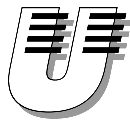


Fachbereich 4: Informatik

# Szeneneditor für ein Echtzeitanimationssystem

*und andere XML konfigurierte und erweiterbare Systeme*

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

**Dimitrios N. Papoutsis**

Betreuer: Dipl.-Inform Dominik Rau
realtime visions (Interaktive Multimedia-Erlebniswelten)

Koblenz, im August 2006

# Erklärung

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Ort, Datum) (Unterschrift)

# Inhaltsverzeichnis

Abbildungs- und Tabellenverzeichnis . . . . . . . . . . . . . . . . . . . . . . . . vi

**1 Einleitendes** **1**

1.1 Intuitive Einführung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Die Ziele . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.3 Die Gliederung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Grundlagen** **3**

2.1 Komplexität durch Abstraktion . . . . . . . . . . . . . . . . . . . . . . . . 3

2.1.1 Herrsche und teile ... teile und herrsche . . . . . . . . . . . . . . . 4

2.2 Das DR - Animations System (DRAS ) . . . . . . . . . . . . . . . . . . . 5

2.2.1 Beschaffenheit der Konfigurationsdatei/en . . . . . . . . . . . . . . 6

2.3 Begriffsklärung: Ordnung, Struktur und Hierarchie . . . . . . . . . . . . 7

2.3.1 Ein allgemeines Objektmodell . . . . . . . . . . . . . . . . . . . . 7

2.3.2 Definitionen und Ableitungen . . . . . . . . . . . . . . . . . . . . 8

2.4 Allgemeine Beschreibung der Anforderungen . . . . . . . . . . . . . . . . 8

2.5 Softwareergonomische Kriterien . . . . . . . . . . . . . . . . . . . . . . . 9

2.5.1 Softwareergonomische Anteile in dieser Arbeit . . . . . . . . . . . 10

2.6 Aspekte der Softwaretechnik . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.6.1 Phasen und Aufgaben . . . . . . . . . . . . . . . . . . . . . . . . . 12

**3 Analyse und Eingrenzung der Anforderungen** **15**

3.1 Idee eines Szeneneditors in dreieinhalb Variationen . . . . . . . . . . . . 15

3.1.1 Ideen 1 und 2: Online vs. Offline . . . . . . . . . . . . . . . . . . . 15

3.1.2 Idee 3: CMS vs. Konfigurationseditor . . . . . . . . . . . . . . . . 16

3.1.3 Idee $3\frac{1}{2}$: CGI basierte Lösung . . . . . . . . . . . . . . . . . . . . 17

3.2 Definition der Anforderungen . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2.1 DRAS: Bindung zur Kommunikation . . . . . . . . . . . . . . . . 17
3.2.2 Verfeinerung der Zielsetzung . . . . . . . . . . . . . . . . . . . . 19
3.3 Zwei konkurrierende Lösungsansätze . . . . . . . . . . . . . . . . . . . . 21
3.3.1 Plugins vs. XML-Beschreibungssprache . . . . . . . . . . . . . . 21

**4 Abstraktion von Information** **23**
4.1 Ordnende Strukturen . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.1.1 Bildung einer Ordnungshierarchie . . . . . . . . . . . . . . . . . . 25
4.1.2 Ordnende Graphen und Objektorientierung . . . . . . . . . . . . 25
4.2 **XML** e**X**tensible **M**arkup **L**anguage . . . . . . . . . . . . . . . . . . 27
4.2.1 Verarbeitende APIs . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.2 Wesentliche Bausteine von XML . . . . . . . . . . . . . . . . . . 29
4.2.3 XML: Beispiel . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2.4 Ist XML bedeutungslos ? . . . . . . . . . . . . . . . . . . . . . . . 31
4.2.5 Wohlgeformtheit und Gültigkeit . . . . . . . . . . . . . . . . . . . 31
4.2.6 Ist XML mono- oder polyhierarchisch ? . . . . . . . . . . . . . . . 31
4.2.7 Namensräume in XML . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 **DTD** **D**ocument **T**ype **D**efinition . . . . . . . . . . . . . . . . . . . . 32
4.3.1 Bewertung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.4 **XSD** **X**ML **S**chema **D**efinition . . . . . . . . . . . . . . . . . . . . . 33
4.4.1 Bewertung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.5 **SLIDL** **SL**ang**I**tem **D**efinition **L**anguage . . . . . . . . . . . . . . . . 34

**5 Analyse und Entwurf** **37**
5.1 Abstraktion von Information . . . . . . . . . . . . . . . . . . . . . . . . 37
5.2 Anforderungen an eine Sprach–Definition . . . . . . . . . . . . . . . . . 38
5.2.1 Objektorientierte Perspektive . . . . . . . . . . . . . . . . . . . . 38
5.2.2 Container und Elementarobjekte . . . . . . . . . . . . . . . . . . 39
5.2.3 Container-Typen, elementare Datentypen und Basistypen . . . . . 39
5.2.4 Abgeleitete elementare Typen . . . . . . . . . . . . . . . . . . . . 40
5.2.5 Klassen und deren Ableitungen . . . . . . . . . . . . . . . . . . . 41
5.3 Abstraktion von Konnektion . . . . . . . . . . . . . . . . . . . . . . . . 42
5.3.1 Konnektionsobjekt . . . . . . . . . . . . . . . . . . . . . . . . . . 42

5.3.2 Konnektionsoperanden I . . . . . . . . . . 42
5.3.3 Konnektionsoperanden II . . . . . . . . . . 43
5.3.4 Konnektionsoperanden III . . . . . . . . . . 43
5.3.5 Konnektion . . . . . . . . . . 44
5.3.6 Inverse Konnektion . . . . . . . . . . 45
5.4 Über persistente Objekte . . . . . . . . . . 45
5.4.1 C++ und Xerces . . . . . . . . . . 46
5.4.2 XML Persistenz . . . . . . . . . . 46
5.5 Die Softwarearchitektur . . . . . . . . . . 46
5.5.1 Verbund von Schichten . . . . . . . . . . 47
5.5.2 Die Persistenz Schicht . . . . . . . . . . 48
5.5.3 Die Deklarations Schicht . . . . . . . . . . 55
5.5.4 Die Instanz Schicht . . . . . . . . . . 55

**6 Werkzeuge der Programmierung 57**
6.1 Der C/C++–Präprozessor (Cpp) . . . . . . . . . . 58
6.1.1 Traditionelle Einsatzgebiete des Präprozessors . . . . . . . . . . 58
6.1.2 Das traditionelle Kompiliermodell . . . . . . . . . . 59
6.1.3 Cpp Templates und C++ Templates . . . . . . . . . . 59
6.1.4 Der Gültigkeitsbereich von *define* Ausdrücken . . . . . . . . . . 60
6.1.5 Objektorientierte Präprozessoren . . . . . . . . . . 61
6.1.6 Zusammenfassung und Folgerungen . . . . . . . . . . 62
6.2 Mehrfach Ableitungen in C++ . . . . . . . . . . 63
6.2.1 Schizophrene Objekte . . . . . . . . . . 63
6.2.2 Virtuelle Ableitungen . . . . . . . . . . 64
6.2.3 Kreuz–Delegation bei virtuellen Mehrfachableitungen . . . . . . . . . . 65
6.2.4 Zusammenfassung und Folgerungen . . . . . . . . . . 66
6.3 Abschließendes . . . . . . . . . . 67

**7 Entwurf und Implementierung 69**
7.1 **SLIDL** als Beschreibungssprache . . . . . . . . . . 69
7.1.1 Sprachdefiniton . . . . . . . . . . 69
7.1.2 Class: Beschreibung untergeordneter Sprachelemente . . . . . . . . . . 71
7.1.3 Typisierungs-System . . . . . . . . . . 73

7.1.4 SLIDLDocument: Repräsentation der Zielsprache . . . . . . . . . 75
7.1.5 SLIDLScript: Auflösung von Laufzeitabhängigkeiten . . . . . . . 75
7.1.6 ConnectorInterface, ConnectorClass und Rule . . . . . . . . . . . 76
7.2 Objektorientiertes Design . . . . . . . . . . . . . . . . . . . . . . . . . 76
7.2.1 Attributklassen . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
7.2.2 Bündelnde Ordnungsklassen . . . . . . . . . . . . . . . . . . . . . 78
7.2.3 Hypervirtuelle Ordnungsklassen . . . . . . . . . . . . . . . . . . 78
7.2.4 Verwendung von Namensräumen . . . . . . . . . . . . . . . . . . 79
7.2.5 Initialisierung von Sprachelementen . . . . . . . . . . . . . . . . 79
7.3 **SLIDL** Deklarations Verwaltung . . . . . . . . . . . . . . . . . . . . 81
7.3.1 Auflösung von Typabhängigkeiten . . . . . . . . . . . . . . . . . 81
7.3.2 Netzwerkfähigkeit . . . . . . . . . . . . . . . . . . . . . . . . . . 83
7.4 **SLIDL** Rückbesinnung **GUI** . . . . . . . . . . . . . . . . . . . . . . . 83
7.4.1 Typabhängige Widgets auf XML Basis . . . . . . . . . . . . . . 83
7.5 **SLIDL** Instanz Verwaltung . . . . . . . . . . . . . . . . . . . . . . . 84
7.6 ComplexObject und SimpleObject . . . . . . . . . . . . . . . . . . . . 86
7.7 Abhängigkeiten von Daten zur Laufzeit . . . . . . . . . . . . . . . . . 87
7.7.1 SLIDLScript: Beispiel . . . . . . . . . . . . . . . . . . . . . . . . 87
7.7.2 SLIDLScript: Syntax . . . . . . . . . . . . . . . . . . . . . . . . . 87
7.7.3 Auswertung von Kommandos . . . . . . . . . . . . . . . . . . . . 88
7.8 Laden und Speichern von Instanzdokumenten . . . . . . . . . . . . . . 89

**8 Ergebnis** **91**
8.1 Entwicklung theoretischer Grundlagen . . . . . . . . . . . . . . . . . . 92
8.2 Vollständig abgeschlossene Systemkomponenten . . . . . . . . . . . . . 92
8.3 Konzeptionell abgeschlossene Systemkomponenten . . . . . . . . . . . . 92
Abkürzungsverzeichnis . . . . . . . . . . . . . . . . . . . . . . . . . . . . I
Literaturverzeichnis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . III

# Abbildungsverzeichnis


2.1 Schichtenmodell der Abstraktion . . . . . . . . . . . . . . . . . . . . 4
2.2 Kriterien der Softwareergonomie . . . . . . . . . . . . . . . . . . . . 10
2.3 Unvollständige Liste von Prinzipien der Softwaretechnik und damit assoziierte Methoden, teilweise aus (Bal00, S. 37) . . . . . . . . . . 11

3.1 DRAS: Bindung als Datenaustausch . . . . . . . . . . . . . . . . . . 18
3.2 Mindmap nötiger Bausteine . . . . . . . . . . . . . . . . . . . . . . 21

4.1 Allgemeine Darstellung hierarchischer Ordnungsstrukturen . . . . . 24
4.2 Vereinfachte Darstellung eines XML–Dokumentes . . . . . . . . . . 30

5.1 Die grobe Architektur . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.2 UML Diagramm: **XMLStr** und **XMLSysEnvironment** . . . . . . . . 50
5.3 UML Diagramm: **XMLReadWriteable** . . . . . . . . . . . . . . . . . 51
5.4 UML Diagramm: **XMLTools** . . . . . . . . . . . . . . . . . . . . . . 52
5.5 C++ Interface: **XMLRegistry** . . . . . . . . . . . . . . . . . . . . . 53
5.6 Die Architektur . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

6.1 Schizophrenes Objekt . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.2 Virtuelle Mehrfach–Ableitungen . . . . . . . . . . . . . . . . . . . . 64
6.3 Kreuzdelegation bei virtueller Mehrfachableitung . . . . . . . . . . 66

7.1 Die SLIDL Sprachdefinition . . . . . . . . . . . . . . . . . . . . . . 70
7.2 SLIDL: Abgeleitete Basistypen . . . . . . . . . . . . . . . . . . . . 74
7.3 Das SLIDLDocument . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.4 SLIDL-API: Ausschnitt aus der Klassenhierarchie . . . . . . . . . . 77
7.5 SLIDL-API: Die Attributklassen . . . . . . . . . . . . . . . . . . . . 78

7.6 SLIDL: UML-Diagramm **SLIDL::Environment::Env** . . . . . . . . . . 80
7.7 Auflösen von deklarativen Abhängigkeiten . . . . . . . . . . . . . . . . 82
7.8 SLIDL: Die Laufzeit-Objekte (Pseudo-UML) . . . . . . . . . . . . . . . 85
7.9 SLIDL: UML-Digramme: **BaseObject**, **ComplexObject** und **SimpleObject** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7.10 SLIDL-API: Klassenhierarchie . . . . . . . . . . . . . . . . . . . . . . . 90

8.1 Implementierungsstand der Architektur . . . . . . . . . . . . . . . . . . 91

# Kapitel 1

# Einleitendes

## 1.1 Intuitive Einführung

Das in der Diplomarbeit von Dominik Rau entwickelte Animationssystem (Rau05), im folgenden abgekürzt durch **DRAS** (**D**ominik **R**aus **A**nimations **S**ystem) verwendet eine XML-basierte Sprache zur Beschreibung interaktiver Multimedia-Szenarien. Das DRAS ist ein weitgehend konfigurierbares, modulares Softwaresystem. Beruhend auf einer Plugin-Architektur bleibt es, innerhalb bestimmter Grenzen, beliebig erweiterbar. Daran unmittelbar gekoppelt ist die Erweiterbarkeit der Konfiguration selbst.

