Konzeption und Implementierung eines Moduls für das Arbeiten mit Kooperationselementen in einem 3D-CAD-System

Studienarbeit

Fachgebiet Datenverarbeitung in der Konstruktion
Technische Hochschule Darmstadt
Prof. Dr.-Ing. R. Anderl

angefertigt von
stud. wirtsch.-ing. Marcel Schefczik
Jahnstraße 38, 64285 Darmstadt

betreut durch
Dipl.-Ing. Kai Schiemenz

Inhaltsverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[SaMo-95] David Sauder, Katherine Morris: *Design of a C++ Software Library for Implementing EXPRESS: The NIST STEP Class Library*
EXPRESS User Group International Conference
Grenoble, France, October 1995

[AnMe-96] R. Anderl, R. Mendgen: *Skript Produktdatentechnologie I - III*
Fachgebiet Datenverarbeitung in der Konstruktion, Technische Universität Darmstadt, 1996

[Hofm-99] F. Hoffmann *Konstruktion eines Mobilfunktelefons unter Einsatz von Kooperationselementen*
Fachgebiet Datenverarbeitung in der Konstruktion, Technische Universität Darmstadt, 1999

[ScKu-95] M. Schader, S. Kuhlins: *Programmieren in C++: Einführung in den Sprachstandard C++*
Springer-Verlag, Mannheim, Dritte Auflage 1995

[SWug-98] N.N., *SolidWorks®98 users guide*
©1998 SolidWorks®Corporation, Concord, Massachusetts

[SWtt-98] N.N., *SolidWorks®98 tutorial*
©1998 SolidWorks®Corporation, Concord, Massachusetts

[SWap-98] N.N., *SolidWorks®98 API users guide: Application Programming Interface for SolidWorks®98*
©1998 SolidWorks®Corporation, Concord, Massachusetts

[STEP] N.N., *ISO 10303 Product Data Representation and Exchange*

[Stro-94] B. Stroustrup: *The Design and Evolution of C++*
Addison-Wesley, Reading, Massachusetts, 1994

[RRZN-98] N.N. *C++ für C-Programmierer*
Regionales Rechenzentrum für Niedersachsen, Hannover, 10.Auflage 1998

Kapitel 1

Einleitung

In der heutigen Zeit sind moderne Produktentwicklungsprozesse von arbeitsteiliger Durchführung gekennzeichnet. Die Arbeit findet in interdisziplinären Teams und immer öfter sowohl örtlich als auch zeitlich verteilt statt. Als Herausforderung entsteht dabei insbesondere die Abstimmung zwischen den Teilaufgaben. Ein Ansatz für eine Verbesserung ist dabei die Arbeit mit Kooperationselementen, die in den Teilaufgaben gleichzeitig genutzte geometrische Elemente darstellen.

Ausgangspunkt ist das in der Aufgabenstellung erwähnte Datenmodell für die Repräsentation von Kooperationselementen. Das Datenmodell wurde am Fachgebiet Datenverarbeitung in der Konstruktion an der Technischen Universität Darmstadt entwickelt, eine genauere Beschreibung erfolgt in Kapitel 2.1. Auf Basis des Datenmodells soll ein Modul für ein 3D-CAD-System konzipiert und prototypisch implementiert werden, welches das Arbeiten mit Kooperationselementen unterstützt. Das Modul soll dabei folgende Funktionen umfassen:

- **Einfache Erzeugung und Verwaltung der Kooperationselemente im CAD-System:** In das CAD-System soll ein zusätzliches Menü eingefügt werden, welches die Menüpunkte für die Behandlung der Kooperationselemente enthält. Einer der Menüpunkte betrifft das Erzeugen eines Kooperationselementes, was auf zwei Arten möglich sein sollte:

  Top-Down: Das Kooperationselement wird direkt in einer Teiledatei erzeugt. Dabei soll die übliche Funktionalität der CAD-Modellierung zur Verfügung stehen.

  Bottom-Up: Ein Feature in einem Bauteil wird selektiert und anhand dessen Geometrie ein entsprechendes Kooperationselement erzeugt.

  Referenzierung auf die Kooperationselemente soll möglich sein; nur so können die Kooperationselemente auch sinnvoll genutzt werden.

  Beim Erzeugen sollen über Geometrie hinausgehende Daten dialogorientiert eingegeben werden.

- **Ausgabe der Kooperationselemente als STEP-Datei:**[1] Ein weiterer Menüpunkt erlaubt das Speichern der Kooperationselemente in einer Datei. Dazu werden die Kooperationselemente, entsprechend dem Datenmodell, in einem Kooperationskontext zusammengefaßt, dessen Erzeugung einem weiteren Menüpunkt zuzuordnen ist.

  Um eine Zusammenarbeit unterschiedlicher Teams, Unternehmensbereiche oder sogar verschiedener Unternehmen zu ermöglichen, bei der i.A. unterschiedliche CAD-Systeme unter verschiedenen Betriebssystemen zur Anwendungen kommen, wird ein neutrales Dateiformat zur Repräsentation verlangt.

- **Einlesen der in einer STEP-Datei repräsentierten Kooperationselemente mit der Erzeugung der entsprechenden Geometrieelemente des CAD-Systems:** Um erstellte Kooperationselemente verschiedenen Anwendern zur Verfügung zu stellen, ist das Einlesen einer Kooperationselementedatei mit einem Menüpunkt zu verknüpfen; dabei sollen die Kooperationselemente in einer eigenen CAD-Teile-Datei visualisiert werden.

Die Implementierung soll unter Nutzung der STEP Class Library des National Institute for Standardisation (NIST) in C++ erfolgen.

[1] STEP: ISO Standard for the Exchange of Product Model Data

Kapitel 2

Vorarbeiten

In diesem Kapitel werden Vorarbeiten zum Projekt beschrieben. Grundlagen, auf denen diese Studienarbeit aufbaut, wie z.B. das Datenmodell, aus dem die einbezogenen Klassen abgeleitet wurden, werden erläutert und verwendete Werkzeuge (SolidWorks, Microsoft Developer Studio, Visual C++) und die Klassenbibliotheken zur STEP-Implementation, die Step Class Library, werden vorgestellt..

Diese Bestandteile werden in ihren Strukturen und Verwendungsmöglichkeiten untersucht und die grundsätzliche Handhabung weitgehend erläutert.

2.1 Beschreibung des Datenmodells

Das Datenmodell (Abb.2.1 und Abb.2.2) für die Arbeit mit Kooperationselementen wurde 1999 von K. Schiemenz und S. Vettermann am Fachgebiet Datenverarbeitung in der Konstruktion an der Technischen Universität Darmstadt entwickelt. Es liegt im EXPRESS-G-Format vor, einer graphischen Datenmodellierungssprache, die im Rahmen der ISO-10303-Entwicklung genormt ist; die daraus abgeleiteten C++-Klassen werden in die Implementation dieses Projekts eingefügt; sie sind Ausgangsbasis der Programmentwicklung.

Das Datenmodell läßt sich in Anlehnung an den Ansatz des integrierten Produktmodells[AnMe-96] in zwei Hauptbestandteile gliedern:

- die *Definition* des Kooperationselementes und
- die *Repräsentation* des Kooperationselementes

Die im integrierten Produktmodell enthaltene *Präsentation* ist nicht Bestandteil des Datenmodells, wird jedoch ein Aspekt in diesem Projekt sein.

2.1.1 Kooperationselementedefinition

Die Kooperationselementedefinition umfaßt, ähnlich den produktdefinierenden Daten im integrierten Produktmodell, Daten zur Identifizierung des Kooperati-

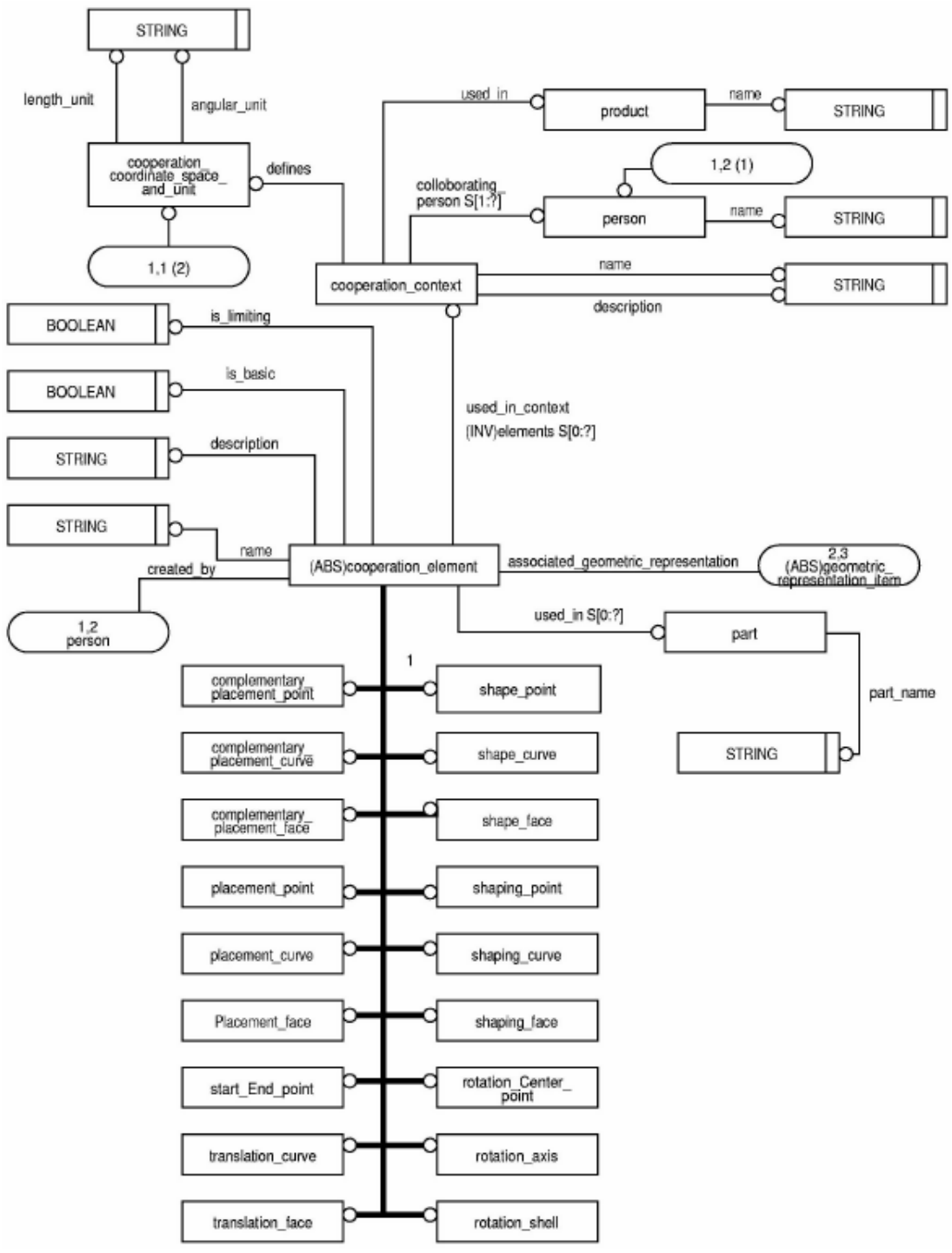

Abbildung 2.1: Datenmodell, Teil 1

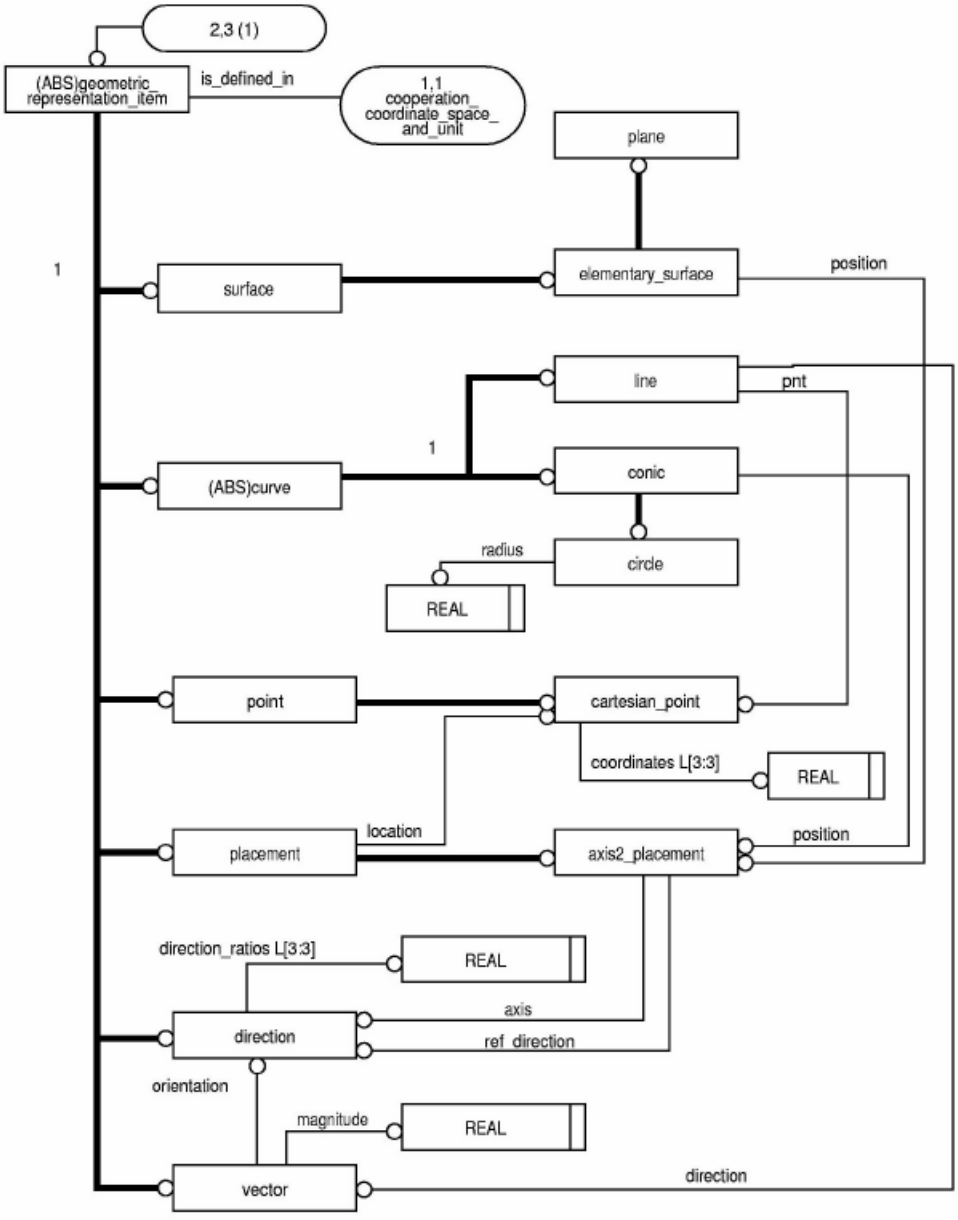

Abbildung 2.2: Datenmodell, Teil 2

onselementes und zur Zuordnung zu einem Kontext und/oder einem Produkt. Darüberhinaus enthält dieser Bereich Informationen über verwendete Maßeinheiten und genauere Erläuterungen zum Typ des Kooperationselementes.

Kooperationselement und Kooperationskontext

Das gegebene Konzept geht davon aus, daß alle Kooperationselemente, die in einer Baugruppe verwendet werden, in einem Kooperationskontext zusammenzufassen sind. Alle Daten, die zu diesem Kontext und seinen Kooperationselementen gehören, werden zusammen in einer Datei abgelegt und im CAD-System gemeinsam visualisiert. Diesem Kontext werden folgende Attribute zugeordnet:

- *name*: Der Name des Kooperationskontextes;
  über diesen wird der Kontext eindeutig identifiziert; gleichzeitig ist dieser Name Standarddateiname fuer die Speicherung der Daten.
- *description*: Weitere Erläuterungen zum Kontext.
- *colloborating_person*: Eine Liste von Personen, die mit diesem Kontext arbeiten.
- *used_in*: Das Produkt, dem der Kontext zugeordnet wird.
- *defines*: Angaben zum Koordinatensystem, hier: Maßeinheiten für Länge und Winkel.
- *elements*: Eine Liste mit allen Kooperationselementen, die in diesem Kontext zusammengefaßt werden.

Klassifizierung der Kooperationselemente

F. Hofmann hat mit Hilfe eines Ansatzes aus der Konstruktionsmethodik einen Entwurf für eine Klassifikation von Kooperationselementen entwickelt [Hofm-99]. Er geht davon aus, daß sich die zu verwendenden Kooperationselemente zum einen nach Dimensionalität und zum anderen nach Form, Lage oder Bewegung gliedern lassen. Letztere Klassifikation wird noch einmal gegliedert, so daß sich ein folgendes Schema ergibt

- Form, Direkt
- Form, Indirekt
- Lage, Hilfsgeometrie
- Lage, Anordnungsgeometrie
- Bewegung, Rotation

- Bewegung, Translation

jeweils mit Punkt, Linie oder Fläche.

Diese Klassifikation wurde im Datenmodell derart berücksichtigt, daß der Typ *cooperation_element* als abstrakter Supertyp definiert wurde, der nicht instanziiert wird. Jeder Art von Kooperationselement wird ein Subtyp von *cooperation_element* zugeordnet; dieser wird bei Erzeugung eines entsprechenden Kooperationselementes instanziiert.

