Beispiel eines XML-Rumpfes:

```
<buch>
  <autor>Max Frisch</autor>
  <titel>Homo Faber</titel>
  <ISBN>3518368540</ISBN>
  <preis>8.oo €</preis>
</buch>
```

Jedes wohlgeformte XML-Dokument besteht aus einem Baum von Elementen. Dieser Baum hat nur eine Wurzel, die Dokumentenwurzel („document root“). An dieser Wurzel hängt ein Teilbaum von Elementen, dessen Wurzel das Dokument-Element („document-element“) ist. Bis auf das „Leere Element-Tag“ muss in XML im Gegensatz zu HTML immer ein End-Tag benutzt werden. Das „Leere Element-Tag“ kann durch eine vereinfachte Schreibweise Start- und End-Tag beinhalten und trägt so zu einer besseren Lesbarkeit des Dokuments bei. Bei Benutzung dieser Form wird ersichtlich, dass dieses Element keinesfalls einen Inhalt haben darf. [69]

2.11.1.2 Attribute

Attribute beschreiben Inhalte von Elementen, ohne Teil des Inhaltes zu sein. In Anlehnung an das obige Beispiel könnte man z.B. das Element <autor> durch das Attribut "geschlecht" erweitern:

```
<autor geschlecht="m">Max Frisch</autor>
```

An welcher Stelle der Einsatz von Attributen sinnvoll ist, ist nicht pauschal zu beantworten. Man könnte die Mehrinformation "geschlecht" auch ohne Informationsverlust als Unterelement von <autor> definieren. Eine Regel kann jedoch grundsätzlich angewandt werden und liefert meist die richtige Dokumentenaufteilung: Ist eine Information nur in Verbindung mit einer anderen existent, dann ist es ratsam, diese als beschreibendes Attribut zu deklarieren. "geschlecht" ergibt hier nur durch <autor> einen Sinn. Können Informationen jedoch völlig autark in anderen Zusammenhängen verwendet oder sogar selbst durch Attribute beschrieben werden, dann sollten sie als Elemente auftauchen.

2.11.2 DTD (Document Type Definition)

Beim Erstellen eines wohlgeformten, d.h. den XML-Spezifikationen entsprechenden Dokumentes, ist es sinnvoll, vor allem bei zunehmendem Komplexitätsgrad, eine Grammatik zu definieren, die gewisse Regeln für den Aufbau und den Zusammenhang der Elemente und Attribute festlegt. DTD, als Teil der Spezifikation von XML, ist in der Lage den strukturellen Aufbau des XML-Dokumentes genau zu beschreiben. Mit Hilfe von DTD ist es validierenden Parsern möglich, XML-Dokumente auf ihre Richtigkeit zu prüfen. XML-Editoren können auf diese Weise eine fehlerhafte Strukturbildung unterbinden, indem sie dem Benutzer schon beim Erstellen des Dokumentes entsprechend der Hierarchieebene nur die zulässigen Elemente und Attribute zur Auswahl anbieten. Zusätzlich stellt die DTD eine Art Dokumentation des Entwicklungsprozesses der Gesamtstruktur dar und kann als zuverlässiges, unmissverständliches Kommunikationsmittel zwischen Entwickler und Programmierer dienen. Ein wohlgeformtes Dokument, in Verbindung mit mündlichen Absprachen oder schriftlichen Dokumentationen, kann nie so effizient sein wie eine DTD, da die Funktionsfähigkeit des Gesamtprogramms in diesem Falle von der nichtüberprüfbaren Richtigkeit des XML-Dokumentes abhängt. Die Verknüpfung einer externen DTD in einem XML Dokument geschieht über die DOCTYPE-Deklaration, gefolgt von dem Schlüsselwort SYSTEM, das dem Parser die Anweisung erteilt, an der folgenden URL/URI nach der DTD zu suchen. Beide Verweise sind möglich:

```
<!DOCTYPE buch SYSTEM "http://www.katalogserver.com/buch.dtd">
<!DOCTYPE buch SYSTEM d:/katalog/buch.dtd">
```

2.11.3 Grundlegende Markup-Deklarationen

Elemente (ELEMENT) und Attribute (ATTLIST) sind die Grundbausteine eines XML Dokuments und werden durch die arbeitserleichternden Hilfskonstrukte ENTITY und NOTATION ergänzt.

- NOTATION behandelt nicht geparste XML-fremde Inhalte.

- ENTITY stellt eine Art von Container für frei wählbare Texte dar, der durch Referenzierung an jeder beliebigen Stelle in der DTD oder im XML-Dokument eingefügt werden kann. Hierbei muss unterschieden werden, ob der Inhalt der ENTITY vom Parser beachtet werden soll oder nicht. Wenn es sich um ein sog. parsed entity handelt, dann muss auch der Inhalt wohlgeformtes XML sein!

Um beispielsweise das Copyright in einem Text nicht in all seiner Ausführlichkeit immer wieder schreiben zu müssen, kann hierfür eine ENTITY eingerichtet werden:

```
<!ENTITY copyright "© RPK Institut für Rechneranwendung... ">
```

Aufgerufen wird sie mit dem von XML reservierten Ampersand: &copyright

Parameter-Entities sind spezielle parsed entities, die nur innerhalb von DTDs benutzt werden. Soll z.B. eine Reihe von Elementen mehreren anderen Elementen zur Verfügung gestellt werden, dann ist es von großem Vorzug, einmalig die gewünschte Menge an Elementen in einer ENTITY zusammenzufassen und diese dann zuzuweisen. Grundtypen alleine reichen zur Beschreibung der Dokumentstruktur nicht aus. Operatoren sind von Nöten, um Abhängigkeiten und Beziehungen zwischen den Elementen und Attributen herstellen zu können. Tabelle 3-10 zeigt die wichtigsten BNF[41] konformen DTD Operatoren:

Operator	Bedeutung	Beschreibung
,	Sequenz	Werden Elemente durch ein Komma getrennt, dann muss die Reihenfolge im XML-Dokumenten entsprechend eingehalten werden

```
<!ELEMENT arm (oberarm, unterarm, hand)>
```

Operator	Bedeutung	Beschreibung
|	Alternative	logisches ODER

```
<!ELEMENT entscheidung (ja | nein)>
```

Operator	Bedeutung	Beschreibung

[41] Backus Naur Form

?	Optionalität	Optionalität eines Elementes

```
<!ELEMENT Kuchen (Zucker, Eier, Mehl, Milch, Hefe, Schokolade?)> // Kuchen muss in der Reihenfolge Zucker, Eier, Mehl, Milch, Hefe mit der Option auf Schokolade gemischt werden.
```

Operator	Bedeutung	Beschreibung
*	0-unendlich	kein Mal bis beliebig oft
+	1-unendlich	ein Mal bis beliebig oft

```
<!ELEMENT myElement ((A | B)+ , (C , D)*)> // A oder B ein bis unendlich mal + C gefolgt von D kein Mal bis unendlich Mal
```

Operator	Bedeutung	Beschreibung
#PCDATA	Parsed Character Data	Benutzerdaten (Inhalt/Wert eines Elementes)

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

XML 1.0 bietet weit mehr Möglichkeiten zur genaueren Spezifikation eines XML-Dokumentes, als die hier dargestellten. Weiteres, wie z.B. der genauere Umgang mit Attributen oder die Referenzierung zwischen Elementen, ist für diese Arbeit nicht von Interesse, kann aber in [70] [71] nachgelesen werden.

2.11.4 DOM (Document Object Model)

DOM repräsentiert eine Plattform- und Sprachen unabhängige Definition einer Schnittstelle für den Zugriff auf XML oder generell Web-Dokumente. Entwickelt und als Standard empfohlen wurde "DOM (Core) Level 1.0" von der W3O. Hierbei handelt es sich nicht um eine

Implementierung, sondern lediglich um eine abstrakte Beschreibung der Schnittstellen für die Objekte; sozusagen eine Anleitung oder Empfehlung wie eine Implementierung auszusehen hat.

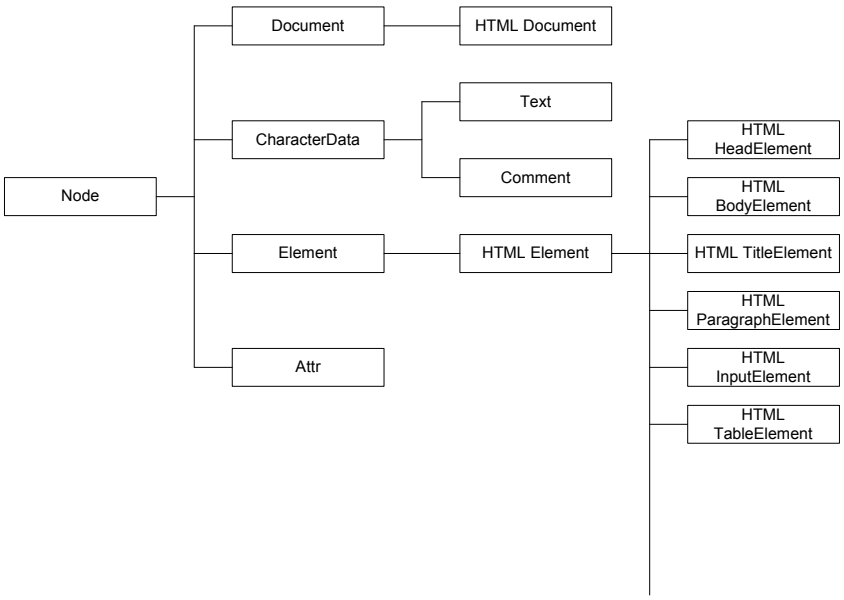


Abbildung 45: Klassenhierarchie des DOM

2.11.5 DOM-Parser

DOM-Parser sind Implementierungen des DOM und als Bibliotheken für die verschiedensten Programmiersprachen erhältlich. Die im folgenden beschriebenen C++ Objekt-Schnittstellen werden von der Firma IBM kostenlos als XML-Parser-Paket [72] angeboten und beschränken sich hier ausschließlich auf die Spezifikationen von "Dokument Object Model Core Level 1.0".

Bevor auf das XML-Dokument zugegriffen werden kann, muss der XML-Parser initialisiert, ein Objekt des Typs DOMParser erzeugt und mit der Methode parse() das XML-Dokument in den Speicher geladen werden:

```
XMLPlatformUtils::Initialize();
DOMParser myParser = new DOMParser;
myParser->parse(Pfad_des_XML_Dokuments);
DOM_Document myDocument = myParser->getDocument();
DOM_Node root = myDocument.getDocumentElement();
```

Das XML-Dokument steht nun als hierarchisch geordneter Baum von Objekten dem Parser zur Verfügung. Die hier beschriebenen Klassen und Methoden sind zum Navigieren und Auslesen notwendig, umfassen jedoch nicht die komplette DOM-Implementierung.

- DOM_Document bildet das Dokument ab.

```
getDocumentElement(); // Gibt das Wurzelelement vom Typ DOM_Node zurück.
```

Methoden:

- DOM_Node

Jedes Element eines XML-Dokumentes wird im Speicher als DOM_Node abgebildet.

```
DOM_Node getFirstChild(); // Gibt das erste Kind zurück.
DOM_Node getLastChild(); // Gibt das letzte Kind zurück.
DOM_Node getNextSibling(); // Gibt den nächsten Nachbarn zurück.
DOM_Node getPriviousSibling(); //Gibt den vorherigen Nachbarn zurück.
DOM_NodeList getChildNodes(); //Gibt eine List aller Kinder zurück.
DOM_NamedNodeMap getAttributes(); // Gibt die Attribute eines Nodes zurück.
DOMString getNodeName(); // Gibt den TAG-Namen zurück.
DOMString getNodeValue(); // Gibt den Wert zwischen den TAGs zurück.
```

Mit den folgenden Methoden kann zu jedem beliebigen Element navigiert werden:

- DOM_NodeList Liste von DOM_Nodes.

Methoden:

```
DOM_Node item(i); // DOM_Node an der Stelle i der Liste
int getLength(); // Gibt die Anzahl der Nodes der Liste zurück
```

Kapitel 3

KONZEPTE FÜR EIN INTEGRIERTES SYSTEM ZUR INFORMATIONSVISUALISIERUNG

Integrierte Informationsvisualisierung über alle Produktlebenslebensphasen hinweg ist nur möglich, wenn alle produktbeschreibenden Informationen, wie z.B. Feature-Informationen aus einem CAD-System oder Anforderungen aus einem System-Engineering (SE) Werkzeug, dem Visualisierungsmodul zur Verfügung stehen. Abbildung 46 zeigt die einzelnen Phasen des Produktlebenslaufs. Es wird exemplarisch die Produktentwicklung mit den vier wichtigsten Werkzeugen, die in dieser Phase eingesetzt werden, herausgegriffen.

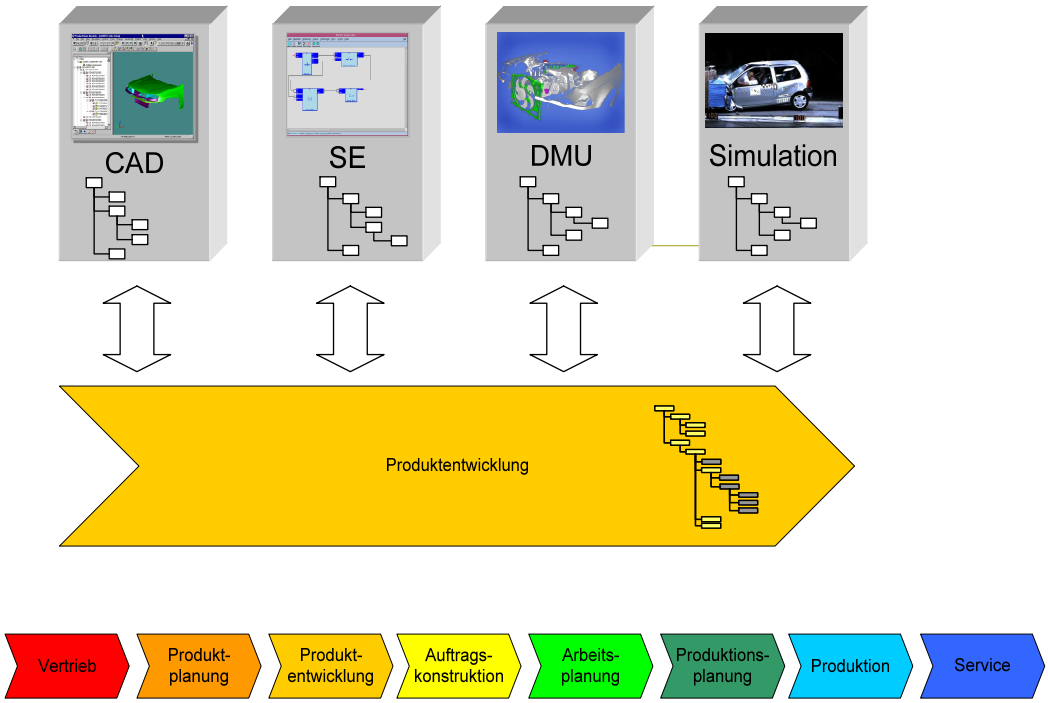


Abbildung 46: Datenquellen entlang des Konstruktionsprozesses

Ausgehend von der Anforderungsmodellierung, die in der Phase der Produktplanung stattgefunden hat, werden die Funktions- und Prinzipstrukturen in einem System Engineering (SE) Werkzeug modelliert. Diese Modelle bilden die Grundlage der Bauteilgestalt, die klassisch in einem CAD-System definiert wird. Ausgehend von dieser Bauteilgestalt, lassen sich digitale Prototypen in einer Digital Mock Up (DMU) Umgebung zusammensetzen und das physikalische Verhalten in Simulationsumgebungen testen. Jedes dieser Werkzeuge besitzt sein eigenes Modell des zu konstruierenden Bauteils (Zusammenbaus). Zwar können diese Werkzeuge in der Regel über Neutralfile-Formate Daten austauschen, eine echte Integration (z.B. Konstruktionshistorie,

Featureinformationen, …) ist nicht gegeben. Bemühungen, diese Daten in einem einzigen monolithischen, konsistenten, redundanzfreien integrierten Produktmodell abzulegen, haben nicht die Erwartungen erfüllt (vgl. Kapitel 2.3). In der Praxis zeigt sich, dass der Ansatz, diese Systeme als verteilte Datenquellen zu betrachten und über eine Adapter-Technologie auf Methodenebene anzusteuern, viel versprechender ist als der Ansatz eines monolithischen integrierten Produktmodells.

3.1 Forderungen an ein integriertes Visualisierungssystem

In diesem Abschnitt werden die Anforderungen an ein integriertes lebensphasenübergreifendes Visualisierungswerkzeug beschrieben. Sie bilden die Grundlage für die Systemarchitektur in Kapitel 4.

3.1.1 Unabhängigkeit von den Datenquellen

Die Akzeptanz, ein Softwaresystem einzuführen, ist sehr davon abhängig, inwieweit bestehende Daten weiterverwendet bzw. ohne Verlust migriert werden können [73] [74]. Die Datenmodelle der CAD-Systeme und der von PDM-Systemen vorgegebene Workflow sind die „de facto" Säulen der Konstruktion. Soll ein plattformübergreifendes interoperables System eingeführt werden, so kann dies in der industriellen Praxis nur so geschehen, dass existierende Strukturen nicht ersetzt werden, sondern in einen semantischen Bezug gebracht werden. Zielsetzung ist, damit bestehende Datenmodelle (z. B. in CAD-Systemen) zu verwenden, um sie semantisch mit Informationen aus anderen Systemen zu verknüpfen. Wenn man sich vorstellt, dass diese Verknüpfung natürlich in hohem Maße von der Datenquelle an sich abhängt, wird schnell klar, dass dieser Verknüpfungsmechanismus, unabhängig von der Datenquelle, realisiert werden muss, um den Aufwand überhaupt bewältigen zu können. Diese Forderung nach Unabhängigkeit von der Datenquelle führt bei der Konzeption des Systems zu einer Architektur, die in Form von Modulkomponenten eine Klasse von Systemen über eine Methodenschnittstelle abstrahiert.

Hierbei ist es jedoch nicht ausreichend, die Daten der einzelnen Werkzeuge (Systeme) bei der Produktentwicklung analog von PDM-Systemen auf Dokumenten-Ebene zu integrieren. Es müssen Relationen zwischen den in den verschiedenen Systemen verborgenen Informationen möglich sein, um diese Abhängigkeiten zwischen den verschiedenen Modellen der Systeme hinterher Visualisieren zu können und sie dem Konstrukteur zu verdeutlichen. Beispielsweise müssen Produktanforderungen aus einem SE-Werkzeug direkt mit Wirkflächenpaarungen aus dem CAD-Modell verknüpfbar sein. Bisherige Konzepte für Integrationsplattformen, insbesondere iViP, leisten dies nicht. Abbildung 47 und Abbildung 48 verdeutlichen den Unterschied bei der

Verknüpfung von Informationen auf Dokumentenebene und der Verknüpfung von Informationen aus den systeminternen Modellen.

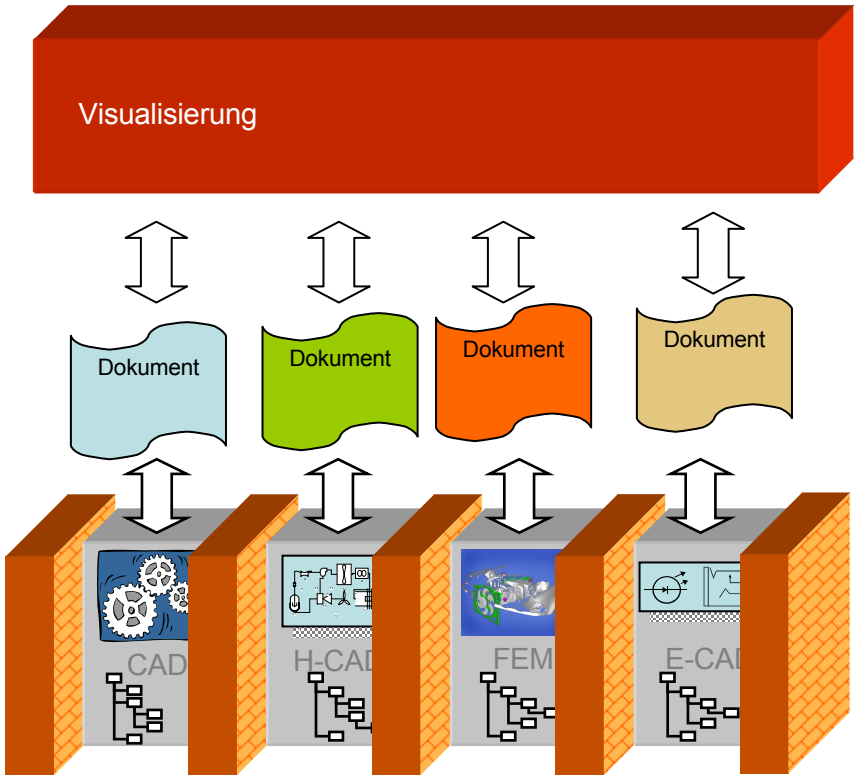


Abbildung 47: Verknüpfung von Informationen auf Dokumentenebene

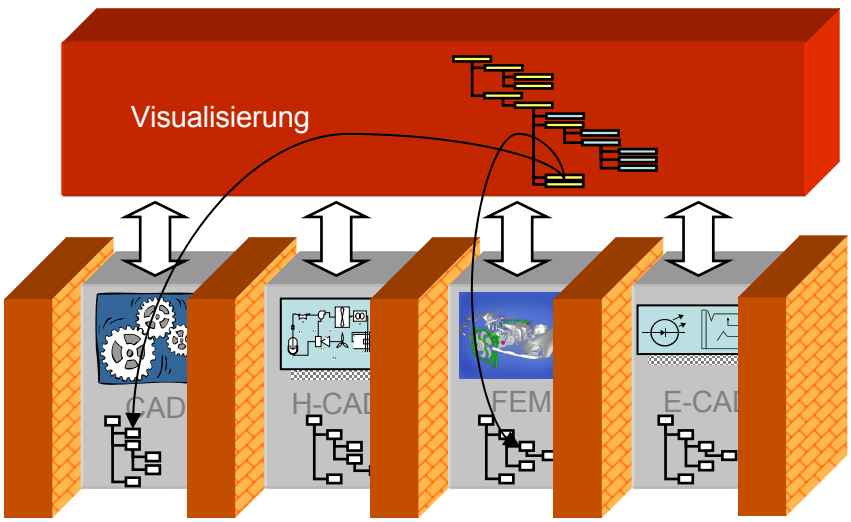


Abbildung 48: Verknüpfung von Informationen in den Systemen

3.1.2 Universalität und Portabilität

Die Definition der Methodenschnittstelle muss standardisiert sein, um die Zusammenarbeit der Komponenten sicherzustellen und die Akzeptanz der Schnittstelle zu gewährleisten. Die Interface Definition Language (IDL) von CORBA bietet sich hierfür an, da sie eine plattform- und sprach-

unabhängige Beschreibung der Methodenschnittstelle erlaubt. (vgl. 2.8.1) IDL ist in jedem Fall unabhängig von den Eigenheiten der letztlich verwendeten Implementierungssprache.

3.1.3 Erweiterbarkeit

Produktivitätssteigerung bei der Implementierung von Software basiert auf mehreren Ansätzen:

- Bessere Erweiterbarkeit von Software durch Modularisierung und eine klare Definition von Schnittstellen zwischen den einzelnen Softwarekomponenten

- Modulgrenzen und Schnittstellen definieren klare Grenzen, was zu einer besseren Kontrolle über Komplexität und Kosten von Software-Wartung führt

- Wiederverwendung existierender Softwarekomponenten:

    - Durch mehrfache Instantiierung allgemein einsetzbarer Softwaremodule (Servicekomponenten) sowie durch Anpassung existierender Komponenten an erweiterte oder abgewandelte Anforderungen (Customizing).

    - Frameworks geben komplette Anwendungsgerüste vor, die durch Anpassung einzelner Komponenten zu einer maßgeschneiderten Anwendung geformt werden können.

Bisherige Integrationsansätze, insbesondere iViP, zeigen, dass eine Integration nur auf Basis der PDM-Enabler-Spezifikation, wenn auch mit gewissen Erweiterungen, nicht den Erwartungen an Funktionalität, wie man sie von einer Integrationsplattform erwartet, erfüllt. Der iViP-Client ist vom Konzept her eine Integrationsplattform für Komponenten, die direkt für den iViP-Client entwickelt wurden. Durch eine Beschränkung auf die PDM-Enabler Spezifikation ist jedoch eine tief greifende Integration bestehender Entwicklungswerkzeuge (z.B. CAD-, CAE, SE-Systeme) eher schwierig. Ein Konzept zum Aufbau wieder verwendbarer modularer Softwarekomponenten ist nicht vorgesehen. D.h. nicht jedes Produktentwicklungswerkzeug lässt sich mittels der PDM-Enabler-Spezifikation integrieren, vielmehr muss für jede Klasse von Systemen eine eigene Methodenschnittstelle geschaffen werden.

Zukünftige Entwicklungen könnten ergeben, dass eine Erweiterung der Methodenspezifikation nötig ist. Diese Erweiterungen müssen durch einen definierten Standardisierungsprozess in die Methodenschnittstellen-Spezifikation eingebracht werden.

3.1.4 Serviceorientierung

Der herkömmliche Weg, CAx-Anwendungen zu integrieren, war bisher, mittels individueller Schnittstellen zwischen jedem dieser Systeme Datenaustausch zu betreiben. Diese Vorgehensweise führt bei einer mittlerweile kaum mehr überschaubaren Anzahl von Systemen zu einer Schnittstellenvielfalt, die nicht mehr zu beherrschen ist. Stand der Technik heute ist eine busähnliche Struktur, bei der rudimentärer Datenaustausch auf Neutralfileformat vorherrscht.

Abbildung 49 zeigt links die vorgehensweise bei Individualschnittstellen. Die beteiligten Systeme können bidirektional Informationen austauschen, wobei es durchaus häufig der Fall ist, dass ein Zwischenschritt über ein drittes System erforderlich ist. Rechts oben wird eine busartige Struktur verwendet um mittels eines Neutralen Austauschformates Informationen (verlustbehaftet) zwischen den Systemen auszutauschen. Rechts unten wird eine serviceorientierte Architektur, bei der der Zugriff auf die Datenelemente in den jeweiligen Quellsystemen über standardisierte Methodenschnittstellen erfolgt, vorgestellt. Wesentliche Informationen die mehrere Systeme benutzen werden in Basisdiensten zur Verfügung gestellt. Jede implementierte CAx-Anwendung nutzt diese Dienste über eine middlewarebasierte Kommunikationsinfrastruktur, was eine hohe Interoperabilität zu Folge hat. Diese Aufteilung in Basisdienste führt zu der in Abbildung 55 vorgestellten Schichtenstruktur.

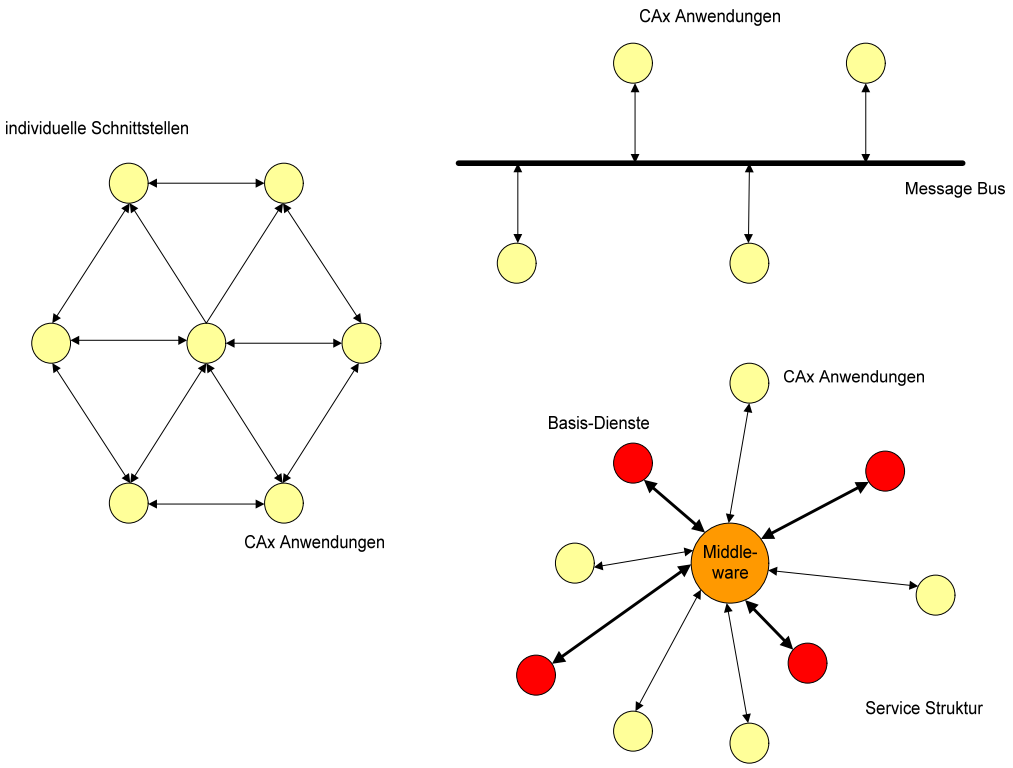


Abbildung 49: Service-Orientierung

3.1.5 Skalierbarkeit / Clustering

Leistungserhöhung wird heute als dominierendes Ziel angesehen, solange andere Aspekte (Kosten!) in einem vertretbaren Rahmen vernachlässigt werden können. Um einen hohen Grad an Immersion bei der Visualisierung zu gewährleisten, ist eine flüssige Darstellung unabdingbar. In der industriellen Praxis werden heute zwei Strategien gefahren:

- Hochleistungsrechner (z.B. SGI Onyx 4) mit mehreren Grafikpipes
- Clustering von Standardrechnern

Da monolithische Hochleistungsrechner mehr und mehr an Bedeutung verlieren und mit vertretbaren Qualitätseinbußen (z.B. Anti-Aliasing) die erforderliche Leistung auch von einem Verbund (Cluster) von Standardrechnern erbracht werden kann, ergibt sich die Forderung durch Clustering skalieren zu können.

Allgemein lässt sich sagen, dass ein Software-Produkt "gut" skaliert, wenn es beispielsweise bei der zehnfachen Nenn-Last (d.h. Leistung) mit den ca. zehnfachen Ressourcen auskommt. Ein "schlecht" skalierendes Produkt hingegen würde vielleicht bei doppelter Last bereits die zehnfachen Ressourcen benötigen und bei zehnfacher Last komplett versagen.

Dementsprechend benötigt ein gut skalierbares paralleles Programm bei gleicher Last aber der doppelten Anzahl von Prozessoren die Hälfte der Rechenzeit. Wegen der Kommunikation zwischen den Prozessoren wird dieser Wert allerdings nie exakt erreicht; es entsteht ein Communication Overhead.

Das Amdahl'sche Gesetz (Formel 1) ist ein Gesetz aus dem Bereich der Informatik, das der Computerarchitekt Gene Amdahl (ehemals bei IBM, dann Gründer des ehemaligen Großrechner-Herstellers Amdahl) aufgestellt hat. Nach diesem Gesetz ist der maximale Speedup s eines Algorithmus, den man durch Einsatz paralleler CPUs (Anzahl P) gewinnen kann, abhängig von dem prozentualen Anteil des sequentiellen Anteils des Algorithmus a, d.h. des Anteils, in dem alle aufeinander folgenden Teilberechnungen von der vorherigen Berechnung abhängig sind:

$$s = \frac{1}{a + (1 - a) / P}$$

Formel 1: Amdahl'sches Gesetz

Die zu entwickelnde Visualisierungssoftware muss in mehreren Hinsichten skalierbar sein. Eine einzige große Projektionswand muss in mehrere Bereiche aufteilbar sein, wobei die einzelnen

Prozessoren jeweils einen Teil des zu visualisierenden Universums[42] erzeugen (vgl. Abbildung 29). Auch andere Konfigurationen, wie z.B. mehrseitige Projektionen mit bis zu zwei Projektoren pro Wand, müssen unterstützt werden.

3.1.6 Immersion

Wie auch in der filmischen Immersion beschreibt der Begriff das Eintauchen in eine künstliche Welt. Im Unterschied zu der passiven, filmischen Immersion gewinnt die Immersion in die Virtuelle Realität, durch die Interaktion mit der virtuellen Umgebung, eine wesentlich höhere Intensität. Abbildung 50 ordnet die Virtuelle Realität in einem Raum aus Visualisierung, Interaktion und Semantik der Geometrie an. „Semantik der Geometrie" bezeichnet in diesem Zusammenhang grafische Metaphern für Informationen, die per se keine grafische Repräsentation haben.

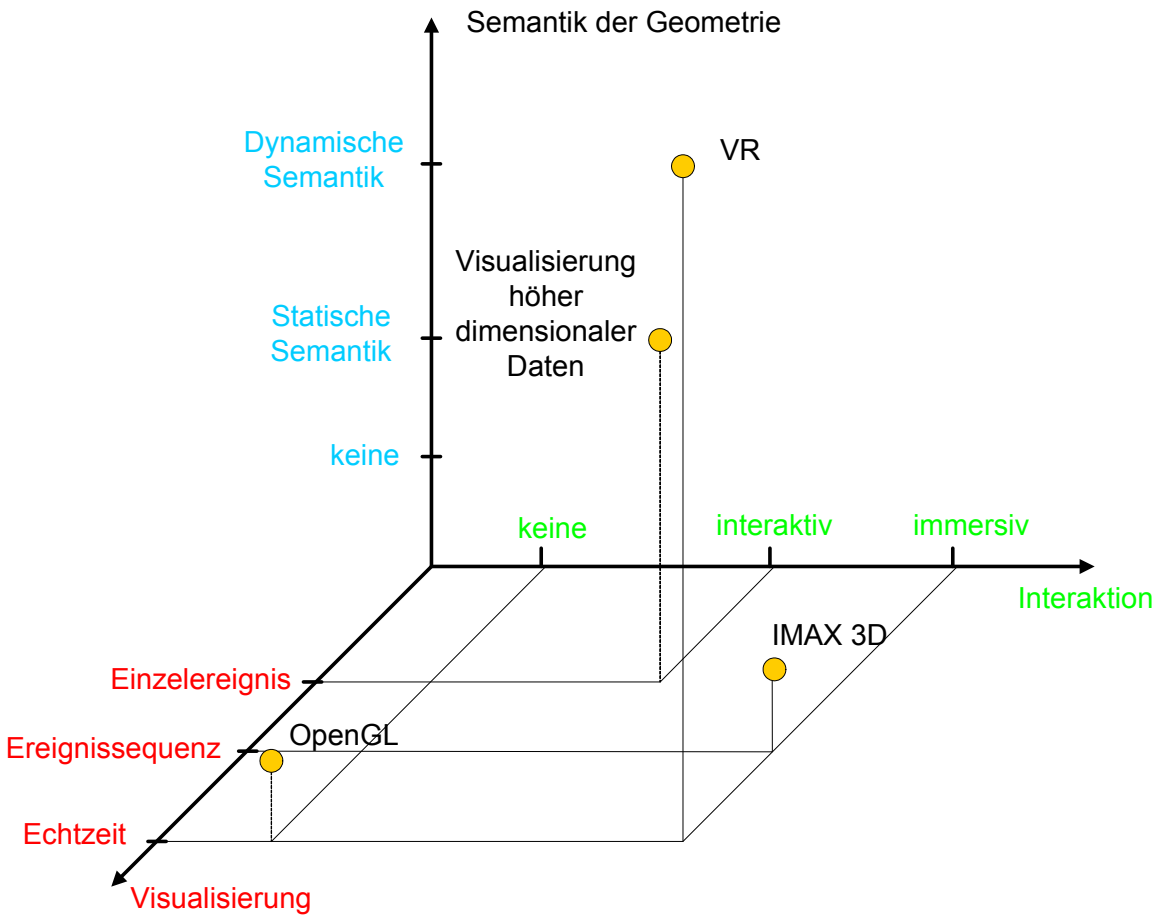


Abbildung 50: Anordnung der VR im dreidimensionalen Raum aus Interaktion, Visualisierung und Semantik der Geometrie (Quelle: Universität Bonn, Computergrafik)

[42] Bei Szenen-Graph basierten Systemen spicht man von einem Universum als Gesamtheit aller Objekte die zu Visualisieren sind (Licht, Kamera, geom. Objekte, …)

Dementsprechend stellt sich die Forderung an das Visualisierungssystem, neben der intuitiven Interaktion und der Visualisierung in Echtzeit, Metainformationen in geometrischen (visualisierbaren) Objekten mit hohem Gehalt an Semantik darzustellen.

3.2 Engineering Objects

Um nichtgeometrische Metainformationen mit geometrischen Metainformationen zum Zwecke der Visualisierung in Verbindung zu setzen, muss eine Systematik existieren, die in der Lage ist, solche Relationen persistent, also über die gesamte Lebenszeit dieser Information, abzubilden.

Im Rahmen des ITEA[43]-BmbF[44] Verbundprojektes „3D-Workbench" wurde von der Daimler-Chrysler Forschungsabteilung RIC/EP im Forschungszentrum Ulm das Konzept der Engineering Objects (UMEO[45]) eingebracht. Ziel des ULEO[46] Ansatzes ist es, ein XML-basiertes (siehe 2.11) Informationsmodell zu Verfügung zu stellen, das Informationsobjekte und deren Relationen entlang des Entwicklungsprozesses abzubilden vermag [75] [76]. MTRT[47] ist ein weiteres Datenmodell, welches eine Abstraktionsschicht höher angesiedelt ist und alle Arten von Relationen innerhalb des UMEO Modells beschreibt.

XML ist nur eine Möglichkeit, die UMEO Klassen in einen physikalischen Entwurf abzubilden. Die Motivation, XML zu benutzen, liegt darin, dass XML sowohl applikations- als auch plattformneutral ist und einen hohen Verbreitungsgrad besitzt. Das UMEO Datenmodell wird durch zwei XML-Schemata (XSD vgl. 2.11) beschrieben, `UMEOclass.xsd` und `UMEOinst.xsd`. Die kompletten Schemata befinden sich in Anhang E.

Diese beiden Schemata werden verwendet, um die Metainformationen, die entlang des Produktenstehungsprozesses in den jeweiligen Systemen entstehen, über Verweise in einer Klassenstruktur strukturiert abzubilden, um sie dem Visualisierungssystem zur Verfügung zu stellen. Sie bilden sozusagen die Basis für die Klassifikation der darzustellenden Metadaten. Die Abbildung von Feature-Informationen (Konstruktions- und Messfeatures) wird in [77] beschrieben. [11] beschreibt, wie Produktanforderungen in das UMEO Schema abgebildet werden können. Die folgenden beiden Abschnitte geben einen kurzen, rudimentären Einblick in die beiden Schemadefinitionen. Eine ausführliche Beschreibung findet sich in [77], [75], [76] und [78].

---

[43] ITEA (engl.) Information Technology for European Advancement

[44] BmbF Bundesministerium für Bildung und Forschung (Projektträger DLR)

[45] Unified Model of Engineering Objects (UMEO)

[46] Universal Linking of Engineering Objects (ULEO)

[47] Meta Taxonomy of Relation Types (MTRT)

3.2.1 XML-Schema für Klassen: UMEOclass

Das UMEOclass-Schema (Abbildung 51 und Abbildung 52) beschreibt Objekt- und Relationsklassen.

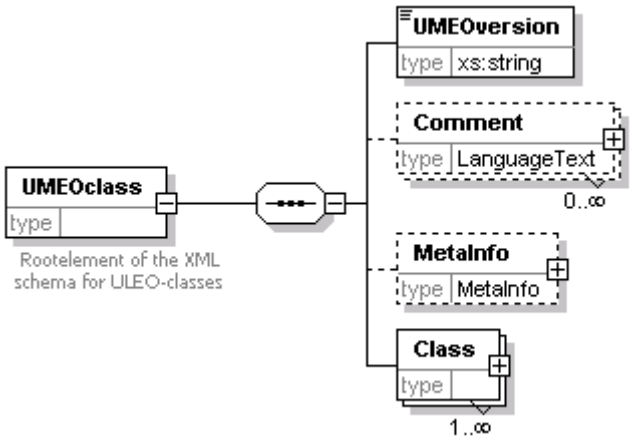


Abbildung 51: Wurzelelement des UMEOclass Schema

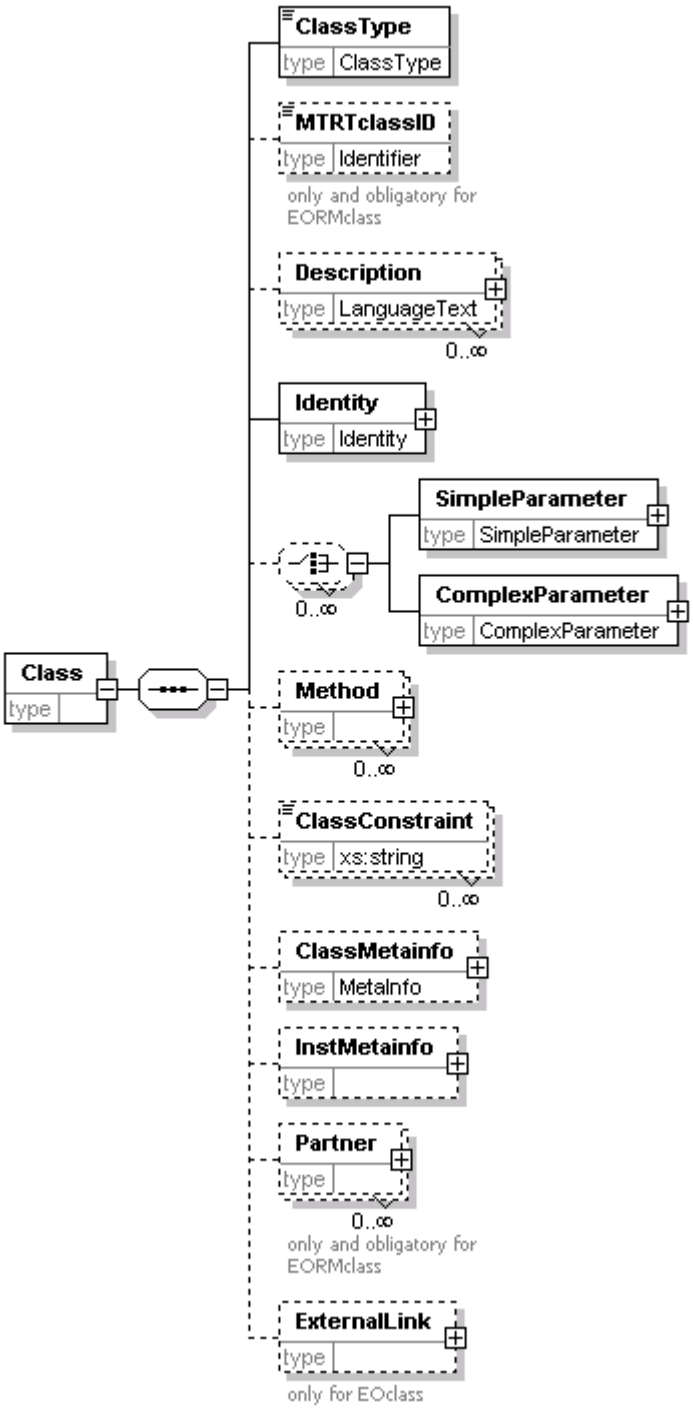


Abbildung 52: UMEOclass Class Element

3.2.2 XML-Schema für Instanzen: UMEOinst

Das UMEOinst Diagramm beschreibt Instanzen von Objekten und Relationen. Abbildung 53 zeigt das Wurzelelement des UMEOinst Schemas. Dieses Element hat natürlich die gleichen Objekte wie das dazugehörige Klassendiagramm UMEOclass, wobei bei den Bezeichnern „Klasse“ durch „Instanz“ ersetzt ist.

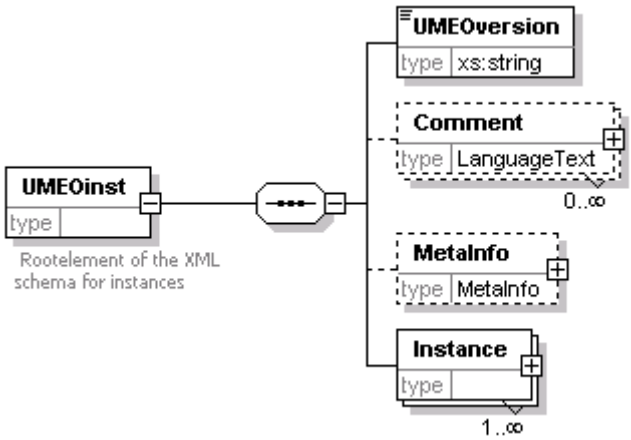


Abbildung 53: Wurzelelement des UMEOinst Schema

Das Instance Element:

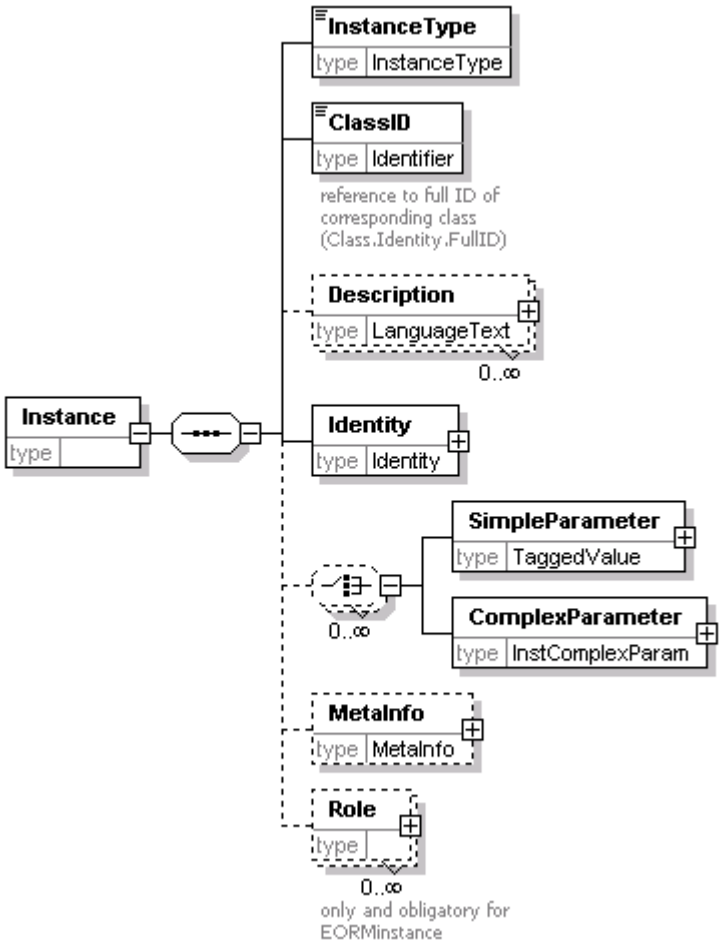


Abbildung 54: UMEOinst Instance-Element

Kapitel 4

KONZEPT EINES PRODUKLEBENSZYKLUSORIENTIERTEN VISUALISIERUNGSMODULS

Die industrielle Praxis zeigt, dass die eingesetzten CAD- und PDM-Systeme des Unternehmens die Säulen der Produktentwicklung sind. Um die Daten aus diesen Systemen direkt zu nutzen, wird in Abschnitt 4.1 ein serviceorientierter Ansatz (verteilte Software-Systeme) herangezogen, um ein Plattformkonzept [79] zu beschreiben, welches die integrierte Informationsvisualisierung, unabhängig von den eingesetzten Datenquellen, möglich macht. Die so erreichbaren Informationen müssen kontext- und problembezogen visualisiert werden. Abschnitt 4.2 beschreibt, wie man mittels der Scripting-Technologie (Abbildungsvorschriften) eine frei konfigurierbare Zuordnung von Informationen und den zugehörigen grafischen Metaphern erreicht. Eine mögliche Ausprägung von grafischen Metaphern für Anforderungen und Features wird in Abschnitt 4.3 aufgezeigt, wobei viele weitere anwendungsbezogene Umsetzungen denkbar sind. Zuletzt wird in Abschnitt 4.4 die Methodenschnittstelle der Visualisierungskomponente definiert, die es ermöglicht, diese nahtlos in das in 4.1 beschriebene Plattformkonzept als Applikationskomponente einzuordnen und so anderen Applikationen den Zugriff zu ermöglichen.

4.1 Plattformkonzept für eine integrierte Informationsvisualisierung

In Abschnitt 3.1.4 wurde die Notwendigkeit einer serviceorientierten modularen Infrastruktur aufgezeigt, die den Zugriff auf die Daten der einzelnen Produktdefinitionssysteme ermöglicht. Im Rahmen dieser Arbeit wurde ein Framework, bestehend aus mehreren Schichten, entwickelt, das die wesentlichen an der Produktenstehung beteiligten Systeme in funktionale Einheiten zerlegt und innerhalb des Frameworks gruppiert.

Unterste Ebene des Schichtenmodells (siehe Abbildung 55) bilden die *Datenmodellierungskomponenten*. Sie erlauben den standardisierten Zugriff auf alle Datenquellen über definierte Schnittstellen (APIs[48]) entlang des Produktlebenslaufs. Typische Beispiele sind hier CAD- oder NC-Systeme. Aufbauend auf der Datenmodellierungsschicht befinden sich die *Anwendungskomponenten*. Im Gegensatz zur Datenmodellierungsschicht befinden sich hier alle Methoden der Plattform. Methoden bezeichnen in diesem Zusammenhang nicht die Methodenschnittstellen, sondern Standardoperationen (Anwendungsfunktionalität), die üblicherweise benötigt werden. Beispiel ist hier das Erzeugen einer tessellierten Ansicht eines CAD-Objektes. Die oberste Ebene bildet die

[48] API (engl. = application programming interface)

Endbenutzeranwendungen. Dies sind z.B. FEM Berechnungs- oder Visualisierungswerkzeuge, die Gebrauch von diesen Schichten machen. Eine Anwendung zum Testen der Systemfunktionalität und als Startpunkt für Anwendungsentwicklung (Testharness) ist ebenfalls vorgesehen. Das Schichtenmodell basiert auf CORBA als Middleware, wobei alle Schnittstellen systemneutral in IDL[49] spezifiziert sind. (Abbildung 55 zeigt das Schichtenmodell und die hierin vorgesehenen Module.)

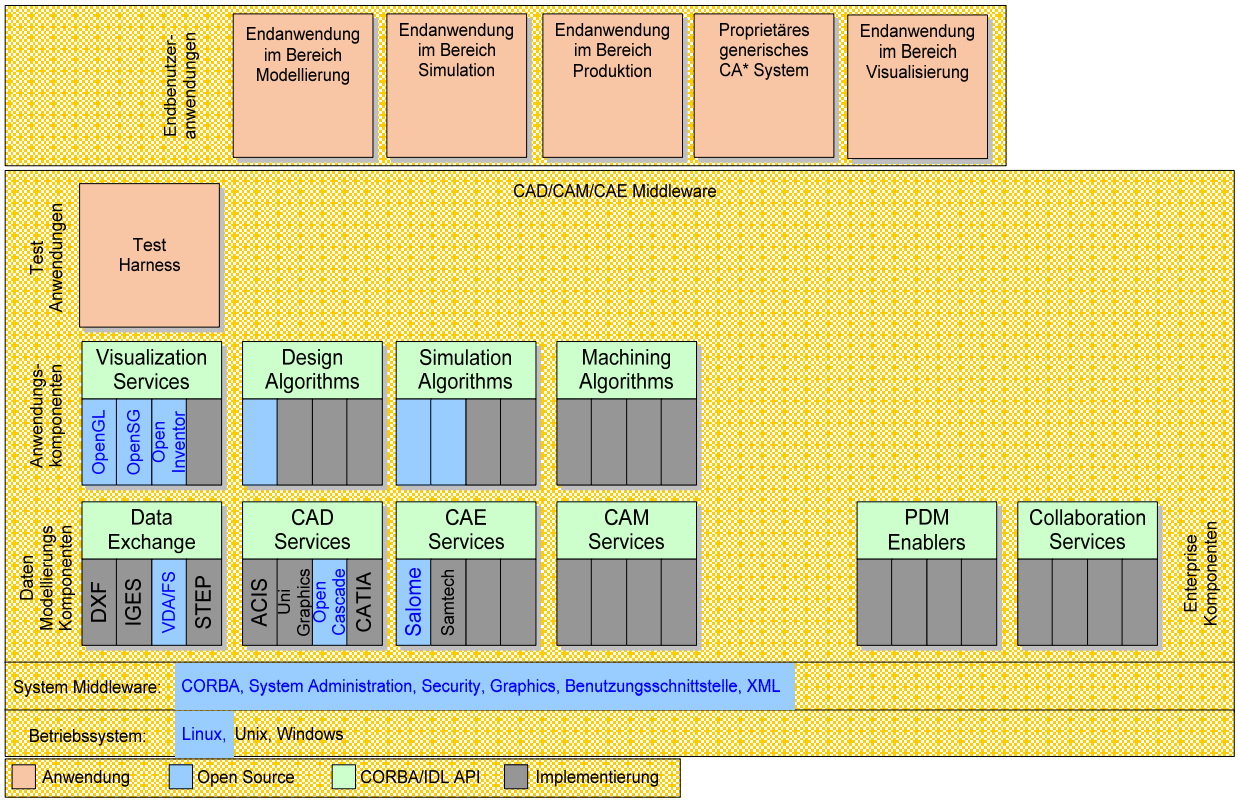


Abbildung 55: Modulares Schichtenmodell [80]

Ein Visualisierungssystem wird innerhalb dieses Schichtenmodells so implementiert, dass die eigentliche Anwendung eine Endbenutzeranwendung ist, die Funktionalität zur Visualisierung aber in einer eigenen Komponente innerhalb der Schicht der Anwendungskomponenten implementiert ist. Abschnitt 4.4 definiert die IDL Methodenschnittstelle der VisualisationServices. Diese Vorgehensweise, Funktionalität einer Anwendung zu abstrahieren und mittels einer geeigneten API innerhalb der Plattform zur Verfügung zu stellen, bildet die Grundlage für den Aufbau einer Softwarearchitektur, die wieder verwendbare Komponenten enthält.

[49] IDL (engl. = interface definition language)

4.1.1 Datenmodellierungskomponenten

Die Schicht der Datenmodellierungskomponenten realisiert den transparenten Zugriff auf ein beliebiges System entlang des Produktlebenslaufs. Die praktische Realisierung der Datenmodellierungskomponenten funktioniert derart, dass zuerst die IDL Spezifikation der Schnittstelle in die CORBA Skeletons (Serverobjekt) überführt wird. In einem zweiten Schritt werden diese Methodenrümpfe dann mittels der API des jeweiligen Systems ausformuliert. Abbildung 56 zeigt den schematischen Ablauf, Abbildung 57 zeigt die Implementierung an einem realen Beispiel in Pro/ENGINEER.

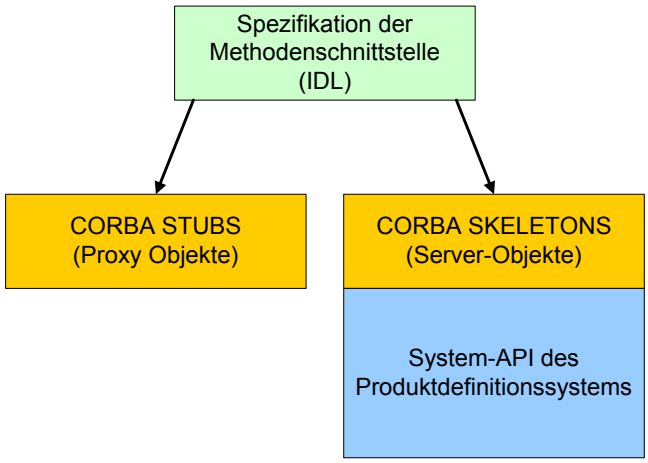


Abbildung 56: Praktische Realisierung der Datenmodellierungskomponenten (CORBA)

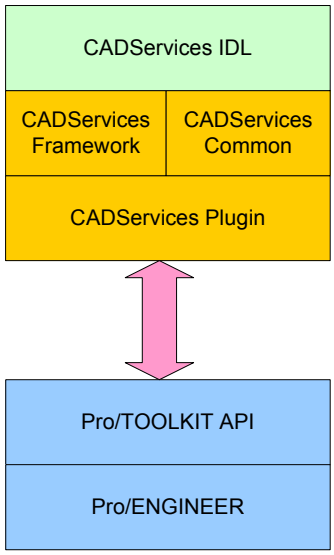
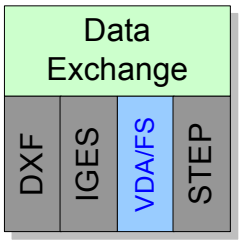


Abbildung 57: Implementierung der CAD Services am Beispiel Pro/ENGINEER

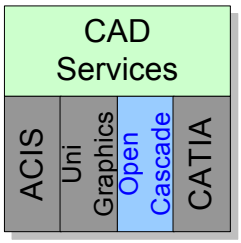
Die folgenden Abschnitte erläutern die einzelnen Datenmodellierungskomponenten im Detail.

DATA Exchange



DATA Exchange Komponenten realisieren innerhalb des Frameworks den Zugriff auf Neutralfileformate wie STEP oder IGES. Obwohl es in einem gewissen Widerspruch zur Philosophie der reinen Methodenschnittstellen auf Middleware-basis steht, muss eine Kompatibilität zu diesen Dateiformaten sichergestellt werden, um Migrationsprozesse zu erleichtern. In der industriellen Praxis findet sich auch unter dem Gesichtspunkt der Datenarchivierung oft noch die Notwendigkeit, diese Dateiformate zu erzeugen.

CAD Services



Die CAD-Services realisieren den systemneutralen Zugriff auf ein beliebiges CAD-System. Die Schnittstellendefinition wurde in Abschnitt 2.9 ausführlich erläutert und beschreibt topologisch-geometrische Strukturmodelle (B-Rep) von CAD-Systemen. In Abbildung 55 sind exemplarisch einige in der Praxis häufig eingesetzte CAD Systeme aufgeführt, wobei einzig der Open-CASCADE Kern im Moment als OpenSource verfügbar ist. Abbildung 57 zeigt die praktische Realisierung eines CAD Service Adapters am Beispiel von Pro/ENGINEER.

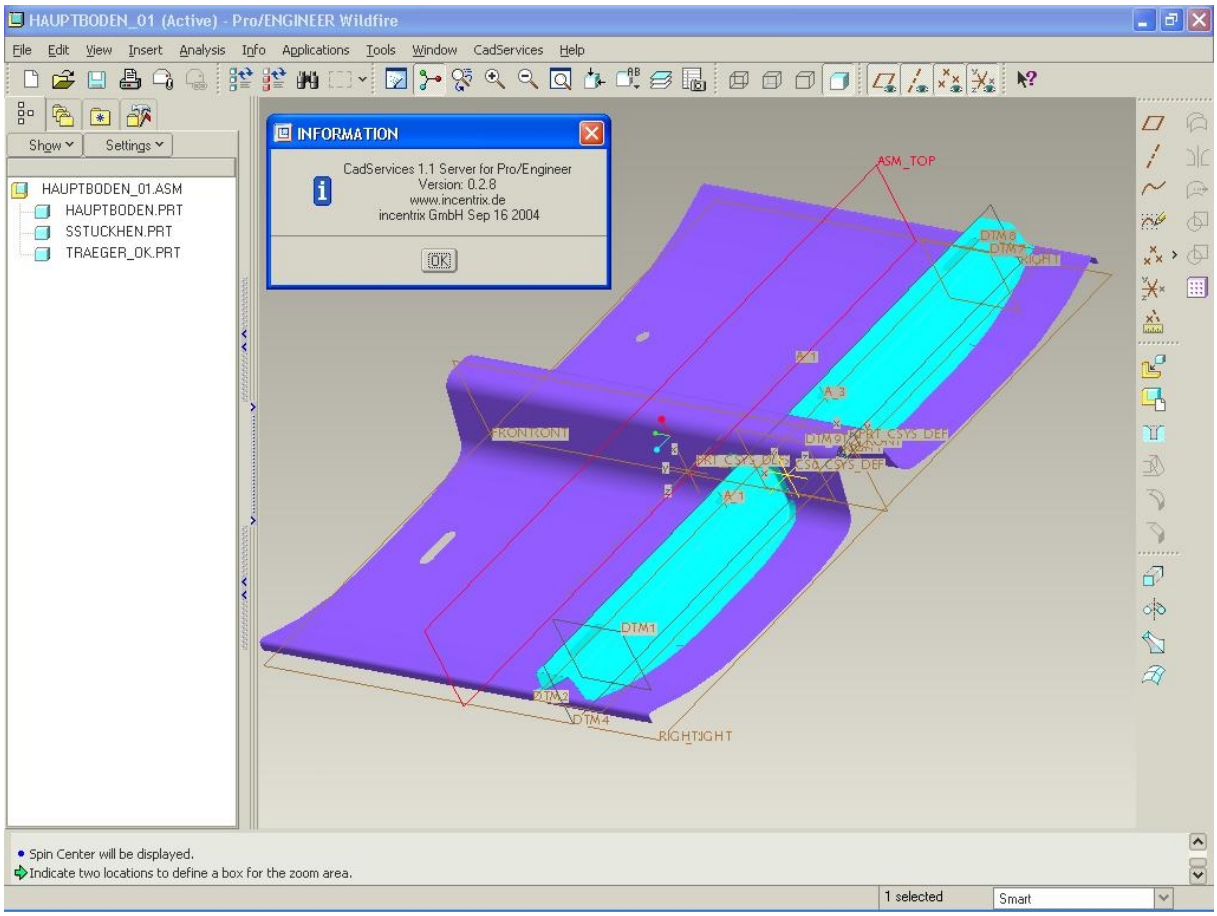
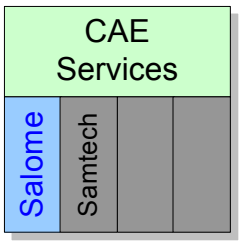


Abbildung 58: Einbindung der CAD Services in die Benutzungsschnittstelle

Der CAD-Services-Server-Plugin ist direkt auf der Pro/TOOLKIT Schnittstelle von Pro/ENGINEER implementiert und erscheint nach dem Systemstart als Menüeintrag in der Applikation (vgl. Abbildung 58). Die Pro/TOOLKIT API-Aufrufe arbeiten direkt auf dem Granite-Kern von Pro/ENGINEER und bilden das topologisch-geometrische Strukturmodell auf die CAD-Services Methodenschnittstelle ab.

CAE Services

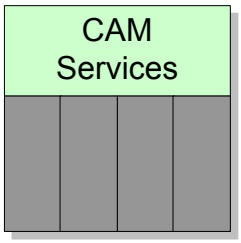


Die CAE-Services bilden die Gesamtheit der Engineering-Systeme im Produktentwicklungsprozess ab. Unter ihnen befinden sich Finite-Elemente-Systeme, Anforderungsmodellierungssysteme, System-Engineering-Anwendungen und CFD[50]-Systeme. Diese Datenmodellierungskomponente ermöglicht nur den Zugriff auf die systeminternen Datenstrukturen, jedoch

[50] CFD (engl. Computational fluid dynamics) Numerische Strömungsberechnungssysteme

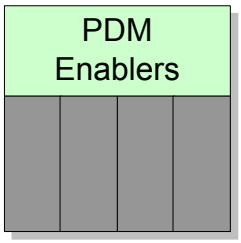
nicht auf die Anwendungsfunktionalität der Systeme. Im weiteren Verlauf des Definitionsprozesses der IDL-Schnittstelle kann lässt zeigen, dass eine weitere domänenabhängige thematische Unterteilung der CAE-Services sinnvoll erscheint.

CAM Services



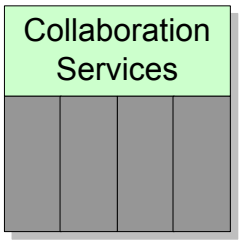
Die CAM-Services stellen die Schnittstelle zu Softwaresystemen für NC-gesteuerte Produktionseinrichtungen (wie z. B. Bearbeitungszentren, NC-Fräsmaschinen, NC-Laserschneidmaschinen und Biegezentren) dar. Hier wird in der Regel ein einfaches tessellierten Oberflächenmodell als Datenstruktur anzutreffen sein, welches traditionell schon seit Jahren Verwendung findet. Obwohl diese Systeme eigentlich genauere Geometrieinformationen benötigen sind hier sehr oft noch „diskrete“ (approximierende) Geometriemodelle anzutreffen. Möglicherweise ist das in dem Ursprung dieser Systeme, den Maschinensteuerungen, begründet die von Beginn an nur Kreisbögen und Polygone unterstützt haben. Beispiele hierfür sind: WICAM PN400, TEBIS, DelCAM, uvm. Mittlerweise stoßen diese Systeme jedoch an die Grenzen der von ihnen bereitgestellten Genauigkeit.

PDM Enabler



Die PDM-Enabler sind die am längsten bekannte, von der OMG standardisierte Komponente in dem Framework und spezifizieren eine produktneutrale Schnittstelle zu PDM-Systemen.

Collaboration Services

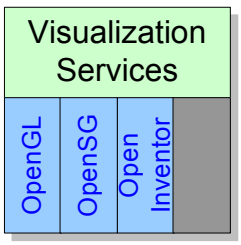


Die Collaboration-Services definieren eine Schnittstelle für Systeme der kooperativen Produktentwicklung und der Telepräsenz. Diese Systeme spielen eine bedeutende Rolle beim Concurrent- bzw. Simultaneous Engineering. Bisher existiert keine von der OMG standardisierte Schnittstelle für diese Komponenten. Sie werden im Rahmen dieser Arbeit nicht weiter betrachtet.

4.1.2 Anwendungskomponenten

Die Schicht der Anwendungskomponenten stellt standardisierte Methodenschnittstellen zu Verfahren (Methoden, Algorithmen) der Produktentwicklung zur Verfügung.

Visualization Services



Die Visualization-Services realisieren eine anwendungsneutrale Definition von Algorithmen zur immersiven Visualisierung von Bauteilgeometrie und den dazugehörigen Metainformationen aus dem Produktentwicklungsprozess. Die vorliegende Arbeit definiert die Methodenschnittstelle in Abschnitt 4.4 auf Basis eines Szenen-Graph basierten Systems. Es wurde sowohl die Server-Komponente der Visualization-Services als auch eine Client-Komponente mit einer grafischen Benutzungsschnittstelle entworfen, um das Zusammenspiel der Systemkomponenten zu verifizieren. Abbildung 59 zeigt schematisch die konzeptionelle Umsetzung der Visualization-Services.

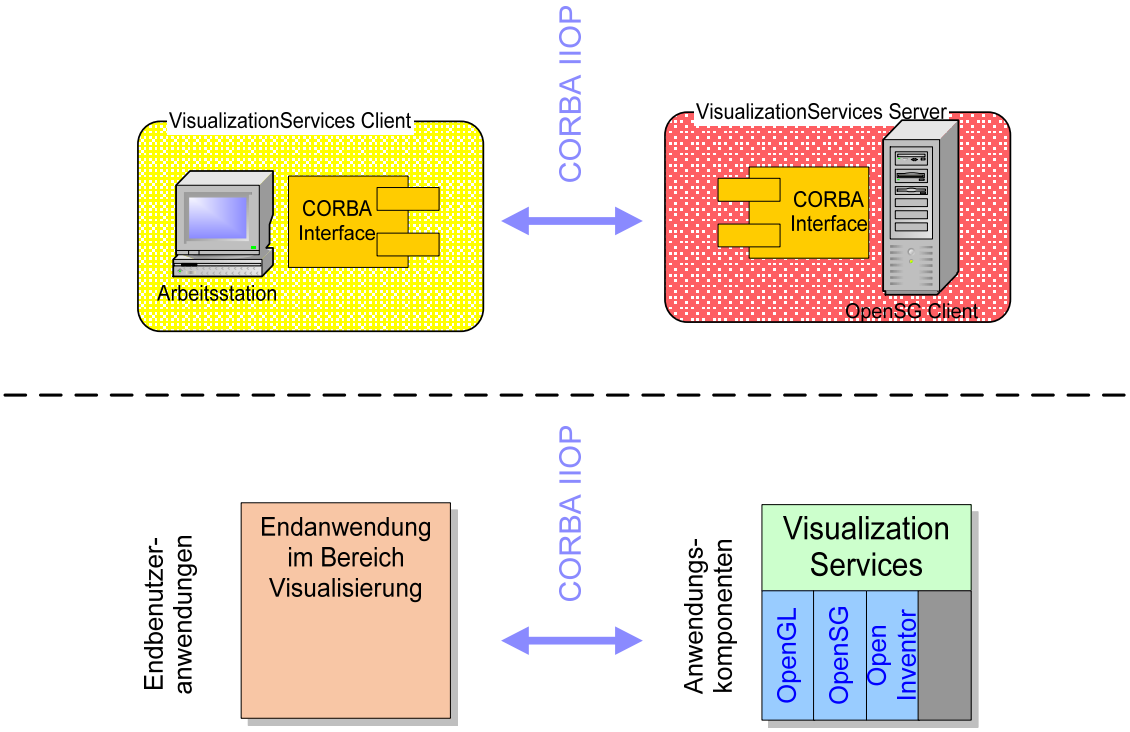


Abbildung 59: Konzeptionelle Umsetzung der Visualization Services
(Trennung von GUI und Funktionalität)

Die Client-Komponente (Desktop-Endbenutzer-Anwendung) übernimmt die Benutzer-Interaktion mit dem Visualization-Services-Server. Die Server-Komponente kann via Multicast-Technologie beliebig viele weitere Server für das Rendering ansteuern, um die Last der Berechungen zu verteilen. Der Szenen-Graph wird jedoch komplett in der Serverkomponente aufgebaut (d.h. auch in den via Multicasting verbundenen Servern), die Kommunikation beschränkt sich jedoch auf Änderungen im Szenen-Graph (z. B. Änderungen der Kameraposition).

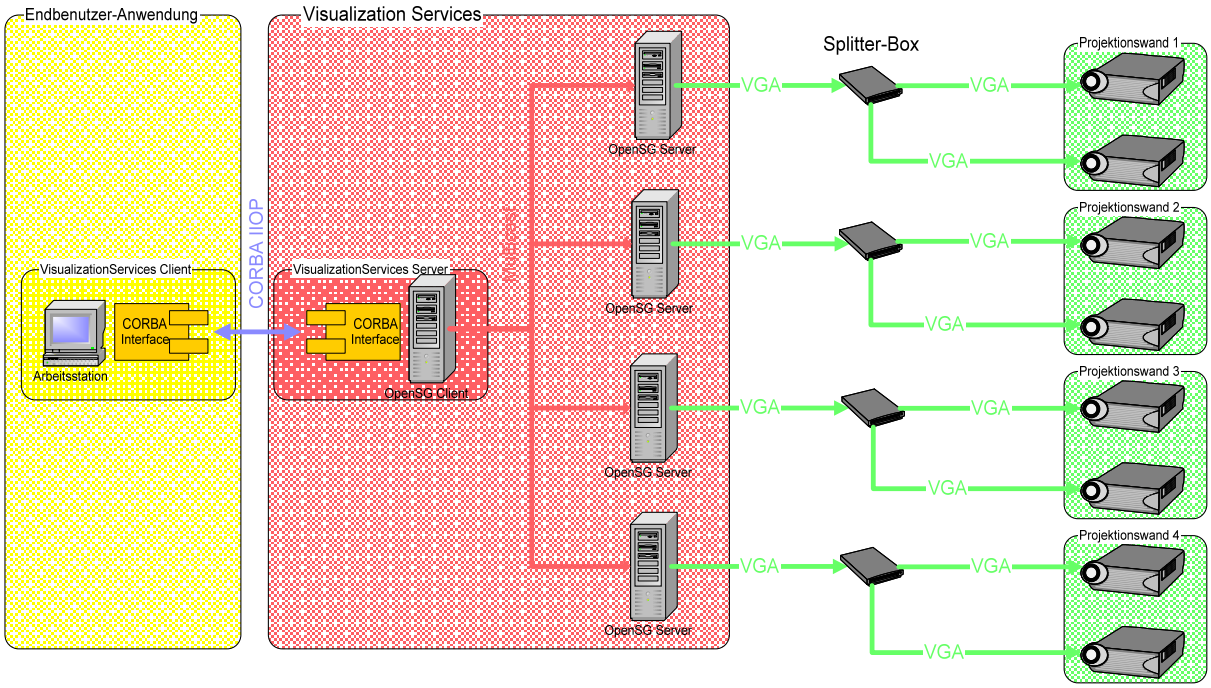
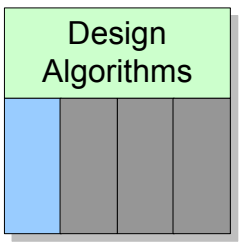


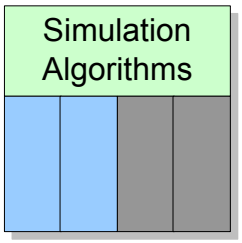
Abbildung 60: Aufbau einer immersiven Projektion mit 4 Projektionsleinwänden (passiv Stereo)

Design Algorithms



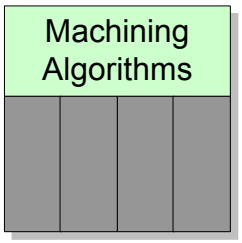
Die Anwendungskomponenten der Design-Algorithms abstrahieren die in der Konstruktionslehre bekannten Algorithmen zur Konstruktion und Grobauslegung von Bauteilen und Zusammenbauten. Hierbei wird unter Design Algorithms weniger das methodische Vorgehen beim Konstruieren als das Erzeugen von Bauteilgeometrie unter Beachtung von vorgegebenen Randbedingungen verstanden.

Simulation Algorithms



Die Simulation-Algorithms-Komponente stellt Methoden zur Simulation des physikalischen Verhaltens von Bauteilen zur Verfügung. Denkbar sind hier Algorithmen aus der Crash- oder Mehrkörpersimulation. Die Abgrenzung zu den CAE-Services ist hier insofern gegeben, als das die CAE-Services Datentypen (z.B. Voxel-Modelle) und Gleichungslöser für spezielle Anwendungen im CAE-Bereich anbieten, die Simulation Algorithms jedoch abstraktere Methoden zur Simulation anbieten. Beispielsweise das Ermitteln von Bewegungsgleichungen (Transformationsmatrizen) aus Zwangsbedingungen des CAD-Modells.

Machining Algorithms



Wichtiger Aspekt der Prozessautomatisierung von CAD zur Fertigung ist, dass unter Bereitstellung von qualitativ hochwertigen CAD-Daten und Technologieinformationen über den speziellen Prozess NC-Daten nahezu vollautomatisch generiert werden können. Beispiel hierfür ist die relativ neue Technik des Pulverauftragsschweißens das mittlerweile bei der Reparatur von teuren Werkzeugen eingesetzt wird.

4.1.3 Endbenutzer-Anwendungen

Oberste Stufe des Application-Frameworks bilden die Endbenutzer-Anwendungen. Sie beinhalten die grafischen Benutzungsschnittstellen und Funktionalitäten, die sich nicht generalisieren und in das Framework einbetten lassen.

Die Testharness stellt eine Referenz-Anwendung dar, die idealerweise alle Komponenten des Frameworks nutzt und deren Anwendung als Beispiel aufzeigt.

4.2 Anwendungsbezogene Informationsvisualisierung

Anwendungsbezogene Informationsvisualisierung hilft dem Benutzer des Visualisierungssystems Informationen angepasst an die jeweilige Aufgabenstellung und in dem Kontext in dem er sich bewegt, intuitiv zu begreifen. Abbildung 61 zeigt ein Bespiel von anwendungsbezogener Informationsvisualisierung aus dem Bereich der Medizin. Hierbei wird die Bestrahlung eines Schädeltumors (Dargestellt als rote Kugel) visualisiert. Abbildung 62 zeigt ein Finite-Elemente Modell eines ganzen Fahrzeugs als Beispiel für anwendungsbezogene Informationsvisualisierung bei der Festigkeitsberechnung (Crash-Simulation).