## 1.2 Die Ziele

Mit dem stetig wachsenden Sprachvolumen ist der Aufwand der Konfiguration *von Hand*, ein zunehmend kompliziertes und arbeitsintensives Unterfangen. Hinblicklich der Lösung dieses Problems, definiert sich das Hauptziel der vorliegenden Arbeit: Die Entwicklung einer graphischen Benutzerschnittstelle (oder eines *Szeneneditors*) die das Bearbeiten, das Laden und das Speichern der Szenarien ermöglicht.
Im Rahmen des Hauptziels sollen Möglichkeiten gefunden werden, den Datenaustausch zwischen Optionen und Zustandswerten, sowie Optionen und Ereignissen angemessen zu visualisieren. Zusätzlich sollen geeignete Wege gefunden werden, auch komplexe Szenen und Abläufe übersichtlich darzustellen.

## 1.3 Die Gliederung

In Kapitel 2 werden Grundlagen gelegt. Grundlagen für verwendete Denkmodelle, Methoden und Vorgehensweisen der folgenden Kapitel. Kapitel 3 ist der Aufspaltung des Hauptziels in Teilziele gewidmet. Hier werden die Anforderungen an eine GUI aufgeschlüsselt und explizit spezifiziert. Es folgen zusätzliche Überlegungen hinsichtlich der Repräsentation der Semantik einer Szenedatei. Schlussfolgernd werden grob notwendige Bausteine des zu erstellenden System abgeleitet.

Kapitel 4 vollzieht grundsätzliche Überlegungen zur *Ordnung* und *Struktur* von Daten und dem Konzept der *Objektorientierung*. Darauf gründend erfolgt die Einführung in die Grundlagen von XML. Die Phasen der *Analyse*, des *Entwurfs* und der *Implementierung* werden in den Kapiteln 5 bis 7 beschrieben. Innerhalb von Kapitel 6 werden zwei Werkzeuge der Programmierung vorgestellt und hinblicklich der Vorteile und Nachteile ihres Einsatzes untersucht.

Den Ausklang dieser Arbeit bildet das Kapitel 8. Dort erfolgt die Erörterung der Ergebnisse bezüglich der Zielsetzung. Ferner wird dort ein Ausblick auf resultierende Möglichkeiten im Rahmen der erzielten Ergebnisse vollzogen.

# Kapitel 2

# Grundlagen

## 2.1 Komplexität durch Abstraktion

Die Betrachtung von Software, am Beispiel der Computergraphik angedeutet, kann der Komplexität ihrer strukturellen Beschaffenheit nach zumindest grob innerhalb eines Schichtenmodells erfolgen. Am Beispiel des oben benannten Animationssystems sollen *grob* drei Schichten der Abstraktion und damit der Komplexität dargestellt werden. Die unterste Schicht umfasst Software, welche Hardware, spezieller eine 3D-API, direkt[1] verwendet, und ferner jegliche Benutzerinteraktion, Szenenobjekte, deren gegenseitige Interaktion, deren Verhalten, als auch beispielsweise physikalische oder beleuchtungsspezifische Modelle und Rahmenbedingungen *hart* implementiert.
Als recht grobkörniges Maß zur Beurteilung der Komplexität von Software in Hinblick auf den gewählten Grad der Abstraktion kann beispielsweise die Anzahl benötigter Dateien (oder Dokumente) zu Rate gezogen werden. Es ist allerdings nur ein hinreichendes Kriterium, kein notwendiges. Zur Verdeutlichung: Im Vergleich nach diesem Maß würden C/C++-Programme relativ zu anderen Sprachen häufig als komplexer gewertet, da schon allein die Konzeption von Code- und Headerdateien, im Regelfall, für eine *größere* Dateianzahl sorgt.

[1] ohne Zuhilfenahme des Konzeptes hierarchisierender Ordnungsstrukturen, wie der eines Szenegraphen

| Abstraktion | | | Komplexität |
|---|---|---|---|
| Raum für weitere Abstraktion | | | |
| Bibliotheken | Code | Daten | 3 |
| generalisierte Funktion-alität (Bibliotheken) | Code + Daten | | 2 |
| Programm (Code) und (Programm-)Daten bilden eine Einheit (Datei) | | | 1 |

Abbildung 2.1: Schichtenmodell der Abstraktion
*Mit steigendem Abstraktionsgrad steigt auch die Komplexität von Software.*

**Gibt es so etwas wirklich ? ... Schicht 2**

Es ist leicht ersichtlich, dass diese Schicht eher theoretischer Natur ist, da sie aus rein pragmatischer Sicht eine Verschwendung von Generalisierungs- und Modularisierungspotential darstellt. Durch den Preis steigender Komplexität auf der Ebene der Abstraktion und der Summe der möglichen unterscheidbaren Bausteine seitens der *Kapselung* und *Modularisierung*, erkauft man sich letztlich die positiven Möglichkeiten der Wiederverwertbarkeit von Konzepten und sogar Teilen des Programmcodes selbst.

Oben aufliegend also, die zweite Schicht - die Programme, die aus *softwaretechnischen*, als auch aus Gründen der effizienten und strukturierten Ordnung einen externen Szenegraphen verwenden. Am Beispiel rasterisierter Graphik steht dieser, einer sonst ungeordneteten Menge von beleuchteten oder unbeleuchteten Primitiven gegenüber. Die Programme dieser Schicht implementieren die sonstigen Eigenschaften der Szene, und auch des Interaktions- bzw. Abhängigkeits-Geflechts ihrer Daten. Seitens der Abstraktion kann man den Sprung in diese Schicht als die *Abstraktion von Funktionalität* bezeichnen. (vgl. Abbildung 2.1)

### 2.1.1 Herrsche und teile ... teile und herrsche

„*Divide et impera.*“

Ludwig XI. (1423-1483)

Beide genannten Schichten teilen das Merkmal von Abstraktionspotential im Sinne der Softwaretechnik. So können die Physik, als auch jegliche Form der Objektinteraktion und des Objektverhaltens verallgemeinert und somit unter dem Aspekt der *Trennung der Belange* und der *Wiederverwertbarkeit* codeextern gelagert werden. Software, welche auf diese Form der *Modularisierung* setzt, fällt in die dritte und von Seiten der Komplexität vorerst obersten Schicht möglicher Software. In dieser Schicht umfasst die Trennung von Daten und Code alle individuellen Verhaltensweisen und Erscheinungsformen von Szene und Szeneobjekten. Diese sind somit Teil einer extern gelagerten Konfiguration. Das DRAS fällt in genau diese Ebene der abstraktiven Komplexität. Eine weitere Eigenschaft des DRAS ist die einfach gestaltete modulare Erweiterbarkeit, basierend auf einer Plugin-Architektur. Diese macht im Rahmen des abstraktiven Schichtenmodells deutlich, dass bezogen auf die Ebenen möglicher Abstraktion und Generalisierung grundsätzlich kaum Grenzen gesetzt sind.

Das vorgestellte Schichtenmodell lässt sich bei der Analyse der *Komplexität durch Abstraktion* offensichtlich beliebig verfeinern. Die Idee dem *Kerncode*[2] unbekannter Plugins kann diesbezüglich als Beispiel dienen.

## 2.2 Das DR - Animations System (DRAS )

Im Kontrast zu der einleitend stark abstrahierenden Sicht auf das eigentliche Kernproblem dieser Arbeit, steht im Mittelpunkt unseres tieferen Interesses die Entwicklung einer graphischen Benutzeroberfläche. Es wird sich jedoch spätestens im Kapitel 3 der Anforderungsanalyse der Problemstellung herausstellen, dass die scheinbar trivial anmutende Aufgabenstellung, je nach Bezugsrahmen den man sich setzt, abgebildet werden kann auf eine Verkettung mehrerer verallgemeinerbarer Einzelprobleme. Aber betrachten wir zunächst in welcher abstraktiven Schicht das DRAS anzusiedeln ist.

[2] Der *Kerncode* bezeichnet einen zentralen Bereich eines Systems, der die groben Muster potentieller Plugins vorgibt, und mit fortwährender Entwicklungsdauer immer weniger Änderungen unterworfen wird.

### 2.2.1 Beschaffenheit der Konfigurationsdatei/en

Allgemein ausgedrückt fungiert das DRAS als ein Abstraktions-Layer, welcher die oben genannten Eigenschaften einer Szene und ihres Inhaltes frei konfigurierbar entgegennimmt und die entsprechende Szene darstellt. Die Besonderheit liegt also im Ausmaß der Einsetzungsmöglichkeiten *eines* einzelnen Systems (interagiergender gekapselter Softwarebausteine) ohne die Essenz des *Programmes*, den Code selbst nämlich, ändern zu müssen. Die zusätzliche Abstraktion der *generalisierten* Pluginarchitektur des DRAS, ist ein mächtiges Mittel, um die Basis an *Kerncode* möglichst unangetastet weiter verwenden zu können. Es ist also grundsätzlich möglich, jegliche Parameter der Szene in XML -Dateien deklarativ zu kodieren. Darum geht es in dieser Arbeit, nach wie vor, um die Entwicklung einer graphischen Benutzeroberfläche, als intuitiv bedienbare Schnittstelle zur Editierung von XML - Inhalten.

**Reicht denn kein XML –Editor ?**

Es muss deutlich zwischen einem reinen XML –Editor und dem gewünschten Zielprogramm unterschieden werden; beide Programme teilen die Motivation des Bearbeitens von Dokumenten, verfasst in XML, doch teilen sie keineswegs das gleiche Vorwissen über die zu bearbeitenden Dateien. Ein XML–Editor ist ausgelegt auf die standardisierte Bearbeitung unbekannter XML -Dateien. Somit ist er per Definition nicht in der Lage hilfreich auf die Darstellung von Inhalten oder gar die Darbietung funktionaler Möglichkeiten (in Bezug zu den Inhalten) einzuwirken.
In diesem letzten Punkt unterscheidet sich der reine Editor von dem gewünschten Szeneneditor: Er soll gewisse semantische Anforderungen erfüllen, die nicht Teil reinen XMLs sind, sondern von konzeptionellen Aspekten der DRAS geprägt werden.

**Was ist eine Szenedatei ? Was ist ein Szeneneditor ?**

Die Szenedatei ist die Konfigurationsdatei des DRAS, eine XML -Datei. Ein Szeneneditor im Rahmen des DRAS ist ein Programm, welches die Schnittstelle zwischen einem Menschen und der Szenedatei bildet, so dass der Benutzer in keinerlei Kontakt zum eigentlichen XML mehr kommen muss. Aufgrund der hohen Anzahl potentieller Fehlerquellen, die das Editieren großer und komplexer XML–Dateien erschwert, kann der Szeneneditor den Vorgang des Erstellens neuer und Bearbeitens existierender Szenen drastisch vereinfachen.

## 2.3 Begriffsklärung: Ordnung, Struktur und Hierarchie

Es soll möglichst ohne die exakte formale mathemathische Definition (mathematisch) intuitiv dargelegt werden, worum es sich bei Ordnung, Struktur und Hierarchie handelt.