2.1.2 Kooperationselementerepräsentation

Die Kooperationselementerepräsentation umfaßt die komplette geometrische Beschreibung der Kooperationselemente (Abb.2.2) und sonstige beschreibende Elemente aus dem Datenmodell, wie z.B. die textuelle Beschreibung, Zuordnung zu bearbeitenden Personen, etc. Da verteilte Entwicklung heutzutage nicht nur kooperatives Arbeiten innerhalb einer Abteilung oder eines Unternehmens bedeutet, sondern auch oftmals Zusammenarbeit von verschiedenen Unternehmen impliziert, haben firmenübergreifende Schnittstellen eine besondere Bedeutung. Es wurde bei Erstellung des Datenmodells daher für die geometrische Beschreibung auf Bereiche der STEP-Norm zurückgegriffen. Wie Abb.2.2 zu entnehmen ist, existiert *geometric_representation_item* als abstrakter Supertyp, von welchem sich die anderen geometrischen Typen ableiten. In dieser Vererbungsstruktur sind alle benötigten Typen enthalten; in der untersten Ebene werden ausschließlich Basistypen (i.d.R. REAL) verwendet. Als Kooperationselemente werden folgende Typen instanziiert:

- plane (= Ebene, zweidimensionales KE)
- line (= Gerade, eindimensionales KE)
- circle (= Kreis, eindimensionales KE)
- cartesian_point (= kartesischer Punkt, nulldimensionales KE)

Beispielhaft wird die Struktur für die Darstellung eines Kreises erläutert:

circle enthält:

radius Typ *REAL*

position Typ *axis2_placement*, geerbt von *conic* enthält:

location Typ *cartesian_point*, geerbt von *placement* enthält:

coordinates Typ *List of 3 REAL*

axis Typ *direction* enthält:

direction_ratios Typ *List of 3 REAL*

ref_direction Typ *direction* enthält:

direction_ratios Typ *List of 3 REAL*

2.1.3 C++-Klassen des Datenmodells

Das im EXPRESS-G-Format vorliegende Datenmodell wurde direkt in das textorientierte EXPRESS-Format überführen.

Aus diesem werden mit Hilfe des Programms WinSTEP die C++-Klassen abgeleitet. WinSTEP liefert als Output folgende Dateien:

- compstructs.cpp
- schema.cpp
- schema.h
- SdaiAll.cpp
- Sdaiclasses.h
- SdaiKOOPE.cpp
- SdaiKOOPE.h
- SdaiKOOPE.init.cpp

Die Dateien *SdaiKOOPE.h* und *SdaiKOOPE.cpp* enthalten sämtliche Klassen- und Methodendefinitionen des Datenmodells. Die Datei *Sdaiclasses.h* enthält einige *typedef*-Anweisungen für Bezeichner, die in den Definitionen verwendet werden.

Der Zugriff auf die Klassen des Datenmodells und die Verwaltung derselben werden über Methoden der STEP Class Library (SCL) abgewickelt; deren Beschreibung erfolgt in Kapitel 2.3.

2.2 Werkzeuge im Überblick

2.2.1 SolidWorks 98

Das Programm SolidWorks 98 (Abb.2.3) ist ein modernes 3D-CAD-Werkzeug für das Betriebssystem Microsoft Windows.

Es bietet neben Modellierungsmöglichkeiten in Bauteil- und Baugruppenmodus einen graphischen Feature-Manager, der dem Benutzer die Kontrolle über die bearbeiteten CAD-Elemente erleichtert. Die Struktur eines CAD-Modells besteht aus Bauteilen[1] und Baugruppen[2]. Ein Bauteil ist die kleinste zusammenhängende Einheit und entspricht i.A. einem Einzelteil im realen Produkt. Bauteile werden im sogenannten Part-Modus erstellt und als eigene Datei gespeichert.

[1] part
[2] assembly

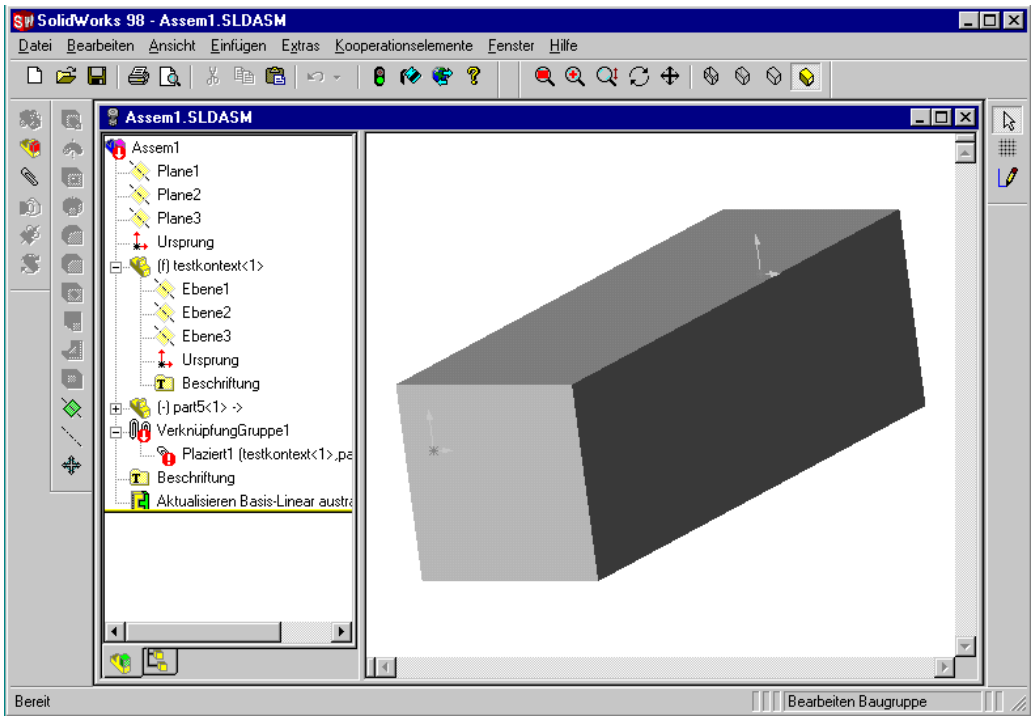

Abbildung 2.3: SolidWorks Oberfläche

Baugruppen sind nach bestimmten Gesichtspunkten geordnete Bereiche im Modell; sie bestehen aus Bauteilen und/oder Unterbaugruppen. Die einzelnen Baugruppenelemente sowie die Baugruppe selbst stellen dabei unabhängige Datenmengen im CAD-System dar. Die Baugruppendaten enthalten nur Verweise auf die verwendeten Baugruppenelemente sowie Information über Anordung (Position und Orientierung) der Baugruppenelemente zueinander.

Auch nach dem Hinzufügen einer Komponente (Bauteil oder Unterbaugruppe) zu einer Baugruppe kann die Komponente verändert werden; Assoziativität zwischen Teilen, Baugruppen und Zeichnungen gewährleistet, daß Änderungen, die in einem Teil vorgenommen werden, automatisch auch in allen übergeordneten Baugruppen und zugehörigen Zeichnungen ausgeführt werden[SWug-98].

2.2.2 SolidWorks 98 API

Für SolidWorks existiert eine Programmierschnittstelle (Application Programming Interface, API), mit dessen Hilfe Zusatzanwendungen für SolidWorks erstellt werden können. Das SolidWorks API ist eine objektorientierte Programmierschnittstelle, die OLE[3] unterstützt Es enthält zahlreiche Klassen mit Me-

[3]Object Linking and Embedding=(engl.) Objektverknüpfung und -einbettung

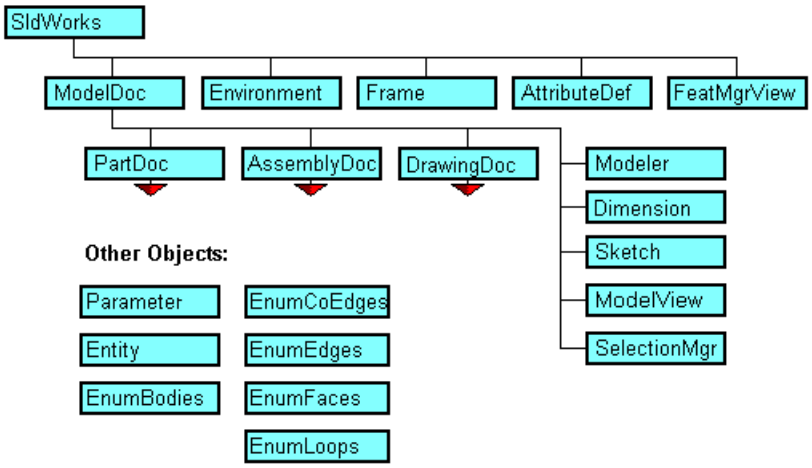

Abbildung 2.4: API-Klassenstruktur, Teil 1

thoden, die aus Visual Basic, VBA[4] (Excel, Access, usw.), C, C++ oder SolidWorks Makrofiles aufgerufen werden können und umfangreichen Zugriff auf SolidWorks-Dokumente und die SolidWorks-Oberfläche erlauben. Möglich ist z.B. das Hinzufügen oder Entfernen von Menüs oder Menüpunkten, das Verknüpfen von Menüpunkten mit aufzurufenden Funktionen und das Erzeugen, Bearbeiten, Speichern oder Löschen von SolidWorks-Dokumenten im Rahmen der normalen SolidWorks-Funktionalität. Im Rahmen dieser Studienarbeit wird auf das API über Visual C++ zugegriffen(Kap.2.2.3).

Die API-Klassenstruktur

Auf der obersten Ebene der Klassenstruktur des SolidWorks API steht die Klasse *SldWorks*. Sie repräsentiert die bestehenden SolidWorks Sitzung und erlaubt Operationen auf dieser Ebene, wie z.B. das Erzeugen, Öffnen oder Schließen von SolidWorks Dokumenten, Wechseln des aktiven Dokuments oder Bearbeiten der Menüstruktur. Über eine Instanz dieser Klasse erhält man Zugriff auf die darunterliegende Ebene der Klassenstruktur (siehe Abb.2.4); in diesem Projekt interessieren außerdem besonders folgende Klassen:

ModelDoc repräsentiert ein SolidWorks Dokument und bietet Zugriff auf dieses. Die Klasse beinhaltet u.a. Operationen, welche die Ansicht manipulieren, Funktionalität zur Erzeugung und Bearbeitung von Features und anderen geometrischen Elementen des Dokumentes. Weiterhin enthält sie den *SelectionManager*, der weiter unten näher beschrieben wird.
Ein *ModelDoc* kann entweder ein *PartDoc* (Bauteil), ein *AssemblyDoc* (Bau-

[4]VBA: Visual Basic for Applications

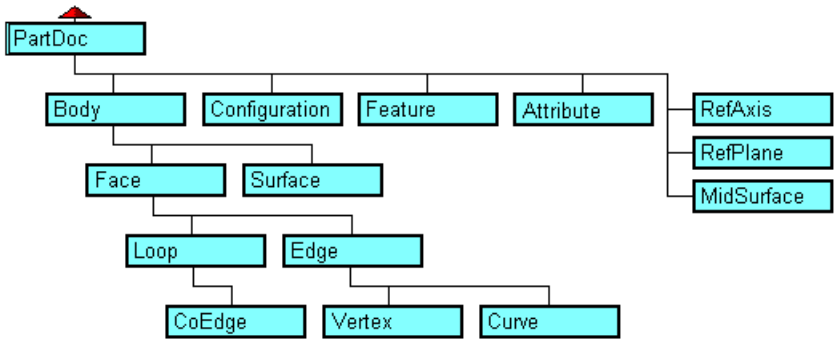

Abbildung 2.5: API-Klassenstruktur, Teil 2

gruppe) oder ein *DrawingDoc* (Zeichnung) sein; in diesem Projekt wird *DrawingDoc* nicht betrachtet.

Frame enthält Funktionen zur Manpulation der Menüstruktur; eine Beschreibung der wichtigsten Möglichkeiten erfolgt im Abschnitt "Hinzufügen neuer Menüpunkte" in Kap.2.2.3.

PartDoc repräsentiert ein Bauteil in SolidWorks. Diese Klasse enthält Methoden zur Manipulationen auf Part-Ebene. Sie erlaubt direkten Zugriff auf Elemente des Bauteils über deren Namen und das Erzeugen von Features; ebenso Zugriff auf Referenzgeometrie. In Abb.2.5 erkennt man die Struktur der unterhalb der Klasse *Body* liegenden Objekte, die im Rahmen der Implementation (siehe Kap.4.2) eine wichtige Rolle spielt. Dort wird diese Struktur anhand von Beispielen erläutert.

AssemblyDoc erlaubt Operationen auf Baugruppenebene, das beinhaltet u.a. Manipulation von Komponenten der Baugruppe (Bauteile oder Unterbaugruppen). Genau wie im Bauteil kann auch hier auf Elemente vom Typ Feature oder Body und auf Referenzgeometrie zugegriffen werden.

SelectionMgr bietet die Möglichkeit, auf im aktuellen Dokument selektierte Objekte zuzugreifen oder Objekte zu selektieren. Methoden dieser Klasse werden in diesem Projekt dazu benutzt, um in der CAD-Präsentation Kooperationselemente zu selektieren.

Weitere Erläuterungen zu den Objekten des API und deren Verwendung finden sich in der Dokumentation zum SolidWorks API [SWap-98].

Ausführen der Zusatzanwendung

Das Laden von Zusatzanwendungen wird in SolidWorks vom Zusatzanwendungs-Manager übernommen, auf den man aus dem Menü *Extras* zugreifen kann.

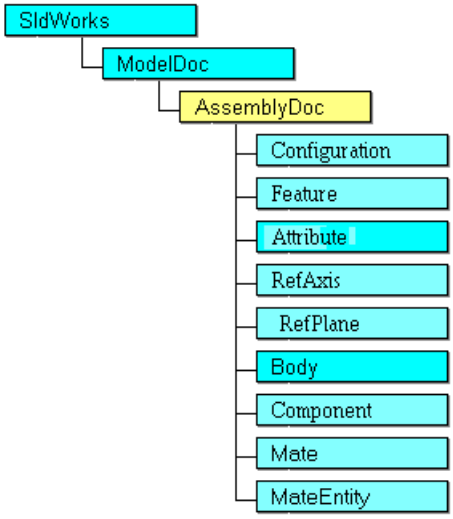

Abbildung 2.6: API-Klassenstruktur, Teil 3

Um eine Anwendung hinzuzufügen, muß sie in der Windows Registrierung angemeldet sein; dies wird vom SolidWorks AddIn Manager im Microsoft Developer Studio (Kap.2.2.3) automatisch erledigt.

Folgende Schritte sind dann noch zum Laden (oder Entfernen) notwendig:

1. Starten einer SolidWorks Sitzung
2. Öffnen oder Neuerstellen eines SW-Dokuments
3. Menü*Datei, Extras, Zusatzanwendungen*
4. Bei hinzuzufügenden Anwendungen das entsprechende Kontrollkästchen mit einem Häkchen versehen (zum Entfernen der Anwendung das Häkchen entfernen)
5. OK klicken

Ist die Zusatzanwendung einmal geladen, wird sie automatisch bei jedem Start von SolidWorks geladen.

2.2.3 Programmiersprache C++ und Programmierumgebung

Die Programmiersprache C++ wurde federführend von Bjarne Stroustrup Anfang der Achtziger Jahre in den AT&T-Bell-Laboratorien entwickelt. 1985 brachte AT&T den ersten kommerziellen C++-Compiler auf den Markt und ab Dezember 1989 existierte ein ANSI-Komitee zur Standardisierung von C++.

C++ kann als Erweiterung von C zu einer objektorientierten Programmiersprache angesehen werden; es wurde starker Wert darauf gelegt, daß C eine echte Untermenge von C++ bildet.

Objektorientierung bedeutet, daß Datenmengen und die Methoden zur Manipulation dieser Daten (einschließlich ihrer Erzeugung und Beseitigung) in sogenannten Objekten zusammengefaßt werden. Eine Klasse ist der Typ eines Objekt; sie beschreibt Typ und Aufbau der Datenkomponenten und Methoden. Die Methoden sind an die Klasse gebunden und werden durch "Botschaften" aktiviert, die an das Objekt gerichtet werden.

Durch Objektorientierung, starke Typenkontrolle und Modularität werden Korrektheit und Robustheit der Programme gefördert; die Möglichkeit von Vererbung[5] und Polymorphie[6] garantieren Erweiterbarkeit und Wiederverwendbarkeit der Objektklassen.

Die beschriebenen Eigenschaften von C++ verlangen und fördern eine gute Strukturierung von Programmen; ebenso wird Teamarbeit erleichtert, was bei in der heutigen Zeit vorkommenden Großprojekten (100000 Codezeilen und mehr) unerläßlich ist [Stro-94][RRZN-98].

Microsoft Visual C++ und Microsoft Developer Studio

Das Microsoft Developer Studio (MSDEV) ist eine Programmierumgebung unter MS Windows. In dieser Arbeit wird es zur Programmentwicklung in der Programmiersprache Visual C++ verwendet. Visual C++ basiert auf Standard-C++, beinhaltet jedoch zusätzlich umfangreiche Klassenbibliotheken für die Programmierung von MS-Windows-Anwendungen, die Microsoft Foundation Classes (MFC).

Das Developer Studio bietet eine Oberfläche zur Programmentwicklung (Abbildung 2.7), die zum einen eine grafische Übersicht über die verwendeten Klassen und Resourcen bietet (z.B. im Class View), und in der ein großer Teil der administrativen Programmierarbeit (z.B. beim Einfügen neuer Klassen, Funktionen oder Variablen) dialogbasiert vorbereitet und automatisch ausgeführt wird. Als wichtigste Beispiele seien genannt:

Neues Projekt: Das Erstellen eines neuen Projektes kann mit dem Application Wizard erfolgen, welcher einen Arbeitsbereich in einem eigenen Projektverzeichnis anlegt. Je nach Art der Anwendung (eigenständiges Programm, DLL, etc.) werden die erforderlichen Dateien und Verzeichnisse vorbereitet; ebenso werden ggf. Standardfenster mit Windows-typischen Menüs in das Projekt eingefügt, welche mit den zu programmierenden Objekten verknüpft werden können.