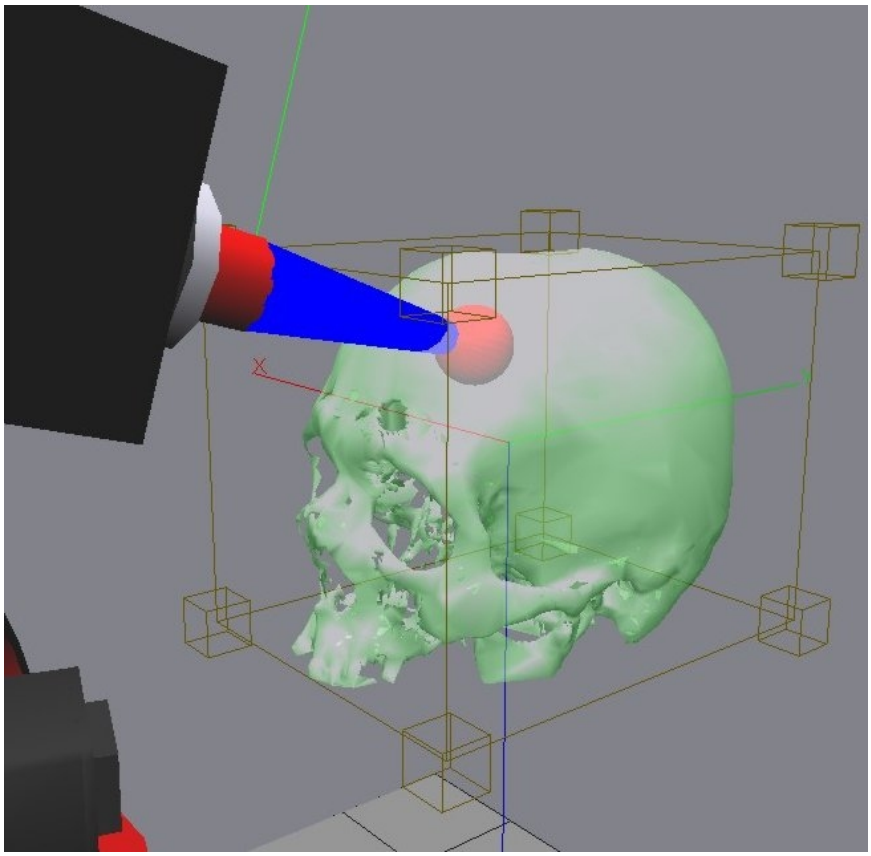


Abbildung 61: Visualisierung einer Bestrahlung eines Schädeltumors [81]

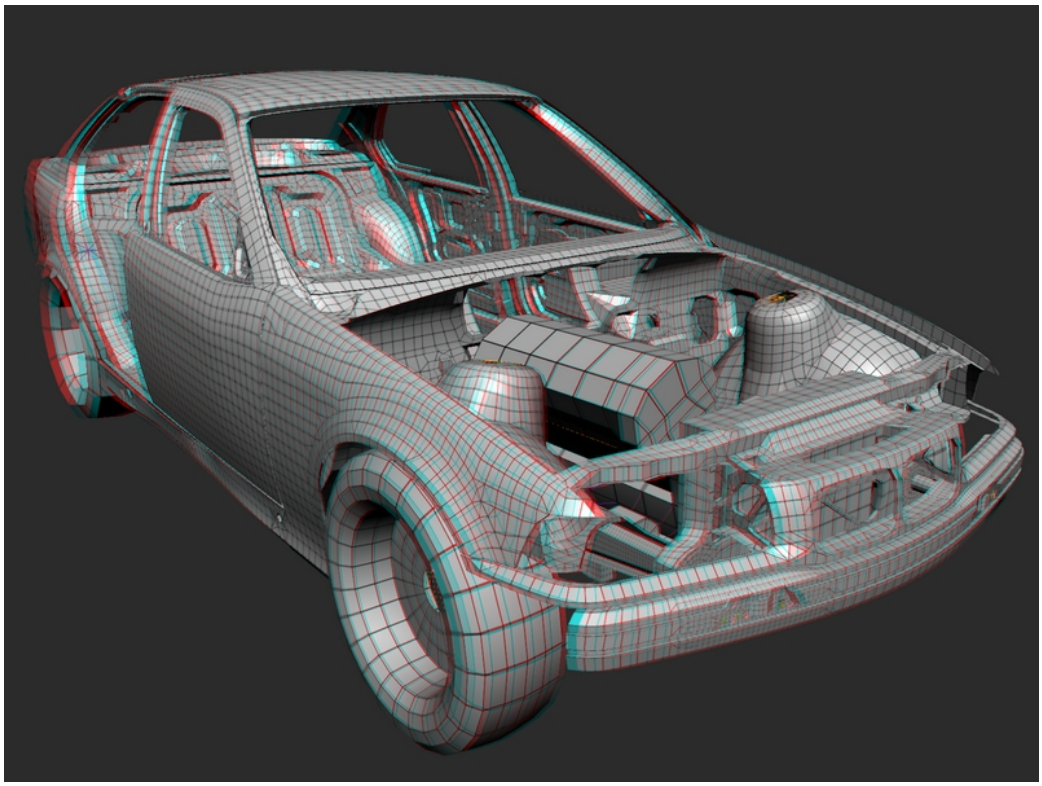


Abbildung 62: Anwendungsbezogene Informationsvisualisierung bei der Crash-Simulation

Auch Prüf- und Messfeatures können grafisch visualisiert werden. Abbildung 63 zeigt einen besonderen Abschnitt eines Strömungskanals mit Toleranzangaben (rote Quadrate) bei der Visualisierung von Qualitätsmerkmalen.

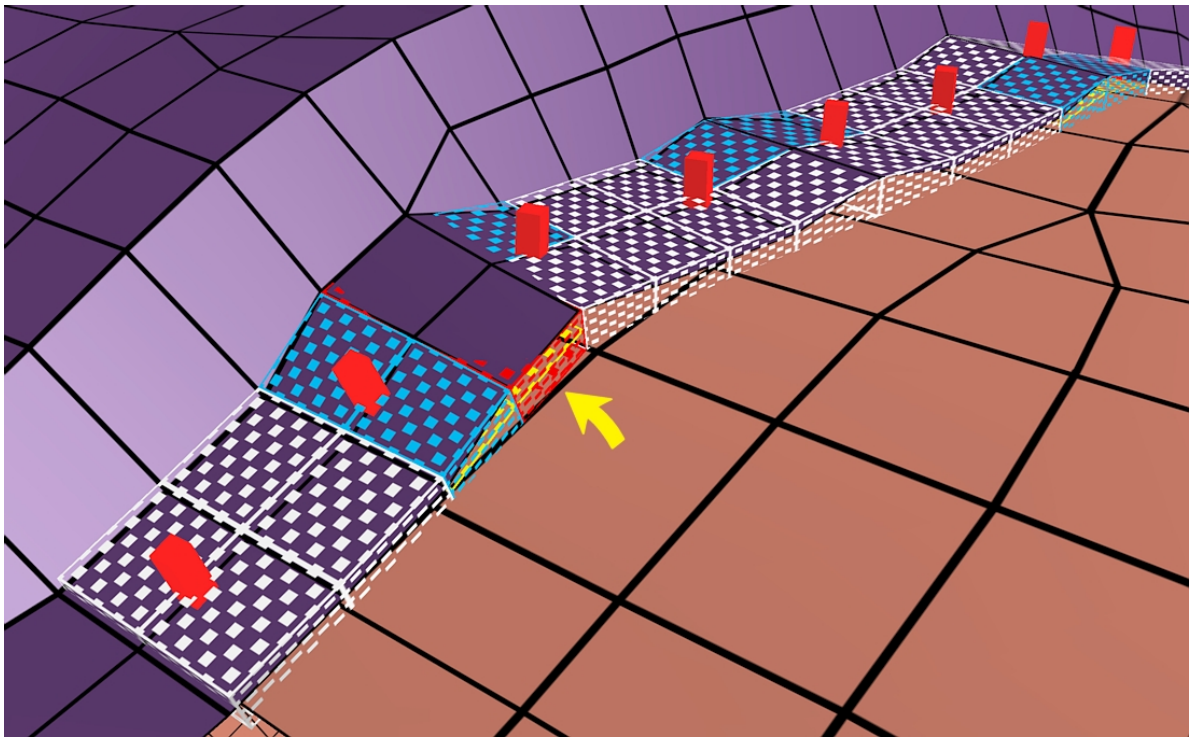


Abbildung 63: Anwendungsbezogene Informationsvisualisierung am Beispiel von Prüf- und Messfeatures

Wir betrachten im Weiteren ausschließlich Informationen, die einen Bezug zur Produktgestalt haben. Selbst wenn anfangs Informationen aus frühen Konstruktionsphasen keinen konkreten Bezug zur effektiven Bauteilgestalt haben, so hat es sich doch als sinnvoll erwiesen, diese Informationen mit einer initialen Produktstruktur zu verbinden. Sobald dann eine Bauteilgestalt analog dieser Produktstruktur ausgeprägt wird, können die damit verbundenen Informationen und ihre Beziehungen untereinander dargestellt werden.

Abbildung 64 zeigt eine Blockstruktur der Informationsflüsse, die es hierbei zu verarbeiten gilt. Eingabe-Datenströme sind zum einen Geometrieinformationen, die direkt aus den CAD-Services Adaptern kommen, zum anderen Metainformationen (z.B. aus frühen Konstruktionsphasen) in Form von XML-Daten (strukturiert in der UMEO-Klassenstruktur). Die Visualisierungs-komponente (VR-Engine) verarbeitet diese Informationen analog der Abbildungsvorschrift des Skriptes und erzeugt die grafische Darstellung anhand einer Grafik-Library.

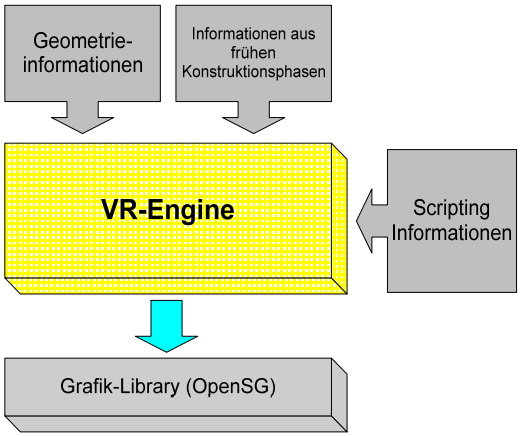


Abbildung 64: Blockstruktur der Visualisierungskomponente

Die genaue Beschreibung der Abbildungsvorschrift, die zu diesem Zweck definiert wurde, wird in Kapitel 4.5 beschrieben. Die CAD-Services (vgl. Kapitel 2.9) erlauben über die standardisierte Methodenschnittstelle einen direkten Zugriff auf die B-Rep Struktur des Bauteils. Um eine eindeutige (persistente) Identifikation von einmal ausgeprägter Geometrie über die Lebenszeit zu ermöglichen, ist jedes geometrische Entity mit einer ID, in Form eines Attributs, versehen. Metainformationen verwenden diese ID, um einen eindeutigen Bezug zur Bauteilgeometrie herzustellen. Diese persistente ID (PID) ist also über Speicher-Lade-Zyklen oder auch Session konstant. Persistente IDs und deren Verwaltung wird mittlerweile von jedem modernen CAD-System über die Programmierschnittstelle angeboten. Im CAD-System sind intern auch die Methoden hinterlegt, die die Strategie der Vergabe von PIDs bei Modellierungsoperationen (z.B. Löschen, Teilen, …) festlegt.

Als Grafik-Library wurden mehrere Systeme untersucht, es stellte sich jedoch heraus, dass Systeme, die auf einer Grafik-Pipeline basieren (z.B. reines OpenGL), nicht geeignet sind. Einzig Szene-Graph basierte Systeme bieten eine Möglichkeit der Einflussnahme auf die darzustellenden Objekte zur Laufzeit, wie sie für diese Übersetzungskomponente notwendig sind. Für die Implementierung der VR-Engine im Verifikationsbeispiel (Kapitel 5.2) wurde OpenSG herangezogen. Die VR-Engine teilt sich in zwei Komponenten, eine Übersetzungskomponente, die die eigentliche grafische Metapher aus der Bibliothek instanziiert und als Teilbaum, über einen Gruppierungs-knoten, zur Verfügung stellt, und eine Visualisierungskomponente, die den Zusammenhang zur Bauteilgeometrie (Lage der Flächen, Abstände, … usw.) bestimmt. Abbildung 65 zeigt den Informationsfluss zwischen den beiden Komponenten:

- Zeiger auf das aktuelle CAD-Services-Entity zum Auslesen des Attributes und Bestimmung von Positionen und Ausrichtungen von Flächen oder Körpern.

- Zeiger auf das bis zu dieser Phase des Methodenaufrufs bestehende OpenSG-Modell zur Berechnung von Intersektionen.

- Zeiger auf den aktuell übersetzten OpenSG Knoten zur Veränderung dieser Substruktur.

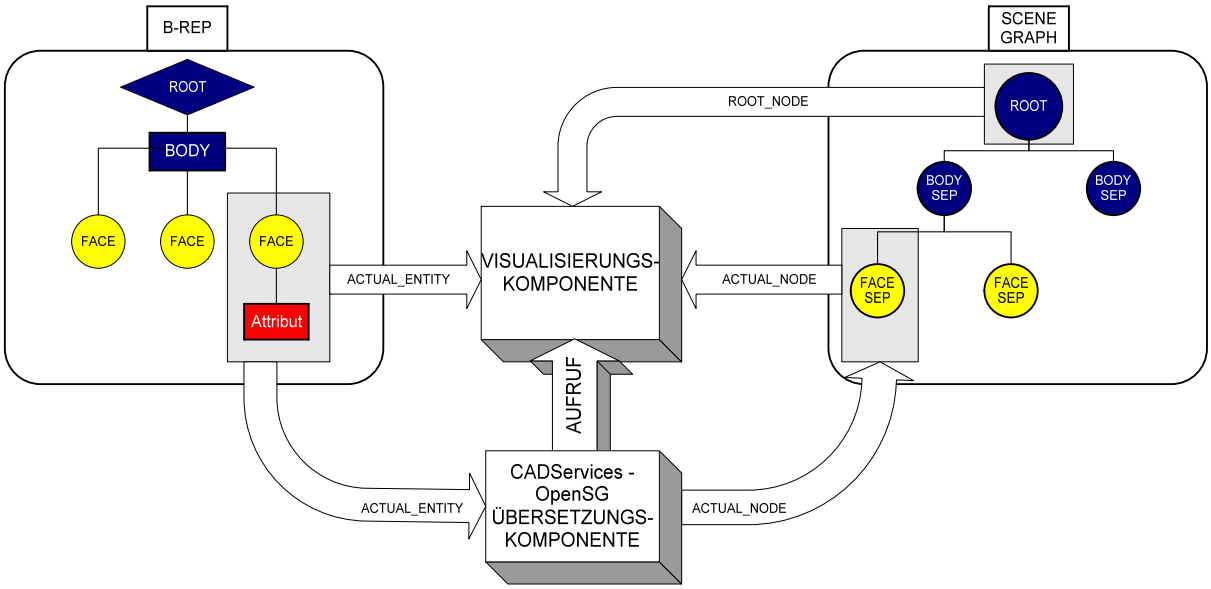


Abbildung 65: Kontrollfluss in der Visualisierungskomponente (VR-Engine)

4.3 Grafische Metaphern für Informationen

Das Konzept zur grafischen Darstellung von Informationen setzt die Existenz grafischer Metaphern voraus, die dem Betrachter die Information intuitiv vermitteln. Betrachter können hierbei aus völlig verschiedenen Domänen kommen, um sich in Teams zusammenzufinden und einen Sachverhalt zu diskutieren. Um die Leistungsfähigkeit dieses pragmatischen Ansatzes zu verdeutlichen, wurden zwei typische Informationsmengen (Produktanforderungen und Features) aus der Produktentstehung gewählt und für unterschiedliche ausgewählte Sichten (Konstruktion, Fertigung, Montage, Prüfung, Vertrieb) grafische Metaphern entworfen.

Die im Rahmen dieser Arbeit entwickelte Menge an grafischen Metaphern stellt lediglich *eine mögliche* Ausprägung an grafischen Metaphern dar. Viele weitere auf den Anwendungsfall spezialisierte Metaphern sind möglich. Soll nun ein anderer Satz an grafischen Metaphern zum Einsatz kommen, müssen diese zunächst modelliert werde. Ein einfacher Weg dies zu tun ist VRML. Diese VRML Modelle werden in einem zweiten Schritt in einer Datenbank (z.B. mysql) als BLOB[51] gespeichert. Nun kann über das Abbildungsskript auf die grafischen Metaphern

[51] BLOB (engl. Binary large object) Objekt einer Datenbank bestehend aus binären Daten

zugegriffen werden. Diese Möglichkeit der benutzerspezifischen Erweiterbarkeit schafft im konkreten Einsatz im industriellen Umfeld die Möglichkeit innerhalb einer Firma maßgeschneiderte Datenbanken mit spezialisierten grafischen Metaphern anzubieten.

4.3.1 Produktanforderungen

Abbildung 66 zeigt die Klassifikation und graphische Darstellung von Produktanforderungen. Für die verschiedenen Phasen der Produktentstehung wurden verschiedene graphische Metaphern entwickelt, die auf intuitive Weise die Informationen repräsentieren sollen. Sie sind in der Matrix von links nach rechts in einer Spalte angeordnet. In den Zeilen von oben nach unten finden sich die verschiedenen Typen von Anforderungen, die bei der Produktenstehung von Relevanz sind. Für einige Typen machte es keinen Sinn, unterschiedliche grafische Repräsentationen zu definieren, weil die verwendeten Metaphern phasenübergreifend verwendet werden können. Parametrisierte Templates dieser grafischen Metaphern wurden im Rahmen dieser Arbeit implementiert und in einer Softwarebibliothek zusammengefasst. Diese Templates können nun gemäß der Abbildungsvorschrift des XML-Scripts instanziiert und in den Szene-Graph eingefügt werden.

Abbildung 67 bis Abbildung 73 zeigen die grafischen Metaphern für die verschiedenen Feature-Typen in Abhängigkeit von der Klasseneinteilung (Tabelle 11).

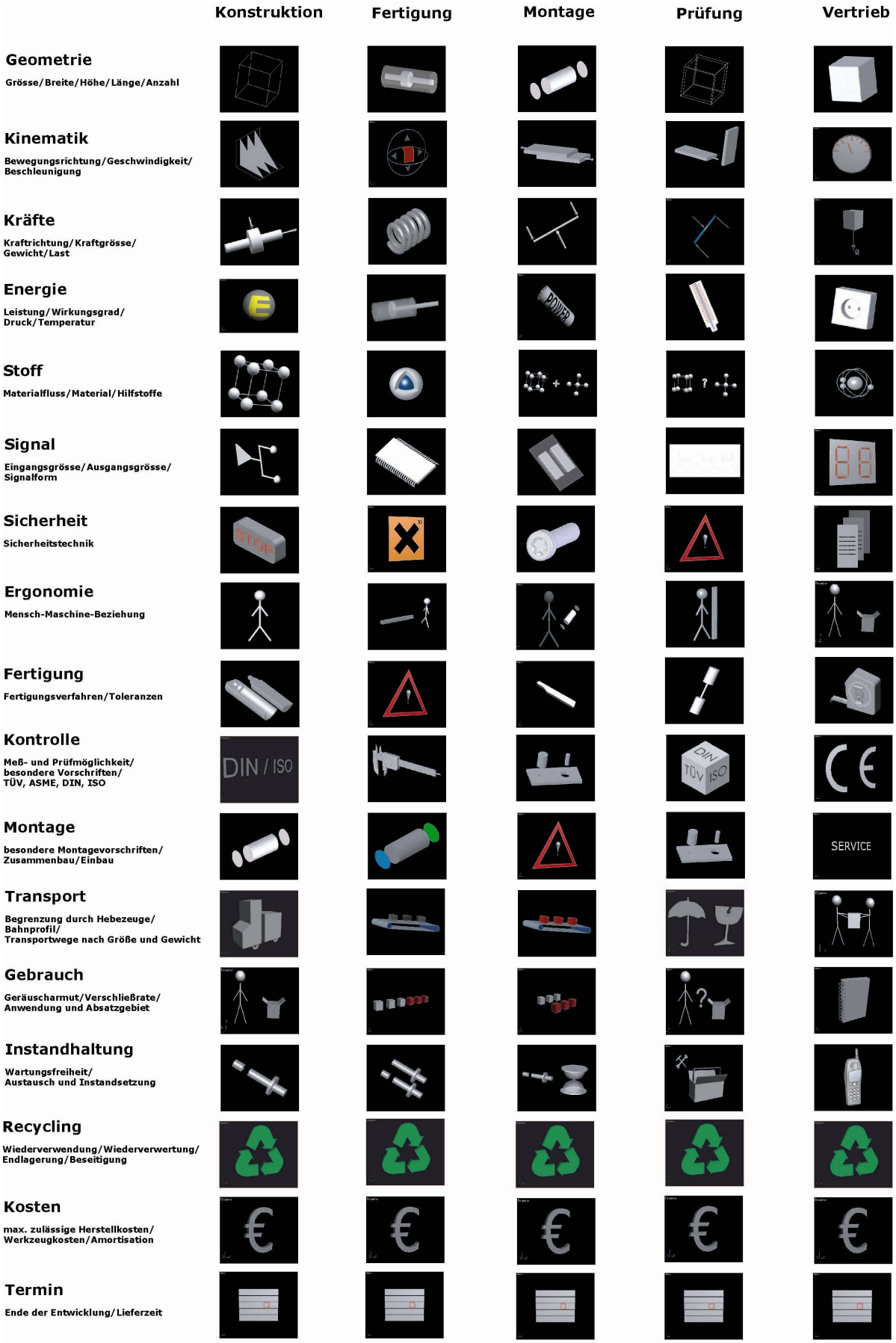


Abbildung 66: Graphische Darstellung von Produktanforderungen

4.3.2 Features

Analog den Anforderungen wurden für Features grafische Metaphern zur Informationsvisualisierung entwickelt. In Tabelle 11 wird eine für die Visualisierung zweckmäßige Klassifikation von Feature-Elementen vorgestellt.

Solid/Flächen Modelle	Rotationssymmetrisch	Gestalterzeugende Elemente	Zylinder
			Kegel
			Rotierender Schnitt
			Durchgangsbohrung
			Gewindebohrung
			Stufenbohrung
			Sackbohrung
			Feder
			Außengewinde
			Schraube
			Mutter
			Sicherungsring
			Lager
			Scheibe
			Bolzen
			Stift
			Flansch
			Niete
			Zahnrad
		Gestaltverändernde Elemente	Fase
			Verrundung
			Freistich
			Formschräge-Element
	Prismatisch	Gestalterzeugende Elemente	Quader
			Nut
			Tasche
			Dünnwandiges Element
		Gestaltverändernde Elemente	Fase
			Verrundung
			Variable Verrundung
			Formschräge-Element
Freiformflächenelemente		Gestalterzeugende Elemente	Rippe/Versteifung
			Schalle
			Freiformgrundelement
			Freiform-Ausschnitt
		Gestaltverändernde Elemente	Sicke/Rille
			Bördeln

Tabelle 11: Klassifikation von Feature-Elementen für die Visualisierung

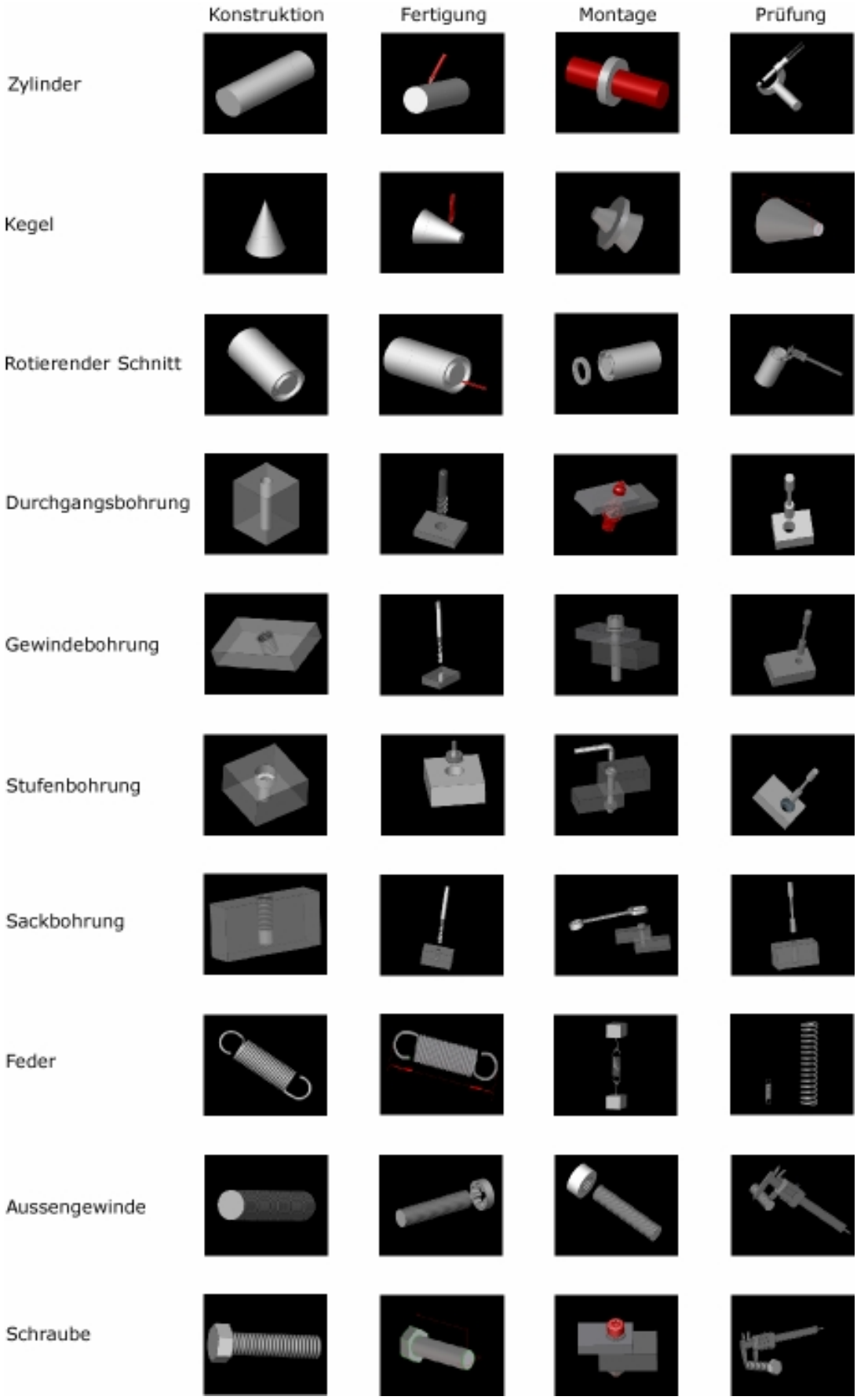


Abbildung 67: Graphische Metaphern für gestalterzeugende Elemente rotationssymmetrischer Solid/Flächen-Modelle 1/2

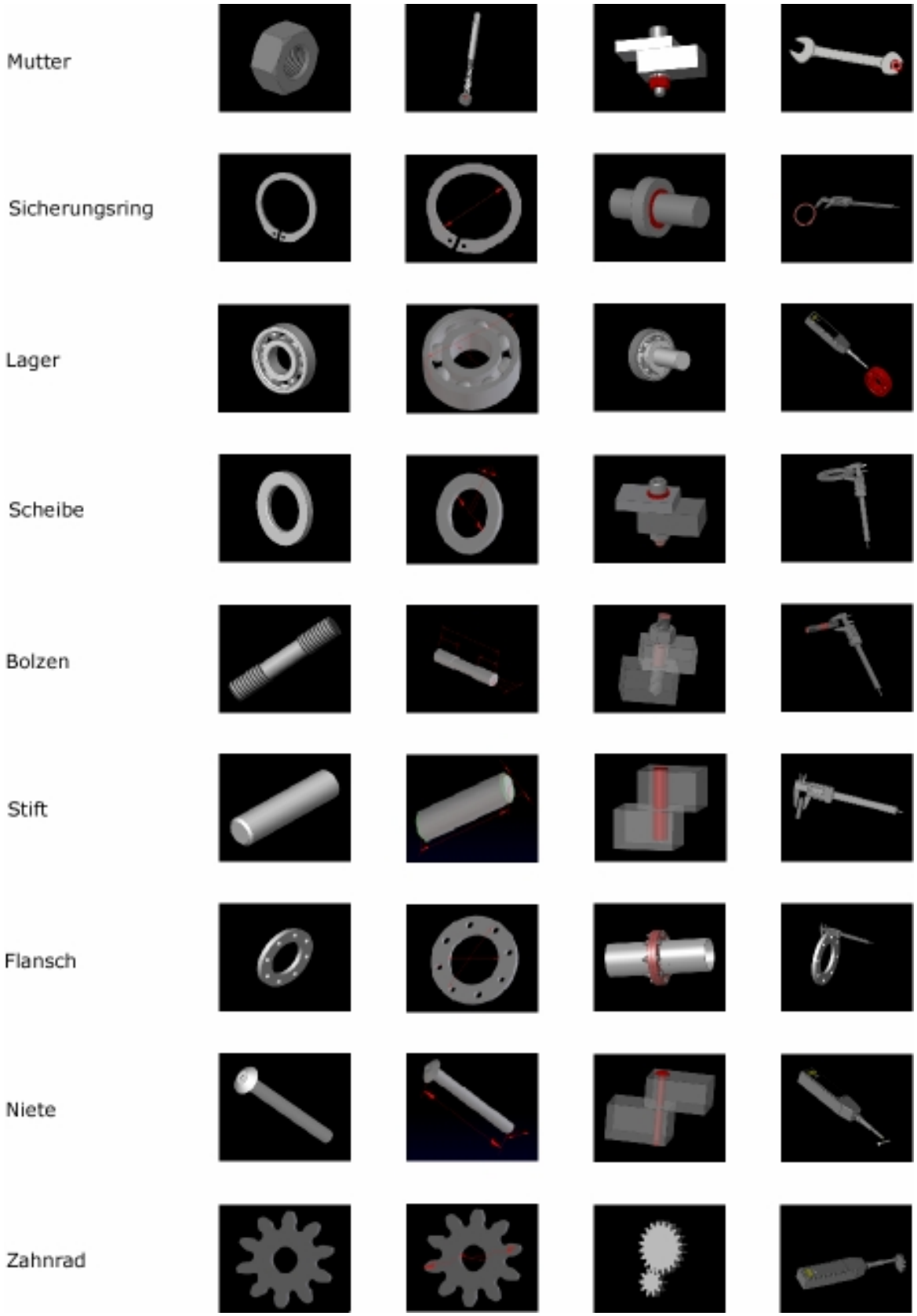


Abbildung 68: Graphische Metaphern für gestalterzeugenden Elemente rotationssymmetrischer Solid/Flächen-Modelle 2/2

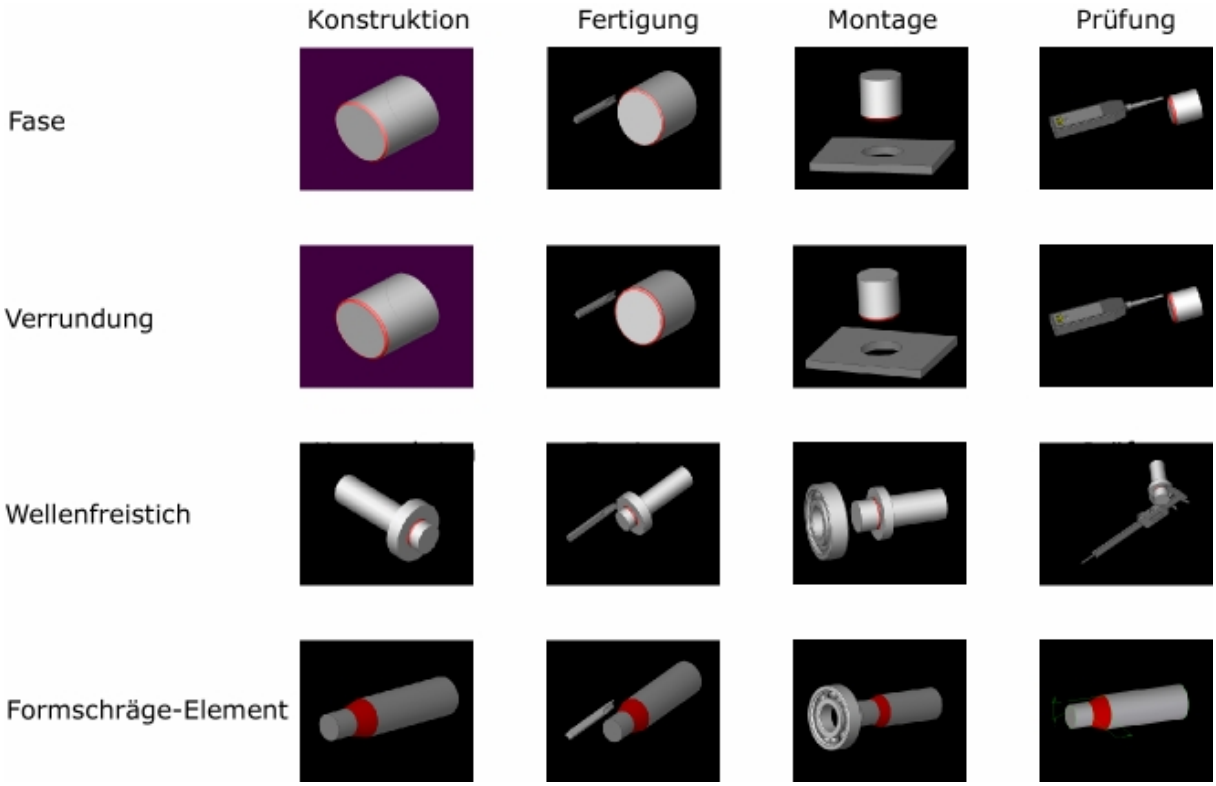


Abbildung 69: Graphische Metaphern für gestaltverändernde Elemente rotationssymmetrischer Solid/Flächen-Modelle

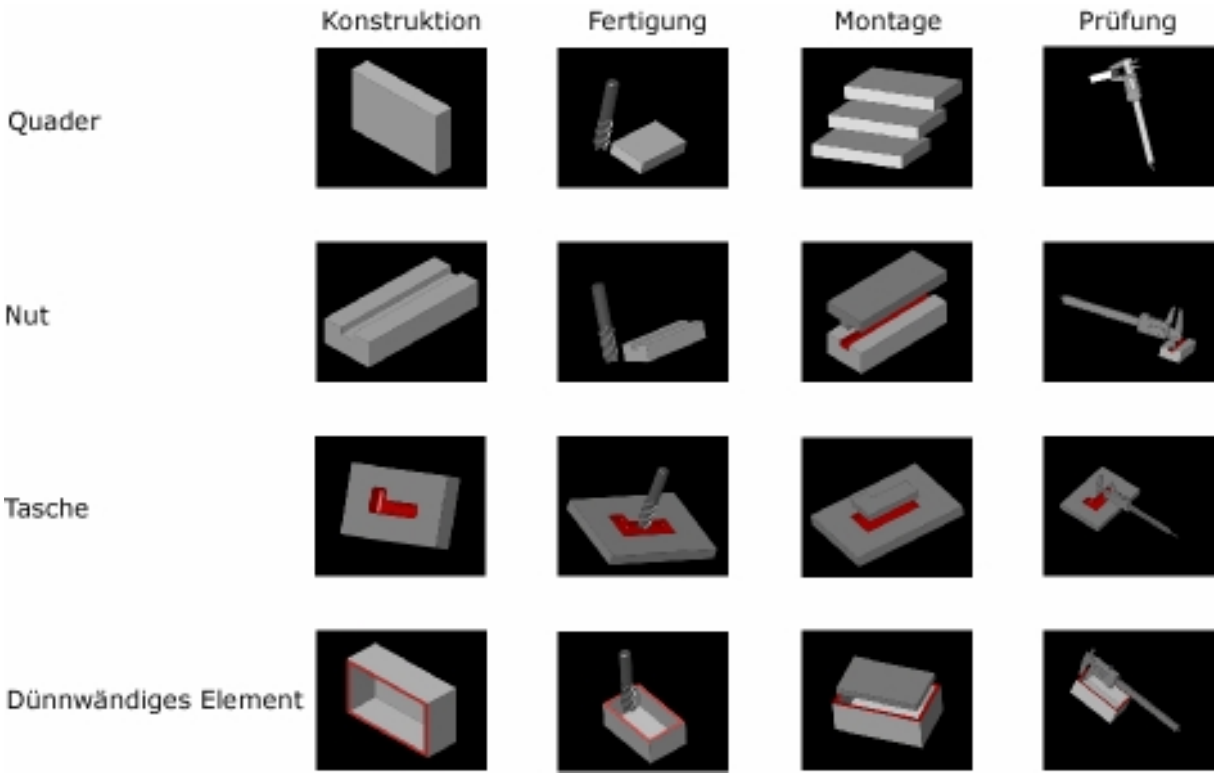


Abbildung 70: Graphische Metaphern für gestalterzeugende Elemente prismatischer Solid/Flächen-Modelle

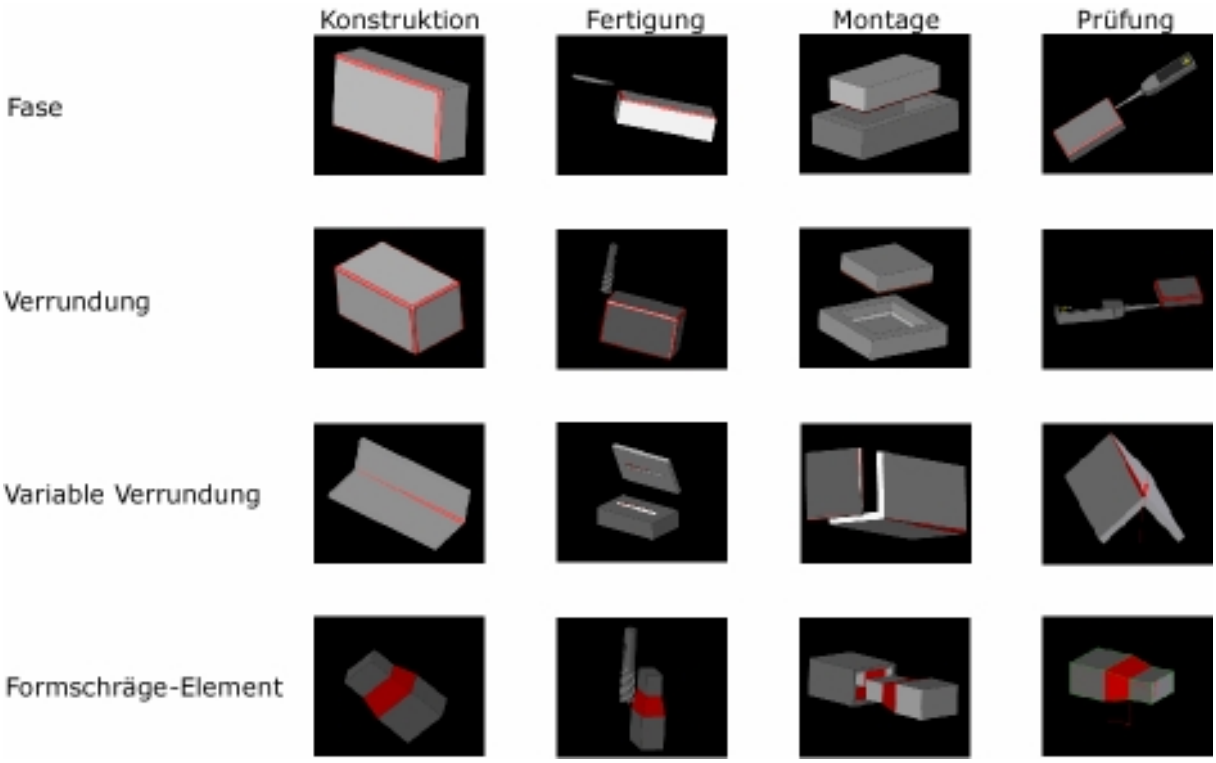


Abbildung 71: Graphische Metaphern für gestaltverändernde Elemente prismatischer Solid/Flächen-Modelle

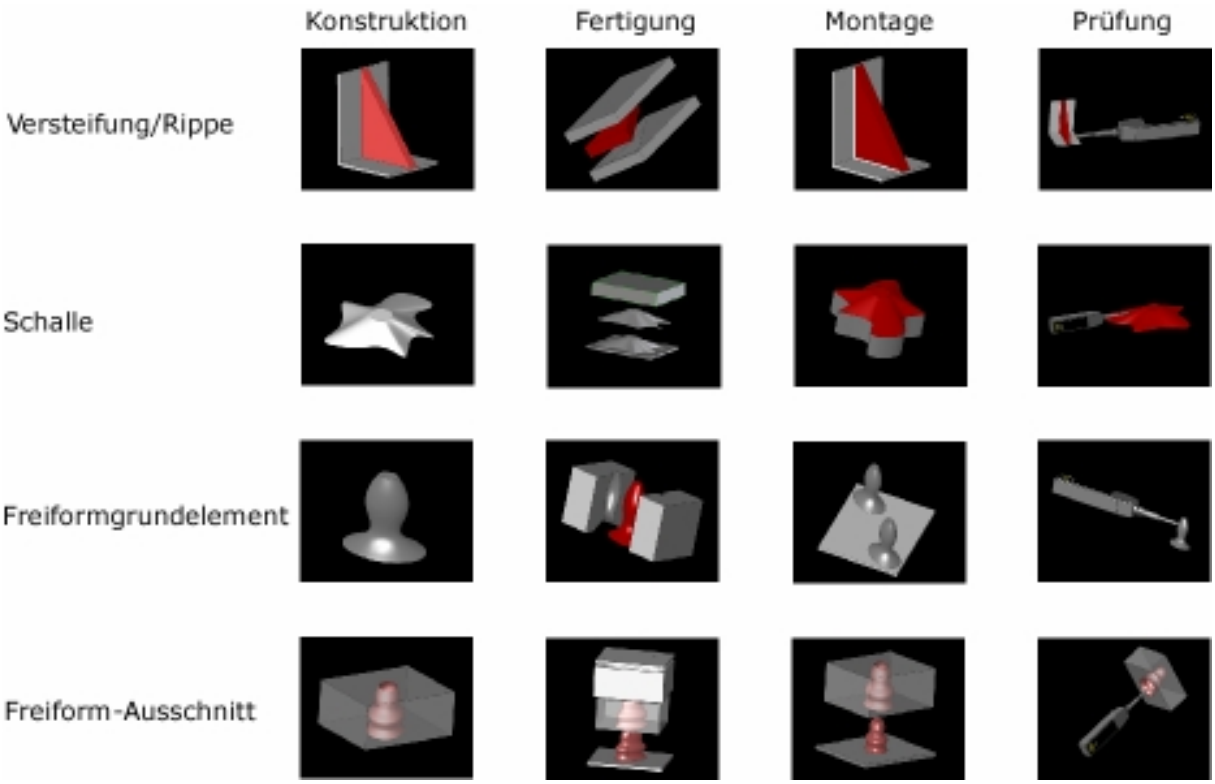


Abbildung 72: Graphische Metaphern für gestalterzeugende Elemente der Freiformflächen-Modelle

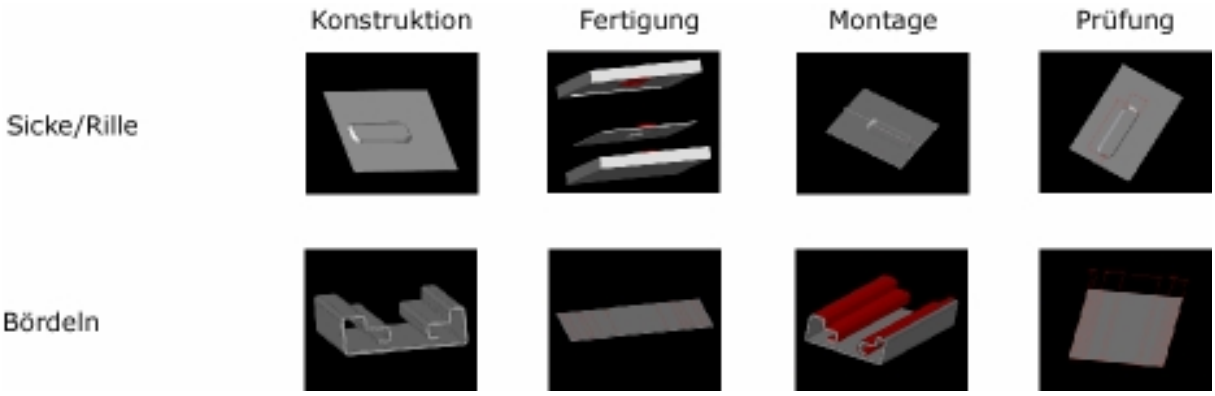


Abbildung 73: Graphische Metaphern für gestaltverändernde Elemente der Freiformflächen-Modelle

4.3.3 Instantiierung der graphischen Metaphern in Szenen-Graph basierten Systemen

Bei der Instantiierung der graphischen Metaphern stellen sich folgende grundsätzliche Anforderungen an die Abbildung:

- Variable Zuweisung und Darstellung einer oder mehrerer Visualisierungen pro Anforderung
- Variable Zuweisung einer oder mehrerer verändernder Funktionen auf vorhandene Strukturen
- Variable Zuweisung einer oder mehrerer verändernder Funktionen auf neue Visualisierungen
- Sichtbare Referenzierung der Visualisierungen zu Attribut-tragenden geometrischen Ausprägungen
- Verhindern von Überschneidungen oder Überlappungen neuer Visualisierungen mit bereits Erstellten oder mit dem Modell
- Benutzergesteuertes Verhalten der Visualisierungen

Abbildung 74 beschreibt den Prozess der Abbildung an sich. Ein Iterator[52] traversiert den gesamten B-Rep Baum, der in einem ersten Schritt mit Attributen angereichert worden ist. Die Attribute beinhalten sowohl Klasse als auch Instanz der Metainformation (z.B. der Anforderung). Die Übersetzungskomponente evaluiert den geometrischen Teilbaum des B-Rep und erzeugt auf der Szene-Graph Seite eine äquivalente tessellierte Darstellung. Die IDs der Attribute werden mit übergeben, um eine Bidirektionalität zu erreichen. Ist an einem geometrischen Entity ein Attribut

[52] Unter einem Iterator versteht man eine Methode die von oben nach unten und von links nach rechts einen Baum durchwandert und dabei jeden Knoten besucht

vorhanden, ermittelt die Methode anhand der Abbildungsvorschrift die zugehörige grafische Metapher und baut sie analog der Abbildungsvorschrift zusammen mit dem Geometrieknoten in den Szene-Graphen ein.

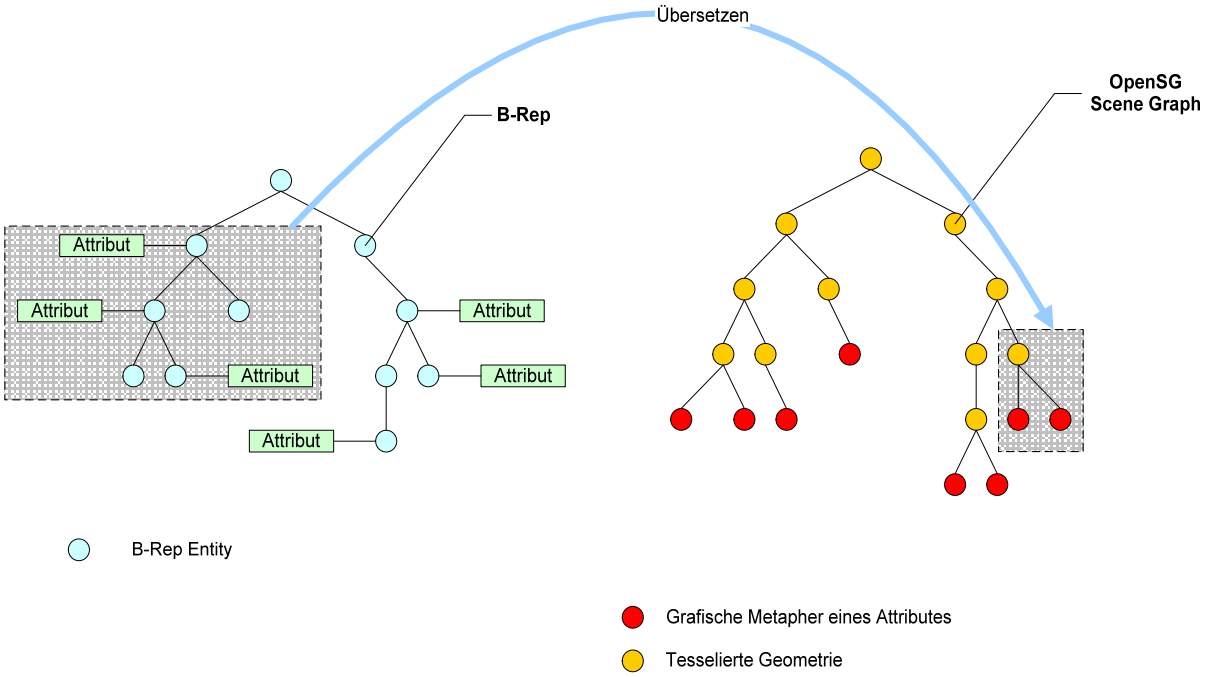


Abbildung 74: Instantiierung der grafischen Metaphern im Szene-Graph

4.4 Schnittstellendefinition des Visualization-Services-Moduls

Eine der zentralen Komponenten dieser Arbeit ist die Definition und Entwicklung einer Methodenschnittstelle zu einem Szene-Graph basierten Visualisierungssystem, das sich in die in Abbildung 55 vorgestellte Infrastruktur nahtlos integrieren lässt. Hierbei soll analog den in Abschnitt 2.9 vorgestellten CAD-Services eine Standardisierung über die OMG erfolgen. Eine prototypische Implementierung (vgl. 5.2) auf Basis von OpenSG wurde im Rahmen dieser Arbeit geleistet. Die kompletten Quelltexte der IDL-Spezifikation finden sich im Anhang A. Die großformatigen UML-Diagramme, mit den Abhängigkeiten der Interfacedefinitionen untereinander, befinden sich in Anhang B.

Die Schnittstellendefinition teilt sich in verschiedene Module auf. Das Modul VizConnect beinhaltet die Funktionalität, eine Instanz des Visualisierungssystems zu erzeugen und sich mit diesem netzwerkbasiert zu verbinden. Im Modul VizMain sind alle grundlegenden Interfaces des Objektraumes, des Szene-Graphen und des Fenstersystems definiert. Die Elemente des Szene-Graph-Systems, die Knoten (Nodes), sind im Modul VizNode definiert. Geometrische Definitionen finden sich im Modul VizGeometry und die Interface Definitionen für Materialien

und Ein- / Ausgabegeräte in den Modulen VizMaterial und VizDevice. Das Interface des Scripting Prozessors, der die Umsetzung der Meta-Informationen in grafische Metaphern leistet, findet sich im Modul VizScripting. Abbildung 76 zeigt die Vererbungsstrukur der Interface-Definitionen. In den Kapiteln 4.4.1 bis 4.4.7 werden die einzelnen Interfaces, die zu entwickeln waren, genau beschrieben.

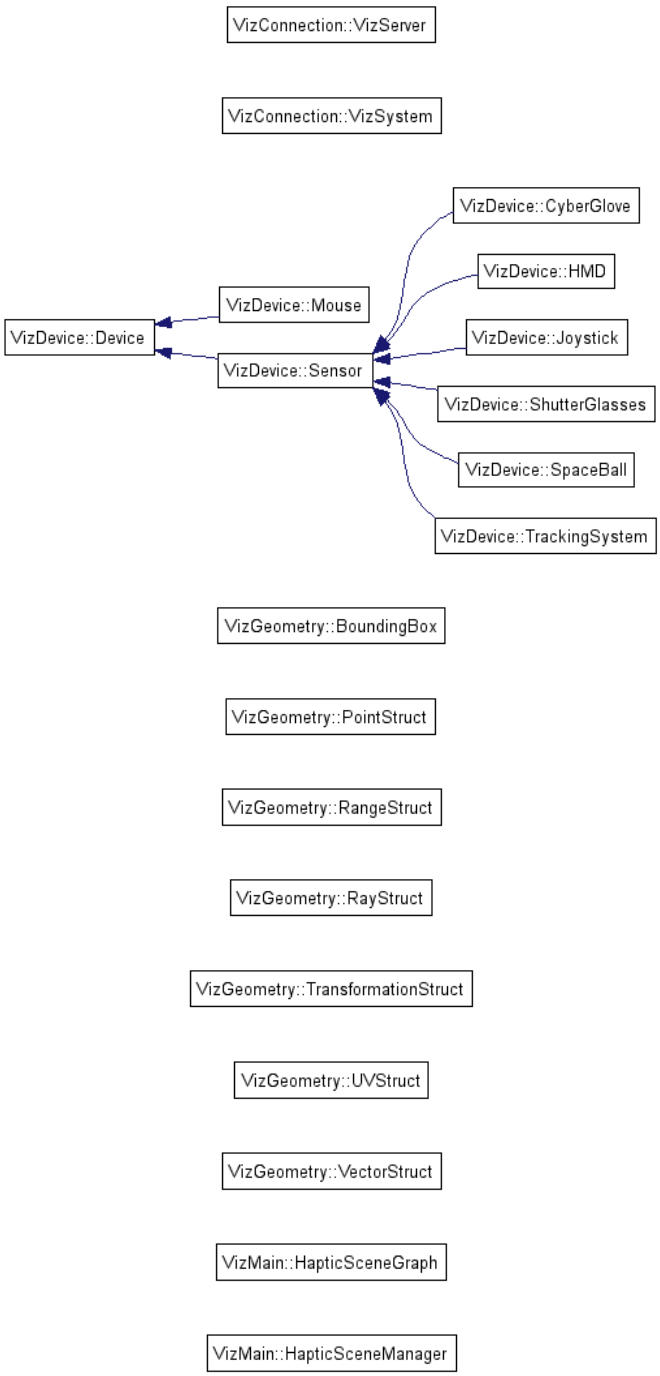


Abbildung 75 : Grafische Darstellung der Interface Struktur (Forsetzung in Abbildung 76)

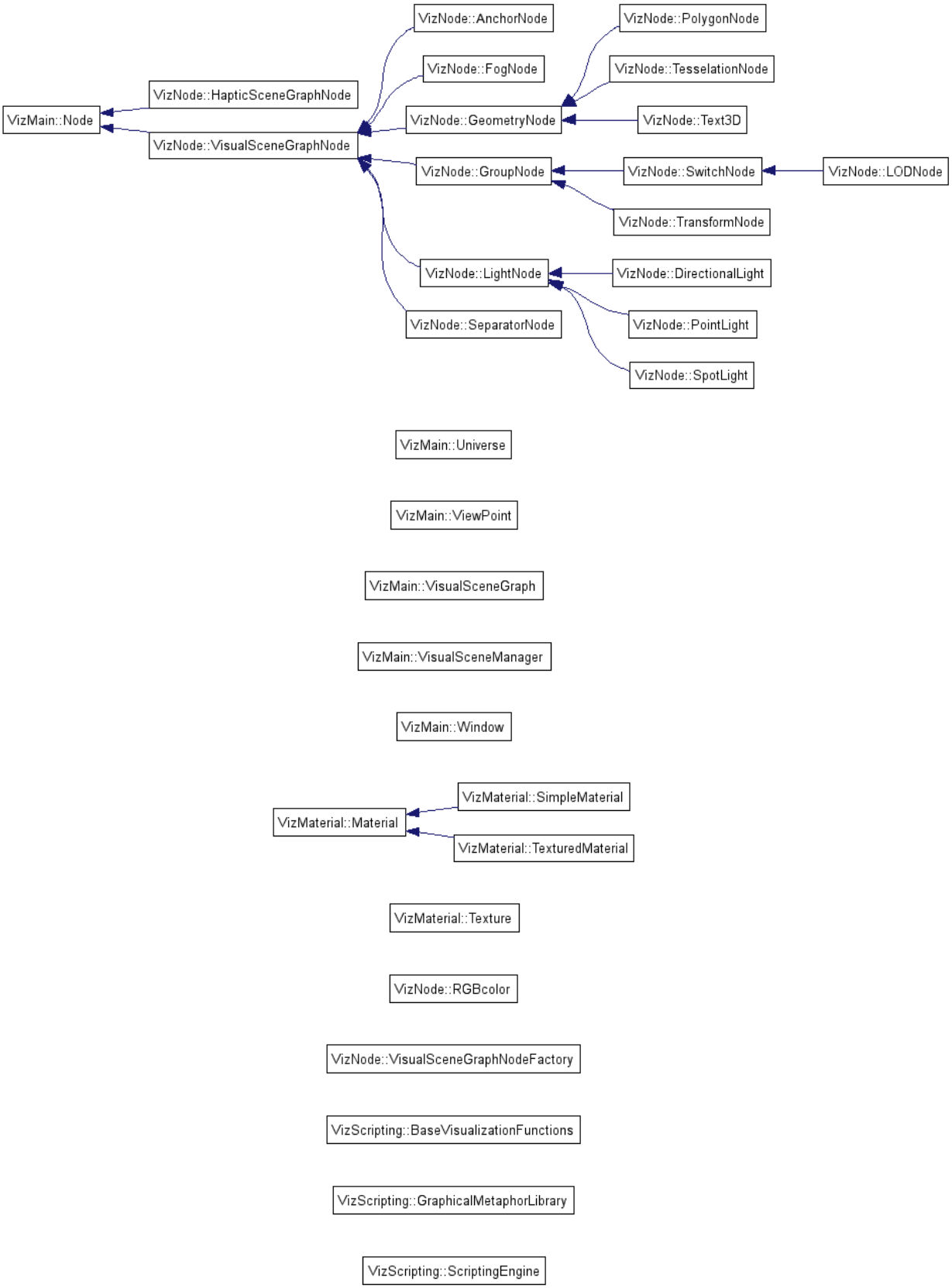


Abbildung 76: Grafische Darstellung der Interface-Struktur (Forsetzung)

4.4.1 VizConnection Modul

Das Modul VizConnection vereinigt alle Interfaces, die zur Verbindungsaufnahme mit dem Visualisierungssystem notwendig sind. Es sind zwei Interfaces `VizServer` und `VizSystem` definiert (siehe Abbildung 78).

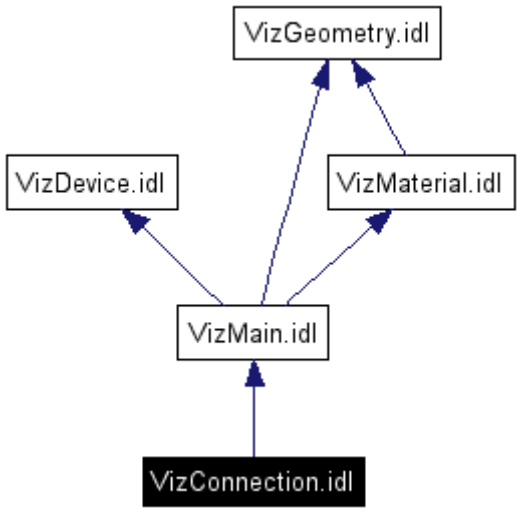


Abbildung 77: Abhängigkeitsgraph des Moduls VizConnection

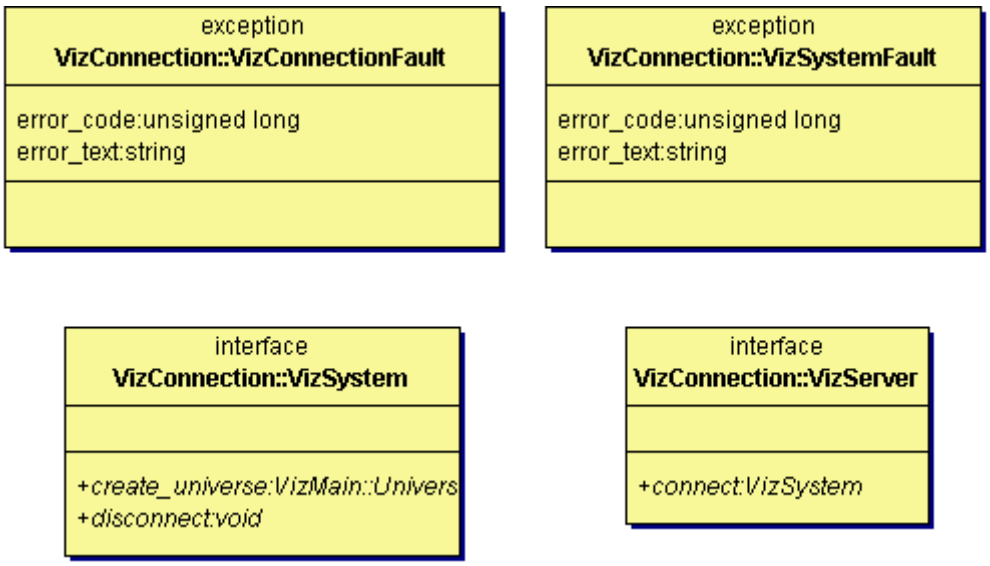


Abbildung 78: Interface-Definitionen des VizConnect Moduls

4.4.1.1 *VizServer*

```
interface VizServer {
  /**
   * @idl-raises VizConnection::VizConnectionFault
   */

    VizSystem connect() raises(VizConnectionFault);
};
```

Analog der CAD-Services Implementierung (vgl. Anhang E) ist das `VizServer` Interface das erste Objekt, dessen Referenz über den ORB bestimmt wird. Die Methode `connect` gibt dann eine Referenz auf die spezifische Implementierung des Visualisierungssystems zurück.

4.4.1.2 *VizSystem*

```
interface VizSystem {
  VizMain::Universe create_universe();

  /**
   * close system and clean-up
   * @idl-raises VizConnection::VizConnectionFault
   */

   void disconnect() raises(VizConnectionFault);
};
```

Das Interface `VizSystem` ist ein Interface zu einer Instanz eines Visualisierungssystems. Mittels dieses Interfaces kann ein neues Universum (`create_universe`) (Gesamtheit aller zu einer Visualisierung gehörigen Objekte) erzeugt werden. Nach Beendigung des Visualisierungsvorganges sorgt die Methode `disconnect` dafür, dass alle lokalen Datenstrukturen freigegeben werden und die Verbindung terminiert wird.

4.4.2 VizMain Modul

Das Modul VizMain beinhaltet alle Interfaces zu den Kontrollstrukturen des Visualisierungssystems. Das System besteht aus einem haptischen und einem visuellen Szene-Graphen mit den dazugehörigen Managern. Manager sind in diesem Zusammenhang Interfaces die auf Interaktion des Benutzers reagieren und eine Änderung zu Folge haben. Einfaches Beispiel wäre ein Maus-Event (z.B. heranzoomen der Szene) das eine Veränderung des Viewpoints der Szene zur Folge hat, d.h. der Szenen-Manager wacht über die Verknüpfung von Benutzer-Events (kommend aus den Devices, vgl. 4.4.6) mit Änderungen im Szenen-Graphen. Das Interface `Node` ist ein „High-level“ Interface für alle Knoten, die die Grundbestandteile der Szene-Graphen sind. Abbildung 79 zeigt den Abhängigkeitsgraphen des `VizMain` Moduls. `VizMain` exportiert Interfaces nach `VizDevice` und `VizMaterial` und importiert Interfaces aus `VizNode` und `VizConnection`.

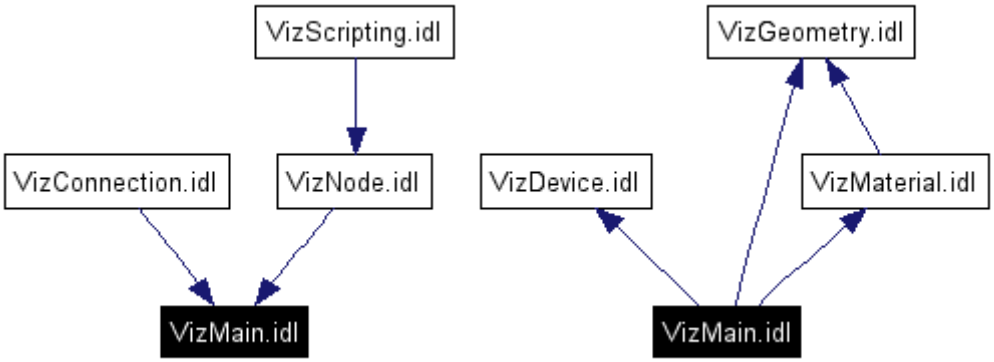


Abbildung 79: Abhängigkeitsgraphen des Moduls VizMain

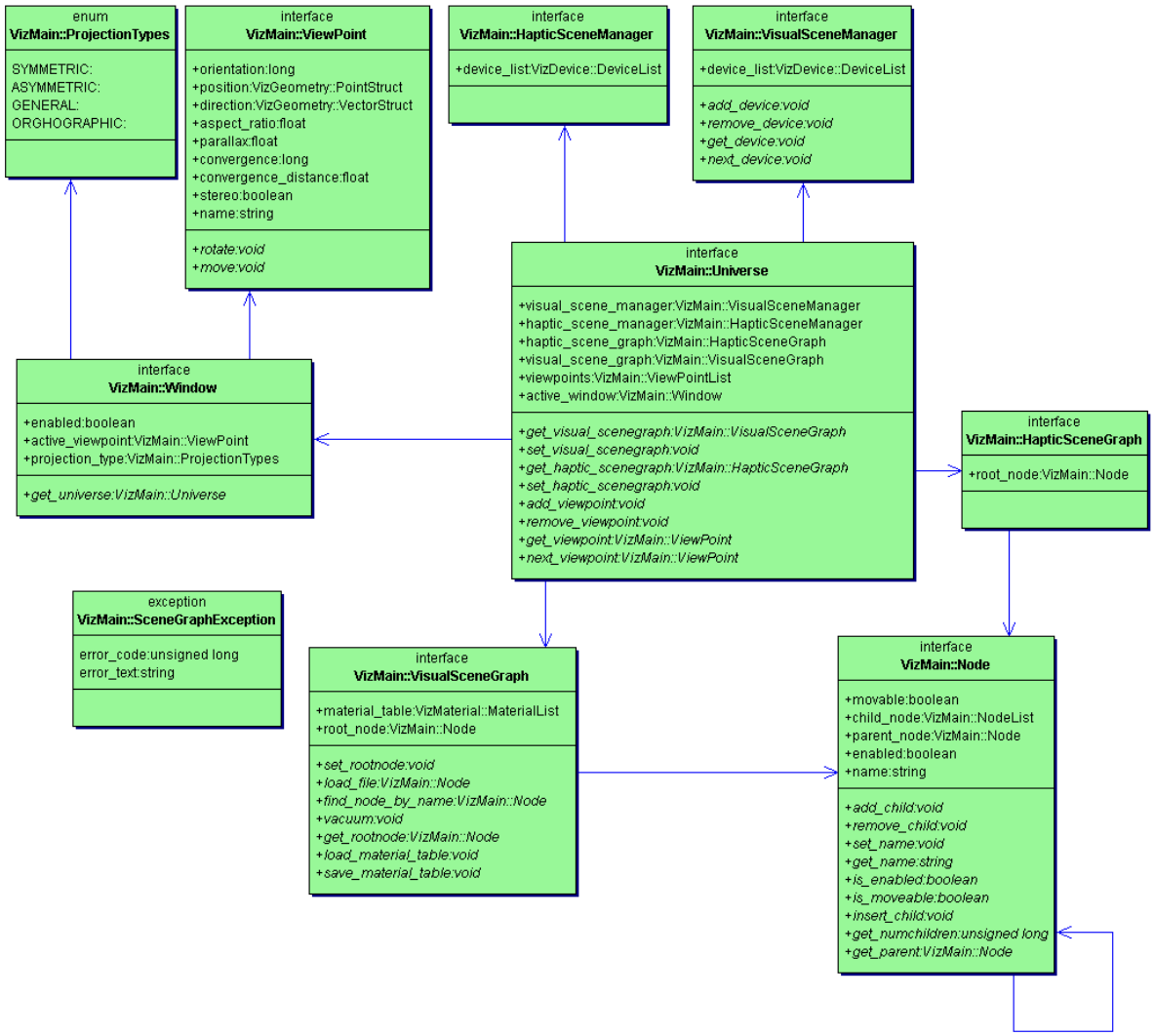


Abbildung 80: Interface Definitionen des VizMain Moduls

4.4.2.1 *HapticSceneGraph*

```
interface HapticSceneGraph {
```

```
        attribute VizMain::Node root_node;
    };
```

Das Interface `HapticSceneGraph` beinhaltet den sog. Root-Knoten des haptischen Szene-Graphen. Dies ist der Knoten, der in der hierarchischen Struktur keine Eltern-Objekte mehr besitzt.

4.4.2.2 HapticSceneManager

```
    interface HapticSceneManager {

        attribute VizDevice::DeviceList device_list;
    };
};
```

Der zugehörige `HapticSceneManager` verwaltet die Liste der Ein-/Ausgabegeräte, die aktiv mit diesem haptischen Graph verbunden sind. Beeinhaltet diese Liste z.B. ein Force-Feedback Device, dann reagiert der haptische ScenenGraph auf dieses Device. SceneManager definieren die Reaktion (Veränderung) des Szene-Graphen auf eine Benutzer-Interaktion.

4.4.2.3 Node

```
    interface Node {
        void add_child(in VizMain::Node child) raises(SceneGraphException);

        void remove_child(in VizMain::Node child) raises(SceneGraphException);

        void set_name(in string aName);

        string get_name();

        boolean is_enabled();

        boolean is_moveable();

        void insert_child(in unsigned long position, in boolean child)
           raises(SceneGraphException);

        unsigned long get_numchildren();

        VizMain::Node get_parent() raises(SceneGraphException);

        attribute boolean movable;
        attribute VizMain::NodeList child_node;
        attribute VizMain::Node parent_node;
        attribute boolean enabled;
        attribute string name;
    };
```

Knoten sind die Grundelemente des Szenen-Graphen. Das Interface `Node` definiert eine abstrakte Oberklasse zu allen Typen von Knoten, die innerhalb eines haptischen oder visuellen Szene-Graphen vorkommen können.

Attribute

`movable`	Ein Flag (Schalter) der bestimmt, ob der Knoten innerhalb des Szene-Graphen verschiebbar ist.
`child_node`	Liste von Knoten der Kind-Objekte. Wird mit `parent_node` zusammen dazu verwendet, die hierarchische Baumstruktur des Szene-Graphen aufzubauen.
`parent_node`	Übergeordneter Knoten.
`enabled`	Ein Flag (Schalter) der bestimmt, ob der Knoten von der Rendering Engine beim traversieren des Szene-Graphen als existent betrachtet wird. Dieser Flag kann verwendet werden, um einen Knoten vorübergehend abzuschalten ohne ihn aus dem Baum zu entfernen. Dieser Flag hat keine Auswirkungen auf alle Kind-Objekte des Knotens, es sei denn es handelt sich um einen Gruppierungsknoten (vgl. 0).
`name`	Benennung des Knotens.

Methoden

`add_child`	Fügt einen Knoten in die Liste der Kind-Knoten ein. Hierbei wird nicht geprüft, ob bereits ein solcher Knoten (z.B. Benennung) vorhanden ist.
`remove_child`	Entfernt einen benannten Knoten aus der Liste der Kind-Objekte.

set_name	Setzt oder ändert den Namen eins Knotens.
get_name	Ermittelt den Namen eines Knotens.
is_enabled	Test, ob der Knoten aktiv ist.
is_movable	Test, ob der Knoten innerhalb des Szene-Graphen bewegt werden kann.
insert_child	Einfügen eines Knotens an eine bestimmte Stelle der Liste aus Kind-Knoten.
get_numchildren	Ermitteln der Anzahl von Kind-Knoten.
get_parent	Ermitteln des Eltern-Knoten.

4.4.2.4 Universe

```
interface Universe {
    VizMain::VisualSceneGraph get_visual_scenegraph();

    void set_visual_scenegraph(in VizMain::VisualSceneGraph scenegraph);

    VizMain::HapticSceneGraph get_haptic_scenegraph();

    void set_haptic_scenegraph(in VizMain::HapticSceneGraph scenegraph);

    void add_viewpoint(in VizMain::ViewPoint aViewpoint)
       raises(SceneGraphException);

    void remove_viewpoint(in VizMain::ViewPoint aViewPoint)
       raises(SceneGraphException);

    VizMain::ViewPoint get_viewpoint() raises(SceneGraphException);

    VizMain::ViewPoint next_viewpoint() raises(SceneGraphException);

    attribute VizMain::VisualSceneManager visual_scene_manager;
    attribute VizMain::HapticSceneManager haptic_scene_manager;
    attribute VizMain::HapticSceneGraph haptic_scene_graph;
    attribute VizMain::VisualSceneGraph visual_scene_graph;
    attribute VizMain::ViewPointList viewpoints;
    attribute VizMain::Window active_window;
};
```

Das `Universe` führt alle Szenen-Graphen und deren Manager zusammen. Es wird ein Fenster definiert, das die Sicht auf das Universum ausgehend vom aktiven `Viewpoint` rendert.

Attribute

`visual_scene_manager`	Der visuelle Szenen Manager.
`haptic_scene_manager`	Der haptische Szenen Mananger.
`haptic_scene_graph`	Der haptische Szenen-Graph.
`visual_scene_graph`	Der visuelle Szenen-Graph.
`viewpoints`	Liste der definierten Viewpoints.
`active_window`	Das zum Universum gehörige Fenster.

Methoden

`get_visual_scenegraph`	Gibt den visuellen Szene-Graphen als Objekt zurück.
`set_visual_scenegraph`	Setzt einen visuellen Szene-Graphen aktiv. Diese Methode wird zum schnellen Austausch des Szene-Graphen verwendet, wenn ein neues Visualisierungsskript geladen wurde.
`get_haptic_scenegraph`	Gibt den haptischen Szene-Graphen zurück.
`set_haptic_scenegraph`	Setzt einen neuen haptischen Szene-Graphen.

`add_viewpoint`	Fügt einen neuen Viewpoint in die Liste der definierten Viewpoints ein.
`remove_viewpoint`	Löscht einen vorhandenen Viewpoint.
`get_viewpoint`	Gibt den aktuellen Viewpoint zurück.
`next_viewpoint`	Springt zum nächsten definierten Viewpoint.

4.4.2.5 Viewpoint

```
interface ViewPoint {
    void rotate();

    void move();

    attribute long orientation;
    attribute VizGeometry::PointStruct position;
    attribute VizGeometry::VectorStruct direction;

    /**
     * Vertical scale factor applied to sceens.
     */

    attribute float aspect_ratio;

    /**
     * Distance between right and left eye.
     */

    attribute float parallax;

    /**
     * Horizontal offset in pixels which is applied to both eyes. It is
     * substracted from the left eye and added to the right eye.
     */

    attribute long convergence;
    attribute float convergence_distance;
    attribute boolean stereo;
    attribute string name;
};


typedef sequence<ViewPoint> ViewPointList;
```

Der `ViewPoint` ist die Position des Betrachters innerhalb des Universums. Er legt sowohl die absolute Position als auch die Blickrichtung fest. Hier werden ebenfalls die Parameter die für das stereoskopische Betrachten des Universums notwendig sind festgelegt.

Attribute

`orientation`	Ausrichtung des `Viewpoints`.
`position`	Ortsvektor zur Position im Weltkoordinatensystem.
`direction`	Richtungsvektor der Blickrichtung.
`aspect_ratio`	Seitenrelation zwischen Breite und Höhe bei der Projektion.
`parallax`	Wert für die scheinbaren Änderungen der Position eines beobachteten Objektes durch eine Verschiebung der Position des Beobachters.
`convergenz`	Wert für die Stellung der Augen, bei der sich die Blicklinien unmittelbar vor den Augen schneiden.
`convergence_distance`	Abstand dieses Blickpunktes.
`stereo`	Flag (Schalter) der festlegt, ob die Szene in Stereoskopischer Ansicht gerendert (betrachtet) wird.
`name`	Name des Viewpoints. Es sind mehrere definierte Viewpoints in einer Szene zulässig, jedoch nur immer ein aktiver Viewpoint. Mittels des Namens kann ein bestimmter Viewpoint schnell selektiert werden.

Methoden

`rotate`	Rotieren der Blickrichtung.
`move`	Bewegen der Blickrichtung.

4.4.2.6 VisualSceneGraph

```
interface VisualSceneGraph {
    void set_rootnode(in VizMain::Node rootnode);

    VizMain::Node load_file(in string aFileName)
       raises(SceneGraphException);

    /**
     * This function finds the first occurence of the node identified by a name.
     */

    VizMain::Node find_node_by_name(in string aName)
       raises(SceneGraphException);

    /**
     * Deletes all non root nodes from the Visual Scene Graph.
     */

    void vacuum() raises(SceneGraphException);

    VizMain::Node get_rootnode() raises(SceneGraphException);

    void load_material_table(in string filename)
       raises(SceneGraphException);

    void save_material_table(in string filename);

    attribute VizMaterial::MaterialList material_table;
    attribute VizMain::Node root_node;
};
```

Das Interface `VisualSceneGraph` stellt Methoden zur Verwaltung des Szenen-Graphs zur Verfügung.

Attribute

`material_table`	Liste mit allen Materialien, die im aktuellen Szene-Graphen verwendet werden. Aus Performance Gründen wird dies Liste am Anfang geladen und dann im Speicher gehalten.

root_node	Root-Knoten.

Methoden

set_rootnode	Setzen des Wurzel-Elements.
load_file	Laden eines (Teil-) Baums des Szene-Graphen aus einer Datei. Dateiformat ist VRML. Diese Methode wird verwendet, um komplexe Visualisierungen vorgefertigt zu laden, um sie direkt verwenden zu können.
find_node_by_name	Finden eines Knotens anhand seines Namens.
vacuum	Löschen aller Knoten im Szene-Graphen.
get_rootnode	Gibt das Wurzel-Element zurück.
load_material_table	Laden einer Liste von Materialien.
save_material_table	Speichern einer Liste von Materialien.