### 2.3.1 Ein allgemeines Objektmodell

Zunächst sei ein *Objekt* eine Einheit, der eine endliche Anzahl an *Eigenschaften* zugeordnet werden kann. Man stelle sich die *Eigenschaften* zur Vereinfachung als reelle Zahlen repräsentiert vor, wobei unter komplexeren Umständen auch Vektoren oder Ähnliches in Frage kommen. Unter diesem (vereinfachten) Blick ordnet man einem Objekt mit $n$ Eigenschaften einen $n$ - dimensionalen Eigenschaftsvektor reeler Zahlen zu.
Möchte man diese *Objekte* einer *Ordnung* unterwerfen, so ist die Ähnlichkeit der beiden Werte eines Eigenschaftspaares (ferner die *Nähe in* einer Eigenschaft) ein geeignetes Mittel. Über zwei Objekten sei zusätzlich eine Funktion f definiert, die eine Matrix $\mathbf{M} \in \mathbb{R}^{K \times 2}$ *zusammengehöriger* Eigenschaftspaare liefert. Die *Zusammengehörigkeit* kann direkt auf Basis des *Eigenschaftstyps*[3] modelliert werden. Zwei *Objekte* heißen *eigenschaftsnah* über einer Eigenschaft $e_i (0 \leq i \leq K)$, wenn die Nähe dieser Eigenschaft einen für $e_i$ definierten Schwellwert $s_e$ unterschreitet.
Es ist nun also denkbar für eine beliebige Anzahl von Objekten, unter vorheriger Festlegung jedes Schwellwertes der *Eigenschaftsnähe* über einer Eigenschaft einen *Beziehungsgraphen* zu erstellen, dessen Kanten zwischen je zwei Objekten deren *Eigenschaftsnähe* über einer gemeinsamen Eigenschaft darstellen. Die *Stärke* einer Beziehung sei abhängig von

1. der *Eigenschaftsnähe* einer gemeinsamen Eigenschaft $e_i$

2. einem festlegbaren Gewichtungsfaktor $b_e$

Die *Gesamtstärke* sei ferner, die gewichtete Anzahl gemeinsamer Einzelbeziehungen zweier Objekte *verknüpft* mit der Nähe derselben. Die Art der Verknüpfung sei frei wählbar und die *Gesamtstärke* repräsentiert durch die Funktion g.

[3] oder. in komplexeren Systemen, z.B. als statistische Abbildung zweier *Eigenschaftstypen* auf die *Zusammengehörigkeit*(In diesem Fall ist ein weiterer Parameter, die *Fehlertoleranz* bezüglich der *Zusammengehörigkeit* zweier Eigenschaften denkbar.)

Es leuchtet ein, dass für eine beliebige Menge an Objekten unendlich viele *Beziehungsgraphen* existieren. Deren Aussehen variiert je nach Wahl der justierbaren Parameter[4]. Dabei beginnt die Palette ihrer möglichen Gestalt bei faserigen chaotisch wirkenden Netzen, in denen jedes Objekt alle Eigenschaften mit unterschiedlichen Partnern bindet, bis hin zu gewöhnlichen ungerichteten zyklischen oder azyklischen Graphen bekannt aus der Graphentheorie.

### 2.3.2 Definitionen und Ableitungen

In diesem Sinne bezeichnet *Ordnung* jeden beliebigen Beziehungsgraphen. Davon ableitbar ist *Struktur* jedes Teilnetz aus einem Beziehungsgraphen, dessen Objekte Beziehungen teilen, deren Gesamtstärke einen vorgegebenen Schwellwert nicht unterschreitet. Unter *Hierarchie* versteht sich eine *Struktur* der Form eines hierarchischen Graphen.
Die wesentliche Implikation dieses Abschnittes ist, dass die *Fokussierung* bestimmter Eigenschaften[5] jeweils unterschiedliche *Strukturen* zu Tage fördert. Während also jeder Beziehungsgraph [6] dem Ist-Zusstand einer *Ordnung* entspricht, also der *Ordnung* selbst, ist eine *Hierarchie* eine *gewichtete Fokussierung*. Dieser Zusammenhang gewinnt im Rahmen von Kapitel 4 besonders an Relevanz.

## 2.4 Allgemeine Beschreibung der Anforderungen

Bis hier konnte oberflächlich dargelegt werden, welche Funktionalität dem DRAS, sowie einem DRAS –Szeneeditor, zugrunde liegt bzw. zugrunde zu liegen hat. In diesem Kapitel sollen die Anforderungen, welche ein Szeneeditor erfüllen muss, aus der Perspektive der *Softwareergonomie*, als auch der Perspektive der *Softwaretechnik* tiefergehend erörtert werden.
Vorweg sei angemerkt, dass sowohl *Softwaretechnik*, als auch *Softwareergonomie*, so fachfremd sie zunächst erscheinen, eine generelle Analogie innerhalb ihrer Entstehungsmotive aufweisen, da sie letztlich beide darauf abzielen Schnittstellen zwischen Progammen (Programmbibliotheken) und Menschen, faktisch belegbar, übersichtlicher zu gestalten. Ihre

[4] Das sind alle genannten Schwellwerte und Gewichtungsfaktoren, die Funktionen f und g, sowie die genaue Definiton der Nähe einzelner Eigenschaftstypen.
[5] durch entsprechende Konfiguration der Parameter
[6] auch: *Beziehungsnetz*

grundsätzliche Unterscheidung beruht auf der Zielgruppe von *Menschen*, und den daraus ableitbaren, zu analysierenden und zu optimierenden *Schnittstellen*, welche sie zu bedienen suchen. Während die Softwareergonomie anwenderorientiert, die Erstellung und Gestaltung von graphischen Benutzerschnittstellen erforscht, untersucht die Softwaretechnik entwicklerorientiert die Möglichkeiten der Erstellung und Gestaltung von Schnittstellen zu Daten und Funkionen.

## 2.5 Softwareergonomische Kriterien

Die Softwareergonomie ist ein verhältnismäßig junger Zweig der Informatik. Seit der Einführung graphischer Benutzerschnittstellen (oder **GUI** (**G**raphical **U**ser **I**nterface)), welche ihrerseits metapher-orientierte Steuerelemente und die (möglichst) selbsterklärende *Präsentation* von Programmfunktionalität implizieren, besteht auch der Bedarf nach einer wissenschaftlich gestützten, ingenieursmäßig strukturierten Vorgehensweise, in Hinblick auf die Erstellung von GUIs. Allgemein umfasst die Softwareergonomie jegliche Bestrebung eine Benutzerschnittstelle eines Programms nach wissenschaftlich fundierten Methoden *benutzerfreundlich* zu gestalten. Ein Benutzer soll sich ohne hohen Lernaufwand, durch das Wiedererkennen selbsterklärender *metaphorischer Steuerelemente* und konsistenter Bedienkonzepte nahezu *intuitiv* in neue Software einfinden können. Beide, *Bedienungskonzepte* und *Steuerelemente*, sollen nach gestaltungstheoretischen, statistisch nachweislich existierenden Prinzipien angeordnet werden und innerhalb der graphischen Benutzerschnittstelle dem Benutzer helfen, die von ihm benötigte Programmfunktionalität leicht zu finden.
Auch wenn viele Kriterien im Text bereits angedeutet wurden, seien hier die grundlegendsten unter ihnen noch einmal tabellarisch dargestellt.
Es hat sich bei der softwareergonomischen *Forschung* zum Beispiel gezeigt[7], dass Quantität der *Funktionalität* und die softwareergonomischen Anforderungen in gegensätzliche Richtungen weisen. Eine sehr ähnliche Erscheinung konnte in Bezug auf Aspekte des Designs von GUIs *festgestellt* werden. Ein Beispiel für ein Objekt der Anschauung, das im Interesse der Softwareergonomie steht, ist das folgendene:

- Menüleisten bei Programmen oder Navigationen bei Webauftritten: Es ist deutlich zu erkennen, dass die Navigation durch Information und die Präsentation auswähl-

[7] wie nicht anders zu erwarten

| | |
|---|---|
| **Aufgabenangemessenheit** | geeignete Funktionalität, Minimierung unnötiger Interaktionen |
| **Selbstbeschreibungsfähigkeit** | Verständlichkeit durch Hilfen / Rückmeldungen |
| **Steuerbarkeit** | Steuerung des Dialogs durch den Benutzer |
| **Erwartungskonformität** | Konsistenz, Anpassung an das Benutzermodell |
| **Fehlertoleranz** | erkannte Fehler verhindern nicht das Benutzerziel, unerkannte Fehler: leichte Korrektur |
| **Individualisierbarkeit** | Anpassbarkeit an Benutzer und Arbeitskontext |
| **Lernförderlichkeit** | Anleitung des Benutzers, Erlernzeit minimal, Metaphern |

Abbildung 2.2: Kriterien der Softwareergonomie
aus http://de.wikipedia.org/wiki/Softwareergonomie

barer Funktionalität sich ähnelnde Funktionen erfüllt. Die Tiefe, Breite und semantische *Ordnungsstruktur* eines Menü- und/oder Navigationssystems[8] stehen im Vordergrund. Das Selbstverständis der Softwareergonomie umfasst dabei besonders die Analyse gängiger Systeme, als auch die Entwicklung von *begründeten*[9] Konzepten für neuartige Gegenvorschläge, die stets statistisch geprüft und bewertet werden müssen.

### 2.5.1 Softwareergonomische Anteile in dieser Arbeit

Da das Hauptziel der Arbeit nicht erfüllt werden konnte, genauer, aufgrund eines komplex angegelegten *System–Designs* die Zeit nicht ausreichte, um bis zu Erstellung eines GUI zu gelangen, können keine softwareergonomischen Anteile formuliert werden. Es ist lediglich anzumerken, dass im Falle der Erfüllung des Ziels die Schnittstelle unter maximaler Einhaltung der oben angedeuteten Kriterien der Softwareergonomie gestaltet worden wäre.[10]

[8] gemeint: System zur Navigation durch Inhalte eines Webauftrittes
[9] gemeint: auf softwareergonomischen Kriterien
[10] Auch wenn dieser Fakt nur minimal Trost spendet.

## 2.6 Aspekte der Softwaretechnik

Zunächst soll geklärt sein, worum es sich bei Softwaretechnik handelt. Ich bediene mich hierfür einer entliehenen Formulierung:

„ ***Software–Technik:*** *Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieursmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen. Zielorientiert bedeutet die Berücksichtigung z.B. von Kosten, Zeit und Qualität.*“ (Bal00, S. 36)

Die Begriffe Werkzeuge, Methoden und Prinzipien werden ferner differenziert. *Methoden* seien demnach „*planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen (im Allgemeinen im Rahmen festgelegter Prinzipien).*“(Bal00, S. 36) Die *Werkzeuge* „*dienen der automatisierten Unterstützung von Methoden.*“(Bal00, S. 38)

Es scheint eine weitere Analogie zwischen Softwaretechnik und Softwareergonomie zu existieren. An Stelle der *softwareergonomischen Kriterien* treten die *softwaretechnischen Prinzipien*. In der folgenden Tabelle sollen einige Prinzipien vorgestellt werden und Methoden, die diesen zugeordnet werden können:

| | |
|---:|---|
| **Hierarchisierung** | Die Zerlegung eines Problems in Teilprobleme, so dass eine Baumhierarchie entsteht. |
| **Modularisierung** | Entwicklung von *Produkten* und *Teilprodukten*, die nur über eine definierte Schnittstelle mit der Umwelt kommunizieren können und sonst kontextunabhängig sind. |
| **Strukturierung** | Entwurf von Programmen, so dass nur Sequenz, Auswahl und Wiederholung vorkommen. |
| **Abstraktion** | vgl. Abschnitt 2.1 |
| **Trennung der Belange** | vgl. Teilabschnitt 2.1.1 |

Abbildung 2.3: Unvollständige Liste von Prinzipien der Softwaretechnik und damit assoziierte Methoden, teilweise aus (Bal00, S. 37)

### 2.6.1 Phasen und Aufgaben

Abschließend sollen nach einer minimalen Sammlung theoretischer Grundlagen zumindest diejenigen Phasen des Softwareentstehungs–Prozesses dargelegt werden, welche innerhalb dieser Arbeit Verwendung fanden. Die genaue Abhandlung der Verwendung folgt allerdings erst in den Kapiteln 3, 5 und 7. Die folgende Aufzählung beruht zu Teilen auf (Bal00, S. 55,98,151–222,696–698,1063–1084) und vorwiegend auf (Som04, S. 314–335):

1. **Analyse:**
   (a) Anforderungsanalyse
   (b) Systemanalyse
   (c) Objektorientierte Analyse
2. **Entwurf:**
   (a) Softwarearchitektur
   (b) Objektorientiertes Design
3. **Implementierung:**
   (a) Objektorientierte Programmierung
   (b) Funktionale Programmierung
4. **Test:**
   (a) Modultests
   (b) Integrationstests

#### 2.6.1.1 Analyse

Die *Anforderungsanalyse* umfasst die Feststellung und explizite Definition der Wünsche und Anforderungen des *Kunden* oder *Auftraggebers*. Daraus folgt die *Systemanalyse* als gröbste Ableitung aus den Anforderungen. Es werden die *Systemgrenzen* zur *Umwelt*, die für eine Fragestellung als relevant betrachteten *Systemelemente*, sowie deren gegenseitige Beziehung extrahiert.
Die letzte Phase, die *objektorientierte Analyse*, verfeinert die systemanalytischen Ergebnisse und trägt die aus den Anforderungen gesammelten Daten und Ableitungen zusammen. Sie ist Teil der objektorientierten Modellierung und leitet diese ein.