[5]Vererbung: Klassen werden aus bereits definierten Klassen abgeleitet und "erben" deren Eigenschaften (zusätzlich zu den in der neuen Klasse definierten Eigenschaften)

[6]Polymorphie: Eigenschaft einer Botschaft, daß ihre Wirkung davon abhängt, an wen sie gerichtet ist

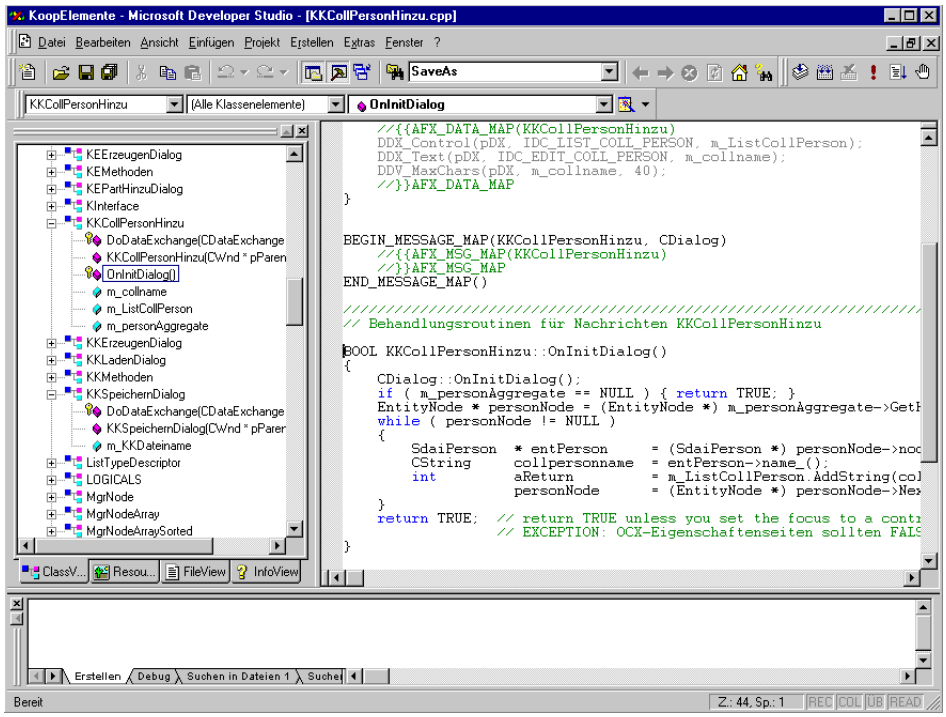

Abbildung 2.7: Microsoft Visual C++ Programmierumgebung

Neue Klassen und deren Elemente: Neue Klassen können sehr einfach einem Projekt hinzugefügt werden; dialogbasiert werden erforderliche Angaben abgefragt (Klassenname, ggf. Oberklassen oder "befreundete" Klassen) und es werden für diese Klasse je eine Header- und eine C++-Datei angelegt und Sourcecode für die Klassendeklaration in diese eingefügt. Im Class View können in diese Klassen die Elemente (Memberfunktionen und Membervariablen) ebenso dialogbasiert angelegt werden; die Definitionen werden automatisch den Klassendateien hinzugefügt.

Dialoge: MSDEV unterstützt das Entwerfen von Dialogboxen und Bildschirmmasken im Microsoft-typischen "look and feel". Die Resourcen für die Dialoge können in einer grafischen Oberfläche (Abbildung 2.8) erstellt und mit wenig Aufwand mit dazugehörenden Dialogklassen verknüpft werden.

Installation des API mit Visual C++

Für das Erstellen und Kompilieren von Zusatzanwendungen unter C++ wird Microsoft Visual C++, Version 5.0 (VC++) mit der Entwicklungsoberfläche MS Developer Studio verwendet. Die SolidWorks-Dokumentation empfiehlt eine Vollinstallation des MSDEV. Bei benutzerdefinierter Installation ist folgendes zu beachten:

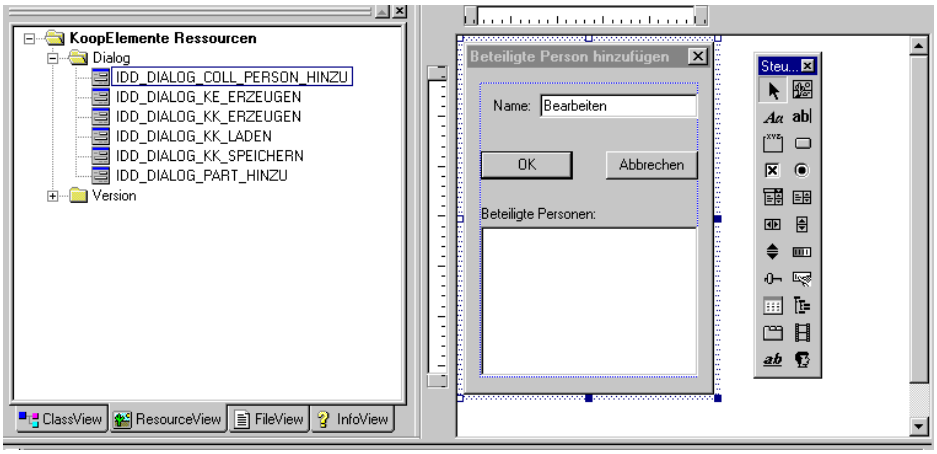

Abbildung 2.8: Microsoft Visual C++ Resource-View

- Für *MS Windows NT* müssen alle UNICODE Bibliotheken installiert werden.
- Für *MS Windows 95/98* müssen MBCS Bibliotheken installiert werden (dies sind die Standardbibliotheken für MSDEV).
- Für *Alpha* müssen alle UNICODE Bibliotheken installiert werden.

Dem Application Wizard des MSDEV kann eine Option für SW-API-Projekte ("SolidWorks AddIn") hinzugefügt werden; das Projekt wird als Dynamic Link Library (DLL) erstellt und automatisch für das Betriebssystem registriert.

Die SolidWorks-Zusatzanwendungen werden mit einer regulären SolidWorks Installation eingesetzt.

Erstellen eines Projektes und Konfiguration im MSDEV

Mit dem Application Wizard des MSDEV wird ein neuer Projekt erstellt; Name und Verzeichnis des Arbeitsbereiches werden dialogorientiert eingegeben. MSDEV trägt die DLL automatisch in die Registry des Betriebssystems ein. Danach muß die aktive Konfiguration auf *Win32 Release* eingestellt werden über das Menü *Erstellen, Aktive Konfiguration festlegen.*

Zugriff auf SolidWorks-Objekte

Wenn eine Zusatzapplikation für SolidWorks erstellt wird, dann erzeugt der SW-Add-In-Manager eine Klasse als Basis für die Applikation, deren Name vom Projektnamen abgeleitet wird; im vorliegenden Projekt *KoopElemente* lautet der Klassenname *CKoopElementeApp*. In dieser Klasse werden u.a. die Methoden eingefügt, die mit den neu hinzugefügten Menüpunkten verknüpft werden; außerdem

enthält die Klasse die Membervariable *m_pSldWorks* vom Typ *LPSLDWORKS*, welche einen Zeiger auf die aktuelle SolidWorks-Sitzung enthält (siehe auch in der Beschreibung der API-Klassenstruktur in Kap.2.2.2).

In der Applikation wird diese Klasse global instanziiert als Zeiger mit dem Namen *TheApplication*:

```
CKoopElementeApp* TheApplication = NULL;
```

Initialisiert wird *TheApplication* in der Funktion *InitUserDLL3*, welche beim Start der Applikation aufgerufen wird. Um auch außerhalb der Klasse *CKoopElementeApp* auf die als *private* deklarierte Variable *m_pSldWorks* zugreifen zu können, wird eine Methode hinzugefügt, die den Wert dieser Variable zurückliefert:

```
LPSLDWORKS CKoopElementeApp::getSWApp()
{
    return m_pSldWorks;
}
```

dadurch erhält man mit dem Aufruf

```
TheApplication->getSWApp()
```

einen Zeiger auf das *SldWorks*-Objekt und kann auf dessen Methoden zugreifen. Wichtig sind dabei besonders folgende Methoden:

IFrameObject liefert einen Zeiger auf das *Frame*-Objekt der Applikation. Dadurch kann die Menüstruktur erweitert werden, siehe im Abschnitt "Hinzufügen neuer Menüpunkte".

INewPart erzeugt ein neues SolidWorks-Dokument für ein Part und liefert gleichzeitig auch einen Zeiger auf das neue Part-Dokument (*LPPARTDOC*). Diese Methode wird bei der Visualisierung der Kooperationselemente benötigt.

get_IActiveDoc liefert einen Zeiger vom Typ *LPMODELDOC* auf das aktuelle SolidWorks-Dokument:

```
LPMODELDOC pModelDoc = NULL;
HRESULT hres;
hres = TheApplication->getSWApp()->get_IActiveDoc(pModelDoc);
```

Diese Methode ist bei nahezu jedem Zugriff auf vorhandene oder zu erstellende CAD-Elemente beteiligt.

Über die Methoden von *ModelDoc* kann man dann in tiefere Ebenen der API-Klassenstruktur vordringen um auf Features oder andere Geometrie-Elemente zuzugreifen, z.B. über den Selectionmanager (*SelectionMgr*).

Wie in diesem Projekt die Kooperationselemente in SolidWorks identifiziert werden, wird in Kap.4.2.1 genau beschrieben.

Hinzufügen neuer Menüpunkte

Ausgehend von einer Instanz der Klasse *LPFRAME*[7], die ihren Inhalt von der aktuellen SolidWorks Session erhält, können mit der Methode *AddMenu* neue Menüs und mit der Methode *AddMenuItem* Menüpunkte der bestehenden SW-Menüstruktur hinzugefügt werden. Mit der Methode *AddMenuPopupItem* werden Menüpunkte in das PopUp-Eigenschaftsmenü eingefügt. Namen und Positionen der hinzugefügten Objekte werden mit diesen Methoden festgelegt und bei Menüpunkten außerdem noch die Funktionen der Applikation, welche mit den Menüpunkten verknüpft werden sollen. Bei Klick auf den Menüpunkt in der SolidWorks Session wird die entsprechende Funktion aufgerufen.

Beispiel:

```
HRESULT hres;
LPFRAME pFrame;
hres = m_pSldWorks -> IFrameObject(&pFrame);

hres = pFrame -> AddMenu(     // Menü hinzufügen
  auT(‘‘&Menu 1’’),           // Name des Menüs
  swMenuPosition,             // Position des Menüs
  &bres);                     // Rückgabewert

hres = pFrame -> AddMenuItem( // Menüpunkt hinzufügen
  auT(‘‘&Menu’’),             // Name des Menüs
  auT(‘‘Menu-&Item’’),        // Name des Menüpunktes
  swLastPosition,             // Position des Menüpunktes
  auT(‘‘Koope9@function,      // Aufzurufende Funktion
  Beschreibung’’),            // Beschreibung der Funktion,
                              // wird in der Statuszeile angezeigt
  &bres);                     // Rückgabewert

hres=pFrame->AddMenuPopupItem(// Menüpunkt zum Eigenschaftenmenü
  swDocPART,                  // in welchem Dokumenttyp (Part,...)
  swSelVERTICES,              // bei welchem SW-Objekt
  auT("Mark vertex")          // Name des Menüpunktes
  auT("Koope9@MarkVertex,     // Aufzurufende Funktion
  Mark Vertex"),              // wird in der Statuszeile angezeigt
  NULL,
  0,
  &bres);                     // Rückgabewert
```

[7] *LPFRAME*: Zeiger auf *Frame*

2.3 NIST Step Class Library

Die NIST STEP Class Library (SCL) ist wesentlicher Bestandteil dieses Projektes; daher wird ihr zur Beschreibung ein eigenes Unterkapitel gewidmet.

Zusätzlich wird noch die Klasse *Infomodel* beschrieben, die einfachen Zugriff auf die grundlegende Funktionalität der SCL bietet: Lesen und Schreiben von STEP-Austauschdateien, Erzeugen von STEP-Entities, etc.

2.3.1 Ursprung der STEP Class Library

Die STEP Class Library wurde am National Institute of Standards and Technology (NIST) entwickelt. Die SCL ist eine C++ Klassenbibliothek zur Unterstützung von Entwicklung von STEP Softwareapplikationen auf Basis von EXPRESS-Datenmodellen[8]. Die Software bietet einen Überblick über EXPRESS Schema Informationen und Funktionalität zur Repräsentation und Manipulation von Instanzen von EXPRESS Datenobjekten. Die SCL wurde in ihrer Entwicklung an die STEP Norm ISO10303[9] angepaßt, unter besonderer Berücksichtigung der Implementierungsmethoden der Norm (Part 21[10], Part 22[11] und Part 23[12]). Ziel bei der Entwicklung der SCL war unter anderem, STEP zu größerer Akzeptanz und Verbreitung zu verhelfen und kommerzielle Entwicklung von STEP-Applikationen zu fördern[SaMo-95].

2.3.2 EXPRESS Sprachelemente und zugeordnete SCL-Klassen

Nachfolgend werden die für dieses Projekt wichtigsten EXPRESS-Sprachelemente und die für diese relevanten SCL-Klassen vorgestellt:

- Ein **Entity** in EXPRESS wird zur Modellierung von Objekt- und Beziehungstypen verwendet. Die Beschreibung von Vererbung zwischen Entities wird in EXPRESS unterstützt.
  Einem *Entity* im EXPRESS-Modell wird in der SCL die Klasse *STEPentity* zugeordnet. Sie ist damit Basis für alle Klassen zur Instanziierung von Entities. In ihr werden Funktionalitäten definiert, die allen Entities gemein sind, z.B. Lesen aus und Schreiben in Part-21-Dateien, Validierung von Attribut-Werten oder generischen Zugriff auf Attribute (siehe nächster Absatz).

[8] ISO10303 Part 11: Description methods: The EXPRESS Language Reference Manual
[9] ISO 10303: Product Data Representation and Exchange
[10] Implementation Methods: Clear text encoding of the exchange structure
[11] Implementation Methods: Standard Data Access Interface (SDAI)
[12] Implementation Methods: C++ language binding to SDAI

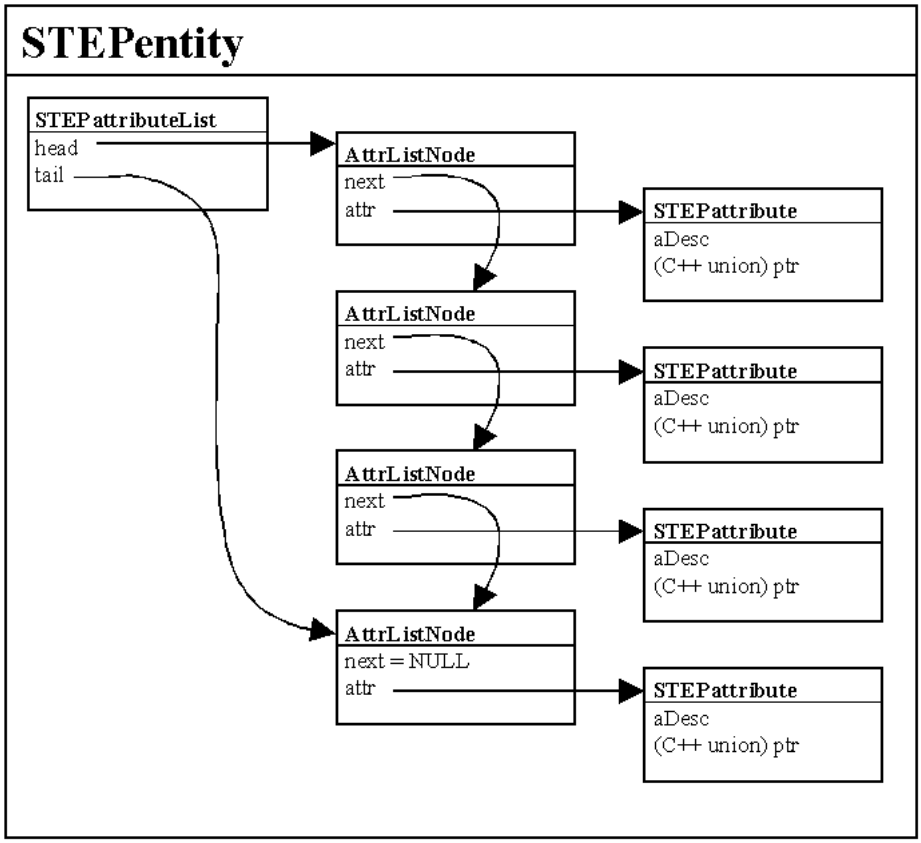

Abbildung 2.9: Beziehungen zwischen Listenelementen in einer Aggregation vom Typ LIST

- **Attribute** bezeichnen die Merkmale der Entities. Auch Beziehungen werden als Attribute von Entities abgebildet. Die Kardinalitäten der Beziehungstypen werden durch die Typen der Attribute (optionale Attribute, inverse Attribute) und durch Zwangsbedingungen auf die Attribute abgebildet.
  Dem *Attribut* in EXPRESS entspricht die Klasse *STEPattribute* in der SCL. Zugriff auf die einzelnen Attribute eines *STEPentity* erfolgt über eine verkettete Liste vom Typ *STEPattributeList*; zur Erläuterung wird auf Abb.2.9 und die Beschreibung von LIST-Aggregationen auf Seite 24 verwiesen.

- Im EXPRESS Datenmodell müssen für die Attribute von Entities deren Typen angegeben werden (vergleichbar mit einer Variablendeklaration in einer Programmiersprache). Dazu bietet EXPRESS unter anderem folgende **vordefinierte Basisdatentypen** an: (SCL-Deklaration in Klammern):

  REAL: (SCLP23(Real)) Gleitkomma-Datentyp

STRING: (SCLP23(String)) String-Datentyp
BOOLEAN: (SCLP23(BOOL)) Logischer Datentyp, TRUE oder FALSE

Neben diesen sogenannten *Simple Types* sind außerdem noch verschiedene Aggregationstypen (ARRAY, BAG, SET, LIST) verfügbar[13].
STEPattributeList entspricht einer Aggregation vom Typ LIST, und wird, da im vorliegenden Datenmodell häufig verwendet, im folgenden Absatz genauer erläutert.

- Eine **Aggregation** vom Typ **LIST** (z.B. Abb.2.9) enthält eine verkettet Liste von Elementen gleichen Typs, im Allgemeinen Attribute, also Entities oder Simple Types. Als C++ Klasse in der SCL umgesetzt besteht eine solche Liste aus einem Element, dessen Typ direkt oder indirekt von der Klasse *SingleLinkList* abgeleitet ist; dies kann z.B. die Klasse *RealAggregate* für eine Liste von Gleitkommazahlen, die Klasse *EntityAggregate* für eine Liste von Entities oder ähnliches sein. *SingleLinkList* enthält Funktionalität zur Verwaltung und Verarbeitung von solchen Listen; das beinhaltet u.a.