4.4.2.7 VisualSceneManager

```
interface VisualSceneManager {
    void add_device();

    void remove_device();

    void get_device();

    void next_device();

    attribute VizDevice::DeviceList device_list;
};
```

Analog dem `HapticSceneManager` verwaltet der `VisualSceneManager` die Liste der mit diesem Szene-Graphen verbundenen Ein-/Ausgabe Devices.

Attribute

`device_list`	Liste der aktiven Devices.

Methoden

`add_device`	Hinzufügen eines Ein-/Ausgabegerätes.
`remove_device`	Entfernen eines Ein-/Ausgabegerätes.
`get_device`	Zeiger auf ein benanntes Ein-/Ausgabegerät.
`next_device`	Hole das nächste Ein-/Ausgabegerät.

4.4.2.8 Window

```
enum ProjectionTypes {
        SYMMETRIC, ASYMMETRIC, GENERAL, ORGHOGRAPHIC
    };

    interface Window {


        VizMain::Universe get_universe() raises(SceneGraphException);

        attribute boolean enabled;
        attribute VizMain::ViewPoint active_viewpoint;
        attribute VizMain::ProjectionTypes projection_type;
    };
```

Zu jedem Universum gehört ein oder mehrere `Windows` in denen der Szenen-Graph dargestellt wird. Im Falle einer dreiseitigen CAVE gehören z.B. drei Windows zum Szenen-Graph, wobei sie sich je nach Anordnung (siehe 2.4.1.4) in der Position und Orientierung des Viewpoints (siehe 4.4.2.5) unterscheiden.

Attribute

`enabled`	Schalter der dieses Fenster als aktiv markiert.
`active_viewpoint`	Zeiger auf den aktiven Viewpoint den der Betrachter gewählt hat.
`projection_type`	Art der Projektion.

Methoden

`get_universe`	Gibt das zu diesem Fenster zugehörige Universum zurück.

4.4.3 VizNode Modul

Das VizNode Modul beihaltet die zentrale Interface Definition eines Knotens des Scene-Graphen. Die Knoten (Nodes) zeigen je nach Ausprägung unterschiedliches Verhalten und bestimmen die eigentliche Szene, welche die Rendering Engine berechnet und darstellt. Die Nodes sind die zentralen Datenelemente der Schnittstellendefinition, denn sie sind die Elemente des Szenen-Graphen.

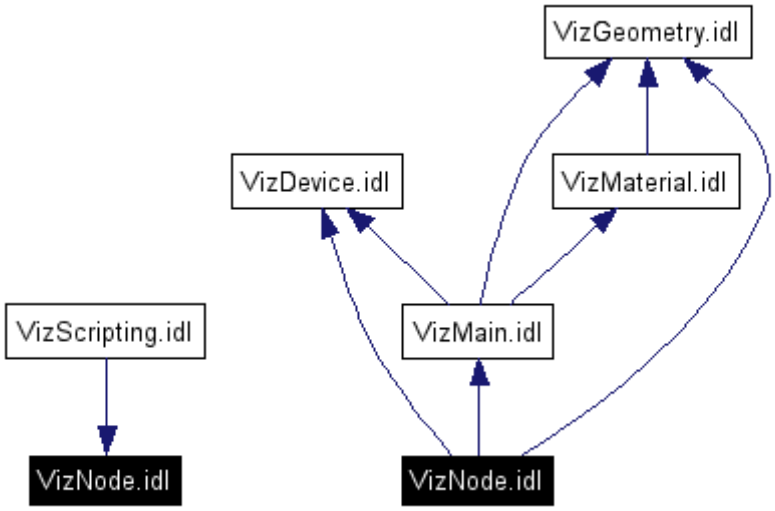


Abbildung 81: Abhänigkeitsgraphen des Moduls VizNode

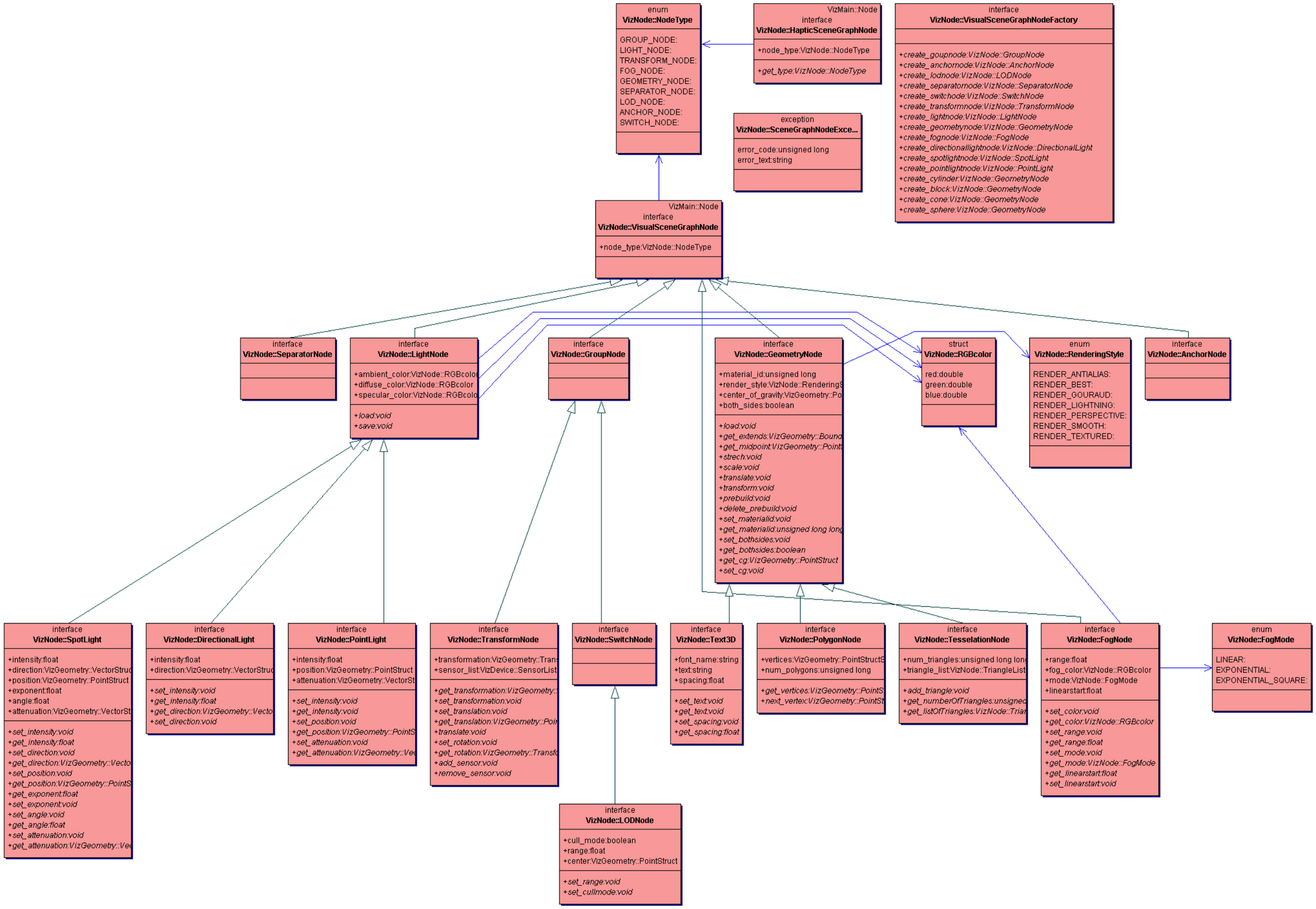


Abbildung 82: Interface Definitionen des VizNode Moduls

4.4.3.1 *HapticSceneGraphNode*

```
interface HapticSceneGraphNode : VizMain::Node {

        VizNode::NodeType get_type();

        attribute VizNode::NodeType node_type;
    };
```

Attribute

node_type	Knotentyp.

Methoden

get_type()	Bestimmen des Knoten-Typs.

4.4.3.2 *VisualSceneGraphNode*

```
interface VisualSceneGraphNode : VizMain::Node {

        attribute VizNode::NodeType node_type;
    };
```

Attribute

Node_type	Knotentyp.

4.4.3.3 *AnchorNode*

```
interface AnchorNode : VisualSceneGraphNode {
    };
```

4.4.3.4 FogNode

```
enum FogNode {
        LINEAR, EXPONENTIAL, EXPONENTIAL_SQUARE
    };

    interface FogNode : VisualSceneGraphNode {
        /**
         * Set the color of the fog.
         */

        void set_color(in VizNode::RGBcolor aColor)
           raises(SceneGraphNodeException);

        /**
         * Get the color of the fog.
         */

        VizNode::RGBcolor get_color() raises(SceneGraphNodeException);

        /**
         * Set the range where all objects are completely blended into fog.
         */

        void set_range(in float range) raises(SceneGraphNodeException);

        float get_range() raises(SceneGraphNodeException);

        void set_mode(in VizNode::FogNode mode)
           raises(SceneGraphNodeException);

        VizNode::FogMode get_mode() raises(SceneGraphNodeException);

        float get_linearstart() raises(SceneGraphNodeException);

        void set_linearstart(in float linearstart)
           raises(SceneGraphNodeException);

        /**
         * Distance where all objects are completely blended into fog.
         */

        attribute float range;

        /**
         * Color of the fog.
         */

        attribute VizNode::RGBcolor fog_color;
        attribute VizNode::FogMode mode;

        /**
         * Distance where the objects are affected by the fog color.
         * (Only in linear mode.)
         */

        attribute float linearstart;
    };
```

Ein `FogNode` ist ein Knoten-Element eines Szene-Graphen, der Nebel innerhalb der virtuellen Welt repräsentiert. Diese Elemente sind oftmals Bestandteile kommerzieller Szenen-Graph Systeme, um einen wirklichkeitsgetreuen Eindruck der virtuellen Welt zu erzeugen. Für technische Systeme spielen sie jedoch eine untergeordnete Rolle. Sie wurden in die Schnittstellenspezifikation aufgenommen um die Kompatibilität zu wahren.

Attribute

`range`	Abstand bei dem alle Objekte im Nebel komplett unsichtbar werden.
`fog_color`	Farbe des Nebels.
`mode`	Funktion die bestimmt, wie schnell das Objekt im Nebel unsichtbar wird.
`linearstart`	Abstand der festlegt, ab wann die Objekte von der Farbe des Nebels beeinflusst werden.

Methoden

`set_colour()`	Setzen der Farbe des Nebels.
`get_colour()`	Ermitteln der Farbe des Nebels.
`set_range()`	Setzen des range Wertes.
`get_range()`	Ermitteln des range Wertes.
`set_mode()`	Setzen des Modus.
`get_mode()`	Ermitteln des Modus.
`get_linearstart()`	Setzen des linearstart Wertes.
`set_linearstart()`	Ermitteln des linearstart Wertes.

4.4.3.5 *GeometryNode*

```
interface GeometryNode : VisualSceneGraphNode {


        void load(in string aFilename) raises(SceneGraphNodeException);

        /**
         * Obtains the extends of the geometry node.
         */

        VizGeometry::BoundingBox get_extends() raises(SceneGraphNodeException);

        /**
         * Obtains the midpoint of the bounding box of the geometry node.
         */

        VizGeometry::PointStruct get_midpoint();

        void strech(in VizGeometry::PointStruct factors,
           in VizGeometry::PointStruct center) raises(SceneGraphNodeException);

        void scale(in float factor) raises(SceneGraphNodeException);

        void translate(in VizGeometry::VectorStruct offset)
           raises(SceneGraphNodeException);

        void transform(in VizGeometry::TransformationStruct transformation)
           raises(SceneGraphNodeException);

        void prebuild() raises(SceneGraphNodeException);

        void delete_prebuild() raises(SceneGraphNodeException);

        void set_materialid(in unsigned long id)
           raises(SceneGraphNodeException);

        unsigned long long get_materialid() raises(SceneGraphNodeException);

        void set_bothsides(in boolean flag) raises(SceneGraphNodeException);

        boolean get_bothsides() raises(SceneGraphNodeException);

        VizGeometry::PointStruct get_cg() raises(SceneGraphNodeException);

        void set_cg(in VizGeometry::PointStruct cg)
           raises(SceneGraphNodeException);

        /**
         * Id with describes the used material for the geometry node. Used togehther
with the material table of a scene graph.
         */

        attribute unsigned long material_id;
        attribute VizNode::RenderingStyle render_style;
        attribute VizGeometry::PointStruct center_of_gravity;
        attribute boolean both_sides;
    };
```

Ein `GeometryNode` ist ein Knoten der Objektgeometrie abbildet. Von ihm sind weitere Knotentypen, wie z.B. `PolygonNode` und `TesselationNode` abgeleitet.

Attribute

`material_id`	Identifikator für das verwendete Material.
`render_style`	Flag, wie der Geometrieknoten gerendert wird.
`center_of_gravity`	Schwerpunkt.
`both_sides`	Polygonmodell ist zweiseitig.

Methoden

`load()`	Lädt Polygondaten in einen Geometrieknoten.
`get_extends()`	Ermittelt die Bounding-Box.
`get_midpoint()`	Ermittelt den Mittelpunkt des Geomerieknotens.
`strech()`	Skalieren entlang einer Fluchtpunktgeraden.
`scale()`	Skaliert den Geometrieknoten.
`translate()`	Verschiebeoperation des Geometrieknotens.
`transform()`	Anwenden einer Transformationsmatrix.
`prebuild()`	Evaluiert den Geometrieknoten teilweise, damit eine optimierte Visualisierung möglich ist.

`delete_prebuild()`	Löscht den vorher erzeugten Prebuild.
`set_materialid()`	Setzt das Material.
`get_materialid()`	Bestimmt das Material.
`set_bothsides()`	Setzt den Flag, dass das Polygonmodell zweiseitig ist.
`get_bothsides()`	Test, ob Polygonmodell zweiseitig ist.
`set_cg()`	Manuelles setzen des Schwerpunktes.
`get_cg()`	Ermitteln des Schwerpunktes.

4.4.3.6 GroupNode

```
interface GroupNode : VisualSceneGraphNode {
      attribute sequence<VisualSceneGraphNode> node_group;
    };
```

Ein `GroupNode` akkumuliert mehrere Knoten in sich, die wie ein Objekt behandelt werden. Es können auch evaluierte Teilbäume aus Performancegründen in diesem Knoten gespeichert werden. In der Praxis empfiehlt es sich z. B. `GeometryNodes`, die alle der gleichen Transformation unterworfen sind, vorher in einem `GroupNode` zusammenzufassen.

Attribute

`node_group`	Gruppe von Knotenelementen.

4.4.3.7 LightNode

```
interface LightNode : VisualSceneGraphNode {


        void load(in string filename) raises(SceneGraphNodeException);

        void save(in string filename) raises(SceneGraphNodeException);

        attribute VizNode::RGBcolor ambient_color;
        attribute VizNode::RGBcolor diffuse_color;
        attribute VizNode::RGBcolor specular_color;
    };
```

Ein `LightNode` repräsentiert eine Lichtquelle im Szenen-Graph. In jedem Universum sollte zumindest eine Lichtquelle vorhanden sein, damit durch ein Rendering Verfahren eine Darstellung der Szene berechnet werden kann. `LightNodes` gibt es in verschiedenen speziellen Ausprägungen (`SpotLight`, `PointLight`, `DirectionalLight`), die im Folgenden nicht nocheinmal gesondert aufgeführt werden.

Attribute

`ambient_color`	Farbe des Ambient-Wertes des Lichts.
`diffuse_color`	Farbe des Diffuse-Wertes des Lichts.
`specular_color`	Farbe des Specular-Wertes des Lichts.

Methoden

`load()`	Laden von Lichtinformationen aus einer Datei.
`save()`	Serialisieren von Lichtinformationen in einer Datei.

4.4.3.8 *SeparatorNode*

```
interface SeparatorNode : VisualSceneGraphNode {
    };
```

Ein `SeparatorNode` ist ein abstraktes Element, das Teile des Baumes voeinander trennt. Möchte man z.B. nicht, dass sich eine Transformation auf alle child-Objekte eines Knotens auswirkt, so fügt man ab der Stelle wo sie nicht mehr wirken soll den Separator Knoten ein. Ein Separator-Knoten besitzt keine Methoden.

4.4.3.9 *PolygonNode*

```
interface PolygonNode : GeometryNode {


        VizGeometry::PointStruct get_vertices()
           raises(SceneGraphNodeException);

        VizGeometry::PointStruct next_vertex() raises(SceneGraphNodeException);

        attribute VizGeometry::PointStructSeq vertices;

        /**
         * Number of polygons in this node.
         */

        attribute unsigned long num_polygons;
    };
```

Der `PolygonNode` zusammen mit dem `TesselationNode` sind die wichtigsten spezialisieren Geometrie Knoten die in einer Visualisierung verwendet werden. Ein `PolygonNode` beinhaltet eine Menge von Knoten, die über Polygone verbunden sind und eine geschlossene Fläche repräsentieren.

Attribute

`vertices`	Anzahl der Kanten.
`num_polygons`	Anzahl der Polygone.

Methoden

`get_vertices()`	Liste der Kanteninformationen als `PointStruct`.

`next_vertex()`	Iteriert über alle Kanten und gibt pro Aufruf die jeweils nächste Kante zurück.

4.4.3.10 TesselationNode

```
interface TesselationNode : GeometryNode {
        void add_triangle(in VizGeometry::PointStruct aTriangle)
           raises(SceneGraphNodeException);

        unsigned long long get_numberOfTriangles()
           raises(SceneGraphNodeException);

        VizNode::TriangleList get_listOfTriangles()
           raises(SceneGraphNodeException);

        attribute unsigned long long num_triangles;
        attribute VizNode::TriangleList triangle_list;
    };
```

Ein `TesselationNode` ist ein spezieller `GeometryNode` der seine geometrischen Informationen in Form von tessellierten Oberflächendaten enthält.

Attribute

`num_triangles`	Anzahl der Dreiecke.
`triangle_list`	Liste mit Dreiecken, bestehend aus den Eckpunkten.

Methoden

`add_triangle()`	Hinzufügen eines Dreiecks.
`get_numberOFTriangles()`	Ermitteln der Anzahl von Dreiecken.
`get_listOfTriangles()`	ermitteln der Liste mit Dreiecken.

4.4.3.11 SwitchNode

```
interface SwitchNode : GroupNode {
      attribute int active_node;
    };
```

Ein `SwitchNode` ist ein spezialisierter GroupNode, der jeweils nur eines seiner child-Objekte sichtbar macht.

Attribute

`active_node`	Momentan aktiver Knoten in der Liste der child-Objekte.

4.4.3.12 LODNode

```
interface LODNode : SwitchNode {


        /**
         * Set the switch out distance for the children.
         */

        void set_range(in float range) raises(SceneGraphNodeException);

        /**
         * Implementation of a quick reject test. Subtree is not traversed if
         * bounding box lies out of the viewing area.
         */

        void set_cullmode(in boolean mode) raises(SceneGraphNodeException);

        attribute boolean cull_mode;
        attribute float range;

        /**
         * Center of the LOD node. This is used to calculate the distance to the
         * viewpoint.
         */

        attribute VizGeometry::PointStruct center;
    };
```

Ein "Level of Detail" (LOD) Mechanismus ist ein leistungsfähiges Werkzeug, um die Komplexität einer Szene zu reduzieren. Grundidee hierbei ist die Tatsache, dass Objekte, die weit von einem Betrachter entfernt sind, nicht im vollen Detaillierungsgrad dargestellt werden müssen, da sie zum einen außerhalb Fokus liegen, zum anderen kaum noch unterscheidbar sind. Ein Level Of Detail Mechanismus lässt es daher zu, für ein bestimmtes Objekt mehrere Instanzen unterschiedlicher Detaillierung zu definieren, um sie je nach Position (Entfernung) des Betrachters zu verwenden.

Intelligente LOD Mechanismen können sogar die Instanzen geringer Detaillierung aus dem Original berechnen und zur Verfügung stellen. Objekte, die als child-Objekte unter einem `LODNode` liegen, können diesen Mechanismus verwenden.

Attribute

`cull_mode`	Wird dieser Wert überschritten werden die Kindobjekte überhaupt nicht mehr traversiert.
`range`	Setzt den Wert für die Entfernung ab dem die Kind-Objekte dieses Knotens ausgeblendet werden.
`center`	Zentrum des LOD-Knotens.

Methoden

`set_range()`	Setzen des `range` Wertes.
`set_cullmode()`	Flag für den Reject-Test (Culling Mode).

4.4.3.13 TransformNode

```
interface TransformNode : GroupNode {
        /**
         * Returns the transformation structure.
         */

        VizGeometry::TransformationStruct get_transformation()
           raises(SceneGraphNodeException);

        /**
         * Replaces the transformation struct of the node.
         */

        void set_transformation(in VizGeometry::TransformationStruct
           aTransformation) raises(SceneGraphNodeException);

        /**
         * Replaces the translation component.
```

```
         */

        void set_translation(in VizGeometry::PointStruct aTranslation)
           raises(SceneGraphNodeException);

        /**
         * Returns the translation component.
         */

        VizGeometry::PointStruct get_translation();

        /**
         * Adding an incremental translation.
         */

        void translate(in VizGeometry::PointStruct parameter0)
           raises(SceneGraphNodeException);

        /**
         * Sets the rotational component.
         */

        void set_rotation(in VizGeometry::VectorStruct i_dir,
           in VizGeometry::VectorStruct k_dir) raises(SceneGraphNodeException);

        /**
         * Gets the rotational component.
         */

        VizGeometry::TransformationStruct get_rotation()
           raises(SceneGraphNodeException);

        /**
         * Attaches a sensor to a transform node.
         */

        void add_sensor(in VizDevice::Sensor aSensor)
           raises(SceneGraphNodeException);

        /**
         * Removes a sensor from a transform node.
         */

        void remove_sensor(in VizDevice::Sensor aSensor)
           raises(SceneGraphNodeException);

        attribute VizGeometry::TransformationStruct transformation;
        attribute VizDevice::SensorList sensor_list;
    };
```

Transformationen bestimmen die Lage eines Objektes relativ zu einem anderen Objekt oder Weltkoordinatensystem. `TransformNodes` werden in der Regel in GroupNodes integriert, um eine relative Positionierung im Weltkoordinatensystem zu erreichen.

Attribute

`transformation()`	Transformieren.

`sensor_list()`	Zeiger auf die Liste der Sensoren.

Methoden

`get_transformation()`	Ermitteln der aktuellen Transformationsmatrix.
`set_transformation()`	Setzen einer Transformationsmatrix.
`set_translation()`	Setzen der Translations-Komponente.
`get_translation()`	Ermitteln der Translationskomponente.
`translate()`	Translahieren.
`set_rotation()`	Setzen der Rotation im Raum.
`get_rotation()`	Ermitteln der Rotation im Raum.
`add_sensor()`	Hinzufügen eines Sensor-Objektes.
`remove_sensor()`	Entfernen eines Sensor-Objektes.

4.4.3.14 Text3D

```
interface Text3D : GeometryNode {

        void set_text();
        void get_text();
        void set_spacing(in float spacing) raises(SceneGraphNodeException);
```

```
        float get_spacing() raises(SceneGraphNodeException);

        /**
         * Name of the used font.
         */

        attribute string font_name;
        attribute string text;
        attribute float spacing;
    };
```

Ein `Text3D` Knoten repräsentiert einen dreidimensionalen Text innerhalb des Universums. Dieser Knoten wird häufig für Annotationen verwendet.

Attribute

`font_name`	Name des Zeichensatzes.
`text`	Text.
`spacing`	Spacing zwischen den Buchstaben.

Methoden

`set_text()`	Setzt den darzustellenden Text.
`get_text()`	Gibt den darzustellenden Text zurück.
`set_spacing()`	Setzt das Character-Spacing.
`get_spacing()`	Ermittelt das Character-Spacing.

4.4.3.15 VisualSceneGraphNodeFactory

Das Interface `VisualSceneGraphNodeFactory` ist dazu da, die verschieden Typen von Nodes zu Instanziieren und einige wichtige Basis-Geometrieelemente (Würfel, Kugel, Konus, …)

zu erzeugen. Die einzelnen Methoden dieses Interfaces werden hier nicht weiter beschrieben, da sie weitestgehend selbsterklärend sind.

```
interface VisualSceneGraphNodeFactory {


        /**
         * Create a new group node.
         */

        VizNode::GroupNode create_goupnode(in VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new anchor node.
         */

        VizNode::AnchorNode create_anchornode(in VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new level of detail node.
         */

        VizNode::LODNode create_lodnode(in VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new separator node.
         */

        VizNode::SeparatorNode create_separatornode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new switch node.
         */

        VizNode::SwitchNode create_switchode(in VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new Transformnode.
         */

        VizNode::TransformNode create_transformnode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new lightnode.
         */

        VizNode::LightNode create_lightnode(in VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new geometry node.
         */

        VizNode::GeometryNode create_geometrynode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new fognode.
```

```
         */

        VizNode::FogNode create_fognode(in VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new directional light.
         */

        VizNode::DirectionalLight
           create_directionallightnode(in VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new spot light.
         */

        VizNode::SpotLight create_spotlightnode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        VizNode::PointLight create_pointlightnode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Creates a cylinder.
         */

        VizNode::GeometryNode create_cylinder(in float height, in float radius,
           in boolean bothsides, in unsigned long tesselation)
           raises(SceneGraphNodeException);

        VizNode::GeometryNode create_block(in float x, in float y, in float z,
           in boolean bothsides) raises(SceneGraphNodeException);

        VizNode::GeometryNode create_cone(in float heigth, in float radius,
           in unsigned long tesselation, in boolean bothsides)
           raises(SceneGraphNodeException);

        VizNode::GeometryNode create_sphere(in float radius,
           in unsigned long tesselation, in boolean bothsides)
           raises(SceneGraphNodeException);

    };
```

Es muss sichergestellt werden, dass alle Knotenelemente des Szene-Graphen über dieses Interface erzeugt werden, damit die Kompatibilität zu verschiedenen Szenen-Graph Systemen erhalten beleibt.

Methoden

`create_anchornode()`	Erzeugen eines Ankerknotens.
`create_lodnode()`	Erzeugen eines LOD-Knotens.

`create_separatornode()`	Erzeugen eines Separatorknotens.
`create_switchnode()`	Erzeugen eines Switchknotens.
`create_transformnode()`	Erzeugen eines Transformationsknotens.
`create_lightnode()`	Erzeugen eines Lichtknotens.
`create_geometrynode()`	Erzeugen eines Geometrieknotens.
`create_fognode()`	Erzeugen eines Nebel-Knotens.
`create_directionallightnode()`	Erzeugen eines gerichteten Lichtknotens.
`create_pointlightnode()`	Erzeugen eines Punktlicht-Knotens.
`create_spotlightnode()`	Erzeugen eines SpotLight-Knotens.
`create_block()`	Erzeugen eines Quaders. (Basisgeometrie)
`create_cone()`	Erzeugen eines Konus.
`create_sphere()`	Erzeugen einer Kugel.

4.4.4 VizGeometry Modul

Das VizGeometryModul enthält einige einfache Strukturdefintionen, die von den anderen Modulen benötigt werden, wie Vektor, Punkt, Strahl und die BoundingBox. Eine BoundingBox ist ein Quader der mit seinen maximalen Abmessungen das betrachtete Geometrieelement vollständig umschließt.

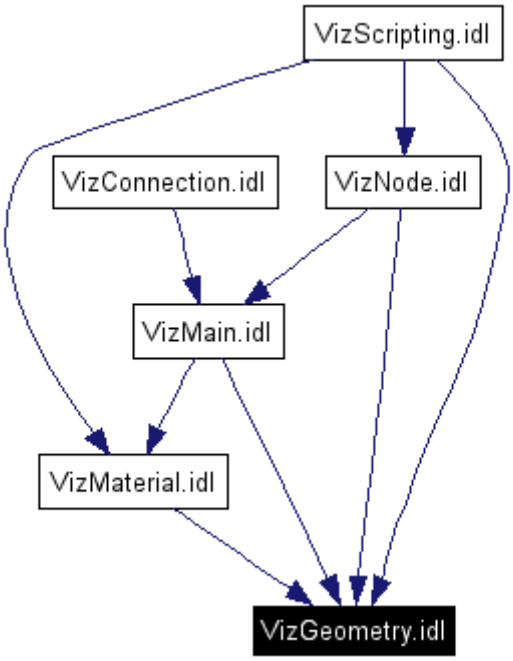


Abbildung 83: Abhängigkeitsgraph des Moduls VizGeometry

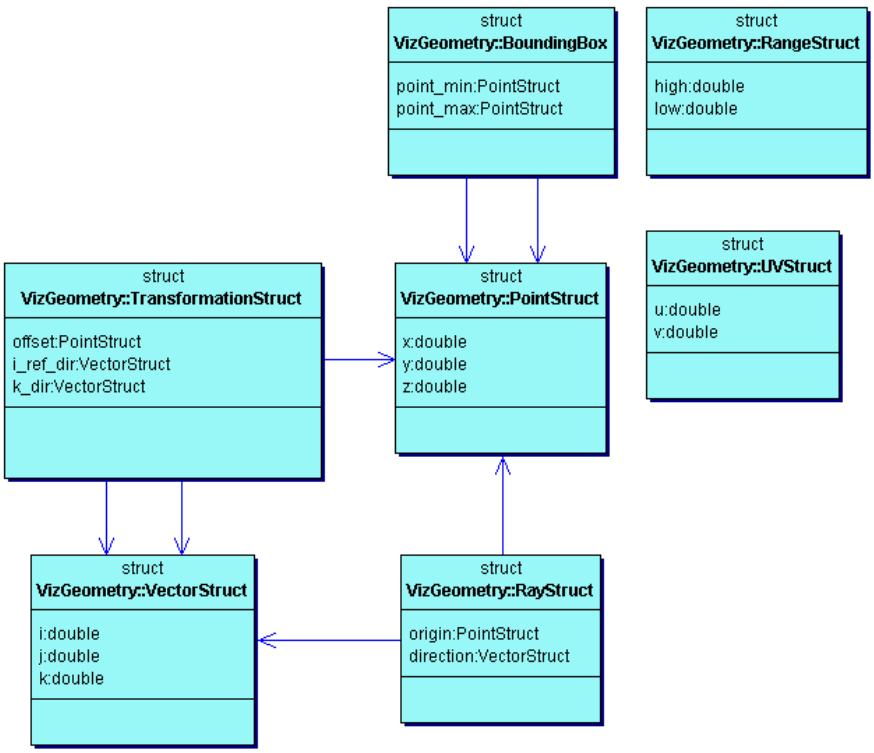


Abbildung 84: Interface Definitionen des VizGeometry Moduls

4.4.5 VizMaterial Modul

Das VizMaterialModul enthält alle Interfaces zur Definition von Materialien und deren Beleuchtungseigenschaften.

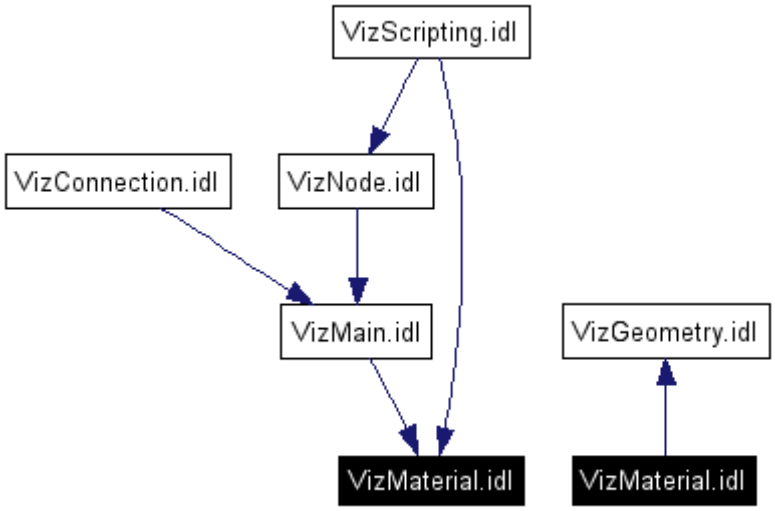


Abbildung 85: Abhängigkeitsgraph des Moduls VizMaterial

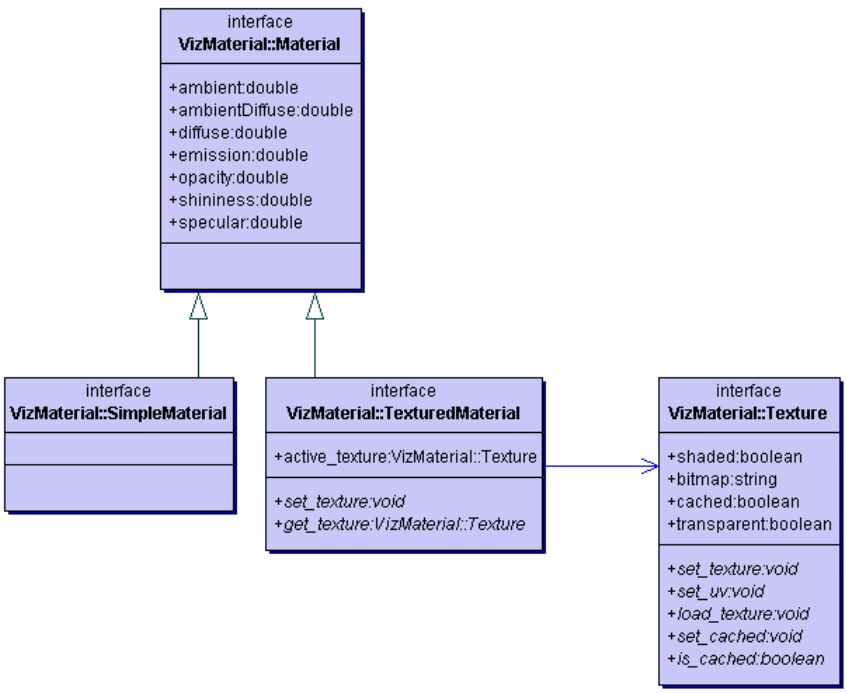


Abbildung 86: Interface Definitionen der VizMaterial Moduls

4.4.5.1 Material

Allgemeine Materialdefinition.

```
interface Material {

        attribute double ambient;
        attribute double ambientDiffuse;
        attribute double diffuse;
        attribute double emission;
        attribute double opacity;
        attribute double shininess;
        attribute double specular;
    };
```

In der Definition der Materialeigenschaften werden die Werte für diverse Beleuchtungseigenschaften festgelegt.

4.4.5.2 SimpleMaterial

Definition eines einfachen Standard-Materials als Default-Material.

```
interface SimpleMaterial : Material {
    };
```

4.4.5.3 Texture

Eine Textur ist das Bild, welches auf der Oberfläche eines virtuellen Körpers dargestellt wird; man spricht hierbei von mapping. Ein Pixel der Textur wird als Texel bezeichnet. Dabei können Texturen praktisch jede Eigeschaft einer Oberfläche gezielt verändern.

```
interface Texture {


        void set_texture(in boolean shaded, in boolean transparent,
           in string bitmap);

        void set_uv(in VizGeometry::UVStruct uvarray);

        void load_texture(in string filename);

        void set_cached(in boolean flag);

        boolean is_cached();

        attribute boolean shaded;
        attribute string bitmap;
        attribute boolean cached;
        attribute boolean transparent;
    };
```

Attribute

`shaded`	Erzeugen eines Schattens.
`bitmap`	Pfad, aus dem das Bitmap geladen wird.
`cached`	Flag, ob die Textur ge-cached wird.
`transparent`	Flag, ob Bitmap transparent ist.

Methoden

`set_texture()`	Mappen eines Bitmaps auf den Körper.
`set_uv()`	Festlegen der Ausrichtung des Bitmaps.
`load_texture()`	Laden der Textur zur Laufzeit.
`set_cached()`	Flag setzen, um die Textur im Cache zu belassen.
`is_cached()`	Abfragen des Flags.

4.4.5.4 TexturedMaterial

Materialknoten mit Textur.

```
interface TexturedMaterial : Material {

        void set_texture(in VizMaterial::Texture aTexture);

        VizMaterial::Texture get_texture();

        attribute VizMaterial::Texture active_texture;
    };
```

Attribute

`active_texture`	Aktuell gültige Textur.

Methoden

`set_texture()`	Setzen der Textur.
`get_texture()`	Ermitteln der Textur.

4.4.6 VizDevice Modul

Geräte, die es dem Benutzer erlauben interaktiv mit der Szene umzugehen, werden als Devices in den Szene-Graphen integriert. Beispiele sind CyberGloves oder ein Tracking-System. Auf die verschiedenen unterstützten Geräte wird im Rahmen dieser Arbeit nicht gesondert eingegangen, da sie zur Visualisierung von Informationen nicht primär notwendig und bisher nur prototypisch implementiert worden sind.

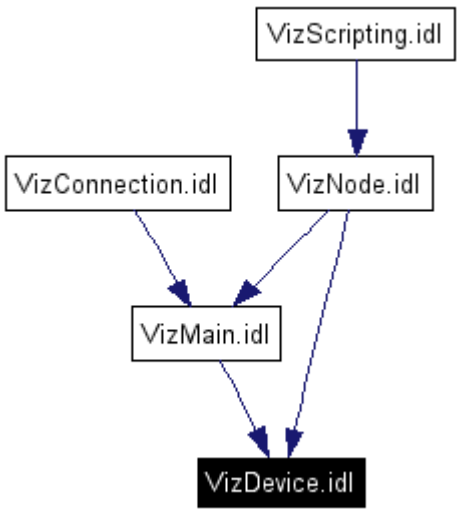


Abbildung 87: Abhängigkeitsgraph des Moduls VizDevice

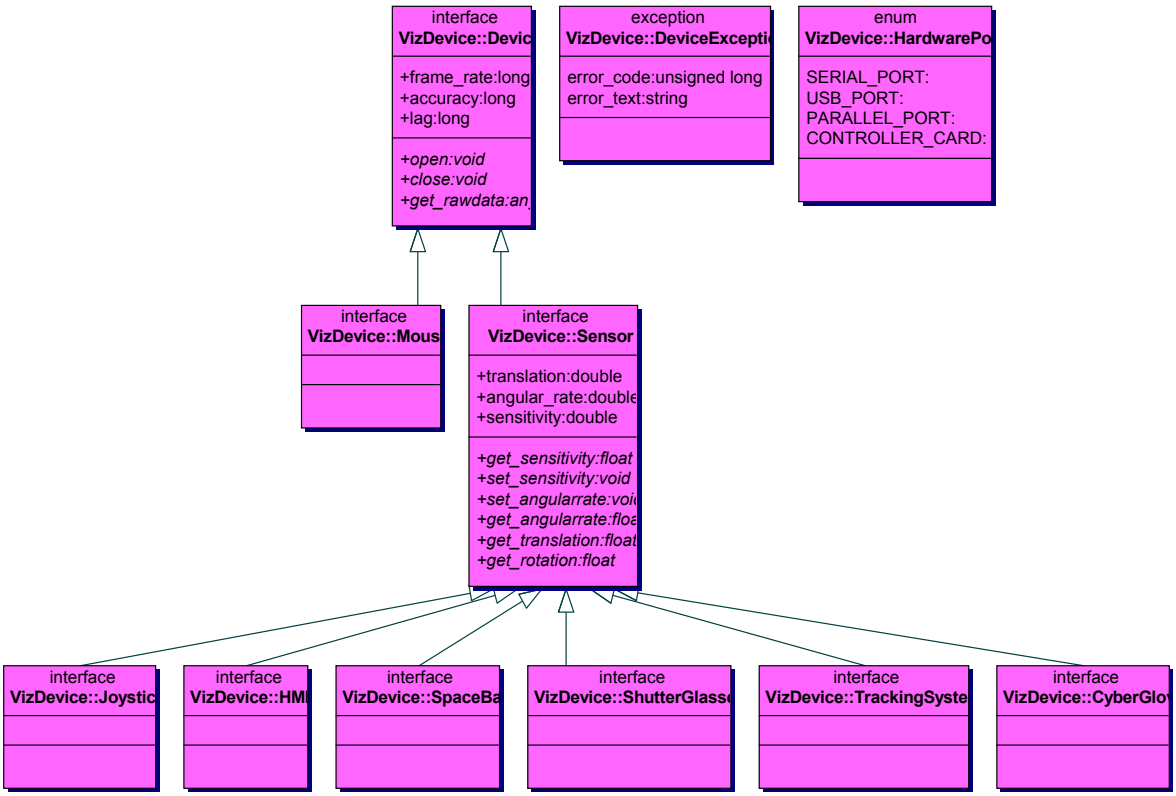


Abbildung 88: Interface Definitionen des VizDevice Moduls

4.4.7 VizScripting Modul

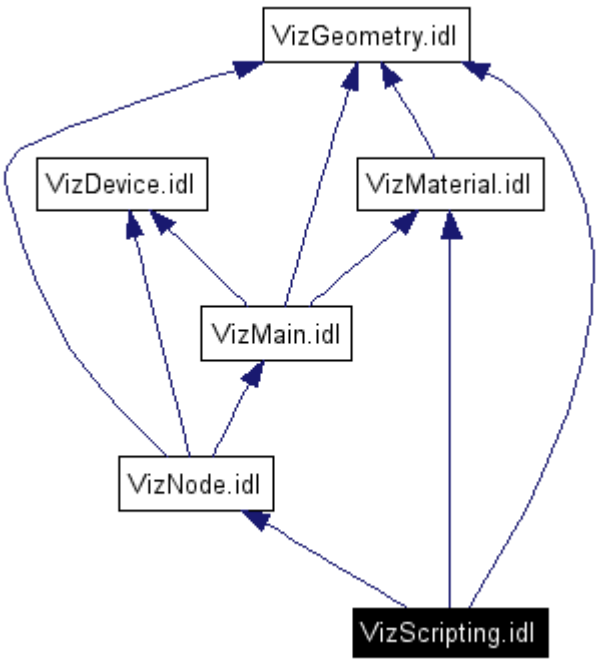


Abbildung 89: Abhängigkeitsgraph des Moduls VizScripting

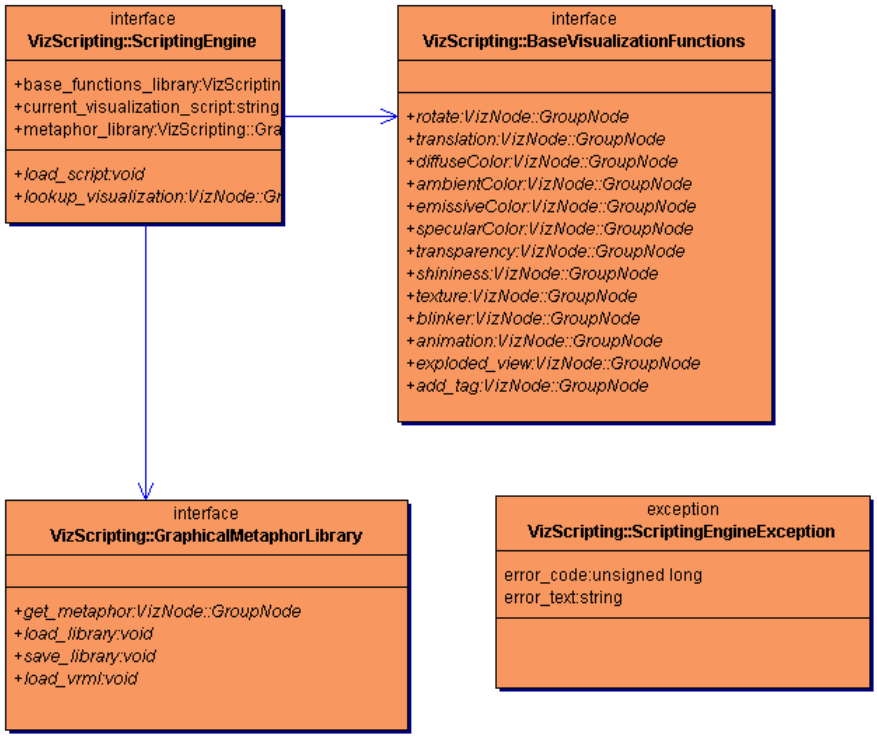


Abbildung 90: Interface Definitionen des VizScripting Moduls

Das VizScripting Modul beinhaltet die Interfaces der ScriptingEngine, die die Umsetzung der Metainformationen in grafische Metaphern steuert. Das Modul ist in die eigentliche `ScriptingEngine` und zwei Bibliotheksmodule aufgeteilt. `BaseVisualisation-Funktions` hält alle Funktionen zum Verändern von Visualisierungen, die `Graphical-MetaphorLibrary` die Templates der grafischen Metaphern.

4.4.7.1 ScriptingEngine

```
interface ScriptingEngine {
  /**
    * Load a translation script.
    */
    void load_script(in string filename) raises(ScriptingEngineException);

 /**
   * Main method of the scripting engine.
   * Gets an entity id in order to look up the visulaization script if there is a
   * visulalisation for this class of meta information. If there is no such class it
   * returns the geometry as it was in a group node. If there is a special
   * visualisation, the script is processed an a new subtree is assembled holding the
   * original geometry, the graphical metaphors and some functions applied.
   */
        VizNode::GroupNode   lookup_visualization(in   string   entity_id)   raises(
ScriptingEngineException );

    attribute VizScripting::BaseVisualizationFunctions base_functions_library;
    attribute string current_visualization_script;
    attribute VizScripting::GraphicalMetaphorLibrary metaphor_library;
    };
```

Attribute

`base_funktions_library`	Bibliothek aus Basisfunktionen die die grafischen Metaphern verändern.
`current_visualization_script`	Visualisierungs-Script das für die aktuelle Visualisierung verwendet wird.
`metaphor_library`	Bibliothek der grafischen Metaphern.

Methoden

`load_script()`	Laden eines Scriptes.
`lookup_visualization()`	Ermitteln der grafischen Methapher für die aktuelle Objektklasse.

4.4.7.2 BaseVisualisationFunctions

BaseVisulaization Funktions ist ein Interface, das sämtliche Methoden zum Verändern von Visualisierungeigenschaften von grafischen Methaphern aufnimmt.

```
interface BaseVisualizationFunctions {

        VizNode::GroupNode rotate(in string entity_id, in VizGeometry::VectorStruct
axis) raises( ScriptingEngineException );

        VizNode::GroupNode       translation(in       string       entity_id,       in
VizGeometry::VectorStruct trans) raises( ScriptingEngineException );

        VizNode::GroupNode diffuseColor(in string entity_id, in VizNode::RGBcolor
color) raises( ScriptingEngineException );

        VizNode::GroupNode ambientColor(in string entity_id, in VizNode::RGBcolor
color) raises( ScriptingEngineException );

        VizNode::GroupNode emissiveColor(in string entity_id, in VizNode::RGBcolor
color) raises( ScriptingEngineException );

        VizNode::GroupNode specularColor(in string entity_id, in VizNode::RGBcolor
color) raises( ScriptingEngineException );

        VizNode::GroupNode transparency(in string entity_id, in float transparency)
raises( ScriptingEngineException );

        VizNode::GroupNode shininess(in float entity_id, in float shininess) raises(
ScriptingEngineException );
```

```
        VizNode::GroupNode        texture(in        string        entity_id,        in
VizMaterial::TexturedMaterial texture) raises( ScriptingEngineException );

        /**
         * Time based switching of the visibility.
         */
        VizNode::GroupNode blinker(in string entity_id, in float interval) raises(
ScriptingEngineException );

        /**
         * Time based switching of the texture.
         */
        VizNode::GroupNode animation(in string entity_id, in float interval) raises(
ScriptingEngineException );

        /**
         * Exploded view of the part.
         */
        VizNode::GroupNode  exploded_view(in  string  entity_id,  in  float  factor)
raises( ScriptingEngineException );

        VizNode::GroupNode add_tag(  in string entity_id, in string text, in boolean
fixed_tag) raises( ScriptingEngineException );
};
```

4.4.7.3 GraphicalMetaphorLibrary

Bibliothek der grafischen Metaphern.

```
interface GraphicalMetaphorLibrary {


        VizNode::GroupNode    get_metaphor(in    string    aMetaphor_id)    raises(
ScriptingEngineException );

        void load_library(in string filename) raises( ScriptingEngineException );

        void save_library(in string filename) raises( ScriptingEngineException );

        /**
         * Load a vrml file which holds a special graphical metaphor.
         */
        void load_vrml(in string aMetaphor_id) raises( ScriptingEngineException );
};
```

Methoden

`get_metaphor()`	Factory für den Root-Knoten einer Instanz einer grafischen Metapher.
`load_library()`	Lädt eine Bibliothek von einem Speichermedium.

`save_library()`	Serialisieren einer Bibliothek auf ein Speichermedium.
`load_vrml()`	Importieren von VRML-daten in die Bibliothek.

4.5 Scripting Mechanismus

Der für die anwendungsbezogene Visualisierung entworfene Scripting-Mechanismus unterscheidet grundsätzlich zwischen zwei verschiedenen Arten von Funktionen:

- verändernde Funktionen
- darstellende Funktionen

Sie werden in den beiden folgenden Abschnitten detailliert erläutert.

4.5.1 Verändernde Funktionen

Die verändernden Funktionen beinhalten keine visualisierenden Komponenten, sondern nur Kombinationen aus OpenSG-`NODES`, die sichtbare Geometrien in ihrem Verhalten oder Aussehen beeinflussen können. Wird eine Funktion dieses Typs in den Szenen-Graph eingefügt, dann werden die Eigenschaften ihrer Substruktur an alle `NODES` rechts davon vererbt. Diese Art von Funktionen können sowohl auf das Modell selbst als auch auf darstellende Funktionen (siehe Kapitel 4.5.2) angewendet werden.

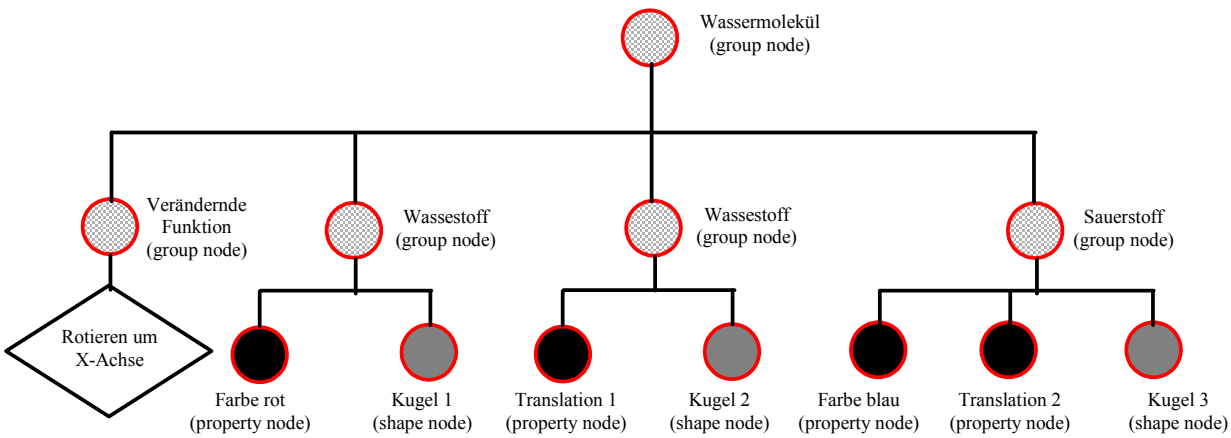


Abbildung 91: Auswirkung einer verändernden Funktion auf den Scene-Graph

Wird, Abbildung 91 entsprechend, eine verändernde Funktion mit der Fähigkeit "Rotation um die X-Achse" in den oben gezeigten Szenengraph (Wassermolekül) eingebracht, dann überträgt sich das Verhalten auf die dahinter liegenden Nodes. Das Molekül wird sich infolgedessen um die X-Achse drehen.

4.5.2 Darstellende Funktionen

Darstellende Funktionen sind im Gegensatz zu verändernden Funktionen gekapselte Visualisierungseinheiten, die keine Auswirkungen auf ihre Umgebung haben. Sie sind in ihrer Struktur nach oben durch einen `SEPARATOR`-Knoten gesichert. Diese Methoden werden in eine eigene Substruktur des Szenen-Graph geschrieben, um sie vor Modell-bezogenen Eingriffen zu schützen. Wird z.B. eine Funktion "transparency" auf das Modell angewandt, dann verhindert die in Abbildung 92 gezeigte Anordnung von `NODES` die Weitergabe dieser Eigenschaft an die Anforderungsvisualisierungen.

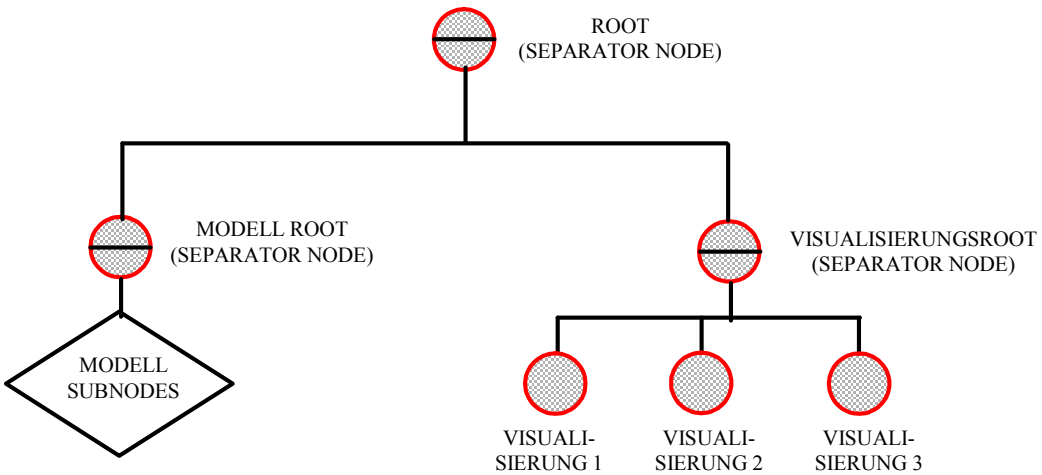


Abbildung 92: Szenengraph ROOT-Struktur

Will man nun jedoch verändernde Funktionen auf darstellende anwenden, so muss eine geeignete Position dieser `NODES` im Szenen-Graph gefunden werden. Der einfachste Fall wäre, alle `NODES` als Kinder des Visualisierungs-`ROOT-NODES` in eine Liste zu schreiben und in dieser Reihenfolge abzuarbeiten. Der große Nachteil dabei ist, dass sich nun die Eigenschaften der verändernden Funktionen auf alle nachstehenden darstellenden NODES auswirken. Ziel sollte jedoch sein, Funktionen gezielt auf einzelne Visualisierungen anwenden zu können. Der logische Schluss muss also sein, die verändernden Funktionen als Unterstruktur (Sub-NODES) in die Visualisierungsmethoden mit einzubinden. Abbildung 93 zeigt den Unterschied zwischen beiden Fällen.

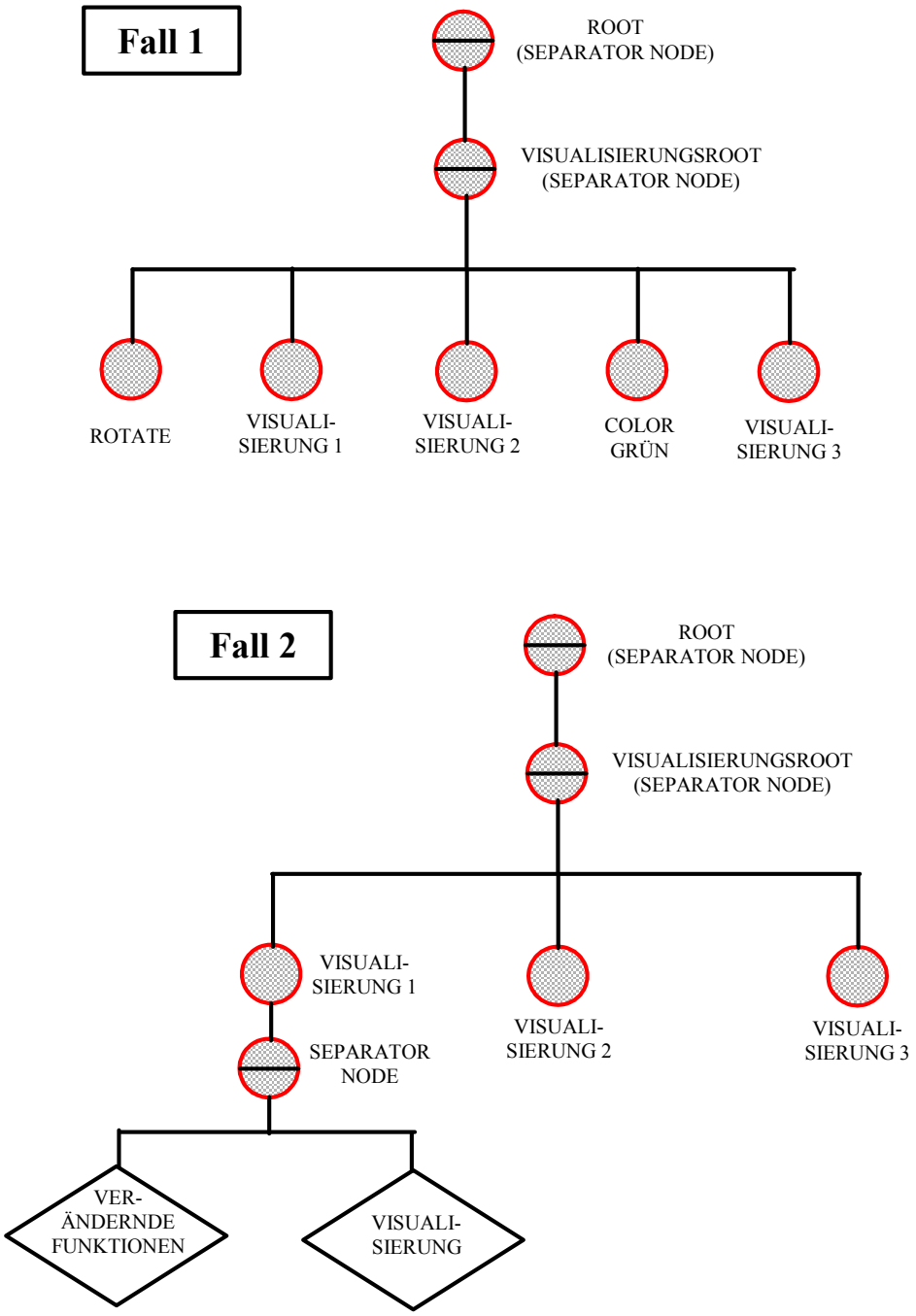


Abbildung 93: Positionierung von verändernden Nodes

Zu Fall 1: Die Rotation wird gesetzt und wirkt sich nun auf die Visualisierungen 1 bis 3 aus. Die Eigenschaft des `COLOR-NODES` betrifft nur Visualisierung 3. Eine gezielte Zuweisung ist also nicht möglich. Programmiertechnisch ist Fall 1 die erheblich einfachere Variante, ist jedoch aufgrund der beschränkten Einsatzfähigkeit nicht tragbar.

Zu Fall 2: Durch die Erweiterung der Parameterliste der darstellenden Funktionen um ein `GroupNode` kann so die gewünschte Substruktur übergeben werden und innerhalb der Funktion an der richtigen Stelle aufgerufen werden. Dem Programmierer wird hiermit die Möglichkeit

eröffnet, darstellende Strukturen zu bilden, die vor dem Aufruf von `GroupNodes` erzeugt werden und somit von verändernden Funktionen nicht beeinflussbar sind.

Für weitere Implementierung wären Funktionen mit Ausprägungen folgender Art vorzusehen:

- **interaktive Visualisierungen**: Möglich wären hier kleine Markierungen (Kugeln, etc.) auf den Attribut-tragenden Geometrien, die durch Anklicken mit der Maus neue darstellende Objekte (3DBox, 3DText etc.) erzeugen. Diese Methode kann sehr zur Übersichtlichkeit des Szenarios beitragen, da im Gegensatz zur kompletten visuellen Umsetzung der Attribute nur die ausgewählten Strukturen sichtbar werden.

- **Visualisierungen mit Verweis auf 2 ENTITYS**: Zur Darstellung von Konkurrenz- oder Ausschlussbeziehungen zwischen zwei Anforderungen im Szenario muss jeweils ein Geometriebezug diesbezüglich hergestellt werden.

- **3D-Symbolik**: Symbole identifizieren die attribut-tragenden Geometrien. Hierbei ist die Bedeutung der Symbole durch eine Legende eindeutig festgelegt. Die geometrische Ausprägung der Symbole kann sowohl vom Attributtyp als auch von Inhalten der Attribute abhängig sein. Die 3D-Symbolik, in Verbindung mit interaktiven Visualisierungen, stellt eine hervorragende Möglichkeit dar, umfangreiche Daten in verschiedenen Detaillierungsstufen dem Benutzer zur Verfügung zu stellen.

- **Visualisierung von physikalischen Größen**: Wird der Attributierungsmechanismus zur Ablage physikalisch berechneter Größen, wie z.B. Geschwindigkeit- oder Druckverteilungen aus Strömungssimulationen genutzt, dann ist es möglich, die vorhandenen Rohdaten mit Geometriebezug zu visualisieren. Bei Druck- oder Temperaturverteilungen kann dies beispielsweise durch Farbgradienten geschehen. Geschwindigkeitsverteilungen könnten über Vektoren dargestellt werden, die durch ihre Länge und Richtung Rückschlüsse auf dahinter stehende Werte zulassen.

4.5.3 Aufbau des Skriptes

In der Scripting-Engine wird das XML-Dokument zur Ansteuerung der Visualisierungen ausgewertet und dementsprechend die Funktionen der Bibliothek mit den Templates in den Szenen-Graph eingefügt. Im Folgenden wird der Aufbau des Abbildungsskriptes in XML beschrieben.

Hauptaufgabe der Skriptsyntax soll sein, verschiedenen Attributtypen darstellende und verändernde Funktionen zuzuweisen und diese durch benutzerspezifische Parameter zu steuern. Wie bereits in Abschnitt 4.5.2 erwähnt, muss es zusätzlich möglich sein, darstellende Funktionen durch Verändernde zu beeinflussen. Der hier gewählte Ansatz für die Syntax beschränkt sich aus Gründen der Übersichtlichkeit nur auf die Verwendung von Elementen. Auf diese Weise lässt sich problemlos eine streng hierarchische Baumstruktur erzeugen.

```
<darstellung>
  <attribut_typ>
    <funct>
      <darstellendeFunktion1>
        <param>default</param>
        <veränderndeFunktion1>
          <param>123</param>
        </ veränderndeFunktion1>
      </ darstellendeFunktion1>
      < veränderndeFunktion2>
        <param>234</param>
      </ veränderndeFunktion2>
    </funct>
  </ attribut_typ >
  <model>
    <funct>
      <veränderndeFunktion3>
        <param>345</param>
      </veränderndeFunktion3>
    </funct>
  </model>
</darstellung>
```

Abbildung 94: Im Rahmen der DTD möglicher Aufbau des XML-Dokuments

Anhand des o. g. Beispiels können alle Fälle von Zuweisungen und Verschachtelungen erläutert werden:

- **<darstellung>**: Stellt das Wurzelelement dar und signalisiert, dass die Substruktur Visualisierungen und graphische Ansteuerungen beinhaltet.
- **<attrib_typ>**: Attributtypen, die visualisiert werden sollen. Derzeit definierte Tag-Namen: <att_qualitativ>; <att_quantitativ>.

- **<funct>**: Beinhaltet eine Liste aller Funktionen, die auf <attrib_typ> angewendet werden können.
- **<darstellendeFunktion>**: Direkter Bezug zum Pendant aus der Klasse BasisVisualizationFunctions. Derzeit definierte Tag-Namen: <box3d>, <text3d>, <text>, <box>. Als Kinder sind hier zusätzlich alle verändernden Funktionen möglich, deren Gültigkeit sich auf diesen TAG beschränkt.
- **<veränderndeFunktion>**: Direkter Bezug zum Pendant aus der Klasse BasisVisualizationFunctions. Derzeit definierte TAG-Namen: <texture>, <rotate>, <rotor>, <diffuseColor>, <transparency>, <shininess>, <emissiveColor>, <ambientColor>, <specularColor>.
- **<param>**: Parameter; jede der o.g. Funktionen muss genau einen Parameter als Kind haben. Der TAG-Inhalt wird der Parameterliste der jeweiligen Funktion übergeben. Wird nichts anderes angegeben, muss der Parameter auf "default" gesetzt werden.
- **<model>**: Von der eigentlichen Aufgabe des XML-Dokuments abweichend, ist es hier schnell und unkompliziert möglich, verändernde Funktionen auf das Gesamtmodell anzuwenden, ohne Anforderungsvisualisierungen miteinzuschließen.

Die dazugehörige DTD lautet:

```
<?xml encoding="ISO-8859-1"?>

<!ELEMENT darstellung    (att_qualitativ | model)+>
<!ELEMENT att_qualitativ (funct)>
<!ELEMENT model          (funct)>

<!ENTITY % changing_function "texture | rotate | rotor | diffuseColor | transparency |
shininess | emissiveColor | ambientColor | specularColor">
<!ENTITY % display_function "box | box3d | text | text3d">

<!ELEMENT texture              (param)>
<!ELEMENT rotate               (param)>
<!ELEMENT rotor                (param)>
<!ELEMENT diffuseColor         (param)>
<!ELEMENT transparency         (param)>
<!ELEMENT shininess            (param)>
<!ELEMENT emissiveColor        (param)>
<!ELEMENT ambientColor         (param)>
<!ELEMENT specularColor        (param)>
<!ELEMENT funct                (%changing_function; | %display_functions;)*>
<!ELEMENT box                  (%changing_function; | param)*>
<!ELEMENT box3d                (%changing_function; | param)*>
<!ELEMENT text                 (%changing_function; | param)*>
<!ELEMENT text3d               (%changing_function; | param)*>
<!ELEMENT param                (#PCDATA)>
```

Abbildung 95: DTD

Die in diesem Kapitel vorgestellten Konzepte werden im folgenden Kapitel anhand der Implementierung eines funktionsfähigen Prototyps verifiziert.

Kapitel 5

KONZEPTVERIFIKATION

Dieses Kapitel beschreibt eine Implementierung der bereits vorgestellten Konzepte. Zuerst wird die Systemarchitektur, die aus dem Softwaredesign resultiert, beschrieben. Der zweite Abschnitt geht explizit auf einige ausgewählte Themenstellung der Implementierung ein, die für das Konzept von Bedeutung sind und die im Rahmen der Projektarbeit so realisiert worden sind. Der letzte Abschnitt beschreibt ein konkretes Anwendungsszenario aus dem Automotive-Bereich, das zur Konzeptverifikation herangezogen wurde und die Leistungsfähigkeit des Ansatzes verdeutlicht.

5.1 Systemarchitektur

Abbildung 96 beschreibt eine Client-Server-Architektur eines auf Multicasting beruhenden Grafik-Clusters mit vier Projektionssystemen.

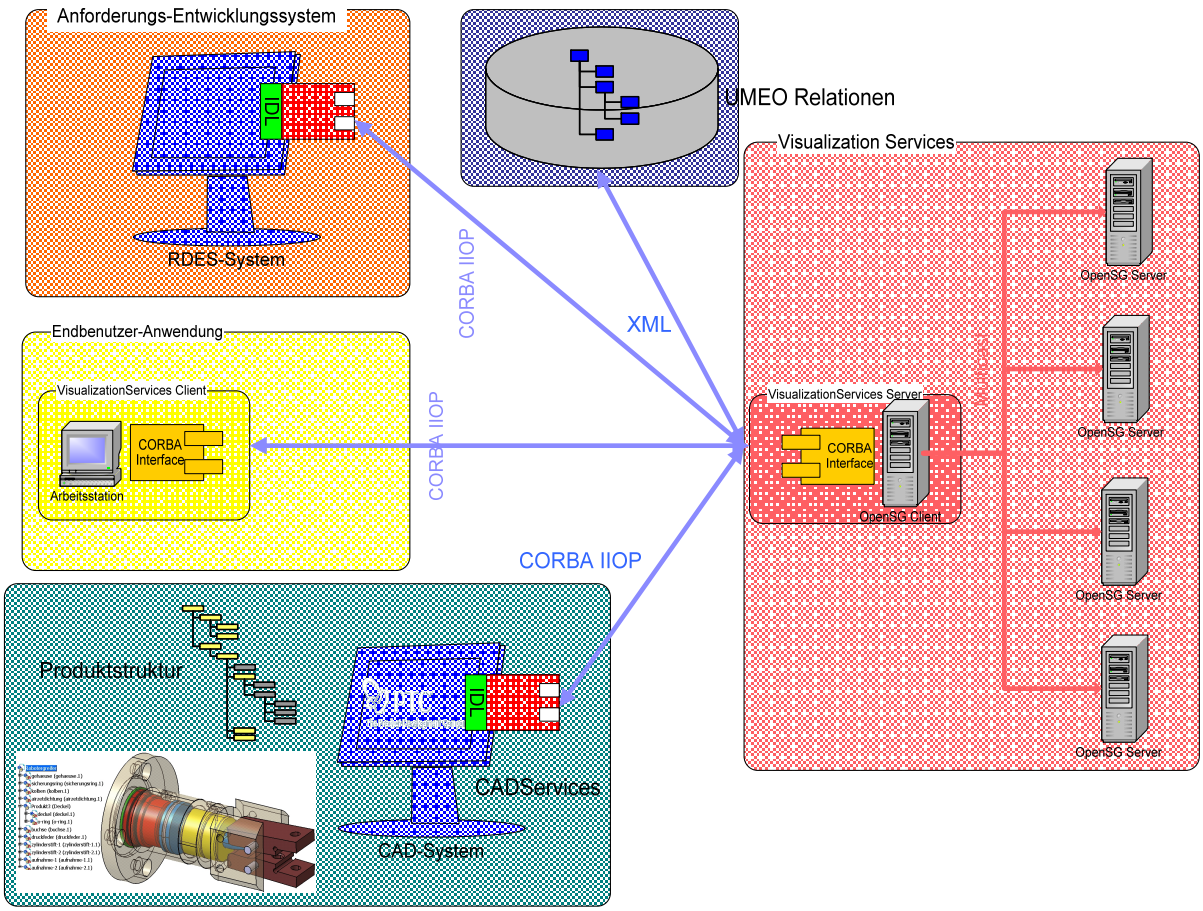


Abbildung 96: Zusammenspiel der Quellsysteme beim Visualisierungprozess

Eine Client-Anwendung (Visualisierung) kommuniziert über den CORBA IIOP mit dem Visualization-Services-Server. Aus Sicht des Grafik-Subsystems OpenSG ist der Visualization Services Server jedoch der Client innerhalb des Grafik Clusters, der seine Dienste von den Server-Applikationen (OpenSG Server) anfordert. Die Kommunikation dieser beiden Komponenten (OpenSG Server und Client) geschieht über Multicasting. Ein "Multicast"-Paket ist ein Netzwerk-Paket, das an mehr als einen, aber nicht alle Rechner im Netzwerk geschickt werden soll. Diese Funktionalität wird dadurch erreicht, dass spezielle Hardware-Adressen an Rechnergruppen zugewiesen werden. Pakete, die an eine dieser besonderen Adressen geschickt werden, sollten von allen Rechnern in der Gruppe empfangen werden. Die Übertragung von Multicast-Paketen ist einfach, weil sie genau wie jedes andere TCP/IP Paket aussehen. Die Schnittstelle überträgt die Pakete über das Kommunikationsmedium, ohne sich die Ziel-Adresse anzusehen. Die OpenSG Server rendern ihrerseits den Scene-Graph je nach Kameraposition im Universum und liefern direkt das VGA-Signal für die Projektoren. Im Fall einer Stereo-Projektion kann ein einzelner OpenSG Server so konfiguriert sein, dass er ein Signal für die Stereo-Projektion erzeugt und zwischen Projektor und VGA-Ausgang der Grafikkarte noch eine sog. Splitter-Box geschaltet ist (Aktiv-Stereo), oder man fasst je zwei Projektoren für eine Projektionswand zusammen, die dann auf eine Leinwand projizieren (Passiv-Stereo). Die Splitterbox hat hierbei die Aufgabe anhand des Sync-Signals die beiden Teilbilder zu trennen und an den jeweiligen Projektor zu schicken. Die Shutter-Brille wird in der gleichen Frequenz getaktet.

Der Informationsfluss innerhalb des Szenarios funktioniert folgendermaßen: Bauteilgeometrie sowie Produktstruktur bezieht der Visualisierungs-Server aus den CAD-Services. Lädt eine Client-Anwendung nun mit der Bauteilgeometrie verbundene Metainformationen (in Abbildung 96 repräsentiert durch ein Anforderungsmodellierungssystem) werden genau diese Metainformationen über eine Methodenschnittstelle geladen, für die Relationen im UMEO Modell existieren. Die Relationen in der UMEO Datenbank referenzieren also die mit der Wirkgeometrie verbundenen Informationen via PIDs.

5.2 Implementierung

Folgende Abschnitte gehen detailliert auf implementierungsspezifische Details der Software ein, die im Rahmen dieser Arbeit entwickelt wurde. Implementierungssprache ist C++ [82]. Zuerst wird erläutert, wie der Zugriff auf die CAD-Bauteilgeometrie realisiert wird. Danach folgt die Implementierung der Visualization-Services.