#### 2.6.1.2 Entwurf

Die *Softwarearchitektur* „*beschreibt die Struktur des Software-Systems durch Systemkomponenten und ihre Beziehungen untereinander.*“(Bal00, S. 696). Durch die Verfeinerung der Ergebnisse, welche die *objektorientierte Analyse* liefert, erzeugt man innerhalb der Entwurfsphase das *objektorientierte Design*. Das Design selbst kann mittels **UML** (**U**nified **M**odeling **L**anguage) und in der Regel unter der Nutzung entsprechender UML–*Werkzeug*e vollzogen werden.

#### 2.6.1.3 Implementierung

Der Zusammenhang bzw. Unterschied zwischen zwei wichtigen unter allen möglichen Formen der Programmierung wird unter Teilabschnitt 4.1.2 erörtert[11] Es sei dorthin verwiesen.

#### 2.6.1.4 Test

Die Verwendung des Prinzips der Modularisierung impliziert das Testen kleinster Moduleinheiten (*Modultests*), als auch deren Zusammenspiel auf höheren Systemebenen. Letztere Tests heißen *Integrationstests*, und sind erst nach erfolgreichen *Modultests* der zu integrierenden Bausteine möglich. Beide fallen noch in den Bereich der *Low–Level–Tests*.

Im Rahmen dieser Arbeit wurde einerseits zum Zweck des Debugging und anderseits, zur ständigen Möglichkeit der Überprüfung beliebiger Programmzustände, eine minimale *Debug/Testing–Library* entwickelt. Mittels Präprozessor definierten Makros lassen sich auf einfache Weise Ausgaben in beliebige Komponenten einflechten. Ein zusätzlich[12] implementierter Präprozessor kann nun, über eine zentrale Datei steuerbar, dateiweise Ausgaben ein– bzw. ausschalten. Ferner werden die Ausgaben hierarchisiert dargestellt in dem Sinn, dass die Ausgabe der Funktionsaufrufe mit wachsender Tiefe in der Aufrufskette, stetig weiter eingerückt werden. Auf diese Weise wird, abgesehen von den Ausgabedaten selbst, die Aufrufshierarchie implizit visuell dargestellt.

[11] Gemeint sind die *prozedurale* und die *objektorientierte* Programmierung.

[12] implementiert in Perl, da sehr geeignet für schnelle und bequeme Verarbeitung von Textdateien

# Kapitel 3

# Analyse und Eingrenzung der Anforderungen

## 3.1 Idee eines Szeneneditors in dreieinhalb Variationen

Der Idee eines Editor für das Bearbeiten vorhandener und das Erstellen neuer Szenedateien des DRAS gingen ursprünglich eine Reihe von Gesprächen mit dem Entwickler des DRAS voraus, während welcher zunächst mögliche Szenarien durchgespielt wurden. Da auch dem Auftraggeber nicht von Anfang an klar vor Augen stand, welche Anforderungen einem zu definierenden Ziel für diese Arbeit zugeordnet werden sollten und welche nicht, spielten wir im groben drei Szenarien durch, die in diesem Abschnitt dargestellt werden sollen.

### 3.1.1 Ideen 1 und 2: Online vs. Offline

Die wesentlichste Unterscheidung an Möglichkeiten ist:

1. **Online–Konfigurationseditor**
2. **Offline–Konfigurationseditor**

*Online* weist darauf hin, dass die Konfiguration zur Laufzeit des DRAS stattfindet. *Offline* bezieht sich auf eine Art spezialisierten XML–Editor (vgl. (Rau05, S. 60)). Die Lösung des *Online*-Editors ist gleich aus mehreren Gründen schwierig, wenn auch nicht undenkbar:

1. Zum jetzigen Zeitpunkt ist das DRAS *nur* auf *einen* Bootvorgang hin konzipiert, da für Änderungen am laufenden System kein Einlesen einer neuen Konfiguration

nötig ist. Das DRAS implementiert eine *funktorbasierte*[1] Schnittstelle (ferner bezeichnet als der *OptionContainer*[2], die eine der zentrale Klassen des *Kernsystems* darstellt. Jede von *OptionContainer* abgeleitete Klasse, verfügt über die Möglichkeit ihre Memberdaten, mittels *Getter*– und *Setterfunktoren* in Abhängigkeit von Zeichenketten zu registrieren. So ist jeder *OptionContainer* bezüglich seiner vermerkten Membervariablen hin während der Laufzeit konfigurierbar. Allerdings völlig entkoppelt vom XML–Hintergrund des Bootvorgangs.

2. Der zweite Punkt geht einher mit dem ersten: Das Design des DRAS sieht bei der *Bindung*[3] zwischen einer *Option* und eines *StateValue* die Übergabe eines Getter- bzw. Setterfunktors an den *StateValue* vor. Dieser kennt die *Option* selbst nicht. Ferner sind bislang keine Möglichkeiten vorgesehen die *Bindung* wieder aufzuheben. Änderungen am *Kernsystem* des DRAS wären somit im Sinne einer *Online*-Lösung unvermeidlich.

3. Ein drittes, der Vollständigkeit halber zu erwähnendes Problem, stellte sich uns, durch das Fehlen einer *einheitlichen* Lösung für das Laden *und* Speichern von XML–Dokumenten.

### 3.1.2 Idee 3: CMS vs. Konfigurationseditor

Die dritte gedankliche Option, im Rahmen der getroffenen Überlegungen, war die grundsätzliche Fragestellung, ob ein Konfigurationseditor für das DRAS nicht prinzipiell den notwendigen Vorbau für ein darauf aufsetzbares **CMS** (**C**ontent **M**anagment **S**ystem) darstellt. Der Editor ist gedacht, als Mittel zur Einstellung beliebiger Systemparameter. Unterwirft man diese Möglichkeit gewissen (benutzerabhängigen) Restriktionen und stattet einen Editor mit vereinfachten Benutzerschnittstellen aus, so genügt er zumindest den konfigurativen Anforderungen eines CMS. In diesem dritten Szenario tuen sich jedoch zusätzlich Konzepte von Client/Server–basierten Editorlösungen auf[4]. Ein CMS bedingt Multiuser–Szenarien, mit den entsprechenden administrativen und netzwerktechnischen

[1] gemeint: Der Oberflächensignatur nach prozedurale Funktionen, zur Kapselung eines Objekts. *Funktoren* ermöglichen anonymen Zugriff auf die Funktionalität von Objekten. vgl. (Rau05, S. 20,38)

[2] vgl. (Rau05, S. 25,28,36-38))

[3] siehe Teilabschnitt 3.2.1

[4] gemeinte Lösung: arbeitet *online* im Sinne von Netzwerken, aber *offline* im Sinne des Laufzeitkriteriums

Implikationen für zusätzlichen programmiertechnischen Overhead.

### 3.1.3 Idee $3\frac{1}{2}$: CGI basierte Lösung

Sehr interessant erschien allerdings auch der Versuch einer webbasierten Editorkomponente, da dort der Client/Server–Problemanteil wegfiele. Das Problem hier: Interaktive Web-Auftritte[5] erfordern die Verknüpfung von bis zu vier Skript- und Markupsprachen zu einer als Gefüge arbeitenden Software. Die Implementierung eines solchen Systems ist eine von der Fehleranfälligkeit nicht zu unterschätzende Aufgabe. Dieser Aspekt löste auch diese Option auf. Im nächsten Abschnitt folgt also was, nach Abzug aller genannten Optionen, als Ziel dieser Arbeit definiert werden konnte.

## 3.2 Definition der Anforderungen

Nachdem nun also ein Negativabbild des Zieles besteht, sollen die gewünschten Anforderungen im Wesentlichen konkretisiert werden. Der Auftraggeber wünscht:

- einen GUI basierten Konfigurationseditor für XML–Dateien
- die Konfiguration ist laufzeitunabhängig umzusetzen (*also offline*)

### 3.2.1 DRAS: Bindung zur Kommunikation

Das DRAS verwendet zwei konfigurativ miteinander verknüpfte Konzepte. Es stellt den *OptionContainer*, als eine Form einheitlicher Schnittstelle für beliebige Klassen, zu klasseninternen Daten, zur Verfügung. Diese Funktionalität ist ausschließlich erbbar. Ableitungen des *OptionContainers* können über sogenannte *Options* Interna[6] einheitlich nach außen hin freigeben, sowohl lesend, als auch schreibend. Dabei erlaubt eine *Option* über ihren Namen (eine Zeichenkette) einen Zugriff auf diese. Diese Möglichkeit erlaubt bei genauer Betrachtung beliebigen Bausteinen Zugriff auf andere Bausteine zu nehmen, ohne deren Implementierungsdetails zu kennen.
Darauf fußt ein wesentliches Konzept zur Synchronisation und Kommunikation zwischen verschiedenen DRAS–System-Komponenten. Es werde ferner als *Bindung* bezeichnet. In

[5] gemeint: im Sinne der Interaktivität einer Desktopapplikation

[6] gemeint: eigene Memberdaten, Memberdaten von Memberobjekten, sofern diese vollen Zugriff erlauben

Abbildung 3.1 soll vereinfacht dargestellt werden, wie die *Bindung* eines *OptionContainers* (einem Szeneobjekt) und den *StateValues* eines DRAS–Zustandsautomaten Kommunikation, oder allgemeiner *Datenaustausch* ermöglicht.

#### 3.2.1.1 Verallgemeinerung der Bindung

Es lässt sich schließen, dass aus Sicht eines Editors *Bindung* unabhängig ist von ihrer Bedeutung, innerhalb eines Systems wie dem DRAS. Ferner soll *Bindung* also verstanden werden, als die Manifestation einer Beziehung zwischen zwei System-Objekten[7] in Form einer XML-Struktur in einer Konfigurationsdatei. Ferner gelte, dass beide System-Objekte ihrerseits auch Teil der Konfiguration sind. Diese Form von *Bindung* sei die rein *strukturelle Bindung*.

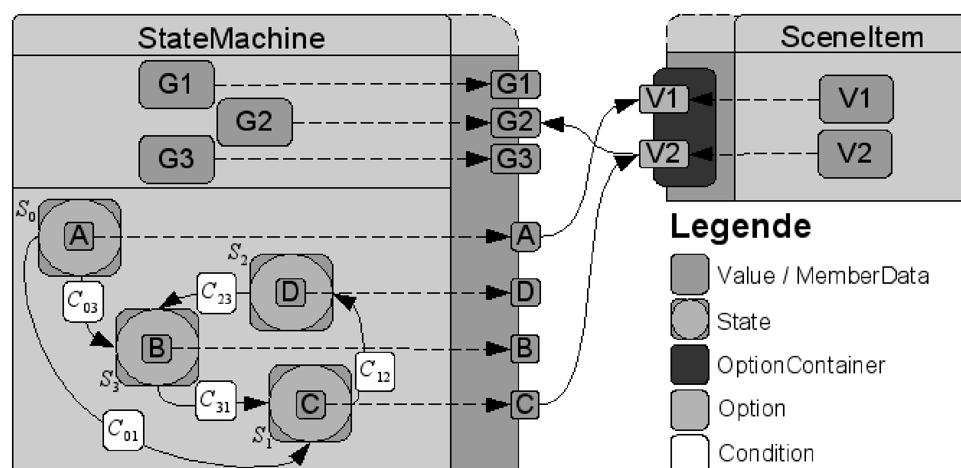


Abbildung 3.1: DRAS: Bindung als Datenaustausch
*Beispielhafte vereinfachte Darstellung bidirektionalen Datenaustauschs zweier DRAS–Komponenten.*
***OptionContainer:*** *Dies ist eine Oberklasse der meisten DRAS–Bausteine und ermöglicht das Bereitstellen interner Memberdaten nach Außen. Dies geschieht einheitlich durch Instanzen der Klasse „Option“.*
***Option:*** *Funktoren ermöglichen einer „Option“ ohne eine Klasse **X** zu kennen, Zugriff auf interne Daten von **X**. Die „Option“ ermöglicht ferner den lesenden und/oder schreibenden Zugriff nach Außen.*

[7] gemeint: Objekte im Sinne der Programmierung

#### 3.2.1.2 Typabhängige Bindung

Im Blickwinkel des allgemeinen Objektmodells folgt für die *strukturelle Beziehung* zweier System-Objekte, dass diese immer zwei *zusammengehörige* Eigenschaften bindet. Die *strukturelle* Bindung ist unabhängig von der Art der mathematischen Modellierung der Zusammengehörigkeit definiert. Die Definition der *strukturellen Bindung* vernachlässigt somit aber einen Aspekt, welcher im DRAS bei der Verknüpfung einer *Option* mit einem *StateValue* eine notwendige Randbedingung darstellt: die *Typgleichheit* zweier *gebundener* Einheiten. Mit Verweis auf Abschnitt 2.3.1 werde dieser Spezialfall verallgemeinert. Demnach sei die *typabhängige Bindung*, die *strukturelle Bindung* zweier Eigenschaften gleichen Typs.