  – Zeiger auf erstes und letztes Element der Liste
  – Methoden zum Hinzufügen und Entfernen eines oder mehrerer Listenelemente
  – Zugriff auf die Anzahl der Listenelemente

  Diese Eigenschaften werden an alle Arten von Listen vererbt. In den Unterklassen werden typspezifische Methoden oder Operatoren hinzugefügt und gegebenenfalls überladen.
  Der Typ der Listenelemente ist i.A. abgeleitet von der Klasse *SingleLinkNode*[14], z.B. *RealNode* oder *EntityNode*. Diese Unterklassen (nodes) enthalten eine Variable, in welcher das zu speichernde Element abgelegt wird, in der Beispielabbildung *(C++ union) ptr*[15]; dies kann ein Basistyp oder auch ein beliebiges Entity sein, als Beispiele seien hier genannt:

  – Die Klasse *RealNode* enthält die Variable *value* vom Typ *Real*[16]
  – Die Klasse *EntityNode* enthält die Variable *node* vom Typ *Application_instance*, es lassen sich dadurch Listen von Entities aus dem Datenmodell erstellen, da deren Klassen alle mittelbar oder unmittelbar von *Application_instance* abgeleitet werden.

  Jeder *node* hat dazu noch einen zweiten elementaren Bestandteil, einen Zeiger auf seinen Nachfolger in der Liste[17].

[13] von *aggregation*=(engl.) Zusammenfassung
[14] Listenelemente werden hier als *nodes*=(engl.) Knoten bezeichnet
[15] C++ Schlüsselwort *union*: Variablentyp wird erst bei Initialisierung festgelegt
[16] Ein Beispiel für die Verwendung findet man in Kap.4.5.1
[17] Beim letzten Element einer Liste enthält dieser natürlich den Nullzeiger

2.3.3 Der Instanzenmanager (Klasse InstMgr)

Die Klasse *InstMgr* aus der SCL dient der Verwaltung der instanziierten Kooperationselemente (Entities).

In der Membervariable *master* sind in einer Liste sämtliche Entities abgelegt; über die Methoden von *InstMgr* kann auf diese zugegriffen werden. Die Methoden bieten Informationen über die Entities, deren Anzahl, Funktionalität zum Erzeugen und Löschen derselben. Auf folgende Methoden wird in diesem Projekt direkt zugegriffen:

Delete: Löscht das Entity, welches als Parameter übergeben wird, aus der Liste der Entities.

InstanceCount: liefert als Ergebnis die Anzahl der Entities.

In den meisten Fällen wird jedoch über die Methoden der Klasse *Infomodel* (Kap.2.3.4) auf die Entities zugegriffen.

2.3.4 Die Klasse Infomodel

Die Klasse *Infomodel* faßt Methoden zusammen, die den Umgang mit den SCL-Klassen und den Entities aus dem Datenmodell erleichtern sollen. Die Klassen bietet Methoden zur Behandlung von Entites, deren Erzeugung und Zerstörung, Lesen und Schreiben von STEP-Dateien.

Wesentlicher Bestandteil ist eine Instanz des *InstMgr*. Im Konstruktor von *Infomodel* wird *InstMgr* mit dem Schlüsselwort *static* instanziiert:

```
static InstMgr Const_instance_list;
```

d.h. während der gesamten Laufzeit des Programms[18] wird dieses Objekt existieren und seinen Wert behalten. Dadurch ist gewährleistet, daß, auch wenn *Infomodel* immer lokal instanziiert wird, immer der gleiche Datenbestand im Instanzenmanager bearbeitet wird.

Nachfolgend werden die verwendeten Methoden beschrieben:

CreateEntity: Diese Methode erzeugt ein neues Entity in der Liste des *InstMgr*. Als Parameter wird der Name des Entities übergeben.
Diese Methode wird beim Erzeugen eines jeden Entities aus dem Datenmodell verwendet.

DeleteEntity: Löscht das Entity, welches als Parameter übergeben wird.

GetEntity: Liefert einen Zeiger auf das durch den Parameter bestimmte Entity.

[18] in diesem Fall SolidWorks mit der SW Zusatzanwendung *Koopelemente.dll*

GetInstanceList: Liefert einen Zeiger auf den Instanzenmanager zurück. Darüber kann man direkt auf die Methoden von *InstMgr* zugreifen und hat erweiterte Kontrolle über die Entities.

SetFileName: Diese Methode bekommt als Parameter den Dateinamen, den die mit *WriteExchangeFile* zu schreibende STEP-Datei erhält.

WriteExchangeFile: Schreibt die STEP-Datei mit allen Entities aus der Liste des Instanzenmanagers. Als Parameter wird das Schreibmedium übergeben,
PRINT_FILE für das Schreiben in eine Datei,
PRINT_CONSOLE für die Ausgabe auf der Konsole.

ReadExchangeFile: Liest eine STEP-Datei, deren Name als Parameter übergeben wird. Die Entities aus der Datei werden erzeugt und in der Liste des Instanzenmanagers abgelegt.

Kapitel 3

Konzeption

3.1 Grobkonzeption

Die Daten der Kooperationselemente werden in Instanzen von Klassen abgelegt, welche automatisch aus dem EXPRESS-G-Datenmodell abgeleitet werden (Koop-Klassen). Zur allgemeinen Verwaltung und Verarbeitung dieser Instanzen bietet die Step Class Library (Kap. 2.3) geeignete Methoden an; teilweise erfolgt der Zugriff über das Infomodel(2.3.4). Zu entwerfen sind also noch die Methoden, in denen die Kooperationselemente instanziiert werden und die entsprechenden Daten aus dem CAD-Modell gelesen und den Instanzen zugeordnet werden. Dabei bietet sich eine Trennung von systemunabhängigen Methoden (Methodenbank) und Schnittstellenmethoden (Interface) an; dadurch ist bei Realisierung des Projektes mit einem anderen CAD-System nur eine Anpassung der Schnittstelle notwendig. Die Wiederverwendbarkeit von bestehenden funktionierenden Klassen und Strukturen ist gewährleistet. Das Konzept ist in Abb. 3.1 dargestellt:

- Der Datenbereich, d.h. die CAD-Daten im **SolidWorks** und die Instanzen der **Koop-Klassen**, ist dunkelgrau unterlegt. Die aus dem EXPRESS-G-Datenmodell abgeleiteten Koop-Klassen werden unter Visual C++ unverändert in das Projekt eingefügt.

- Vorgefertigte Klassen zur Manipulation und Verwaltung der Daten werden durch den hellgrau unterlegten Bereich dargestellt. Auf die Koop-Klassen wird über die Klassen der **SCL** und die Klasse **Infomodel** zugegriffen, das **API** ermöglicht den Zugriff auf die CAD-Daten. SCL und Infomodel werden, wie die Koop-Klassen, unverändert in das Projekt eingefügt; das API ist Grundbestandteil des mit dem SolidWorks-AddIn-Manager erstellten VC++-Projektes, dem die anderen Klassen hinzugefügt werden (siehe Kap.2.2.3).

- Die **Methodenbank** und das **Interface**, in der Abbildung weiß dargestellt, sind der Bereich des Projektes, der noch zu erstellen ist. Dieser Bereich

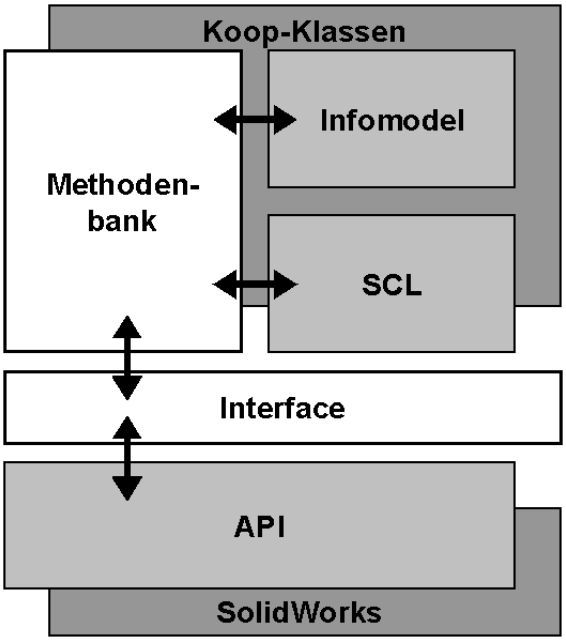

Abbildung 3.1: Programmkonzept

enthält die Klassen, die die Funktionalität zum Erzeugen, Speichern, Laden und Visualisieren der Kooperationselemente enthält.

3.2 Konzeption der Programm-Module

Bei der Programmentwicklung wird man sich den Kriterien Übersichtlichkeit und Wiederverwendbarkeit durch Modularisierung und Kapselung annähern, d.h. durch einen objektorientierten Ansatz, wie er in der Programmiersprache C++ vorgesehen ist und auch verlangt wird. Dieser Ansatz konkurriert jedoch im vorliegenden Projekt ein wenig mit der Vorgabe der Portierbarkeit der aus dem Datenmodell[1] abgeleiteten Klassen, da diese unverändert in das Projekt eingebunden werden sollen; ein rein objektorientierter Ansatz würde verlangen, daß in den Objekten (hier die Entities aus dem Datenmodell, z.B. ein Kooperationselement) nicht nur die Daten, sondern auch die Methoden zur Manipulation der Daten enthalten sind. In den abgeleiteten Klassen wird jedoch nur grundlegende Funktionalität zur Verwaltung zur Verfügung gestellt. Daher wird auf das Methodenbankkonzept zurückgegriffen. Nach diesem Konzept wird in einer Methodenbank Funktionalität zur Manipulation und Verarbeitung von Objekten abgelegt.

[1] siehe Kapitel 2.1

Dementsprechend werden im Projekt zwei Klassen als Methodenbank hinzugefügt:

- **KKMethoden:** enthält Methoden zur Manipulation von Kooperationskontexten, d.h. Erzeugen, Speichern, Laden, Löschen und natürlich auch Zuweisen eines Namens und der Daten; außerdem Funktionen, die Informationen über Eigenschaften des Kontext zurückliefern, z.B. Anzahl der enthaltenen Kooperationselemente, Name des Kontext, ...

- **KEMethoden:** enthält Methoden zur Manipulation von Kooperationselementen, d.h. Erzeugen, Ermitteln der Geometriedaten, Zuweisen der Daten; ein wesentlicher Teil sind hier auch die Funktionen zur Präsentation der Kooperationselemente, d.h. Erzeugen der Referenzgeometrie im CAD-System.

Um die Portierbarkeit der Methodenbank auf ein anderes CAD-System zu erleichtern, ist es sinnvoll, zwischen Methodenbank und API[2] eine Schnittstelle[3] zu installieren, die die Nachrichten, die zwischen diesen beiden Systemen ausgetauscht werden, in das benötigte Format übersetzt.

Diese Schnittstelle wird durch folgende Klasse dargestellt:

- **KInterface:** enthält Methoden, um auf Nachrichten, die von der Methodenbank geschickt werden, zu reagieren (siehe Abb.3.1 und Abb.3.2).
  Wenn diese Nachricht z.B. eine Anfrage nach Geometriedaten eines Kooperationselementes ist, leitet KInterface diese über das API weiter an die CAD-Objekte und erhält die Geometriedaten im vom API zur Verfügung gestellten Format als Antwort. Die Daten werden durch Funktionen des KInterface in das von der STEP-Norm geforderte Format der Geometriebeschreibung übersetzt[4] und an die Methodenbank als Antwort auf die Nachricht zurückgeliefert.
  Ein anderes Beispiel ist die Präsentation eines Kooperationselements im CAD-System; die Methodenbank liefert einen entsprechenden Auftrag an KInterface inklusive der Geometriedaten im STEP-Format. Die Daten werden in das vom API geforderte Format übersetzt und die entsprechenden Funktionen des API aufgerufen, um die Referenzgeometrie im CAD zu erzeugen.

Weitere Klassen werden für die Dialogverwaltung benötigt; diese können in Visual C++ direkt mit den Dialogresourcen[5] erstellt und verknüpft werden; weitere Erläuterungen zum Thema Dialog finden sich in Kapitel in Kapitel 4.6.

[2] Application Programming Interface, siehe Kap.2.2.2

[3] Schnittstelle = interface

[4] z.B. kann eine Ebene in Parameterform oder in Normalenform dargestellt werden

[5] Dialogboxen

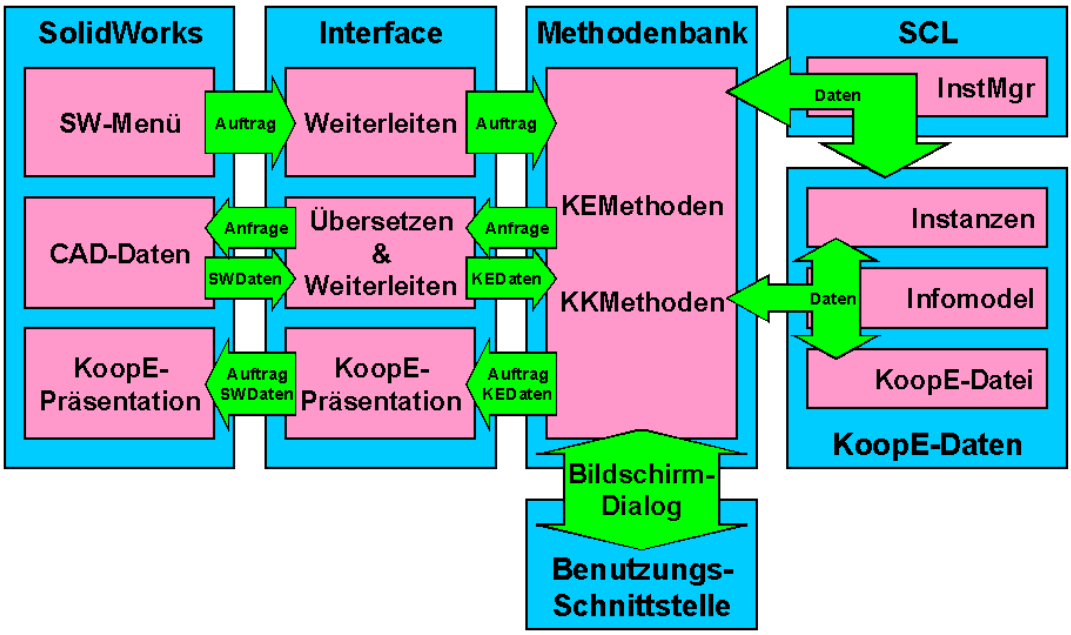

Abbildung 3.2: Programmstruktur

3.3 Struktur der Klassenmodule

3.3.1 Programmtechnische Anforderungen

Eine mit Visual C++ und dem SW AddIn Wizard erstellte Zusatzanwendung bietet folgende grundlegende Möglichkeit, SolidWorks zu erweitern:

- Hinzufügen neuer Menüs und Menüpunkte
- Verknüpfen von Menüpunkten und Funktionen aus der Zusatzanwendung; d.h. beim Wählen des Menüpunktes wird die entsprechende Funktion ausgeführt.
- Gleiches gilt für Menüpunkte im Eigenschaftenmenü, welches als Popupmenü erscheint, wenn ein Objekt mit der rechten Maustaste selektiert wird.

Daraus und aus dem KInterface-Konzept ergibt sich die in Abb.3.2 erkennbare Struktur in der Klasseninteraktion, die an einem fiktiven Beispiel erläutert wird, in dem der Menüpunkt **Aktion ausführen** mit einer Methode **Aktion** aus der Methodenbank, z.B. aus der Klasse **KKMethoden**, verknüpft werden soll:

Bei Auswahl des (fiktiven) Menüpunktes **Aktion** wird die Funktion **CKoopElemente::Aktion** ausgeführt:

```
void CKoopElemente::Aktion()
{
    KInterface * iface = new KInterface;
    iface->Aktion();
    delete KInterface;
    return;
}
```

Es wird eine Instanz des KInterface erzeugt und die Methode **KInterface::Aktion** aufgerufen, welche folgenden Aufbau hat:

```
void KInterface::Aktion()
{
    KKMethoden * kkmeth = new KKMethoden;
    kkmeth->Aktion();
    delete kkmeth;
    return;
}
```

In der Methode wird eine Instanz von KKMethoden erzeugt und die Methode **KKMethoden::Aktion** aufgerufen.

Nach diesem Prinzip werden sämtliche Anfragen über das Interface weitergeleitet. Das Interface kann natürlich auch Anfragen in entgegengesetzter Richtung weiterleiten, Es könnte eine Methode aus der Methodenbank das Interface instanziieren und eine Methode des Interface aufrufen, die z.B. Methoden aus dem API aufruft und auf SolidWorks-Daten zugreift.

Es ist natürlich ebenso möglich, mit den Anfragen Parameter zu übergeben oder Rückgabevariablen zu definieren.

3.3.2 Anforderungen an die Menüstruktur

Folgende Anforderungen an die Menüstruktur der Zusatzanwendung sind vorgegeben:

- Menü **Kooperationselemente** in die SW-Menüstruktur einfügen.
- Menüpunkt **Kooperationskontext erzeugen** instanziiert einen Kooperationskontext mit seinen Attributen. Dialogorientierte Eingabe von Daten ist vorzusehen.
- Menüpunkt **Kooperationselement erzeugen** instanziiert das aktuell selektierte SW-Element als Kooperationselement; gleichzeitig wird es in einem eigenen SolidWorks-Part als Referenzgeometrie visualisiert. Die nichtgeometrischen Daten sind dialogorientiert einzugeben.

- Menüpunkt **Kooperationskontext speichern** speichert den aktuellen Kooperationskontext mit allen Kooperationselementen in einer STEP-Datei.
- Menüpunkt **Kooperationskontext laden** lädt einen auszuwählenden Kooperationskontext aus einer STEP-Datei, die Kooperationselemente werden als Instanzen erzeugt und in einem eigenen SolidWorks-Part als Referenzgeometrie visualisiert.

Das Menü und alle geforderten Menüpunkte werden implementiert. Zusätzlich wird ein Menüpunkt **Kontext Löschen** eingefügt, mit der der aktuelle Kontext komplett entfernt werden kann.

Mit diesen Menüpunkten werden entsprechende Funktionen in der Zusatzanwendung verknüpft, die analog zu dem in Kap.3.3.1 beschriebenen Verfahren über das Interface auf die Methodenbank zugreifen.

Zusätzlich wird für das Erzeugen von Kooperationselementen ein Menüpunkt in den Eigenschaftenmenüs der SW-CAD-Elemente eingefügt.

Die Beschreibung der Menüs und der zur Dateneingabe erforderlichen Dialogboxen erfolgt in Kapitel 4.6.

Kapitel 4

Implementierung

In diesem Kapitel wird anhand von Auszügen aus dem Quelltext die Implementierung der Methoden erläutert, die zum Erzeugen, Speichern, Laden und Visualisieren der Kooperationselemente im Kooperationskontext benötigt werden. Das betrifft, neben dem Zugriff auf die STEP-Datei, das Erfassen der Geometrie und die Instanziierung von Kooperationskontext und Kooperationselementen; bei der Visualisierung den Zugriff auf die instanziierten Kooperationselemente und das Erzeugen der Referenzgeometrie im CAD-System.