5.2.1 Zugriff auf die CAD Bauteilgeometrie über die CAD Services

Im Folgenden wird detailliert erläutert, wie der Zugriff auf die B-Rep Bauteilgeometrie über die CAD Services API realisiert wird. Zuerst wird eine Verbindung zu einem existierenden System über einen Nameserver hergestellt (Quelltext 1):

```
lHost = nameserver_host;
lPort = nameserver_port;
lIOR = "corbaloc:iiop:" + lHost + ":" + lPort + "/NameService";
…
lProperties = new OB::Properties();
lProperties->setProperty("ooc.orb.service.NameService", lIOR.c_str());
…
lNameServiceObj = gOrb->resolve initial references("NameService");
lNamingContext = CosNaming::NamingContext:: narrow(lNameServiceObj);
…
try
{
  lObj = lNamingContext->resolve(lBindName);
  if(!CORBA::is nil(lObj))
  {
    lCadServer = CadConnection::CadServer:: narrow(lObj);

    if(!CORBA::is_nil(lCadServer))
      lCadSystem=lCadServer->connect(CosPropertyService::Properties());
    else
        …
  }
…
}

catch(CORBA::Exception &ex)
{
  …
};
```

Quelltext 1: Herstellen der Verbindung zu einem Server

Nachdem die Verbindung zu einem Server aufgebaut ist, werden die existierenden Modelle abgefragt und wird ein Modell geladen (Quelltext 2).

```
lModelNames = lCadSystem->available models();
for(int i = 0; i < lModelNames->length(); i++)
  lModelList[i] = strdup(lModelNames[i].ptr());

…

try
{
  lModel = lCadSystem->open_model(modelname, CadConnection::ACTIVE_READONLY);
}
```

Quelltext 2: Ermitteln der vorhandenen Modellnamen und öffnen eines Models

Sobald das Modell geladen ist, kann die Modellgeometrie in einem zweistufigen Prozess abgefragt werden. Im oberen Code-Segment werden zuerst alle Einzelteile (Parts) eines Zusammenbaus (Assembly) mit der Funktion `model_children()` ermittelt (Quelltext 3). In einer Schleife über alle Einzelteile wird dann die Modellgeometrie eines Bauteils mit der dazugehörigen relativen Position im Weltkoordinatensystem gelesen. Das Bauteil-Koordinatensystem wird durch drei

Vektoren festgelegt. Mittels einer Berechnungsvorschrift (vgl. Gleichung 1) wird aus zwei Vektoren ein Orthogonalsystem hergestellt. Der dritte Vektor bestimmt die Translation im Weltkoordinatensystem.

$$\vec{i} = \vec{x} - (\vec{x} \cdot \vec{y})\vec{y}$$
$$\vec{j} = \vec{y} \times \vec{i}$$
$$\vec{k} = \vec{y}$$

Gleichung 1: Berechung des Bauteilkoordinatensystems relativ zum Weltkoordinatensystem

Um ein Bauteil zu lesen, müssen zunächst alle Objekte in der höchsten Hierarchiestufe mit der Funktion `top_level_entities()` ausgewertet werden. Betrachtet werden lediglich Bodies und Shells. In beiden Fällen wird mit der Methode `tesselate()` aus der ursprünglichen Fläche eine tessellierte Oberfläche erzeugt. Für jede dieser Flächen wird in einer Schleife die Methode `drawSurfaceTesselationWithNormals()` aufgerufen. Diese Methode ist eine virtuell überladene Methode[53] des jeweiligen Grafik-Subsystems und visualisiert die Fläche, bestehend aus n Dreiecken und $9n$ Punktkoordinaten. Auf die Verwaltung von Indexlisten und der damit einhergehenden Reduktion der notwendigen Punktkoordinaten wurde bei diesem Prototyp zugunsten der Einfachheit und Performance verzichtet. Die Flächennormalen werden ebenfalls ausgewertet, um Lichtreflexionen auf der Oberfläche berechnen zu können.

```
// hole alle child models (parts) aus dem model (assembly)
model_instances = (lModel)->model_children();
if((model_instances->length()) > 0) // es gibt also parts
{
  // Schleife ueber alle parts
  for(int i=0; i < model instances->length(); i++)
  {
    part = (*model_instances)[i]->component();
    part_position = (*model_instances)[i]->location();

    drawTesselatedModel(&part, &part position);                                    }
}

…

// hole alle toplevel-entities
lEntities = (*model)->top level entities(CadUtility::TypeCodeSeq());
// Schleife ueber alle toplevel-entities ...
for(register int i = 0; i < lEntities->length(); i++)
{
  // test ob entity ein body ist ...
  if((*lEntities)[i]-> is a(CadBrep:: tc Body->id()))
  {
    CadBrep::Body* pActiveBody = CadBrep::Body:: narrow((*lEntities)[i]);
    pConnFaceTess =
      CadBrep::Body::_narrow(pActiveBody)->tessellate(cType,lParams, lParamsChanged);
    for(register int j = 0; j < pConnFaceTess->all_faces.length(); j++)
    {
```

[53] Eine Methodenüberladung tritt auf, wenn eine Klasse (oder eine abgeleitete Klasse) zwei Methoden mit dem gleichen Namen, aber unterschiedlichen Signaturen enthält. Methodenüberladungen werden verwendet, um unterschiedliche Methoden bereitzustellen, die semantisch die gleiche Funktion haben.

```
      // lese die Tesselierungsdaten ...
      drawSurfaceTesselationWithNormals(pConnFaceTess -> all_faces[j].face_tessellation);
    }
  }
  // test ob entity eine shell ist ...
  else if((*lEntities)[i]-> is a(CadBrep:: tc Shell->id()))
  {
    CadBrep::Shell* pActiveShell = CadBrep::Shell::_narrow((*lEntities)[i]);
    pFaceTessSeq =
          CadBrep::Shell::_narrow(pActiveShell)->tessellate(cType, lParams, lParamsChanged);
    //setShellDefaultColor();
   for(register int j = 0; j < pFaceTessSeq->length(); j++)
   {
     // lese die Tesselierungsdaten ...
     drawSurfaceTesselationWithNormals (((*pFaceTessSeq)[j]).face_tessellation);
   }

  }

  …

}
```

Quelltext 3: Lesen eines Assembly und eines Parts

5.2.2 Implementierung der Visualization-Services

Startpunkt der Implementierung ist das Initialisieren des ORBs. Anschließend wird eine Referenz auf den Persistent Object Adapter (POA) gesucht und somit der POA Manager ermittelt. Im nächsten Schritt wird das wichtigste Objekt `CadServer_Impl` instanziiert und dem POA übergeben. Dies ist die erste Referenz, die eine Client Applikation der Visualization-Services ermittelt. Die Referenz wird auf dem CORBA Name-Server gesucht und der logische Name „`CADServicesACIS`“ des Dienstes angemeldet. Danach wird nur noch der POA Manager aktiviert und der ORB gestartet (Quelltext 4).

```
#include <CORBA.h>
#include <coss/CosNaming.h>
#include "account.h"
#include "account_impl.h"
#include "CadConnection_impl.h"

using namespace std;

int main (int argc, char *argv[])
{
  cout << "*** CADServices for ACIS Version 1.0 ***\n\n" << flush;

  /*
   * Initialize ORB, get the Root POA, and register Account object as usual
   */

  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  /*
   * Obtain a reference to the RootPOA and its Manager
   */

  CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
  PortableServer::POAManager_var mgr = poa->the_POAManager();
```

```
  // Create a CADServer and activate it ...

  CadServer_impl* CadServerACIS = new CadServer_impl;
  PortableServer::ObjectId_var cadserver_oid = poa->activate_object (CadServerACIS);
  CORBA::Object_var cadserver_ref = poa->id_to_reference (cadserver_oid.in());


  /*
   * Acquire a reference to the Naming Service
   */

  CORBA::Object_var nsobj = orb->resolve_initial_references ("NameService");
  CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow (nsobj);

  if (CORBA::is_nil (nc)) {
    cerr << "oops, I cannot access the Naming Service!" << endl;
    exit (1);
  }

  /*
   * Construct Naming Service name for our service
   */

  CosNaming::Name name;
  name.length (1);
  name[0].id = CORBA::string_dup ("CADServicesACIS");
  name[0].kind = CORBA::string_dup ("");

  /*
   * Store a reference to our CADServer in the Naming Service. We use 'rebind'
   * here instead of 'bind', because rebind does not complain if the desired
   * name " CADServicesACIS" is already registered, but silently overwrites it (the
   * existing reference is probably from an old incarnation of this server).
   */

  cout << "Binding CADServicesACIS in the Naming Service ... " << flush;
  nc->rebind (name, ref);
  cout << "done." << endl;

  /*
   * Activate the POA and start serving requests
   */

  printf ("Running.\n");

  mgr->activate ();
  orb->run();

  /*
   * Shutdown (never reached)
   */

  poa->destroy (TRUE, TRUE);
  delete CadServerACIS;

  return 0;
}
```

Quelltext 4: main-Funktion der VisualizationServices

5.2.3 Präsentationsgraph

Der Aufbau des Präsentationsgraphen ist so implementiert, dass ein Iterator[54] rekursiv den Baum durchläuft und für jedes Part eines Assembly (Toplevel Entity) einen `GroupNode` erzeugt. Unterhalb dieses `GroupNodes` wird für jeden Body eines Assembly wiederum ein `GroupNode` erzeugt. Einzig Flächen (Faces) und Kanten (Vertices) erzeugen Geometrieelemente, die unterhalb eines `GroupNode` eines Bodys in den Scene-Graph eingefügt werden. Wird nun zu einer Fläche eine grafische Metapher hinzugefügt, so wird diese zuerst aus der Bibliothek mittels eines VRML Templates instanziiert und dann in den gleichen `GroupNode` eingefügt. Abbildung 97 verdeutlicht die Vorgehensweise. Primär sind Flächen im Geometriemodell und im Szenen-Graph per Attributierung mit der gleichen persistenten ID[55] (PID) versehen. Die Verbindung zu den zugehörigen Metainformationen wird erst über die UMEO Relationen hergestellt.

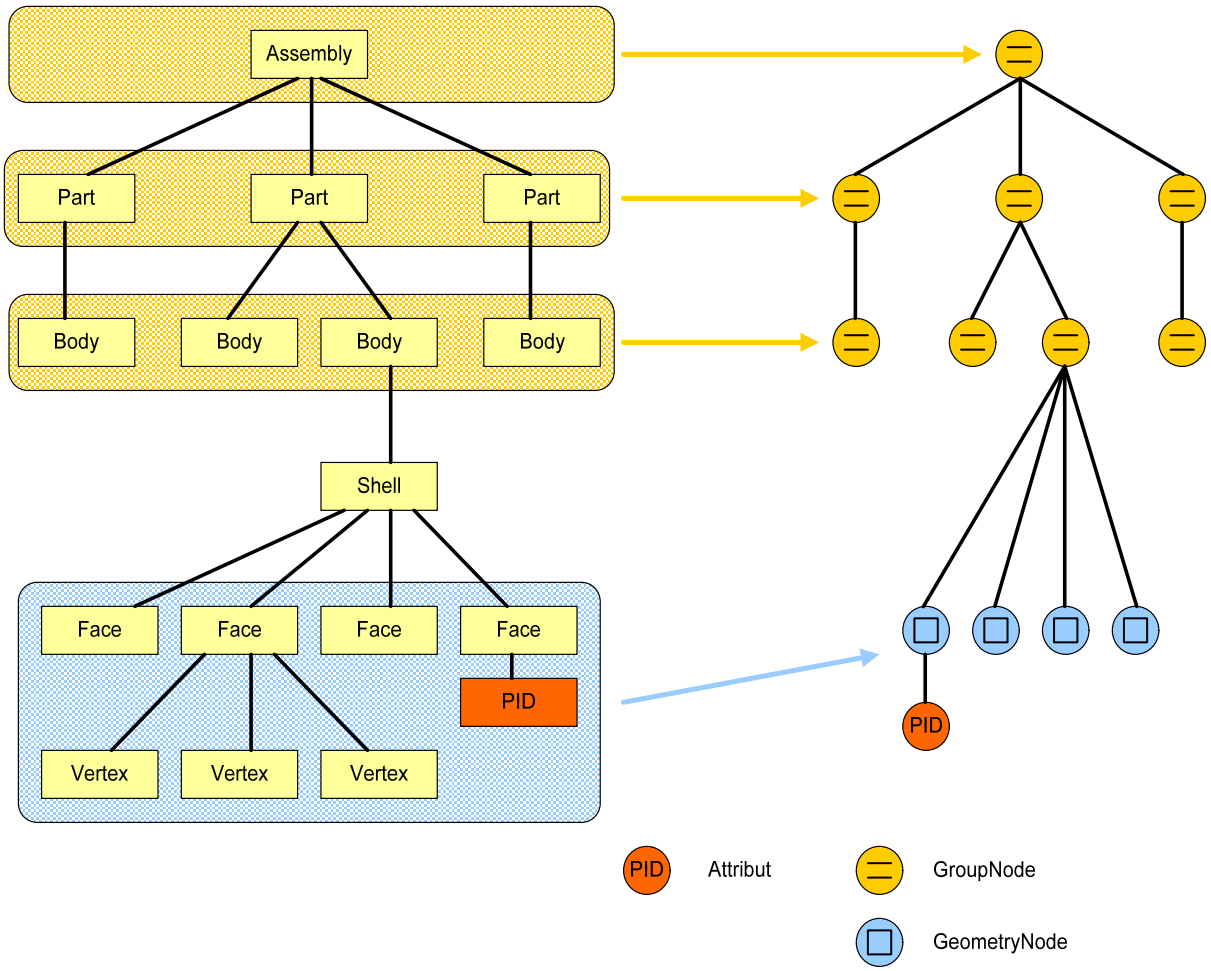


Abbildung 97: Baumstruktur des B-Reps und des dazugehörigen Szene-Graphen

[54] Iteratoren sind Methoden die ihrerseits Funktionen als Argument bekommen und diese Funktion rekursiv auf jeden Knoten eines dynamischen Datentyps anwenden.

[55] Bezeichner

5.2.4 Benutzungsschnittstelle der Client-Komponente

Im Rahmen dieser Arbeit wurde auch eine Client Komponente auf Basis von OpenSG und QT[56] entwickelt, die die Visualization-Services interaktiv ansteuern kann. Sie kann z.B. verwendet werden, um während einer Präsentation von einer Arbeitsstation aus die Visualisierung zu steuern.

Zunächst wird die Produktgeometrie interaktiv von einem Server im Netzwerk (Inter- oder Intranet) geladen. Dazu wird zuerst die Server-Verbindung aufgebaut und dann interaktiv abfragt welche CAD-Modelle auf dem Server verfügbar sind. Abbildung 98 zeigt einen Screenshot des Dialoges und die notwendigen Parameter. Um eine Verbindung aufzubauen ist lediglich der Name des Servers auf dem der CORBA-Naming Service zu finden ist, die Portnummer und der logische Name des Dienstes (in diesem Fall eine CAD-Services Implementierung für Pro/ENGINEER) anzugeben. Optional ist auch noch die Angabe von Benutzerkennung und Passwort für eine Zugriffskontrolle möglich.

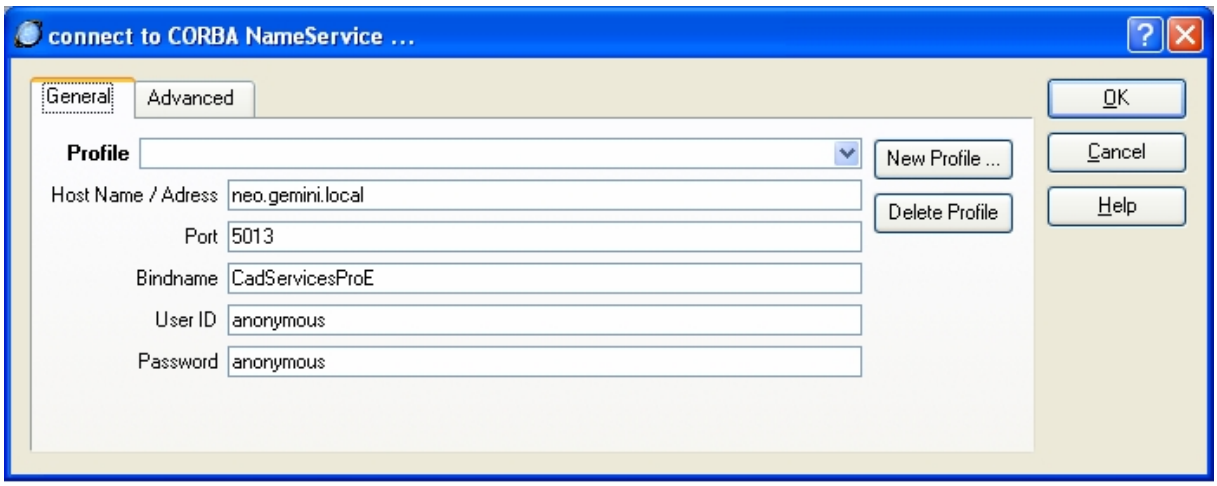


Abbildung 98: Server Connect-Dialog

Um die Verwaltung von Zugriffsprofilen auf den Server zu vereinfachen, kann man diese lokal abspeichern.

Das nachfolgende Laden eines Modells geschieht derart, dass zunächst angefragt wird, welche Modelle auf dem Server zur Verfügung stehen, um dann selektiv das gewünschte Modell auszuwählen (vgl. Abbildung 99). Wie in Kapitel 5.2.1 beschrieben, wird die Bauteilgeometrie in Form von tessellierten Oberflächendaten geladen, wobei trotzdem zu jedem Zeitpunkt der Bezug zur Original B-Rep-Fläche durch die persistenten Bezeichner erhalten bleibt. Zur Tessellierung der Fläche können Methoden der CAD-Services herangezogen werden, bzw. wurden in den

[56] QT Toolkit http://www.trolltech.no

Visualization-Services eigene, für spezielle Aufgabengebiete optimierte, Mesher[57] implementiert. Während des Ladens der Geometrie lässt sich die Qualität der tessellierten Daten durch die drei Schieberegler festlegen. Diese Regler definieren Maximalwerte für spezielle charakteristische Werte der Abweichung der Genauigkeit einer tessellierten Oberfläche gemessen an ihrem mathematischen Ideal. Wird einer dieser Grenzwerte überschritten wird eine feinere Tessellierung (mit mehr Dreiecken) erzeugt.

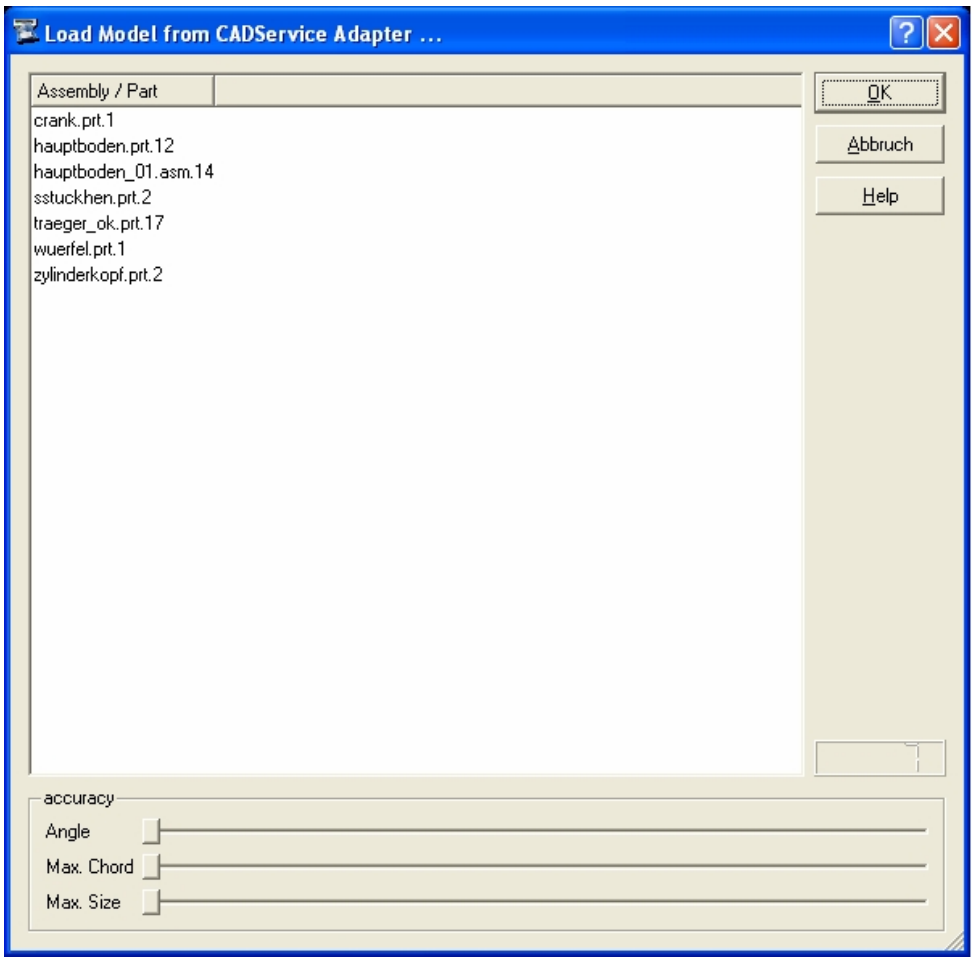


Abbildung 99: Screenshot Load-Model-Dialog

Abbildung 100 zeigt einen Screenshot der Client-Komponente (vgl. Abbildung 59, Abbildung 60) der Visualization-Services. Sichtbar ist ein Assembly (Zusammenbau) eines vereinfachten Unterbodens eines DaimlerChrysler[58] A-Klasse Personenkraftwagens.

[57] Softwarekomponente zum Erzeugen von tesselierten Oberflächendaten aus Freiformflächen

[58] Testdaten mit freundlicher Genehmigung der DaimlerChrysler AG im Rahmen des ITEA-DLR Verbundprojektes 3D Workbench

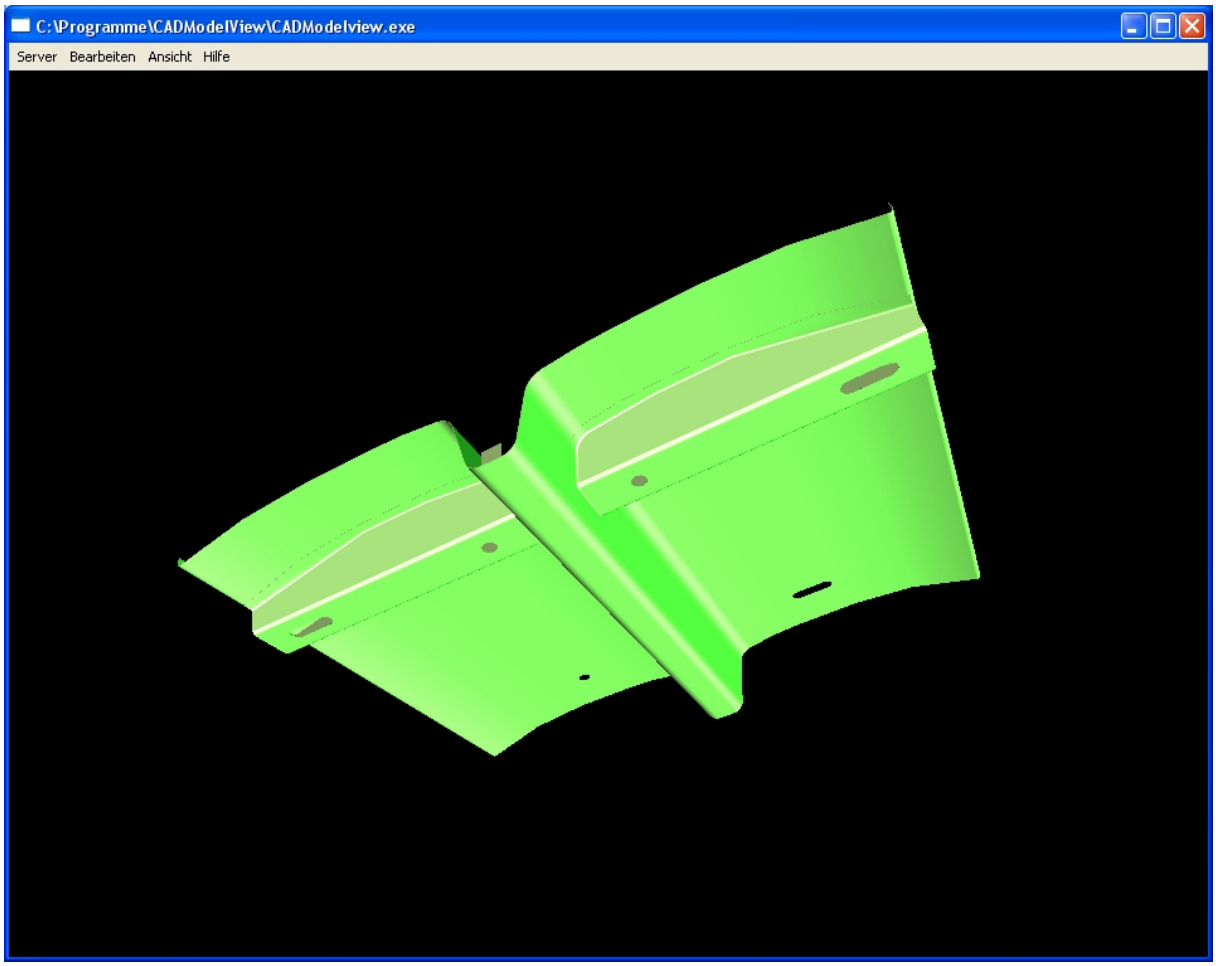


Abbildung 100: Screenshot der Client-Komponente der Visualization-Services

Nach dem Laden der Bauteilgeometrie werden die Metainformationen im UMEO-Format (siehe Kapitel 3.2) geladen (siehe Abbildung 101). Die Metainformationen stehen hierbei in Form einer XML-Datei zu Verfügung, die die Instanzen der UMEO-Objekte enthält, und zwei DTD[59]-Dateien, die diese Datei validieren. Um eine Visualisierung zu erzeugen, muss zuletzt noch das Visualisierungsscript für den relevanten Anwendungsfall selektiert werden (siehe Abbildung 102). Die einzelnen Visualisierungsskripte sind innerhalb der Datenbank hierarchisch geordnet. Die technische Umsetzung im Prototyp ist noch auf Dateiebene, in der kommerziellen Ausbaustufe soll jedoch ein Zugriff auf eine mysql-Datenbank implementiert werden, in der diese Skripte dann unternehmensweit abgelegt und bei Bedarf herangezogen werden können. Neue Scripte sind dann importierbar durch die Client-Schnittstelle und stehen anderen Mitarbeitern zu Verfügung.

[59] DTD (engl. Document Type Definition) siehe Kapitel 2.11.2

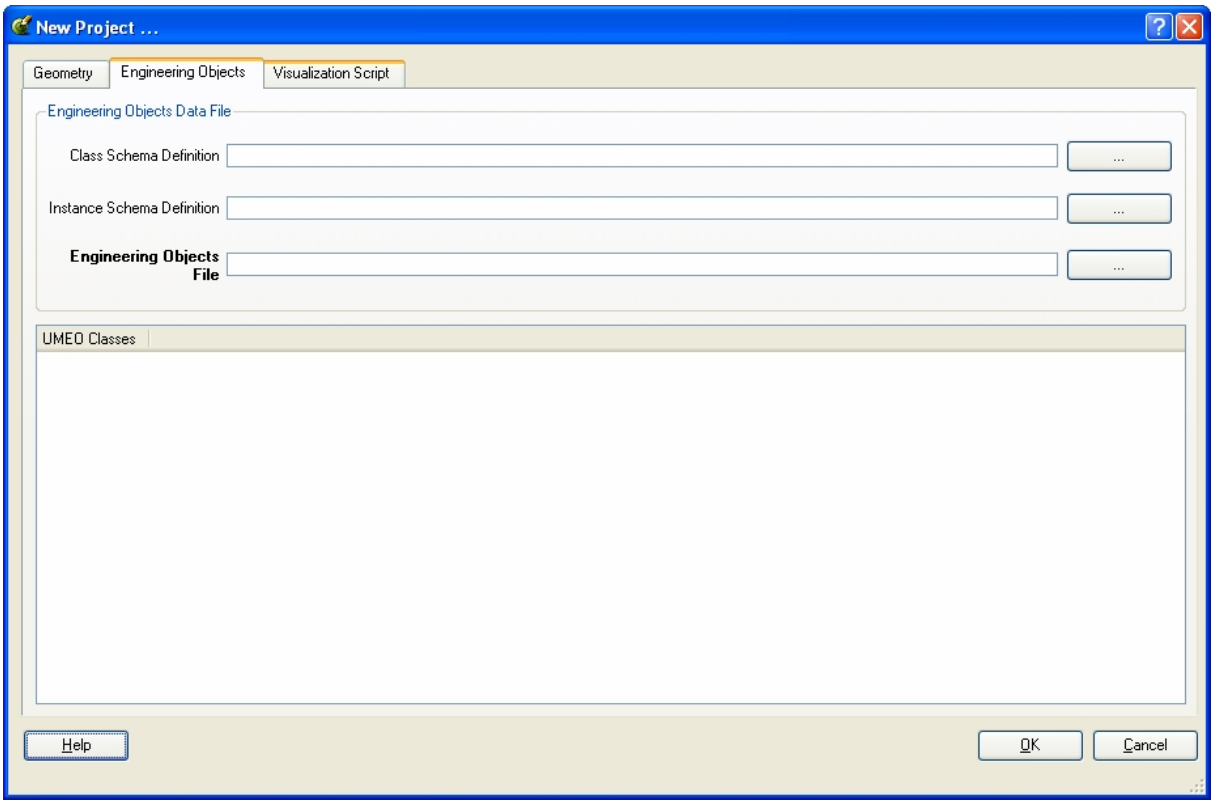


Abbildung 101: Laden der Metainformationen als UMEO-Klassen im XML-Format

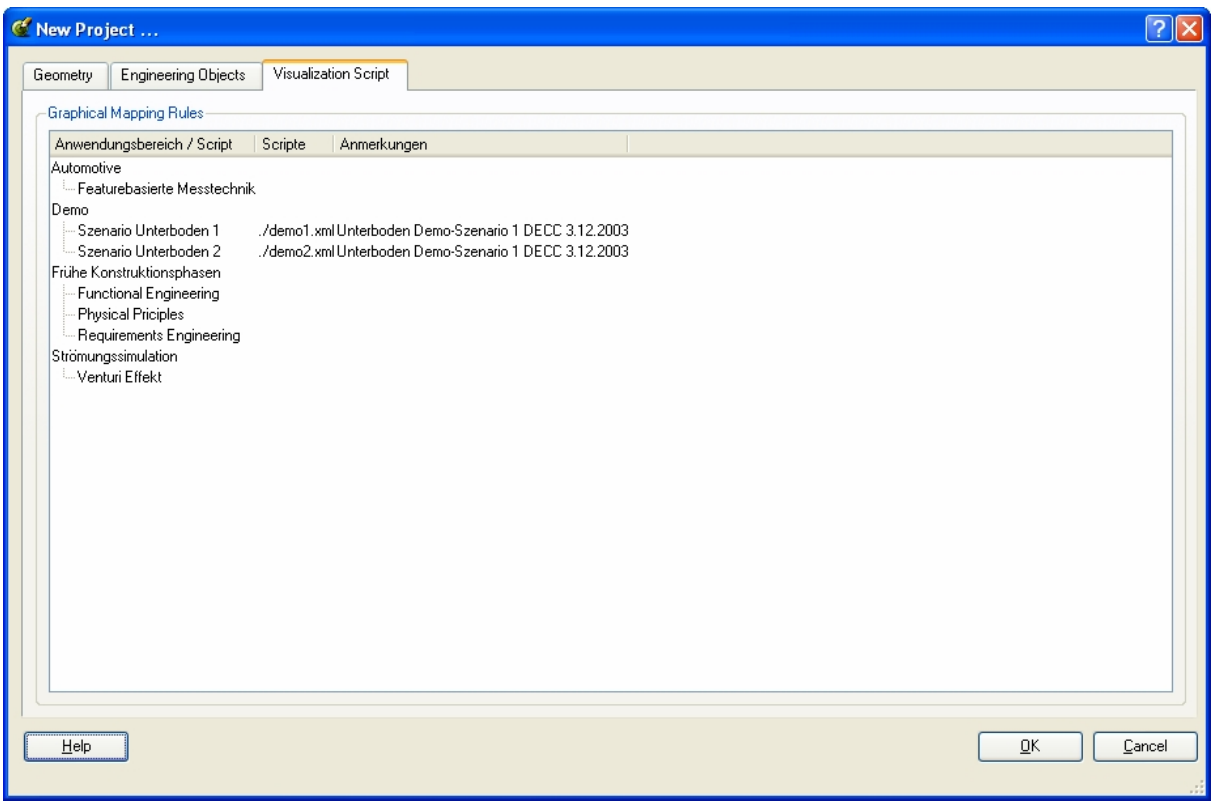


Abbildung 102: Spezifikation des Visualisierungs-Skriptes

Als Resultat wird der Szenengraph neu erzeugt und damit auch eine neue Visualisierung. Abbildung 103 zeigt exemplarisch einen Hauptboden mit drei Features. Die Features sind Bohrung, Sicke und dünnwandiges Element (vgl. Abbildung 67 - Abbildung 73).

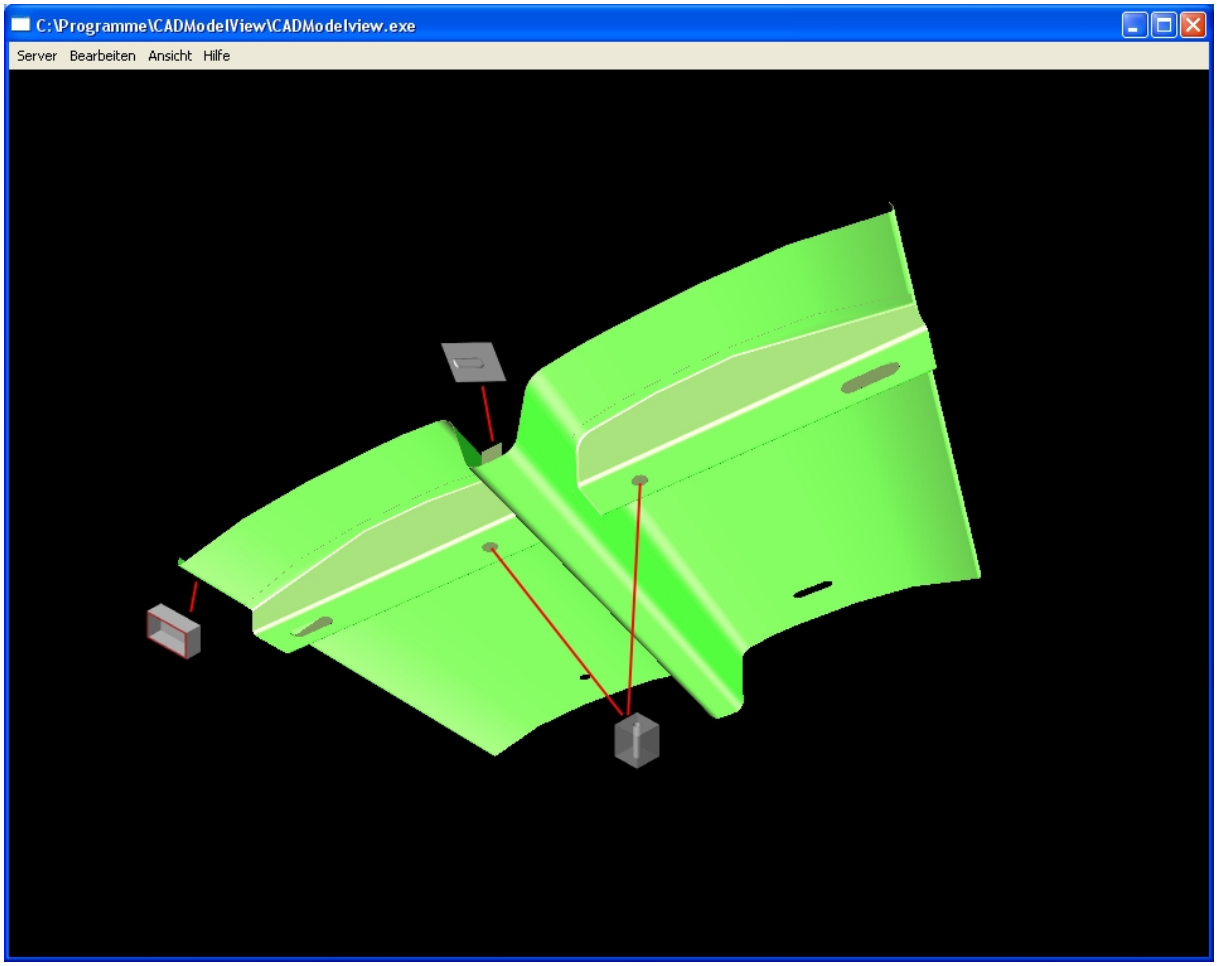


Abbildung 103: Integrierte Visualisierung von Feature-Informationen
(Bohrung, Sicke, dünnwandiges Element) aus UMEO

Die erweiterten Visualisierungen sind jetzt bereits in den Szenen-Graph eingebaut, jedoch nicht sichtbar. Abbildung 104 zeigt eine Visualisierung mit einem selektierten Feature (Gewindebohrung). Die vernetzte Kugelstruktur zeigt das selektierte Feature in roter Farbe und alle verbundenen Features durch ein Verbindungssegment. In der Darstellung des Bauteils ist die Gewindebohrung ebenfalls rot selektiert. Eine Selektion eines Kugelkörpers würde umgekehrt die bezogene Bauteilgeometrie farblich hervorheben. Bei Bedarf können am rechten Rand die quantitativen Werte in Form von Text angezeigt werden.

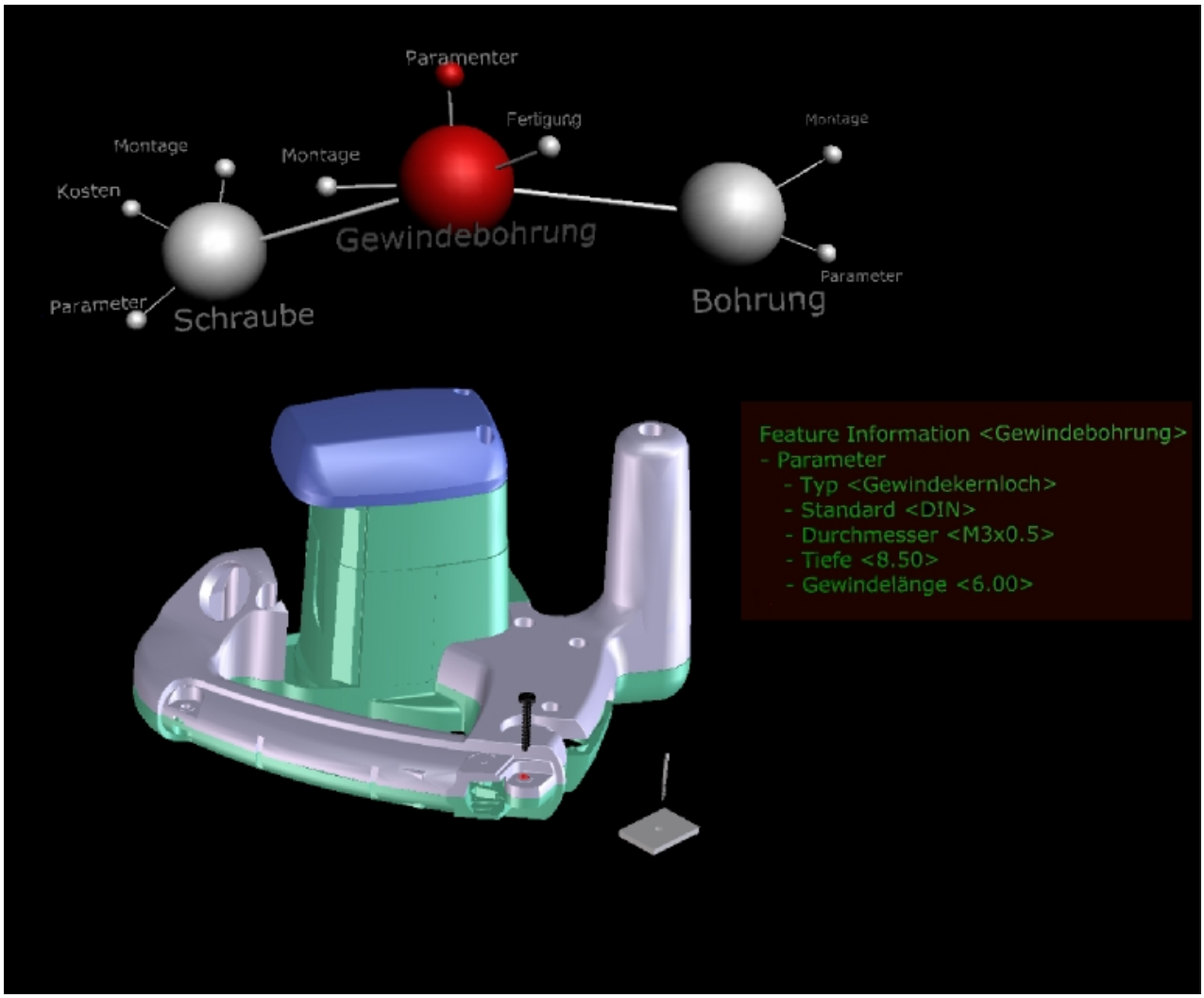


Abbildung 104: Visualisierung von Features Informationen am Beispiel einer Gewindebohrung

5.3 Anwendungsszenario

Neben dem Einsatz der in Kapitel 5.2.4 beschriebenen Client-Komponente ist auch der direkte Einsatz der Visualization-Services ohne die Client-Komponente möglich. Folgendes Anwendungsszenario beschreibt, wie die Visualization-Services innerhalb des Frameworks eingesetzt werden können, um Metainformationen in Form von Anforderungen aus einer Anforderungsmodellierungskomponente zu visualisieren. Abbildung 105 gibt einen schematischen Überblick.

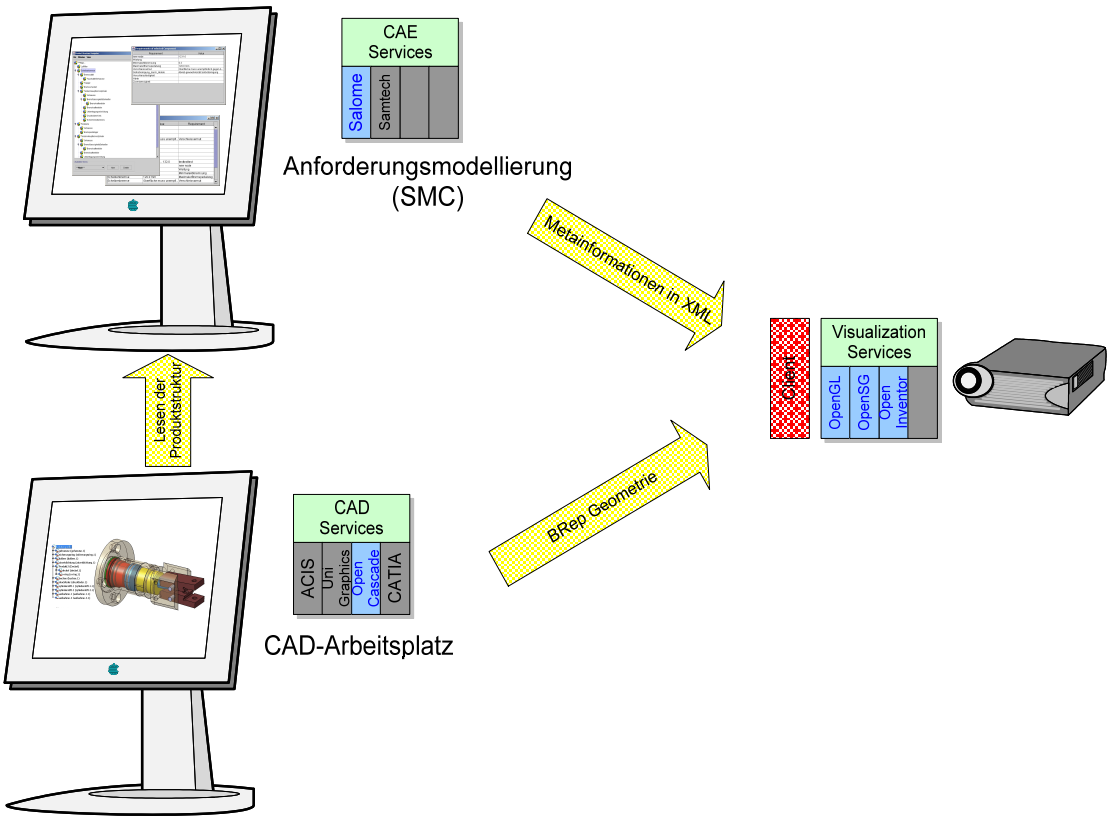


Abbildung 105: Integration der Anforderungsmodellierung

Das Szenario geht von einer existierenden Produktstruktur aus. Im industriellen Umfeld ist es häufig so, dass für die Konstruktionen vorgefertigte Produktstrukturen zur Verfügung stehen (sog. Templates), da die Firmen gewisses Know-how über den generellen Aufbau ihrer Produkte besitzen. Selbst wenn es sich um eine Neukonstruktion handelt, sollte das Produkt zuerst grob strukturiert werden, bevor Produktanforderungen damit verknüpft werden.

Im ersten Schritt wird über die Methodenschnittstelle des CAD-Services die Anforderungs-modellierungskomponente (SMC[60]) geladen. Lazarov[61] beschreibt die Integration der CAD-Service-Schnittstelle in die Anforderungsmodellierungskomponente.

In Schritt zwei werden Anforderungen mit der Produktstrukur über persistente Identifikatoren (Bezeichner) verbunden, so dass sie über die „Lebenszeit“ dieser Struktur (Bauteile, Körper bis hin zu Flächen) erhalten bleiben. Der Begriff Lebenszeit bezieht sich hierbei auf das Vorhandensein dieser Identifikatoren auch nach mehreren Speicher-/Lade-Zyklen, wie sie z.B. im Zusammenhang

[60] SMC (engl. Specification Modelling Component), diese Komponente wurde am Institut für Rechneranwendung in Planung und Konstruktion von Herrn O. Klaar entwickelt [10].

[61] D. Lazarov, "Integration der Anforderungsentwicklung mit Werkzeugen zur Produktmodellierung am Beispiel Computer Aided Design (CAD)," in Institut für Rechneranwendung in Planung und Konstruktion (TH). Karlsruhe: Universität Karlsruhe (TH), 2004.

mit PDM-Systemen vorkommen. Über die Methodenschnittstelle der Anforderungsmodellierungskomponente kann nach Fertigstellung ein XML-Datenstrom abgefragt werden, der genau diese Anforderungen mit den Verknüpfungen zur Produktstruktur enthält.

Im nächsten Schritt können nun die Visualization-Services über eine Clientkomponente vom Anforderungsmodellierungssystem oder direkt vom Benutzer gestartet werden. Es wird zunächst die Bauteilgeometrie über die CAD-Services geladen. Nach dem Selektieren eines geeigneten Abbildungsskriptes wird der XML-Datenstrom des Anforderungsmodellierungssystems gelesen. Werden in diesem Datenstrom Instanzen von Klassen identifiziert, für die eine grafische Repräsentation in dem Abbildungsskript existiert, werden diese instanziiert und zusammen mit den Geometrie-Knoten in den Szene-Graph eingefügt.

Abbildung 106 zeigt ein Beispiel einer solchen Visualisierung. Es handelt sich hierbei um das Gehäuse einer Kreissäge. Mit dem grün selektierten Bereich des Gehäuses sind Anforderungen bezüglich des Materials verknüpft. Selektiert der Benutzer diesen Teil des Gehäuses, so erscheint die grafische Metapher für Werkstoffanforderungen (vgl. Abschnitt 4.3.1). Über dem Gehäuse erscheint eine Struktur von Kugelobjekten mit Verbindungen. Zentral angeordnet und rot selektiert ist die grafische Repräsentation der Materialanforderungen. Um diese Anforderung herum angeordnet sind alle bezogenen Anforderungen, die in Wechselwirkung mit dieser Anforderung stehen. Selektiert man eine solche bezogene Anforderung, so werden die betroffenen Bauteilgeometrien grün markiert. So lässt sich ein eindeutiger Bezug zwischen Anforderungen und der Produktgeometrien, auf die diese Anforderung wirkt, herstellen. Bei Bedarf können im linken Bereich noch die quantitativen Ausprägungen der Anforderungen visualisiert werden.

Die Art und Weise der grafischen Repräsentation ist völlig wahlfrei in dem Abbildungsskript definiert und kann zur Laufzeit in jede beliebige andere Darstellungsart geändert werden. Das einfache Ändern der grafischen Metaphern z.B. von der Konstruktion in die der Montage ist nur ein Beispiel. Auch das Ausblenden von für diesen Anwendungsfall unwichtigen Daten ist einfach möglich, indem man der Metainformationsklasse einfach keine grafische Metapher zuweist.

Dieses Beispiel zeigt anschaulich die Leistungsfähigkeit dieses pragmatischen Konzeptes der Abbildungsskripte. Durch die Anwendung dieser Software wird in einer Firma über die Zeit eine Reihe von verschiedenen Abbildungsskripten zur Verfügung stehen, die die wichtigsten Anwendungsfälle, angepasst an die firmenspezifischen Besonderheiten, abbilden.

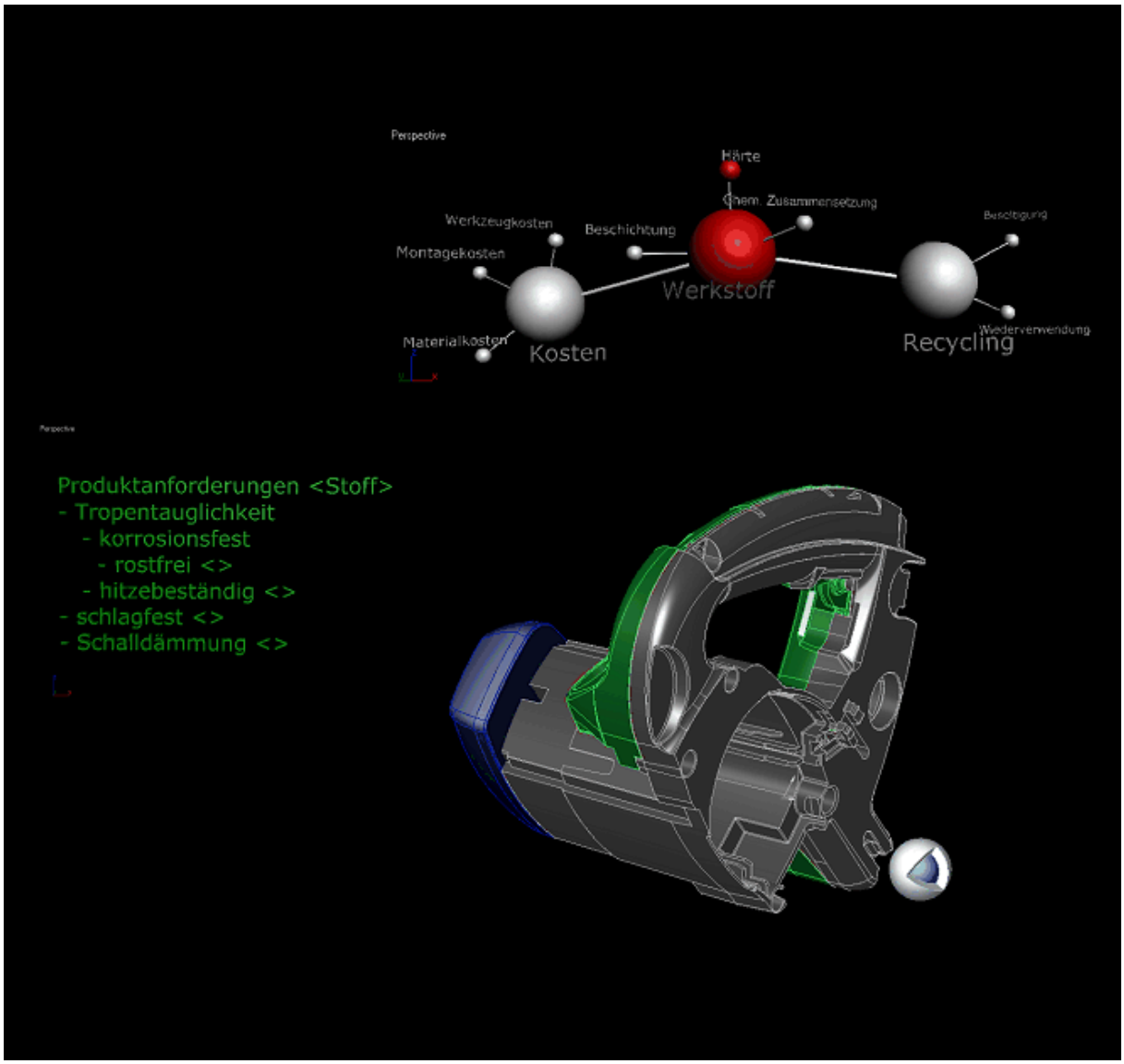


Abbildung 106: Anwendungsbeispiel für die Verknüpfung von Produktgeometrie und Produktanforderungen

Kapitel 6

ZUSAMMENFASSUNG UND AUSBLICK

6.1 Zusammenfassung

Unternehmen im Maschinen- und Anlagenbau leiden neben dem für alle gültigen Kostendruck auch, bedingt durch Globalisierung, unter einer erheblich verschärften Wettbewerbssituation. Auf Marktanforderungen muss flexibel reagiert werden bei gleichzeitig wachsender Komplexität der Produkte. Unternehmenskonzentration und Auslagerung von Prozessen oder kompletten Unternehmensbereichen haben in den letzten Jahren den Bedarf geweckt, die Kernprozesse der Fertigungsindustrie besser miteinander zu integrieren, zu optimieren und dazu die Möglichkeiten moderner Technologien noch effektiver zu nutzen. Die wachsende Komplexität der Produkte und die immer stärkere Produktindividualisierung führen, bei immer kürzeren Produktentwicklungszeiten, zu einem außerordentlich schwer zu beherrschenden Abhängigkeitsgeflecht von Informationen in der Produktentwicklung. Dem Erkennen und Verstehen dieser Abhängigkeiten durch Visualisierung, auch über die klassischen Domänengrenzen (wie z.B. Mechanik, Elektronik, Software, …) hinweg, kommt deshalb eine zentrale Bedeutung bei der Produktdefinition zu. Insbesondere existieren heute keine Ansätze Informationen aus frühen Konstruktionsphasen, wie z.B. der Anforderungsmodellierung, integriert mit der sich daraus ergebenden Bauteilgestalt zu visualisieren.

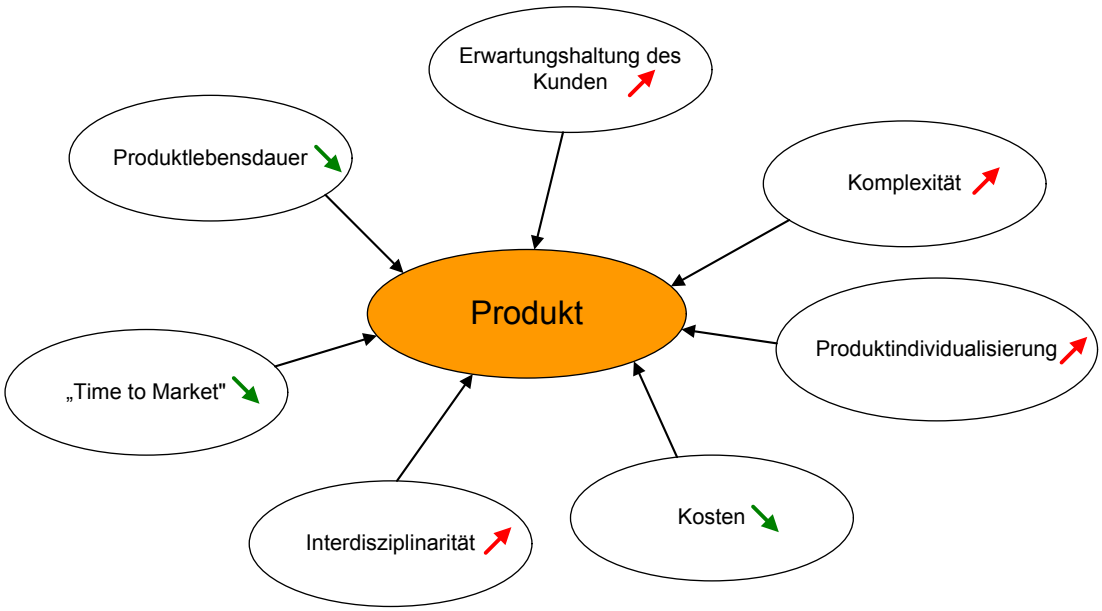


Abbildung 107: Spannungsfeld Produkt

Der Ansatz, dem Konstrukteur ein solches Werkzeug der Informationsvisualisierung an die Seite zu stellen, muss folgende Kernpunkte umfassen:

- Bereitstellen der Informationen aus den unterschiedlichen Arten von Produktentwicklungswerkzeugen entlang des Produktlebenszyklus unabhängig vom letztlich eingesetzten System

- Konzept zur Abbildung der Beziehungen zwischen Informationen aus vorhandenen Produktwicklungswerkzeugen entlang des Produktlebenslaufs mit vorhandener Produktgestalt

- Immersive, problembezogene Visualisierung der komplexen Zusammenhänge in einer verteilten heterogenen Systemlandschaft

Demzufolge betrachtet die vorliegende Arbeit, ausgehend vom aktuellen Stand der Technik und unter Einbeziehung der industriellen Praxis der Informationsvisualisierung und Produktentwicklung, vier Schwerpunkte, die eine Informationsvisualisierung entlang des Produktentwicklungsprozesses möglich machen:

- Allumfassende monolithische integrierte Produktmodelle entlang des Produktentstehungsprozess haben sich in der Praxis aufgrund mangelnder Verfügbarkeit und immenser Komplexität bei der Implementierung bisher nicht durchgesetzt. Vielmehr wird zur Umsetzung einer produktlebenslauforientierten Visualisierung von Informationen eine Verknüpfung bestehender Lösungskomponenten (Anwendungen des Produktentstehungsprozesses, z.B. CAD-, CAE-, CAM-, VR-, PDM-Systeme) benötigt [83] [84]. Das hier vorgestellte **modulare Framework** aus Methodenschnittstellen erlaubt einen Zugriff auf die Informationsquellen der Produktdefinitions-Softwaresysteme (wie z. B. CAD-Systeme) systemneutral. Methodenbausteine, die auf dieser Schicht aufbauen, kapseln, wiederum systemneutral, Anwendungsfunktionalität auf Basis dieser Datenmodellierungskomponenten. In der Endausbausstufe werden durch diese zwei Schichten alle Informationen in der Produktenstehung erreichbar. Die Beziehungen zwischen den Informationen, insbesondere von nichtgeometrischen Meta-Informationen werden durch ein XML-Beziehungsschema, den Engineering Objects, abgebildet. Wichtig ist hierbei, dass nicht die Informationen aus den Quellsystemen herangezogen werden, um sie in dem XML-Schema der Engineering Objects zu hinterlegen, sondern das lediglich die Beziehungen *zwischen* den Informationen aus den Quellsystemen abgebildet werden. Somit müssen keine redundanten Datenmodelle gepflegt werden. Dieser serviceorientierte Ansatz reduziert in erheblichem Maße den

Aufwand, den Systementwickler im CAx-Bereich treiben müssen, um Anwendungsfunktionalität, z. B. auf Basis von CAD-Systemen, realisieren zu können.

- Die entwickelte **Methodenschnittstelle für Visualisierungssysteme**, stellt im Verbund dieses Frameworks Anwendungsfunktionalität zur Informationsvisualisierung bereit, die von anderen Systemen einfach genutzt werden kann. Analog dem Zertifizierungs- und Standardisierungsvorgehen der OMG CAD-Services sollen die *Visualization-Services* ebenfalls bei der OMG standardisiert werden. Die Visualization-Services implementieren eine Methodenschnittstelle für Szenen-Graph-basierte Systeme, die einen hohen Abstraktionsgrad von grafischen Subsystemen aufweisen und somit universell einsetzbar sind. Zur Konzeptverifikation wurde ein plattformunabhängiger Softwareprototyp implementiert, der CAD-Bauteilgeometrie mit Hilfe der CAD-Services liest und zusammen mit Anforderungen immersiv (stereoskopisch) auf einer Projektionswand visualisiert. Durch Hinzunahme weiterer Client-Module kann die Projektion auf $2n$ Wände erweitert werden, so dass beliebige Anordnungen von Projektionswänden bedienbar werden.

- Exemplarisch für ein Produktentwicklungssystem aus den frühen Konstruktionsphasen wurde ein System zur Anforderungsmodellierung gewählt. Für diese Systeme gibt es bislang noch keine Unterstützung bei der integrierten Informationsvisualisierung. Dieses System zur Modellierung von Anforderungen integriert sich ebenfalls mittels einer Methodenschnittstelle in das Framework, wobei die Produktstruktur, angereichert mit den Produktanforderungen, über die Methodenschnittstelle ausgeleitet wird. Die Umsetzung der Anforderungen in XML-Form aus dem Anforderungsmodellierer (SMC) findet innerhalb eines **Script-Prozessors** statt, der durch eine Abbildungsvorschrift die Zuordnung von Klassen von Metainformationen zu den entsprechenden graphischen Metaphern vornimmt. Dieser Script Prozessor analysiert hierbei die B-Rep-Struktur der Bauteilgeometrie und erzeugt zusammen mit einer Bibliothek aus grafischen Metaphern einen Szene-Graphen der die Bauteilgeometrie integriert mit den Metainformationen darstellt. Als Abbildungsvorschrift wurde hier eine einfache, aber leistungsfähige **Script-Sprache** definiert, die die Zuordnung von Informationsklassen zu den grafischen Repräsentationen festlegt.

- Zur Visualisierung von Anforderungen wurde eine **Bibliothek aus grafischen Metaphern** entworfen, die die Semantik der Informationen in grafische Repräsentationen umsetzt. Die Bibliothek betrachtet die wesentlichen Phasen der Produktentwicklung: Konstruktion, Fertigung, Montage, Prüfung und Vertrieb.

Die Konzepte dieser Arbeit wurden durch eine prototypische Implementierung eines Visualisierungssystems verifiziert. Die Serverkomponente des Visualisierungssystems implementiert die in dieser Arbeit konzeptionell entworfene Methodenschnittstelle und stellt diese im beschriebenen Framework zur Verfügung. Die Clientkomponente zur Steuerung der Serverkomponente realisiert ein System zur integrierten Visualisierung von Produktanforderungen und Featureinformationen zusammen mit der bezogenen Produktgeometrie. Die Bauteilgeometrie wird dabei CAD-systemneutral über die CAD-Services-Schnittstelle ausgelesen und mittels der Visualisierungsfunktionalität der Visualization-Services-Schnittelle der Server-Komponente dargestellt. Zur Visualisierung der Anforderungen bzw. Features werden grafische Metaphern aus einer parametrischen Bibliothek aus VRML-Objekten instanziiert und in die Szene zusammen mit einem Abhängigkeitsgraph eingebracht. Der Benutzer kann nun interaktiv in der Szene sowohl Bauteilgeometrie als auch die Abhängigkeiten der zu visualisierenden Metainformationen (Anforderungen, Features, …) explorieren und die Zusammenhänge betrachten. Quantitativ vorhandene Informationen werden auf Wunsch textuell verfügbar gemacht.

In der vorliegenden Arbeit wird eine Systematik beschrieben, in der Informationen aus frühen Konstruktionsphasen zusammen mit existierender Bauteilgeometrie immersiv visualisiert werden. Exemplarisch wurden Produktanforderungen und Gestaltfeatures ausgewählt, der Ansatz ist jedoch allgemeingültig für alle Informationen aus Produktentwicklungswerkzeugen entlang des Produktlebenslaufs. Benefiz für den Konstrukteur ist ein einfach zu bedienendes Werkzeug, welches ihm ermöglicht, die immer komplexer werdenden Zusammenhänge in der Konzeptphase zu begreifen und Fehler durch Nichtbeachten von Abhängigkeiten zu vermeiden.

6.2 Ausblick

In Abbildung 55 wurde eine Middleware-basierte Infrastruktur vorgestellt, die Applikationsentwicklung im CAx-Umfeld für kleine und mittelständische Zulieferfirmen und Softwareanbieter wieder erreichbar macht. Einige der Modulkomponenten, wie die CAD-Services oder die PDM-Enabler, existierten bereits, und die Lücke der Visualisierungssysteme wurde durch diese Arbeit geschlossen. Es existiert jedoch noch eine ganze Reihe von „weißen Flecken“ in diesem komponentenbasierten Framework. Weder sind zum jetzigen Zeitpunkt der Zugriff auf CAM-Module noch auf Design-Algorithmen möglich. Gerade aber das „Kapseln“ und „zur-Verfügung-stellen“ von Wissen, via standardisierten Methodenschnittstellen, rund um die NC-gesteuerte Fertigung wäre ein wichtiger Schritt für durchgängige Prozessketten.

Mit den serviceorientierten Modulen wird ein Grundstein für eine neue Art von PDM-Systemen geschaffen, die nicht nur Datenmanagement und Prozesse auf Dateiebene betreiben, sondern

welche Daten, die in den Systemen liegen, miteinander verknüpfen können. Abhängigkeiten zwischen Engineering-Systemen aus frühen Konstruktionsphasen und CAD-Systemen werden so sichtbar. Das Framework hält den zu betreibenden Implementierungsaufwand in vertretbaren Größenordnungen.

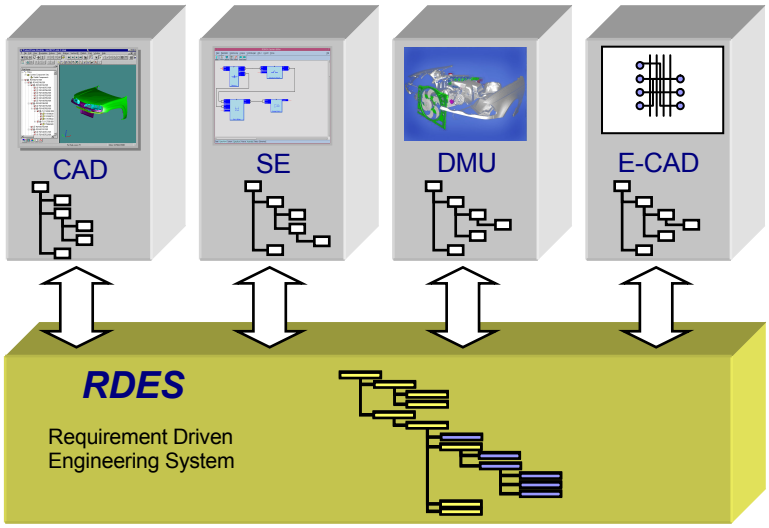


Abbildung 108: xPLM - eXtendend Product Lifecycle Management

Der serviceorientierte Ansatz kann ebenso in Zukunft anstatt nur zur reinen Applikations-entwicklung auch für Dienstleistungen im CAD-Sektor benutzt werden. Es ist heute noch gängige Praxis, dass Datenaustausch mit OEM's entweder im nativen Dateiformat der OEM's (hohe Anschaffungskosten bei niedriger Auslastung der CAD-Systeme beim Zulieferer) oder via Neutralfileformate (hoher Datenverlust) stattfindet. Mit der serviceorientierten Struktur könnten Dienstleistungszentren etabliert werden, die CAD-Daten in hoher Qualität online den Zulieferern für ihre NC-Programme zur Fertigung zur Verfügung zu stellen, ohne dass die CAD-Systeme vom Zulieferer gekauft werden müssen. Abbildung 109 zeigt auf, wie die Kollaboration von Zuliefer-Firmen mit OEM's auf Basis der CAD-Services nach dem Application-Sharing-Modell[62] in Zukunft von statten gehen könnte. Bisher wurden native CAD-Daten direkt an den Zulieferer gegeben, damit dieser sie weiterverarbeiten (NC-Daten) und fertigen kann. Dies setzt voraus, dass der Zulieferer das gleiche CAD-System vorweisen kann wie der OEM. Beim Datenaustausch auf Basis der CAD-Services würde der OEM die nativen CAD-Daten an ein zentrales Dienstleistungszentrum geben welches alle CAD-Systeme anbietet. Über die CAD-Services Schnittstelle werden dann die erforderlichen Daten den entsprechenden Zulieferern zugänglich gemacht ohne, dass er das spezifische CAD-System besitzen muss. Denkbar wäre auch ein völliger Verzicht auf das Dienstleistungszentrum und die direkte Installation der CAD-Services beim OEM.

[62] Application-Sharing ermöglicht zwei oder mehreren Benutzern die synchrone Benutzung einer beliebigen Anwendung

Sicherheitsüberlegungen mancher OEM's mögen aber dieser Möglichkeit entgegenstehen. Als Intranet-Ansatz innerhalb einer Firma bei geografisch verteilten Standorten bietet sich eine solche Lösung, bei kombinierter Nutzung von VPN[63]-Tunnel Verbindungen, jedoch an.

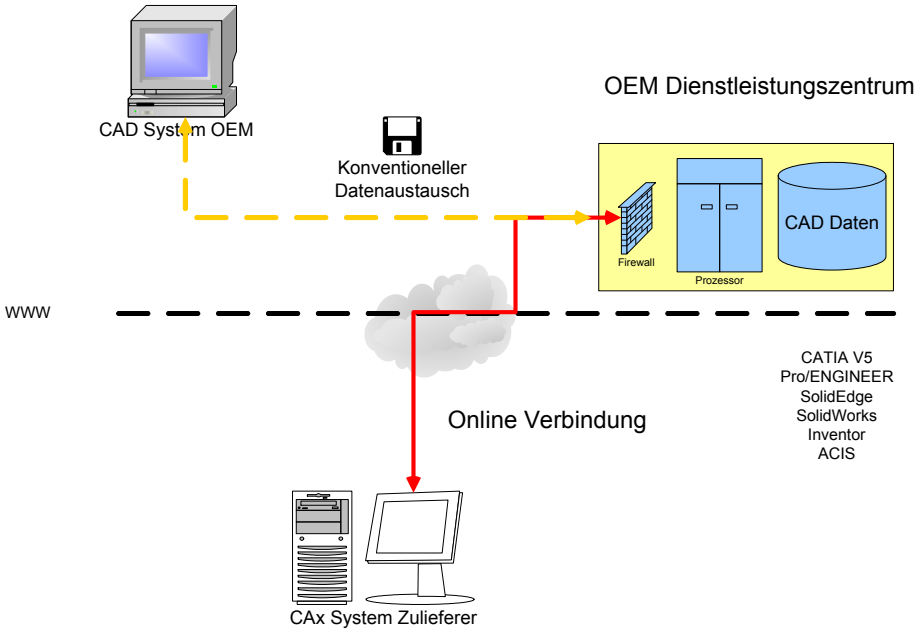


Abbildung 109: OEM Zulieferer Zusammenarbeit mit Dienstleistungszentren

[63] Ein VPN (Virtual Private Network, "Virtuelles Privates Netzwerk") verbindet zwei Netzwerke, einen Computer mit einem Netzwerk oder zwei Computer über öffentliche Verbindungen wie zum Beispiel das Internet.. Damit die Datenübertragung nicht von außen eingesehen werden kann gibt es ein so genanntes Tunneling-Protokoll, das die Daten, die ausgetauscht werden, ver- bzw. entschlüsselt.

LITERATURVERZEICHNIS

[1] H. Heilmann, H.-J. Etzel, und R. Richter, *IT-Projektmanagement - Fallstricke und Erfolgsfaktoren*, 2., überarbeitete und erweiterte Auflage 2003 ed. Heidelberg: dpunkt.verlag, 2003.

[2] J.-F. Grätz, *Handbuch der 3D-CAD Technik - Modellierung mit 3D Volumensystemen*: Siemens AG, 1989.

[3] J. Corney, *3D Modelling with the ACIS Kernel and Toolkit*: Wiley, 1997.

[4] J. Corney und T. Lim, *3D Modelling with ACIS*: Saxe-Coburg Publications, 2001.

[5] VDI, "VDI Richtlinie 2209 - 3D Produktmodellierung."

[6] VDI, *VDI Richtlinie 2218 - Feature Technologie*: Verein Deutscher Ingenieure (VDI).

[7] C. Weber, "What do we Call a "Feature" and What is its Use? - Results of FEMEX Working Group I "Feature Definition and Classification"," aus Tagungsband International Symposium on the Tools and Methods for Concurrent Engineering 1996 (TMCE 96)„ Budapest / Ungarn, 1996.

[8] G. Booch, J. Rumbaugh, und I. Jacobson, *Das UML-Benutzerhandbuch*: Addison-Wesley, 1999.

[9] S. Haasis, *Integrierte CAD-Anwendungen - Rationalisierungspotentiale und zukünftige Einsatzgebiete*. Berlin: Springer, 1995.

[10] U. Sendler, *3D-CAD, Die Produktivität der neuen Systemgeneration*. Berlin Heidelberg: Springer-Verlag, 1994.

[11] O. Klaar, "Computerunterstützte Merkmalsgewinnung und Verknüpfung in integrierten Produktmodellen," in *Institut für Rechneranwendung in Planung und Konstruktion (RPK)*: Universität Karlsruhe (TH), 2004.

[12] G. P. W. Beitz, *Konstruktionslehre (3. neubearb. u. erw. Auflage)*. Berlin: Springer-Verlag, 1994.

[13] G. P. W. Beitz, *Konstruktionslehre (4. Auflage)*. Berlin: Springer-Verlag, 1997.

[14] W. Beitz, *Durchgängige, flexible Rechnerunterstützung für den Konstruktionsprozess*. Düsseldorf: VDI Verlag, 1986.

[15] V. H. W. E. Eder, *Einführung in die Konstruktionswissenschaft*. Berlin: Springer - Verlag, 1992.

[16] K. Ehrlenspiel, *Integrierte Produktenentwicklung*. München: Carl Hanser Verlag, 1995.

[17] W. B. F. Hansen, *Rationelles Konstruieren*. Berlin: Technik, 1953.

[18] V. Hubka, *Theorie der Konstruktionsprozesse*. Berlin: Springer Verlag, 1976.

[19] R. Koller, *Konstruktionslehre für den Maschinenbau*, 4. Neubearb. und erw. Auflage ed. Berlin: Springer - Verlag, 1998.

[20] G. Pahl, *Konstruieren mit 3D-CAD Systemen*. Berlin: Springer-Verlag, 1990.

[21] K. Roth, *Konstruieren mit Konstruktionskatalogen. Bd. 2 Kataloge (2. erw. u. neu gest. Auflage)*. Berlin: Springer Verlag, 1994.

[22] K. Roth, *Konstruieren mit Konstruktionskatalogen Bd. 1 Konstruktionslehre (3. erw. u. neu gest. Auflage)*. Berlin: Springer Verlag, 2000.

[23] R. Sietmann, "Erfinden nach Plan - Die TRIZ-Methodik hilft dem Aha-Effekt auf die Sprünge," *c't - Magazin für Computertechnik*, vol. 23, 2001.

[24] G. Spur und F.-L. Krause, *Das virtuelle Produkt: Management der CAD-Technik*. München, Wien: Carl-Hanser, 1997.

[25] H. Grabowski, S. Rude, und G. Grein, *Universal Design Theory*. Aachen: Shaker Verlag, 1998.

[26] H. Grabowski, "Rechneruntestütztes Konstruieren und Erstellen von Fertigungsunterlagen (CAD 1+2)," 2003.

[27] M. Kress, A. Weber, M. F. Zäh, H. J. Helml, U. Krönert, und G. Reinhardt, *Integriertes Produktmodell - Von der Idee*

zum fertigen Produkt: Herbert Utz Verlag, 1997.
[28] H. Grabowski, R. Anderl, und A. Polly, *Integriertes Produktmodell*. Berlin: Beuth, 1993.
[29] O. Abeln, *CAD-Referenzmodell: Zur arbeitsgerechten Gestaltung zukünftiger computergestützte Konstruktionsarbeit*. Stuttgart: B. G. Teubner, 1995.
[30] A. Alzoobi, *Objektorientierte Modellierung der Kosten in Hochbauprojekten*. Brandenbourg, 2003.
[31] S. Rude, *Wissensbasierte Konstruktionssysteme*. Aachen: Shaker Verlag, 1998.
[32] D. Renner, *Informationsräume zur Visualisierung struktureller Zusammenhänge von Produktmodellen*: Shaker Verlag, 2002.
[33] ProSTEP, http://www.prostep.de/de/, 2005
[34] P. R. Keller und M. M. Keller, *Visual Cues, Practical Data Visualization*. Los Alamitos, California: IEEE Computer Society Press, 1993.
[35] VDI, *VDI Norm 66234 - Entwurf Teil 1 "Kennwerte für die Anpassung von Bildschirmarbeitsplätzen an den Menschen; Geometrische Gestaltung der Schriftzeichen"*: Verein Deutscher Ingenieure (VDI).
[36] VDI, *VDI Norm 66234 - Entwurf Teil 2 "Bildschirmarbeitsplätze; Wahrnehmbarkeit von Zeichen auf Bildschirmen"*: Verein Deutscher Ingenieure (VDI).
[37] VDI, *VDI Norm 66234 - Entwurf Teil 3 "Bildschirmarbeitsplätze; Gruppierung und Formatierung von Daten"*: Verein Deutscher Ingenieure (VDI).
[38] VDI, *VDI Norm 66234 - Entwurf Teil 5 "Bildschirmarbeitsplätze; Codierung von Information"*: Verein Deutscher Ingenieure (VDI).
[39] P. Young, "Three Dimensional Information Visualisation," Department of Computer Science University of Durham 1996.
[40] P. K. Robertson und L. D. Ferrari, "Systematic Approaches to Visualization: Is a Reference Model needed ?," aus Tagungsband ONR Workshop on Data Visualization, Darmstadt, 1993.
[41] H. Englberger, "Computergestützte Informationsvisualisierung," in *Fakultät für Informatik*. München: Technische Universität München, 1995.
[42] M. Chalmers und P. Chitson, "BEAD: explorations in information visualization," aus Tagungsband 15th annual international ACM SIGIR conference on Research and development in information retrieval, Copenhagen, Denmark, 1992.
[43] J. Fost, "*Toward the Glass Bead Game - a rhetorical invention*", http://mitpress2.mit.edu/e-journals/LEA/GALLERY/glassbeadgame/,
[44] G. Ericksson, "*Stereo Rendering Using OpenGL and GLUT*", http://www.acmc.uq.edu.au/~gbe/stereodisplay, 2004
[45] Stereographics, http://www.stereographics.com/,
[46] Infitec, http://www.infitec.net/,
[47] Pyramid Systems, "*ImmersaDesk*", http://www.avl.iu.edu/technology/idesk/,
[48] Electronic Visualization Laboratory of Illinois Chicago, http://www.ncsa.uiuc.edu,
[49] V. Raja und N. Matthews, "*Cybersphere Brings Star Trek's Holodeck Closer to Reality*", http://www.warwick.ac.uk/news/pr/230,
[50] Virtual Realities, http://www.vrealities.com/cyber.html,
[51] 3Dconnexion, http://www.3dconnexion.com/products/3a2.php,
[52] J. D. Bernd Fröhlich, Hans-Jörg Bullinger, "*Cubic Mouse*", http://www.imk.fraunhofer.de/sixcms/media.php/130/cubmouse.pdf,
[53] S. Technologies, http://www.sensable.com/products/phantom_ghost/phantom.asp,
[54] T. Akenine-Möller und E. Haines, *Real-Time Rendering*, second edition ed: A K Peters Natik, 2002.