#### 3.2.1.3 Ableitung

Da das DRAS frei konfigurierbar ist, muss eine *Bindung* sich notwendiger Weise in der Szenedatei, der Konfiguration des DRAS manifestieren. Die Konfiguration ist der Ankerpunkt aller Überlegungen im Rahmen dieser Arbeit. Es folgt also, dass die DRAS-Bindung, aus Sicht der Konfiguration nur eine bestimmte XML-Struktur, in Abhängigkeit von zwei Systemobjekten darstellt, welche wiederum innerhalb der Konfiguration in Form von XML-Strukturen zu finden sind. Dies impliziert, dass aus Sicht eines Konfigurationseditors eine allgemeine Möglichkeit gefunden werden muss, eine *Bindung* auf GUI Ebene konsistent nach XML zu überführen. Umgekehrt aber auch die Fähigkeit unbekannte Konfigurationen zu lesen, *Bindungen* zu erkennen, und diese schließlich in das GUI abzubilden. Die Begriffe *Bindung* und *Konnektion* seien, bis zu einer differenzierteren Betrachtung in Kapitel 5, als *synonym* zu verstehen.

### 3.2.2 Verfeinerung der Zielsetzung

Im diesem Abschnitt erfolgt die weitere Verfeinerung der Anforderungen. Dabei erfolgt dies zunächst eng am Rahmen der Hauptanforderung: dem GUI. Es liegen eine Reihe von Forderungen bezogen auf die Verwendung bestimmter Steuerelemente und die Implementierung spezieller Bausteine vor:

- Die Szeneobjekte seien ihrer hierarchischen Struktur nach, in einer Baumansicht[8] anzuordnen, welche der Ankerpunkt zur Navigation durch Szeneelemente sein soll.

[8] gemeint: TreeView

- Zur Auswahl in die Hierarchie einzufügender[9] Objekte soll ein Auswahlpanel zur Verfügung stehen, vergleichbar mit den Auswahlpanelen zur GUI-Generierung in vielen integrierten Entwicklungsumgebungen.

- Zu den grundsätzlichen Operationen auf dem TreeView gehören das Hinzufügen und Entfernen von Knoten der Hierarchie. Diese Funktionalität sei per Kontextmenü erreichbar.

- Ferner seien die Funktionen Kopieren und Einfügen innerhalb des Kontextmenüs, als auch über Tastenkombination aufrufbar.

- Die Möglichkeit Szeneobjekten (typabhängig) individuelle Konfigurationspanele zuordnen zu können sei implementiert.

- Das DRAS lässt die Definition von *Zustandsautomaten*[10] zu. Gerade diese kann sehr mühselig sein. Es soll eine graphisch umgesetzte Lösung zur Definiton und Verschaltung von Zustandsautomaten im Sinne des DRAS gefunden werden. Vorgaben diesbezüglich sind:

  – Die Zeichenfläche für die Editierung von Zustandsautomaten soll rasterisiert und mit einer Einrastfunktion ausgestattet sein.

- Das Konzept der *typabhängigen Bindung* (vgl. Abschnitt 3.2.1) soll für beliebige *bindbare* DRAS Objekte ermöglicht werden. Darunter fällt auch die *Bindung* einer *Option* mit dem Wert eines *Events*[11]. Ein entsprechendes Panel ist zu implementieren. Dabei ist eine graphische Lösung vorzuziehen, aber nicht zwingend.

- Zur grundsätzlichen Funktionalität des Editors sollen gezählt werden

  – das Erstellen neuer Dokumente,

  – das typsichere Laden und Speichern von Dokumenten (auch hinsichtlich der Konnektion)

  – für den Fall einer graphischen Umsetzung der Konnektionsschnittstelle, soll deren Ist-Zustand speicherbar und wiederherstellbar angelegt werden.

[9] und semantisch tatsächlich einfügbarer
[10] vgl. (Rau05, S. 39–40)
[11] vgl. (Rau05, S. 35)

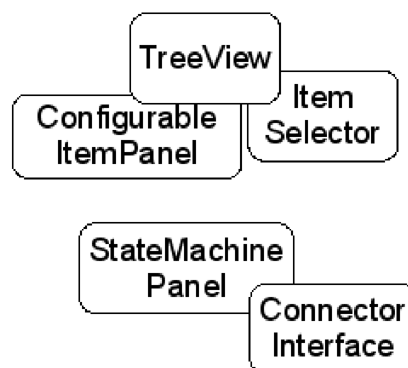


Abbildung 3.2: Mindmap nötiger Bausteine
*Diese Bausteine ergeben sich (grob) aus den Anforderungen. Räumliche Nähe deutet auf kontextuelle Nähe hin*

Interessant bezüglich der recht umfangreichen Liste von Anforderungen ist, dass keine Vorgaben getroffen werden, hinblicklich der Repräsentation der Semantik einer Szenedatei. Das bedeutet für die weitere Vorgehensweise, dass zunächst ermittelt werden muss welche Möglichkeiten grundsätzlich zur Verfügung stehen dieses Problem, im Rahmen wachsender Systemstrukturen[12], handhabbar zu machen.
Im Anschluss muss begründend abgewogen werden, welches das geeignetste Mittel unter den gefundenen ist. Die soll Abschnitt 3.3 klären.

## 3.3 Zwei konkurrierende Lösungsansätze

Im Sinne einer späteren *Systemanalyse* und der darauf fußenden Softwarearchitektur (vgl. Abschnitt 5.5) soll noch in diesem Abschnitt eine grundsätzliche Entscheidung gefällt werden, hinsichtlich zweier denkbarer Konzepte zur Bewältigung der Anforderung der Erweiterbarkeit des zu erstellenden Systems. Die Erweiterbarkeit dieses Systems ist allein aus der Erweiterbarkeit des DRAS heraus zu fordern. Aber welche zwei Möglichkeiten sind gemeint ?

### 3.3.1 Plugins vs. XML-Beschreibungssprache

Die Repräsentation der Semantik von XML-Dateien kann grundsätzlich auf zwei Arten implementiert werden. Eine Möglichkeit wäre es über ein Framework *Plugins* zu realisie-

[12]also im Rahmen einer wachsenden Sprache

ren. Dies würde von einem Entwickler eines DRAS-Plugins erfordern, dass er zusätzlich ein Plugin für den Szeneneditor implementieren müsste. Die Motivation der Entstehung von XML fußt gerade auf dem gegenteiligen Ansatz. XML soll genau dieser Art der Vorgehensweise hilfreich entgegen wirken.

Im nächsten Kapitel werden Möglichkeiten dargestellt die Beschreibung semantischer Strukturen von XML-Dateien zu vollziehen. Dies ist auch der Ansatz der vorliegenden Arbeit: Die Erweiterbarkeit des Szeneneditors soll gewährleistet werden durch die Verwendung einer XML-Beschreibungssprache. In diesen Zusammenhang ist die *Bindung* von DRAS-Systemkomponenten konzeptionell einzuflechten. Dem Entwickler eines DRAS-Plugins soll auf diese Weise die Möglichkeit gegeben werden die von einem Plugin erwarteten XML-Strukturen zu beschreiben.

# Kapitel 4

# Abstraktion von Information

## 4.1 Ordnende Strukturen

In diesem Kapitel wollen wir tieferen Einblick in die Möglichkeiten der Abstraktion von Daten nehmen, da XML bei genauerem Hinsehen nur eine Sammlung syntaktischer Regeln zur textuellen Darstellung beliebiger Dateninhalte darstellt. Um der grundsätzlichen Beschaffenheit von Daten näher zu kommen sollen uns Ordnungsprinzipien menschlichen Denkens als Beispiel dienen. Dem Leser sei zunächst nahegelegt sich Abschnitt 2.3 und besonders die Ableitungen von Teilabschnitt 2.3.2 in Erinnerung zu rufen.

Der wesentliche Kern des allgemeinen Objektmodells ist, dass es über seine Parameter justiert auf Objekte angewendet werden kann, und dabei nicht vorgibt, ob deren Beziehungsstrukturen eher Netzen oder poly– oder monohierarchischen Graphen entsprechen.

Das beschriebene Modell ist jedoch nicht zur direkten Anwendung, sondern vielmehr zur Veranschaulichung einer Denkweise dienlich, die annimmt, dass eine oder mehrere *ordnende* Strukturen auf ein und derselben Menge von Objekten synchron existieren können. Spezieller, dass die Justierung des Systems durch den Betrachter, interpretierbar wird, als dessen fokussierende Wahrnehmung seiner Umwelt. Darum ist es auch geeignet auf *Datenstrukturen* genauer *Datenhierarchien* angewendet zu werden. Es gibt nicht die hierarchische Anordnung von Daten, sondern beliebig viele strukturelle Möglichkeiten. Allerdings ist es wenig zweckmäßig auf eine Hierarchiestruktur gänzlich zu verzichten.

Schon in Abschnitt 2.1 wurde dieser Zusammenhang deutlich, als von Ordnungsstrukturen in Software die Rede war, welche am Beispiel eines Szenegraphen dargestellt wurden, als

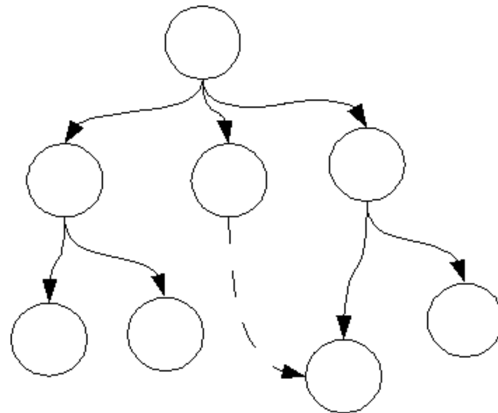

Abbildung 4.1: Allgemeine Darstellung hierarchischer Ordnungsstrukturen

Mittel der Abstraktion von Programmstrukuren. Der mögliche Zweck der Generalisierung funktioneller Programmanteile und deren *Ballung* zu einer Einheit, entspricht allerdings weniger dem etwas senilen Wunsch nach wachsender Komplexität [1], als vielmehr der Ordnung *zusammengehörig* wirkender, von andersartigen unterscheidbarer, aber untereinander gemeinsame Eigenschaften teilender Einheiten.

Ein Verweis auf Georg Cantors Erklärungsansatz des Begriffes *Mengen* (Can95) offenbart die Tragweite und den Raum des Einflusses von *Hierarchiestrukturen*:

„*Unter einer Menge verstehen wir jede Zusammenfassung von bestimmten wohlunterschiedenen Objekten unserer Anschauung oder unseres Denkens zu einem Ganzen.*“

Eine Entsprechung bietet die folgende Definition, welche ein *System* versteht als „*eine Menge von Elementen, die miteinander durch Beziehungen verbunden sind und gemeinsam einen bestimmten Zweck zu erfüllen haben.*“ (Mül00, S. 48)

[1] dies ist eine eher notwendige Randerscheinung

### 4.1.1 Bildung einer Ordnungshierarchie

**Ein kleines Ordnungsspiel ...**

Der Vergleich beider Auffassungen offenbart, neben einer subtilen Analogie, auch einen Unterschied; den *Zweck* einerseits, und die *Beziehung* andererseits. Während die *Menge* selbst die einzige *Beziehung* zwischen den ihr zugeordneten *Elementen* sein darf und dieser Begriff somit nicht eigens verwendet wird, verleiht der Urheber des zweiten Zitates dem *Zweck* Nachdruck. Insofern ist das *System* eine Ableitung der *Menge*. Da jedoch die *Menge* selbst einen Zweck innerhalb der Mengenlehre erfüllt, die selbst Teilbereich der Mathematik ist, kann die *Menge* als Element des Systems *Mengenlehre*, einem Subsystem der Mathematik verstanden werden, welches seinen Zweck unter anderem in den Computerwissenschaften jeden Tag neu erfüllt; als kleines Beispiel: in der Definition, Entwicklung und Analyse von *Systemen*.

Die im letzten Absatz geführte Ordnung der zwei unterscheibaren Begrifflichkeiten wirft also zwangsläufig die Frage auf, welche ordnende Hierarchie denn die gültige ist. Die Antwort liefert Abschnitt 2.3: Der gewählte Rahmen oder *Kontext* entscheidet welche Hierarchie die geeignete zu einem Zeitpunkt ist[2]. Das System als Ableitung der Menge zu sehen ist ebenso trivial erfassbar, wie die Menge als System zu sehen. Beide Ordnungen gelten. Allerdings nicht im Rahmen der Definition eines Systems, die ja den Fokus auf besondere Eigenschaften lenkt.
Bei der Ordnung von *Daten* verhält es sich keineswegs anders. Wenn Menschen und/oder Programme Daten austauschen, müssen diese geordneten Strukturen unterliegen. Daten sind Objekte des Denkens und somit kontextabhängig hierarchisierbar. Diesen Aspekt versucht die Definition von XML, einer menschen- und maschinenlesbaren Sprachspezifikation zur Darstellung von geordneten (optional über Rechnernetze verteilten) Datenhierarchien zu berücksichtigen.