4.1 Implementierung der Kontextmethoden

In diesem Abschnitt werden die Methoden erläutert, die zur Erzeugung eines Kooperationskontextes verwendet werden.

- Erster Schritt ist die Instanziierung des Kooperationskontextes mit der *CreateEntity*-Funktion des Infomodel:

  ```
  Infomodel * IM1;
  const SchemaDescriptor * schema1 = IM1.Schema();
  SdaiCooperation_context * entCooperation_context =
      (SdaiCooperation_context *)
      IM1.CreateEntity("Cooperation_context");
  ```

- Dann erfolgt die Zuweisung der Kontext-Daten an die Attribute des Kontextes, hier beispielhaft gezeigt am Attribut *used_in* vom Typ *Product*:

  ```
  SdaiProdukt * entProdukt = ... ;
  entCooperation_context->used_in_(entProduct);
  ```

  In der ersten Zeile ist die Erzeugung des Entites vom Typ *Product* angedeutet, in der zweiten Zeile erfolgt die Zuweisung des *Product* an *used_in*.

Weiterhin werden folgende Methoden zur Kontextverwaltung implementiert:

KontextHolen: Liefert einen Zeiger auf den aktuellen Kooperationskontext zurück; wird u.a. bei Erzeugung eines Kooperationselementes benötigt, um das Attribut *used_in_context* zu belegen.

Da in dieser Implementation genau ein Kontext in einer Datei abgelegt wird, und der Kontext Grundelement des Datenmodells ist, ohne den keine Kooperationselemente erzeugt werden können, wird davon ausgegangen, daß das erste Entity im Instanzenmanager der Kooperationskontext ist:

```
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
InstMgr * instance_list = IM1.GetInstanceList();
STEPentity * kkentity = IM1.GetEntity(0);
SdaiCooperation_context * entCooperation_context
        = (SdaiCooperation_context *) kkentity;
return kkentity;
```

KontextLaden: Lädt einen Kooperationskontext aus einer Datei; siehe Kap.4.4.2.

KontextSpeichern: Speichert den aktuellen Kooperationskontext in einer Datei; siehe Kap.4.4.1.

KontextLoeschen: Diese Methode löscht den gesamten Kooperationskontext, inklusive aller Kooperationselemente. Die Entity-Liste des Instanzenmanagers wird durchlaufen, und die einzelnen Entities werden gelöscht:

```
Infomodel IM1;
...
for (int index = (AnzahlEntities-1); index>=0; index--)
{
   STEPentity * entity = IM1.GetEntity(index);
   IM1.DeleteEntity(entity);
}
```

KeinKontext: Mit dieser Methode läßt sich die Existenz eines Kooperationskontextes überprüfen. Sie liefert den Wert *TRUE*, wenn kein Kontext in der aktuellen Sitzung aktiv ist: entweder wenn keine Entites im Instanzenmanager vorhanden sind, oder wenn das erste Entity kein Kooperationskontext ist.

KontextName: Liefert den Namen des aktuellen Kooperationskontextes als String zurück:

```
SdaiCooperation_context *entCooperation_context
                                = KontextHolen();
CString kkname = entCooperation_context->name_();
return kkname;
```

4.2 Erfassen der Geometrie

Thema in diesem Abschnitt ist das Erfassen der Geometriedaten der zu erzeugenden Kooperationselemente.

Es wird analysiert, welche CAD-Daten SolidWorks über das API zurückliefert. Dabei sind folgende geometrischen Grundelemente zu betrachten, welche im Datenmodell als Kooperationselemente instanziiert werden können:

- Punkt
- Gerade
- Kreis
- Ebene

Nachfolgend wird der Zugriff auf ein selektiertes CAD-Element beschrieben. Dann wird zu jedem dieser Elemente erläutert,

- in welcher Form die geometrische Beschreibung durch welche API-Funktion geliefert wird,
- welche Form das Datenmodell verlangt, und
- in welcher Weise gegebenenfalls eine Umrechnung erfolgt.

4.2.1 Selektion eines Elements

Die Zusatzanwendung ist eine Instanz der Klasse *CKoopElementeApp* mit dem Bezeichner *TheApplication*. *CKoopElementeApp* enthält den Zugang von SolidWorks zum API, die Funktionen, welche zusätzliche Menüs hinzufügen und jene Funktionen, welche mit den neuen Menüpunkten verknüpft werden. Außerdem erlaubt diese Klasse den Zugriff des API auf SolidWorks, sie enthält eine Member-Variable *m_pSolidWorks* vom Typ *LPSLDWORKS*; diese stellt einen Zeiger auf die aktuelle SolidWorks-Session dar.

Die Funktion *getSWApp* liefert diesen Zeiger als Ergebnis zurück, so kann auf dieses Objekt auch von anderen Klassen zugegriffen werden.

TheApplication ist global deklariert, daher auch überall erreichbar.

Über die Funktion *get_IActiveDoc* der Klasse *LPSLDWORKS* erhält man einen Zeiger auf das gerade in der SolidWorks-Session aktive Dokument, in der Konstruktionsphase ein Bauteil (Part) oder eine Baugruppe (Assembly). Dieser Zeiger hat den Typ *LPMODELDOC*.

Über *LPMODELDOC* wiederum erhält man mit der Funktion *get_ISelection Manager* Zugriff auf den sogenannten Selection Manager; dieser bietet unter anderem folgende Funktionalität zu den in SolidWorks selektierten Objekten:

- *GetSelectedObjectType* liefert den Typ des selektierten Objektes (Punkt, Ebene, Achse, Kante,...)
- *IGetSelectedObject2* liefert einen Zeiger auf das selektierte Objekt; unabhängig vom Typ des Objekts hat der Zeiger den Typ *LPUNKNOWN*

Ein Zeiger vom Typ *LPUNKNOWN* kann mit der Funktion *QueryInterface* in einen bestimmten Zeiger umgewandelt werden.

So ergibt sich folgendes Programmgerüst:

```
HRESULT         hres;     // Prüfvariable
long            retval;   // Typ des sel. Objekts
LPSLDWORKS      pSldWorks;// Zeiger auf SW-Session
LPMODELDOC      pModelDoc;// Zeiger auf SW-Dokument
LPSELECTIONMGR pSelMgr;  // Zeiger auf Selektion Manager
LPUNKNOWN       pUnknown; // Zeiger auf Objekt
LPVERTEX        pVertex;  // Zeiger auf Vertex = Eckpunkt

pSldWorks = TheApplication-> getSWApp();
hres      = pSldWorks     -> get_IActiveDoc(&pModelDoc);
hres      = pModelDoc     -> get_ISelectionManager(&pSelMgr);
hres      = pSelMgr       -> GetSelecteObjectType(1, &retval);
hres      = pSelMgr       -> IGetSelectedObject2(1, &pUnknown);

if ( retval == swSelVERTICES )  // wenn Objekt ein Vertex ist
{
  hres = pUnknown->QueryInterface(IID_IVertex,(LPVOID*)&pVertex);
}
```

Wenn ein anderer Objekttyp selektiert wurde, kann dieser ebenfalls mit *GetSelectedObjektType* identifiziert und mit *QueryInterface* einem entsprechenden Zeiger zugeordnet werden.

4.2.2 Punkt

In SolidWorks wird ein Punkt im allgemeinen durch einen sogenannten *Vertex* dargestellt; die geometrische Beschreibung erfolgt hier, wie auch im Datenmodell, durch drei Richtungskoordinaten in einem orthogonalen Koordinatensystem. Daher ist keine Umrechnung erforderlich.

Der Zugriff auf einen Vertex erfolgt über ein Objekt vom Typ *LPVERTEX*; dies ist ein Zeiger auf einen Vertex. Über dieses Objekt wird eine Methode *IGetPoint* aufgerufen, welche die Koordinaten des Punktes in einem Array von zurückliefert:

```
LPVERTEX pVertex = NULL;
... // Zuweisung an pVertex
double pDouble[3]; // Array von drei double-Werten
hres = pVertex->IGetPoint(pDouble);
```

4.2.3 Gerade

Eine Gerade kann bei der Erzeugung eines Kooperationselementes in SolidWorks über verschiedene CAD-Elemente selektiert werden; in der prototypenhaften Implementierung dieses Projekts betrachten wir Achsen[1] und Kanten[2].

Eine Kante wird in SolidWorks als *Edge* bezeichnet, eine Achse als *RefAxis*. das API liefert in beiden Fällen Start- und Endpunkt der Kante in kartesischen Koordinaten, das Datenmodell verlangt jedoch einen Aufpunkt, einen Richtungsvektor und die Länge der Geraden.

Die Umrechnung erfolgt folgendermaßen:

- Der Aufpunkt ist gleich dem Startpunkt.
- Ein Richtungsvektor entspricht der vektoriellen Differenz von Endpunkt und Startpunkt.
- Die Länge entspricht dem Betrag des Richtungsvektors
- Danach wird der Richtungsvektor normiert.

Für Edge und Ref Axis müssen unterschiedliche Objekte erzeugt werden, *LPEDGE* und *IRef Axis*.

Implementation für LPEDGE

Für diesen Fall muß noch geprüft werden, ob es sich bei der Edge um eine geradlinige Kante handelt, denn eine selektierte Kante könnte auch gekrümmt sein[3].

[1] Referenzachsen, Achsen von zylindrischen CAD-Elementen
[2] geradlinige Körperkanten
[3] z.B. ein Kreis, siehe Kap.4.2.4

Dazu liefert *IGetCurve* aus *LPEDGE* einen Zeiger auf eine Kurve (*LPCURVE*), der Kurventyp[4] läßt sich mit der Funktion *Identity* feststellen.

```
LPCURVE pCurve  = NULL;
long    curveID = 0;
hres = pEdge  -> IGetCurve(&pCurve);
hres = pCurve -> Identity(&curveID);
if (curveID == LINE_TYPE)   ... // Gerade
```

Über ein Objekt vom Typ LPEDGE können die Funktionen *IGetStartVertex* und *IGetEndVertex* aufgerufen werden, diese liefern jeweils ein Objekt vom Typ LPVERTEX zurück, nämlich Startpunkt und Endpunkt. Die Koordinaten dieser beiden Punkte können wie in Kap.4.2.2 beschrieben ermittelt werden.

```
LPEDGE   pEdge   = NULL;
LPVERTEX pVertex = NULL;
... // Zuweisung an pEdge
hres = pEdge->IGetStartVertex(&pVertex);
hres = pEdge->IGetEndVertex(&pVertex);
...  // Koordinaten der Punkte holen
```

Implementation für IRefAxis*

In diesem Fall sind folgende Schritten nötig:

- Umwandeln des *LPUNKNOWN* in *LPFEATURE*
- *LPFEATURE* enthält die Funktion *IGetSpecific Feature*, die einen Zeiger auf ein spezifiziertes Feature unbekannten Typs zurückliefert (*LPUNKNOWN*)
- Umwandeln des *LPUNKNOWN* in *IRefAxis** mit *QueryInterface*
- Die Funktion *IGetRefAxisParams* liefert in einem Array aus sechs Werten die Koordinaten von Start- und Endpunkt zurück.

```
IRefAxis* pRefAxis = NULL;
LPUNKNOWN pUnknown = NULL;
LPFEATURE pFeature = NULL;
double    pDouble[6];
hres = pSelMgr  -> IGetSelectedObject2(1, &pUnknown);
hres = pUnknown ->
       QueryInterface(IID_IFeature, (LPVOID *) &pFeature);
hres = pFeature -> IGetSpecificFeature(&pUnknown);
```

[4] hier sind Kreis (*CIRCLE_LINE*) und Gerade (*LINE_TYPE*) relevant

```
hres = pUnknown ->
       QueryInterface(IID_IRefAxis, (LPVOID *) &pRefAxis);
hres = pRefAxis -> IGetRefAxisParams(pDouble);
```

4.2.4 Kreis

Ein Kreis kann in unserem Fall eine kreisförmige Körperkante (Edge) sein, d.h. ein Objekt vom Typ *LPEDGE*. Genau wie bei der geradlinigen Kante wird die Form über die Funktion *Identity* aus LPCURVE verifiziert:

```
LPCURVE pCurve  = NULL;
long    curveID = 0;
hres  = pEdge  -> IGetCurve(&pCurve);
hres  = pCurve -> Identity(&curveID);
if (curveID == CIRCLE_TYPE) ... // Kreis
```

Die Funktion *get_ICircleParams* aus *LPCURVE* liefert die Parameter des Kreises zurück: Mittelpunkt, Richtungsvektor der Achse und den Radius.

```
double  pRetval[7];
hres  = pCurve -> get_ICircleParams(&pRetval);
```

Das Datenmodell verlangt für einen Kreis folgende Werte: Mittelpunkt, Radius und zwei senkrecht aufeinander stehende Achsen, die die Ebene aufspannen, in welcher der Kreis liegt.

Vom Richtungsvektor zu den aufspannenden Achsen (oder Vektoren) kommt man mit folgenden Schritten:

- Man erzeuge einen beliebigen Vektor, der nicht linear abhängig vom Richtungsvektor ist.
- Man bilde das Kreuzprodukt aus diesem Vektor und dem Richtungsvektor; das Ergebnis ist ein Richtungsvektor der ersten Achse, dieser kann ggf. noch normiert werden.
- Aus diesem Ergebnisvektor und dem ursprünglichen Richtungsvektor bilde man wieder das Kreuzprodukt; das Ergebnis ist ein Richtungsvektor der zweiten Achse.

4.2.5 Ebene

Auch bei einer Ebene gibt es, wie bei der Gerade, verschieden CAD-Elemente, die für eine Selektion in Frage kommen; wir betrachten Flächen (Face) und Referenzebenen (RefPlane).

SolidWorks liefert im ersten Fall Aufpunkt (Ursprung) und Normalenvektor, bei der Referenzebene zusätzlich einen Richtungsvektor der X-Achse der Ebene.

Das Datenmodell verlangt, ähnlich wie bei einem Kreis, einen Aufpunkt und zwei Richtungsvektoren, die die Ebene aufspannen.

Die Berechnung erfolgt analog dem Schema, welches beim Kreis angewandt wird; bei der Referenzebene muß nur noch der zweite Richtungsvektor berechnet werden.

Implementierung bei Körperoberflächen

Eine Oberfläche muß nicht zwingend eben sein, daher erfolgt zunächst, analog zum Verfahren bei der Körperkante, die Verifizierung dieser Eigenschaft. Von der Oberfläche (*LPFACE*) erhält man einen Zeiger auf ein Objekt vom Typ Surface (*LPSURFACE*), dies enthält eine Funktion *Identity* mit dem gleichen Zweck wie die gleichnamige Funktion aus *LPCURVE*. Wenn *Identity* den Wert *PLANE_TYPE* zurückliefert, ist die Oberfläche eben, und die Geometrie der Ebene kann mit der Funktion *get_IPlaneParams* ermittelt werden:

```
LPSURFACE pSurface = NULL;
long      retval   = 0;
double    pDouble[6];
hres    = pFace    -> IGetSurface(&pSurface);
hres    = pSurface -> Identity(&retval);
if (retval == PLANE_TYPE)
    {  hres = pSurface -> get_IPlaneParams(pDouble);  }
```

Implementierung bei Referenzebenen

Ähnlich wie bei Referenzachsen, muß ein "Umweg" über ein Feature gemacht werden, bevor die Ebene als Objekt vom Typ *IRefPlane* vorhanden ist. Dann kann mit *IGetRefPlaneParams* die Geometrie abgefragt werden:

```
IRefPlane* pRefPlane = NULL;
LPUNKNOWN  pUnknown  = NULL;
LPFEATURE  pFeature  = NULL;
double     pDouble[9];
hres    =  pSelMgr   -> IGetSelectedObject2(1, &pUnknown);
hres    =  pUnknown  ->
           QueryInterface(IID_IFeature,(LPVOID*)&pFeature);
hres    =  pFeature  -> IGetSpecificFeature(&pUnknown);
hres    =  pUnknown  ->
           QueryInterface(IID_IRefPlane,(LPVOID*)&pRefPlane);
hres    =  pRefPlane -> IGetRefPlaneParams(pDouble);
```

4.3 Instanziieren der Kooperationselemente und Zuweisen der Geometrie

Nach dem in Kap.4.2 erläuterten Erfassen der Geometrie erfolgt das Instanziieren des Kooperationselementes und die Daten (Definition und Repräsentation) werden dieser Instanz zugewiesen. Dieser Vorgang wird in den nachfolgenden Abschnitten beschrieben.

4.3.1 Instanz des KE erzeugen

Zuerst wird unter Benutzung der *CreateEntity*-Funktion aus dem Infomodel für das Kooperationselement eine Instanz der Klasse *SdaiCooperation_element* erzeugt:

```
Infomodel IM1;
SdaiCooperation_element * entCooperation_element;
entCooperation_element =
        (SdaiShape_point *) IM1.CreateEntity("Shape_point");
```

Die Schlüsselwörter *SdaiShape_point* und *Shape_point* weisen auf den Typ des Kooperationselementes hin, jeder andere Typ (*Placement_curve*, *Shaping_face*, etc.) könnte, je nach Bedarf, auch verwendet werden.

An diese Instanz werden zunächst die geometrischen Daten übergeben; abhängig von der Dimensionalität des Kooperationselementes (Punkt, Linie, Kreis, Fläche) wird eine entsprechende Funktion aufgerufen. Jede solche Funktion erhält als Parameter die Geometriedaten, die entsprechend dem in Kap.4.2 beschriebenen Verfahren vorliegen, als Liste von *double*-Werten. Zurückgeliefert wird ein Zeiger auf ein *geometric_representation_item*, genauer gesagt eine Unterklasse davon. Welche Unterklasse dies ist, hängt ebenfalls von der Dimensionalität des Kooperationselementes ab[5]. Das Attribut *associated_geometric_representation* des Kooperationselementes erhält als Wert diesen Zeiger.

In diesem Beispiel wird ein Punkt instanziiert; die dazugehörige Funktion heißt *PunktErzeugen*, hat als Parameter drei Koordinatenwerte und liefert als Ergebnis einen *cartesian_point*:

```
entCooperation_element->associated_geometric_representation_(
    (SdaiGeometric_representation_item *)
    PunktErzeugen(g1, g2, g3) );
```

Das explizite Zuweisen der Geometrie erfolgt in den erwähnten Funktionen (*PunktErzeugen*, ...) und wird in Kap.4.3.2ff. erklärt.