[55] J. Wernecke, *The Inventor Toolmaker: Extending Open Inventor*: Addisson-Wesley, 1994.
[56] Sun, "*Java3D*", http://java.sun.com/products/java-media/3D/,
[57] J. Isdale, "*What is Virtual Reality ?*" http://www.cms.dmu.ac.uk/~cph/VR/whatisvr.html,
[58] B. Fröhlich, J. Deisinger, und H.-J. Bullinger, "Immersive Projection Technology and Virtual Environments 2001," in *Proceedings of the Eurographics Workshop in Stuttgart, Germany, May 16-18, 2001*: Springer, 2001, pp. 284.
[59] A. Aregger und A. Przygienda, http://www.ifi.unizh.ch/study/Vorlesungen/Visualisierung/Aregger/methoden_d.html,
[60] K. Kücherer, "Angereicherte Wirklichkeit," *c't - Magazin für Computertechnik*, vol. 16, pp. 80-83, 2003.
[61] F.-L. Krause, T. Tang, und U. Ahle, *Leitprojekt integrierte virtuelle Produktenstehung (Abschlussbericht)*. Stuttgart: Fraunhofer IRB Verlag, 2002.
[62] I. Meinard und M. Gubsch, "*Leitprojekt integrierte virtuelle Produktenstehung - Fortschrittsbericht April 2001: Integration von CAD-Systemen*," F. L. Krause, T. Tang, und U. Ahle, (Hrsg.). Stuttgart: Fraunhofer IRB Verlag, 2001.
[63] E. Gamma, Richard Helm, Ralph Johnson, und J. Vlissides, *Entwurfsmuster (Elemente wiederverwendbarer objektorientierter Software)*: Addison-Wesley, 1996.
[64] H. Grabowski, "Abschlussbericht Transferbereich 16 - Rechnerintegrierte Konstruktion und Fertigung von Bauteilen," Universität Fridericiana Karlsruhe (TH), Karlsruhe.
[65] P. Loucopoulos und J. Mylopoulos, "Requirements Engineering," vol. 5, 2000.
[66] M. Gebauer, "Kooperative Produktentwicklung auf der Basis verteilter Anforderungen," in *Institut für Rechneranwendung in Planung und Konstruktion*. Karlsruhe: Universität Karlsruhe, 2001, pp. 161.
[67] O. Klaar, M. Jörg, und R.-S. Lossack, "Requirement Driven Engineering in Collaborative Environments," aus Tagungsband eChallenges 2003, Prag, 2003.
[68] G. Ahrens, *Das Erfassen und Handhaben von Produktanforderungen*. TU Berlin: Disertation, 2000.
[69] R. Anderson, M. Bibeck, und M. Kay, *XML Professionell*, 2000.
[70] H. Wittenbrink und Bergmann, *XML*. Berlin: SPC Lehrbuch, 2005.
[71] E. R. Harold und W. S. Means, *XML in a nutshell*: O'Reilly, 2005.
[72] T. Apache-Software-Foundation, "*The Apache XML-Project: XERCES-C++*", http://xml.apache.org/xerces-c/,
[73] M. Richter, "Gegen die Abschottung," *Digital Engineering Magazin (WIN Verlag)*, pp. 34-35, 2004.
[74] A.-W. Scheer, *Architektur integrierter Informationssysteme*. Berlin: Springer, 1992.
[75] T. v. d. Elst, "UMEO XML Schema Dokumentation." Ulm, 2003.
[76] J. U. Zimmermann, S. Haasis, und F. J. A. M. v. Houten, "Applying Universal Linking of Engineering Objects in the Automotive Industry - Practical Aspects, Benefits, and Prototypes," aus Tagungsband Design Engineering Technical Conferences / Design Automation Conferences (DETC2002/DAC), Montreal, Canada, 2002.
[77] J. U. Zimmermann, S. Haasis, und F. J. A. M. v. Houten, "ULEO - Universal Linking of Engineering Objects."
[78] J. U. Zimmermann, *Informational Integration of Product Development Software in the Automotive Industry - The ULEO Approach*. Twente (NL), 2005.
[79] M. Richter, "New working environments for product development - A componet framework for CAx application development," aus Tagungsband eChallenges, Wien, 2004.
[80] M. Richter, "3D Workbench: CAx-Systeme für Innovationen öffnen," in

CAD-CAM Report, vol. 9, 2004, pp. 70-73.

[81] Cosimir, "*COSIMIR® VR-System*", http://www.cosimir.com/VR/German/Medizin/medizin.htm, 2005

[82] R. Isernhagen, *Softwaretechnik in C und C++*, 2. Auflage ed: Hanser, 2000.

[83] U. Sendler, "*Sendler Circle*", http://www.sendlercircle.com/, 2005

[84] U. Sendler, *CAD und PDM: Prozessoptimierung durch Integration*: Hanser Verlag München, 2005.

INDEX

Anhang A

INTERFACE DEFINITIONEN VIZSERVICES

File: VizConnection.idl

```
/* VizConnection.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZCONNECTION_DEFINED
#define __VIZCONNECTION_DEFINED

#include "VizMain.idl"

module VizConnection {

    //forward references
    interface VizServer;
    interface VizSystem;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    exception VizConnectionFault {
        unsigned long error_code;
        string error_text;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    exception VizSystemFault {
        unsigned long error_code;
        string error_text;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface VizServer {
        /**
         * @idl-raises VizConnection::VizConnectionFault
         */

        VizSystem connect() raises(VizConnectionFault);
    };


/**
 * @author Dipl.-Ing. Martin Richter
```

```
 * @version 1.0
 */

    interface VizSystem {
        VizMain::Universe create_universe();

            /**
             * close system and clean-up
             * @idl-raises VizConnection::VizConnectionFault
             */

        void disconnect() raises(VizConnectionFault);
    };
};

#endif

// eof VizConnection.idl
```

File: VizDevice.idl

```
/* VizDevice.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZDEVICE_DEFINED
#define __VIZDEVICE_DEFINED

module VizDevice {
/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    exception DeviceException {
        unsigned long error_code;
        string error_text;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    enum HardwarePort {
        SERIAL_PORT, USB_PORT, PARALLEL_PORT, CONTROLLER_CARD
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Device {


        void open(in VizDevice::HardwarePort connected_to)
           raises(DeviceException);
```

```
        void close() raises(DeviceException);

        any get_rawdata() raises(DeviceException);

        /**
                 * NUmber of frames that the system displays.
                 */

        attribute long frame_rate;

        /**
         * Range of values a sensor may return when in a given state.
         */

        attribute long accuracy;
        attribute long lag;
    };


    typedef sequence<Device> DeviceList;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Mouse : Device {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Sensor : Device {
            /**
             * Return the sensors sensitivity value.
             */

        float get_sensitivity() raises(DeviceException);

            /**
             * Sets the sensitivity value for the sensor.
             */

        void set_sensitivity(in float sensitivity) raises(DeviceException);

        void set_angularrate(in float rate) raises(DeviceException);

        float get_angularrate() raises(DeviceException);

        float get_translation() raises(DeviceException);

        float get_rotation() raises(DeviceException);

        attribute double translation;
        attribute double angular_rate;
        attribute double sensitivity;
    };


    typedef sequence<Sensor> SensorList;
```

```
/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Joystick : Sensor {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 * @since 2003, 2004
 */

    interface CyberGlove : Sensor {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface TrackingSystem : Sensor {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface SpaceBall : Sensor {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface ShutterGlasses : Sensor {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface HMD : Sensor {
    };
};

#endif

// eof VizDevice.idl
```

File: VizGeometry.idl

```
/* VizGeometry.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZGEOMETRY_DEFINED
#define __VIZGEOMETRY_DEFINED

module VizGeometry {

/**
 * three dimensional location.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    struct PointStruct {
        double x;
        double y;
        double z;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    struct BoundingBox {
        PointStruct point_min;
        PointStruct point_max;
    };


    typedef sequence<PointStruct> PointStructSeq;
    typedef sequence<PointStructSeq> PointStructSeqSeq;

/**
 * Direction in 3D
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    struct VectorStruct {
        double i;
        double j;
        double k;
    };


    typedef sequence<VectorStruct> VectorStructSeq;
    typedef sequence<VectorStructSeq> VectorStructSeqSeq;
    typedef sequence<VectorStructSeqSeq> VectorStructSeqSeqSeq;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    struct TransformationStruct {
        PointStruct offset;
        VectorStruct i_ref_dir;
        VectorStruct k_dir;
```

```
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    struct RayStruct {
        PointStruct origin;
        VectorStruct direction;
    };


    typedef sequence<RayStruct> RayStructSeq;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    struct UVStruct {
        double u;
        double v;
    };


    typedef sequence<UvStruct> UvStructSeq;

    /**
     * basic range information
     * @author Dipl.-Ing. Martin Richter
     * @version 1.0*/

    struct RangeStruct {
        double high;
        double low;
    };

};

#endif

// eof VizGeometry.idl
```

File: VizMaterial.idl

```
/* VizMaterial.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZMATERIAL_DEFINED
#define __VIZMATERIAL_DEFINED
#include "VizGeometry.idl"

module VizMaterial {
/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */
```

```
    interface Texture {


        void set_texture(in boolean shaded, in boolean transparent,
           in string bitmap);

        void set_uv(in VizGeometry::UVStruct uvarray);

        void load_texture(in string filename);

        void set_cached(in boolean flag);

        boolean is_cached();

        attribute boolean shaded;
        attribute string bitmap;
        attribute boolean cached;
        attribute boolean transparent;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Material {

        attribute double ambient;
        attribute double ambientDiffuse;
        attribute double diffuse;
        attribute double emission;
        attribute double opacity;
        attribute double shininess;
        attribute double specular;
    };


    typedef sequence<Material> MaterialList;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface SimpleMaterial : Material {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface TexturedMaterial : Material {


        void set_texture(in VizMaterial::Texture aTexture);

        VizMaterial::Texture get_texture();

        attribute VizMaterial::Texture active_texture;
```

```
    };

};

#endif

// eof VizMaterial.idl
```

File: VizMail.idl

```
/* VizMain.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZMAIN_DEFINED
#define __VIZMAIN_DEFINED

#include "VizDevice.idl"
#include "VizMaterial.idl"
#include "VizGeometry.idl"

module VizMain {

    //forward references
    interface Universe;
    interface VisualSceneGraph;
    interface HapticSceneGraph;
    interface VisualSceneManager;
    interface HapticSceneManager;
    interface Node;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    exception SceneGraphException {
        unsigned long error_code;
        string error_text;
    };

    typedef sequence<Node> NodeList;

/**
 * Basic Node. Superclass to all types of nodes.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface Node {
        void add_child(in VizMain::Node child) raises(SceneGraphException);

        void remove_child(in VizMain::Node child)
raises(SceneGraphException);

        void set_name(in string aName);

        string get_name();

        boolean is_enabled();
```

```
        boolean is_moveable();

        void insert_child(in unsigned long position, in boolean child)
           raises(SceneGraphException);

        unsigned long get_numchildren();

        VizMain::Node get_parent() raises(SceneGraphException);

        attribute boolean movable;
        attribute VizMain::NodeList child_node;
        attribute VizMain::Node parent_node;
        attribute boolean enabled;
        attribute string name;
    };



/**
 * Orientation and position of the scene viewer.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface ViewPoint {
        void rotate();

        void move();

        attribute long orientation;
        attribute VizGeometry::PointStruct position;
        attribute VizGeometry::VectorStruct direction;

        /**
         * Vertical scale factor applied to sceens.
         */

        attribute float aspect_ratio;

        /**
         * Distance between right and left eye.
         */

        attribute float parallax;

        /**
         * Horizontal offset in pixels which is applied to both eyes. It is
substracted from the left eye and added to the right eye.
         */

        attribute long convergence;
        attribute float convergence_distance;
        attribute boolean stereo;
        attribute string name;
    };


    typedef sequence<ViewPoint> ViewPointList;

    enum ProjectionTypes {
        SYMMETRIC, ASYMMETRIC, GENERAL, ORGHOGRAPHIC
    };

/**
```

```
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 * @since 2003, 2004
 */

    interface Window {


        VizMain::Universe get_universe() raises(SceneGraphException);

        attribute boolean enabled;
        attribute VizMain::ViewPoint active_viewpoint;
        attribute VizMain::ProjectionTypes projection_type;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Universe {
        VizMain::VisualSceneGraph get_visual_scenegraph();

        void set_visual_scenegraph(in VizMain::VisualSceneGraph scenegraph);

        VizMain::HapticSceneGraph get_haptic_scenegraph();

        void set_haptic_scenegraph(in VizMain::HapticSceneGraph scenegraph);

        void add_viewpoint(in VizMain::ViewPoint aViewpoint)
           raises(SceneGraphException);

        void remove_viewpoint(in VizMain::ViewPoint aViewPoint)
           raises(SceneGraphException);

        VizMain::ViewPoint get_viewpoint() raises(SceneGraphException);

        VizMain::ViewPoint next_viewpoint() raises(SceneGraphException);

        attribute VizMain::VisualSceneManager visual_scene_manager;
        attribute VizMain::HapticSceneManager haptic_scene_manager;
        attribute VizMain::HapticSceneGraph haptic_scene_graph;
        attribute VizMain::VisualSceneGraph visual_scene_graph;
        attribute VizMain::ViewPointList viewpoints;
        attribute VizMain::Window active_window;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface VisualSceneGraph {
        void set_rootnode(in VizMain::Node rootnode);

        VizMain::Node load_file(in string aFileName)
           raises(SceneGraphException);

        /**
         * This function finds the first occurence of the node identified by
a name.
```

```
         */

        VizMain::Node find_node_by_name(in string aName)
           raises(SceneGraphException);

        /**
         * Deletes all non root nodes from the Visual Scene Graph.
         */

        void vacuum() raises(SceneGraphException);

        VizMain::Node get_rootnode() raises(SceneGraphException);

        void load_material_table(in string filename)
           raises(SceneGraphException);

        void save_material_table(in string filename);

        attribute VizMaterial::MaterialList material_table;
        attribute VizMain::Node root_node;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface HapticSceneGraph {

        attribute VizMain::Node root_node;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface VisualSceneManager {
        void add_device();

        void remove_device();

        void get_device();

        void next_device();

        attribute VizDevice::DeviceList device_list;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface HapticSceneManager {

        attribute VizDevice::DeviceList device_list;
    };
};
```

```
#endif

// eof VizMain.idl
```

File: VizNode.idl

```
/* VizNode.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZNODE_DEFINED
#define __VIZNODE_DEFINED

#include "VizDevice.idl"
#include "VizMain.idl"
#include "VizGeometry.idl"

module VizNode {

    // forward references
    interface VisualSceneGraphNode;

/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    exception SceneGraphNodeException {
        unsigned long error_code;
        string error_text;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    enum NodeType {
        GROUP_NODE, LIGHT_NODE, TRANSFORM_NODE, FOG_NODE, GEOMETRY_NODE,
           SEPARATOR_NODE, LOD_NODE, ANCHOR_NODE, SWITCH_NODE
    };

    typedef sequence<VisualSceneGraphNode> VisualSceneGraphNodeList;

/**
 * Definition of a color.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    struct RGBcolor {
        double red;
        double green;
        double blue;
    };


/**
 * @author Dipl.-Ing. Martin Richter
```

```
 * @version 1.0
 */

    interface VisualSceneGraphNode : VizMain::Node {

        attribute VizNode::NodeType node_type;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface HapticSceneGraphNode : VizMain::Node {


        VizNode::NodeType get_type();

        attribute VizNode::NodeType node_type;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface GroupNode : VisualSceneGraphNode {
    };


/**
 * List of sensors that are attached to a transform node in oder to control
it.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface TransformNode : GroupNode {
        /**
         * Returns the transformation structure.
         */

        VizGeometry::TransformationStruct get_transformation()
           raises(SceneGraphNodeException);

        /**
         * Replaces the transformation struct of the node.
         */

        void set_transformation(in VizGeometry::TransformationStruct
           aTransformation) raises(SceneGraphNodeException);

        /**
         * Replaces the translation component.
         */

        void set_translation(in VizGeometry::PointStruct aTranslation)
           raises(SceneGraphNodeException);

        /**
         * Returns the translation component.
         */
```

```
        VizGeometry::PointStruct get_translation();

        /**
         * Adding an incremental translation.
         */

        void translate(in VizGeometry::PointStruct parameter0)
           raises(SceneGraphNodeException);

        /**
         * Sets the rotational component.
         */

        void set_rotation(in VizGeometry::VectorStruct i_dir,
           in                VizGeometry::VectorStruct                k_dir)
raises(SceneGraphNodeException);

        /**
         * Gets the rotational component.
         */

        VizGeometry::TransformationStruct get_rotation()
           raises(SceneGraphNodeException);

        /**
         * Attaches a sensor to a transform node.
         */

        void add_sensor(in VizDevice::Sensor aSensor)
           raises(SceneGraphNodeException);

        /**
         * Removes a sensor from a transform node.
         */

        void remove_sensor(in VizDevice::Sensor aSensor)
           raises(SceneGraphNodeException);

        attribute VizGeometry::TransformationStruct transformation;
        attribute VizDevice::SensorList sensor_list;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    enum RenderingStyle {
        RENDER_ANTIALIAS, RENDER_BEST, RENDER_GOURAUD, RENDER_LIGHTNING,
           RENDER_PERSPECTIVE, RENDER_SMOOTH, RENDER_TEXTURED
    };


/**
 * Geometry node.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface GeometryNode : VisualSceneGraphNode {
```

```
        void load(in string aFilename) raises(SceneGraphNodeException);

        /**
         * Obtains the extends of the geometry node.
         */

        VizGeometry::BoundingBox                                    get_extends()
raises(SceneGraphNodeException);

        /**
         * Obtains the midpoint of the bounding box of the geometry node.
         */

        VizGeometry::PointStruct get_midpoint();

        void strech(in VizGeometry::PointStruct factors,
           in                VizGeometry::PointStruct                center)
raises(SceneGraphNodeException);

        void scale(in float factor) raises(SceneGraphNodeException);

        void translate(in VizGeometry::VectorStruct offset)
           raises(SceneGraphNodeException);

        void transform(in VizGeometry::TransformationStruct transformation)
           raises(SceneGraphNodeException);

        void prebuild() raises(SceneGraphNodeException);

        void delete_prebuild() raises(SceneGraphNodeException);

        void set_materialid(in unsigned long id)
           raises(SceneGraphNodeException);

        unsigned long long get_materialid() raises(SceneGraphNodeException);

        void set_bothsides(in boolean flag) raises(SceneGraphNodeException);

        boolean get_bothsides() raises(SceneGraphNodeException);

        VizGeometry::PointStruct get_cg() raises(SceneGraphNodeException);

        void set_cg(in VizGeometry::PointStruct cg)
           raises(SceneGraphNodeException);

        /**
         * Id with describes the used material for the geometry node. Used
togehther with the material table of a scene graph.
         */

        attribute unsigned long material_id;
        attribute VizNode::RenderingStyle render_style;
        attribute VizGeometry::PointStruct center_of_gravity;
        attribute boolean both_sides;
    };


    typedef sequence<float> TriangleList;

/**
 * List of triangles. Coordinates are listed in one sequence because of
performance reasons. This is the main data structure current mesher engines
deliver.
```

```
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface TesselationNode : GeometryNode {
        void add_triangle(in VizGeometry::PointStruct aTriangle)
           raises(SceneGraphNodeException);

        unsigned long long get_numberOfTriangles()
           raises(SceneGraphNodeException);

        VizNode::TriangleList get_listOfTriangles()
           raises(SceneGraphNodeException);

        attribute unsigned long long num_triangles;
        attribute VizNode::TriangleList triangle_list;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface Text3D : GeometryNode {


        void set_text();

        void get_text();

        void set_spacing(in float spacing) raises(SceneGraphNodeException);

        float get_spacing() raises(SceneGraphNodeException);

        /**
         * Name of the used font.
         */

        attribute string font_name;
        attribute string text;
        attribute float spacing;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface PolygonNode : GeometryNode {


        VizGeometry::PointStruct get_vertices()
           raises(SceneGraphNodeException);

        VizGeometry::PointStruct                                    next_vertex()
raises(SceneGraphNodeException);

        attribute VizGeometry::PointStructSeq vertices;

        /**
         * Number of polygons in this node.
```

```
         */

        attribute unsigned long num_polygons;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface LightNode : VisualSceneGraphNode {


        void load(in string filename) raises(SceneGraphNodeException);

        void save(in string filename) raises(SceneGraphNodeException);

        attribute VizNode::RGBcolor ambient_color;
        attribute VizNode::RGBcolor diffuse_color;
        attribute VizNode::RGBcolor specular_color;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface DirectionalLight : LightNode {


        void           set_intensity(in           float           intensity)
raises(SceneGraphNodeException);

        float get_intensity() raises(SceneGraphNodeException);

        VizGeometry::VectorStruct get_direction()
           raises(SceneGraphNodeException);

        void set_direction(in VizGeometry::VectorStruct direction)
           raises(SceneGraphNodeException);

        attribute float intensity;
        attribute VizGeometry::VectorStruct direction;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface SpotLight : LightNode {


        void           set_intensity(in           float           intensity)
raises(SceneGraphNodeException);

        float get_intensity() raises(SceneGraphNodeException);

        void set_direction(in VizGeometry::VectorStruct direction)
           raises(SceneGraphNodeException);
```

```
        VizGeometry::VectorStruct get_direction()
           raises(SceneGraphNodeException);

        void set_position(in VizGeometry::PointStruct position)
           raises(SceneGraphNodeException);

        VizGeometry::PointStruct get_position()
           raises(SceneGraphNodeException);

        float get_exponent() raises(SceneGraphNodeException);

        void            set_exponent(in            float            exponent)
raises(SceneGraphNodeException);

        void set_angle(in float angle) raises(SceneGraphNodeException);

        float get_angle() raises(SceneGraphNodeException);

        void set_attenuation(in VizGeometry::VectorStruct attenuation)
           raises(SceneGraphNodeException);

        VizGeometry::VectorStruct get_attenuation()
           raises(SceneGraphNodeException);

        attribute float intensity;
        attribute VizGeometry::VectorStruct direction;
        attribute VizGeometry::PointStruct position;
        attribute float exponent;
        attribute float angle;
        attribute VizGeometry::VectorStruct attenuation;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface PointLight : LightNode {


        void set_intensity() raises(SceneGraphNodeException);

        void get_intensity() raises(SceneGraphNodeException);

        void set_position(in VizGeometry::PointStruct position)
           raises(SceneGraphNodeException);

        VizGeometry::PointStruct get_position()
           raises(SceneGraphNodeException);

        void set_attenuation(in VizGeometry::VectorStruct attenuation)
           raises(SceneGraphNodeException);

        VizGeometry::VectorStruct get_attenuation()
           raises(SceneGraphNodeException);

        attribute float intensity;
        attribute VizGeometry::PointStruct position;
        attribute VizGeometry::VectorStruct attenuation;
    };
```

```
/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    enum FogMode {
        LINEAR, EXPONENTIAL, EXPONENTIAL_SQUARE
    };

/**
 * Fog node.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface FogNode : VisualSceneGraphNode {
        /**
         * Set the color of the fog.
         */

        void set_color(in VizNode::RGBcolor aColor)
           raises(SceneGraphNodeException);

        /**
         * Get the color of the fog.
         */

        VizNode::RGBcolor get_color() raises(SceneGraphNodeException);

        /**
         * Set the range where all objects are completely blended into fog.
         */

        void set_range(in float range) raises(SceneGraphNodeException);

        float get_range() raises(SceneGraphNodeException);

        void set_mode(in VizNode::FogNode mode)
           raises(SceneGraphNodeException);

        VizNode::FogMode get_mode() raises(SceneGraphNodeException);

        float get_linearstart() raises(SceneGraphNodeException);

        void set_linearstart(in float linearstart)
           raises(SceneGraphNodeException);

        /**
         * Distance where all objects are completely blended into fog.
         */

        attribute float range;

        /**
         * Color of the fog.
         */

        attribute VizNode::RGBcolor fog_color;
        attribute VizNode::FogMode mode;

        /**
         * Distance where the objects are affected by the fog color. (Only
in linear mode.)
```

```
         */

        attribute float linearstart;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface SeparatorNode : VisualSceneGraphNode {
    };


/**
 * Group node that enables only one child at a time.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface SwitchNode : GroupNode {
    };


/**
 * Specialised switch node which switches out its childred triggered by a
certain treshold (switch out distance).
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0*/

    interface LODNode : SwitchNode {


        /**
         * Set the switch out distance for the children.
         */

        void set_range(in float range) raises(SceneGraphNodeException);

        /**
         * Implementation of a quick reject test. Subtree is not traversed
if bounding box lies out of the viewing area.
         */

        void set_cullmode(in boolean mode) raises(SceneGraphNodeException);

        attribute boolean cull_mode;
        attribute float range;

        /**
         * Center of the LOD node. This is used to calculate the distance to
the viewpoint.
         */

        attribute VizGeometry::PointStruct center;
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */
```

```
    interface AnchorNode : VisualSceneGraphNode {
    };


/**
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface VisualSceneGraphNodeFactory {


        /**
         * Create a new group node.
         */

        VizNode::GroupNode create_goupnode(in VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new anchor node.
         */

        VizNode::AnchorNode                                  create_anchornode(in
VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new level of detail node.
         */

        VizNode::LODNode    create_lodnode(in    VizNode::VisualSceneGraphNode
parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new separator node.
         */

        VizNode::SeparatorNode create_separatornode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new switch node.
         */

        VizNode::SwitchNode                                   create_switchode(in
VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new Transformnode.
         */

        VizNode::TransformNode create_transformnode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new lightnode.
         */
```

```
        VizNode::LightNode create_lightnode(in VizNode::VisualSceneGraphNode
           parent) raises(SceneGraphNodeException);

        /**
         * Create a new geometry node.
         */

        VizNode::GeometryNode create_geometrynode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new fognode.
         */

        VizNode::FogNode   create_fognode(in   VizNode::VisualSceneGraphNode
parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new directional light.
         */

        VizNode::DirectionalLight
           create_directionallightnode(in      VizNode::VisualSceneGraphNode
parent)
           raises(SceneGraphNodeException);

        /**
         * Create a new spot light.
         */

        VizNode::SpotLight create_spotlightnode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        VizNode::PointLight create_pointlightnode(in
           VizNode::VisualSceneGraphNode parent)
           raises(SceneGraphNodeException);

        /**
         * Creates a cylinder.
         */

        VizNode::GeometryNode  create_cylinder(in  float  height,  in  float
radius,
           in boolean bothsides, in unsigned long tesselation)
           raises(SceneGraphNodeException);

        VizNode::GeometryNode create_block(in float x, in float y, in float
z,
           in boolean bothsides) raises(SceneGraphNodeException);

        VizNode::GeometryNode create_cone(in float heigth, in float radius,
           in unsigned long tesselation, in boolean bothsides)
           raises(SceneGraphNodeException);

        VizNode::GeometryNode create_sphere(in float radius,
           in unsigned long tesselation, in boolean bothsides)
           raises(SceneGraphNodeException);

    };
```

```
 /**
  * basic material with properties for light calculation
  * @author Dipl.-Ing. Martin Richter
  */


};

#endif

// eof VizNode.idl
```

File: VizScripting.idl

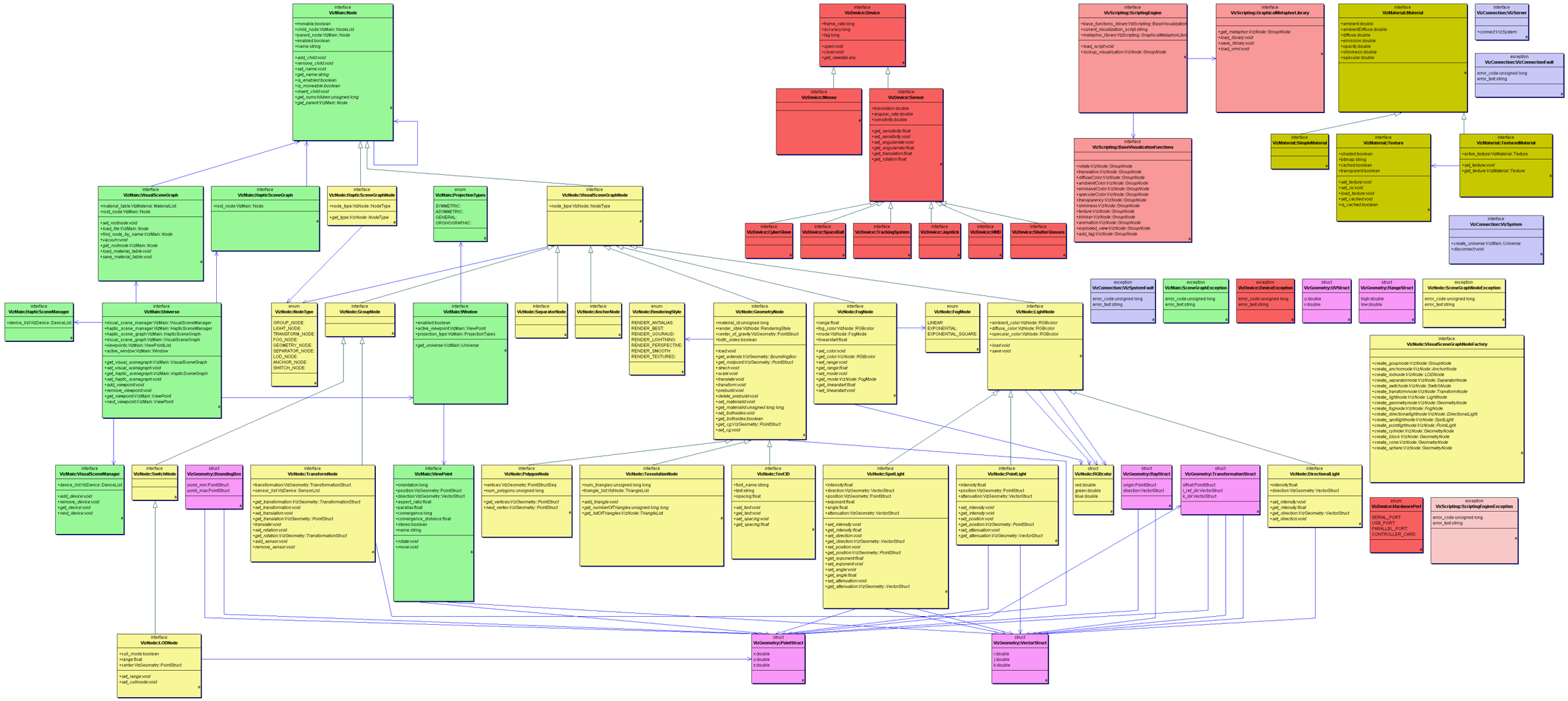
```
/* VizScripting.idl
 *
 * (c)2003, 2004 Dipl.-Ing. Martin Richter
 */

#ifndef __VIZSCRIPTING_DEFINED
#define __VIZSCRIPTING_DEFINED
#include "VizNode.idl"
#include "VizGeometry.idl"
#include "VizMaterial.idl"

module VizScripting {

    interface BaseVisualizationFunctions;
    interface GraphicalMetaphorLibrary;

    exception ScriptingEngineException {
        unsigned long error_code;
        string error_text;
    };

/**
 * The scripting engine is the main processor which identifies a geometrical
entity by is unique id, then looks up the script if there is an appropriate
visualisation and then assembles a subtree (group node) which holds both,
geometrical informations and the graphical metaphors for the information.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 */

    interface ScriptingEngine {


        /**
         * Load a translation script.
         */

        void            load_script(in            string            filename)
raises(ScriptingEngineException);

        /**
         * Main method of the scripting engine. Gets an entity id in order
to look up the visulaization script if there is a visulalisation for this
class of meta information. If there is no such class it returns the geometry
as it was in a group node. If there is a special visualisation, the script
```

```
is processed an a new subtree is assembled holding the original geometry,
the graphical metaphors and some functions applied.
         */
        VizNode::GroupNode lookup_visualization(in string entity_id) raises(
ScriptingEngineException );

        attribute                    VizScripting::BaseVisualizationFunctions
base_functions_library;
        attribute string current_visualization_script;
        attribute VizScripting::GraphicalMetaphorLibrary metaphor_library;
    };


/**
 * This interface is the library of the graphical metaphors. Every meta
information which has to be visualised has a corresponding graphical
metaphor inside ethe library.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 * @since 2003, 2004
 */
    interface GraphicalMetaphorLibrary {


        VizNode::GroupNode  get_metaphor(in  string  aMetaphor_id)  raises(
ScriptingEngineException );

        void       load_library(in       string       filename)       raises(
ScriptingEngineException );

        void       save_library(in       string       filename)       raises(
ScriptingEngineException );

        /**
         * Load a vrml file which holds a special graphical metaphor.
         */
        void       load_vrml(in       string       aMetaphor_id)       raises(
ScriptingEngineException );
};


/**
 * Base visualisation functions for the scripting engine. Every method
returns a group container which consists of the visualized part geometry
(i.e. a tesselation node) and the appropriate visualisation.
 * @author Dipl.-Ing. Martin Richter
 * @version 1.0
 * @since 2003, 2004
 */
    interface BaseVisualizationFunctions {


        VizNode::GroupNode      rotate(in      string      entity_id,      in
VizGeometry::VectorStruct axis) raises( ScriptingEngineException );

        VizNode::GroupNode     translation(in     string     entity_id,     in
VizGeometry::VectorStruct trans) raises( ScriptingEngineException );

        VizNode::GroupNode    diffuseColor(in    string    entity_id,    in
VizNode::RGBcolor color) raises( ScriptingEngineException );

        VizNode::GroupNode    ambientColor(in    string    entity_id,    in
VizNode::RGBcolor color) raises( ScriptingEngineException );
```

```
        VizNode::GroupNode    emissiveColor(in    string    entity_id,    in
VizNode::RGBcolor color) raises( ScriptingEngineException );

        VizNode::GroupNode    specularColor(in    string    entity_id,    in
VizNode::RGBcolor color) raises( ScriptingEngineException );

        VizNode::GroupNode  transparency(in  string  entity_id,  in  float
transparency) raises( ScriptingEngineException );

        VizNode::GroupNode shininess(in float entity_id, in float shininess)
raises( ScriptingEngineException );

        VizNode::GroupNode     texture(in     string     entity_id,     in
VizMaterial::TexturedMaterial texture) raises( ScriptingEngineException );

        /**
         * Time based switching of the visibility.
         */
        VizNode::GroupNode blinker(in string entity_id, in float interval)
raises( ScriptingEngineException );

        /**
         * Time based switching of the texture.
         */
        VizNode::GroupNode animation(in string entity_id, in float interval)
raises( ScriptingEngineException );

        /**
         * Exploded view of the part.
         */
        VizNode::GroupNode  exploded_view(in  float  entity_id,  in  float
factor) raises( ScriptingEngineException );

        VizNode::GroupNode add_tag(  in string entity_id, in string text, in
boolean fixed_tag) raises( ScriptingEngineException );
};

};

#endif

// eof VizScripting.idl
```

Anhang B

UML-DIAGRAMME VIZSERVICES



INTERFACE DEFINITIONEN CADSERVICES

File: CadBrep.idl

```
//File: CadBrep.idl
//CAD Services V1.2

#ifndef __CADBREP_DEFINED
#define __CADBREP_DEFINED

#include "CadUtility.idl"
#include "CadFoundation.idl"
#include "CadFeature.idl"
#include "CadGeometry.idl"

#pragma prefix "omg.org"

module CadBrep
{

  // forward references
  interface BrepEntity;
  interface Shell;
  interface OrientedShell;
  interface OrientedFace;
  interface Vertex;
  interface VertexLoop;
  interface Edge;
  interface EdgeLoop;
  interface Face;
  interface OrientedEdge;
  interface OrientedEdgeLoop;
  interface Body;

  typedef sequence<OrientedEdge>     OrientedEdgeSeq;
  typedef sequence<OrientedShell>   OrientedShellSeq;
  typedef sequence<OrientedFace>     OrientedFaceSeq;
  typedef sequence<Shell>          ShellSeq;
  typedef sequence<VertexLoop> VertexLoopSeq;
  typedef sequence<Edge>            EdgeSeq;
  typedef sequence<Vertex>        VertexSeq;
  typedef sequence<EdgeLoop>    EdgeLoopSeq;
  typedef sequence<OrientedEdgeLoop>  OrientedEdgeLoopSeq;
  typedef sequence<Face>     FaceSeq;
  typedef sequence<Body>     BodySeq;

  exception MultipleFaces
  {
                                                    FaceSeq multiples;
  };

  struct PropertyStruct
  {
    double  surface_area;
    double   volume;
    double   mass;
    double   solid_density;
    // Solid density is provide as a reference value
```

```
    CadUtility::VectorStruct  centroid;
    CadUtility::VectorStruct  inertial_moments;
    CadUtility::VectorStruct  inertial_products;
    CadUtility::VectorStruct  principle_x_axis;
    CadUtility::VectorStruct  principle_y_axis;
    CadUtility::VectorStruct  principle_z_axis;
    CadUtility::VectorStruct  gyration_radii;
    // Items relative to the frame

    CadUtility::VectorStruct inertial_moments_centroidal;
    CadUtility::VectorStruct inertial_products_centroidal;
    CadUtility::VectorStruct principle_moments_centroidal;
    CadUtility::VectorStruct gyration_radii_centroidal;
    // Items relative to the centroid

    double surface_area_error;
    double volume_error;
    double mass_error;
    // Error Values

  };


  interface BrepEntity : CadFoundation::Entity
  {

    CadFeature::DesignFeatureSeq       design_features       ()       raises
(CadUtility::CadError);
    // Sequence of the design features directly involved with the
    // creation of this entity.

    boolean is_manifold() raises (CadUtility::CadError);


  };
  typedef sequence<BrepEntity> BrepEntitySeq;

  interface Body : BrepEntity
  {
    // A collection of Brep entities defining a closed volume, aka solid.

    PropertyStruct   property_info(   inout   double   accuracy)   raises
(CadUtility::CadError);
    // Returns a structure with property info



    OrientedShellSeq oriented_shells () raises (CadUtility::CadError);
    // Returns a sequence of the associated oriented shells. The first
oriented
    // shell in the list defines the external or outside boundary of the
body.

    FaceSeq unique_faces() raises (CadUtility::CadError);
    // Returns a sequence of the unique faces composing this body

    EdgeSeq unique_edges() raises (CadUtility::CadError);
    // returns a sequence of the unique edges in this body

    VertexSeq unique_vertices() raises (CadUtility::CadError);
    // returns a sequence of unique vertices in this body
```

```
    CadGeometry::ConnectedFaceTessellationStruct tessellate (
        in CadGeometry::TessType t_type,
        inout CadGeometry::TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters
    // If Flag is true the TessParameters were changed
  };

  interface OrientedShell : BrepEntity
  {
    // An oriented use of a shell.
    // An oriented shell must always be used by at least one body

    Body get_body () raises (CadUtility::CadError);
    // Returns the body that uses this oriented shell.



    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the oriented shell agrees with the
    //direction of the underlying shell.


    Shell get_shell () raises (CadUtility::CadError);
    // Returns the shell associated with this oriented entity.
  };


  interface Shell : BrepEntity
  {
    // An collection of oriented faces.
    // An independent, open shell can represent a skin or quilt.

    double area( inout double accuracy) raises (CadUtility::CadError);
    // shell area - accuracy is implementation defined




    boolean is_closed() raises (CadUtility::CadError);



    OrientedFaceSeq oriented_faces () raises (CadUtility::CadError);
    // Returns a sequence of the oriented faces in this shell.
    //The ordering of the oriented faces in this sequence has no
significance.



    OrientedShellSeq oriented_shells () raises (CadUtility::CadError);
    // Returns a sequence of the oriented shells that use this shell.
    //Returns an empty sequence if this shell is independent.



    CadGeometry::FaceTessellationStructSeq          tessellate          (in
CadGeometry::TessType t_type,
        inout CadGeometry::TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters
    // If Flag is true the TessParameters were changed
  };
```

```
  interface Vertex : BrepEntity
  {
    // A topological point.

    EdgeSeq get_edges () raises (CadUtility::CadError);
    // Returns a sequence of the edges that use this vertex.
    // Returns an empty sequence if this vertex is independent.




    CadUtility::PointStruct location() raises (CadUtility::CadError);
    // Returns the 3D coordinates.


    VertexLoopSeq vertex_loops() raises (CadUtility::CadError);
    // Returns a sequence of the vertex loops that use this vertex.

  };


  interface VertexLoop : BrepEntity
  {
    // A topological pole or point location used to define the boundary of a
face.
    // Examples include the pole of a sphere or a cone.
    // A vertex loop must always be used by a face (never independent).

    Vertex loop_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the 3D location of this vertex loop.



    Face get_face () raises (CadUtility::CadError);
    // Returns the face that uses this vertex loop.
    // Since vertex loops cannot be independent, this object must be used to
    // construct an edge loop before it is considered valid.


  };

  interface EdgeLoop : BrepEntity
  {
    OrientedEdgeLoopSeq oriented_edge_loops() raises (CadUtility::CadError);
    // oriented edge loops that reference this edge loop

    OrientedEdgeSeq oriented_edges() raises (CadUtility::CadError);
    // oriented edges that compose the edge loop

  };

  interface OrientedEdgeLoop : BrepEntity
  {
    boolean      sense() raises (CadUtility::CadError);
    // true indicates agreement with the underlying edge loop

    Face      get_face() raises (CadUtility::CadError);
    EdgeLoop get_edge_loop() raises (CadUtility::CadError);
  };
```

```
  interface OrientedFace : BrepEntity
  {
    // An oriented use of a face.

    Face get_face () raises (CadUtility::CadError);
    // Returns the face associated with this oriented entity.


    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the oriented face agrees with the
    // direction of the underlying face.


    Shell get_shell () raises (CadUtility::CadError);
    // Returns the shell that uses this oriented face.
  };

  enum Location
  {
                                                    // Location enumeration
                                                                  INSIDE,
                                                            ON_BOUNDARY,
                                                                 OUTSIDE

  };

  interface Face : BrepEntity
  {
    readonly attribute CadUtility::RangeStruct range_u;
    readonly attribute CadUtility::RangeStruct range_v;
    // bounds of the active region of the face as defined by the inner and
outer loops.


    double area( inout double accuracy) raises (CadUtility::CadError);
    // Evaluates area to a specified accuracy.
    // Accuracy is implementation defined.



    boolean intersect_ray ( in CadUtility::RayStruct ray, in double
tolerance,
        out CadUtility::PointStructSeq intersection_points,
        out CadUtility::UvStructSeq intersection_parameters)
         raises (CadUtility::CadError);
    // Evaluates the intersections between the specified ray and the face.
    // The tolerance defines how close the ray must come to the face to be
considered
    // an intersection. Returns TRUE if any intersections were found, FALSE
if not.
    // Any intersections are returned in two sequences: one of 3D points and
one of
    // corresponding 2D parameter values on the face's surface.


    Location is_location_inside (in CadUtility::UvStruct location)
        raises (CadUtility::CadError);
    // Queries if a location (defined by uv parameter values) is in the
active region
    // of the face as defined by the inner and outer loops.
```

```
    OrientedEdgeLoopSeq oriented_edge_loops () raises
(CadUtility::CadError);
    // Returns a list of the associated Brep.OrientedEdgeLoop entities.
    // The first oriented edge loop in the list defines the outside boundary
of the face.



    OrientedFaceSeq oriented_faces () raises (CadUtility::CadError);
    // Returns a list of the associated CadBrep::OrientedFace entities.
    //Returns an empty list if this face is independent, e.g. a trimmed
surface.


    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the face agrees with the parametric
(normal)
    // direction of the underlying surface.
    // Critical for determining the "outside" of a face in a body, for
example.



    CadGeometry::Surface surface () raises (CadUtility::CadError);
    // Returns the CadGeometry::Surface entity that defines the shape of
this face.


    CadGeometry::FaceTessellationStruct tessellate (in CadGeometry::TessType
t_type,
        inout CadGeometry::TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters
    // If Flag is true the TessParameters were changed



    VertexLoopSeq vertex_loops () raises (CadUtility::CadError);
    // Returns a sequence of any vertex loops defined on this face.
  };


  interface OrientedEdge : BrepEntity
  {
    // An oriented use of an edge.

    Edge get_edge () raises (CadUtility::CadError);
    // Returns the edge associated with this oriented entity.


    EdgeLoop edge_loop () raises (CadUtility::CadError);
    // Returns the edge loop that uses this oriented edge.


    Vertex start_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the start of this oriented edge.

    Vertex end_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the end of this oriented edge.
    // Takes into account any sense differences.
```

```
    OrientedFace      oriented_face      ()      raises      (MultipleFaces,
CadUtility::CadError);
    // Returns the oriented face that uses this oriented edge.
    // Returns NULL if the oriented edge is in an independent edge loop or
    // bounds an independent face.
    // Raises an exception if more than one oriented face uses this oriented
edge.

    Face get_face() raises (MultipleFaces, CadUtility::CadError);



    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the oriented edge (from start to
    // end vertices) agrees with the direction of the underlying edge.
 };


  interface Edge : BrepEntity
  {
    // A trimmed portion of a curve.An edge that uses the same vertex for
both start and
    // end vertices must be defined as a closed edge on a closed curve
starting and ending
    // at this vertex.  An independent edge can be used to represent a
trimmed curve.

    CadGeometry::Curve curve()  raises (CadUtility::CadError);
    // Returns the curve that defines the shape of this edge in model space.


    double length ( inout double accuracy )  raises (CadUtility::CadError);
    // Evaluates the length of the edge to a specified accuracy.
    // The accuracy is implementation defined.


    CadUtility::NurbsCurveStruct   nurbs_representation   (   inout   double
tolerance )
         raises (CadUtility::CadError);
    //  Returns a NURBS curve that approximates this edge within the
specified tolerance.


    OrientedEdgeSeq oriented_edges () raises (CadUtility::CadError);
    // Returns a sequence of the oriented edges that use this edge
    // Returns an empty sequence if this edge is independent.


    boolean sense () raises (CadUtility::CadError);
    //  Queries  whether  the  direction  of  the  edge  (from  start  to  end
vertices)
    // agrees with the parametric direction of the underlying curve.


    double start_parameter () raises (CadUtility::CadError);
    // Returns the curve parameter corresponding to the start vertex.

    double end_parameter ()  raises (CadUtility::CadError);
    // Returns the curve parameter corresponding to the end vertex.

    Vertex start_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the start of this edge.
```

```
    Vertex end_vertex () raises (CadUtility::CadError);
    // Returns the final vertex that defines this edge.


    CadGeometry::EdgeTessellationStruct tessellate ( in double tolerance)
         raises (CadUtility::CadError);
    // Tessellates the edge to a specified chordal deviation tolerance.


    CadUtility::VectorStruct unit_tangent (in double parameter,in boolean
sense)
         raises (CadUtility::CadError);
    // Evaluates the unit tangent vector of the edge at the specified
parameter and sense.
    // If the sense is TRUE, the tangent vector is oriented with the edge.
    // If the sense is FALSE, the tangent vector is oriented in the opposite
direction.




    VertexSeq unique_vertices () raises (CadUtility::CadError);

    // Returns a sequence of unique vertices used by this edge.
  };
};
#endif
```

File: CadConnection.idl

```
//File: CadConnection.idl
//CAD Services V1.2

#ifndef __CADCONNECTION_DEFINED
#define __CADCONNECTION_DEFINED


#include "CadMain.idl"
#include "CadUtility.idl"
#include "CadFoundation.idl"
#include "CosPropertyService.idl"

#pragma prefix "omg.org"

module CadConnection
{
  //forward references
  interface CadServer;
  interface CadSystem;
  interface CadUserInterface;

  enum ActivationMode { ACTIVE_READONLY, ACTIVE_CHECKOUT, ACTIVE_DETAILED };


  // various exceptions
  exception CadConnectionFault
  {
    unsigned long error_code;
                                                        string error_text;
  };

  exception ValidationError
```

```
{
  unsigned long error_code;
                                                    string error_text;
};

exception InvalidProperties
{
  unsigned long error_code;
                                                    string error_text;
};

exception PermissionDenied
{
  unsigned long error_code;
                                                    string error_text;
};

exception InvalidModel
{
  unsigned long error_code;
                                                    string error_text;
};

exception BadCommand
{
                                              unsigned long error_code;
                                                    string error_text;
};
exception ModelNotLoaded
{
  unsigned long error_code;
  string error_text;
};
exception BadParameters
{
  CosPropertyService::Properties flawed_params;
  CadUtility::StringSeq reasons;
};

struct OptionsStruct
{
  boolean is_interactive;
  boolean is_persistent_id;
  boolean is_accurate; // for some calculated properties
  boolean is_parametric; // design feature support
  boolean is_extended_geometry;
};

struct NativeCadAttributesStruct
{
  string  cad_sys_name;
  string  vendor_name;
  string  cad_version;
  OptionsStruct optional_properties;
  CosPropertyService::Properties  other_properties;
};

typedef sequence<string> ModelList;

interface CadServer
{
  readonly attribute NativeCadAttributesStruct native_properties;
  // NativeCADSys contains CAD Vendor name, version, etc.
```

```
    readonly attribute CosPropertyService::Properties  launch_properties;
    // This attribute contains all needed information to launch
    // the CAD system

    CadSystem connect( in CosPropertyService::Properties props)
        raises ( CadConnectionFault );
    // secure connection

    CadSystem connect_with_password(  in string user, in string password,
        in CosPropertyService::Properties props )
        raises ( CadConnectionFault );
    // open wire (unsecure connection)

  };

  struct Path
  {
    boolean  is_dir;  //  Indicates  whether  Path  represents  a  file  or  a
directory
    string    unique_path;      //  Canonical,  absolute,  and  unique  path
representing a
                    // file or a directory
  };

  typedef sequence<string> RootsList;
  typedef sequence<Path>   PathList;

 interface CadSystem
 {
    CadMain::Model  open_model(in string model, in ActivationMode access)
        raises (InvalidModel, PermissionDenied, CadUtility::CadError);
    // CAD model related operations

    CadMain::Model create_model(in string new_name,
        in CadUtility::MassUnit m_unit,in CadUtility::LengthUnit l_unit,

                            in CosPropertyService::Properties model_params)
        raises (PermissionDenied, BadParameters, CadUtility::CadError);
    // Creates and opens a new model in native CAD system

    CosPropertyService::Properties get_properties();
    void set_properties( in CosPropertyService::Properties props );
    // Allows reading and changing of CadSystem properties

    ModelList  available_models();
    // returns a list if model available to the active  CAD System.

    CadMain::ModelSeq opened_models();
    // returns a list of active models

    void execute_cad_command(in string command_string , inout any comm_out )
                                                        raises (BadCommand);
    // extensible Cad system command interface

    CadUserInterface get_gui() raises (CadFoundation::GuiUnsupported);
    // access to GUI interface

    void disconnect() raises ( CadConnectionFault );
    // close system and clean-up

    // v 1.2 additions for directory information
    // Creates a new folder in a given directory.
```

```
    // param parent_dir: The (canonical, absolute, and unique) path of the
    // directory, in which the folder will be created. This must be either
    // one of the root directories (see get_root_directories()) or a
    // directory that exists in one of the root directory's subdirectories
    // param folder_name: The name of the folder to be created. It must be a
    // valid and a not already existing name.
    // raises CadUtility::CadError: If there was an error in creating the
    // folder.
    void create_new_folder(in string parent_dir, in string folder_name)
    raises (CadUtility::CadError);

    // Returns a list of (canonical, absolute, and unique) pathnames repre-
    // senting the root directories. In a root directory or in any other
    // subdirectory, model files can be found.
    // return: List of pathnames representing root directories.
    RootsList get_root_directories();

    // Returns a list of subdirectories and model files of a given
directory.
    // param directory: (canonical, absolute, and unique) path of the
    // directory from which to get the list of subdirectories and model
    // files. This must be either one of the root directories or a
subdirectory.
    // return: A list representing subdirectories and/or model files
    // of the given directory.
    // raises CadUtility::CadError If there was an error retrieving the
list.
    PathList get_models_and_folders(in string directory)
           raises (CadUtility::CadError);

    // Returns the parent directory of a given directory.
    // param directory: (canonical, absolute, and unique) path of the
    // directory from which to get its parent directory. The given directory
    // should be a subdirectory of one of the root directories.
    // return: The path of the parent directory.
    // raises CadUtility::CadError If there was an error retrieving the
    // parent directory.
    string get_parent_directory(in string directory)
          raises (CadUtility::CadError);
    };


  // OPTIONAL interface supporting an interactive User Interface on native
CAD system

  struct UiMessageStruct
  {
    unsigned long code; //recommended
    string note;
  };

  interface CadUserInterface
  {

    CadFoundation::EntitySeq          get_selected_entities()          raises
(CadUtility::CadError);
    // returns a sequence of entities selected in UI

    CadUtility::WarningStructSeq                          highlight_entities(in
CadFoundation::EntitySeq marked);
    // highlight these entities in the UI. returns warning if all entities
cannot be highlighted
```

```
    void     set_entity_label_visibility     (in     CadFoundation::EntitySeq
entities,in boolean visibility)
        raises (CadUtility::CadError);
    // Sets the visibility of entity labels.

    CadFoundation::EntitySeq select_entities (in UiMessageStruct prompt, out
UiMessageStruct error_message)
        raises (CadUtility::CadError);
    //prompt explains the selection request

    UiMessageStruct  prompt_for_string  (in  UiMessageStruct  prompt)  raises
(CadUtility::CadError);
    // Prompts the user to input a value (in UI) which is returned as an
    // unsigned long (recommended) string.

    void    createWindow    (    in    CadMain::Model    model_gui)    raises
(CadUtility::CadError);
    // create window for model and place in foreground

    void    deleteWindow    (    in    CadMain::Model    model_gui)    raises
(CadUtility::CadError);
    // delete window associated with the model

    void    hideWindow    (    in    CadMain::Model    model_gui)    raises
(CadUtility::CadError);
    // places window in background

    void   foregroundWindow   (   in   CadMain::Model   model_gui)   raises
(CadUtility::CadError);
    // place window in foreground
  };

};

#endif
```

File: CadFeature.idl

```
//File: CadFeature.idl
//CAD Services V1.2

#ifndef __CADFEATURE_DEFINED
#define __CADFEATURE_DEFINED

#include "CadUtility.idl"
#include "CadFoundation.idl"

#pragma prefix "omg.org"

module CadFeature
{
  interface DesignFeature;
  interface Parameter;

  typedef sequence<Parameter>    ParameterSeq;
  typedef sequence<DesignFeature> DesignFeatureSeq;

  interface Parameter
  {
    //   data   structures   that   capture   the   (parametric)   features   of
ModelEntites

    readonly attribute boolean            is_read_only;
```

```
    readonly attribute boolean            is_independent;
    readonly attribute string             name;

    string   get_expression() raises (CadUtility::CadError);
    void set_expression(in string e_value) raises (CadUtility::CadError);
    // operations to allow an expression that may drive geometry

    CadUtility::EntityAttrib  get_value() raises (CadUtility::CadError);
    void     set_value(in     CadUtility::EntityAttrib     value)     raises
(CadUtility::CadError);
    // operations providing access to parameter value
  };

  interface DesignFeature  :  CadFoundation::Entity
  {
    // A distinct step or node in the parametric definition of a model.
    //  It  drives  the  creation  of  a  set  of  Brep  entities  in  the  fully-
evaluated
    // form of the model.

    boolean         is_suppressed() raises (CadUtility::CadError);
    void           set_suppression() raises (CadUtility::CadError);
    ParameterSeq get_parameter_set() raises (CadUtility::CadError);
  };
};
#endif
```

File: CadFoundation.idl

```
//File: CadFoundation.idl
//CAD Services V1.2

#ifndef __CADFOUNDATION_DEFINED
#define __CADFOUNDATION_DEFINED

#include "CadUtility.idl"

#pragma prefix "omg.org"

module CadFoundation
{
  // Encapsulates general elements and behavior that are shared by all model
entities.

  // forward references

  interface Attributable;
  interface Entity;

  interface EntityGroup;
  interface Layer;

  typedef sequence<Entity> EntitySeq;
  typedef sequence<Layer> LayerSeq;
  typedef sequence<EntityGroup> EntityGroupSeq;

  // exceptions

  exception PidUnsupported {};
  exception GuiUnsupported {};
  exception UnBoundedEntity
  {
                                                    string unbounded_name;
```

```
  };
  exception NotIndependent{
                                                        string dependancy;
  };
  exception ReadOnlyEntity{};

  interface Attributable
  {
                             // General interface allowing geometry tagging
//  The  following  operations  should  use  DynAnys  to  extract  attribute
information

                             any  get_info() raises (CadUtility::CadError);
            void set_info(in any dyn_value) raises (CadUtility::CadError);
   };

  struct EntityPropsStruct
  {
    // Properties of Entity in a struct

                                                      boolean is_top_level;
                                                       string native_label;
                                                         string native_type;
                                                      boolean is_committed;
                          CadUtility::PresentationStruct presentation_info;
                                                             long dimension;
                                                  boolean is_native_valid;
                                                       boolean is_visible;
                                                           long unique_id;
                                                     string persistent_id;
                                     CadUtility::PointStructSeq ref_position;
         CadUtility::TransformationStruct global_position;
  };

 interface Entity : Attributable
 {
    // Provides CAD functionality + properties to be inherited by geometry
objects
    EntityPropsStruct get_entity_props() raises (CadUtility::CadError);
    // operation providing grouped access to Entity properties

    readonly attribute boolean is_top_level;
    // Top level entity?

    readonly attribute string native_label;
    //  Provides  a  brief  description  of  the  entity  using  system-specific
terminology.
    // Not guaranteed to be unique within a model.

    readonly attribute string native_type;
    // The system-specific type name of this entity.

    CadUtility::PresentationStruct get_presentation_info()
         raises (CadUtility::CadError);
    // Struct containing relevant presentation information.

    void    set_color(in    CadUtility::ColorStruct        color)    raises
(CadUtility::CadError);
    //  set color


    CadUtility::BoundingBox   bounding_box   ()   raises   (UnBoundedEntity,
CadUtility::CadError);
```

```
    // Returns an approximate BoundingBox around the entity.
    // Returns an error if the entity is unbounded in one or more
directions.


    Object  cad_model() raises (CadUtility::CadError);
    // Returns the CadMain::Model object that contains this entity.
    // Reference must be narrowed to CadMain::Model


    long euclidean_dimension ()  raises (CadUtility::CadError);
    // Returns the Euclidean dimension of the entity.


    boolean is_native_valid () raises (CadUtility::CadError);
    // Queries if the native entity is valid according to any internal
checks
    // provided by the CAD system.


    boolean is_visible ()  raises (CadUtility::CadError);
    // Queries whether the entity is visible or not (blanked, no-showed,
hidden).


    LayerSeq entity_layers ()  raises (CadUtility::CadError);
    // Returns a sequence of the Layers that contain this entity


    long get_unique_id() raises ( CadUtility::CadError );
    // Identifier that is guaranteed to be unique across all entities in a
Model.
    // This identifier is not persistent (i.e.valid only during CadServer
Session).


    string      get_persistent_id      ()      raises      (PidUnsupported,
CadUtility::CadError);
    // Returns an identifier intended to identify this entity between
interface sessions.


    CadUtility::PointStructSeq      reference_position      ()      raises
(CadUtility::CadError);
    // Returns a sequence of reference coordinates on
    // the entity that are unique relative to neighboring entities.
    // Provided as a convenience for client graphics and other tagging
applications

    CadUtility::TransformationStruct        global_location()        raises
(CadUtility::CadError);
    // Provides global coordinate location information


    void transform (in CadUtility::TransformationStruct  transformation)
```

```
        raises (NotIndependent, ReadOnlyEntity, CadUtility::CadError);


    // Applies the specified transformation (rotations and translation) to
the entity.
    // Throws an exception if the entity cannot be transformed.

  };

  interface EntityGroup
  {
    // A generalized grouping of entities within a model that is not related
to
    // layering. Provides a mechanism for grouping whose semantics lie
outside the
    // standard, e.g. application-specific collections

    readonly attribute long count;
    // Number of entities in group.

                            readonly attribute string entity_group_label;
                                        // a label for this entity group

    CadFoundation::EntitySeq entities ()  raises (CadUtility::CadError);
    // Returns a sequence of entities defined in this group.

    void add_entities (in CadUtility::LongSeq entities_uids)
         raises (CadUtility::CadError);
    // Adds the specified entities to this group.
    // v1.2 use unique ids as opposed to entities

    void remove_entities (in CadUtility::LongSeq entities_uids)
        raises (CadUtility::CadError);
    // Removes the specified entities from this group.
    // Does not delete the entity objects.
    // v1.2 use unique ids as opposed to entities
  };

  interface Layer : EntityGroup
  {
    // An collection of entities that corresponds to the layers.

    readonly attribute string layer_identifier;
    // String identifier of layer.

    CadUtility::ColorStruct get_color()
         raises (CadUtility::CadError);

    void set_color( in CadUtility::ColorStruct new_color);
    // change the color of all entities in this layer

    boolean is_visible() raises (CadUtility::CadError);

  };

};

#endif
```

File: CadGeometry.idl

```
//File: CadGeometry.idl
```

```
//CAD Services V1.2

#ifndef __CADGEOMETRY_DEFINED
#define __CADGEOMETRY_DEFINED

#include "CadUtility.idl"
#include "CadFoundation.idl"

#pragma prefix "omg.org"

module CadGeometry
{
  // Fundamental Geomety defintitions

  //forward references
  interface Curve;
  interface Surface;

  typedef sequence<Curve>        CurveSeq;
  typedef sequence<Surface>      SurfaceSeq;

  enum TessType
  {
    // an enumeration of possible types of tessellations

    WIREFRAME,
    VISUALIZATION
  };

  struct TessParametersStruct
  {
    // parameters used with the Tessellation creation

    double max_chord;
    // maximum deviation between triangle center and surface

    double max_size;

    double angle;
    // deviation between normals of facets - in degrees
  };

  struct IndexStruct
  {
    // struct supporting triangle specification
    // i1 connects to i2, i2 to i3, and i3 to i1

    long i1;
    long i2;
    long i3;
  };
  typedef sequence<IndexStruct> IndexSeq;

  struct EdgeTessellationStruct
  {
    // edge tessellation

    Object obj_ref;
    // Object reference to underlying topology

    CadUtility::PointStructSeq epts;
    // sequence of pts defining edge tessellation (first struct is the
starting pt)
```

```
  CadUtility::LongSeq vertex_number;
  // index numbering for all points - relating to epts above

  CadUtility::DoubleSeq t_values;
  //sequence of doubles for t parameters
};
typedef sequence<EdgeTessellationStruct> EdgeTessellationStructSeq;

struct TessellationStruct
{
  // basic tessellation structure, please see JCAD submission for indexing

  Object obj_ref;
  // Object reference to underlying topology

  TessType t_type;
  // Application specific type for this tessellation

  CadUtility::PointStructSeq xyz;
  // sequence of 3D pts defining triangles on the Face(length = npts)

  CadUtility::LongSeq face_pts;
  // index numbering for all points - relating to xyz above

  CadUtility::VectorStructSeq normals;
  // sequence of normals at vertices

  CadUtility::UvStructSeq uv;
  // uv parameters associated with the pts (length = npts)

  IndexSeq index_list;
  // Index list is a set of 3 values (i1,i2,i3) as pointers into the
  // points/normals/uv values to define a triangle. To allow pt
  // sharing accross faces the vertex_number sequence is consistent
  // with face_pts. Please see Appendix B.
};

struct FaceTessellationStruct
{
  EdgeTessellationStructSeq edges;
  // sequence of edge tessellations

  TessellationStruct face_tessellation;
  // Face - specific tessellation data
};

typedef sequence<FaceTessellationStruct> FaceTessellationStructSeq;

struct ConnectedFaceTessellationStruct
{

  Object obj_ref;
  // Object reference to underlying topology

  long max_vertex_number;
  // total vertices used for tessellation (all faces)

  FaceTessellationStructSeq all_faces;
  // all face tessellations supoorting this body
};

struct CurvePropsStruct
```

```
  {
    // Properties of a Curve

    boolean is_bounded;
    boolean is_closed;
    CadUtility::RangeStruct range;
  };

  interface Curve : CadFoundation::Entity
  {
    CurvePropsStruct get_curve_props()
         raises (CadUtility::CadError);
    // recommended access operation for curve properties

    readonly attribute boolean is_bounded;

    readonly attribute boolean is_closed;


    double length(inout double accuracy)  raises (CadUtility::CadError);
    // Accuracy request is implementation defined.

    CadUtility::RangeStruct range()  raises (CadUtility::CadError);


    CadUtility::PointStructSeq evaluate_points (
        in CadUtility::DoubleSeq parameters,
        in boolean direction_sense,
        in long derivative_count,
        out CadUtility::VectorStructSeqSeq derivatives)
         raises (CadUtility::CadError);
    // Evaluates a curve at the specified parameters.



    CadUtility::DoubleSeq   evaluate_curvatures   (in   CadUtility::DoubleSeq
parameters,
        in boolean direction_sense) raises (CadUtility::CadError);
    // Evaluates the curvature of a curve at the specified parameters.




    CadUtility::VectorStructSeq evaluate_normals (
        in CadUtility::DoubleSeq parameters,
        in boolean direction_sense) raises (CadUtility::CadError);
    // Evaluates the normal of a curve at the specified parameters.

    boolean   intersect_ray   (in   CadUtility::RayStruct   i_ray,   in   double
tolerance,
        out CadUtility::PointStructSeq intersection_points,
        out CadUtility::DoubleSeq intersection_parameters)
         raises (CadUtility::CadError);
    // Evaluates the intersections between the specified ray and the curve.




    boolean is_planar (out CadUtility::RayStruct ray)
         raises (CadUtility::CadError);
    // Queries if the curve is planar.
    //If so, the returned ray defines a point and direction for this plane.
```

```
    CadUtility::NurbsCurveStruct    nurbs_representation    (inout    double
tolerance,
        in double t_min,in double t_max)  raises (CadUtility::CadError);
    // Returns a NURBS curve that represents this curve within the specified
tolerance.
    // If the representation is exact tolerance will be returned as a
negative value



    boolean project_points_to_nearest (in CadUtility::PointStructSeq points,
        out CadUtility::DoubleSeq params,
        out CadUtility::PointStructSeq projected_points,
        out CadUtility::WarningStructSeq warnings)
         raises (CadUtility::CadError);
    // Projects each specified point to the nearest point on the curve.


    boolean project_point_to_nearest (in CadUtility::PointStruct point,
        out double param,
        out    CadUtility::PointStruct    projected_point,    out    string
warning_string)
         raises (CadUtility::CadError);
    // Projects a single point (not recommended for points)


    CadGeometry::EdgeTessellationStruct tessellate ( inout double tolerance)
         raises (CadUtility::CadError);
    // Tessellates the curve to a specified chordal deviation tolerance.
    // v 1.2 If tessellation is exact, tolerance will be returned as a
negative

  };


  struct SurfacePropsStruct
  {
    // surface properties

    boolean is_bounded_u;
    boolean is_bounded_v;
    CadUtility::RangeStruct range_u;
    CadUtility::RangeStruct range_v;
    boolean is_closed_u;
    boolean is_closed_v;
  };

  struct SurfaceCurvatureStruct
  {
    double min_curvature;
    double max_curvature;
    CadUtility::VectorStruct min_princ_direction;
    CadUtility::VectorStruct max_princ_direction;
  };
  typedef sequence<SurfaceCurvatureStruct> SurfaceCurvatureStructSeq;

  interface Surface : CadFoundation::Entity
  {
    SurfacePropsStruct get_surface_props()
        raises (CadUtility::CadError);
```

```
    // recommended access operation for surface properties

    readonly attribute boolean is_bounded_u;
    readonly attribute boolean is_bounded_v;

    CadUtility::RangeStruct range_u()  raises (CadUtility::CadError);
    CadUtility::RangeStruct range_v()  raises (CadUtility::CadError);

    boolean is_closed_u() raises (CadUtility::CadError);
    boolean is_closed_v() raises (CadUtility::CadError);

    double area (inout double accuracy) raises (CadUtility::CadError);
    // Evaluates the area to a specified accuracy.
    // Accuracy is implementation defined.



    CadUtility::PointStructSeq evaluate_points (
        in CadUtility::UvStructSeq uv_parameters,
        in boolean direction_sense_u,in boolean direction_sense_v,
        in long derivative_count,
        out CadUtility::VectorStructSeqSeqSeq derivatives)
         raises (CadUtility::CadError);
    // Evaluates a surface at the specified parameters.


    CadUtility::PointStruct    evaluate_point    (in    CadUtility::UvStruct
uv_point,
        in boolean direction_sense_u,in boolean direction_sense_v,
        in long derivative_count,
        out CadUtility::VectorStructSeqSeq derivatives)
         raises (CadUtility::CadError);
    // Single point operation (not recommended).

    SurfaceCurvatureStructSeq             evaluate_curvatures             (in
CadUtility::UvStructSeq uv_parameters,
        in boolean direction_sense_u,in boolean direction_sense_v)
         raises (CadUtility::CadError);
    // Evaluates the curvature of a surface at the specified parameters.



    CadUtility::VectorStructSeq evaluate_normals (in CadUtility::UvStructSeq
uv_parameters,
        in boolean direction_sense_u,in boolean direction_sense_v)
         raises (CadUtility::CadError);
    // Evaluates the normal of a surface at the specified parameters.




    CadUtility::NurbsSurfaceStruct   nurbs_representation   (inout   double
tolerance,
        in double low_bound_u, in double high_bound_u,
        in double low_bound_v,in double high_bound_v)
         raises (CadUtility::CadError);
    // Returns a NURBS surface that represents this surface.
    // If nurbs representation is exact, tolerance will be returned as a
negative



    CadGeometry::TessellationStruct tessellate (in TessType t_type,
```

```
        inout TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters.
    // If Flag is true the TessParameters were changed (original values
could not be achieved)

    boolean project_points_to_nearest (in CadUtility::PointStructSeq points,
        out CadUtility::UvStructSeq params,
        out CadUtility::PointStructSeq projected_points,
        out CadUtility::WarningStructSeq warnings)
         raises (CadUtility::CadError);
    // Projects each specified point to the nearest point on the surface.



    boolean project_point_to_nearest (in CadUtility::PointStruct point,
        out CadUtility::UvStruct param,
        out    CadUtility::PointStruct    projected_point,    out    string
warning_string)
         raises (CadUtility::CadError);
    // Projects a single point - NOT recommended

    boolean intersect_ray (in CadUtility::RayStruct i_ray,
        in double tolerance,
        out CadUtility::PointStructSeq intersection_points,
        out CadUtility::UvStructSeq intersection_parameters)
         raises (CadUtility::CadError);
    // Evaluates the intersections between the specified ray and
    // the surface.

    boolean is_planar (out CadUtility::RayStruct ray)
         raises (CadUtility::CadError);
    // Queries if the surface is planar.
    // If so, the returned ray defines a point and direction for
    // this plane.

  };
};
#endif
```

File: CadGeometryExtens.idl

```
//File: CadGeometryExtens.idl
//CAD Services V1.2

#ifndef  CADGEOMETRYEXTENS_DEFINED
#define                                          CADGEOMETRYEXTENS_DEFINED

#include "CadGeometry.idl"
#include "CadFoundation.idl"
#include "CadUtility.idl"

#pragma prefix "omg.org"

module CadGeometryExtens{

module CadSurface{

  interface BoundedSurface; //forward reference

  enum     TransitionCode{DISCONTINUOUS,     CONTINUOUS,     CONT_SAME_GRAD,
CONT_SAME_GRAD_SAME_CURVATURE};
```

```
  struct SurfacePatchStruct{
    BoundedSurface parent_surface;
    TransitionCode u_transition;
    TransitionCode v_transition;
    boolean u_sense;
    boolean v_sense;
  };

  interface BoundedSurface : CadGeometry::Surface {
    SurfacePatchStruct bsurface_info() raises (CadUtility::CadError);
  };

  struct ConicalSurfStruct{
    CadUtility::TransformationStruct location;
    double radius;
    double semi_angle;
  };

  interface ConicalSurf : CadGeometry::Surface{
    ConicalSurfStruct cs_info() raises (CadUtility::CadError);
  };

  struct CylinderStruct{
    CadUtility::TransformationStruct location;
    double radius;
  };

  interface Cylinder: CadGeometry::Surface {
    CylinderStruct cylinder_info() raises (CadUtility::CadError);
  };

  struct HyperbolaStruct{
    CadUtility::TransformationStruct location;
    double semi_axis;
    double semi_imag_axis;
  };

  interface NurbsSurface : BoundedSurface{
    CadUtility::NurbsSurfaceStruct            nurbs_info()            raises
(CadUtility::CadError);
  };

  interface OffsetSurface : CadGeometry::Surface {
    CadGeometry::Surface basis_surface() raises (CadUtility::CadError);
    double distance() raises (CadUtility::CadError);
    boolean self_intersect() raises (CadUtility::CadError);
  };

  interface SurfaceRev : CadGeometry::Surface {
    CadGeometry::Curve swept_curve() raises (CadUtility::CadError);
    CadUtility::RayStruct axis_line() raises (CadUtility::CadError);
  };

  struct SphereStruct{
    double radius;
    CadUtility::TransformationStruct location;
  };

  interface Sphere: CadGeometry:: Surface{
    SphereStruct sphere_info() raises (CadUtility::CadError);
  };

  struct ToroidStruct{
```

```
    CadUtility::TransformationStruct location;
    double major_radius;
    double minor_radius;
  };

  interface Toroid : CadGeometry::Surface{
    ToroidStruct toroid_info() raises (CadUtility::CadError);
  };

  interface Plane : CadGeometry::Surface{
    CadUtility::TransformationStruct location()
            raises (CadUtility::CadError);
  };

  interface SurfLinExtrusion : CadGeometry:: Surface{
    CadGeometry::Curve swept_curve() raises (CadUtility::CadError);
    CadUtility::VectorStruct extrusion_axis() raises (CadUtility::CadError);
  };
};

module CadCurve{

  struct CircleStruct{
    CadUtility::TransformationStruct location;
    double radius;
  };

  interface Circle : CadGeometry::Curve{
    CircleStruct circle_info() raises (CadUtility::CadError);
  };

  struct CompositeCurveStruct{
    long count;
    CadGeometry::CurveSeq segments;
    CadUtility::BooleanSeq senses;
  };

  interface CompositeCurve: CadGeometry::Curve{
    CompositeCurveStruct comp_curve_info() raises (CadUtility::CadError);
  };

  struct ParabolaStruct{
    CadUtility::TransformationStruct location;
    double focal_distance;
  };

  interface Parabola : CadGeometry::Curve {
    ParabolaStruct parabola_info() raises (CadUtility::CadError);
  };

  interface Hyperbola : CadGeometry::Curve {
    CadGeometryExtens::CadSurface::HyperbolaStruct hyperbola_info()
              raises (CadUtility::CadError);
  };

  struct LineStruct{
    CadUtility::PointStruct the_point;
    CadUtility::VectorStruct direction;
    };

  interface Line : CadGeometry::Curve {
    LineStruct line_info() raises (CadUtility::CadError);
  };
```

```
  struct OffsetCurveStruct{
    double distance;
    CadUtility::VectorStruct ref_direction;
    boolean self_intersect;
  };

  interface OffsetCurve : CadGeometry::Curve{
    OffsetCurveStruct offset_info() raises (CadUtility::CadError);
  };

  struct EllipseStruct{
    CadUtility::TransformationStruct location;
    double semi_axis_1;
    double semi_axis_2;
  };

  interface Ellipse : CadGeometry::Curve {
    EllipseStruct ellipse_info() raises (CadUtility::CadError);
  };


  struct TrimmedCurveStruct{
    CadUtility::PointStruct trim_1;
    CadUtility::PointStruct trim_2;
    boolean sense_agreement;
  };

  interface TrimmedCurve : CadGeometry::Curve{
    TrimmedCurveStruct t_curve() raises (CadUtility::CadError);
  };

  struct SurfaceCurveStruct{
    CadGeometry::Curve curve_3d;
    long basis_count;
    CadGeometry::SurfaceSeq basis_surfaces;
  };
 };
};

#endif
```