### 4.1.2 Ordnende Graphen und Objektorientierung

Eine, der geordneten hierachisierenden Repräsentation von Daten, analoge Entwicklung hat auch in der Welt der Programmierung Einzug erhalten. Die Rede ist vom objektorientierten Paradigma. Bis vor dem Zeitpunkt des Pradigmenwechsels wurden, im Sinne der

[2] in anderen Worten: die Fokussierung

Kapselung, Daten, Datentypen und Funktionen als getrennte Objekte der Anschauung gewertet. Dieser Umstand wirkte sich bis auf die unterste Ebene aus; den Programmcode und den Ausrichtungen der Programmiersprachen. Letztlich förderte er sogar die tatsächlich räumliche Trennung der Definitionen von Datentypen einerseits, und der von Funktionalität operierend auf den vorher definierten Typen andererseits. Aus Sicht des Programmierers bedeutete dies zweierlei Fragen:

1. Welche Datentypen brauche ich ?

2. Welche Funktionen brauche ich, die auf meinen definierten Datentypen operieren ?

Es ist leicht ersichtlich, dass die Daten gemäß der geltenden Struktur für Ordnungshierarchien, beschrieben in Abschnitt 4.1 und Teilabschnitt 4.1.1, definierbar sein sollten, um so ein näheres Abbild der repräsentierten Objekte einer außerhalb des Programmcodes liegenden Wirklichkeit zu gewährleisten. Die Abtrennung der Daten von Funktionen, so sehr die Belange auch getrennt erscheinen, steht der menschlichen Wahrnehmung von Zusammengehörigkeit von Dingen, die sich auch räumlich nah sind, eher kontraproduktiv gegenüber und wirkt dadurch *künstlich*, in sich weniger geschlossen, als die Abstraktion über Datentypen. Das Hauptproblem dieser gedanklichen Trennung besteht in großen Vielfalt an Möglichkeiten Funktionen sinnvoll gegen die ihnen zugehörigen Datentypen zu ordnen.
Erst mit dem Einzug des *Datenobjekt–Moduls*[3] und später der *Objektorientierung* konnte dieser Kontrast auch bezüglich des Programmcodes deutlich entschärft werden. Es wurde eine neue Form von Daten-/Code Einheit erdacht, die im Wesentlichen alle Daten und die mit ihnen verknüpfte Funktionalität konzeptionell miteinander verband, in Form von Objekten. Dieser Wandel in der Auffassungen von den *Einheiten der Programmierung* ermöglicht (wie üblich unter Erhöhung der Abstraktionsgrades) die Modellierung und Betrachtung von Software in einer dem menschlichen Denken *analogen* Weise. Aus Sicht der Programmierung müssen die oben gestellten Fragen umformuliert werden:

1. Welche Arten (Klassen) von Objekten brauche ich ?

2. Welche Eigenschaften (Daten) haben die Objekte, die ich brauche ?

3. Was muss ein Objekt können ? Was nicht ? (Welche Funktionen braucht es ?)

[3] vgl. (Bal00, S. 1035)

## 4.2 XML eXtensible Markup Language

Das Motiv und der Zweck von Abschnitt 4.1 und 2.1- in Verknüpfung mit dem letzten Teilabschnitt, ist die Aufschüttung eines soliden Bodens gedanklicher Grundlagen von Objektorientierung. Diese ist der rote Faden, angefangen bei der menschlichen Wahrnehmung und internen Wirklichkeitsrepräsentation, über die Darstellung von ordnenden Datenhierarchien bis hin zu der modernen Abstraktion hinblicklich eines assoziativen Progammierparadigmas.
Im Rahmen dieses Abschnittes soll das geordnet hierachisierende Datenformat XML vorgestellt werden; damit verknüpft die mit XML vorgesehenen Möglichkeiten und die sich aus XML ergebenden Notwendigkeiten. Grundsätzlich handelt es sich um eine Spezifikation zur Repräsentation objektorientierter Daten. Eine andere Auffassung des gleichen Gegenstandes ist die folgende: „*XML ist eine Sprache, mit der sich Dokumente beschreiben lassen.*"(Bal00, S. 982). XML ist selbst Nachkomme von **SGML** (**S**tandard **G**eneralized **M**arkup **L**anguage) einer standardisierten Meta–Sprache zur Definition von Dokumenttypen verschiedener *Auszeichnungssprachen*[4]. Aufgrund seiner Komplexität, erreichte SGML in der breiten Öffentlichkeit nie die Akzeptanz die XML heute erfährt. Zu einer Meta–Sprachen-Spezifikation der Gestalt von XML, gehören im Wesentlichen zwei zusätzliche Komponenten von Technologien:

- **APIs** zur Verarbeitung von Daten (und Dokumenten) die XML –Syntax aufweisen
- Sprachen für die Beschreibung der *Beschaffenheit* von Daten (und Dokumenten), die in XML gespeichert werden sollen

### 4.2.1 Verarbeitende APIs

Es existieren zwei grundsätzlich unterschiedliche Konzepte in Bezug auf das Laden von XML–Dokument–Daten:

- **SAX** (**S**imple **A**PI for **X**ML)
- **DOM** (**D**ocument **O**bject **M**odel)

[4]Unter einer Auszeichnungssprache (engl. **M**arkup **L**anguage (**ML**), wörlich: *Aufwertungssprache*), versteht man eine Menge von Symbolen zur Markierung von Formaten in textuellen Inhalten, und zum Teil auch Beschreibungen des auf Dokumentdaten anzuwendenden Verfahrens. In diesem Sinne unterscheidet man zwischen *deskriptiven* und *prozeduralen* Markup Sprachen.

Vorneweg sei angemerkt, dass beide APIs unabhängig von einer speziellen Programmiersprache definiert sind. Dies ist daher vorteilhaft, als dass es die breite Verfügbarkeit der beiden APIs nicht an die Geschmäcker oder Anforderungen der API-Entwickler bindet, sondern vielmehr konzeptionell bereits eine große Palette verschiedenster Zielgruppen berücksichtigt. Als kleines Beispiel: **C++** bietet Möglichkeiten, die in dieser Form nicht zwangsweise allgemein verbreitet sind. Speziell die Option generischen parametrisierten Codes oder die Verwendung multipler Ableitungen erschweren eine automatisierte Portierung in andere Programmiersprachen, da dort häufig die keine Entsprechungen definiert sind.

**SAX 1/2 : Ereignisbasierte Parser mit Rückruffunktionalität**

Bei der Programmier–Schnittstelle SAX handelt sich um einen nachfrageorientierten Parser für XML Dokumente. *Nachfrageorientiert* bezieht sich auf den Zeitpunkt der Verfügbarkeit der Daten aus dem Dokument. Die Funktionsweise von SAX ist eventgesteuert, und ermöglicht einer externen Anwendung Callback–Funktionen für vordefinierte Parsing–Zustände zu hinterlegen, die bei Eintritt des entsprechenden Ereignisse aufgerufen werden, um selbst die von einer Applikation benötigten Informationen zu beziehen.

**SAX 2/2 : Wofür eignet es sich ? Wofür nicht ?**

Die Konzeption von SAX liefert offensichtlich zu keinem Zeitpunkt das gesamte Dokument. Es obliegt der Applikation selbst sich an den entsprechenden Ereignispunkten mit Daten zu versorgen, den weiteren Fluss anzustoßen oder gegebenenfalls den Parsing–Vorgang abzubrechen. Für ein Programm, welches möglichst viele und weit (über die XML–Hierarchie) verteilte Daten benötigt ist SAX weniger geeignet, da der Programmieraufwand für die Callback-Funktionen zum Aufbau größerer Teile der Baumstruktur, bei weitem größer ist, als die Verwendung von DOM .
Hat eine Applikation es hingegen mit einer großen Datenmenge zu tun, oder nur mit dem Transfer von großen XML–Dokumentdaten–Volumina, eignet sich SAX deutlich besser. Hier ist also das Haupteinsatzgebiet von SAX anzusiedeln.

**DOM 1/2 : Aufbau der internen Ordnungsstruktur**

Einen konträren Ansatz wählt das DOM; die vollständige Repräsentation der Datenhierarchie ist nach Aufruf der Parser Routine im Speicher (als Baumstruktur) vorhanden und

steht von diesem Moment an für informative und manipulative Zugriffe zur Verfügung. [5]

**DOM 2/2 : Wofür eignet es sich ? Wofür nicht ?**

Es verhält sich von der Einsetzbarkeit invers zum SAX. Das DRAS verwendet aus eben diesem Grunde das DOM: Um die *interne Repräsentation* des Szenegraphen, sowie der *Kontrollstrukturen* zur Steuerung, Abfrage und Beeinflussung der Szenenobjekte aufzubauen[6], benötigt es die Daten nur einmal (initial), dafür aber in ihrer Vollständigkeit, bei nicht allzu großem Dokumentvolumen. Also ist das DRAS ein äußerst gutes Beispiel für ein System, welches sich der Konzeption des DOM nach, im Zentrum dessen Zielgruppe befindet.

### 4.2.2 Wesentliche Bausteine von XML

Da nun die beiden grundlegenden APIs als Zugang zu *XML–Dokumenten* ausreichend vorgestellt worden sind, ist an der Zeit sich die Oberflächenstruktur von XML aus direkter Nähe zu betrachten. Erinnert sei an Abbildung 4.1 als Verallgemeinerung jeglicher Form ordnender *Hierarchiestrukturen*. An diesem Modell offenbart sich bereits grob der wesentliche Aufbau beliebiger XML–Dokumente. So seien jegliche Einheiten aus XML Knoten eines hierarchischen ungerichteten azyklischen Graphen. Folglich repräsentiert der Graph das XML–Dokument. Ferner ist die Unterscheidung dreier Typen von Knoten möglich:

- dem Wurzelknoten
- den Knoten an den *Blättern*
- den Knoten die weder Blatt noch Wurzel sind

Tatsächlich ist die Struktur eines XML–definierten Datenbaumes, etwas anders als die Unterscheidung auf *rein* graphentheoretischer Ebene es zulässt. Da eine allgemeine Ordnungsstruktur wie auf Abbildung 4.1 nur die Abstraktion jeglicher Form unterordnender[7] Strukturen repräsentiert, ist dies aber nicht weiter verwunderlich — im Gegenteil: es ist offensichtlich das eine Spezialisierung der Verallgemeinerung nicht wieder die Verallgemeinerung selbst sein kann. Wie aber steht es nun um Bausteine einer XML–Hierarchie ?

[5] vgl. http://www.w3.org/DOM/

[6] also einer eigenen den Szenegraphen umgebenden Ordnungshierarchie / vgl. Abschnitte 2.1 und 4.1

[7] gemeint: hierachisierend

**Es existieren zwei Typen unterscheibarer Einheiten:**

- **Elemente**
- **Attribute**

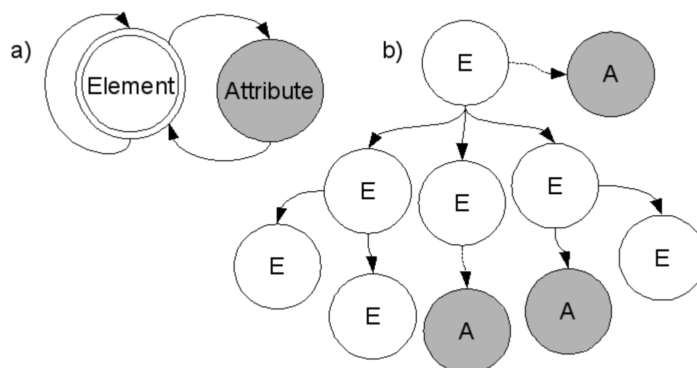


Abbildung 4.2: Vereinfachte Darstellung eines XML–Dokumentes

Da XML eine Ableitung ordnender Hierarchiegraphen ist, müssen die genannten Einheiten letztlich die oben getroffene Unterscheidung allgemeiner Graphen implizieren. So wie oben der Begriff *Knoten* für alle drei Arten von Knoten in Frage kommt, muss entweder das Element oder das Attribut mehrere Arten der Position im Baum abdecken können. Wie in Abbildung 4.2b) ersichtlich, übernimmt dabei das *Element* allein alle Knoten *die nicht Blatt sind.*

### 4.2.3 XML: Beispiel

```
<IchBinEinElement IchEinAttribut=„und das ist gut“>
  <Hausponz />
  <Hausponz mitAttribut=„jaja und schau mal wie ich ende“/>
  <Hausponz mitAttribut=„welches ende ?!?“>
    <Gumpen siehHin=„wie wir ausschauen“>
      <Hingeschmierter/>
    </Gumpen>
    <Beispiel doof=„Alle Attribute sind: doof“/>
  </Hausponz>
</IchBinEinElement>
```

### 4.2.4 Ist XML bedeutungslos ?

Die Antwort lautet: *Ja*. XML ist die Definition einer Syntax zur Hierarchiesierung von Daten. Damit bedeutet sie im Rahmen eines XML–Dokuments nichts. Sie gibt nur die Ordnungsstruktur vor. Eine syntaktische Restriktion die bislang nicht genannt wurde ist, dass Bezeichner für Attribute nur einmalig pro Element erlaubt sind. Allgemein gilt[8]:

1. Ein Dokument hat ein Wurzelelement.
2. Ein Starttag eines Elementes **x** mit Kindelementen hat folgende Syntax: **<x>**
3. Ein Element kann beliebige Elemente enthalten.
4. Ein Element kann beliebige Attribute mit verschiedenen Namen enthalten.
5. Hat ein Element keine Kindelemente, dann endet dessen Starttag auf: **/>**
6. Hat ein Element **x** Kindelemente, dann hat **x** ein Endtag: **</x>**

### 4.2.5 Wohlgeformtheit und Gültigkeit

Wenn keine syntaktische Regel von XML verletzt wird, so heißt ein XML–Dokument *wohlgeformt*. Ist zusätzlich keine beliebig definierbare semantische Meta-Regel verletzt, so heißt ein XML–Dokument *gültig*. Im Rahmen von Meta-Sprachen, zur Definition der semantischen Struktur *gültiger* XML–Dokumente, werden die Abschnitte 4.3, 4.4 und 4.5 folgende XML–Definitions-Sprachen vorstellen: DTD, XSD und SLIDL.