Nach dem Zuweisen der Geometrie wird das Kooperationselement in die Liste der Kooperationselemente im Kontext eingefügt; diese Liste ist als Attribut

[5]Punkt: *cartesian_point*, Linie: *line*, Kreis: *circle*, Fläche: *plane*, siehe dazu auch Abb.2.2

elements eine List-Aggregation von *cooperation_element*; dieses Attribut ist invers als *used_in_context* ein Attribut von *cooperation_element*:

```
EntityAggregate * elementAggregate = new EntityAggregate;
EntityNode      * elementNode      = new EntityNode;
elementAggregate  = entCooperation_context->elements_();
elementNode->node = entCooperation_element;
elementAggregate->AddNode(elementNode);
entCooperation_context->elements_(elementAggregate);
```

4.3.2 Punkt - *cartesian_point*

Anknüpfend an den vorherigen Abschnitt wird die Zuweisung der Daten an die Instanz beschrieben. Die Geometriedaten liegen hier, wie auch bei den nachfolgenden Elementen, bereits im STEP-konformen Format vor, wie es in Kap.4.2 beschrieben wurde.

Zuerst wird das Entity *Cartesian_point* erzeugt:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiCartesian_point entCartesian_point =
    (SdaiCartesian_point *) IM1.CreateEntity("Cartesian_point");
```

Somit existiert das Entity in der Entity-Liste von *InstMgr*. Danach werden eine List-Aggregation von Gleitkommazahlen *RealAggregate* und drei *RealNode* erzeugt, an die die Koordinaten des Punktes übergeben werden:

```
RealAggregate * ra = new RealAggregate;
RealNode      * rn = new RealNode;
rn->value = x1; ra->AddNode(rn);
rn->value = x2; ra->AddNode(rn);
rn->value = x3; ra->AddNode(rn);
```

Die Aggregation wird an das Attribut *coordinates* des *Cartesian_point* übergeben, und zuletzt wird das Attribut *is_defined_in* zugewiesen:

```
entCartesian_point->coordinates_(ra);
entCartesian_point->is_defined_in_(
        kkmeth->KontextHolen()->defines_() );
```

Damit ist die Attribut-Belegung des *Cartesian_point* komplett und das Entity kann als Attribut in ein übergeordnetes Entity (z.B. in die Instanz des Kooperationselementes) eingebunden werden.

Das Attribut *is_defined_in* wird bei jedem Geometrie-Entity auf die gleiche Art belegt; es wird daher bei den folgenden nicht mehr erwähnt.

4.3.3 Richtung - *direction*

Die Datenzuweisung erfolgt hier, wie auch bei allen anderen Geometrieelementen, analog dem Verfahren beim Punkt; sie beginnt mit der Erzeugung des Entities:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiDirection entDirection =
    (SdaiDirection *) IM1.CreateEntity("Direction");
```

Danach werden eine List-Aggregation von Gleitkommazahlen *RealAggregate* und drei *RealNode* erzeugt, an die die Koordinaten des Richtungsvektors übergeben werden:

```
RealAggregate * ra = new RealAggregate;
RealNode      * rn = new RealNode;
rn->value = x1; ra->AddNode(rn);
rn->value = x2; ra->AddNode(rn);
rn->value = x3; ra->AddNode(rn);
```

Die Aggregation wird an das Attribut *direction_ratios* der *direction* übergeben:

```
entDirection->direction_ratios_(ra);
```

Damit ist die Attribut-Belegung von *direction* komplett und das Entity kann als Attribut in ein übergeordnetes Entity (z.B. *vector* oder *axis2_placement*) eingebunden werden.

4.3.4 Vektor - *vector*

Erzeugung des Entities:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiVector entVector =
    (SdaiVector *) IM1.CreateEntity("Vector");
```

Danach wird die Länge des Vektors an das Attribut *magnitude* und ein Entity *direction* mit den Richtungskoordinaten an das Attribut *orientation* übergeben:

```
entVector->magnitude_(realLaenge);
entVector->orientation_(entDirection);
```

Damit ist die Attribut-Belegung von *vector* komplett und das Entity kann als Attribut in ein übergeordnetes Entity (z.B. *line*) eingebunden werden.

4.3.5 Linie - *line*

Erzeugung des Entities:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiLine entLine =
    (SdaiLine *) IM1.CreateEntity("Line");
```

Danach wird ein Entity *cartesian_point* mit den Startpunktkoordinaten an das Attribut *pnt* und ein Entity *vector* mit Richtungskoordinaten und Länge an das Attribut *direction* übergeben:

```
entLine->pnt_(entCartesian_point);
entLine->direction_(entVector);
```

Damit ist die Attribut-Belegung von *line* komplett und das Entity kann als Attribut in ein übergeordnetes Entity eingebunden werden.

4.3.6 Koordinatenkreuz - *axis2_placement*

Erzeugung des Entities:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiAxis2_placement entAxis2_placement =
    (SdaiAxis2_placement *) IM1.CreateEntity("Axis2_placement");
```

Danach wird ein Entity *cartesian_point* mit den Ursprungskoordinaten des Koordinatenkreuzes an das Attribut *location* und zwei Entities *direction* mit Richtungskoordinaten der zwei Achsenrichtungen an die Attribute *axis* und *ref_direction* übergeben:

```
entAxis2_placement->location_(entCartesian_point);
entAxis2_placement->axis_(entDirectionEins);
entAxis2_placement->ref_direction_(entDirectionZwei);
```

Damit ist die Attribut-Belegung von *axis2_placement* komplett und das Entity kann als Attribut in ein übergeordnetes Entity (z.B.*plane* oder *circle*) eingebunden werden.

4.3.7 Kreis - *circle*

Erzeugung des Entities:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiCircle entCircle =
    (SdaiCircle *) IM1.CreateEntity("Circle");
```

Danach wird der Radius des Kreises an das Attribut *radius* und ein Entity *axis2_placement* mit den Mittelpunktkoordinaten und Koordinatenwerten zur Ausrichtung des Kreises an das Attribut *position* übergeben:

```
entCircle->position_(entAxis2_placement);
entCircle->radius_(realRadius);
```

Damit ist die Attribut-Belegung von *circle* komplett und das Entity kann als Attribut in ein übergeordnetes Entity eingebunden werden.

4.3.8 Ebene - *plane*

Erzeugung des Entities:

```
KKMethoden * kkmeth = new KKMethoden;
Infomodel IM1;
const SchemaDescriptor * schema1 = IM.Schema();
SdaiPlane entPlane =
    (SdaiPlane *) IM1.CreateEntity("Plane");
```

Danach ein Entity *axis2_placement* mit Koordinatenwerten zur Ausrichtung der Ebene an das Attribut *position* übergeben:

```
entPlane->position_(entAxis2_placement);
```

Damit ist die Attribut-Belegung von *plane* komplett und das Entity kann als Attribut in ein übergeordnetes Entity eingebunden werden.

4.4 Schreiben und Lesen der Daten

4.4.1 Schreiben

Der aktuelle Kooperationskontext kann mit der Methode *KontextSpeichern* aus der Klasse *KKMethoden* in einer Datei abgelegt werden. Der Kontext und alle zugehörigen Kooperationselemente werden dabei mit sämtlichen definierenden und beschreibenden Daten in eine STEP-Datei geschrieben.

Standarddateiname ist der Name des Kontextes mit der Dateierweiterung *.koop*; dieser kann jedoch vom Benutzer geändert werden:

```
CString dateierweiterung = ".koop";
char * dateiname = CStringToChar(KontextName()+dateierweiterung);
...
Infomodel IM1;
const SchemaDescriptor schema1 = IM1.Schema();
IM1.SetFileName(dateiname);
IM1.WriteExchangeFile(PRINT_FILE);
```

Eine Beispieldatei ist im Anhang angefügt.

4.4.2 Lesen

Die Methode *KontextLaden* beinhaltet Funktionalität zum Lesen einer bestehenden Kooperationskontext-Datei, wie sie entsprechend Kap.4.4.1 erzeugt wurde, und der Präsentation der Kooperationselemente des Kontextes im CAD-System.

- **Lesen der Datei:** Nach Eingabe des Dateinamens durch den Benutzer wird die STEP-Datei geladen:

  ```
  char * dateiname = ...
  Infomodel IM1;
  IM1.ReadExchangeFile(dateiname);
  ```

  Die Instanzen der Kooperationselemente und des Kontextes sind nun vorhanden als ob sie gerade erzeugt worden wären. Über den Instanzenmanager können sie verwaltet werden.

- **Präsentation der Daten:** Die Präsentation der Daten entspricht dem Erzeugen der Referenzgeometrie im CAD-System; die entsprechende Funktion ist *KooperationselementeAnlegen*. Zuerst wird im CAD ein neues Part erzeugt, und ihm wird als Name der Name des Kontextes zugewiesen:

  ```
  LPPARTDOC    pPart;
  LPMODELDOC   pModel;
  HRESULT      hres;
  VARIANT_BOOL retval;
  hres = TheApplication->getSWApp()->INewPart(&pPart);
  hres = TheApplication->getSWApp()->get_IActiveDoc(&pModel);
  CString name = entCooperation_context->name_() + ".SLDPRT";
  _bstr_t newname = name;
  hres = pModel->SaveAs(newname, &retval);
  ```

  Für jedes Kooperationselement im Kontext wird ein passendes Geometrieelement im Part angelegt:

```
KEMethoden * kemeth = new KEMethoden;
EntityNode * elementNode =
    (EntityNode*) KontextHolen()->elements_()->GetHead();
while (elementNode != 0)
{
    kemeth->CADElementAnlegen(
            (SdaiCooperation_element*) elementNode->node);
    elementNode = (EntityNode *) elementNode->NextNode();
}
```

Anschließend wird das Part gespeichert:

```
LPMODELDOC   pModel;
HRESULT      hres;
hres = TheApplication->getSWApp()->get_IActiveDoc(&pModel);
hres = pModel->Save();
```

Eine genaue Beschreibung des Erzeugens der Geometrie erfolgt in Kap.4.5.

4.5 Erzeugen der Referenzgeometrie

In diesem Abschnitt wird auf die Präsentation der Kooperationselemente eingegangen, d.h. auf das Erzeugen der Referenzgeometrie im CAD-System. Ausgangspunkt ist die Methode *CADElementAnlegen* in der Klasse *KEMethoden*. Sie wird aufgerufen

- wenn ein Kooperationselement neu angelegt wurde oder
- wenn ein Kooperationskontext geladen wurde und die darin enthaltenen Kooperationselemente in das CAD-Part übertragen werden.

Parameter der Methode ist ein Zeiger auf das zu visualisierende Kooperationselement.

Der Name des Typs[6] gibt Aufschluß über die Dimensionalität des zu erstellenden CAD-Elements. Dementsprechend wird eine Methode des KInterface aufgerufen, in der die Geometrie erstellt wird, Parameter ist ein Zeiger auf das die Geometrie repräsentierende Entity aus dem Datenmodell:

- *CADPunktErzeugen* für Punkte
  Parameter ist ein *SdaiCartesian_point**
- *CADEbeneErzeugen* für Ebenen
  Parameter ist ein *SdaiPlane**

[6]SHAPE_POINT, SHAPE_CURVE, etc.

- *CADLinieErzeugen* für Kreise und Geraden
  Parameter ist ein Zeiger auf das Kooperationselement, weil in diesem Fall noch zwischen Gerade und Kreis unterschieden werden muß; daher wird mit

  ```
  entCooperation_element->
      associated_geometric_representation_()->EntityName()
  ```

  der Name des Geometrie-Entities (*line* oder *circle*) überprüft und die entsprechende Funktion aufgerufen (siehe folgende zwei Punkte)

- *CADGeradeErzeugen* für Kreise
  Parameter ist ein *SdaiLine**

- *CADKreisErzeugen* für Geraden
  Parameter ist ein *SdaiCircle**

Nachfolgend wird die Erzeugung der Geometrieelemente beschrieben.

4.5.1 Geometrieelement Punkt

Die drei Koordinaten des Punktes sind im Attribut *coordinates* des Entities *cartesian_point* abgelegt. Dieses Attribut ist eine List-Aggregation[7] von Gleitkommazahlen, auf die folgendermaßen zugegriffen wird:

```
double P1[3] = {0, 0, 0};
P1[0] = ((RealNode*) entCartesian_point->
    coordinates_()->GetHead())->value;
P1[1] = ((RealNode*) entCartesian_point->
    coordinates_()->GetHead()->NextNode())->value;
P1[2] = ((RealNode*) entCartesian_point->
    coordinates_()->GetHead()->NextNode()->NextNode())->value;
```

Mit der Funktion *CreatePoint* aus dem SolidWorks API kann ein Punkt als Element einer SolidWorks Skizze erzeugt werden; Parameter sind die drei Koordinaten des Punktes; mit der Funktion *InsertSketch* wird die Skizze als Element dem Part hinzugefügt:

```
hres = pModel->CreatePoint(P1[0], P1[1], P1[2], &retval);
hres = pModel->InsertSketch();
```

Das SolidWorks API bietet leider nicht die Möglichkeit, einen Referenzpunkt direkt als Feature dem Part hinzuzufügen, jedoch kann in SolidWorks auch auf einen Skizzenpunkt referenziert werden.

[7] siehe auch Kap.2.3.2

4.5.2 Geometrieelement Gerade

Die Geometriedaten der Gerade sind in den Attributen *pnt* (Startpunkt) und *direction* (Richtungsvektor mit Länge) des Entities *line* abgelegt. Zugriff erfolgt über eine Verkettung von Attributen; als Beispiel sei hier der Zugriff auf die erste Koordinate des Startpunktes gezeigt:

```
P1[0] = ((RealNode*) entLine->pnt_()->
    coordinates_()->GetHead())->value;
```

Das Entity *line* hat u.a. das Attribut *pnt* vom Typ *cartesian_point*, dieses wiederum hat das Attribut *coordinates* (siehe Kap.4.5.1). Die anderen Daten sind mit analog zu diesem Verfahren erreichbar.

Die im SolidWorks API vorhandene Funktion*InsertAxis* läßt keine Erzeugung der Referenzachse anhand von Koordinaten zu; ausschließlich selektierte Geometrieelemente sind Grundlage der Achsenlage. Daher wird, ähnlich wie beim Punkt, eine Gerade in einer Skizze erzeugt mit der Funktion *CreateLine*; Parameter sind Start- und Endpunkt der Geraden als Felder vom Typ *double*; anschließend wird die Skizze dem Part hinzugefügt:

```
double P1[3] = ... ;
double P2[3] = ... ;
hres = pModel->CreateLine(P1, P2);
hres = pModel->InsertSketch();
```

Die Umrechnung der Daten erfolgt nach folgendem Schema:

- Koordinaten des Startpunktes sind bekannt.
- Normierter Richtungsvektor und Länge der Geraden sind bekannt, daraus lassen sich die Koordinaten des Endpunktes berechnen:
  Richtungsvektor mit Länge multiplizieren, Vektoraddition zu den Koordinaten des Startpunktes ergibt den Endpunkt.

Auf die so erzeugte Gerade in der Skizze kann referenziert werden.

4.5.3 Geometrieelement Kreis

Die geometrische Beschreibung des Kreises liegt in den Attributen *radius* und *position* des Entities *circle*; letzteres wird vom Entity *conic* geerbt, hat den Typ *axis2_placement* und damit weitere Attribute, die der Beschreibung des Kreises dienen:

- *location* (geerbt von *placement*) ist der Mittelpunkt des Kreises.
- *axis* und *ref_direction* spannen eine Ebene auf, in der der Kreis liegt.

Der Zugriff folgt dem in Kap.4.5.2 erklärten Schema; beispielhaft gezeigt an der ersten Koordinate des Mittelpunktes des Kreises:

```
P1[0] = ((RealNode *) entCircle->position_()->location_()->
    coordinates_()->GetHead())->value;
```

Das Geometrieelement wird mit der Funktion *CreateCircle* erzeugt; diese erwartet als Parameter die Koordinaten des Mittelpunktes und die Koordinaten eines Punktes auf der Kreisbahn:

```
hres = pModel->CreateCircle(P1[0], P1[1], P1[2],
                            P2[0], P2[1], P2[2],
                            &retval);
```

Die Umrechnung der Daten erfolgt nach folgendem Schema:

- Koordinaten des Mittelpunktes sind bekannt.
- Normierte Richtungsvektoren, welche die Kreisebene aufspannen und der Radius sind bekannt; daraus lassen sich die Koordinaten eines Kreispunktes berechnen:
  einen der Richtungsvektoren mit Radius multiplizieren, Vektoraddition zu den Koordinaten des Mittelpunktes ergibt Punkt auf der Kreisbahn.

4.5.4 Geometrieelement Ebene

Die Geometriedaten der Ebene werden durch das Attribut *position* des Entities *plane* repräsentiert; das Attribut wird vom Entity *elementary_surface* geerbt. Das Attribut hat den Typ *axis2_placement* und beschreibt die Ebene folgendermaßen:

- Das Attribut *location* des Entities *axis2_placement* (geerbt vom Entity *placement*) beschreibt einen Punkt, der auf der Ebene liegt.
- Die Attribute *axis* und *ref_direction*) von *axis2_placement* beschreiben zwei normierte Richtungsvektoren die eine Ebene aufspannen, in diesem Fall die zu erzeugende Referenzebene.

Zugriff auf die Koordinaten erhält man nach dem in den vorangestellten Abschnitten erläuterten Schema.

Die Funktion *ICreatePlaneFixed* aus dem API erzeugt eine echte Referenzebene; Parameter sind drei Punkte, die diese Ebene aufspannen. Nach Erzeugung wird der Ebene (als Feature im Featurebaum des SW Part) der Name des Kooperationselementes zugewiesen:

```
double P1[3] = ... ;
double P2[3] = ... ;
double P3[3] = ... ;
hres = pModel->ICreatePlaneFixed(P1, P2, P3, TRUE);
...
_bstr_t Name = ... ;
hres = pFeature->put_Name(Name);
```

Die Umrechnung der Daten erfolgt nach folgendem Schema:

- Koordinaten des ersten Punktes sind bekannt: Attribut *location*
- Normierte Richtungsvektoren, welche die Ebene aufspannen sind bekannt; daraus lassen sich die Koordinaten der beiden anderen Punkte berechnen: jeweils einen der Richtungsvektoren vektoriell zu den Koordinaten des ersten Punktes addieren:
  ergibt jeweils einen weiteren Punkt.