File: CadMail.idl

```
//File: CadMain.idl
//CAD Services V1.2

#ifndef __CADMAIN_DEFINED
#define __CADMAIN_DEFINED

#include "CadUtility.idl"
#include "CadBrep.idl"
#include "CadFeature.idl"

#pragma prefix "omg.org"

module CadMain
{
  // forward references
  interface ModelInstance;
  interface Model;

  // exceptions
```

```
exception RegenerationException
{
  string reason;
  any support;
};
exception ReturnToValidFail
{
  string reason;
};
exception UnboundedEntity {};

exception  NotValidCadType
{
  CadUtility::TypeCodeSeq bad_types;
};

exception  NotValidUid
{
  CadUtility::LongSeq bad_ids;
};

exception SaveFault
{
                                                    string error_text;
};

exception SaveAsFault{
                                                    string error_text;
};

exception CloseFault{
                                                    string error_text;
};

exception IncorrectIndex{
  CadUtility::LongSeq bad_indices;
  CadUtility::StringSeq message;
};

exception EntityOutOfModel{};

enum PidStatus
{
  UNMODIFIED,
  MODIFIED,
  DELETED,
  UNDEFINED
};

struct TransientIdsStatusStruct
{
  // data structures supporting EntityFactory commit output mapping

  long transient_id;
  boolean success;
  string warning;
};

struct TransientIdsEntityStruct
{
  long transient_id;
  CadFoundation::Entity valid_entity;
};
```

```
  typedef sequence<TransientIdsStatusStruct> TransientIdsStatusStructSeq;
  typedef sequence<TransientIdsEntityStruct> TransientIdsEntityStructSeq;
  typedef sequence<PidStatus>     PidStatusSeq;
  typedef sequence<ModelInstance> ModelInstanceSeq;
  typedef sequence<Model> ModelSeq;


  interface ModelInstance : CadFoundation:: Entity
  {
    CadUtility::TransformationStruct            location()            raises
(CadUtility::CadError);
    // returns location information

    Model component_model() raises (CadUtility::CadError);
    // Returns the Model that defines this instance.
    // minor change for V 1.2 - component becomes component_model


  };

  interface EntityFactory
  {
    void cleanup()  raises (CadUtility::CadError);
    //clean-up any expensive book-keeping following multiple index_xxxx(),
create() cycles

    CadUtility::LongSeq                                  index_nurbs_curves(in
CadUtility::NurbsCurveStructSeq nurbs)
         raises (CadUtility::CadError);
    CadUtility::LongSeq index_edges (in CadUtility::LongSeq start_vertices,
in CadUtility::LongSeq end_vertices,
        in CadUtility::LongSeq curve, in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_vertices(in CadUtility::PointStructSeq pts)
         raises (CadUtility::CadError);
    CadUtility::LongSeq         index_faces(in         CadUtility::LongSeqSeq
oriented_eloops,
        in   CadUtility::LongSeqSeq   vertex_loops,   in   CadUtility::LongSeq
surfaces)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq      index_edge_loops(in      CadUtility::LongSeqSeq
oriented_edges)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq     index_shells     (in     CadUtility::LongSeqSeq
oriented_faces)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq     index_bodies     (in     CadUtility::LongSeqSeq
oriented_shells)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_vertex_loops (in CadUtility::LongSeq vertices)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq  index_oriented_edges(in  CadUtility::LongSeq  edges,
in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_faces(in CadUtility::LongSeq faces,
        in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq    index_oriented_edgeloops(in    CadUtility::LongSeq
edgeloops,
        in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_shells(in CadUtility::LongSeq shells,
```

```
        in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_surfaces( in CadUtility::NurbsSurfaceStructSeq
nurbs)
         raises (CadUtility::CadError);

    void create (out TransientIdsStatusStructSeq status_flags,
        out TransientIdsEntityStructSeq final_entities)
         raises (CadUtility::CadError);
    // final creation step
  };

  interface ModelInstanceFactory
  {
    CadMain::ModelInstance              new_model_instance              (in
CadUtility::TransformationStruct               global_location)       raises
(CadUtility::CadError);
    // Creates a new ModelInstance with initial transformation according the
global coordinate system
  };

  interface  Model : CadFoundation:: Entity
  {
    //  An  aggregation  of  all  entities  and  high-level  behaviors  that
represent
    //  a  single  CAD  model.    Includes  product  structure,  boundary
representations,
    // geometric entities, features, text entities, and datums.     All
entities within
    // a CAD model are arcwise connected unless related to each other
through an instance.



    readonly attribute CadUtility::MassUnit  mass_unit;
    readonly attribute CadUtility::LengthUnit length_unit;
    // Defines units used in the Model. Angles use degrees

    CadUtility::BoundingBox  model_bounding_box  (in  CadUtility::TypeCodeSeq
entity_types)
         raises (UnboundedEntity, NotValidCadType, CadUtility::CadError);
    //  Returns  an  approximate  BoundingBox  around  all  entities  of  the
specified type(s)
    // in the model.

    CadFeature::ParameterSeq           get_parameter_set()           raises
(CadUtility::CadError);
    // Returns a sequence of parammeters for this model.



    ModelInstanceSeq model_children() raises (CadUtility::CadError);
    // Returns a sequence of any ModelInstances contained in this model.


    ModelInstanceSeq model_parents() raises (CadUtility::CadError);
    // Returns a sequence of parent models.

    CadFoundation::EntitySeq top_level_entities (
        in CadUtility::TypeCodeSeq entity_types)
        raises (NotValidCadType, CadUtility::CadError);
    // Returns a sequence of the top level entities of the specified
    // type(s).
```

```
    string file_name () raises (CadUtility::CadError);
    // Returns the complete name, including absolute path (if possible),
    // of the physical file that stores this model.
    // Returns an empty string if the model is not defined in a file.


    boolean is_embedded();
    // indicates if Model is an embedded part (e.g. CATIA Dittos,
    // ACAD blocks, etc) or a non-embedded part (e.g. parts in a ProE
assembly).


    boolean is_modified () raises (CadUtility::CadError);
    // Queries if the model has been modified since last saved.


    boolean is_read_only ()  raises (CadUtility::CadError);
    // Queries whether the model can be modified.


    boolean is_update_pending () raises (CadUtility::CadError);
    // Queries if the model is being updated (regenerated).


    CadFoundation::LayerSeq model_layers () raises (CadUtility::CadError);
    // Returns a sequence of the Layers defined in this model.

    CadFoundation::EntityGroupSeq          model_entity_groups          (out
CadUtility::StringSeq group_names)
        raises (CadUtility::CadError);
    // Returns a sequence of the EntityGroups defined in this model.


    string modification_date () raises (CadUtility::CadError);
    // Returns the date and time the model was last modified.

    string model_name () raises (CadUtility::CadError);
    // Returns the user-interpretable name of this model.


    CadUtility::StringSeq   new_persistent_ids   (in   CadUtility::StringSeq
persistent_ids)
         raises (CadFoundation::PidUnsupported, CadUtility::CadError);
    // Returns a sequence of new persistent ID's for any entities not
referenced in
    // the specified sequence of persistent ID's (e.g. new or modified IDs).


    PidStatusSeq     persistent_ids_status     (in     CadUtility::StringSeq
persistent_ids)
         raises (CadFoundation::PidUnsupported, CadUtility::CadError);
    // Returns whether the entities a sequence of persistent IDs reference
are
    // unmodified, modified, deleted.   The returned sequence of status
enumerations are in
    // the same order as the input sequence of persistent IDs.



    CadFoundation::EntitySeq persistent_ids_to_entities (
        in CadUtility::StringSeq persistent_ids)
```

```
        raises (CadFoundation::PidUnsupported, CadUtility::CadError);
    // Returns a sequence of entity objects corresponding to a sequence of
persistent IDs.
    // The returned sequence of entity objects is in the same order as the
input sequence
    // of persistent IDs.  A NULL item in this sequence means an entity was
not available
    // for the corresponding persistent ID.



    void regenerate () raises (RegenerationException, CadUtility::CadError);
    // If any DesignFeatures exist and have been modified, forces a complete
regeneration
    // of modified Entities in the model.  Otherwise this operation does
nothing.
    // Throws an exception if the regeneration process is unsuccessful.



    void    return_to_last_valid_state    ()    raises    (ReturnToValidFail,
CadUtility::CadError);
    // Returns the model and all entities it contains to their state just
after the last
    // successful regeneration or after initially opening the model.
    // Throws an exception if unable to return to a valid state.

    CadFoundation::EntitySeq unique_entities (
        in CadUtility::TypeCodeSeq entity_types)
        raises (NotValidCadType, CadUtility::CadError);
    // Returns a sequence of the unique entities of the specified
    // type(s).


    unsigned   long   unique_entities_count   (in   CadUtility::TypeCodeSeq
entity_types)
         raises (NotValidCadType, CadUtility::CadError);
    // Returns the count of entities of the specified type(s) in this model.



    CadFoundation::EntitySeq  unique_ids_to_entities  (in  CadUtility::LongSeq
unique_ids)
        raises (NotValidUid, CadUtility::CadError);
    // Returns (in sequental order) a sequence of entities corresponding to
an input sequence
    // of unique IDs.



    void    save_model() raises (SaveFault);
    void    save_model_as(in string new_name) raises (SaveAsFault);
    void    close_model() raises ( CloseFault );
    // operations for saving and terminating  an active session



    EntityFactory new_entity_factory () raises (CadUtility::CadError);
    // Entity creation factory interface - called to create new CAD entities
in current model
```

```
    ModelInstanceFactory      new_model_instance_factory      ()      raises
(CadUtility::CadError);
   //Creates the ModelInstanceFactory, which is used to add ModelInstances

    void      add_child(in      ModelInstance      child_model)      raises
(CadUtility::CadError);

    void     remove_child(in     ModelInstance     child_model)     raises
(CadUtility::CadError);

    void delete_uid_entity(in long uid)
        raises (EntityOutOfModel, CadUtility::CadError);
    // Removes ModelInstance, BrepEntity, Curve or Surface from the model

    void delete_entity(in CadUtility::LongSeq entities_uids)
        raises (EntityOutOfModel, CadUtility::CadError);
    // Removes model entities from the model
    // v1.2 change from EntitySeq to LongSeq - simplifies mapping


  };
};
#endif
```

File: CadUtility.idl

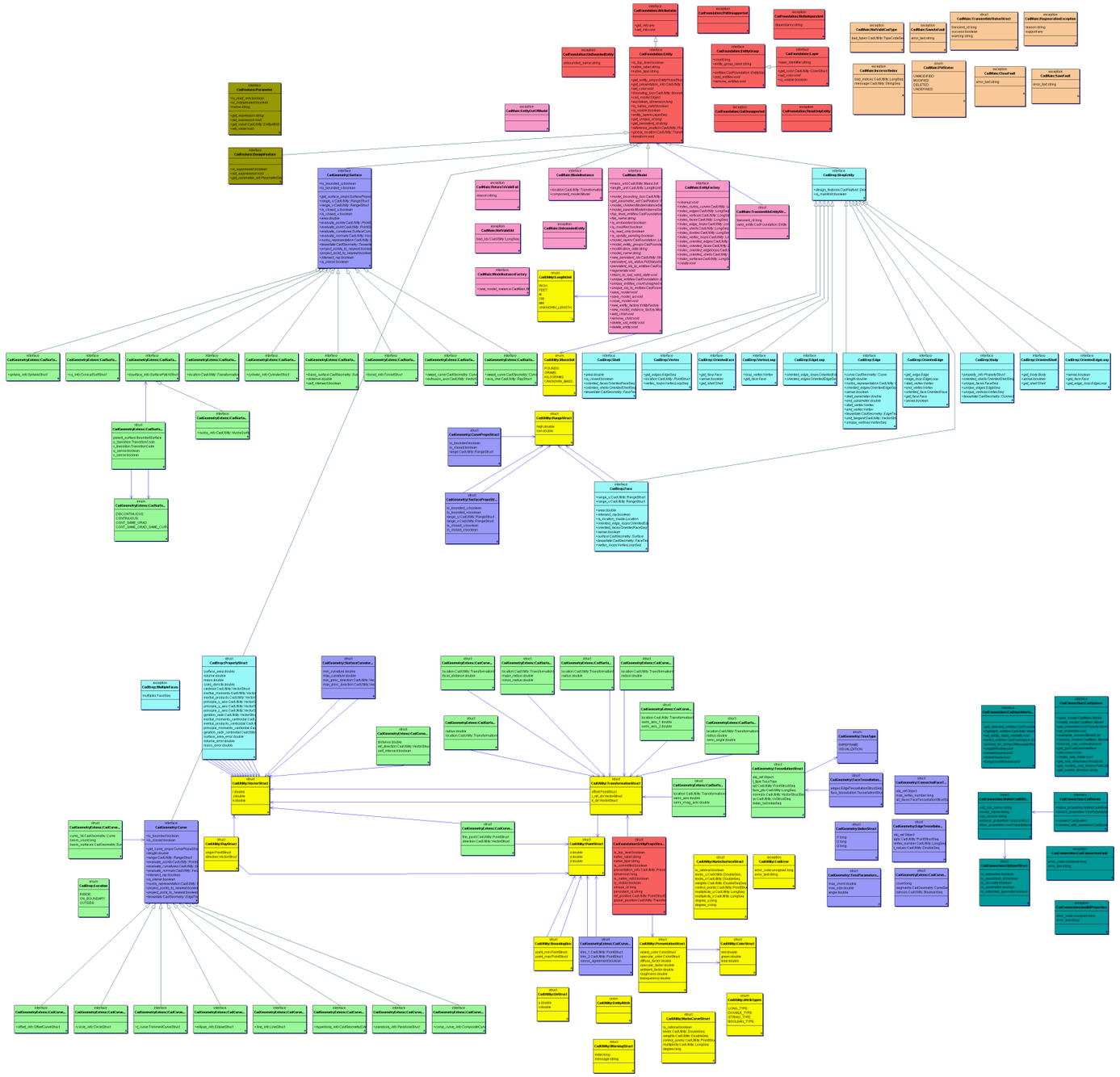
```
//File: CadUtility.idl
//CAD Services V1.1

#ifndef __CADUTILITY_DEFINED
#define __CADUTILITY_DEFINED

#include <orb.idl>

#pragma prefix "omg.org"

module CadUtility
{

  // basic geometric structures

  struct PointStruct
  {
    // three dimensional location

    double x;
    double y;
    double z;
  };

  struct BoundingBox
  {
    PointStruct   point_min;
    PointStruct   point_max;
  };
  typedef sequence<PointStruct>     PointStructSeq;
  typedef sequence<PointStructSeq>  PointStructSeqSeq;

    struct VectorStruct
  {
    // Direction in 3D

    double i;
```

```
    double j;
    double k;
  };

  typedef sequence<VectorStruct> VectorStructSeq;
  typedef sequence<VectorStructSeq> VectorStructSeqSeq;
  typedef sequence<VectorStructSeqSeq> VectorStructSeqSeqSeq;

  struct TransformationStruct
  {
    PointStruct   offset;
    VectorStruct    i_ref_dir;
    VectorStruct    k_dir;
  };

 struct RayStruct
  {
    PointStruct     origin;
    VectorStruct    direction;
  };
  typedef sequence<RayStruct> RayStructSeq;

  struct UvStruct
  {
    double u;
    double v;
  };
  typedef sequence<UvStruct> UvStructSeq;

  // Global Cad Error

  exception CadError
  {
    unsigned long error_code;
    string error_text;
  };

  // Sequences of basic types

  typedef sequence<boolean> BooleanSeq;
  typedef sequence<long>    LongSeq;
  typedef sequence<LongSeq> LongSeqSeq;
  typedef sequence<double>  DoubleSeq;
  typedef sequence<string>  StringSeq;
  typedef sequence<DoubleSeq> DoubleSeqSeq;
  typedef sequence<any>     AnySeq;
  typedef sequence<CORBA::TypeCode> TypeCodeSeq;

  enum MassUnit
  {
    // mass unit options

    POUNDS,
    GRAMS,
    KILOGRAMS,
    UNKNOWN_MASS
   };

  enum LengthUnit
  {
    // length unit options

    INCH,
```

```
  FEET,
  M,
  CM,
  MM,
  UNKNOWN_LENGTH
};

// enum + union supporting parameters

enum AttribTypes
{
  LONG_TYPE,
  DOUBLE_TYPE,
  STRING_TYPE,
  BOOLEAN_TYPE
};

union EntityAttrib switch(AttribTypes)
{
  case LONG_TYPE: long     l_value;
  case DOUBLE_TYPE: double d_value;
  case STRING_TYPE: string s_value;
  case BOOLEAN_TYPE: boolean b_value;
};

struct ColorStruct
{
  // basic color information in RGB form
  // Valid values range from 0.0 to 1.0

  double red;
  double green;
  double blue;
};
typedef sequence<ColorStruct> ColorStructSeq;

struct PresentationStruct
{
  // CAD system presentation data
  // Unsupported features will return a negative value
  // Valid values range from 0.0 to 1.0

  ColorStruct   object_color;
  ColorStruct   specular_color; //light source color
  double diffuse_factor;
  double specular_factor;
  double ambient_factor;
  double roughness;
  double transparency; //  0. is opaque and 1. is transparent
};

struct RangeStruct
{
  // basic range information

  double high;
  double low;
};

struct WarningStruct
{
  // struct for warning messages
```

```
    long   index;
    string message;
  };
  typedef sequence<WarningStruct>WarningStructSeq;

  // Nurbs data structures

  struct NurbsCurveStruct
  {

                                                        boolean is_rational;
                                                  // rational or polynomial?



                                                   CadUtility::DoubleSeq knots;
                                                  // A sequence of knot values.



                                                 CadUtility::DoubleSeq weights;
                                                // A sequence of weight values.



                                      CadUtility::PointStructSeq control_points;
                                           // A sequence of control points in 3D

                                              CadUtility::LongSeq multiplicity;
                                                            long        degree;
  };
  typedef sequence<NurbsCurveStruct> NurbsCurveStructSeq;

  struct NurbsSurfaceStruct
  {
    boolean is_rational;
    // rational or polynomial?



    CadUtility::DoubleSeq knots_u;
    CadUtility::DoubleSeq knots_v;
    // Sequence of knot values.



    CadUtility::DoubleSeqSeq weights;
    // A sequence of weight values.



    CadUtility::PointStructSeqSeq control_points;
    // A sequence of control points.
    // Each point is a sequence of a sequence of 3D points.

    CadUtility::LongSeq multiplicity_u;
    CadUtility::LongSeq multiplicity_v;
    long   degree_u;
    long   degree_v;
  };
  typedef sequence<NurbsSurfaceStruct> NurbsSurfaceStructSeq;
};
#endif
```

UML DIAGRAMME CADSERVICES



Anhang E

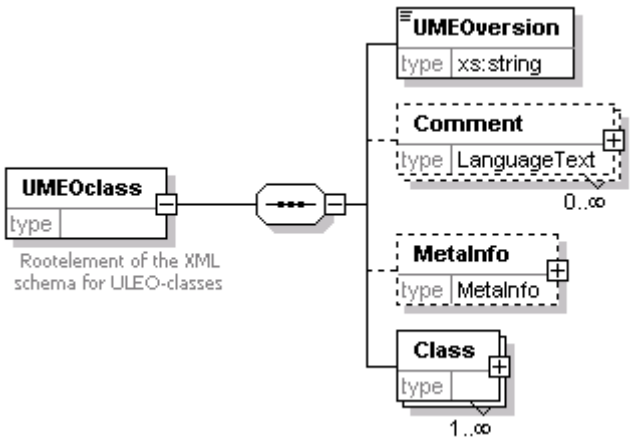
UMEO SCHEMA DEFINITIONEN[1]

UMEOclass Schema

schema location: **D:\Martin\UMEOclassV1c.xsd**

Elements	Complex types	Simple types
UMEOclass	**ComplexDefault**	**AccessState**
	ComplexInfo	**ClassType**
	ComplexLink	**Direction**
	ComplexParameter	**Identifier**
	Identity	**Language**
	InstComplexParam	
	LanguageText	
	MetaInfo	
	Role	
	SimpleParameter	
	TaggedValue	
	Vector	

element **UMEOclass**

diagram	
children	**UMEOversion** **Comment** **MetaInfo** **Class**
annotation	documentation Rootelement of the XML schema for ULEO-classes
source	<xs:element name="UMEOclass"> <xs:annotation> <xs:documentation> Rootelement of the XML schema for ULEO-classes</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="UMEOversion" type="xs:string"/> <xs:element name="Comment" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>

[1] Veröffentlichung der UMEO Klassendiagramme mit freundlicher Genehmigung von Dr. Johann Ulrich Zimmermann, DaimlerChrysler Research & Technology, Laboratory of IT for Engineering and Processes (REI/I), Department of Product and Production Modeling (REI/IP) vom 17.11.2005

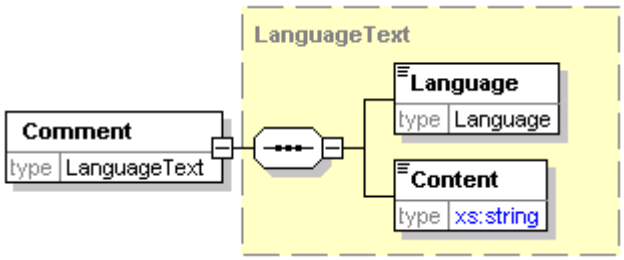
```
    <xs:element name="Class" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ClassType" type="ClassType"/>
       <xs:element name="MTRTclassID" type="Identifier" minOccurs="0">
        <xs:annotation>
         <xs:documentation>only and obligatory for EORMclass</xs:documentation>
        </xs:annotation>
       </xs:element>
       <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element name="Identity" type="Identity"/>
       <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="SimpleParameter" type="SimpleParameter"/>
        <xs:element name="ComplexParameter" type="ComplexParameter"/>
       </xs:choice>
       <xs:element name="Method" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Identity" type="Identity"/>
          <xs:element name="ScriptLanguage" minOccurs="0"/>
          <xs:element name="ReturnType" minOccurs="0"/>
          <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
           <xs:element name="SimpleParameter" type="SimpleParameter"/>
           <xs:element name="ComplexParameter" type="ComplexParameter"/>
          </xs:choice>
          <xs:element name="FunctionBody"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="ClassConstraint" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element name="ClassMetainfo" type="MetaInfo" minOccurs="0"/>
       <xs:element name="InstMetainfo" minOccurs="0">
        <xs:complexType>
         <xs:choice maxOccurs="unbounded">
          <xs:element name="SimpleInfo" type="SimpleParameter"/>
          <xs:element name="ComplexInfo" type="ComplexParameter"/>
         </xs:choice>
        </xs:complexType>
       </xs:element>
       <xs:element name="Partner" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
         <xs:documentation>only and obligatory for EORMclass</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence>
          <xs:element name="PartnerClassType" type="ClassType">
           <xs:annotation>
            <xs:documentation>only classes with classtype (Class.ClassType) EOclass, MTRTclass or
EORMclass</xs:documentation>
           </xs:annotation>
          </xs:element>
          <xs:element name="PartnerClassID" type="Identifier">
           <xs:annotation>
            <xs:documentation>references to the full ID (Class.Identity.FullID) of the
partnerclass</xs:documentation>
           </xs:annotation>
          </xs:element>
          <xs:element name="Direction" type="Direction"/>
          <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="PartnerRole" type="Role" maxOccurs="unbounded"/>
          <xs:element name="EORMrole" type="Role" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="ExternalLink" minOccurs="0">
        <xs:annotation>
         <xs:documentation>only for EOclass</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:choice maxOccurs="unbounded">
          <xs:element name="SimpleLink" type="TaggedValue"/>
          <xs:element name="ComplexLink" type="ComplexLink"/>
         </xs:choice>
        </xs:complexType>
       </xs:element>
```

```
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

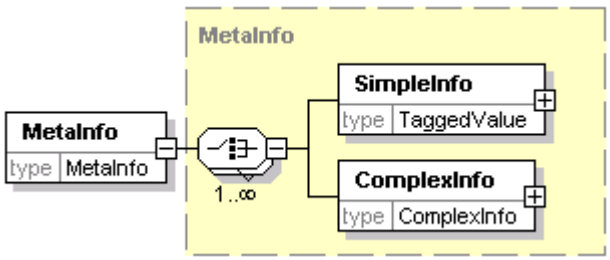
element **UMEOclass/UMEOversion**

diagram	
type	**xs:string**
source	<xs:element name="UMEOversion" type="xs:string"/>

element **UMEOclass/Comment**

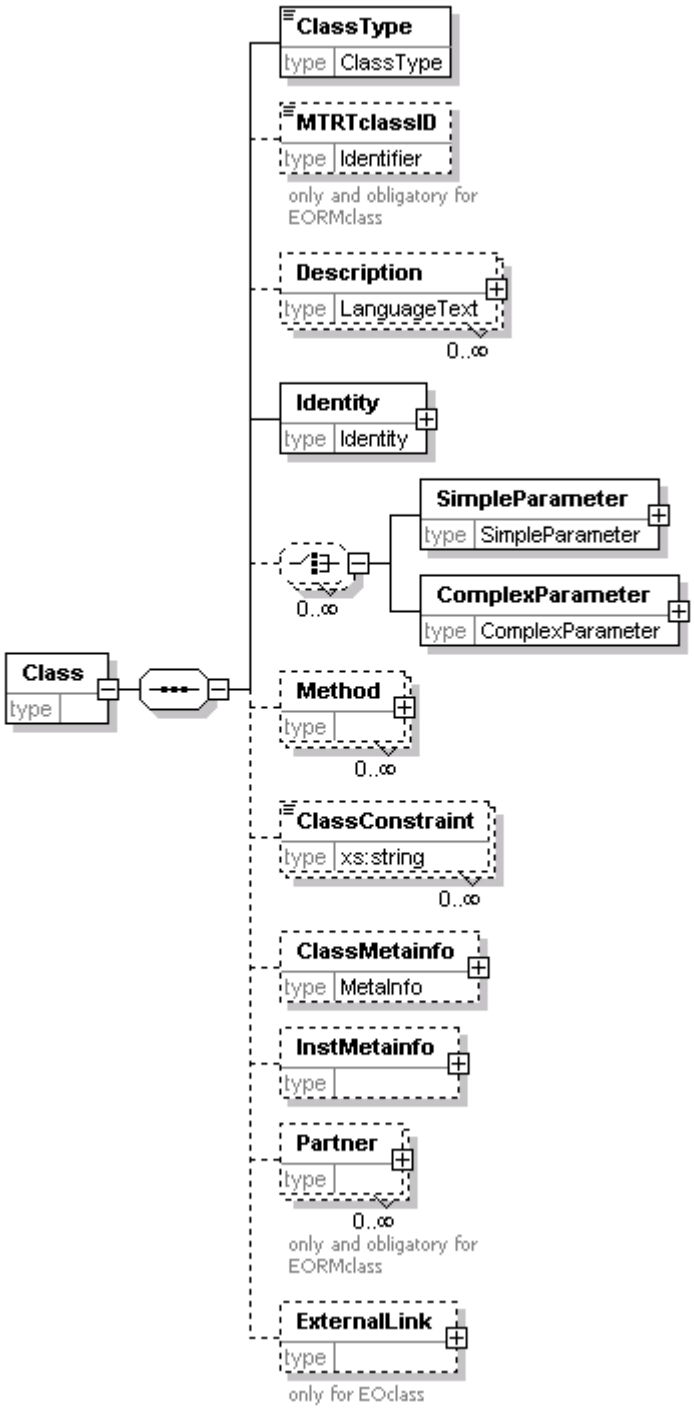
diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Comment" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

element **UMEOclass/MetaInfo**

diagram	
type	**MetaInfo**
children	**SimpleInfo** **ComplexInfo**
source	<xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>

element **UMEOclass/Class**

diagram



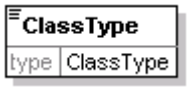
children **ClassType MTRTclassID Description Identity SimpleParameter ComplexParameter Method ClassConstraint ClassMetainfo InstMetainfo Partner ExternalLink**

source

```
<xs:element name="Class" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="ClassType" type="ClassType"/>
   <xs:element name="MTRTclassID" type="Identifier" minOccurs="0">
    <xs:annotation>
     <xs:documentation>only and obligatory for EORMclass</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="Identity" type="Identity"/>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="SimpleParameter" type="SimpleParameter"/>
    <xs:element name="ComplexParameter" type="ComplexParameter"/>
```

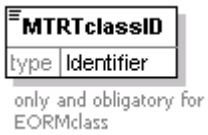
```
    </xs:choice>
    <xs:element name="Method" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Identity" type="Identity"/>
       <xs:element name="ScriptLanguage" minOccurs="0"/>
       <xs:element name="ReturnType" minOccurs="0"/>
       <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
       <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="SimpleParameter" type="SimpleParameter"/>
        <xs:element name="ComplexParameter" type="ComplexParameter"/>
       </xs:choice>
       <xs:element name="FunctionBody"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="ClassConstraint" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="ClassMetainfo" type="MetaInfo" minOccurs="0"/>
    <xs:element name="InstMetainfo" minOccurs="0">
     <xs:complexType>
      <xs:choice maxOccurs="unbounded">
       <xs:element name="SimpleInfo" type="SimpleParameter"/>
       <xs:element name="ComplexInfo" type="ComplexParameter"/>
      </xs:choice>
     </xs:complexType>
    </xs:element>
    <xs:element name="Partner" minOccurs="0" maxOccurs="unbounded">
     <xs:annotation>
      <xs:documentation>only and obligatory for EORMclass</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:sequence>
       <xs:element name="PartnerClassType" type="ClassType">
        <xs:annotation>
         <xs:documentation>only classes with classtype (Class.ClassType) EOclass, MTRTclass or
EORMclass</xs:documentation>
        </xs:annotation>
       </xs:element>
       <xs:element name="PartnerClassID" type="Identifier">
        <xs:annotation>
         <xs:documentation>references to the full ID (Class.Identity.FullID) of the
partnerclass</xs:documentation>
        </xs:annotation>
       </xs:element>
       <xs:element name="Direction" type="Direction"/>
       <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element name="PartnerRole" type="Role" maxOccurs="unbounded"/>
       <xs:element name="EORMrole" type="Role" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="ExternalLink" minOccurs="0">
     <xs:annotation>
      <xs:documentation>only for EOclass</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:choice maxOccurs="unbounded">
       <xs:element name="SimpleLink" type="TaggedValue"/>
       <xs:element name="ComplexLink" type="ComplexLink"/>
      </xs:choice>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
</xs:element>
```

element **UMEOclass/Class/ClassType**

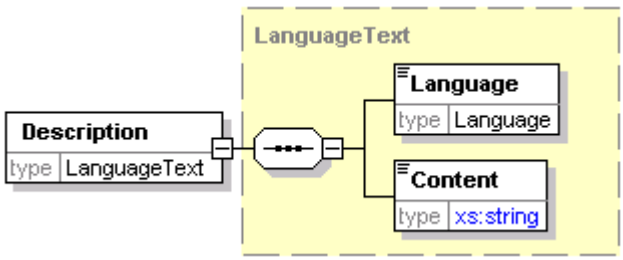
diagram	
type	**ClassType**

facets	enumeration EOclass enumeration MTRTclass enumeration EORMclass enumeration ComplexDataType enumeration MTRTrelation
source	<xs:element name="ClassType" type="ClassType"/>

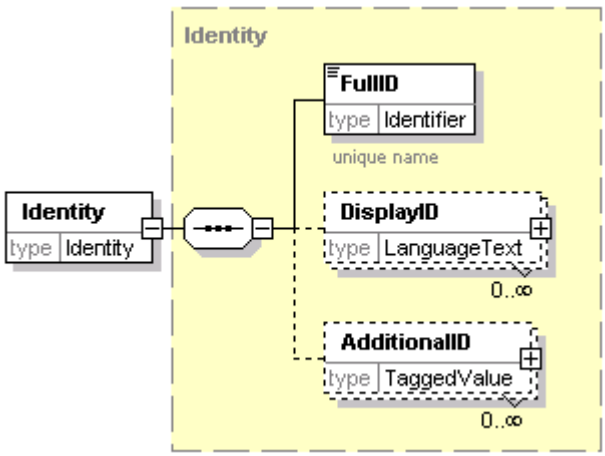
element **UMEOclass/Class/MTRTclassID**

diagram	
type	**Identifier**
annotation	documentation only and obligatory for EORMclass
source	<xs:element name="MTRTclassID" type="Identifier" minOccurs="0"> <xs:annotation> <xs:documentation>only and obligatory for EORMclass</xs:documentation> </xs:annotation> </xs:element>

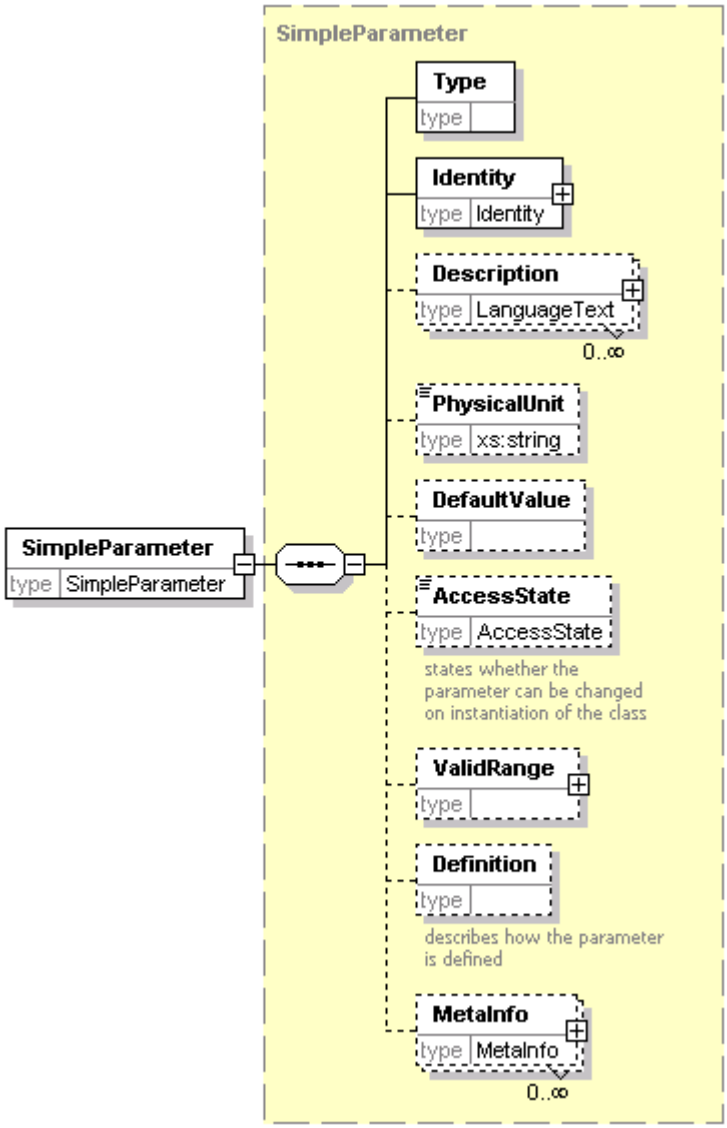
element **UMEOclass/Class/Description**

diagram	
type	**LanguageText**
children	**Language Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

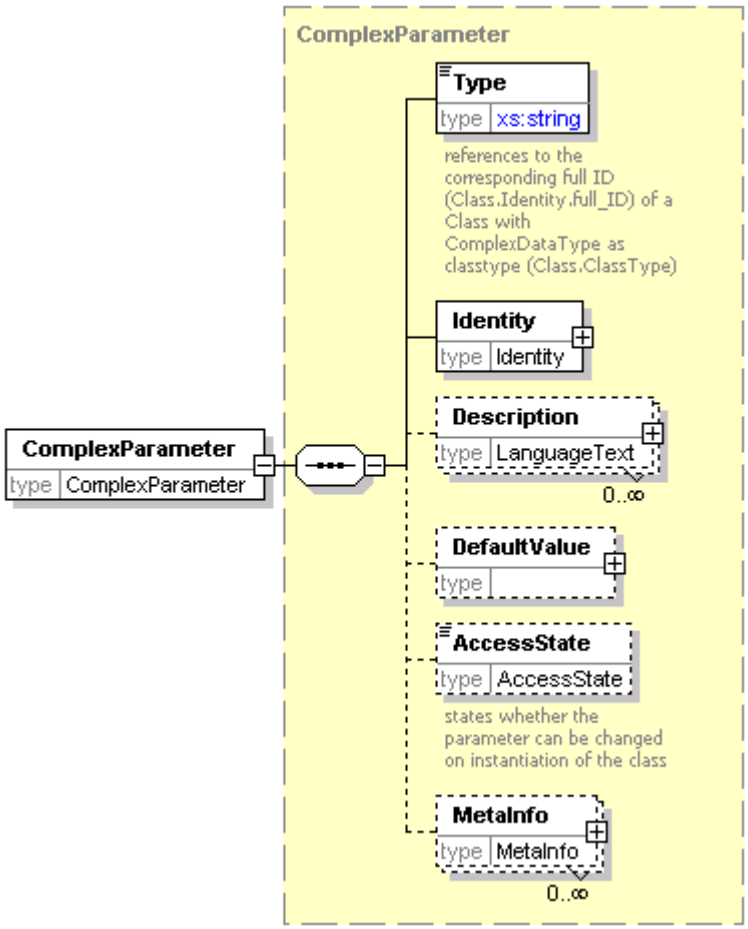
element **UMEOclass/Class/Identity**

diagram	
type	**Identity**
children	**FullID DisplayID AdditionalID**
source	<xs:element name="Identity" type="Identity"/>

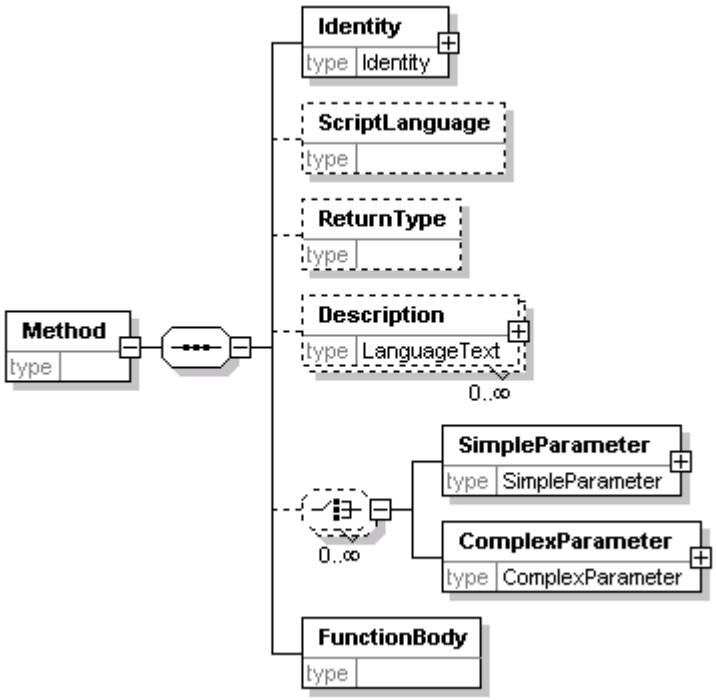
element **UMEOclass/Class/SimpleParameter**

diagram	
type	**SimpleParameter**
children	**Type Identity Description PhysicalUnit DefaultValue AccessState ValidRange Definition MetaInfo**
source	<xs:element name="SimpleParameter" type="SimpleParameter"/>

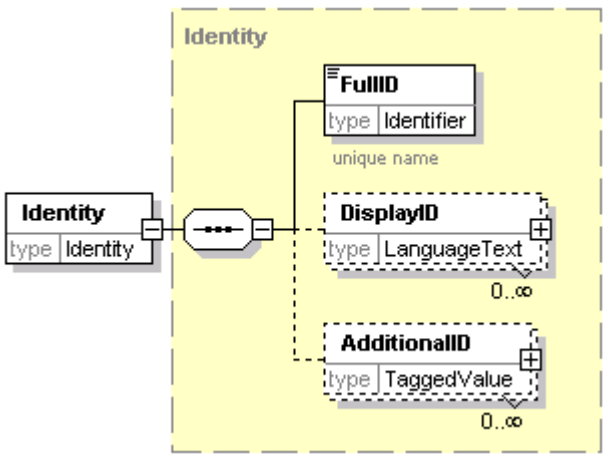
element **UMEOclass/Class/ComplexParameter**

diagram	
type	**ComplexParameter**
children	**Type Identity Description DefaultValue AccessState MetaInfo**
source	<xs:element name="ComplexParameter" type="ComplexParameter"/>

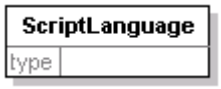
element **UMEOclass/Class/Method**

diagram	
children	**Identity ScriptLanguage ReturnType Description SimpleParameter ComplexParameter FunctionBody**
source	<xs:element name="Method" minOccurs="0" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="Identity" type="Identity"/> <xs:element name="ScriptLanguage" minOccurs="0"/> <xs:element name="ReturnType" minOccurs="0"/> <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> <xs:choice minOccurs="0" maxOccurs="unbounded"> <xs:element name="SimpleParameter" type="SimpleParameter"/> <xs:element name="ComplexParameter" type="ComplexParameter"/> </xs:choice> <xs:element name="FunctionBody"/> </xs:sequence> </xs:complexType> </xs:element>

element **UMEOclass/Class/Method/Identity**

diagram	
type	**Identity**
children	**FullID DisplayID AdditionalID**
source	<xs:element name="Identity" type="Identity"/>

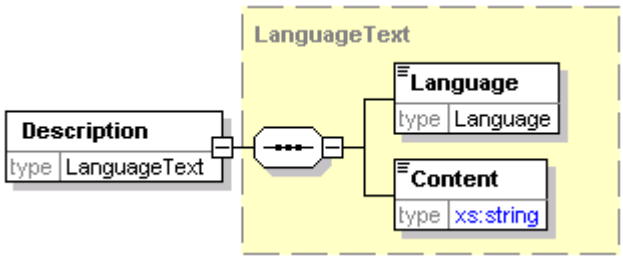
element **UMEOclass/Class/Method/ScriptLanguage**

diagram	
source	<xs:element name="ScriptLanguage" minOccurs="0"/>

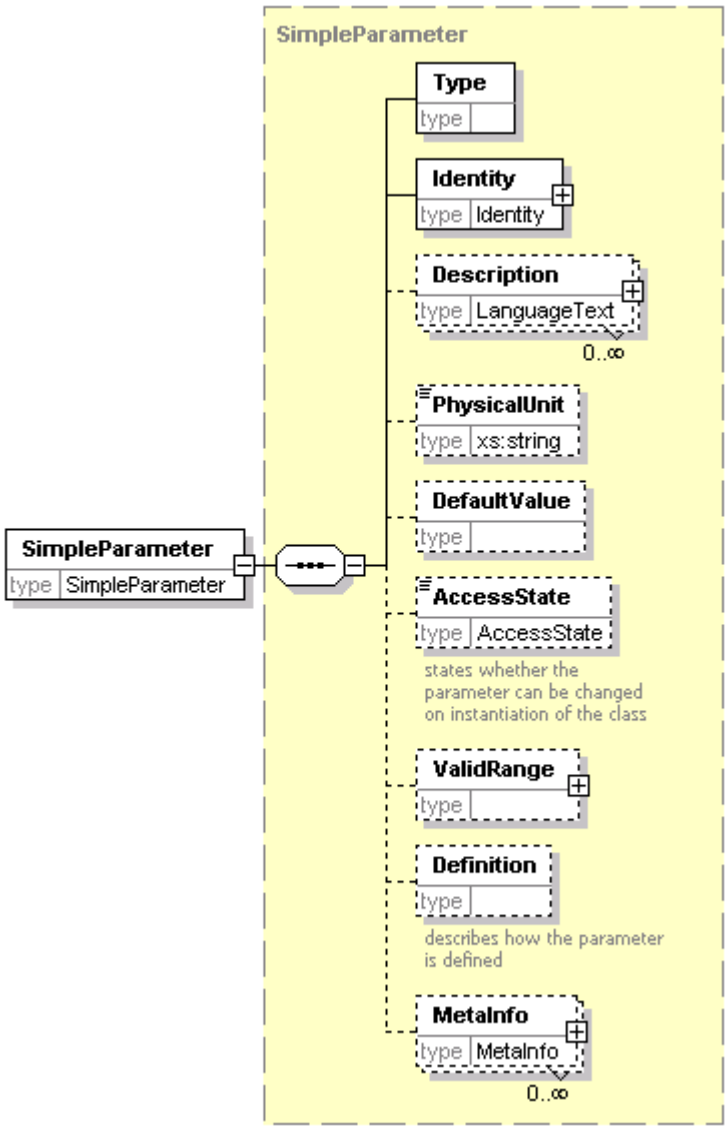
element **UMEOclass/Class/Method/ReturnType**

diagram	
source	<xs:element name="ReturnType" minOccurs="0"/>

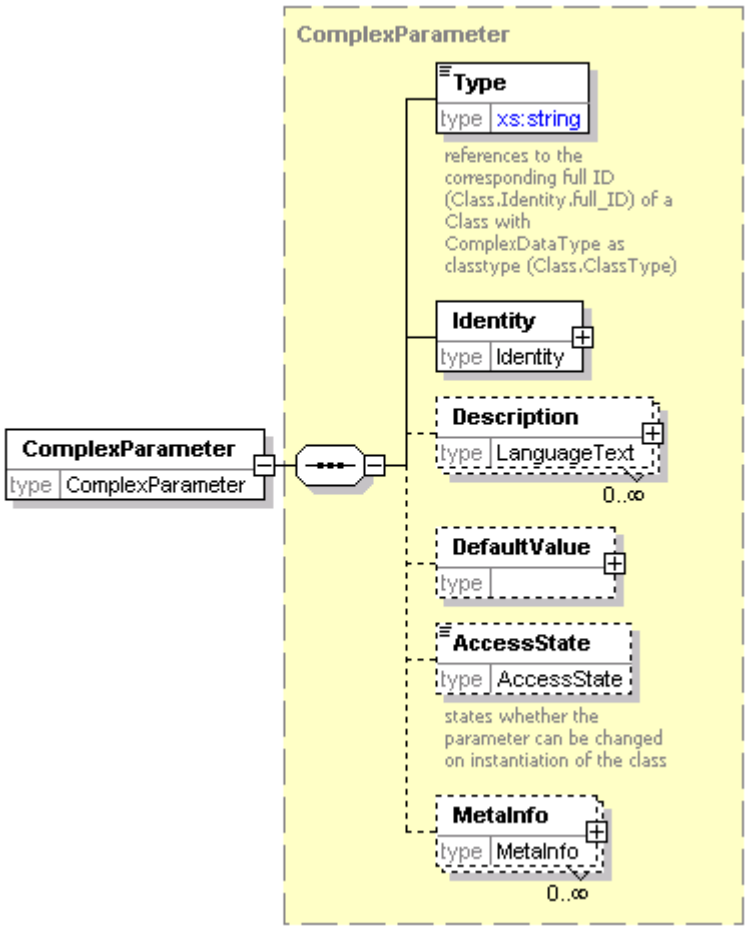
element **UMEOclass/Class/Method/Description**

diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

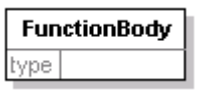
element **UMEOclass/Class/Method/SimpleParameter**

diagram	
type	**SimpleParameter**
children	**Type Identity Description PhysicalUnit DefaultValue AccessState ValidRange Definition MetaInfo**
source	<xs:element name="SimpleParameter" type="SimpleParameter"/>

element **UMEOclass/Class/Method/ComplexParameter**

diagram	
type	**ComplexParameter**
children	**Type Identity Description DefaultValue AccessState MetaInfo**
source	<xs:element name="ComplexParameter" type="ComplexParameter"/>

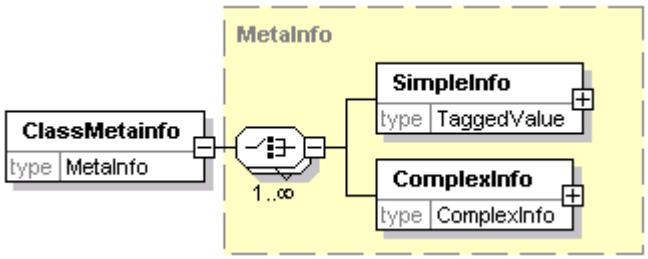
element **UMEOclass/Class/Method/FunctionBody**

diagram	
source	<xs:element name="FunctionBody"/>

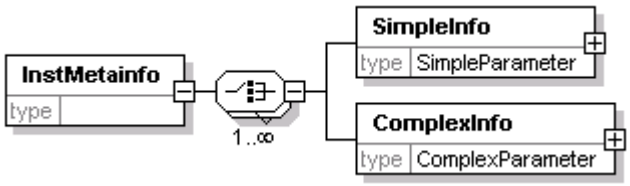
element **UMEOclass/Class/ClassConstraint**

diagram	
type	**xs:string**
source	<xs:element name="ClassConstraint" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>

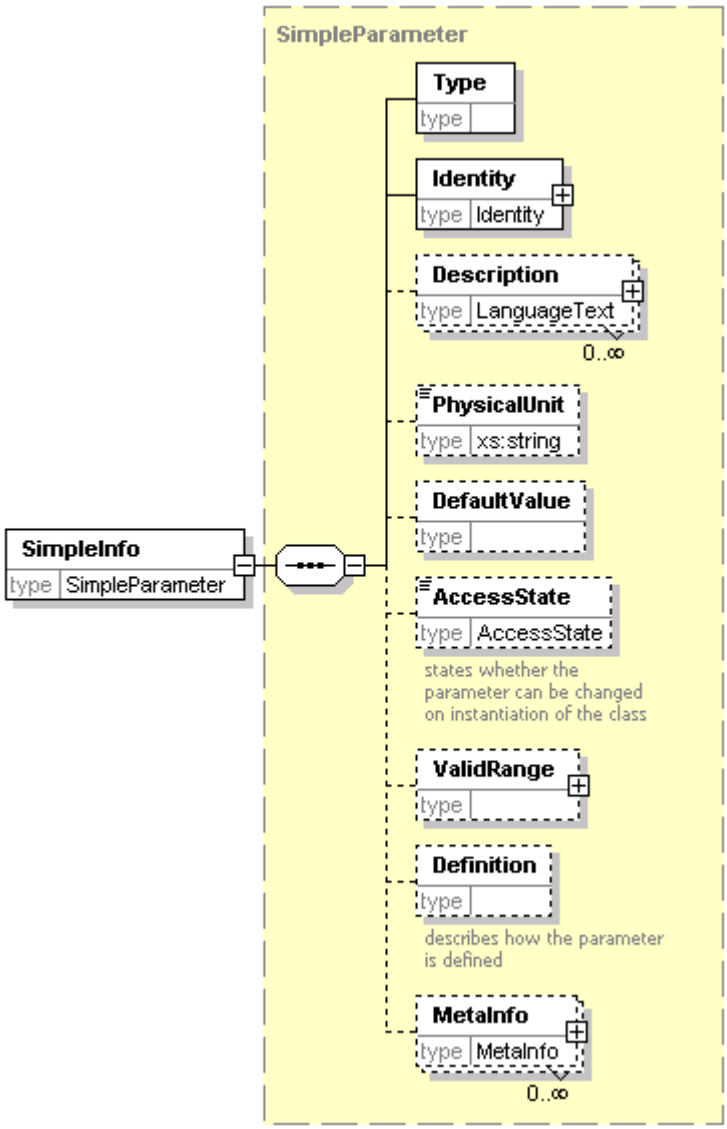
element **UMEOclass/Class/ClassMetainfo**

diagram	
type	**MetaInfo**
children	**SimpleInfo ComplexInfo**
source	<xs:element name="ClassMetainfo" type="MetaInfo" minOccurs="0"/>

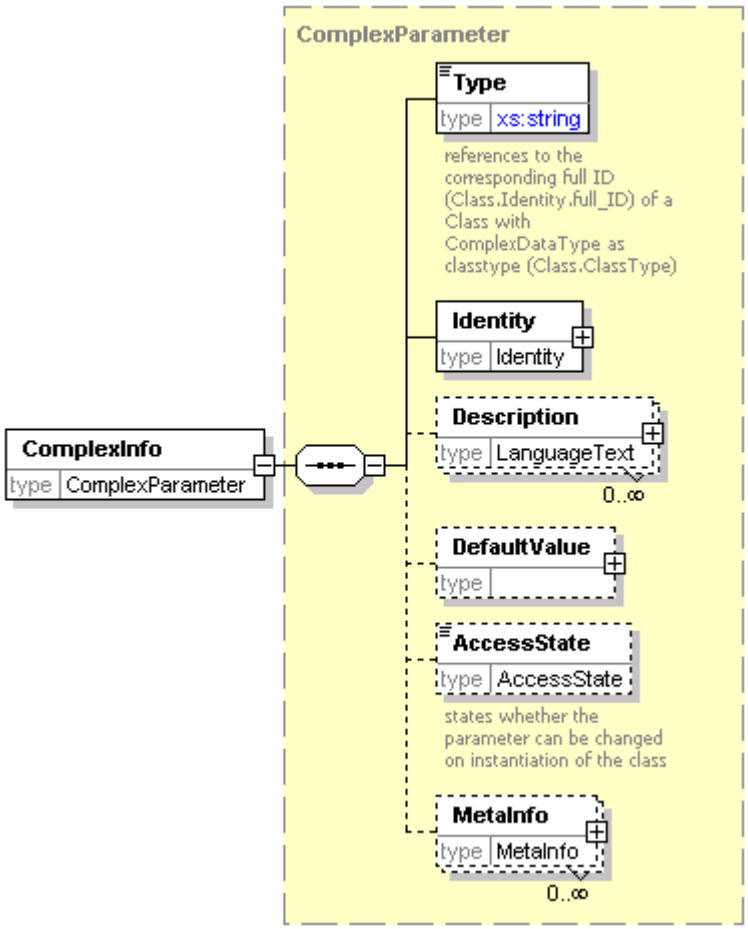
element **UMEOclass/Class/InstMetainfo**

diagram	
children	**SimpleInfo ComplexInfo**
source	<xs:element name="InstMetainfo" minOccurs="0"> <xs:complexType> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleInfo" type="SimpleParameter"/> <xs:element name="ComplexInfo" type="ComplexParameter"/> </xs:choice> </xs:complexType> </xs:element>

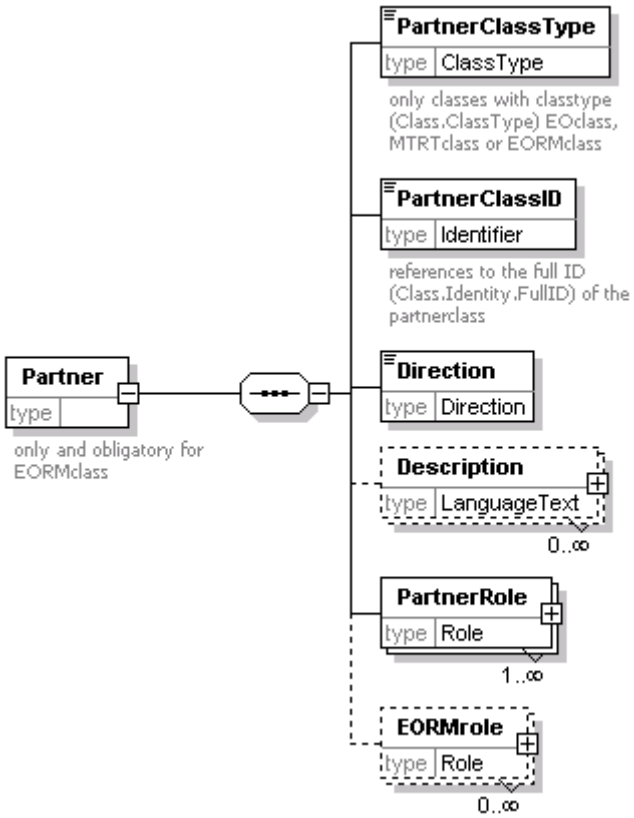
element **UMEOclass/Class/InstMetainfo/SimpleInfo**

diagram	
type	**SimpleParameter**
children	**Type Identity Description PhysicalUnit DefaultValue AccessState ValidRange Definition MetaInfo**
source	<xs:element name="SimpleInfo" type="SimpleParameter"/>

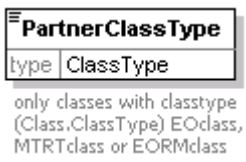
element **UMEOclass/Class/InstMetainfo/ComplexInfo**

diagram	
type	**ComplexParameter**
children	**Type Identity Description DefaultValue AccessState MetaInfo**
source	<xs:element name="ComplexInfo" type="ComplexParameter"/>

element **UMEOclass/Class/Partner**

diagram	
children	**PartnerClassType PartnerClassID Direction Description PartnerRole EORMrole**
annotation	documentation only and obligatory for EORMclass
source	<xs:element name="Partner" minOccurs="0" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>only and obligatory for EORMclass</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="PartnerClassType" type="ClassType"> <xs:annotation> <xs:documentation>only classes with classtype (Class.ClassType) EOclass, MTRTclass or EORMclass</xs:documentation> </xs:annotation> </xs:element> <xs:element name="PartnerClassID" type="Identifier"> <xs:annotation> <xs:documentation>references to the full ID (Class.Identity.FullID) of the partnerclass</xs:documentation> </xs:annotation> </xs:element> <xs:element name="Direction" type="Direction"/> <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="PartnerRole" type="Role" maxOccurs="unbounded"/> <xs:element name="EORMrole" type="Role" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element>

element **UMEOclass/Class/Partner/PartnerClassType**

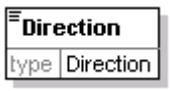
diagram	
type	**ClassType**

facets	enumeration EOclass enumeration MTRTclass enumeration EORMclass enumeration ComplexDataType enumeration MTRTrelation
annotation	documentation only classes with classtype (Class.ClassType) EOclass, MTRTclass or EORMclass
source	<xs:element name="PartnerClassType" type="ClassType"> <xs:annotation> <xs:documentation>only classes with classtype (Class.ClassType) EOclass, MTRTclass or EORMclass</xs:documentation> </xs:annotation> </xs:element>

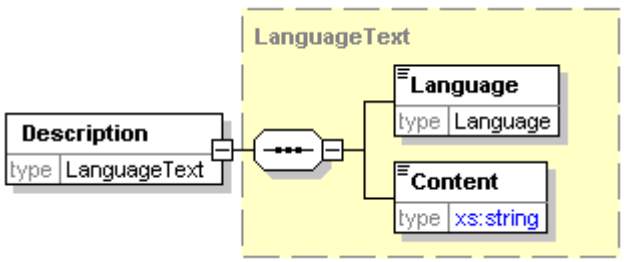
element **UMEOclass/Class/Partner/PartnerClassID**

diagram	
type	**Identifier**
annotation	documentation references to the full ID (Class.Identity.FullID) of the partnerclass
source	<xs:element name="PartnerClassID" type="Identifier"> <xs:annotation> <xs:documentation>references to the full ID (Class.Identity.FullID) of the partnerclass</xs:documentation> </xs:annotation> </xs:element>

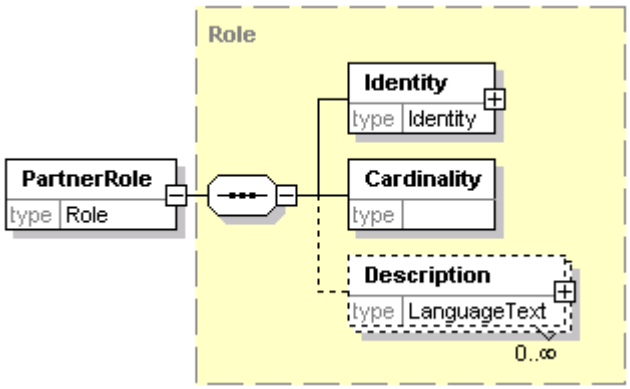
element **UMEOclass/Class/Partner/Direction**

diagram	
type	**Direction**
facets	enumeration in enumeration out enumeration undirected
source	<xs:element name="Direction" type="Direction"/>

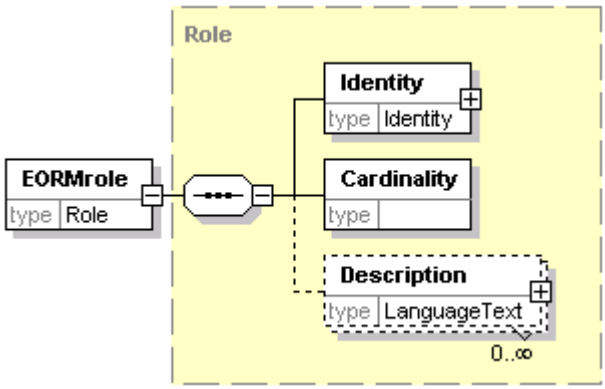
element **UMEOclass/Class/Partner/Description**

diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

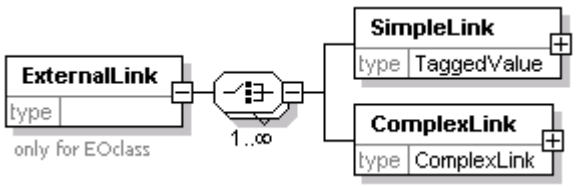
element **UMEOclass/Class/Partner/PartnerRole**

diagram	
type	**Role**
children	**Identity Cardinality Description**
source	<xs:element name="PartnerRole" type="Role" maxOccurs="unbounded"/>

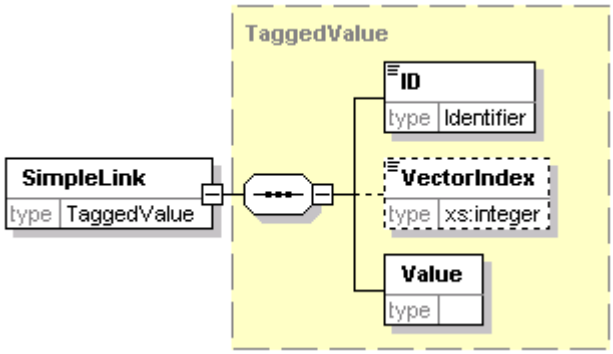
element **UMEOclass/Class/Partner/EORMrole**

diagram	
type	**Role**
children	**Identity Cardinality Description**
source	<xs:element name="EORMrole" type="Role" minOccurs="0" maxOccurs="unbounded"/>

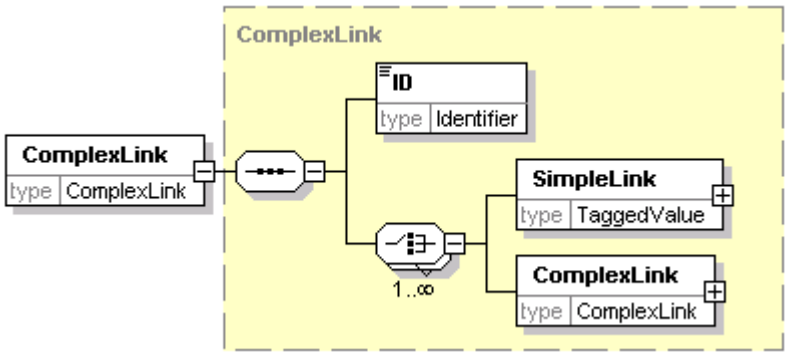
element **UMEOclass/Class/ExternalLink**

diagram	
children	**SimpleLink ComplexLink**
annotation	documentation only for EOclass
source	<xs:element name="ExternalLink" minOccurs="0"> <xs:annotation> <xs:documentation>only for EOclass</xs:documentation> </xs:annotation> <xs:complexType> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleLink" type="TaggedValue"/> <xs:element name="ComplexLink" type="ComplexLink"/> </xs:choice> </xs:complexType> </xs:element>

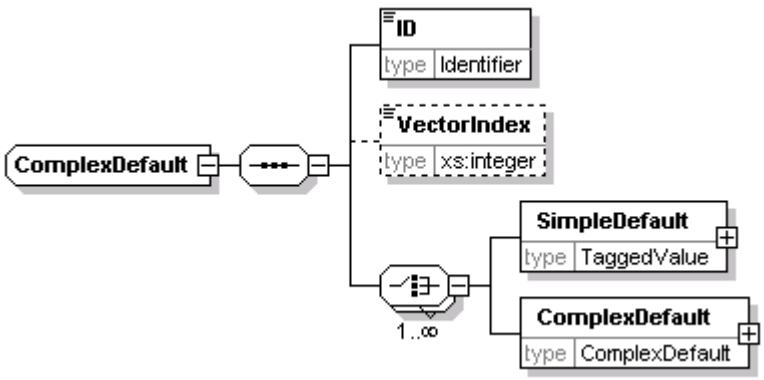
element **UMEOclass/Class/ExternalLink/SimpleLink**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value**
source	<xs:element name="SimpleLink" type="TaggedValue"/>

element **UMEOclass/Class/ExternalLink/ComplexLink**

diagram	
type	**ComplexLink**
children	**ID** **SimpleLink** **ComplexLink**
source	<xs:element name="ComplexLink" type="ComplexLink"/>

complexType **ComplexDefault**

diagram	
children	**ID** **VectorIndex** **SimpleDefault** **ComplexDefault**
used by	elements **ComplexParameter/DefaultValue/ComplexDefault** **ComplexDefault/ComplexDefault**
source	<xs:complexType name="ComplexDefault"> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:element name="VectorIndex" type="xs:integer" minOccurs="0"/>

```
    <xs:choice maxOccurs="unbounded">
      <xs:element name="SimpleDefault" type="TaggedValue"/>
      <xs:element name="ComplexDefault" type="ComplexDefault"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

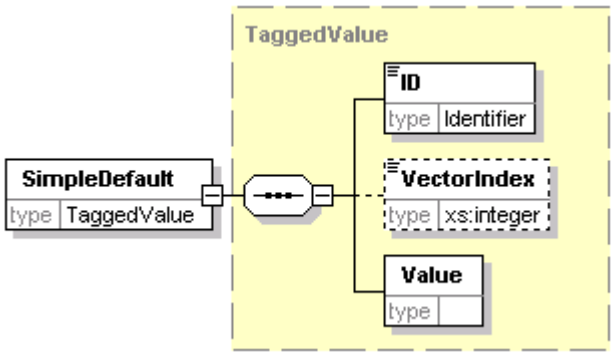
element **ComplexDefault/ID**

diagram	
type	**Identifier**
source	`<xs:element name="ID" type="Identifier"/>`

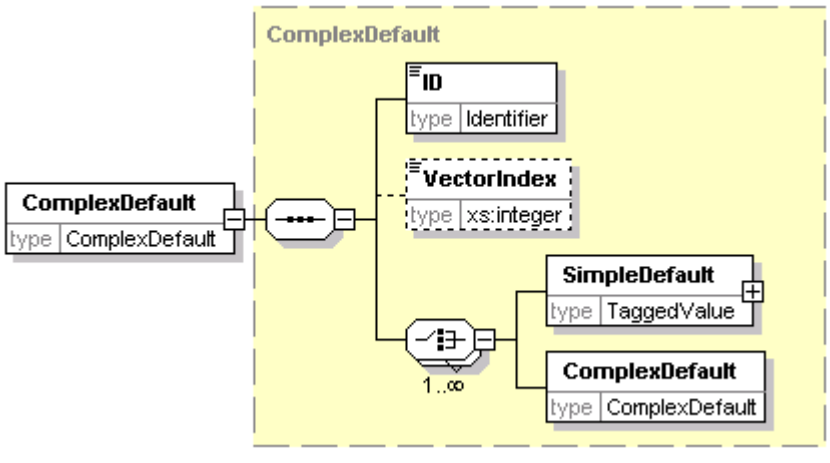
element **ComplexDefault/VectorIndex**

diagram	
type	**xs:integer**
source	`<xs:element name="VectorIndex" type="xs:integer" minOccurs="0"/>`

element **ComplexDefault/SimpleDefault**

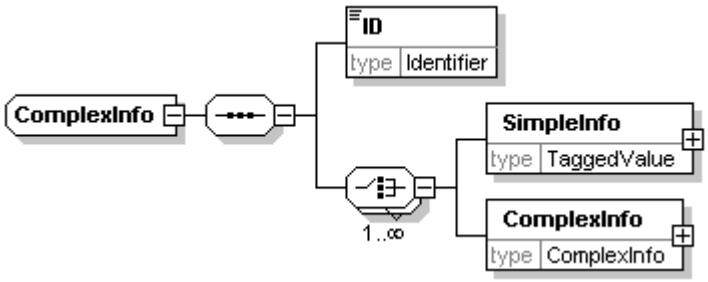
diagram	
type	**TaggedValue**
children	**ID VectorIndex Value**
source	`<xs:element name="SimpleDefault" type="TaggedValue"/>`

element **ComplexDefault/ComplexDefault**

diagram	

type	**ComplexDefault**
children	**ID VectorIndex SimpleDefault ComplexDefault**
source	<xs:element name="ComplexDefault" type="ComplexDefault"/>

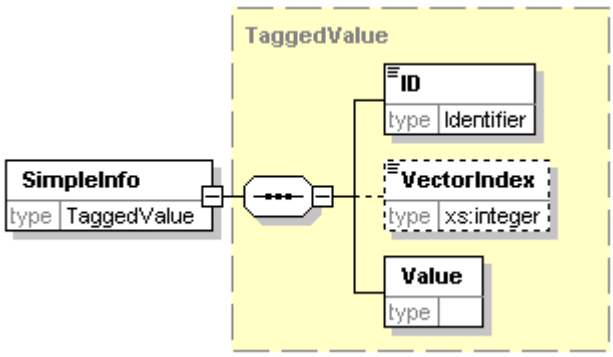
complexType **ComplexInfo**

diagram	
children	**ID SimpleInfo ComplexInfo**
used by	elements **MetaInfo/ComplexInfo ComplexInfo/ComplexInfo**
source	<xs:complexType name="ComplexInfo"> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleInfo" type="TaggedValue"/> <xs:element name="ComplexInfo" type="ComplexInfo"/> </xs:choice> </xs:sequence> </xs:complexType>

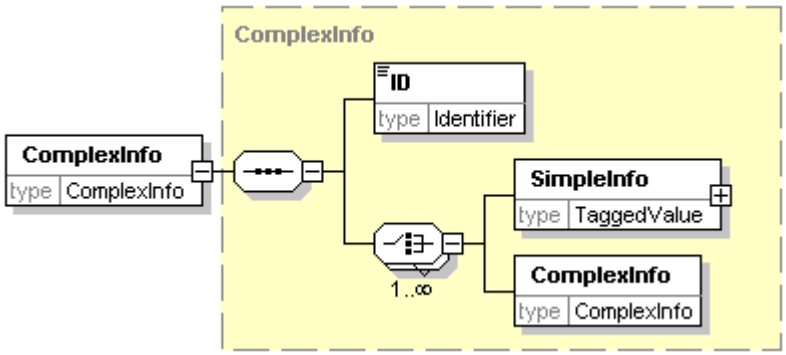
element **ComplexInfo/ID**

diagram	
type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

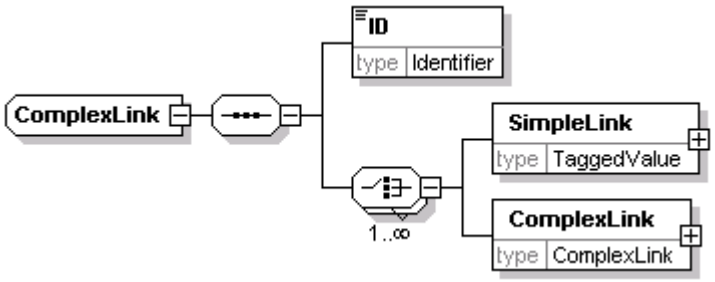
element **ComplexInfo/SimpleInfo**

diagram	
type	**TaggedValue**
children	**ID VectorIndex Value**
source	<xs:element name="SimpleInfo" type="TaggedValue"/>

element **ComplexInfo/ComplexInfo**

diagram	
type	**ComplexInfo**
children	**ID** **SimpleInfo** **ComplexInfo**
source	<xs:element name="ComplexInfo" type="ComplexInfo"/>

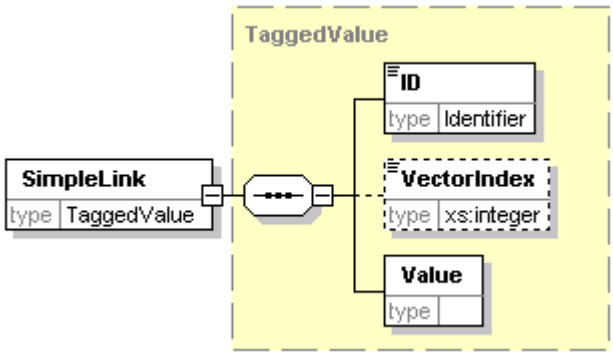
complexType **ComplexLink**

diagram	
children	**ID** **SimpleLink** **ComplexLink**
used by	elements **UMEOclass/Class/ExternalLink/ComplexLink** **ComplexLink/ComplexLink**
source	<xs:complexType name="ComplexLink"> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleLink" type="TaggedValue"/> <xs:element name="ComplexLink" type="ComplexLink"/> </xs:choice> </xs:sequence> </xs:complexType>

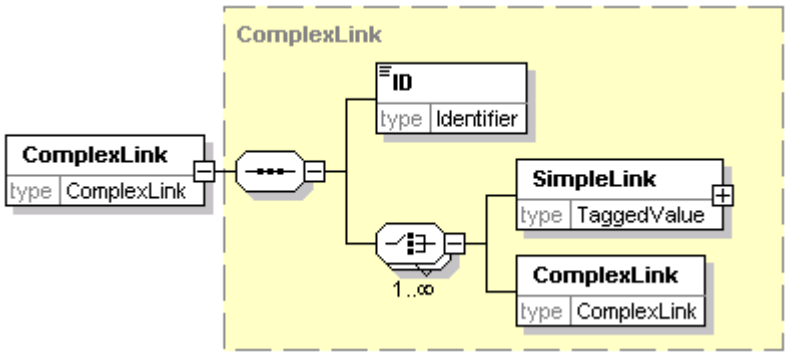
element **ComplexLink/ID**

diagram	
type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

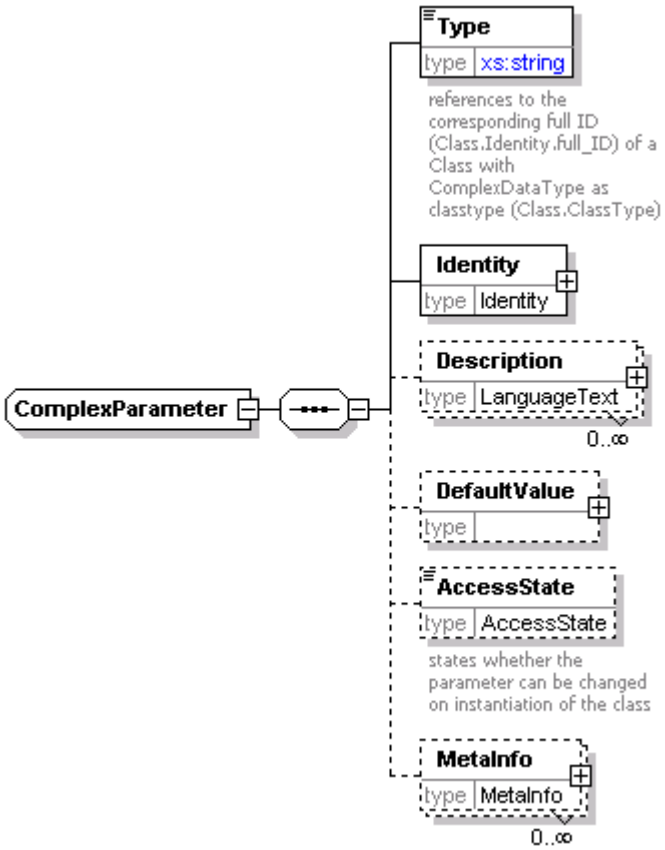
element **ComplexLink/SimpleLink**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value**
source	<xs:element name="SimpleLink" type="TaggedValue"/>

element **ComplexLink/ComplexLink**

diagram	
type	**ComplexLink**
children	**ID** **SimpleLink** **ComplexLink**
source	<xs:element name="ComplexLink" type="ComplexLink"/>

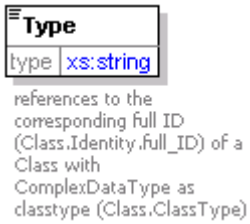
complexType **ComplexParameter**

diagram	
children	**Type Identity Description DefaultValue AccessState MetaInfo**
used by	elements **UMEOclass/Class/InstMetainfo/ComplexInfo UMEOclass/Class/ComplexParameter UMEOclass/Class/Method/ComplexParameter**