### 4.2.6 Ist XML mono- oder polyhierarchisch ?

Das ist eine gute Frage. Es kommt auf den gesetzten Rahmen der Frage an. Rein syntaktisch gesehen ist ein XML-Dokument immer ein monohierarchischer Graph. Auf semantischer Ebene ist die Frage für XML unentscheidbar, denn XML ist Syntax.

### 4.2.7 Namensräume in XML

Durch die bislang beschriebene Struktur von XML wird schnell ersichtlich, dass Kollisionen von Namen ein ernstzunehmendes Problem darstellen. Darum verwendet XML das, aus anderen objektorientierten Sprachen eingeführte, Konzept von *Namensräumen*. Dabei gilt: „*Namensräume müssen weltweit eindeutig sein.*“ (vgl. (Bal00, S. 974)) Damit dies

[8] vgl. http://www.w3.org/XML/Core/

gewährleistet werden kann, ist es eine Empfehlung, eine nicht-existente **URL** (**U**nified **R**esource **L**ocator) als Namensraum zu verwenden. Ein kleines Beispiel:

```
<pansen:Pansen xmlns:pansen=„www.hundeko.ra/llen/oh.weh“>
  <pansen:Gumpen/>
  <pansen:Gumpen mit=„Brotribut“>
    <pansen:UndWurstOhneInhalt/>
  </pansen:Gumpen>
</pansen:Pansen>
```

In der ersten Zeile dieses Beispiels wurde dem Platzhalter **pansen** mittels des XML-Schlüsselwortes **xmlns** die URL *www.hundeko.ra/llen/oh.weh* zugewiesen. Ohne einen Platzhalter müsste jedem Element der vollständige Namensraum vorangestellt werden. Dies ginge allerdings mit einem drastischen Zuwachs der Dateigrösse einher, und ist wenig wünschenswert.

Bislang ist XML also eine *Syntax* zur Darstellung von *Datenhierarchien*. Ferner existiert für XML ein weiteres Mittel der Ordnung, welches syntax-struktrell entkoppelt ist: Namensräume. Es fehlt also noch die Möglichkeit semantische Beziehungen zwischen den Inhalten von syntaktischen Strukturen zu definieren. Die folgenden drei Abschnitte beleuchten diesen Aspekt in Abhängigkeit von XML.
Die folgenden Beispiele sind äußerst minimal angelegt, und erheben ferner keinen Anspruch auf Vollständigkeit der Darstellung an Möglichkeiten der Definition. Es sollen lediglich die jeweils wesentlichen Eigenschaften einer semantischen Definitionssprache vorgestellt und bewertet werden.

## 4.3 DTD Document Type Definition

Die primitivste Möglichkeit zur Festlegung der Semantik[9] eines XML-Dokuments bietet eine DTD. *Primitiv* bezieht sich auf gestellte Datentypen in DTDs: #PCDATA und #CDATA. Ersterer weist ein verarbeitendes Programm an einen Inhalt zu *parsen*, um ggf. Ersetzungen vorzunehmen oder eine Typprüfung zu vollziehen. Letzterer entspricht der An-

[9]Es sei darauf hingewiesen, dass verschiedene Autoren in diesem Zusammenhang auch schlicht von der *Struktur* eines XML-Dokumentes. Diese Anschauung verträgt sich zunächst nicht, mit der hier vertretenen Auffassung von Struktur, bzw. ermöglicht Verwechslungen. Darum sei die *Semantik* eines Dokumentes auch verstanden, als dessen *semantische Struktur*.

weisung, keine besondere Maßnahme zu ergreifen. Ebenso *primitiv* ist die eingeschränkte Möglichkeit der Angabe der Anzahl von Unterelementen, welche sich auf die Quantoren[10] „+“, „*“ und „?“, also „ein oder mehrmals“, „kein oder mehrmals“ oder „ein oder kein Mal“ beschränkt. Das Fehlen eines Quantors steht für „genau einmal“. Ein kleines Beispiel:

```
<!DOCTYPE Boxer [
  <!ELEMENT Name (#PCDATA) >
  <!ELEMENT Vorname (#PCDATA+)>
  <!ELEMENT Gewicht (#PCDATA?)>
  <!ATTLIST Gewicht Klasse (#CDATA) >
]>
```

Eine weitere Auffälligkeit von DTDs ist die Form der Definition von Attributen. Diese ist syntaktisch entkoppelt von der Definition eines Elementes.

### 4.3.1 Bewertung

Die schwachen Optionen zur Definition von Datentypen sind der größte Nachteil von DTDs. So ist es nicht möglich Zahlenräume in Typen abzubilden. Ein weiterer Nachteil ist die Tatsache, dass DTDs selbst nicht in XML formuliert werden, sich selbst also nicht beschreiben können.

## 4.4 XSD XML Schema Definition

Die Nachteile von DTDs werden mit dem Konzept von XSDs behoben. XSDs stellen eine Reihe vorgegebener Datentypen zur Verfügung, erlauben die Definition neuer Typen und eine objektorientierte Form der Definition von semantischen Strukturen. Ihre Syntax ist zudem XML. Das bedeutet, das eine XSD die semantische Struktur von XSDs beschreiben kann. Es folgt das obige Beispiel der DTD vereinfacht formuliert in XSD:

[10] im Sinne der Quantoren in erweiterten regulären Ausdrücken

```
<schema>
  <complexType name=„GewichtsDaten“>
    <attribute name=„Klasse“ type=„string“>
  </complexType>

  <element name=„Boxer“>
    <element name=„Name“ type=„string“/>
    <element name=„Vorname“ type=„string“ minOccur=„1“ maxOccur=„unbounded“/>
    <element name=„Gewicht“ type=„GewichtsDaten“ minOccur=„0“ maxOccur=„1“/>
  </element>
</schema>
```

### 4.4.1 Bewertung

Die Vorteile von XSDs gegenüber DTDs sind schwerwiegend. Allein die Möglichkeit benutzerdefinierter *Typen* ermöglicht eine vollständige Beschreibung von Daten. Wie die Analyse der Anforderungen ergab, ist ein wesentlicher Baustein des zu erstellenden Systems verantwortlich für die Bereitstellung definitorischer Mittel für XML-Dokumente, dies insbesondere bezüglich der *Konnektion*. Die erste Problemstellung wäre grundsätzlich allein mit XSDs lösbar. Die zweite bedarf jedoch des tieferen Verständnisses des Konzeptes der *Konnektion*. Dieses soll in Kapitel 5 vermittelt werden.

## 4.5 SLIDL SLangItem Definition Language

Ein Hauptaspekt dieser Arbeit ist die Entwicklung einer XML-Beschreibungssprache, die der *Konnektion* mächtig ist[11], sowie einer API zu dieser Sprache. Diese *konnektionsmächtige* Sprache heiße SLIDL. Obgleich SLIDL Teile von XSDs scheinbar *neu erfindet* ist dies unwesentlich im Rahmen bisheriger und auch späterer Überlegungen. Deren Grad an Allgemeinheit ist grundsätzlich so gewählt, dass generelle Strukturen sichtbar werden, welche sich sowohl in SLIDL, als auch in XSDs manifestieren. Das bisherige Beispiel in SLIDL:

[11] ferner auch *konnektionsmächtige* Beschreibungssprache genannt

```
<SLIDLDocument>
  <Class name=„GewichtsDaten“>
    <Attribute name=„Klasse“ btype=„STRING“>
  </Class>

  <Class name=„Boxer“>
    <Element name=„Name“ btype=„STRING“/>
    <Element name=„Vorname“ btype=„STRING“ cntmin=„1“ cntmax=„[$MAXINT]“/>
    <Element name=„Gewicht“ bclass=„GewichtsDaten“ cntmin=„0“ cntmax=„1“/>
  </Class>
</SLIDLDocument>
```

Dieses Beispiel offenbart nur den Teil von SLIDL, der die *Definition* von XML-Strukturen ermöglicht, und ähnelt *struktrell*, in markanter Weise, seinem XSD Gegenstück. Offensichtlich ist SLIDL mehr am Paradigma des *objekt-orientierten* Programmierens angelehnt. Kapitel 5 vertieft Aspekte der grundsätzlichen *syntaktischen* und *semantischen* Struktur von XML, sowie Aspekte der *Konnektion*, welche eine solide Basis des Verständnisses von Kapitel 7 bilden sollen. Dort wird die Phase des Entwurfs und der Implementierung einer API zu SLIDL dargestellt.

# Kapitel 5

# Analyse und Entwurf

In Kapitel 4 wurden die Grundlagen der semantischen Hierarchisierung von Daten eingeführt. Die Spezifizierung der einzelnen, vom *Auftraggeber*, erwünschten Anforderungen an das zu erstellende System ist im Kapitel 3 aufgeschlüsselt worden. Es stellte die Phase der *Anforderungsanalyse* dar; die erste und mitunter wichtigste Phase, wie unter Teilabschnitt 2.6.1 im Block 1a bereits vorab dargestellt. Die Resultate dieser Phase des Softwareentwicklungsprozesses sind maßgeblich für die Struktur und Ergebnisse, der ihr folgenden Phasen.
In diesem Kapitel werden die Anforderungen des deklarativen Ansatzes (vgl. Abschnitt 3.3) grob auf die benötigten Ressourcen zur Lösung des gestellten Problems abgebildet[1]; im Sinne der *Systemanalyse* (vgl. Teilabschnitt 2.6.1 1b). Gründend auf der *Systemanalyse* wird schrittweise eine *Softwarearchitektur* ausgearbeitet.

## 5.1 Abstraktion von Information

Kapitel 4 vertiefte die Sicht auf Daten als hierarchisierbare Objekte. Die Abschnitte über die Möglichkeiten der Definition der Semantik[2] von Daten konnten darlegen, dass die Abstraktion von *Daten* zu *Datenobjekten* sich aus der Struktur von XML selbst ergibt. Diese fußt wiederum auf dem Wesen von *Daten*, als *Objekte* im Sinne des *allgemeinen Objektmodells*[3], über *verknüpfbare Eigenschaften* zur Bildung von Ordnungsstrukturen zu ver-

[1] SLIDL ist eine Instanz *aller* Lösungen
[2] Abschnitt 4.3: DTD, Abschnitt 4.4: XSD, Abschnitt 4.5: SLIDL
[3] siehe Abschnitt 2.3

fügen. Die Hierarchisierung als *Fokussierung* auf bestimmte Eigenschaftstypen erleichtert den Vorgang der Kategoriesierung von Datenklassen, stellt aber keine *notwendige* Erscheinung, sondern eine gängige *Konvention*[4] dar.
Welche *Struktur* ergibt die Betrachtung einer *Szenedatei* des DRAS ? Welche Anforderungen ergeben sich dadurch an die definitorischen Fähigkeiten SLIDLs ? Welche für dessen APIs ?

Diese Fragen sollen innerhalb des folgenden Abschnittes beantwortet werden. Dies wird unter zwei Gesichtspunkten geschehen. Einerseits hinsichtlich rein XML–abhängiger Eigenschaften. Andererseits in Hinblick auf den rein von einer DRAS–*Szenedatei* implizierten Eigenschaften.

## 5.2 Anforderungen an eine Sprach–Definition

Trotz der Allgemeinheit des Titels, sowie der Allgemeinheit der Definitionen der folgenden Teilabschnitte, stehen im Mittelpunkt des Interesses, all diejenigen Anforderungen, die speziell in SLIDLs konzeptionelle Grundlagen einzufließen haben. Es wird sich zeigen, dass jede explizit definierte Sprache wie SLIDL, lediglich *eine* mögliche Ausprägung allgemeiner Prinzipien ist, die von monohierarchisch gegliederten *Datenstrukturen grundsätzlich* vorgegeben werden.