Die Ebene erscheint mit dem Namen des Kooperationselementes im FeatureManager des SolidWorks Parts, und auf sie kann referenziert werden. Somit ist die Ebene als einziges Kooperationselement vollständig in SolidWorks präsentierbar, da den anderen Kooperationselementen nicht unmittelbar ein Name zugewiesen werden kann und sie nicht direkt als Feature erscheinen.

4.6 Benutzungsschnittstelle

Die Benutzungsschnittstelle wird zum einen Teil über die in Kapitel 3.3.2 beschriebene Menüstruktur verwirklicht, zum anderen über ein Dialogkonzept, welches die Eingabe der nicht-geometrischen Daten durch den Benutzer über Dialogboxen erlaubt.

Zusatzmenü

Das zusätzliche Menü wird mit den in Kapitel 2.2.3 erläuterten Methoden hinzugefügt und mit den entsprechenden Funktionen verknüpft. Das eingefügte Menü ist in Abbildung 4.1 zu sehen.

Die Alternative zum Menüpunkt *Kooperationselement erzeugen* aus dem zusätzlichen Pull-Down-Menü ist der neue Menüpunkt im Eigenschaftsmenü der Geometrieelemente; beispielhaft ist in Abb.4.2 eine darüber selektierte Kante gezeigt.

Dialogboxen

Die Dialogboxen zur Dateneingabe werden mit der in Visual C++ üblichen Methodik implementiert. Die entsprechende Box wird mit der grafischen Oberfläche

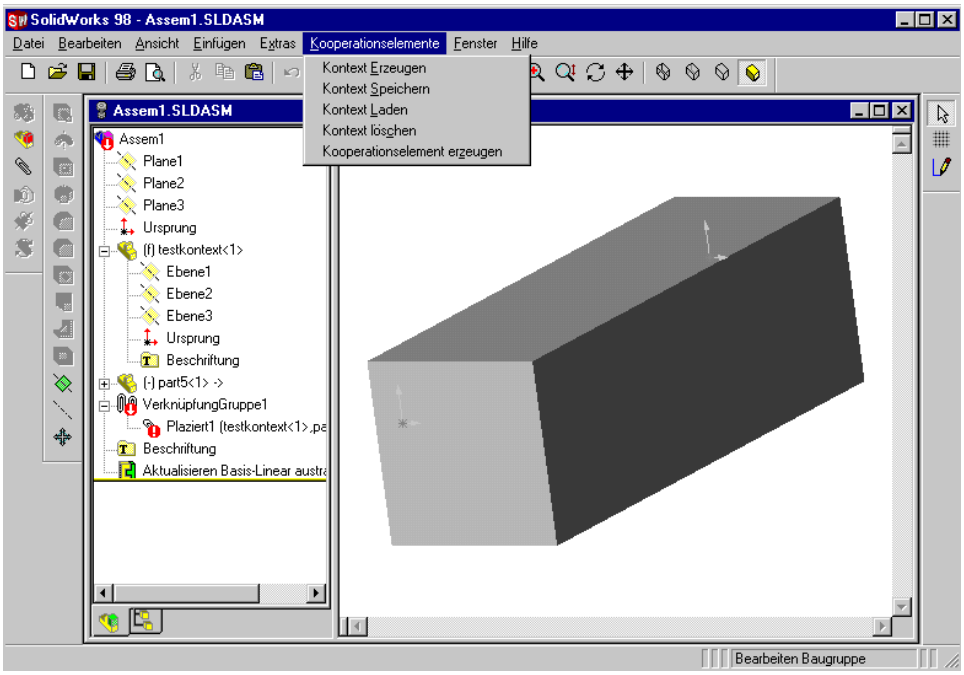

Abbildung 4.1: Menü Kooperationselemente

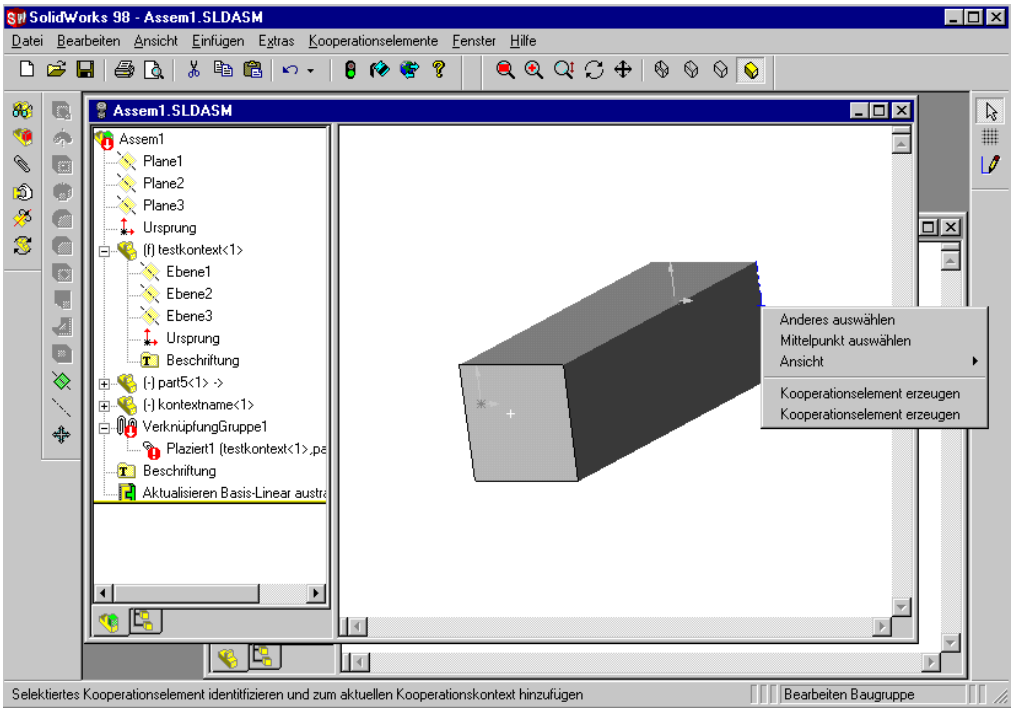

Abbildung 4.2: Kooperationselement über Eigenschaftenmenü erzeugen

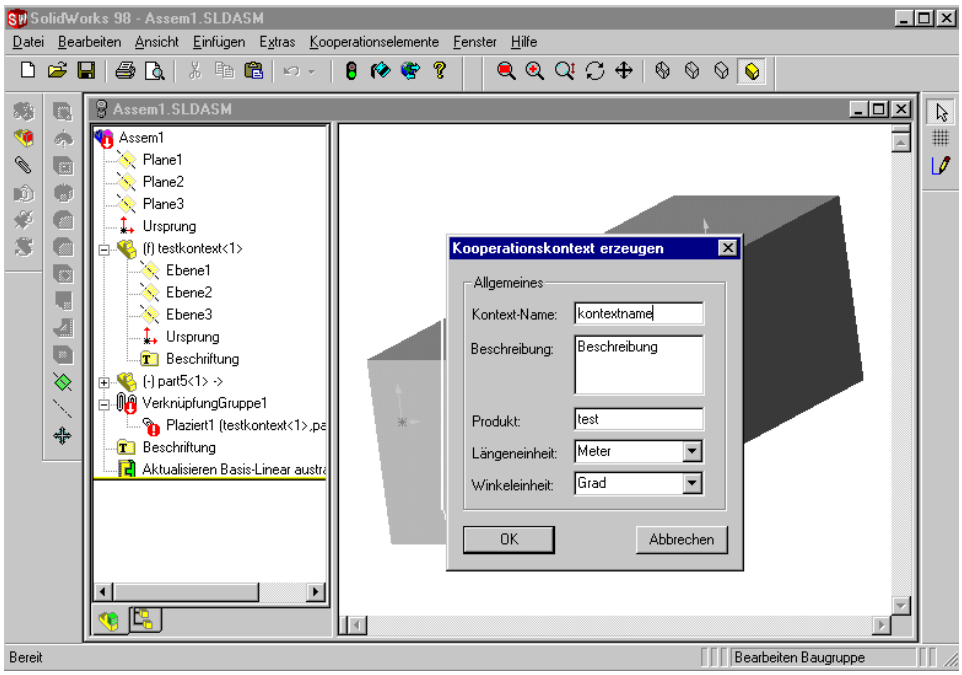

Abbildung 4.3: Kontext erzeugen und Attribute eingeben

von Visual C++ erstellt (siehe dazu Abb.2.8) und automatisch mit einer Dialogklasse verknüpft. Im Programm selbst muß diese Klasse instanziiert werden und der Dialog wird über die Methode *DoModal* aus dieser Klasse aufgerufen. Verknüpft mit den Feldern der Dialogbox sind Datenelemente der Dialogklasse, über die man Zugriff auf die eingegebenen Daten hat.

Folgende Dialoge sind implementiert:

Abb.4.3: Dateneingabe für neuen Kooperationskontext

Abb.4.4: Dateneingabe für neues Kooperationselement

Abb.4.5: Hinzufügen eines neuen Mitarbeiters am aktuellen Kontext

Abb.4.6: Hinzufügen einer neuen Part-Beziehung zum Kooperationselement

Abb.4.7: Dateinamen eingeben zum Speichern oder Laden des Kontext

Visualisierung

Zur Benutzungsschnittstelle gehört auch die Visualisierung der Kooperationselemente. Beim Erzeugen eines Kooperationskontextes wird in die aktuelle Baugruppe (Abb.4.8) eine Teile-Datei eingefügt (Abb.4.9 und Abb.4.10), in der die Kooperationselemente als Referenzgeometrie dargestellt werden. Die Elemente des Kontext werden auch in der Baugruppe dargestellt (Abb.4.11).