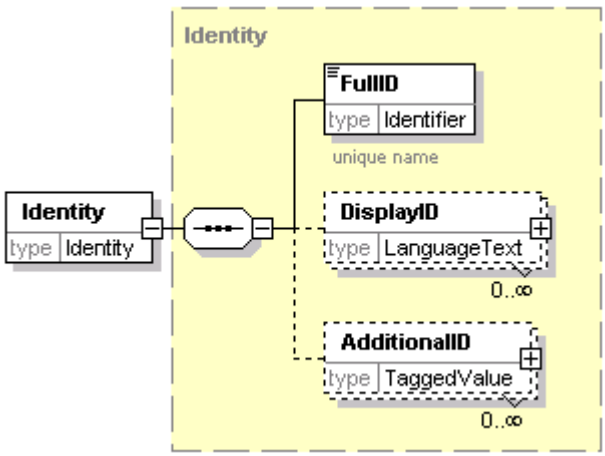
source

```
<xs:complexType name="ComplexParameter">
 <xs:sequence>
  <xs:element name="Type">
   <xs:annotation>
    <xs:documentation>references to the corresponding full ID (Class.Identity.full_ID) of a Class with
ComplexDataType as classtype (Class.ClassType) </xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:simpleContent>
     <xs:extension base="xs:string">
      <xs:attribute name="VectorSize" type="xs:integer" use="optional" default="1"/>
     </xs:extension>
    </xs:simpleContent>
   </xs:complexType>
  </xs:element>
  <xs:element name="Identity" type="Identity"/>
  <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
  <xs:element name="DefaultValue" minOccurs="0">
   <xs:complexType>
    <xs:choice maxOccurs="unbounded">
     <xs:element name="SimpleDefault" type="TaggedValue"/>
     <xs:element name="ComplexDefault" type="ComplexDefault"/>
    </xs:choice>
   </xs:complexType>
  </xs:element>
  <xs:element name="AccessState" type="AccessState" minOccurs="0">
   <xs:annotation>
    <xs:documentation>states whether the parameter can be changed on instantiation of the
class</xs:documentation>
   </xs:annotation>
  </xs:element>
  <xs:element name="MetaInfo" type="MetaInfo" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

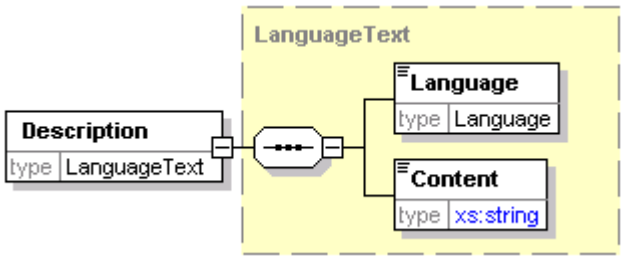
element **ComplexParameter/Type**

diagram	
type	extension of **xs:string**
attributes	Name: VectorSize; Type: xs:integer; Use: optional; Default: 1; Fixed: ; Annotation:
annotation	documentation: references to the corresponding full ID (Class.Identity.full_ID) of a Class with ComplexDataType as classtype (Class.ClassType)
source	<xs:element name="Type"> <xs:annotation> <xs:documentation>references to the corresponding full ID (Class.Identity.full_ID) of a Class with ComplexDataType as classtype (Class.ClassType) </xs:documentation> </xs:annotation> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute name="VectorSize" type="xs:integer" use="optional" default="1"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element>

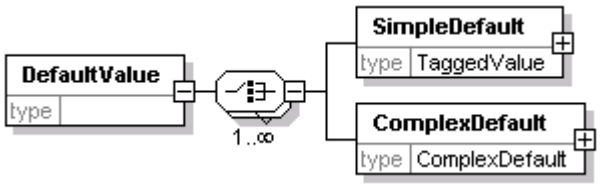
element **ComplexParameter/Identity**

diagram	
type	**Identity**
children	**FullID** **DisplayID** **AdditionalID**
source	<xs:element name="Identity" type="Identity"/>

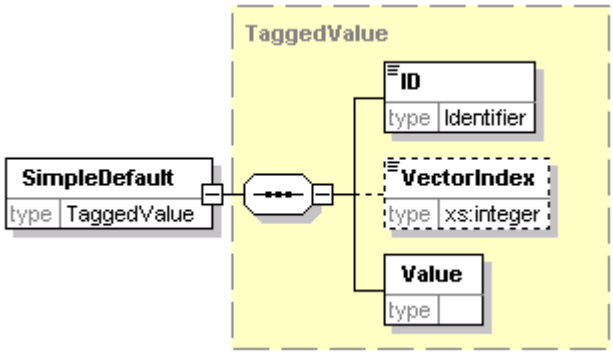
element **ComplexParameter/Description**

diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

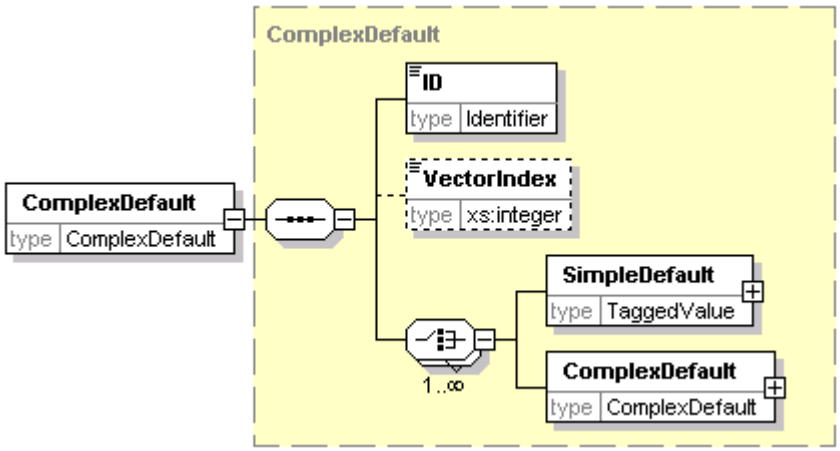
element **ComplexParameter/DefaultValue**

diagram	
children	**SimpleDefault** **ComplexDefault**
source	<xs:element name="DefaultValue" minOccurs="0"> <xs:complexType> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleDefault" type="TaggedValue"/> <xs:element name="ComplexDefault" type="ComplexDefault"/> </xs:choice> </xs:complexType> </xs:element>

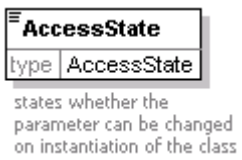
element **ComplexParameter/DefaultValue/SimpleDefault**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value**
source	<xs:element name="SimpleDefault" type="TaggedValue"/>

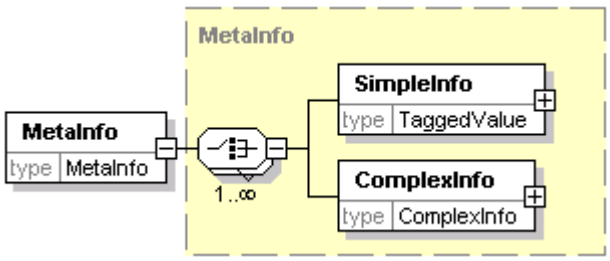
element **ComplexParameter/DefaultValue/ComplexDefault**

diagram	
type	**ComplexDefault**
children	**ID** **VectorIndex** **SimpleDefault** **ComplexDefault**
source	<xs:element name="ComplexDefault" type="ComplexDefault"/>

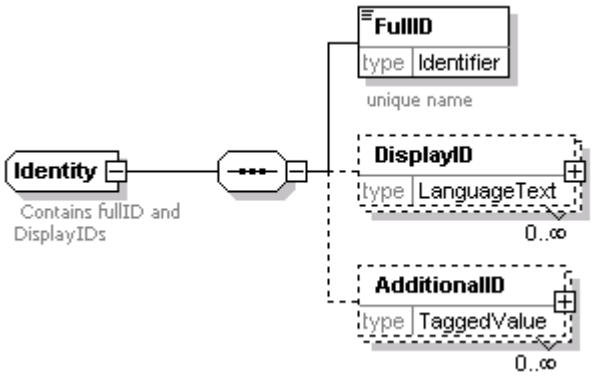
element **ComplexParameter/AccessState**

diagram	
type	**AccessState**
facets	enumeration unlocked enumeration locked
annotation	documentation states whether the parameter can be changed on instantiation of the class
source	<xs:element name="AccessState" type="AccessState" minOccurs="0"> <xs:annotation> <xs:documentation>states whether the parameter can be changed on instantiation of the class</xs:documentation> </xs:annotation> </xs:element>

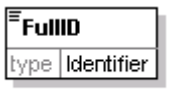
element **ComplexParameter/MetaInfo**

diagram	
type	**MetaInfo**
children	**SimpleInfo** **ComplexInfo**
source	<xs:element name="MetaInfo" type="MetaInfo" minOccurs="0" maxOccurs="unbounded"/>

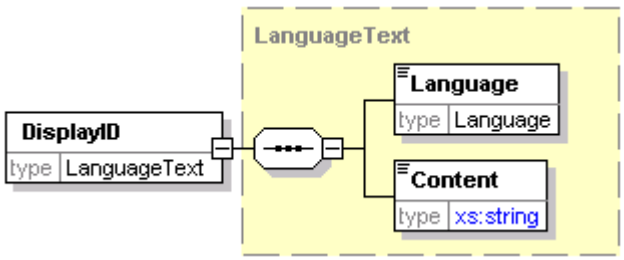
complexType **Identity**

diagram	
children	**FullID** **DisplayID** **AdditionalID**
used by	elements **UMEOclass/Class/Identity** **UMEOclass/Class/Method/Identity** **SimpleParameter/Identity** **ComplexParameter/Identity** **Role/Identity**
annotation	documentation Contains fullID and DisplayIDs
source	<xs:complexType name="Identity"> <xs:annotation> <xs:documentation> Contains fullID and DisplayIDs</xs:documentation> </xs:annotation> <xs:sequence> <xs:element name="FullID" type="Identifier"> <xs:annotation> <xs:documentation>unique name</xs:documentation> </xs:annotation> </xs:element> <xs:element name="DisplayID" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="AdditionalID" type="TaggedValue" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType>

element **Identity/FullID**

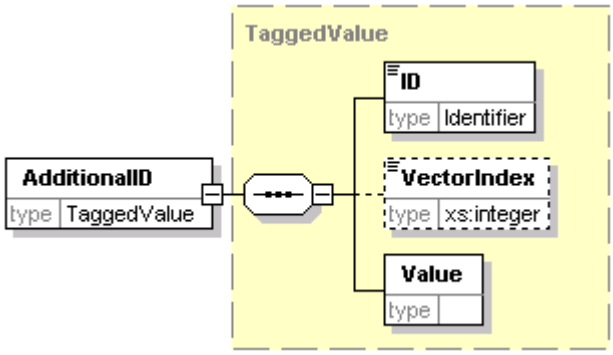
diagram	
type	**Identifier**
annotation	documentation unique name
source	<xs:element name="FullID" type="Identifier"> <xs:annotation> <xs:documentation>unique name</xs:documentation> </xs:annotation> </xs:element>

element **Identity/DisplayID**

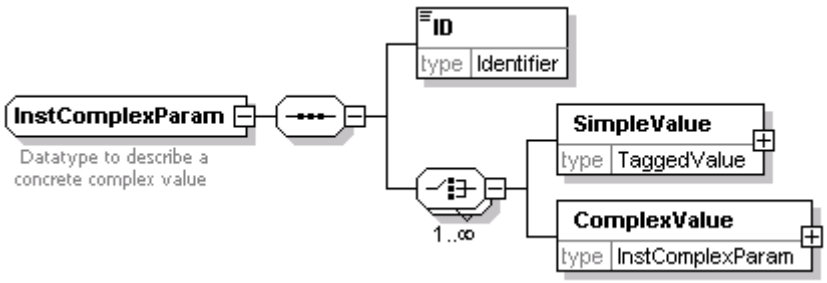
diagram	
type	**LanguageText**
children	**Language** **Content**

source	<xs:element name="DisplayID" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

element **Identity/AdditionalID**

diagram	
type	**TaggedValue**
children	**ID VectorIndex Value**
source	<xs:element name="AdditionalID" type="TaggedValue" minOccurs="0" maxOccurs="unbounded"/>

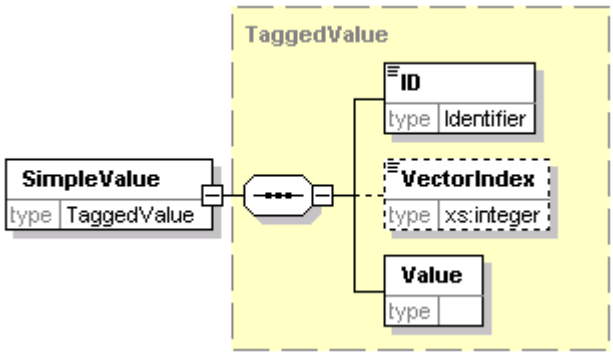
complexType **InstComplexParam**

diagram	
children	**ID SimpleValue ComplexValue**
used by	element **InstComplexParam/ComplexValue**
annotation	documentation Datatype to describe a concrete complex value
source	<xs:complexType name="InstComplexParam"> <xs:annotation> <xs:documentation> Datatype to describe a concrete complex value</xs:documentation> </xs:annotation> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleValue" type="TaggedValue"/> <xs:element name="ComplexValue" type="InstComplexParam"/> </xs:choice> </xs:sequence> </xs:complexType>

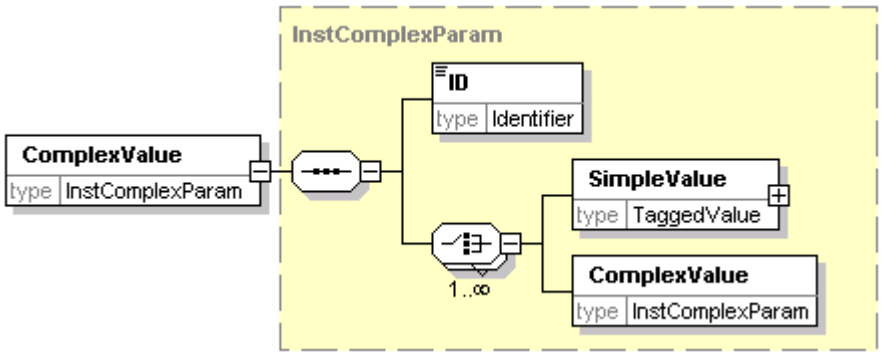
element **InstComplexParam/ID**

diagram	
type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

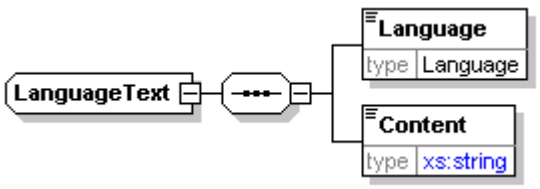
element **InstComplexParam/SimpleValue**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value**
source	<xs:element name="SimpleValue" type="TaggedValue"/>

element **InstComplexParam/ComplexValue**

diagram	
type	**InstComplexParam**
children	**ID** **SimpleValue** **ComplexValue**
source	<xs:element name="ComplexValue" type="InstComplexParam"/>

complexType **LanguageText**

diagram	
children	**Language** **Content**
used by	elements **UMEOclass/Comment** **UMEOclass/Class/Description** **UMEOclass/Class/Method/Description** **UMEOclass/Class/Partner/Description** **SimpleParameter/Description** **ComplexParameter/Description** **Role/Description** **Identity/DisplayID**
source	<xs:complexType name="LanguageText"> <xs:sequence> <xs:element name="Language" type="Language"/> <xs:element name="Content"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute name="Language" type="xs:string" use="optional" default="German"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element>

```
  </xs:sequence>
</xs:complexType>
```

element **LanguageText/Language**

diagram

type **Language**

facets
enumeration English (USA)
enumeration English (UK)
enumeration German
enumeration French
enumeration Japanese
enumeration Spanish
enumeration Portugese
enumeration Dutch

source

```
<xs:element name="Language" type="Language"/>
```

element **LanguageText/Content**

diagram

type extension of **xs:string**

attributes

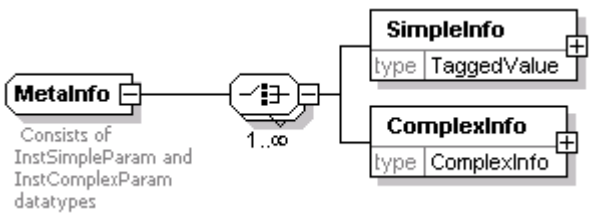
Name	Type	Use	Default	Fixed	Annotation
Language	xs:string	optional	German		

source

```
<xs:element name="Content">
 <xs:complexType>
  <xs:simpleContent>
   <xs:extension base="xs:string">
    <xs:attribute name="Language" type="xs:string" use="optional" default="German"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>
```

complexType **MetaInfo**

diagram



children **SimpleInfo** **ComplexInfo**

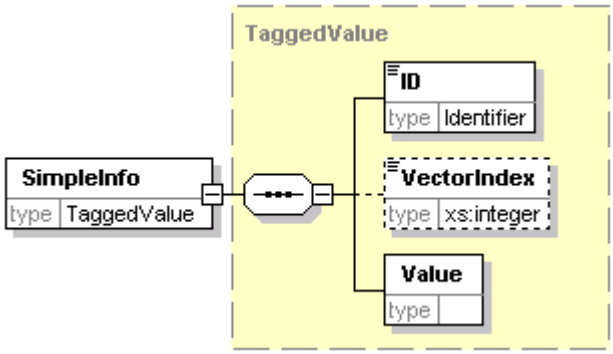
used by elements **UMEOclass/Class/ClassMetainfo** **UMEOclass/MetaInfo** **SimpleParameter/MetaInfo** **ComplexParameter/MetaInfo**

annotation documentation Consists of InstSimpleParam and InstComplexParam datatypes

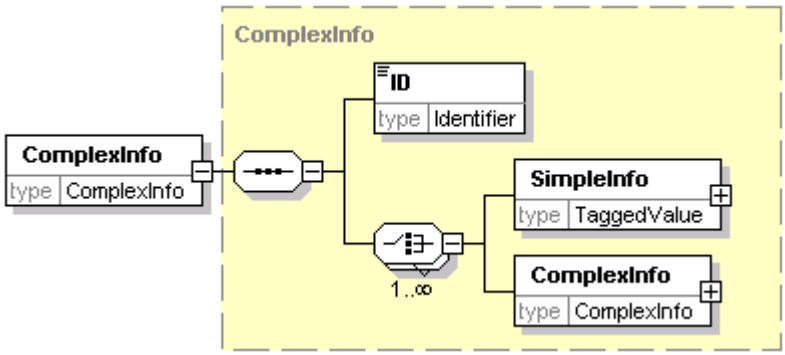
source

```
<xs:complexType name="MetaInfo">
 <xs:annotation>
  <xs:documentation> Consists of InstSimpleParam and InstComplexParam datatypes</xs:documentation>
 </xs:annotation>
 <xs:choice maxOccurs="unbounded">
  <xs:element name="SimpleInfo" type="TaggedValue"/>
  <xs:element name="ComplexInfo" type="ComplexInfo"/>
 </xs:choice>
</xs:complexType>
```

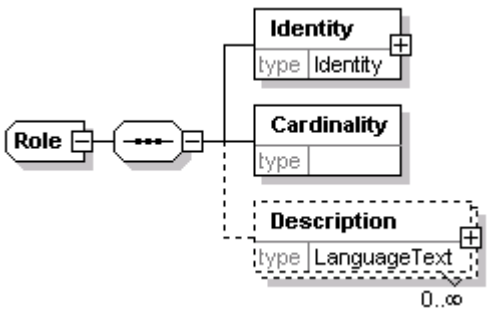
element **MetaInfo/SimpleInfo**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value**
source	<xs:element name="SimpleInfo" type="TaggedValue"/>

element **MetaInfo/ComplexInfo**

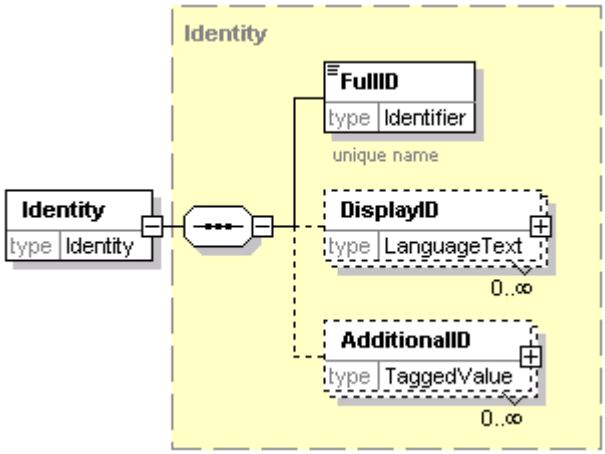
diagram	
type	**ComplexInfo**
children	**ID** **SimpleInfo** **ComplexInfo**
source	<xs:element name="ComplexInfo" type="ComplexInfo"/>

complexType **Role**

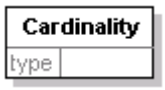
diagram	
children	**Identity** **Cardinality** **Description**
used by	elements **UMEOclass/Class/Partner/EORMrole** **UMEOclass/Class/Partner/PartnerRole**
source	<xs:complexType name="Role"> <xs:sequence> <xs:element name="Identity" type="Identity"/> <xs:element name="Cardinality"/> <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence>

	</xs:complexType>

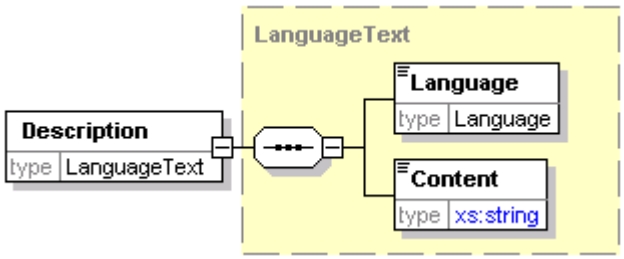
element **Role/Identity**

diagram	
type	**Identity**
children	**FullID** **DisplayID** **AdditionalID**
source	<xs:element name="Identity" type="Identity"/>

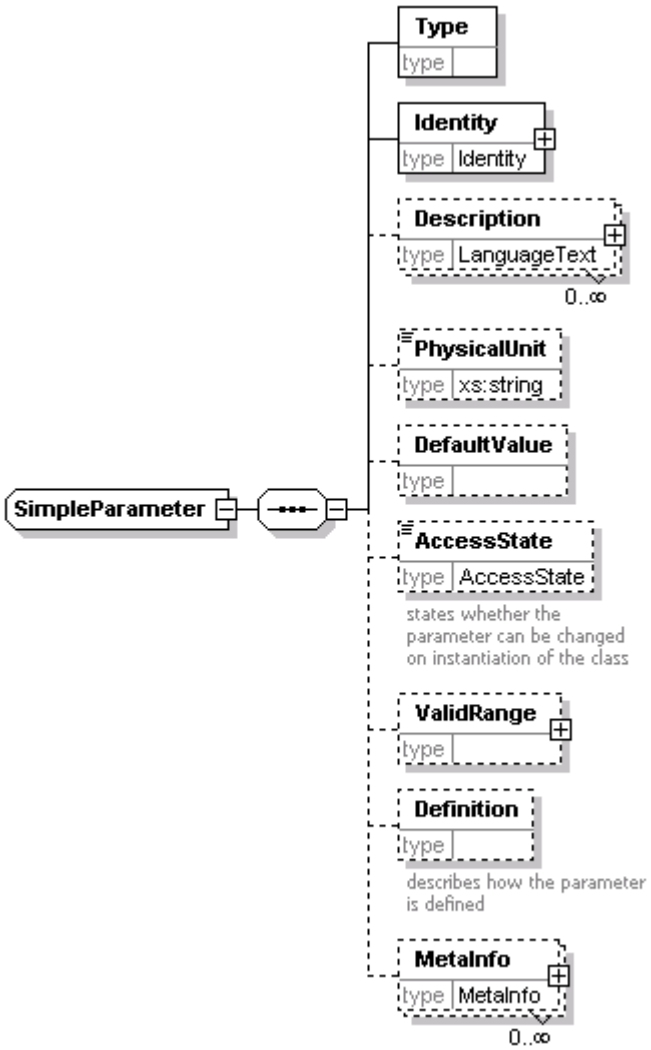
element **Role/Cardinality**

diagram	
source	<xs:element name="Cardinality"/>

element **Role/Description**

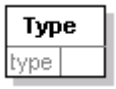
diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

complexType **SimpleParameter**

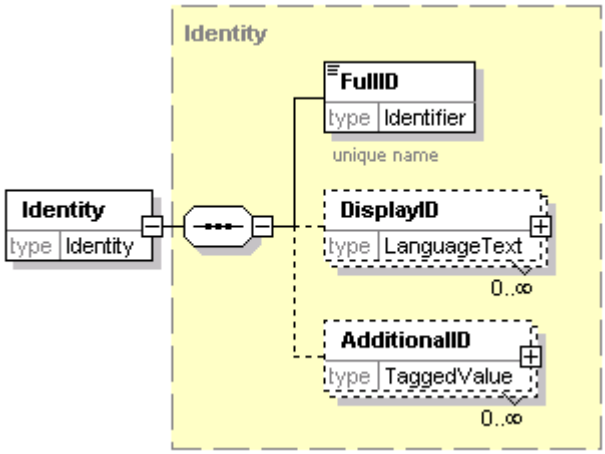
diagram	
children	**Type Identity Description PhysicalUnit DefaultValue AccessState ValidRange Definition MetaInfo**
used by	elements **UMEOclass/Class/InstMetainfo/SimpleInfo UMEOclass/Class/SimpleParameter UMEOclass/Class/Method/SimpleParameter**
source	<xs:complexType name="SimpleParameter"> <xs:sequence> <xs:element name="Type"/> <xs:element name="Identity" type="Identity"/> <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="PhysicalUnit" type="xs:string" minOccurs="0"/> <xs:element name="DefaultValue" minOccurs="0"/> <xs:element name="AccessState" type="AccessState" minOccurs="0"> <xs:annotation> <xs:documentation>states whether the parameter can be changed on instantiation of the class</xs:documentation> </xs:annotation> </xs:element> <xs:element name="ValidRange" minOccurs="0"> <xs:complexType> <xs:choice> <xs:sequence> <xs:element name="UpperValue"/> <xs:element name="LowerValue"/> <xs:element name="StepSize" minOccurs="0"/> </xs:sequence> <xs:element name="Value" maxOccurs="unbounded"/> </xs:choice> </xs:complexType> </xs:element> <xs:element name="Definition" minOccurs="0"> <xs:annotation>

```
      <xs:documentation>describes how the parameter is defined</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="MetaInfo" type="MetaInfo" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
```

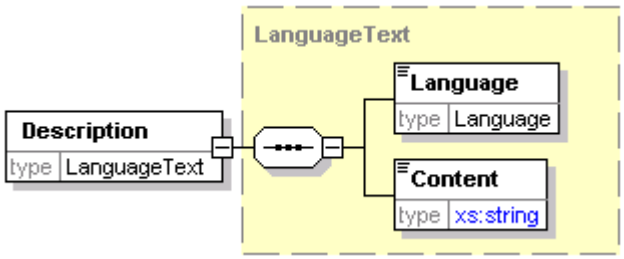
element **SimpleParameter/Type**

diagram	
source	<xs:element name="Type"/>

element **SimpleParameter/Identity**

diagram	
type	**Identity**
children	**FullID** **DisplayID** **AdditionalID**
source	<xs:element name="Identity" type="Identity"/>

element **SimpleParameter/Description**

diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

element **SimpleParameter/PhysicalUnit**

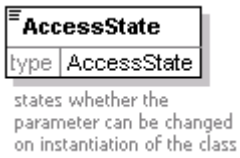
diagram	
type	**xs:string**

source	<xs:element name="PhysicalUnit" type="xs:string" minOccurs="0"/>

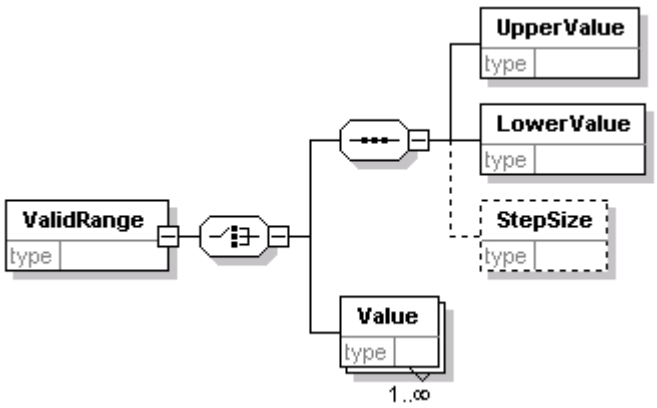
element **SimpleParameter/DefaultValue**

diagram	
source	<xs:element name="DefaultValue" minOccurs="0"/>

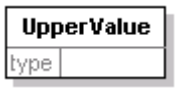
element **SimpleParameter/AccessState**

diagram	
type	**AccessState**
facets	enumeration unlocked enumeration locked
annotation	documentation states whether the parameter can be changed on instantiation of the class
source	<xs:element name="AccessState" type="AccessState" minOccurs="0"> <xs:annotation> <xs:documentation>states whether the parameter can be changed on instantiation of the class</xs:documentation> </xs:annotation> </xs:element>

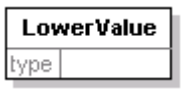
element **SimpleParameter/ValidRange**

diagram	
children	**UpperValue LowerValue StepSize Value**
source	<xs:element name="ValidRange" minOccurs="0"> <xs:complexType> <xs:choice> <xs:sequence> <xs:element name="UpperValue"/> <xs:element name="LowerValue"/> <xs:element name="StepSize" minOccurs="0"/> </xs:sequence> <xs:element name="Value" maxOccurs="unbounded"/> </xs:choice> </xs:complexType> </xs:element>

element **SimpleParameter/ValidRange/UpperValue**

diagram	
source	<xs:element name="UpperValue"/>

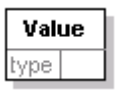
element **SimpleParameter/ValidRange/LowerValue**

diagram	
source	<xs:element name="LowerValue"/>

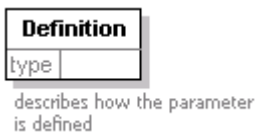
element **SimpleParameter/ValidRange/StepSize**

diagram	
source	<xs:element name="StepSize" minOccurs="0"/>

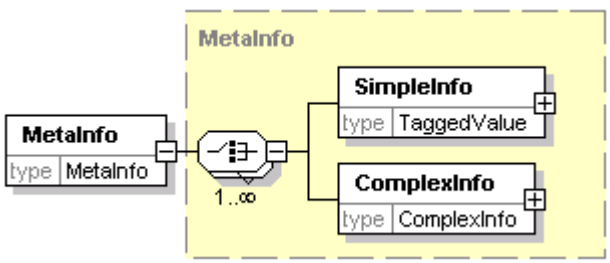
element **SimpleParameter/ValidRange/Value**

diagram	
source	<xs:element name="Value" maxOccurs="unbounded"/>

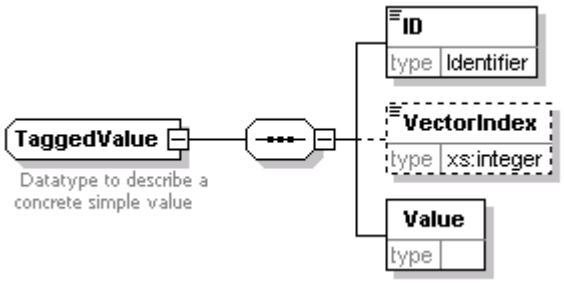
element **SimpleParameter/Definition**

diagram	
annotation	documentation describes how the parameter is defined
source	<xs:element name="Definition" minOccurs="0"> <xs:annotation> <xs:documentation>describes how the parameter is defined</xs:documentation> </xs:annotation> </xs:element>

element **SimpleParameter/MetaInfo**

diagram	
type	**MetaInfo**
children	**SimpleInfo** **ComplexInfo**
source	<xs:element name="MetaInfo" type="MetaInfo" minOccurs="0" maxOccurs="unbounded"/>

complexType **TaggedValue**

diagram	
children	**ID VectorIndex Value**
used by	elements **Identity/AdditionalID ComplexParameter/DefaultValue/SimpleDefault ComplexDefault/SimpleDefault MetaInfo/SimpleInfo ComplexInfo/SimpleInfo UMEOclass/Class/ExternalLink/SimpleLink ComplexLink/SimpleLink InstComplexParam/SimpleValue**
annotation	documentation Datatype to describe a concrete simple value
source	<xs:complexType name="TaggedValue"> <xs:annotation> <xs:documentation> Datatype to describe a concrete simple value</xs:documentation> </xs:annotation> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:element name="VectorIndex" type="xs:integer" minOccurs="0"/> <xs:element name="Value"/> </xs:sequence> </xs:complexType>

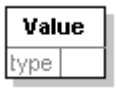
element **TaggedValue/ID**

diagram	
type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

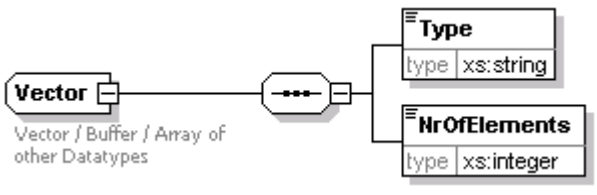
element **TaggedValue/VectorIndex**

diagram	
type	**xs:integer**
source	<xs:element name="VectorIndex" type="xs:integer" minOccurs="0"/>

element **TaggedValue/Value**

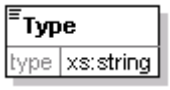
diagram	
source	<xs:element name="Value"/>

complexType **Vector**

diagram	

children	**Type NrOfElements**
annotation	documentation Vector / Buffer / Array of other Datatypes
source	<xs:complexType name="Vector"> <xs:annotation> <xs:documentation>Vector / Buffer / Array of other Datatypes</xs:documentation> </xs:annotation> <xs:sequence> <xs:element name="Type" type="xs:string"/> <xs:element name="NrOfElements" type="xs:integer"/> </xs:sequence> </xs:complexType>

element **Vector/Type**

diagram	
type	**xs:string**
source	<xs:element name="Type" type="xs:string"/>

element **Vector/NrOfElements**

diagram	
type	**xs:integer**
source	<xs:element name="NrOfElements" type="xs:integer"/>

simpleType **AccessState**

type	restriction of **xs:string**
used by	elements **SimpleParameter/AccessState ComplexParameter/AccessState**
facets	enumeration unlocked enumeration locked
annotation	documentation Enumerated {unlocked, locked}
source	<xs:simpleType name="AccessState"> <xs:annotation> <xs:documentation> Enumerated {unlocked, locked}</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"> <xs:enumeration value="unlocked"/> <xs:enumeration value="locked"/> </xs:restriction> </xs:simpleType>

simpleType **ClassType**

type	restriction of **xs:string**
used by	elements **UMEOclass/Class/ClassType UMEOclass/Class/Partner/PartnerClassType**
facets	enumeration EOclass enumeration MTRTclass enumeration EORMclass enumeration ComplexDataType enumeration MTRTrelation
annotation	documentation Enumerated {EOclass, MTRTclass, EORMclass, ComplexDataType, MTRTrelation}
source	<xs:simpleType name="ClassType"> <xs:annotation>

	<xs:documentation> Enumerated {EOclass, MTRTclass, EORMclass, ComplexDataType, MTRTrelation}</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"> <xs:enumeration value="EOclass"/> <xs:enumeration value="MTRTclass"/> <xs:enumeration value="EORMclass"/> <xs:enumeration value="ComplexDataType"/> <xs:enumeration value="MTRTrelation"/> </xs:restriction> </xs:simpleType>

simpleType **Direction**

type	restriction of **xs:string**
used by	element **UMEOclass/Class/Partner/Direction**
facets	enumeration in enumeration out enumeration undirected
annotation	documentation Enumerated {in, out, undirected}
source	<xs:simpleType name="Direction"> <xs:annotation> <xs:documentation> Enumerated {in, out, undirected}</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"> <xs:enumeration value="in"/> <xs:enumeration value="out"/> <xs:enumeration value="undirected"/> </xs:restriction> </xs:simpleType>

simpleType **Identifier**

type	**xs:string**
used by	elements **Identity/FullID** **ComplexDefault/ID** **ComplexInfo/ID** **ComplexLink/ID** **TaggedValue/ID** **InstComplexParam/ID** **UMEOclass/Class/MTRTclassID** **UMEOclass/Class/Partner/PartnerClassID**
annotation	documentation Datatype to describe IDs, based on xs:string
source	<xs:simpleType name="Identifier"> <xs:annotation> <xs:documentation> Datatype to describe IDs, based on xs:string</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"/> </xs:simpleType>

simpleType **Language**

type	restriction of **xs:string**
used by	element **LanguageText/Language**
facets	enumeration English (USA) enumeration English (UK) enumeration German enumeration French enumeration Japanese enumeration Spanish enumeration Portugese enumeration Dutch
annotation	documentation Enumerated {English (USA), English (UK), German, French, Japanese, Spanish, Portugese, Dutch}
source	<xs:simpleType name="Language"> <xs:annotation> <xs:documentation> Enumerated {English (USA), English (UK), German, French, Japanese, Spanish, Portugese, Dutch}</xs:documentation> </xs:annotation>

```
  <xs:restriction base="xs:string">
    <xs:enumeration value="English (USA)"/>
    <xs:enumeration value="English (UK)"/>
    <xs:enumeration value="German"/>
    <xs:enumeration value="French"/>
    <xs:enumeration value="Japanese"/>
    <xs:enumeration value="Spanish"/>
    <xs:enumeration value="Portugese"/>
    <xs:enumeration value="Dutch"/>
  </xs:restriction>
</xs:simpleType>
```

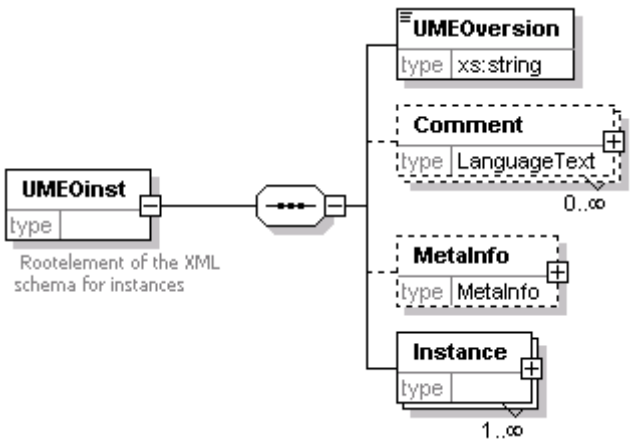
UMEOinst Schema

schema location: **D:\Martin\UMEOinstV1c.xsd**

Elements	Complex types	Simple types
UMEOinst	**ComplexInfo**	**Identifier**
	Identity	**InstanceType**
	InstComplexParam	**Language**
	LanguageText	
	MetaInfo	
	TaggedValue	

element **UMEOinst**

diagram



children **UMEOversion** **Comment** **MetaInfo** **Instance**

annotation documentation Rootelement of the XML schema for instances

source

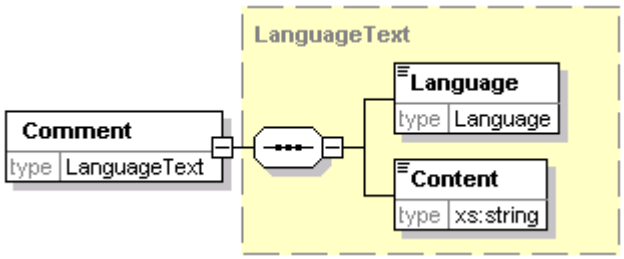
```
<xs:element name="UMEOinst">
  <xs:annotation>
    <xs:documentation> Rootelement of the XML schema for instances</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="UMEOversion" type="xs:string"/>
      <xs:element name="Comment" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>
      <xs:element name="Instance" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="InstanceType" type="InstanceType"/>
            <xs:element name="ClassID" type="Identifier">
              <xs:annotation>
                <xs:documentation>reference to full ID of corresponding class
(Class.Identity.FullID)</xs:documentation>
```

```
          </xs:annotation>
        </xs:element>
        <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="Identity" type="Identity"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="SimpleParameter" type="TaggedValue"/>
          <xs:element name="ComplexParameter" type="InstComplexParam"/>
        </xs:choice>
        <xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>
        <xs:element name="Role" minOccurs="0" maxOccurs="unbounded">
          <xs:annotation>
            <xs:documentation>only and obligatory for EORMinstance</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="Identifier">
                <xs:annotation>
                  <xs:documentation>reference to the full ID of the PartnerRole
(Class.Partner.PartnerRole.Identity.FullID)</xs:documentation>
                </xs:annotation>
              </xs:element>
              <xs:element name="Cardinality" minOccurs="0">
                <xs:annotation>
                  <xs:documentation>number of links with 'Identity'within the EORMinstance</xs:documentation>
                </xs:annotation>
              </xs:element>
              <xs:element name="RolePartnerInstanceType" type="InstanceType">
                <xs:annotation>
                  <xs:documentation>reference to the InstanceType of partnerinstance
(Instance.InstanceType)</xs:documentation>
                </xs:annotation>
              </xs:element>
              <xs:element name="RolePartner" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Index" type="xs:integer">
                      <xs:annotation>
                        <xs:documentation>Index [0,.., Cardinality-1]</xs:documentation>
                      </xs:annotation>
                    </xs:element>
                    <xs:element name="InstanceID" type="Identifier">
                      <xs:annotation>
                        <xs:documentation>reference to full ID of partnerinstance
(Instance.Identity.FullID)</xs:documentation>
                      </xs:annotation>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

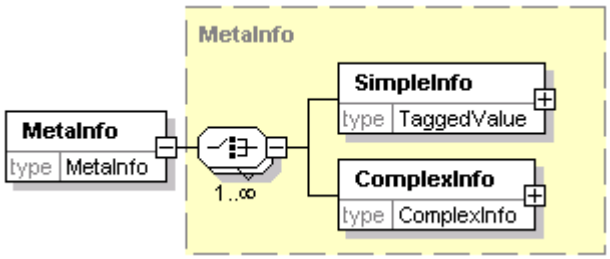
element **UMEOinst/UMEOversion**

diagram	
type	**xs:string**
source	`<xs:element name="UMEOversion" type="xs:string"/>`

element **UMEOinst/Comment**

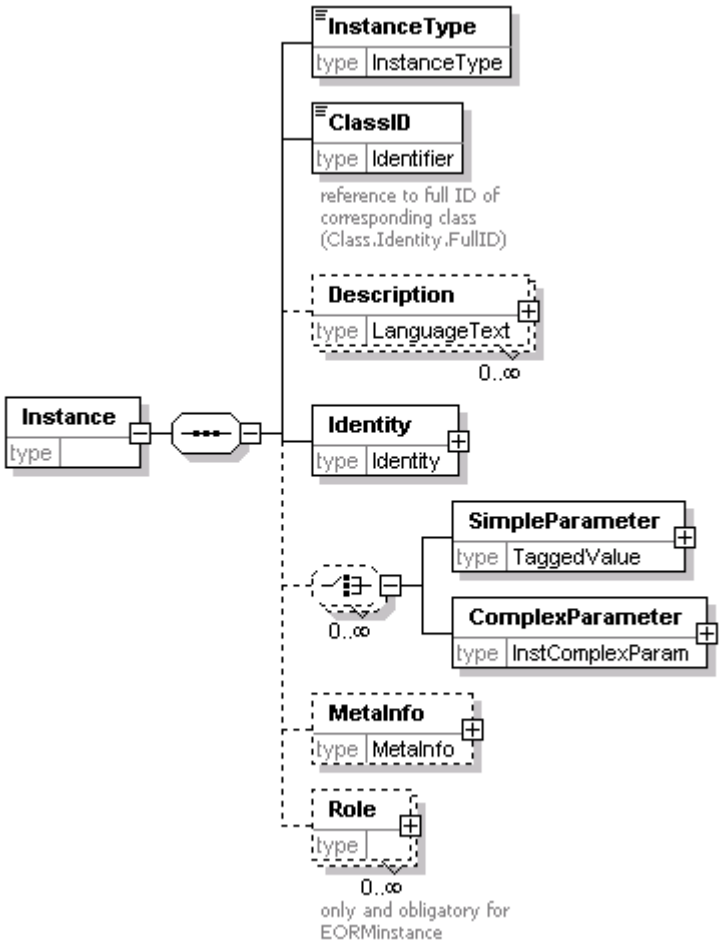
diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Comment" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

element **UMEOinst/MetaInfo**

diagram	
type	**MetaInfo**
children	**SimpleInfo** **ComplexInfo**
source	<xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>

element **UMEOinst/Instance**

diagram



children **InstanceType ClassID Description Identity SimpleParameter ComplexParameter MetaInfo Role**

source

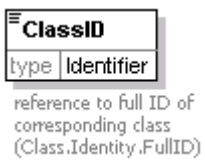
```
<xs:element name="Instance" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="InstanceType" type="InstanceType"/>
   <xs:element name="ClassID" type="Identifier">
    <xs:annotation>
     <xs:documentation>reference to full ID of corresponding class (Class.Identity.FullID)</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="Identity" type="Identity"/>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="SimpleParameter" type="TaggedValue"/>
    <xs:element name="ComplexParameter" type="InstComplexParam"/>
   </xs:choice>
   <xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>
   <xs:element name="Role" minOccurs="0" maxOccurs="unbounded">
    <xs:annotation>
     <xs:documentation>only and obligatory for EORMinstance</xs:documentation>
    </xs:annotation>
    <xs:complexType>
     <xs:sequence>
      <xs:element name="ID" type="Identifier">
       <xs:annotation>
        <xs:documentation>reference to the full ID of the PartnerRole
(Class.Partner.PartnerRole.Identity.FullID)</xs:documentation>
       </xs:annotation>
      </xs:element>
      <xs:element name="Cardinality" minOccurs="0">
       <xs:annotation>
        <xs:documentation>number of links with 'Identity'within the EORMinstance</xs:documentation>
       </xs:annotation>
      </xs:element>
      <xs:element name="RolePartnerInstanceType" type="InstanceType">
```

```
            <xs:annotation>
              <xs:documentation>reference to the InstanceType of partnerinstance
(Instance.InstanceType)</xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="RolePartner" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Index" type="xs:integer">
                  <xs:annotation>
                    <xs:documentation>Index [0,.., Cardinality-1]</xs:documentation>
                  </xs:annotation>
                </xs:element>
                <xs:element name="InstanceID" type="Identifier">
                  <xs:annotation>
                    <xs:documentation>reference to full ID of partnerinstance
(Instance.Identity.FullID)</xs:documentation>
                  </xs:annotation>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

element **UMEOinst/Instance/InstanceType**

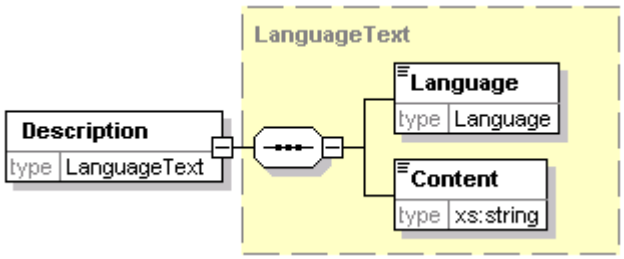
diagram	
type	**InstanceType**
facets	enumeration EOinstance enumeration EORMinstance
source	`<xs:element name="InstanceType" type="InstanceType"/>`

element **UMEOinst/Instance/ClassID**

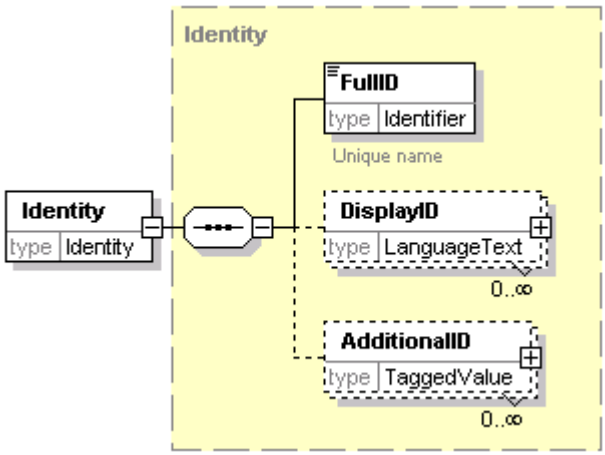
diagram	
type	**Identifier**
annotation	documentation reference to full ID of corresponding class (Class.Identity.FullID)

source	`<xs:element name="ClassID" type="Identifier">`
	`  <xs:annotation>`
	`  <xs:documentation>reference to full ID of corresponding class (Class.Identity.FullID)</xs:documentation>`
	`  </xs:annotation>`
	`</xs:element>`

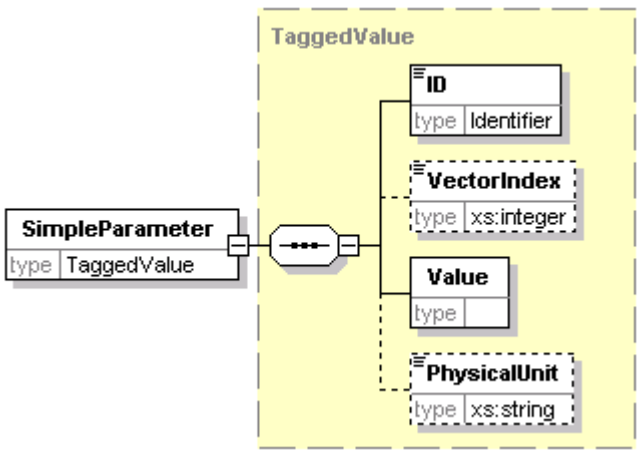
element **UMEOinst/Instance/Description**

diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="Description" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

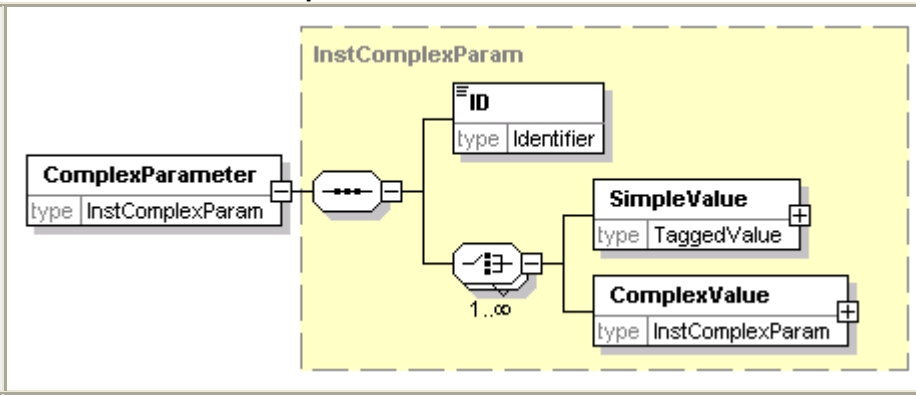
element **UMEOinst/Instance/Identity**

diagram	
type	**Identity**
children	**FullID** **DisplayID** **AdditionalID**
source	<xs:element name="Identity" type="Identity"/>

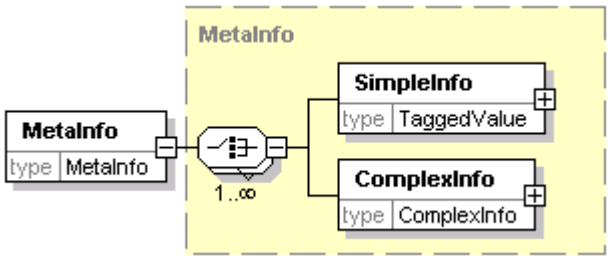
element **UMEOinst/Instance/SimpleParameter**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value** **PhysicalUnit**
source	<xs:element name="SimpleParameter" type="TaggedValue"/>

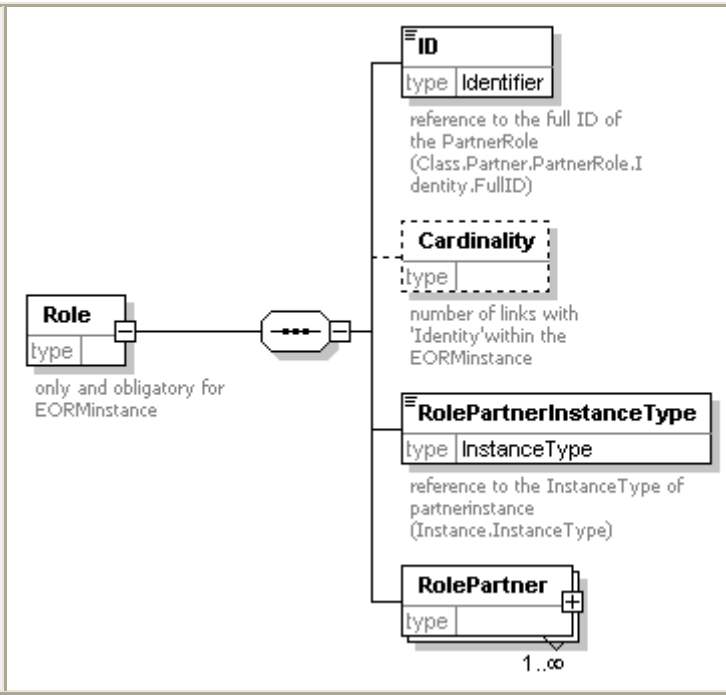
element **UMEOinst/Instance/ComplexParameter**

diagram	
type	**InstComplexParam**
children	**ID SimpleValue ComplexValue**
source	<xs:element name="ComplexParameter" type="InstComplexParam"/>

element **UMEOinst/Instance/MetaInfo**

diagram	
type	**MetaInfo**
children	**SimpleInfo ComplexInfo**
source	<xs:element name="MetaInfo" type="MetaInfo" minOccurs="0"/>

element **UMEOinst/Instance/Role**

diagram	

children	**ID Cardinality RolePartnerInstanceType RolePartner**
annotation	documentation only and obligatory for EORMinstance

source

```
<xs:element name="Role" minOccurs="0" maxOccurs="unbounded">
 <xs:annotation>
  <xs:documentation>only and obligatory for EORMinstance</xs:documentation>
 </xs:annotation>
 <xs:complexType>
  <xs:sequence>
   <xs:element name="ID" type="Identifier">
    <xs:annotation>
     <xs:documentation>reference to the full ID of the PartnerRole
(Class.Partner.PartnerRole.Identity.FullID)</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="Cardinality" minOccurs="0">
    <xs:annotation>
     <xs:documentation>number of links with 'Identity'within the EORMinstance</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="RolePartnerInstanceType" type="InstanceType">
    <xs:annotation>
     <xs:documentation>reference to the InstanceType of partnerinstance
(Instance.InstanceType)</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="RolePartner" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Index" type="xs:integer">
       <xs:annotation>
        <xs:documentation>Index [0,.., Cardinality-1]</xs:documentation>
       </xs:annotation>
      </xs:element>
      <xs:element name="InstanceID" type="Identifier">
       <xs:annotation>
        <xs:documentation>reference to full ID of partnerinstance (Instance.Identity.FullID)</xs:documentation>
       </xs:annotation>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

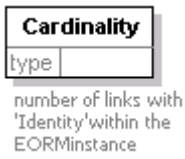
element **UMEOinst/Instance/Role/ID**

diagram	
type	**Identifier**
annotation	documentation reference to the full ID of the PartnerRole (Class.Partner.PartnerRole.Identity.FullID)

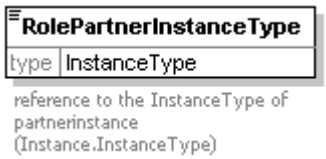
source

```
<xs:element name="ID" type="Identifier">
 <xs:annotation>
  <xs:documentation>reference to the full ID of the PartnerRole
(Class.Partner.PartnerRole.Identity.FullID)</xs:documentation>
 </xs:annotation>
</xs:element>
```

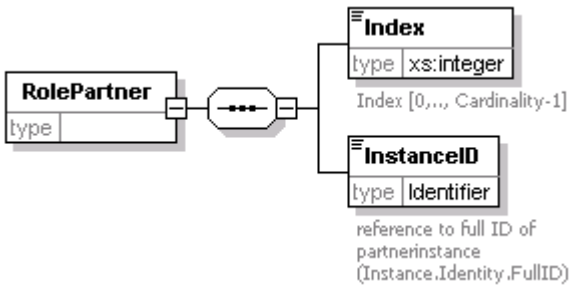
element **UMEOinst/Instance/Role/Cardinality**

diagram	
annotation	documentation number of links with 'Identity'within the EORMinstance
source	<xs:element name="Cardinality" minOccurs="0"> <xs:annotation> <xs:documentation>number of links with 'Identity'within the EORMinstance</xs:documentation> </xs:annotation> </xs:element>

element **UMEOinst/Instance/Role/RolePartnerInstanceType**

diagram	
type	**InstanceType**
facets	enumeration EOinstance enumeration EORMinstance
annotation	documentation reference to the InstanceType of partnerinstance (Instance.InstanceType)
source	<xs:element name="RolePartnerInstanceType" type="InstanceType"> <xs:annotation> <xs:documentation>reference to the InstanceType of partnerinstance (Instance.InstanceType)</xs:documentation> </xs:annotation> </xs:element>

element **UMEOinst/Instance/Role/RolePartner**

diagram	
children	**Index** **InstanceID**
source	<xs:element name="RolePartner" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="Index" type="xs:integer"> <xs:annotation> <xs:documentation>Index [0,.., Cardinality-1]</xs:documentation> </xs:annotation> </xs:element> <xs:element name="InstanceID" type="Identifier"> <xs:annotation> <xs:documentation>reference to full ID of partnerinstance (Instance.Identity.FullID)</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element>

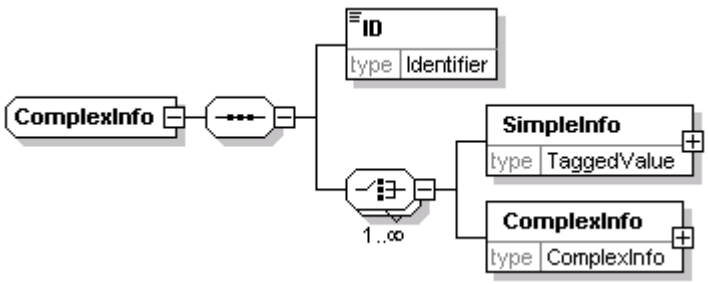
element **UMEOinst/Instance/Role/RolePartner/Index**

diagram	
type	**xs:integer**
annotation	documentation Index [0,.., Cardinality-1]
source	<xs:element name="Index" type="xs:integer"> <xs:annotation> <xs:documentation>Index [0,.., Cardinality-1]</xs:documentation> </xs:annotation> </xs:element>

element **UMEOinst/Instance/Role/RolePartner/InstanceID**

diagram	
type	**Identifier**
annotation	documentation reference to full ID of partnerinstance (Instance.Identity.FullID)
source	<xs:element name="InstanceID" type="Identifier"> <xs:annotation> <xs:documentation>reference to full ID of partnerinstance (Instance.Identity.FullID)</xs:documentation> </xs:annotation> </xs:element>

complexType **ComplexInfo**

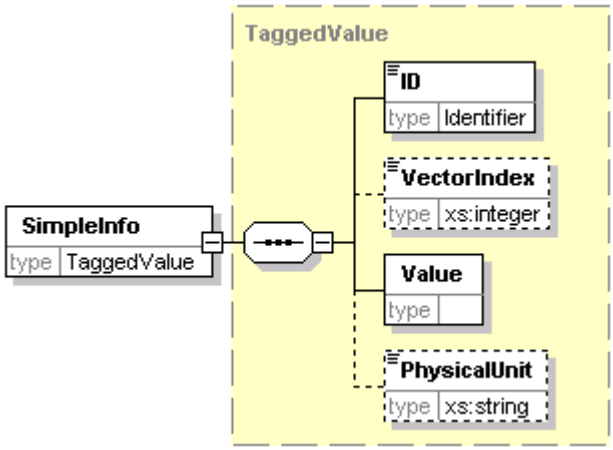
diagram	
children	**ID SimpleInfo ComplexInfo**
used by	elements **MetaInfo/ComplexInfo ComplexInfo/ComplexInfo**
source	<xs:complexType name="ComplexInfo"> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleInfo" type="TaggedValue"/> <xs:element name="ComplexInfo" type="ComplexInfo"/> </xs:choice> </xs:sequence> </xs:complexType>

element **ComplexInfo/ID**

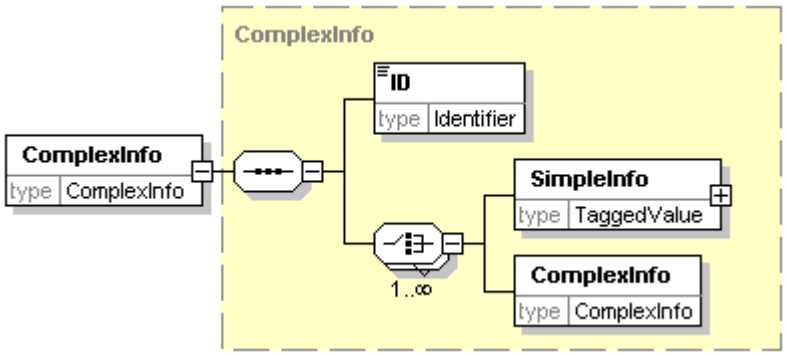
diagram	

type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

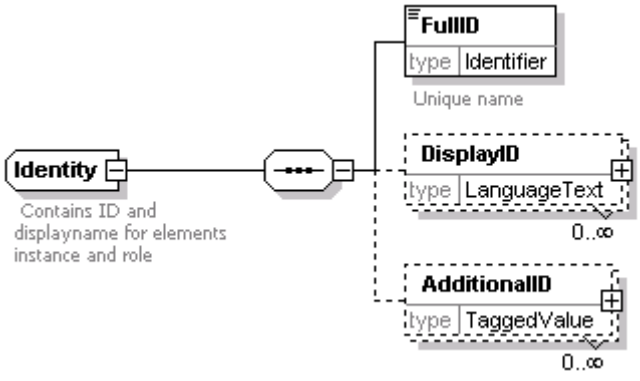
element **ComplexInfo/SimpleInfo**

diagram	
type	**TaggedValue**
children	**ID VectorIndex Value PhysicalUnit**
source	<xs:element name="SimpleInfo" type="TaggedValue"/>

element **ComplexInfo/ComplexInfo**

diagram	
type	**ComplexInfo**
children	**ID SimpleInfo ComplexInfo**
source	<xs:element name="ComplexInfo" type="ComplexInfo"/>

complexType **Identity**

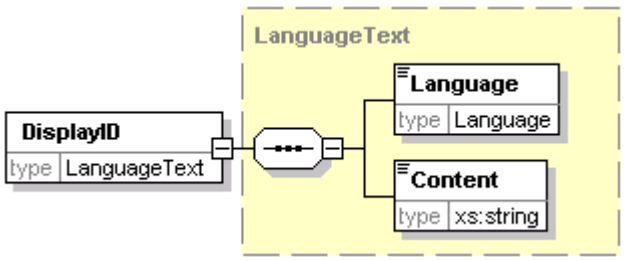
diagram	

children	**FullID** **DisplayID** **AdditionalID**
used by	element **UMEOinst/Instance/Identity**
annotation	documentation Contains ID and displayname for elements instance and role
source	<xs:complexType name="Identity"> <xs:annotation> <xs:documentation> Contains ID and displayname for elements instance and role</xs:documentation> </xs:annotation> <xs:sequence> <xs:element name="FullID" type="Identifier"> <xs:annotation> <xs:documentation>Unique name</xs:documentation> </xs:annotation> </xs:element> <xs:element name="DisplayID" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="AdditionalID" type="TaggedValue" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType>

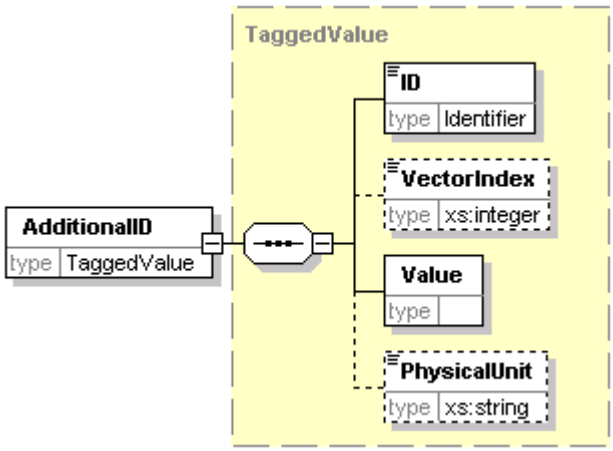
element **Identity/FullID**

diagram	
type	**Identifier**
annotation	documentation Unique name
source	<xs:element name="FullID" type="Identifier"> <xs:annotation> <xs:documentation>Unique name</xs:documentation> </xs:annotation> </xs:element>

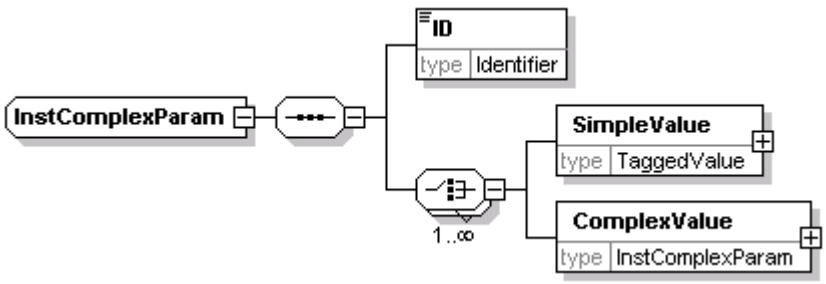
element **Identity/DisplayID**

diagram	
type	**LanguageText**
children	**Language** **Content**
source	<xs:element name="DisplayID" type="LanguageText" minOccurs="0" maxOccurs="unbounded"/>

element **Identity/AdditionalID**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value** **PhysicalUnit**
source	<xs:element name="AdditionalID" type="TaggedValue" minOccurs="0" maxOccurs="unbounded"/>

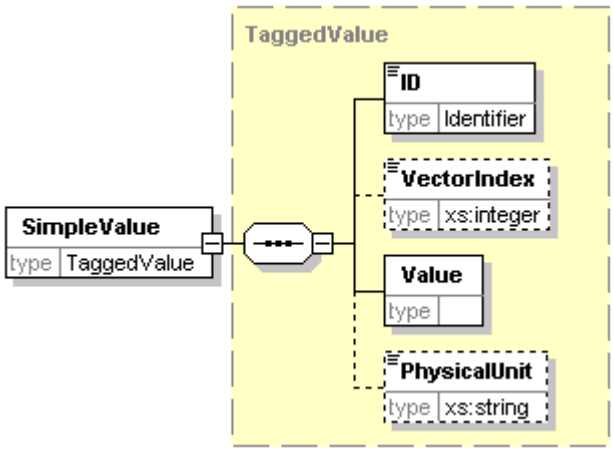
complexType **InstComplexParam**

diagram	
children	**ID** **SimpleValue** **ComplexValue**
used by	elements **UMEOinst/Instance/ComplexParameter** **InstComplexParam/ComplexValue**
source	<xs:complexType name="InstComplexParam"> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleValue" type="TaggedValue"/> <xs:element name="ComplexValue" type="InstComplexParam"/> </xs:choice> </xs:sequence> </xs:complexType>

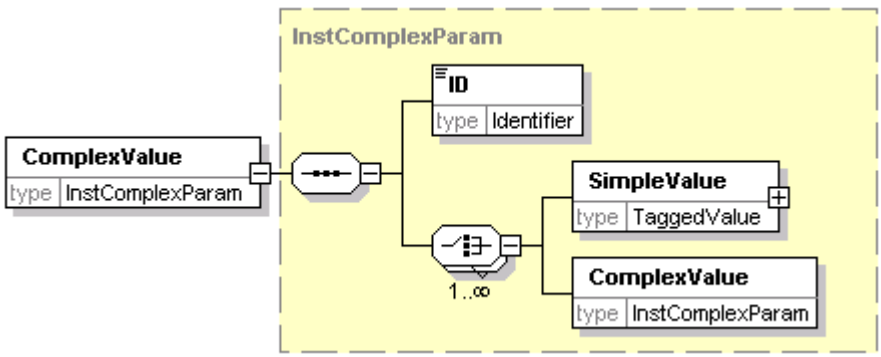
element **InstComplexParam/ID**

diagram	
type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

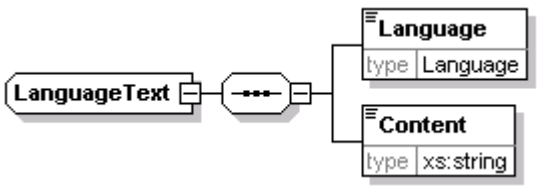
element **InstComplexParam/SimpleValue**

diagram	
type	**TaggedValue**
children	**ID** **VectorIndex** **Value** **PhysicalUnit**
source	<xs:element name="SimpleValue" type="TaggedValue"/>

element **InstComplexParam/ComplexValue**

diagram	
type	**InstComplexParam**
children	**ID** **SimpleValue** **ComplexValue**
source	<xs:element name="ComplexValue" type="InstComplexParam"/>

complexType **LanguageText**

diagram	
children	**Language** **Content**
used by	elements **UMEOinst/Comment** **UMEOinst/Instance/Description** **Identity/DisplayID**
source	<xs:complexType name="LanguageText"> <xs:sequence> <xs:element name="Language" type="Language"/> <xs:element name="Content" type="xs:string"/> </xs:sequence> </xs:complexType>

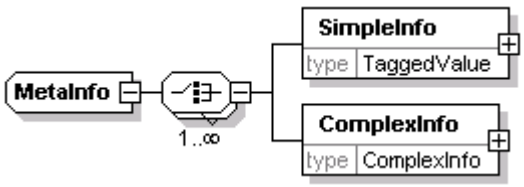
element **LanguageText/Language**

diagram	
type	**Language**
facets	enumeration English (USA) enumeration English (UK) enumeration German enumeration French enumeration Japanese enumeration Spanish enumeration Portugese enumeration Dutch
source	<xs:element name="Language" type="Language"/>

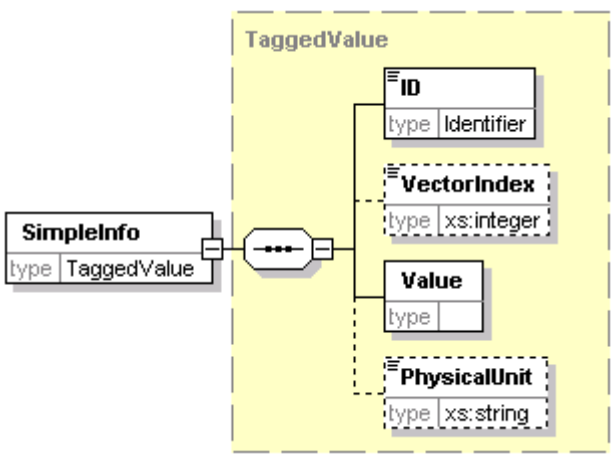
element **LanguageText/Content**

diagram	
type	**xs:string**
source	<xs:element name="Content" type="xs:string"/>

complexType **MetaInfo**

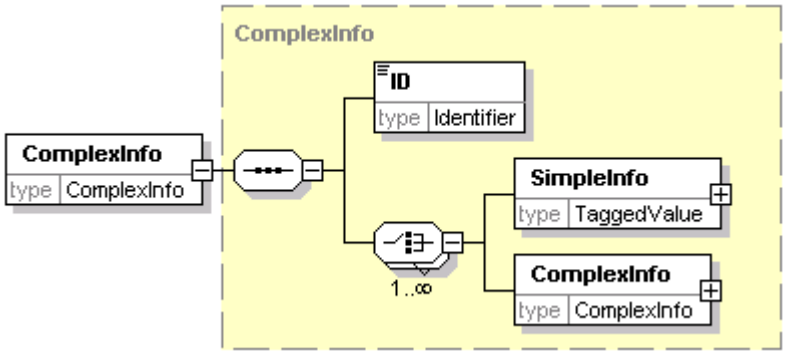
diagram	
children	**SimpleInfo** **ComplexInfo**
used by	elements **UMEOinst/MetaInfo** **UMEOinst/Instance/MetaInfo**
source	<xs:complexType name="MetaInfo"> <xs:choice maxOccurs="unbounded"> <xs:element name="SimpleInfo" type="TaggedValue"/> <xs:element name="ComplexInfo" type="ComplexInfo"/> </xs:choice> </xs:complexType>

element **MetaInfo/SimpleInfo**

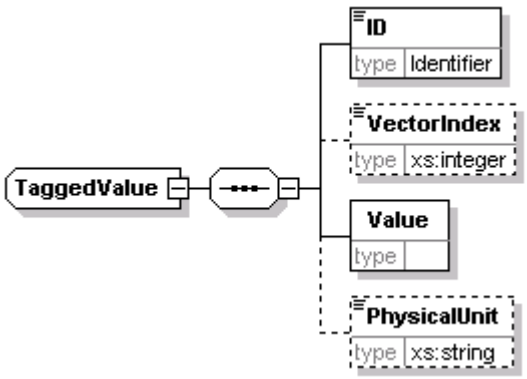
diagram	
type	**TaggedValue**

children	**ID VectorIndex Value PhysicalUnit**
source	<xs:element name="SimpleInfo" type="TaggedValue"/>

element **MetaInfo/ComplexInfo**

diagram	
type	**ComplexInfo**
children	**ID SimpleInfo ComplexInfo**
source	<xs:element name="ComplexInfo" type="ComplexInfo"/>

complexType **TaggedValue**

diagram	
children	**ID VectorIndex Value PhysicalUnit**
used by	elements **Identity/AdditionalID MetaInfo/SimpleInfo ComplexInfo/SimpleInfo UMEOinst/Instance/SimpleParameter InstComplexParam/SimpleValue**
source	<xs:complexType name="TaggedValue"> <xs:sequence> <xs:element name="ID" type="Identifier"/> <xs:element name="VectorIndex" type="xs:integer" minOccurs="0"/> <xs:element name="Value"/> <xs:element name="PhysicalUnit" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType>

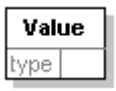
element **TaggedValue/ID**

diagram	
type	**Identifier**
source	<xs:element name="ID" type="Identifier"/>

element **TaggedValue/VectorIndex**

diagram	
type	**xs:integer**
source	<xs:element name="VectorIndex" type="xs:integer" minOccurs="0"/>

element **TaggedValue/Value**

diagram	
source	<xs:element name="Value"/>

element **TaggedValue/PhysicalUnit**

diagram	
type	**xs:string**
source	<xs:element name="PhysicalUnit" type="xs:string" minOccurs="0"/>

simpleType **Identifier**

type	**xs:string**
used by	elements **UMEOinst/Instance/ClassID** **Identity/FullID** **UMEOinst/Instance/Role/ID** **TaggedValue/ID** **InstComplexParam/ID** **ComplexInfo/ID** **UMEOinst/Instance/Role/RolePartner/InstanceID**
annotation	documentation Datatype to describe IDs, based on xs:string
source	<xs:simpleType name="Identifier"> <xs:annotation> <xs:documentation> Datatype to describe IDs, based on xs:string</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"/> </xs:simpleType>

simpleType **InstanceType**

type	restriction of **xs:string**
used by	elements **UMEOinst/Instance/InstanceType** **UMEOinst/Instance/Role/RolePartnerInstanceType**
facets	enumeration EOinstance enumeration EORMinstance
annotation	documentation Enumerated {EOinstance, EORMinstance}
source	<xs:simpleType name="InstanceType"> <xs:annotation> <xs:documentation> Enumerated {EOinstance, EORMinstance}</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"> <xs:enumeration value="EOinstance"/> <xs:enumeration value="EORMinstance"/> </xs:restriction> </xs:simpleType>

simpleType **Language**

type	restriction of **xs:string**
used by	element **LanguageText/Language**
facets	enumeration English (USA)

	enumeration English (UK) enumeration German enumeration French enumeration Japanese enumeration Spanish enumeration Portugese enumeration Dutch
annotation	documentation Enumerated {English (USA), English (UK), German, French, Japanese, Spanish, Portugese, Dutch}
source	<xs:simpleType name="Language"> <xs:annotation> <xs:documentation> Enumerated {English (USA), English (UK), German, French, Japanese, Spanish, Portugese, Dutch}</xs:documentation> </xs:annotation> <xs:restriction base="xs:string"> <xs:enumeration value="English (USA)"/> <xs:enumeration value="English (UK)"/> <xs:enumeration value="German"/> <xs:enumeration value="French"/> <xs:enumeration value="Japanese"/> <xs:enumeration value="Spanish"/> <xs:enumeration value="Portugese"/> <xs:enumeration value="Dutch"/> </xs:restriction> </xs:simpleType>

Anhang E

OMG CAD SERVICES

In diesem Abschnitt werden die CADServices Adapter ausführlich erläutert, da sie im Framework (vgl. Abbildung 62) eine bedeutende Rollte spielen im Hinblick auf den Zugriff auf die B-Rep Geometrie des Produktmodells.

Die Schnittstellendefinition der CAD Services besteht aus acht Modulen (Namespaces):

- **CadConnection,**
- **CadMain,**
- **CadFoundation,**
- **CadGeometry,**
- **CadBrep,**
- **CadFeature,**
- **CadUtility** und
- **CadGeometryExtens.**

In einem zusätzlichen Modul `CadError` werden die Exceptions (Ausnahmen) zur Fehlerbehandlung definiert. Die einzelnen Module (außer CadError) werden nachfolgend genauer betrachtet. Für jedes Modul existiert ein UML–Diagramm. Die Umsetzung der in IDL abgefassten Spezifikationen ist in den verschiedenen Programmiersprachen unterschiedlich. Tabelle 1 gibt einen Überblick.

IDL	C++	Java
Module	Namespace	Namespace
Struct	Struct	
Interface	Class	Interface
Class	Class	Class
Exception	Exception oder Struct	Exception

Tabelle 1: Umsetzung von IDL in C++

Das Modul CadConnection

Dieses Modul implementiert eine Verbindung auf hoher Abstraktionsebene zum Interface `Model` aus dem Modul `CadMain`. Das `CadServer` Interface ist hierbei das erste Interface das von einem Client benutzt wird. Es wird verwendet, um eine Verbindung zum `CadSystem` Interface zu bekommen. Das `CadServer` Interface hat zwei Attribute, die das zugrunde liegende CAD-System genauer spezifizieren und die Parameter festlegen, die zu dessen Start notwendig sind. Abbildung 1 zeigt den Abhängigkeitsgraph des Moduls CadConnection.

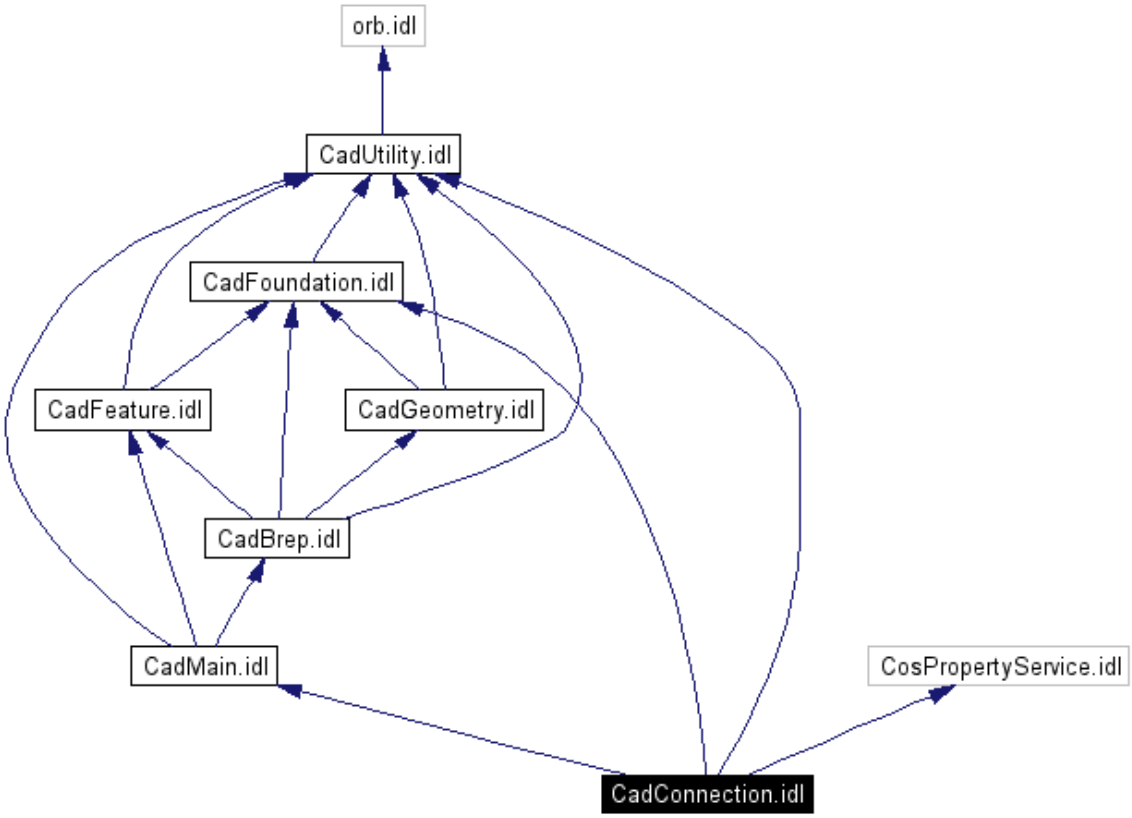


Abbildung 1: Abhängigkeitsgraph Modul CadConnection

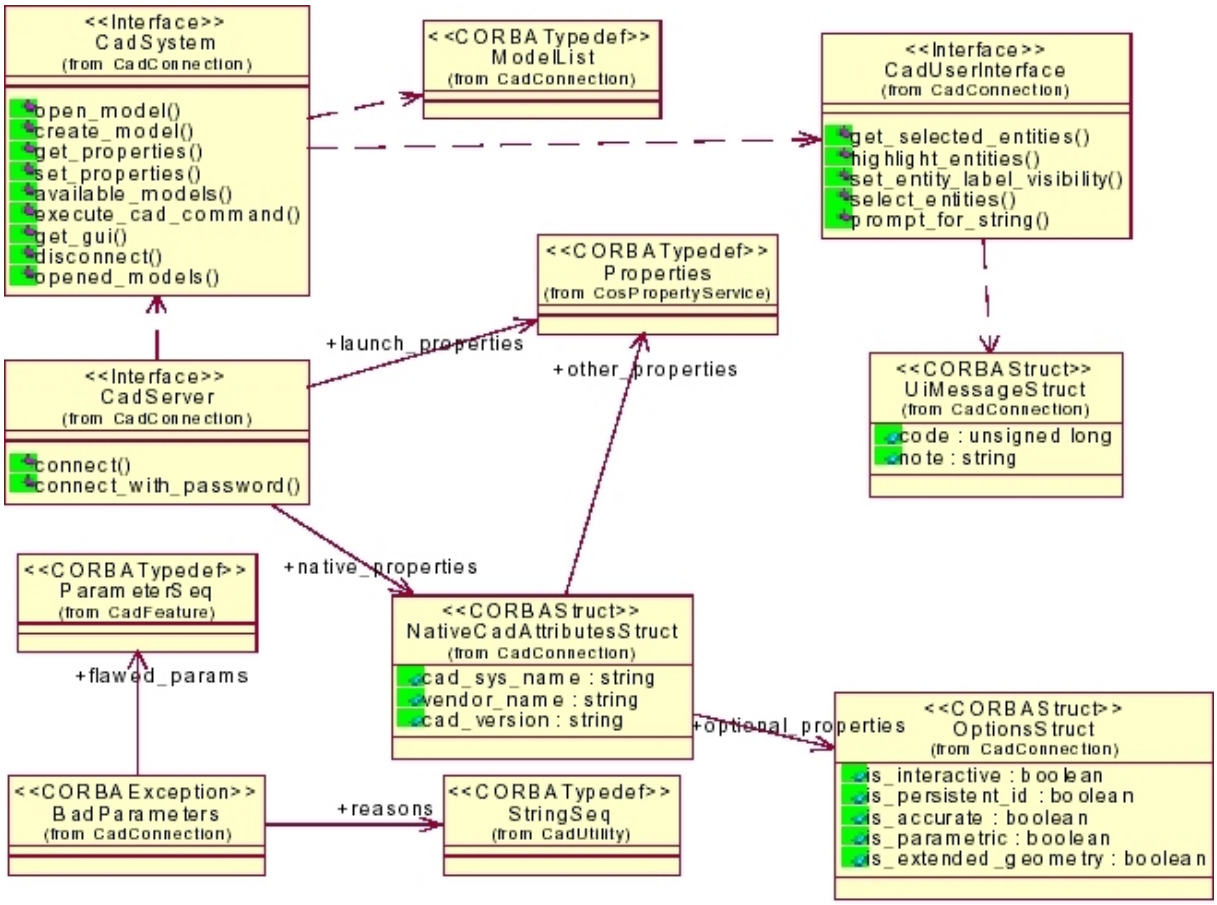


Abbildung 2: UML Diagramm CadConnection

Interface CadServer

```
interface CadServer
  {
    readonly attribute NativeCadAttributesStruct native_properties;
    // NativeCADSys contains CAD Vendor name, version, etc.

    readonly attribute CosPropertyService::Properties  launch_properties;
    // This attribute contains all needed information to launch
    // the CAD system

    CadSystem connect( in CosPropertyService::Properties props)
        raises ( CadConnectionFault );
    // secure connection

    CadSystem connect_with_password(  in string user, in string password,
        in CosPropertyService::Properties props )
        raises ( CadConnectionFault );
    // open wire (unsecure connection)

  };
```

Attribute

`native_properties`	Eigenschaften des implementierten CAD-Systems

launch_properties	Spezielle Eigenschaften für den Aufruf des CAD-Systems

Methoden

connect	Aufbau einer Verbindung mit einem CAD-System
connect_with_password	Aufbau einer sicheren Verbindung zu einem CAD-System mit Authentifizierung

Interface CadSystem

```
interface CadSystem
 {
    CadMain::Model  open_model(in string model, in ActivationMode access)
        raises (InvalidModel, PermissionDenied, CadUtility::CadError);
    // CAD model related operations

    CadMain::Model create_model(in string new_name,
        in CadUtility::MassUnit m_unit,in CadUtility::LengthUnit l_unit,
        in CosPropertyService::Properties model_params)
        raises (PermissionDenied, BadParameters, CadUtility::CadError);
    // Creates and opens a new model in native CAD system

    CosPropertyService::Properties get_properties();

    void set_properties( in CosPropertyService::Properties props );
    // Allows reading and changing of CadSystem properties

    ModelList  available_models();
    // returns a list if model available to the active  CAD System.

    CadMain::ModelSeq opened_models();
    // returns a list of active models

    void execute_cad_command(in string command_string , inout any comm_out )
       raises (BadCommand);
    // extensible Cad system command interface

    CadUserInterface get_gui() raises (CadFoundation::GuiUnsupported);
    // access to GUI interface

    void disconnect() raises ( CadConnectionFault );
    // close system and clean-up

    // v 1.2 additions for directory information
    // Creates a new folder in a given directory.
    // param parent_dir: The (canonical, absolute, and unique) path of the
    // directory, in which the folder will be created. This must be either
    // one of the root directories (see get_root_directories()) or a
    // directory that exists in one of the root directory's subdirectories
    // param folder_name: The name of the folder to be created. It must be a
    // valid and a not already existing name.
    // raises CadUtility::CadError: If there was an error in creating the
```

```
    // folder.
    void create_new_folder(in string parent_dir, in string folder_name)
    raises (CadUtility::CadError);

    // Returns a list of (canonical, absolute, and unique) pathnames repre-
    // senting the root directories. In a root directory or in any other
    // subdirectory, model files can be found.
    // return: List of pathnames representing root directories.
    RootsList get_root_directories();

    // Returns a list of subdirectories and model files of a given directory.
    // param directory: (canonical, absolute, and unique) path of the
    // directory from which to get the list of subdirectories and model
    // files. This must be either one of the root directories or a subdirectory.
    // return: A list representing subdirectories and/or model files
    // of the given directory.
    // raises CadUtility::CadError If there was an error retrieving the list.
    PathList get_models_and_folders(in string directory)
           raises (CadUtility::CadError);

    // Returns the parent directory of a given directory.
    // param directory: (canonical, absolute, and unique) path of the
    // directory from which to get its parent directory. The given directory
    // should be a subdirectory of one of the root directories.
    // return: The path of the parent directory.
    // raises CadUtility::CadError If there was an error retrieving the
    // parent directory.
    string get_parent_directory(in string directory)
          raises (CadUtility::CadError);
    };
```

Methoden

open_model	Öffnet ein vorhandenes Modell. Eingabeparameter ist der Modellname
create_model	Erzeugen eines neuen Modells im CAD-System
get_properties	Ermitteln der Eigenschaften des Modells
set_properties	Setzen der Eigenschaften des Modells
available_models	Ermitteln der zur Verfügung stehenden Modelle
opened_models	Ermitteln der geöffneten Modelle
execute_cad_command	Ausführen eines Kommandos im nativen CAD-System

get_gui	Erlaubt den Zugriff auf die Benutzungsschnittstelle
disconnect	Trennen der Verbindung
create_new_folder	Erzeugen eines neuen Ordners
get_root_directories	Ermitteln des in der Verzeichnis-Hierarchie am weitesten oben liegenden Ordners, der Modelle enthält
get_models_and_folders	Ermitteln der Modelle und deren Verzeichnisse
get_parent_directory	Ermittelt das übergeordnete Verzeichnis des Verzeichnisses das als Parameter übergeben wird

Das Modul CadMain

Das Modul CADMain kapselt alle Interfaces die CAD-Modelle beschreiben. Es besteht aus dem Inteface für Modelle und zwei Factory Intefaces um neue Instanzen zu erzeugen.

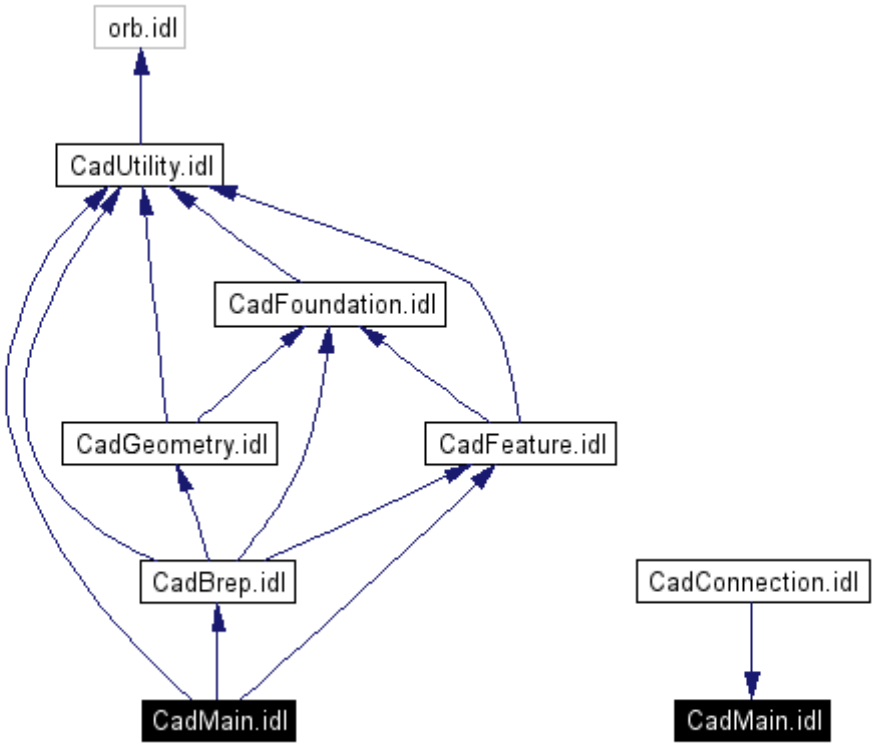


Abbildung 3: Abhängigkeitsgraph Modul CadMain

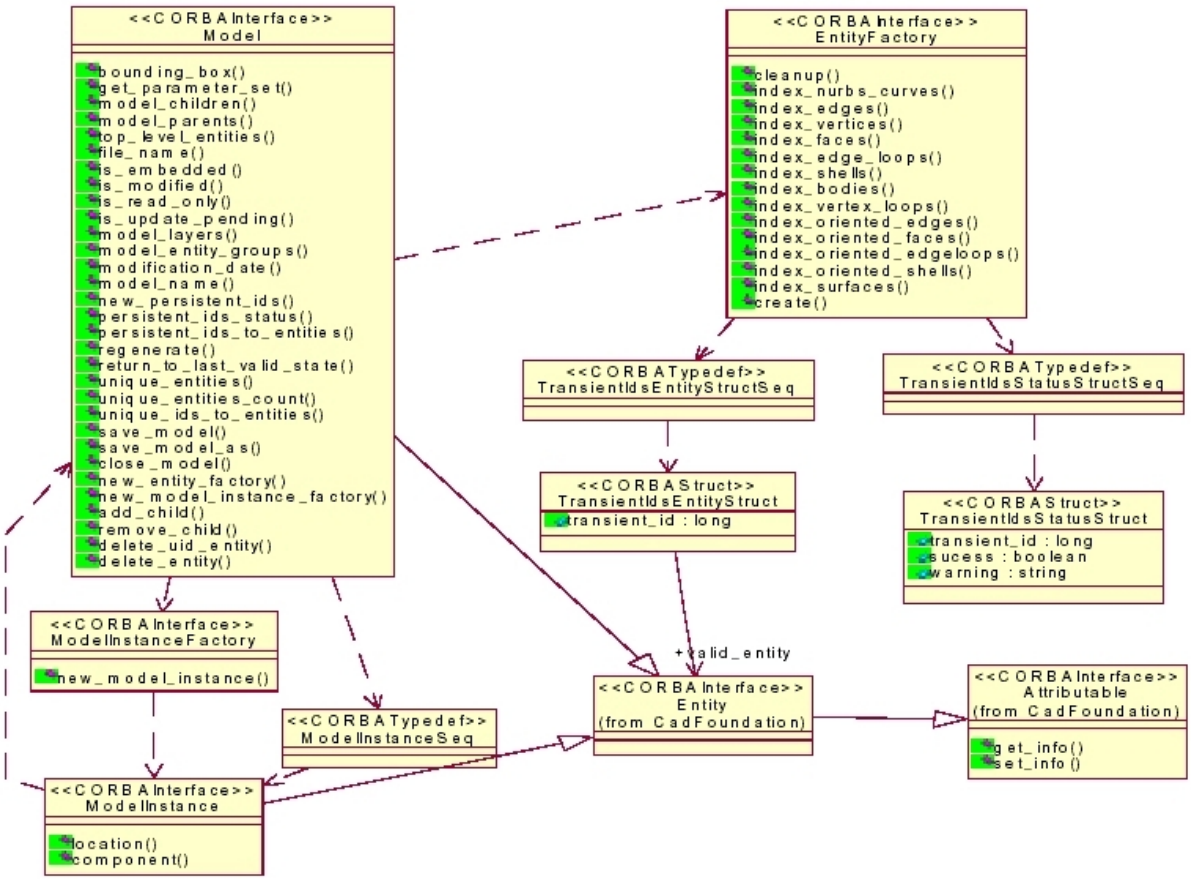


Abbildung 4: UML Diagramm CadMain

Interface Model

```
interface  Model : CadFoundation:: Entity
  {
    // An aggregation of all entities and high-level behaviors that represent
    // a single CAD model.  Includes product structure, boundary representations,
    // geometric entities, features, text entities, and datums.   All entities within
    // a CAD model are arcwise connected unless related to each other through an
    // instance.

    readonly attribute CadUtility::MassUnit  mass_unit;
    readonly attribute CadUtility::LengthUnit length_unit;
    // Defines units used in the Model. Angles use degrees

    CadUtility::BoundingBox     model_bounding_box     (in     CadUtility::TypeCodeSeq
entity_types)
                raises (UnboundedEntity, NotValidCadType, CadUtility::CadError);
    // Returns an approximate BoundingBox around all entities of the
    // specified type(s)
    // in the model.

    CadFeature::ParameterSeq get_parameter_set() raises (CadUtility::CadError);
    // Returns a sequence of parammeters for this model.

    ModelInstanceSeq model_children() raises (CadUtility::CadError);
    // Returns a sequence of any ModelInstances contained in this model.

    ModelInstanceSeq model_parents() raises (CadUtility::CadError);
    // Returns a sequence of parent models.

    CadFoundation::EntitySeq top_level_entities (
        in CadUtility::TypeCodeSeq entity_types)
        raises (NotValidCadType, CadUtility::CadError);
    // Returns a sequence of the top level entities of the specified
    // type(s).

    string file_name () raises (CadUtility::CadError);
    // Returns the complete name, including absolute path (if possible),
    // of the physical file that stores this model.
    // Returns an empty string if the model is not defined in a file.

    boolean is_embedded();
    // indicates if Model is an embedded part (e.g. CATIA Dittos,
    // ACAD blocks, etc) or a non-embedded part (e.g. parts in a ProE assembly).

    boolean is_modified () raises (CadUtility::CadError);
    // Queries if the model has been modified since last saved.

    boolean is_read_only ()  raises (CadUtility::CadError);
    // Queries whether the model can be modified.

    boolean is_update_pending () raises (CadUtility::CadError);
    // Queries if the model is being updated (regenerated).

    CadFoundation::LayerSeq model_layers () raises (CadUtility::CadError);
    // Returns a sequence of the Layers defined in this model.

    CadFoundation::EntityGroupSeq   model_entity_groups   (out   CadUtility::StringSeq
group_names)
        raises (CadUtility::CadError);
    // Returns a sequence of the EntityGroups defined in this model.

    string modification_date () raises (CadUtility::CadError);
    // Returns the date and time the model was last modified.

    string model_name () raises (CadUtility::CadError);
    // Returns the user-interpretable name of this model.
```

```
    CadUtility::StringSeq      new_persistent_ids      (in      CadUtility::StringSeq
persistent_ids)
         raises (CadFoundation::PidUnsupported, CadUtility::CadError);
    // Returns a sequence of new persistent ID's for any entities not referenced in
    // the specified sequence of persistent ID's (e.g. new or modified IDs).

    PidStatusSeq persistent_ids_status (in CadUtility::StringSeq persistent_ids)
         raises (CadFoundation::PidUnsupported, CadUtility::CadError);
    // Returns whether the entities a sequence of persistent IDs reference are
    // unmodified, modified, deleted.  The returned sequence of status enumerations
    // are in the same order as the input sequence of persistent IDs.

    CadFoundation::EntitySeq persistent_ids_to_entities (
        in CadUtility::StringSeq persistent_ids)
         raises (CadFoundation::PidUnsupported, CadUtility::CadError);
    // Returns a sequence of entity objects corresponding to a sequence of persistent
    // IDs.
    // The returned sequence of entity objects is in the same order as the
    // input sequence of persistent IDs.  A NULL item in this sequence means
    // an entity was not available
    // for the corresponding persistent ID.

    void regenerate () raises (RegenerationException, CadUtility::CadError);
    //  If  any  DesignFeatures  exist  and  have  been  modified,  forces  a  complete
regeneration
    // of modified Entities in the model.  Otherwise this operation does nothing.
    // Throws an exception if the regeneration process is unsuccessful.

    void        return_to_last_valid_state()        raises        (ReturnToValidFail,
CadUtility::CadError);
    // Returns the model and all entities it contains to their state just after
    // the last
    // successful regeneration or after initially opening the model.
    // Throws an exception if unable to return to a valid state.

    CadFoundation::EntitySeq unique_entities (
        in CadUtility::TypeCodeSeq entity_types)
        raises (NotValidCadType, CadUtility::CadError);
    // Returns a sequence of the unique entities of the specified
    // type(s).

    unsigned long unique_entities_count (in CadUtility::TypeCodeSeq entity_types)
          raises (NotValidCadType, CadUtility::CadError);
    // Returns the count of entities of the specified type(s) in this model.

    CadFoundation::EntitySeq    unique_ids_to_entities    (in    CadUtility::LongSeq
unique_ids)
         raises (NotValidUid, CadUtility::CadError);
    // Returns (in sequental order) a sequence of entities corresponding to an input
sequence
    // of unique IDs.

    void    save_model() raises (SaveFault);
    void    save_model_as(in string new_name) raises (SaveAsFault);
    void    close_model() raises ( CloseFault );
    // operations for saving and terminating  an active session

    EntityFactory new_entity_factory () raises (CadUtility::CadError);
    // Entity creation factory interface - called to create new CAD entities in
current model

    ModelInstanceFactory new_model_instance_factory () raises (CadUtility::CadError);
   //Creates the ModelInstanceFactory, which is used to add ModelInstances

    void add_child(in ModelInstance child_model) raises (CadUtility::CadError);

    void remove_child(in ModelInstance child_model) raises (CadUtility::CadError);

    void delete_uid_entity(in long uid)
```

```
        raises (EntityOutOfModel, CadUtility::CadError);
    // Removes ModelInstance, BrepEntity, Curve or Surface from the model

    void delete_entity(in CadUtility::LongSeq entities_uids)
        raises (EntityOutOfModel, CadUtility::CadError);
    // Removes model entities from the model
    // v1.2 change from EntitySeq to LongSeq - simplifies mapping
  };
```

Attribute

`mass_unit`	Gewichtseinheit.
`length_unit`	Masseneinheit.

Methoden

`model_bounding_box()`	Maximale Abmessungen des Modells in Form eines Quaders.
`get_parameter_set()`	Liste mit Parameter, die dieses Modell bestimmen.
`model_children()`	Liste von ModelInstances, die in diesem Modell enthalten sind.
`model_parents()`	Gibt die Objekte an zu denen dieses Modell gehört.
`toplevel_entities()`	Gibt die Entities der höchsten Hierarchiestufe zurück.
`filename()`	Dateiname auf einem physischen Datenträger.
`is_update_pending()`	Test, ob Modell regeneriert werden muss.
`is_readonly()`	Nur-Lesen Zugriff.
`is_modified()`	Test ob Modell modifiziert worden ist.
`is_embedded()`	Test, ob Modell ein „embedded-Part“ ist.

`model_layers()`	Liste der vorhandenen Layer im Modell.
`model_entity_groups()`	Ermitteln von logischen Entity-Gruppen.
`modification_date()`	Datum, wann das Modell zuletzt modifiziert worden ist.
`model_name()`	Name des Modells.
`new_persistent_ids()`	Vergabe von PIDs falls keine vorhanden.
`persistent_ids_status()`	Status der PIDs.
`persistent_ids_to_entities()`	Iterator über alle Entities des Modells – die Selektion findet dabei über die angegebenen PID statt.
`regenerate()`	Regeneriert das Modell.
`return_to_last_valid_state()`	Versetzt das Modell in den letzten validen Status zurück.
`unique_entities()`	Liste der Entities die dieses Modell beinhaltet.
`unique_entities_count()`	Anzahl dieser Entities.
`unique_ids_to_entities()`	Finden eines Entity aufgrund seiner ID.
`save_model()`	Speichern des Modells.
`save_model_as()`	Speichern des Modells unter einem bestimmten Pfad.
`close_model()`	Schließen des Modells.
`new_entity_factory()`	Erzeugen eines Factory Objektes für `ModelInstances`.

add_child()	Hinzufügen einer ModelInstance.
remove_child()	Löschen einer ModelInstance.
delete_uid_entity()	Löschen einer ModelInstance anhand der PID.
delete_entity()	Löschen einer Entity.

Interface ModelInstance

```
interface ModelInstance : CadFoundation:: Entity
  {
    CadUtility::TransformationStruct location() raises (CadUtility::CadError);
    // returns location information

    Model component_model() raises (CadUtility::CadError);
    // Returns the Model that defines this instance.
    // minor change for V 1.2 - component becomes component_model


  };
```

Methoden

location()	Position und Orientierung der ModelInstance.
component_model()	Pointer auf die ModelInstance.

Interface EntityFactory

EntityFactory wird benutzt, um Instanzen von Objekten zu erzeugen.

```
interface EntityFactory
  {
    void cleanup()  raises (CadUtility::CadError);
    //clean-up any expensive book-keeping following multiple index_xxxx(), create()
cycles

    CadUtility::LongSeq index_nurbs_curves(in CadUtility::NurbsCurveStructSeq nurbs)
         raises (CadUtility::CadError);
    CadUtility::LongSeq  index_edges  (in  CadUtility::LongSeq  start_vertices,  in
CadUtility::LongSeq end_vertices,
        in CadUtility::LongSeq curve, in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_vertices(in CadUtility::PointStructSeq pts)
         raises (CadUtility::CadError);
    CadUtility::LongSeq index_faces(in CadUtility::LongSeqSeq oriented_eloops,
        in CadUtility::LongSeqSeq vertex_loops, in CadUtility::LongSeq surfaces)
```

```
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_edge_loops(in CadUtility::LongSeqSeq oriented_edges)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_shells (in CadUtility::LongSeqSeq oriented_faces)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_bodies (in CadUtility::LongSeqSeq oriented_shells)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_vertex_loops (in CadUtility::LongSeq vertices)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq   index_oriented_edges(in   CadUtility::LongSeq   edges,   in
CadUtility::BooleanSeq sense)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_faces(in CadUtility::LongSeq faces,
        in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_edgeloops(in CadUtility::LongSeq edgeloops,
        in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_shells(in CadUtility::LongSeq shells,
        in CadUtility::BooleanSeq sense)
        raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_surfaces( in CadUtility::NurbsSurfaceStructSeq nurbs)
         raises (CadUtility::CadError);

    void create (out TransientIdsStatusStructSeq status_flags,
        out TransientIdsEntityStructSeq final_entities)
         raises (CadUtility::CadError);
    // final creation step
  };
```

Das Modul CadFoundation

Das Modul definiert alle grundlegenden Strukturen von denen weitere Interfaces abgeleitet sind.

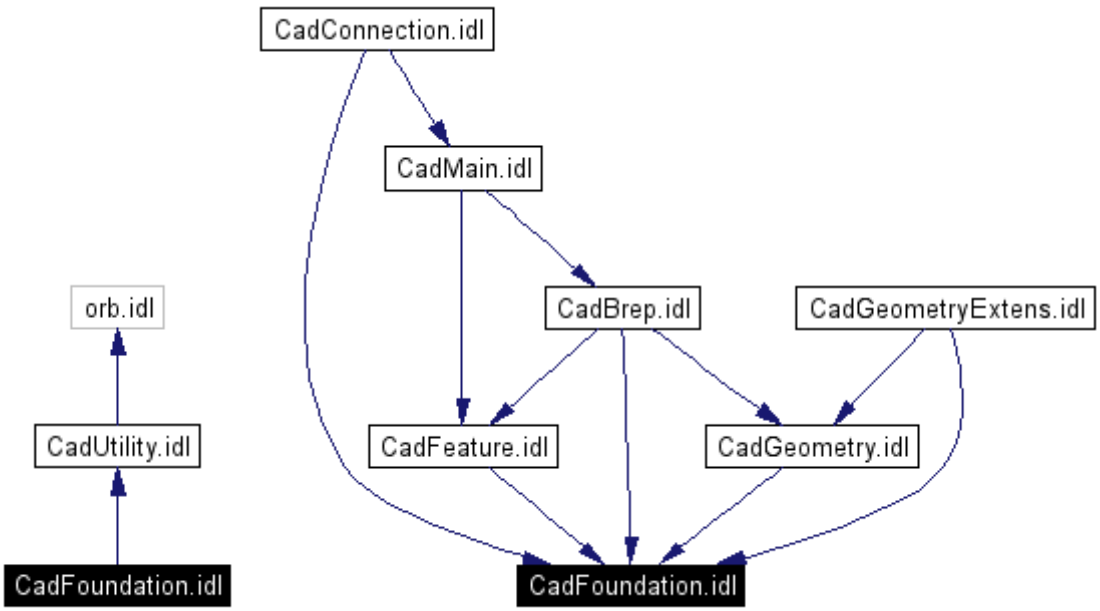


Abbildung 5: Abhängigkeitsgraph Modul CadFoundation

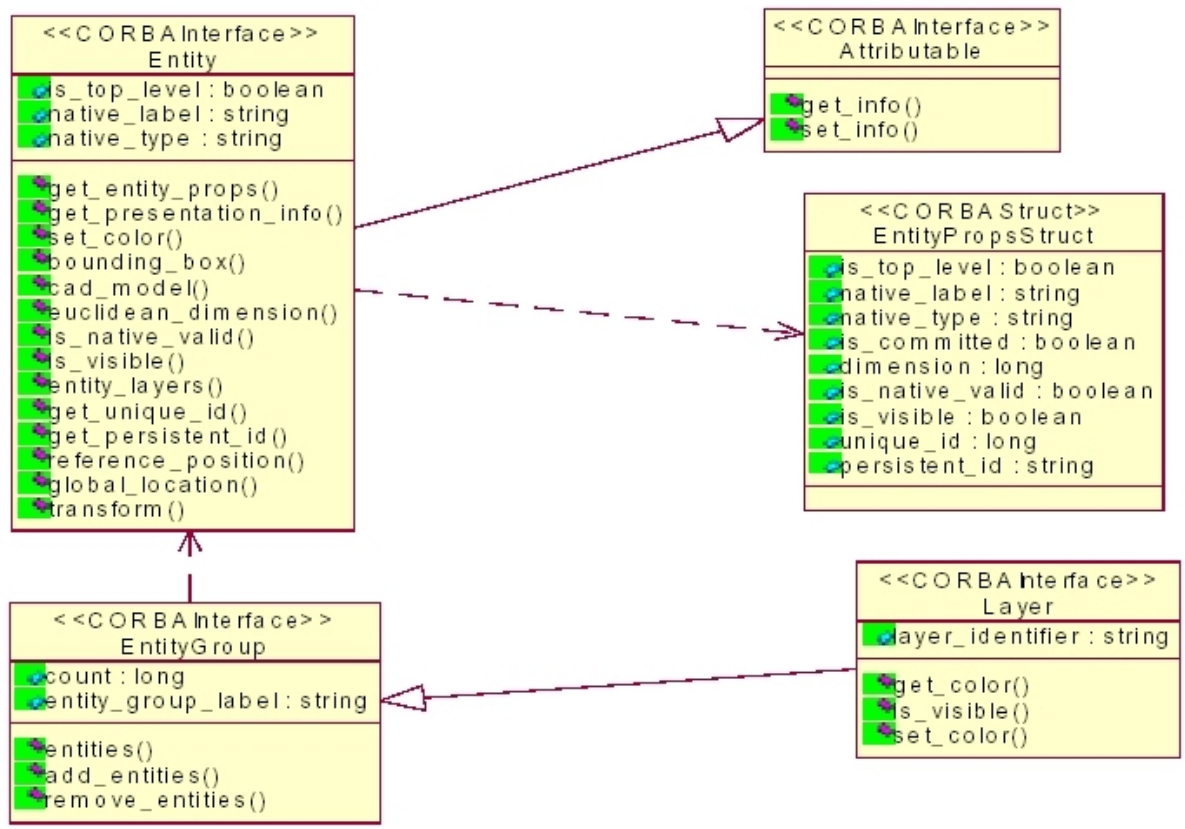


Abbildung 6: UML Diagramm CadFoundation

Interface Entity

Basis-Interface für alle Objekte des topologisch-geometrischen Strukturbaumes.

```
interface Entity : Attributable
 {
    // Provides CAD functionality + properties to be inherited by geometry objects
    EntityPropsStruct get_entity_props() raises (CadUtility::CadError);
    // operation providing grouped access to Entity properties

    readonly attribute boolean is_top_level;
    // Top level entity?

    readonly attribute string native_label;
    // Provides a brief description of the entity using system-specific terminology.
    // Not guaranteed to be unique within a model.

    readonly attribute string native_type;
    // The system-specific type name of this entity.

    CadUtility::PresentationStruct get_presentation_info()
         raises (CadUtility::CadError);
    // Struct containing relevant presentation information.

    void set_color(in CadUtility::ColorStruct  color) raises (CadUtility::CadError);
    //  set color

    CadUtility::BoundingBox     bounding_box     ()     raises     (UnBoundedEntity,
CadUtility::CadError);
    // Returns an approximate BoundingBox around the entity.
    // Returns an error if the entity is unbounded in one or more directions.
```

```
    Object  cad_model() raises (CadUtility::CadError);
    // Returns the CadMain::Model object that contains this entity.
    // Reference must be narrowed to CadMain::Model

    long euclidean_dimension ()  raises (CadUtility::CadError);
    // Returns the Euclidean dimension of the entity.

    boolean is_native_valid () raises (CadUtility::CadError);
    // Queries if the native entity is valid according to any internal checks
    // provided by the CAD system.

    boolean is_visible ()  raises (CadUtility::CadError);
    // Queries whether the entity is visible or not (blanked, no-showed, hidden).

    LayerSeq entity_layers ()  raises (CadUtility::CadError);
    // Returns a sequence of the Layers that contain this entity

    long get_unique_id() raises ( CadUtility::CadError );
    // Identifier that is guaranteed to be unique across all entities in a Model.
    // This identifier is not persistent (i.e.valid only during CadServer Session).

    string get_persistent_id () raises (PidUnsupported, CadUtility::CadError);
    // Returns an identifier intended to identify this entity between
    // interface sessions.

    CadUtility::PointStructSeq reference_position () raises (CadUtility::CadError);
    // Returns a sequence of reference coordinates on
    // the entity that are unique relative to neighboring entities.
    // Provided as a convenience for client graphics and other tagging applications

    CadUtility::TransformationStruct global_location() raises (CadUtility::CadError);
    // Provides global coordinate location information

    void transform (in CadUtility::TransformationStruct  transformation)
        raises (NotIndependent, ReadOnlyEntity, CadUtility::CadError);

    // Applies the specified transformation (rotations and translation)
    // to the entity.
    // Throws an exception if the entity cannot be transformed.

  };
```

Attribute

`is_toplevel`	Flag, ob das Entity ein toplevel-Entity ist.
`native_label`	Label des nativen CAD-Systems.
`native_type`	Entity-Typ im nativen CAD-System.

Methoden

`get_entity_props()`	Zugriff auf alle Properties des Entity.

`get_presentation_info()`	Informationen zur Darstellung des Objektes.
`set_color()`	Objektfarbe.
`bounding_box()`	Dimensionen in Form eines Quaders, die das Objekt begrenzen.
`cad_model()`	Zugehöriges `Model`.
`euclidian_dimension()`	Euklidischen Abmessungen des Modells.
`is_native_valid()`	Test ob Modell im Ursprungs-CAD-System valide ist.
`is_visible()`	Test auf Sichtbarkeit.
`entity_layers()`	Ermitteln der verbundenen Layer.
`get_unique_id()`	Ermitteln der UID diese ist nur innnerhalb dieses Modells eindeutig.
`get_persistent_id()`	Ermitteln der PID. Diese ist session-übergreifend.
`reference_position()`	Ermitteln der Referenz-Position relativ zu anderen Entities.
`global_location()`	Globale Position.
`transform()`	Anwenden einer definierten Transformation.

Interface Attributable

Interface für einfache textuelle Attribute an Entities.

```
interface Attributable
  {
    // General interface allowing geometry tagging
    // The following operations should use DynAnys to extract attribute information

       any  get_info() raises (CadUtility::CadError);
       void set_info(in any dyn_value) raises (CadUtility::CadError);
   };
```

Methoden

`get_info()`	Ermitteln von Informationen die mit einer Entity verbunden sind.
`set_info()`	Definieren von Informationen die mit einer Entity verbunden sind.

Interface EntityGroup

Implementiert die hierarchische Gruppierung von Entities.

```
interface EntityGroup
  {
    // A generalized grouping of entities within a model that is not related to
    // layering. Provides a mechanism for grouping whose semantics lie outside the
    // standard, e.g. application-specific collections

    readonly attribute long count;
    // Number of entities in group.

    readonly attribute string entity_group_label;
    // a label for this entity group

    CadFoundation::EntitySeq entities ()  raises (CadUtility::CadError);
    // Returns a sequence of entities defined in this group.

    void add_entities (in CadUtility::LongSeq entities_uids)
         raises (CadUtility::CadError);
    // Adds the specified entities to this group.
    // v1.2 use unique ids as opposed to entities

    void remove_entities (in CadUtility::LongSeq entities_uids)
        raises (CadUtility::CadError);
    // Removes the specified entities from this group.
    // Does not delete the entity objects.
    // v1.2 use unique ids as opposed to entities
  };
```

Attribute

`count`	Anzahl der Entities in der Gruppe.
`entity_group_label`	Text-Label der Gruppe.

Methoden

`entities()`	Iterator über alle Entities der Gruppe.

`add_entities()`	Hinzufügen eines Entity.
`remove_entities()`	Löschen eines Entity.

Interface Layer

```
interface Layer : EntityGroup
  {
    // An collection of entities that corresponds to the layers.

    readonly attribute string layer_identifier;
    // String identifier of layer.

    CadUtility::ColorStruct get_color()
         raises (CadUtility::CadError);

    void set_color( in CadUtility::ColorStruct new_color);
    // change the color of all entities in this layer

    boolean is_visible() raises (CadUtility::CadError);

  };
```

Attribute

`layer_identifier`	Label für den Layer.

Methoden

`get_color()`	Farbcode des Layers.
`set_color()`	Setzen der Layer-Farbe.
`is_visible()`	Flag für die Sichtbarkeit.

Das Modul CadGeometry

CadGeometry beinhaltet alle Interface Definitionen der geometrischen Objekte des topologisch geometrischen Strukturmodells, wie z. B. Punkte und Vektoren. Die einzelnen Interfaces werden wegen ihrer Einfachheit hier nicht aufgeführt.

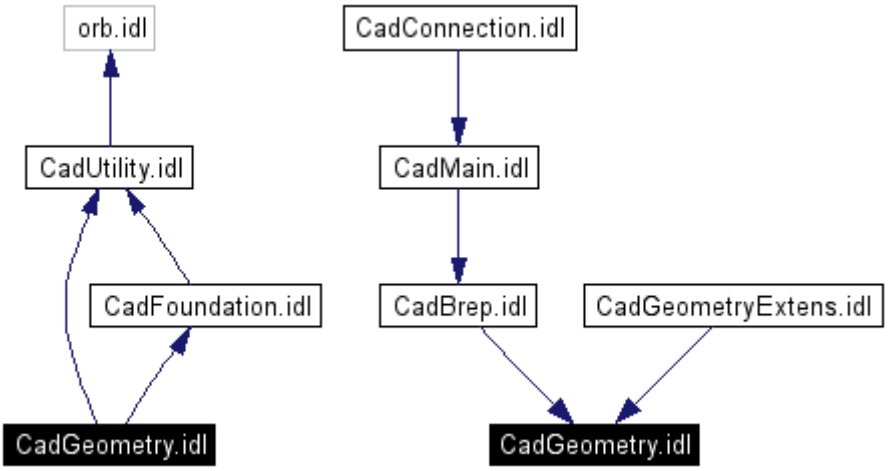


Abbildung 7: Abhängigkeitsgraph Modul CadGeometry

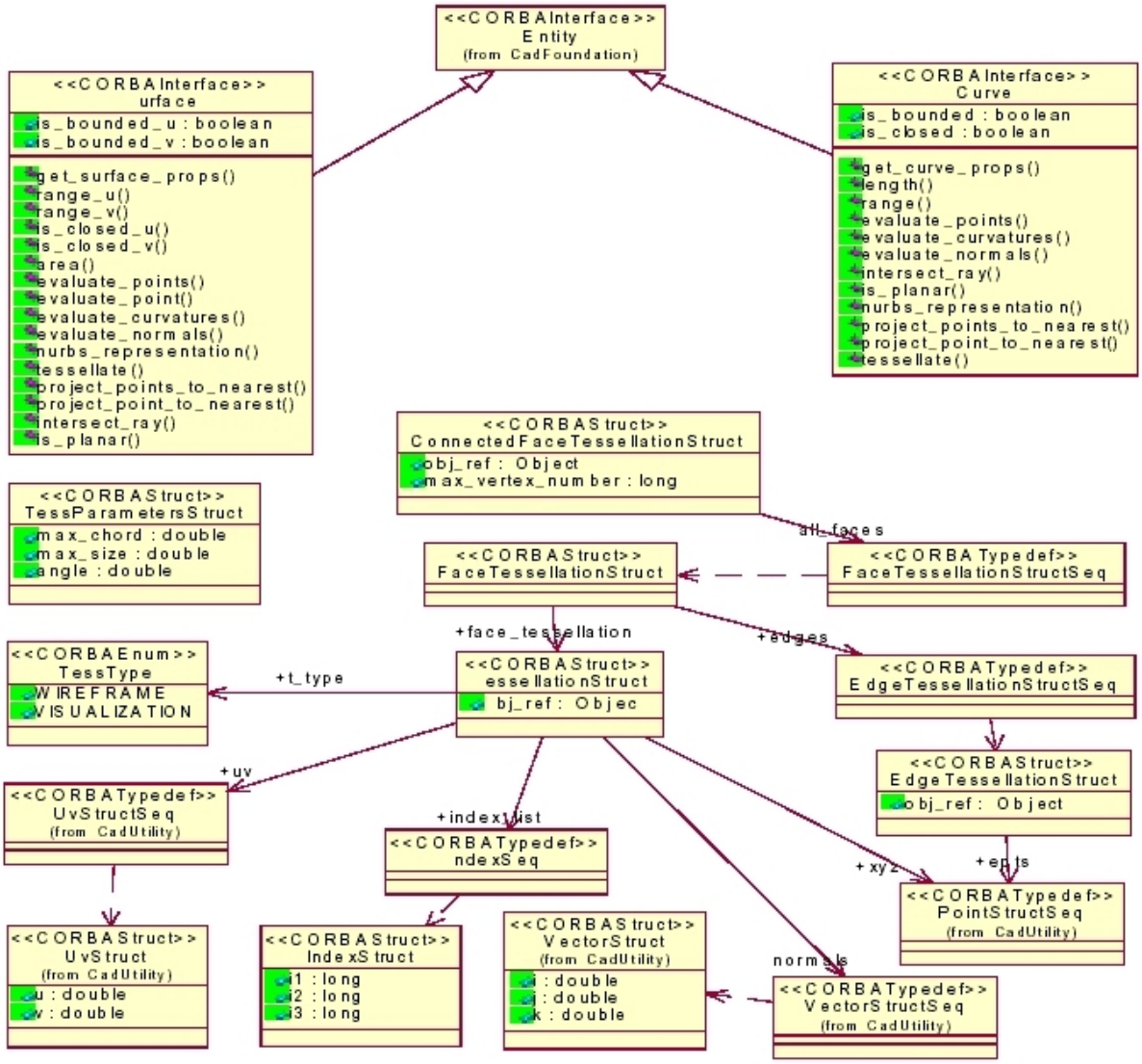


Abbildung 8: UML Diagramm CadGeometry

Das Modul CadBrep

Das Modul `CadBrep` bildet das Interface für die topologisch-geometrischen Strukturmodelle. Sie bestehen aus einer verlinken Struktur von Topologie-Elementen, die mit ihrer jeweiligen geometrischen Struktur verknüpft sind.

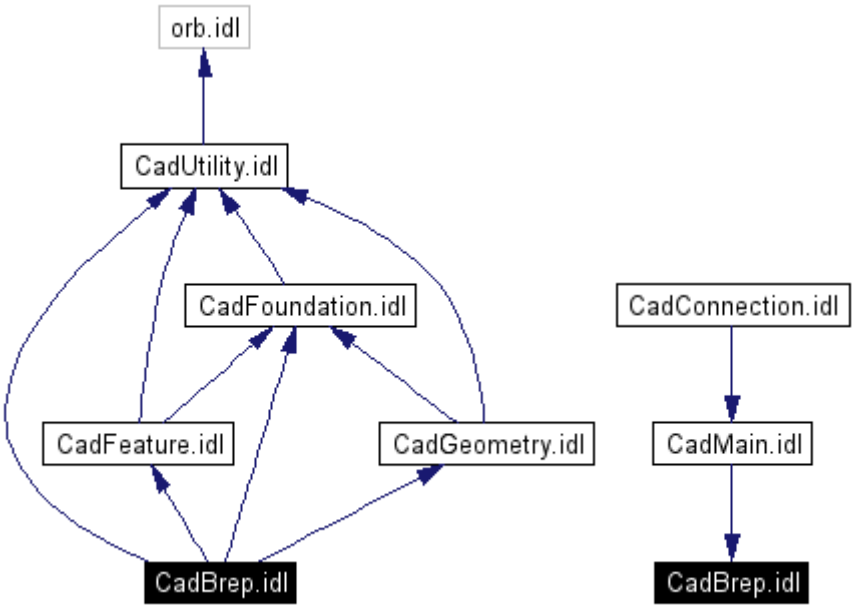


Abbildung 9: Abhängigkeitsgraph Modul CadBrep

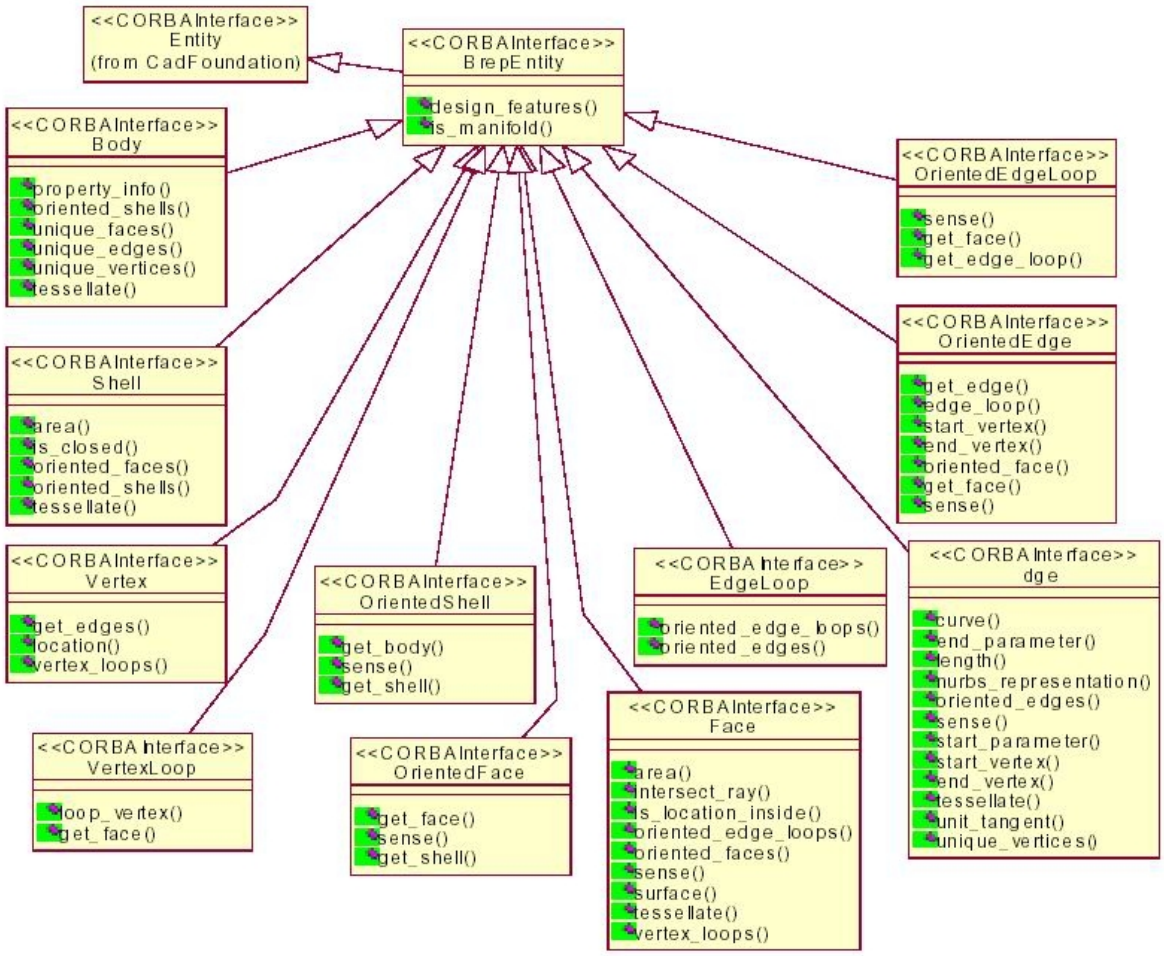


Abbildung 10: UML Diagramm CadBrep

Interface BrepEntity

```
interface BrepEntity : CadFoundation::Entity
  {

    CadFeature::DesignFeatureSeq design_features () raises (CadUtility::CadError);
    // Sequence of the design features directly involved with the
    // creation of this entity.

    boolean is_manifold() raises (CadUtility::CadError);
  };

typedef sequence<BrepEntity> BrepEntitySeq;
```

Methoden

`design_features()`	Gibt eine Liste von DesignFeatures zurück, die verwendet wurden, um den B-Rep aufzubauen.
`is_manifold()`	Test, ob es sich um eine „Fertigbare"-Geometrie handelt. Beispiel für eine nicht-fertigbare Geometrie sind nichtvollständige Körper.

Interface Body

Interface eines Body-Topologie-Elements.

```
interface Body : BrepEntity
  {
    // A collection of Brep entities defining a closed volume, aka solid.

    PropertyStruct property_info( inout double accuracy) raises
(CadUtility::CadError);
    // Returns a structure with property info

    OrientedShellSeq oriented_shells () raises (CadUtility::CadError);
    // Returns a sequence of the associated oriented shells. The first oriented
    // shell in the list defines the external or outside boundary of the body.

    FaceSeq unique_faces() raises (CadUtility::CadError);
    // Returns a sequence of the unique faces composing this body

    EdgeSeq unique_edges() raises (CadUtility::CadError);
    // returns a sequence of the unique edges in this body

    VertexSeq unique_vertices() raises (CadUtility::CadError);
    // returns a sequence of unique vertices in this body

    CadGeometry::ConnectedFaceTessellationStruct tessellate (
        in CadGeometry::TessType t_type,
        inout CadGeometry::TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters
    // If Flag is true the TessParameters were changed
```

```
  };
```

Methoden

`property_info()`	Eigenschaften des Body.
`oriented_shells()`	Ermitteln aller Flächenverbunde (Hüllen). Erster Flächenverbund ist die äußere Hülle.
`unique_faces()`	Flächen, die diesen Körper zusammensetzen.
`unique_edges()`	Kanten des Körpers.
`unique_vertices()`	Eckpunkte des Körpers.
`tesselate()`	Erzeugt ein tesseliertes Modell des Körpers mit dem CAD-System eigenen Mesher.

Interface OrientedShell

```
interface OrientedShell : BrepEntity
  {
    // An oriented use of a shell.
    // An oriented shell must always be used by at least one body

    Body get_body () raises (CadUtility::CadError);
    // Returns the body that uses this oriented shell.

    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the oriented shell agrees with the
    //direction of the underlying shell.

    Shell get_shell () raises (CadUtility::CadError);
    // Returns the shell associated with this oriented entity.
  };
```

Methoden

`get_body()`	Ermitteln des Parent-Objektes – der zugehörige Body.

`sense()`	Test, ob die Orientierung der Oriented-Shell mit der Shell übereinstimmt.
`get_shell()`	Gibt die zugehörige Shell zurück.

Interface Shell

```
interface Shell : BrepEntity
  {
    // An collection of oriented faces.
    // An independent, open shell can represent a skin or quilt.

    double area( inout double accuracy) raises (CadUtility::CadError);
    // shell area - accuracy is implementation defined

    boolean is_closed() raises (CadUtility::CadError);

    OrientedFaceSeq oriented_faces () raises (CadUtility::CadError);
    // Returns a sequence of the oriented faces in this shell.
    //The ordering of the oriented faces in this sequence has no significance.

    OrientedShellSeq oriented_shells () raises (CadUtility::CadError);
    // Returns a sequence of the oriented shells that use this shell.
    //Returns an empty sequence if this shell is independent.

    CadGeometry::FaceTessellationStructSeq tessellate (in CadGeometry::TessType
t_type,
        inout CadGeometry::TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters
    // If Flag is true the TessParameters were changed
  };
```

Methoden

`area()`	Flächeninhalt.
`is_closed()`	Test, ob Fläche geschlossen ist.
`oriented_faces()`	Gibt eine Sequenz von orientierten Flächen, die zu dieser Shell gehören, zurück.
`oriented_shells()`	Ermittelt die zugehörige `OrientedShell` (Parent-Objekt).

`tesselate()`	Erzeugen der tesselierten Darstellung.

Interface Vertex

Topologie-Element: Punkt.

```
interface Vertex : BrepEntity
  {
    // A topological point.

    EdgeSeq get_edges () raises (CadUtility::CadError);
    // Returns a sequence of the edges that use this vertex.
    // Returns an empty sequence if this vertex is independent.

    CadUtility::PointStruct location() raises (CadUtility::CadError);
    // Returns the 3D coordinates.

    VertexLoopSeq vertex_loops() raises (CadUtility::CadError);
    // Returns a sequence of the vertex loops that use this vertex.
  };
```

Methoden

`get_edges()`	Sequenz von Kanten, die diesen Punkt benutzen.
`location()`	Koordinaten des Punktes.
`vertex_loop()`	Sequenz von Kantenzügen, die diesen Punkt benutzen.

Interface VertexLoop

Topologie-Element eines (Eck-)Punktes. Mit ihm wird die berandende Kurve einer Fläche beschrieben.

```
interface VertexLoop : BrepEntity
  {
    // A topological pole or point location used to define the boundary of a face.
    // Examples include the pole of a sphere or a cone.
    // A vertex loop must always be used by a face (never independent).

    Vertex loop_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the 3D location of this vertex loop.

    Face get_face () raises (CadUtility::CadError);
    // Returns the face that uses this vertex loop.
    // Since vertex loops cannot be independent, this object must be used to
    // construct an edge loop before it is considered valid.
  };
```

Methoden

`loop_vertex()`	Position des Kantenzuges (topologisch).
`get_face()`	Fläche zu der dieser Punkt gehört.

Interface EdgeLoop

Topologie-Element: Kantenzug

```
interface EdgeLoop : BrepEntity
  {
    OrientedEdgeLoopSeq oriented_edge_loops() raises (CadUtility::CadError);
    // oriented edge loops that reference this edge loop

    OrientedEdgeSeq oriented_edges() raises (CadUtility::CadError);
    // oriented edges that compose the edge loop
  };
```

Methoden

`oriented_edge_loops()`	Orientierter Kantenzug dieses Kantenzuges.
`oriented_edges()`	Orientierte Kanten, die diesen Kantenzug zusammensetzen.

Interface OrientedEdgeLoop

Topologie-Element: Orientierter Kantenzug

```
interface OrientedEdgeLoop : BrepEntity
  {
    boolean      sense() raises (CadUtility::CadError);
    // true indicates agreement with the underlying edge loop

    Face      get_face() raises (CadUtility::CadError);
    EdgeLoop get_edge_loop() raises (CadUtility::CadError);
  };
```

Methoden

`sense()`	Richtung. TRUE im Fall, dass der zugehörige `EdgeLoop` die gleiche Richtung hat.
`get_face()`	Zugehörige Fläche.
`get_edge_loop()`	Gibt den zugehörigen `EdgeLoop` zurück.

Interface Face

Topologie-Element einer Fläche.

```
interface Face : BrepEntity
  {
    readonly attribute CadUtility::RangeStruct range_u;
    readonly attribute CadUtility::RangeStruct range_v;
    // bounds of the active region of the face as defined by the inner and outer
loops.

    double area( inout double accuracy) raises (CadUtility::CadError);
    // Evaluates area to a specified accuracy.
    // Accuracy is implementation defined.

    boolean intersect_ray ( in CadUtility::RayStruct ray, in double tolerance,
        out CadUtility::PointStructSeq intersection_points,
        out CadUtility::UvStructSeq intersection_parameters)
         raises (CadUtility::CadError);
    // Evaluates the intersections between the specified ray and the face.
    // The tolerance defines how close the ray must come to the face to be considered
    // an intersection. Returns TRUE if any intersections were found, FALSE if not.
    // Any intersections are returned in two sequences: one of 3D points and one of
    // corresponding 2D parameter values on the face's surface.

    Location is_location_inside (in CadUtility::UvStruct location)
        raises (CadUtility::CadError);
    // Queries if a location (defined by uv parameter values) is in the active region
    // of the face as defined by the inner and outer loops.
```

```
    OrientedEdgeLoopSeq oriented_edge_loops () raises (CadUtility::CadError);
    // Returns a list of the associated Brep.OrientedEdgeLoop entities.
    // The first oriented edge loop in the list defines the outside boundary of the
face.

    OrientedFaceSeq oriented_faces () raises (CadUtility::CadError);
    // Returns a list of the associated CadBrep::OrientedFace entities.
    //Returns an empty list if this face is independent, e.g. a trimmed surface.

    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the face agrees with the parametric (normal)
    // direction of the underlying surface.
    // Critical for determining the "outside" of a face in a body, for example.

    CadGeometry::Surface surface () raises (CadUtility::CadError);
    // Returns the CadGeometry::Surface entity that defines the shape of this face.

    CadGeometry::FaceTessellationStruct tessellate (in CadGeometry::TessType t_type,
        inout CadGeometry::TessParametersStruct params, out boolean t_flag)
         raises (CadUtility::CadError);
    // Tessellates the surface to the specified TessParameters
    // If Flag is true the TessParameters were changed

    VertexLoopSeq vertex_loops () raises (CadUtility::CadError);
    // Returns a sequence of any vertex loops defined on this face.
  };
```

Attribute

`range_u`	Abmessungen des Flächenpatches in u-Richtung.
`range_v`	Abmessungen des Flächenpatches in v-Richtung.

Methoden

`oriented_edge_loops()`	Begenzender Kantenzug der Fläche.
`area()`	Flächeninhalt.
`intersect_ray()`	Test ob ein definierter Strahl die Fläche trifft. Diese Operationen sind z.B. bei der Selektion innerhalb einer Benutzungsschnittstelle wichtig.
`is_location_inside()`	Test, ob ein Punkt in einem Flächen-Patch liegt.

`oriented_face()`	Assozierte `OrientedFace`.
`sense()`	Test, ob die Fläche die gleiche Orientierung des Normalen-vektors hat.
`surface()`	Geometrie-Element der zugehörigen Fläche.
`tesselate()`	Erzeugt eine tesselierte Darstellung.
`vertex_loops()`	Gibt eine Sequenz von `VertexLoops` dieser Fläche zurück.

Interface OrientedFace

```
interface OrientedFace : BrepEntity
  {
    // An oriented use of a face.

    Face get_face () raises (CadUtility::CadError);
    // Returns the face associated with this oriented entity.

    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the oriented face agrees with the
    // direction of the underlying face.

    Shell get_shell () raises (CadUtility::CadError);
    // Returns the shell that uses this oriented face.
  };
```

Methoden

`get_face()`	Ermittelt die assoziierte Fläche.
`sense()`	Test, ob die Orientierungen mit der assoz. Fläche übereinstimmen.
`get_shell()`	Ermittelt die `Shell`, die diese Fläche benutzt.

Interface OrientedEdge

```
interface OrientedEdge : BrepEntity
  {
    // An oriented use of an edge.

    Edge get_edge () raises (CadUtility::CadError);
    // Returns the edge associated with this oriented entity.

    EdgeLoop edge_loop () raises (CadUtility::CadError);
    // Returns the edge loop that uses this oriented edge.

    Vertex start_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the start of this oriented edge.

    Vertex end_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the end of this oriented edge.
    // Takes into account any sense differences.

    OrientedFace oriented_face () raises (MultipleFaces, CadUtility::CadError);
    // Returns the oriented face that uses this oriented edge.
    // Returns NULL if the oriented edge is in an independent edge loop or
    // bounds an independent face.
    // Raises an exception if more than one oriented face uses this oriented edge.

    Face get_face() raises (MultipleFaces, CadUtility::CadError);

    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the oriented edge (from start to
    // end vertices) agrees with the direction of the underlying edge.
 };
```

Methoden

`get_edge()`	Zugehörige `Edge`.
`edge_loop()`	Zugehöriger Kantenzug.
`start_vertex()`	Startpunkt.
`end_vertex()`	Endpunkt.
`oriented_face()`	Zugehörige `OrientedFace`.
`get_face()`	Zugehörige Fläche.
`sense()`	Test, ob die Richtung des assz. Entity übereinstimmt.

Interface Edge

```
interface Edge : BrepEntity
  {
    // A trimmed portion of a curve.An edge that uses the same vertex for both
    // start and
    // end vertices must be defined as a closed edge on a closed curve starting
    // and ending
    // at this vertex.  An independent edge can be used to represent a trimmed curve.

    CadGeometry::Curve curve()  raises (CadUtility::CadError);
    // Returns the curve that defines the shape of this edge in model space.

    double length ( inout double accuracy )  raises (CadUtility::CadError);
    // Evaluates the length of the edge to a specified accuracy.
    // The accuracy is implementation defined.

    CadUtility::NurbsCurveStruct nurbs_representation ( inout double tolerance )
         raises (CadUtility::CadError);
    // Returns a NURBS curve that approximates this edge within the specified
tolerance.

    OrientedEdgeSeq oriented_edges () raises (CadUtility::CadError);
    // Returns a sequence of the oriented edges that use this edge
    // Returns an empty sequence if this edge is independent.

    boolean sense () raises (CadUtility::CadError);
    // Queries whether the direction of the edge (from start to end vertices)
    // agrees with the parametric direction of the underlying curve.

    double start_parameter () raises (CadUtility::CadError);
    // Returns the curve parameter corresponding to the start vertex.

    double end_parameter ()  raises (CadUtility::CadError);
    // Returns the curve parameter corresponding to the end vertex.

    Vertex start_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the start of this edge.

    Vertex end_vertex () raises (CadUtility::CadError);
    // Returns the final vertex that defines this edge.

    CadGeometry::EdgeTessellationStruct tessellate ( in double tolerance)
         raises (CadUtility::CadError);
    // Tessellates the edge to a specified chordal deviation tolerance.

    CadUtility::VectorStruct unit_tangent (in double parameter,in boolean sense)
         raises (CadUtility::CadError);
    // Evaluates the unit tangent vector of the edge at the specified parameter
    // and sense.
    // If the sense is TRUE, the tangent vector is oriented with the edge.
    // If the sense is FALSE, the tangent vector is oriented in the
    // opposite direction.

    VertexSeq unique_vertices () raises (CadUtility::CadError);

    // Returns a sequence of unique vertices used by this edge.
};
```

Methoden

`curve()`	Zugehörige Kurve der Kante.
`length()`	Länge des Kurvensegments.
`nurbs_representation()`	Zugriff auf die NURBS Repräsentation.
`oriented_edges()`	Ermitteln der zugehörigen `OrientedEdges`.
`sense()`	Test, ob Orientierung mit dem zugehörigen Kantenzug gleich ist.
`start_parameter()`	Wert, der bei der parametrischen Kurve den Startpunkt ergibt.
`end_parameter()`	Wert, der bei der parametrischen Kurve den Endpunkt ergibt..
`start_vertex()`	Startpunkt.
`end_vertex()`	Endpunkt.
`tesselate()`	Tesselierte Darstellung.
`unit_tangent()`	Test, ob der zugehörige Tangentenvektor der Kurve die gleiche Orientierung hat.
`unique_vertices()`	Punkte dieser Kurve.

Das Modul CadFeature

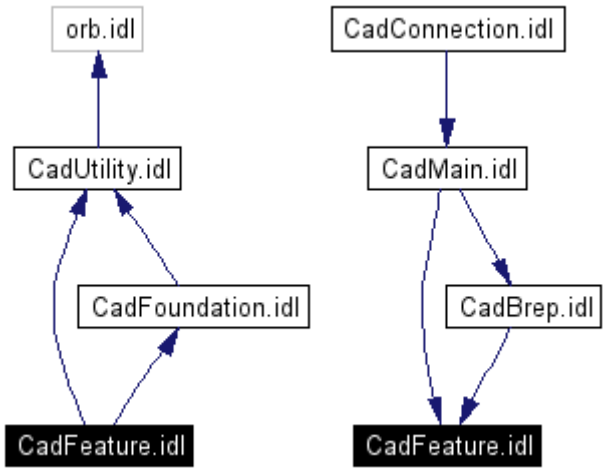


Abbildung 11: Abhängigkeitsgraph CadFeature

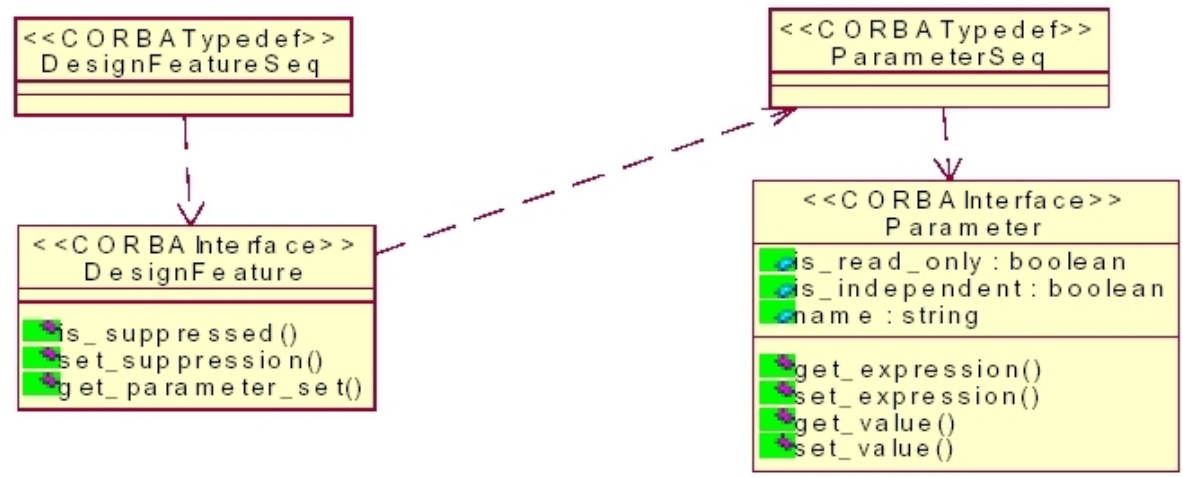


Abbildung 12: UML Diagramm CadFeature

Interface DesignFeature

Interface eines Design-Features.

```
interface DesignFeature  :  CadFoundation::Entity
  {
    // A distinct step or node in the parametric definition of a model.
    // It drives the creation of a set of Brep entities in the fully-evaluated
    // form of the model.

    boolean    is_suppressed() raises (CadUtility::CadError);
    void       set_suppression() raises (CadUtility::CadError);
    ParameterSeq get_parameter_set() raises (CadUtility::CadError);
  };
```

Methoden

`is_suppressed()`	Test, ob das Feature aktiv ist.
`set_suppression()`	Flag, der das Feature aktiv setzt.
`get_parameter_set()`	Parameter, die dieses Feature beschreiben.

Interface Parameter

Interface eins Parameters, der ein Feature beschreibt.

```
interface Parameter
  {
    // data structures that capture the (parametric) features of ModelEntities

    readonly attribute boolean   is_read_only;
    readonly attribute boolean   is_independent;
    readonly attribute string    name;

    string   get_expression() raises (CadUtility::CadError);
    void set_expression(in string e_value) raises (CadUtility::CadError);
    // operations to allow an expression that may drive geometry

    CadUtility::EntityAttrib  get_value() raises (CadUtility::CadError);
    void set_value(in CadUtility::EntityAttrib value) raises (CadUtility::CadError);
    // operations providing access to parameter value
  };
```

Attribute

`is_read_only`	Flag, ob der Parameter veränderbar ist.
`is_independent()`	Test auf Unabhängigkeit des Parameters.
`name()`	Name des Parameters.

Methoden

`get_expression()`	Gibt den Parameter-Ausdruck zurück.
`set_expression()`	Setzt den Parameter-Ausdruck.
`get_value()`	Setzt den Wert des Parameters.

Das Modul CadUtility

Das Modul CadUtility beinhaltet wichtige allgemeine Definitionen in Form von `structs` und `typedefs` die in allen anderen Modulen benötigt werden. Wie in Abbildung 13 (Anhang) zu erkennen ist wird das Modul CadUtility von allen anderen Modulen importiert, importiert jedoch selbst keine Module.

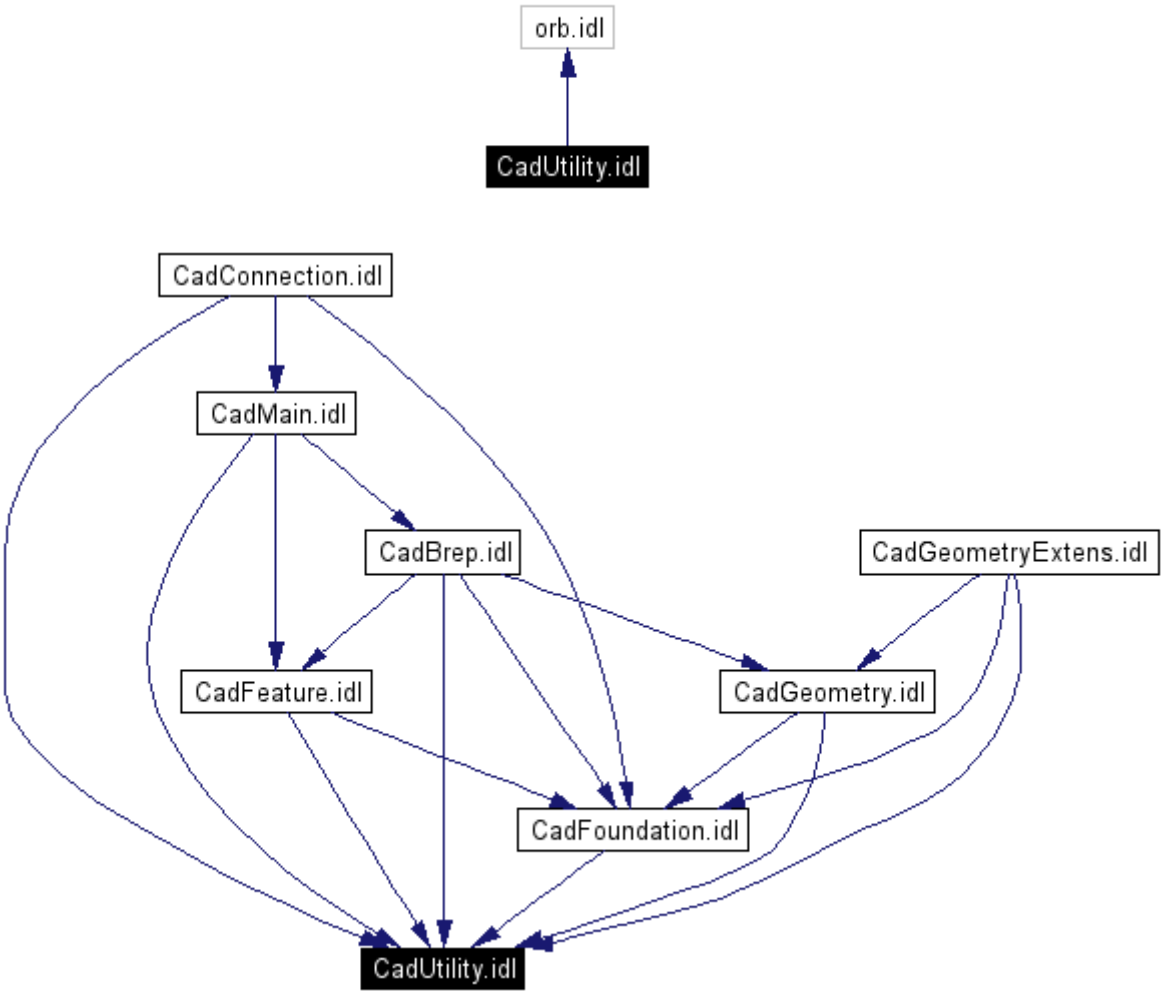


Abbildung 13: Abhängigkeitsgraph CadUtility

```
struct PointStruct
  {
    // three dimensional location

    double x;
    double y;
    double z;
  };
```

`PointStruct` definiert den Ortsvektor eines Punktes im dreidimensionalen Raum.

```
typedef sequence<PointStruct>     PointStructSeq;
typedef sequence<PointStructSeq>  PointStructSeqSeq;
```

`PointStructSeq` und `PointStructSeqSeq` definieren die zugehörigen Sequenzen (Listen) aus Ortsvektoren und Ortsvektorlisten.

```
struct BoundingBox
  {
```

```
    PointStruct   point_min;
    PointStruct   point_max;
  };
```

`BoundingBox` gibt die maximalen Abmessungen einer Entity in Form eines Quaders an. Hierbei werden die beiden diagonal gegenüberliegenden Eckpunkte, die den Quader aufspannen, ermittelt.

```
struct VectorStruct
  {
    // Direction in 3D

    double i;
    double j;
    double k;
  };

typedef sequence<VectorStruct> VectorStructSeq;
typedef sequence<VectorStructSeq> VectorStructSeqSeq;
typedef sequence<VectorStructSeqSeq> VectorStructSeqSeqSeq;
```

`VectorStruct` ist ein Richtungsvektor im dreidimensionalen Raum. `VectorStructSeq`, `VectorStructSeqSeq`, `VectorStructSeqSeqSeq` die dazugehörigen Container-Objekte.

```
struct TransformationStruct
  {
    PointStruct     offset;
    VectorStruct    i_ref_dir;
    VectorStruct    k_dir;
  };
```

Ein `TransformationStruct` legt die relative Position eines Objektes zu einem anderen Objekt fest. Der `offset` ist eine Translation. Die beiden Vektoren `i_ref_dir` und `k_dir` bestimmen ein orthogonales Koordinatensystem, das die Rotation des Objektes festlegt.

```
struct RayStruct
  {
    PointStruct     origin;
    VectorStruct    direction;
  };

typedef sequence<RayStruct> RayStructSeq;
```

Ein `RayStruct` ist ein Datentyp, der eine Position (`origin`) mit einer (Blick-)Richtung (`direction`) verbindet. `RayStructSeq` ist der dazugehörige Listen-Container.

```
struct UvStruct
  {
    double u;
    double v;
  };

typedef sequence<UvStruct> UvStructSeq;
```

`UvStruct`'s werden verwendet, um u-v-Parameterwerte bei Flächenbeschreibungen abzubilden. Eine `UvStructSeq` ist ein Listen-Container für solche Parameterwerte.

```
enum MassUnit
  {
    // mass unit options

    POUNDS,
    GRAMS,
    KILOGRAMS,
    UNKNOWN_MASS
   };

  enum LengthUnit
  {
    // length unit options

    INCH,
    FEET,
    M,
    CM,
    MM,
    UNKNOWN_LENGTH
  };
```

`MassUnit` bestimmt die Gewichtseinheit, die global in dem Modell festgelegt ist, `LengthUnit` die möglichen Längeeinheiten.

```
struct ColorStruct
  {
    // basic color information in RGB form
    // Valid values range from 0.0 to 1.0

    double red;
    double green;
    double blue;
  };

typedef sequence<ColorStruct> ColorStructSeq;
```

`ColorStruct` ist ein Datentyp, der eine Farbe im RGB-Farbraum spezifiziert. Es wird hierbei ein Wert zwischen 0.0 und 1.0 für die Mischungsfarben Rot, Grün und Blau angegeben.

```
struct PresentationStruct
  {
    // CAD system presentation data
    // Unsupported features will return a negative value
    // Valid values range from 0.0 to 1.0

    ColorStruct   object_color;
    ColorStruct   specular_color; //light source color
    double diffuse_factor;
    double specular_factor;
    double ambient_factor;
    double roughness;
    double transparency; //  0. is opaque and 1. is transparent
  };
```

Ein `PresentationStruct` definiert die Darstellungseigenschaften des CAD-Systems. Durch zwei `ColorStruct`'s wird die Objektfarbe und die Farbe des verwendeten Lichtes bestimmt. Weitere Attribute sind z. B. Transparenz und Glanzeigenschaften.

```
struct RangeStruct
  {
    // basic range information

    double high;
    double low;
  };
```

Ein `RangeStruct` definiert einen Wertebereich.

```
// Nurbs data structures

struct NurbsCurveStruct
  {

  boolean is_rational;
  // rational or polynomial?

  CadUtility::DoubleSeq knots;
  // A sequence of knot values.

  CadUtility::DoubleSeq weights;
  // A sequence of weight values.

   CadUtility::PointStructSeq control_points;
   // A sequence of control points in 3D

   CadUtility::LongSeq multiplicity;
   long degree;
  };

typedef sequence<NurbsCurveStruct> NurbsCurveStructSeq;

struct NurbsSurfaceStruct
  {
    boolean is_rational;
    // rational or polynomial?

    CadUtility::DoubleSeq knots_u;
    CadUtility::DoubleSeq knots_v;
    // Sequence of knot values.

    CadUtility::DoubleSeqSeq weights;
    // A sequence of weight values.

    CadUtility::PointStructSeqSeq control_points;
    // A sequence of control points.
    // Each point is a sequence of a sequence of 3D points.

    CadUtility::LongSeq multiplicity_u;
    CadUtility::LongSeq multiplicity_v;
    long   degree_u;
    long   degree_v;
  };

typedef sequence<NurbsSurfaceStruct> NurbsSurfaceStructSeq;
```

`NurbsCurveStruct` und `NurbsSurfaceStruct` definieren Datentypen zur Modellierung von NURBS[2]-Kurven und NURBS-Flächen. Im `NurbsCurveStruct` wird das Stützpolynom durch die Knotenpunkte in `knots` aufgespannt. Die Sequenz aus `double` Werten in `weights` ist die Gewichtung der Stützpunkte. Analog hierzu wird die NURBS-Fläche modelliert. `knots_u` und `knots_v` liefern die Stützpunkte der Fläche in u- bzw. v-Richtung.

[2] NURBS (engl. Non-Uniform rational Basis-Spline)