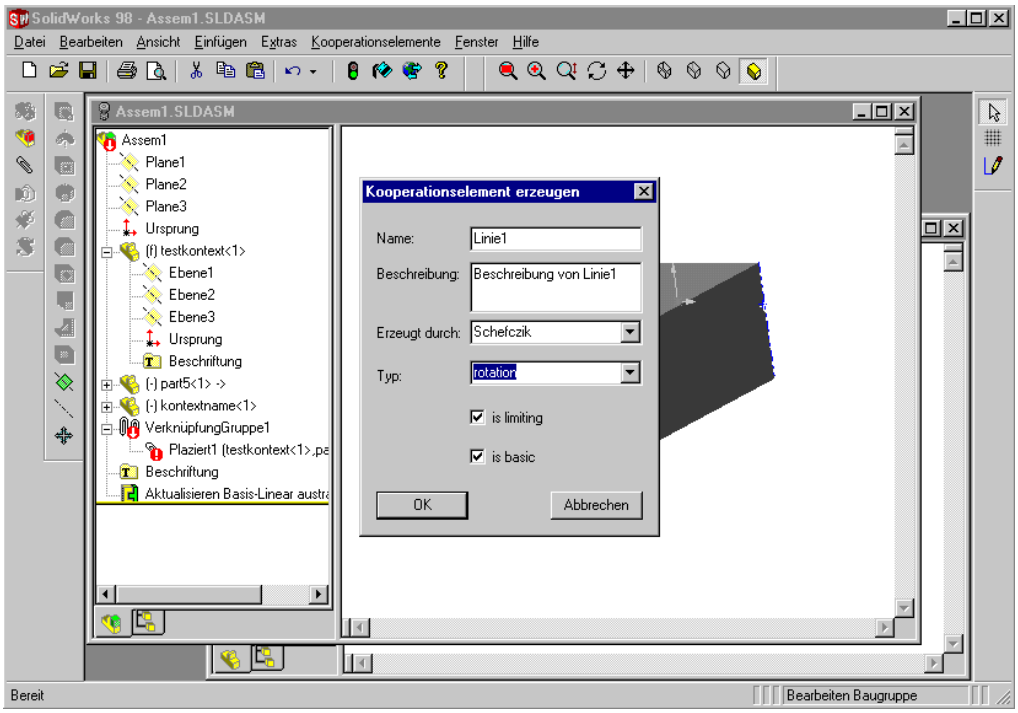

Abbildung 4.4: Kooperationselement erzeugen und Attribute eingeben

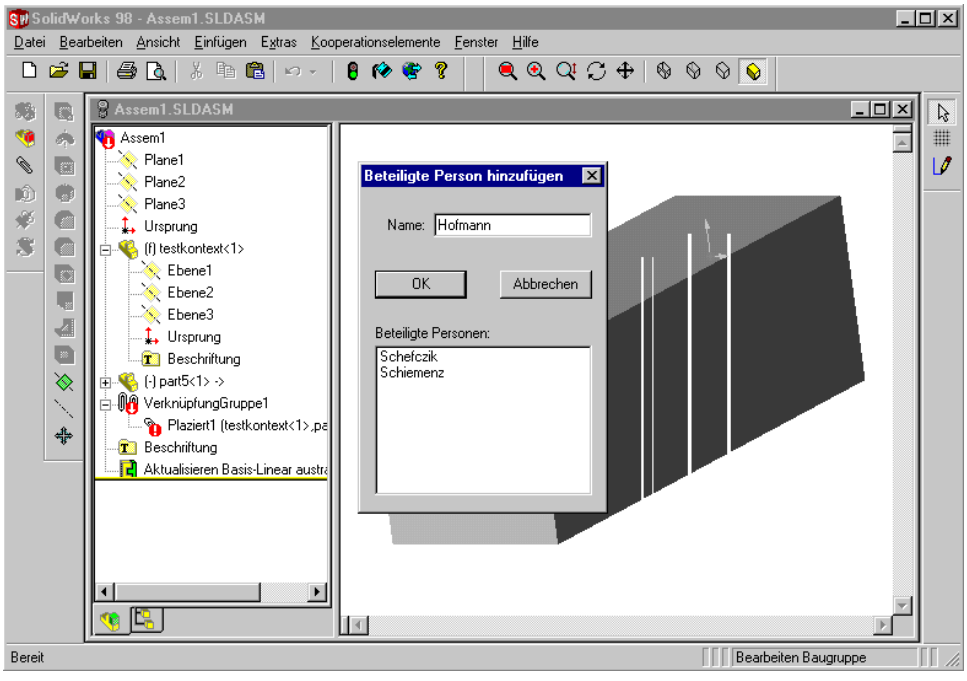

Abbildung 4.5: *colloborating_person* zum Kontext hinzufügen

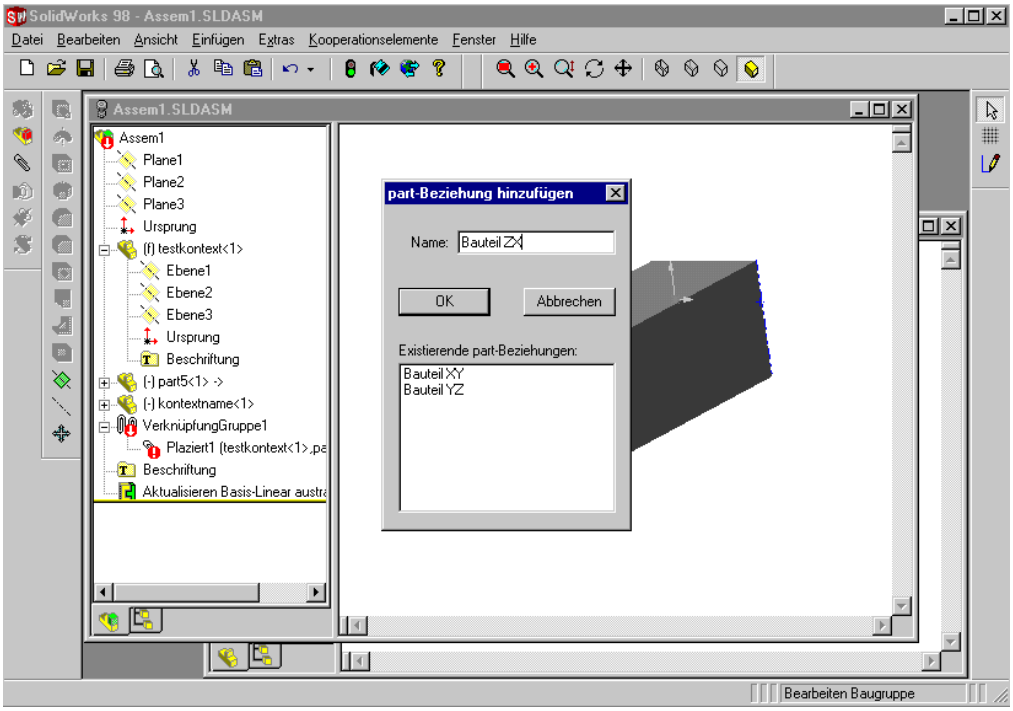

Abbildung 4.6: Attribut *used_in* des Kooperationselementes hinzufügen

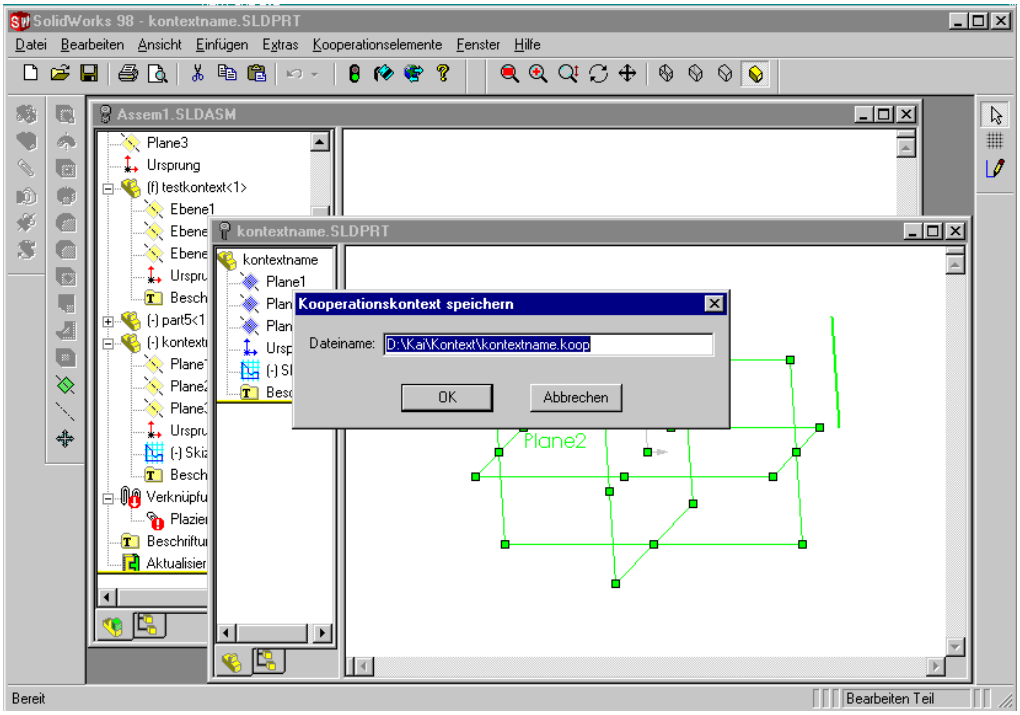

Abbildung 4.7: Dateiname eingeben und Kontext speichern

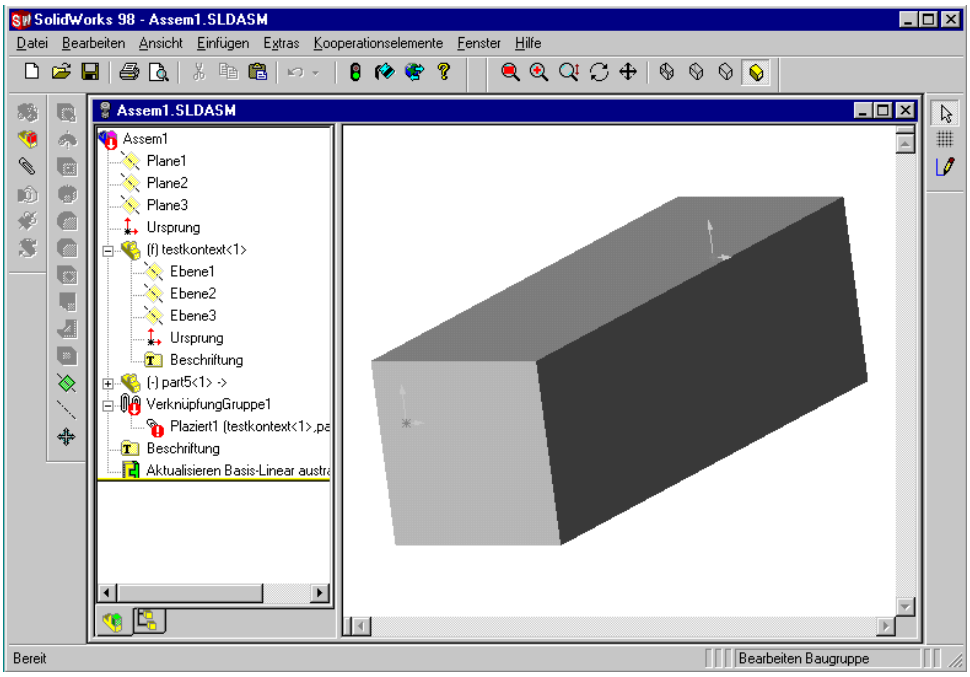

Abbildung 4.8: Baugruppe

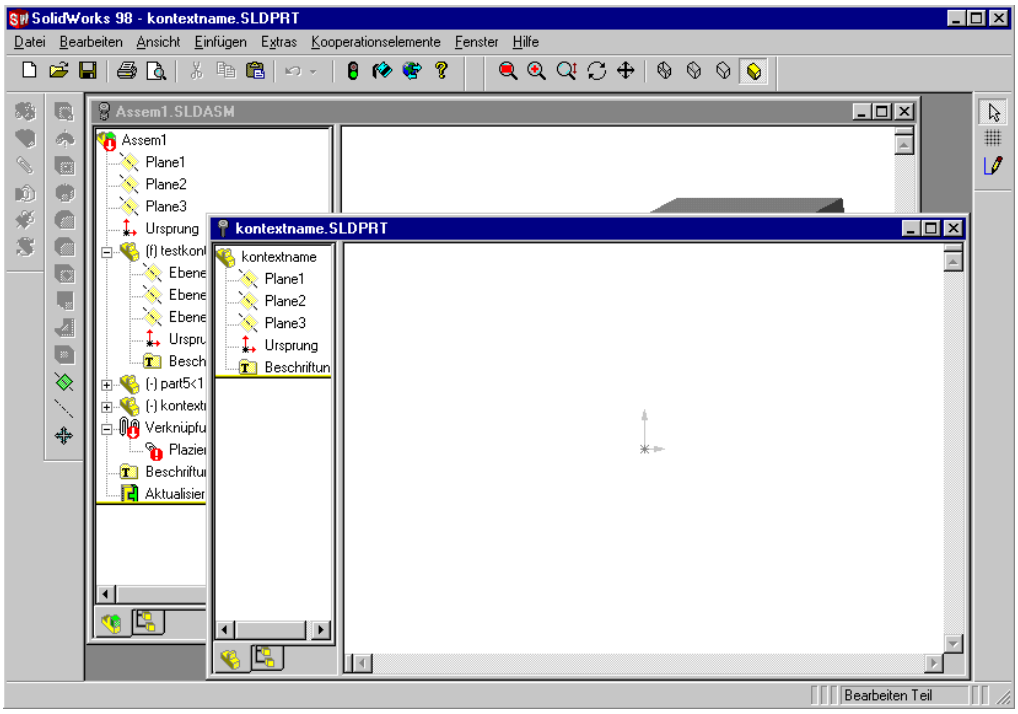

Abbildung 4.9: Präsentation des Kontext ohne Elemente im Part

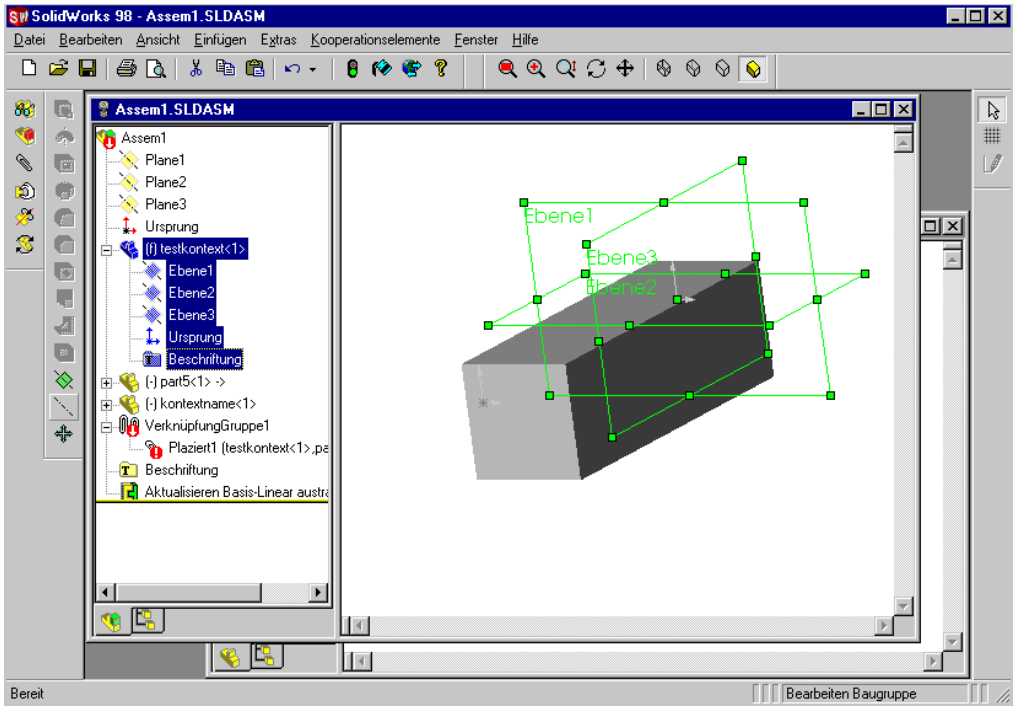

Abbildung 4.10: SW-Part des Kontext in Baugruppe eingefügt

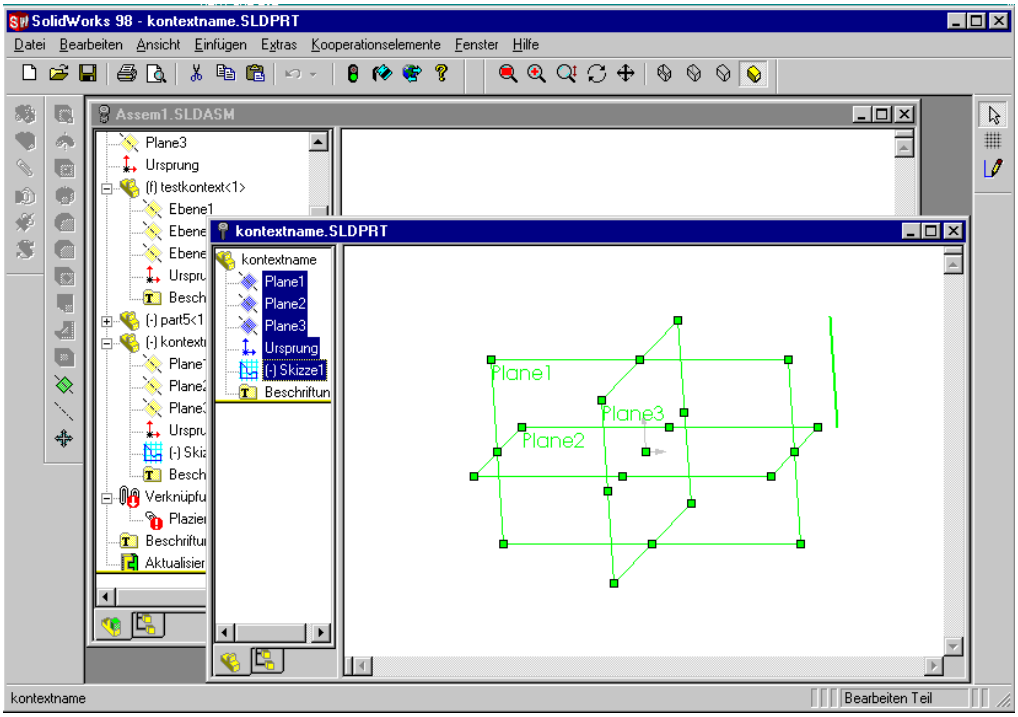

Abbildung 4.11: Baugruppe mit Präsentation des Kontext

Kapitel 5

Zusammenfassung und Ausblick

Abschließend soll in diesem Kapitel eine Bewertung vorgenommen werden; Probleme werden dargestellt, Erkenntnisse zusammengefaßt und ein Ausblick auf mögliche Weiterentwicklung gegeben.

5.1 Probleme bei der Implementierung

Probleme im API

Eines der Problemfelder bei der Implementierung war die teilweise unzureichende Dokumentation der Programmierschnittstelle des SolidWorks API. Einige der Funktionen waren nicht mit allen Parametern beschrieben, was beim Compile-Vorgang zu Fehlermeldungen führte. Diese Fehler konnten zwar durch Untersuchung des Quellcode der API-Klassen behoben werden, weil die Parameter dort in den Deklarationen der Methoden zu finden sind; allerdings bedeutete dies natürlich einen Zeitverlust, da zum einen aufgrund der Größe des Projektes die Kompilierung einige Zeit in Anspruch nahm, und zum anderen die Untersuchung des Quellcodes zusätzlichen Aufwand bedeutete.

Ein weiteres Problem ergab sich bei der Präsentation der Kooperationselemente. Zur Erstellung einer Referenzebene existiert im API eine ideale Funktion; jedoch war die Präsentation der anderen Elemente nur eingeschränkt möglich. Punkte und Kreise können nur in Skizzen erzeugt werden, als Referenzgeometrie im Raum sind sie in der SolidWorks-Datenstruktur nicht vorgesehen. Das Erstellen einer Referenzachse ist prinzipiell möglich, allerdings nicht aufgrund von Geometriedaten, sondern nur anhand bestehender Geometrie, z.B. eine Kante, zwei sich schneidene Ebenen, zwei Punkte, etc. Daher werden solche Linien auch in Skizzen präsentiert. Daraus ergibt sich das Problem, daß diesen Elementen in der Präsentation keine Namen direkt zugeordnet werden können.

Entwicklungsumgebung

Das Betriebssystem Windows NT von der Firma Microsoft zeigte oft Stabilitätsprobleme, Programmabstürze ohne erkennbaren Grund störten den Entwicklungsprozeß.

Jedoch ergaben sich durch die auf die Windows-Oberfläche abgestimmte Entwicklungsumgebung Microsoft Developer Studio erhebliche Vorteile in der Implementierung der Benutzungsschnittstelle. Zusätzliche Erleichterung brachte der grafische Klassenbrowser des Developer Studio, die umfassenden Suchmöglichkeiten desselben und die umfangreiche integrierte Online-Dokumentation der Programmiersprache Visual C++.

5.2 Kritische Anmerkungen und Ausblick

Zu bearbeitende Forderungen

Für eine über den Prototypenstatus hinausgehende Implementierung sind vor allem noch folgende Forderungen zu bearbeiten:

Redundante Datenhaltung: Die Kooperationselemente werden in einem neutralen Dateiformat repräsentiert, welches an das STEP-Format aus der Norm ISO 10303 angelehnt ist. Da die Präsentation der Kooperationselemente im CAD-System erfolgt, und auch die Referenzierung der Bauteilgeometrie auf die Kooperationselemente die Existenz einer CAD-Datei voraussetzt, sind die Geometriedaten der Kooperationselemente redundant, d.h. in zwei Dateien gleichzeitig abgelegt, was bei Änderung in einer der Dateien problematisch werden kann, wenn die Änderung nicht sofort in die andere Datei übertragen wird. Daraus könnten besonders bei der Referenzierung Probleme entstehen.

Eine mögliche Lösung des Problems könnte sich dadurch ergeben, daß das CAD-System eine Referenzierung auf die Repräsentation in der STEP-Datei erlaubt, d.h. die Präsentation im CAD wäre ausschließlich eine Visualisierung der Kooperationselemente. Zusätzlich sollte eine in der Präsentation vorgenommene Änderung automatisch direkt in der STEP-Datei übernommen werden.

Visualisierung: Für die Visualisierung fordert Hofmann eine eindeutige farbliche hervorhebung der Kooperationselemente[Hofm-99]. Dies erleichtert die Übersicht, besonders für die Darstellung im Baugruppenmodus, wenn noch viele andere Geometrieelemente präsentiert werden.

Fehler- und Ausnahmebehandlung: Im vorliegenden Prototypen werden nur einige wenige Fehler abgefangen, z.B. wenn vor dem Erzeugen eines Kooperationselementes keine Geometrie selektiert wurde. Die Fehlerbehandlung

hatte jedoch nur geringe Priorität, da Hauptziel die Validierung des Konzeptes mit dem vorliegenden Datenmodell war.

Mehrere Kooperationskontexte: Zur Zeit ist nur die Existenz eines Kooperationskontextes implementiert. Da die Zusammenarbeit mit verschiedenen Teams in der Konstruktion und Entwicklung heutzutage weit verbreitet ist, sollte auch die Verwaltung mehrerer Kontexte möglich sein.

Produktdatenmanagement: Die Integration in ein bestehendes Produktdatenmanagement erfordert eine Untersuchung solcher bestehender Systeme und ein entsprechendes Konzept, vor allem im Hinblick auf Freigabewesen, Datensicherheit und unternehmensübergreifende Zugriffsmöglichkeiten bei Zusammenarbeit mit externen Zulieferern.

Chancen und Vorteile des Konzeptes

Programmstruktur: Das in Kapitel 3 vorgestellte Konzept bietet durch die Trennung von Daten, Methoden und Interface die Möglichkeit, die Applikation schnell und einfach an verschiedene CAD-System anzupassen. Weitergehende Kapselung und Modularisierung kann den Aufwand der Anpassung zusätzlich minimieren. Das Interface-Konzept hat sich insofern bewährt, daß die Implementierung der Kooperationskontext- und Kooperationselemente-Klassen komplett unabhängig von SolidWorks, bzw. dem SolidWorks API durchgeführt werden konnte, und daß gezeigt werden konnte, daß jede Art von Nachricht, die zwischen Methoden und API ausgetauscht wird, vom Interface angepaßt und weitergeleitet werden kann.

Für weitergehende Bearbeitung des Konzeptes wird empfohlen, das Inteface zu modularisieren; z.B. jeweils eigene Module zu entwerfen für

- Erfassen der CAD-Daten,
- Umwandlung CAD-Daten in STEP-Format oder umgekehrt,
- Präsentation der Kooperationselemente

oder für eine andere geeignete Struktur.

STEP Class Library: Die STEP Class Library bietet große Vorteile bei der Verwendung von Datenmodellen im EXPRESS-G- oder EXPRESS-Format. Die aus dem Datenmodell abgeleiteten C++-Klassen und die SCL-Klassen konnten ohne Veränderung in das Projekt übernommen werden und zeichneten sich durch hohe Stabilität und Funktionalität aus. Der Zugriff auf die unterschiedlichen Entities gestaltete sich nachvollziehbar und an die Erfordernisse der STEP-Norm angepaßt. Eine weitere Verwendung der SCL kann in jedem Fall empfohlen werden.

Infomodel: Die Klasse *Infomodel* erleichterte die Verwendung der SCL-Klassen. Das Erzeugen oder Löschen der Entities, Zugriff auf ein einzelnes Entity oder auf Aggregationen von Entities, Lesen oder Schreiben von Kooperationselementedateien im STEP-Format werden durch die Klasse *Infomodel* unterstützt und vereinfacht.

Es wird jedoch empfohlen, der Klasse einige Methoden hinzuzufügen, welche sie um die in Kapitel 2.3.3 beschriebene Funktionalität erweitern. dadurch kann die Verwendung der STEP Class Library weiter vereinfacht werden und der Programmieraufwand für die Methodenbank wird verringert, da die Übersichtlichkeit erhöht wird.

Anhang A

Quellcode und Beispieldateien

Auf den nachfolgenden Seiten werden folgende Dateien dargestellt:

- Quellcode der Klasse KEMethoden
- Quellcode der Klasse KKMethoden
- Quellcode der Klasse KInterface
- Quellcode der Klasse KKErzeugenDialog
- Quellcode der Klasse KEErzeugenDialog
- Quellcode der Klasse PartHinzufuegenDialog
- Beispiel einer Kooperationselemente-Datei