### 5.2.1 Objektorientierte Perspektive

Die Konzeption und Architektur des DRAS sind *objektorientiert*. Ebenso die allgemeine Auffassung von Daten und die daraus ableitbare XML Definition. Eine *Szenedatei* ist eine Instanz eines *XML–Dokumentes*. Schon die Betrachtung von XSDs (siehe Abschnitt 4.4) zeigte deutlich, welche Vorteile die Definition von XML–Dokument–Klassen in einer *objektorientierten Sicht* auf Daten mit sich führt, gegenüber einer *weniger* am Programmierparadigma der Objektorientierung angelehnten Sicht (z.B. aus dem Blickwinkel einer DTD (vgl. Abschnitt 4.3)).
Diese positive Eigenschaft der XSDs teilt auch SLIDL. Zunächst seien XML–Datenobjekte differenziert, anhand grundsätzlich unterscheidbarer syntaktischer Merkmale. In einem

[4]gemeint: eine hierarchische Ordnung und ein *semantisches* Netz, sind unterscheidbar im *Zweck* und der *Motivation* ihrer Entstehung. Die Frage nach der *richtigeren* Ordnung ist also hinfällig.

zweiten Schritt werden diese abgebildet auf eine Container–Perspektive. Abschließend wird die neue Perspektive auf SLIDL abgebildet:

1. Direkt aus XML ableitbare Syntaxstrukturen (ferner auch als *syntax–strukturelle* XML–Objekte bezeichnet):
    (a) Mehrzeilige Elemente
    (b) Einzeilige Elemente
    (c) Attribute von Elementen
2. Durch Fokussierung auf die Eigenschaft des *Enthaltens von Unterelementen* ergibt sich die Container–Perspektive auf Strukturen in XML–Syntax:
    (a) Container Elemente
    (b) Elemente
    (c) Attribute von Elementen
3. Man verstärkt diese Sicht, indem man annimmt, dass das Enthalten von *syntax–strukturellen* Unterobjekten allgemein, also auch das Enthalten von Attributen ein Container–Element ausmacht. Dies entspricht der Sichtweise von SLIDL:
    (a) Klassen (Container–Elemente)
    (b) Elemente von Klassen
    (c) Attribute von Klassen

### 5.2.2 Container und Elementarobjekte

Die Fokussierung auf die Eigenschaft des Nicht-Enthaltens *syntax–struktureller* Unterobjekte, eröffnet die Möglichkeit einer weiteren Ableitung; der Zusammenfassung von *Elementen* und *Attributen* zu *elementaren Objekten* (oder *Elementarobjekten*). *Klassen* seien dementsprechend *Container–Objekte* (oder *Container*).

### 5.2.3 Container-Typen, elementare Datentypen und Basistypen

Die im letzten Abschnitt genannte Eigenschaft ist aus Sicht der gespeicherten Daten auch interpretierbar, als die Eigenschaft des Enthaltens der tatsächlichen (faktischen) Information. Es muss dem Leser deutlich vor Augen sein, dass im Rahmen von XML *rein strukturgebende* von *rein datenenthaltenden* Objekten unterschieden werden können. Dieser Umstand geht einher mit der *strukturellen* Unterscheidung des letzten Abschnittes.

In diesem Sinn und unter dem Aspekt, dass die Struktur eines *bestimmten* aber *beliebigen* Containers, als die von ihm enthaltene Information angesehen werden kann, entspricht die *Struktur* eines Containers dessen *Datentyp*. Ein *struktureller Datentyp* heißt auch *Strukturtyp*.

Es offenbart sich die (gerade dargelegte) grundsätzliche Unterscheidbarkeit der Auffassung von:

1. Containern, als Repräsentanten für Träger *struktureller Datentypen*
2. Elementarobjekten, als Repräsentanten für Träger *faktischer Datentypen*

Ein *struktureller Typ* werde ferner auch aufgefasst, als *Container–Typ*. Entsprechend seien *faktische Datentypen* ferner auch aufgefasst, als *elementare Datenypen* (oder *Elementartypen*). In diesem Zusammenhang ist grundsätzlich zu unterscheiden zwischen:

1. skalaren Inhalten eines *elementaren Typs*
2. vektoriellen Inhalten eines *elementaren Typs*

#### 5.2.3.1 Folgerungen für SLIDL und einer API zu SLIDL

Eine vordefinierte Menge elementarer Typen, die Menge der *Basistypen* ist von Seiten SLIDLs zu stellen. Eine mögliche API zur Verwendung SLIDLs benötigt eine Schnittstelle zu einer API von XML. Diese muss das typsichere Lesen und Schreiben von C++-internen Datentypen, skalarer und vektorieller Art implementieren. Prototypisch ist die Beschränkung der Implementierung dieser Schnittstelle auf *eine* **STL** (**S**tandard **T**emplate **L**ibrary)–Container-Klasse[5] akzeptabel.

### 5.2.4 Abgeleitete elementare Typen

Um alle Möglichkeiten von Erscheinungsformen typisiert einzuordnender Daten abdecken zu können, muss SLIDL die Möglichkeit zur Verfügung stellen, neue Typen auf Basis bereits existierender Typen definierbar zu machen. Innerhalb von SLIDL wird streng unterschieden zwischen *Container–Objekten* und *elementaren Objekten* einerseits, sowie *Container–Typen* und *elementaren Typen* andererseits. Folglich unterscheidet SLIDL auch zwischen den Ableitungen dieser beiden Kategorien von Typen.

[5] Die vorliegende Arbeit beschränkt sich zunächst auf die Klasse **std::vector**

Hier geht es um die Erschaffung neuer elementarer Typen. Dies muss sowohl bezüglich der von SLIDL gestellten *Basistypen*, als auch hinsichtlich derer elementaren Ableitungen ermöglicht werden. Ferner sind die folgenden Optionen für abgeleitete Elementartypen zu implementieren:

1. die Einschränkung des Wertebereichs bei numerischen Basistypen
2. die Angabe erweiterter regulärer Ausdrücke zur *manuellen* Typprüfung
3. die Definition von Auswahlmengen, die wie folgt unterscheidbar sind:
   (a) nur Mehrfachauswahl
   (b) nur Einfachauswahl
4. geeignete Wege eine getroffene Auswahl mit vordefinierten Inhalten zu ersetzen

### 5.2.5 Klassen und deren Ableitungen

Im letzten Teilabschnitt sind die wesentlichen erwünschten Möglichkeiten bei der Erzeugung benutzerdefinierter Elementartypen aufgeschlüsselt worden. Es folgt analog dazu, eine Aufzählung an erwünschten Möglichkeiten, bei der Erzeugung neuer Containertypen:

1. die Mehrfach–Ableitung einer neuen Klasse aus den restlichen definierten Klassen
2. die Deaktivierung von geerbten Elementen und Attributen
3. die Möglichkeit geerbte Elemente und Attribute im Zugriff *von außen* zu beschränken
4. die Möglichkeit die Inhalte geerbter Elemente und Attribute initial festzulegen

Die letzten Abschnitte waren bemüht auf der allgemeinsten DRAS–unabhängigen Ebene, auf der Ebene von XML, die Anforderungen an die Fähigkeiten einer SLIDL–Umgebung [6] zusammen zu tragen. Stillschweigend sind jedoch bereits Annahmen, die direkt aus der Motivation einer GUI–basierten Konfigurations–Schnittstelle des DRAS[7] resultieren, in die eine oder andere konzeptionelle Forderung eingegangen. Welche dies sind, soll in späteren Abschnitten beschrieben werden.

[6] also der Definition von SLIDL, und der dazu passenden API (vgl. Abschnitt 4.2)
[7] speziell die Punkte 3a, 3b und 4 Teilabschnitt 5.2.4

## 5.3 Abstraktion von Konnektion

### 5.3.1 Konnektionsobjekt

Eine spezielle Eigenschaft des DRAS wurde bereits in Teilabschnitt 3.2.2 vorgestellt. Es handelt sich um das Konzept der *Bindung* zwischen Instanzen der Klassen *Option* und *StateValue* bzw. dem Wert einer Instanz der Klasse *Event*. Völlig ungeachtet dessen[8], was DRAS mit einer *Konnektion* funktional verbindet: Das Ergebnis dieser *Operation* schlägt sich konfigurativ nieder in einem verschachtelten XML–Objekt, welches grundsätzlich an jeder beliebigen Stelle eines XML–Dokumentes erzeugt werden kann. In der Denkweise SLIDLs ist die Rede von einem *Konnektionsobjekt*: einem *Container*–Objekt, innerhalb eines zunächst beliebigen anderen *Containers*.

### 5.3.2 Konnektionsoperanden I

Die elementaren Unterobjekte eines *Konnektionsobjekt*es werden dabei *parametrisiert* mit Werten, welche abhängig von zwei zu *bindenden* DRAS–Objekten zu bestimmen sind. SLIDL muss also um das Konzept der *Konnektionsoperanden* erweitert werden. Ein *Konnektionsoperand* ist das *generalisierte* SLIDL–Analogon einer *Option*, eines *StateValue* oder des Wertes eines *Events* des DRAS. Genauer: Eine Definition der DRAS–Konfiguration in SLIDL ordnet jedem *bindbaren* DRAS–Objekt[9] genau einen *Konnektionsoperand* zu. Zusammenfassend ist also festzustellen, dass die Konnektion zweier *Konnektionsoperanden*[10] die Erzeugung eines *Konnektionsobjektes* anstößt.

#### 5.3.2.1 Folgerungen für SLIDL und einer API zu SLIDL

Es kristallisiert sich heraus, dass SLIDL ein Konzept zur Definition von

1. **Konnektionsoperanden**
2. parametrisierten Klassen von **Konnektionsobjekten**

bereit stellen muss. Innerhalb der API von SLIDL muss dieses Konzept einer später aufsetzenden GUI zugänglich gemacht werden.

[8] Es leuchtet schnell ein: Für die Konzeption einer Metasprache, welche XML Semantiken und das Konzept der Konnektion verbindet, ist es irrelevant, wie in ihr definierte Semantiken eingesetzt werden.

[9] z.B. einer *Option* oder einem *StateValue*

[10] also auch einer *Bindung* des DRAS

### 5.3.3 Konnektionsoperanden II

Wie sind die *Konnektionsoperanden* genau zu verstehen ? Für die Beantwortung dieser Frage, ist die folgende Überlegung zu treffen:

1. Die DRAS–Objekte **Option**, **EventData** und **StateValue** sind *bindbar*.

2. Einem *bindbaren* DRAS–Objekt kann aus der Sicht SLIDLs, genau ein *Konnektionsobjekt* zugeordnet werden.

3. Innerhalb einer Instanz einer, in SLIDL-formulierten, DRAS-Konfiguration existieren im Falle einer *Konnektion* drei Container: ein *Konnektionsobjekt* und zwei weitere Container, die *ableitbar* sind aus den *Konnektionsoperanden*.

4. Das Konzept des *Bindens bindbarer* DRAS–Objekte, ist damit abbildbar auf das Konzept der *Konnektion* zweier *Konnektionsoperanden*.

Anhand der gerade vollzogenen Überlegung, ist leicht ersichtlich, dass ein *Konnektionsoperand* korrespondiert zu einem *bindbaren* DRAS–Objekt. Die Auffassung der *Konnektion* als Operation hilft zu verdeutlichen, dass die *Bindung* zweier *bindbarer* DRAS–Objekte durch mindestens *eine* XML–Struktur (in der DRAS–Szenedatei) repräsentiert wird.
Der 2 ist nicht zu verwechseln mit der Möglichkeit einem *StateValue* mehrere *Options* zuordnen zu können. Trotz dieser Möglichkeit benötigt jede *Option*, sowie jeder *StateValue* genau einen Operanden. Die mehrfache Bindung manifestiert sich in der Existenz mehrerer *Konnektionsobjekte* bezüglich eines *StateValues*.

### 5.3.4 Konnektionsoperanden III

Es ist klar geworden, dass in einem SLIDL–Dokument, ein *Konnektionsoperand* grundsätzlich mit einem Container assoziiert wird. Es ist jedoch nicht ausreichend klar, ob ein *Konnektionsoperand* abhängig ist von Elementarobjekten oder Einträgen aus Dokumenten, die außerhalb des SLIDL–Sprachraums liegen. Es können drei wesentliche Typen von *Konnektionsoperanden* unterschieden und wie folgt zusammengefasst werden:

1. **interne Konnektionsoperanden**

   (a) mit Elementarobjekt–Korrespondenz

   (b) ohne Elementarobjekt–Korrespondenz

2. **externe Konnektionsoperanden**, mit Korrespondenz zu Entitäten aus Nicht-SLIDL-Dokumenten

#### 5.3.4.1 Folgerungen für SLIDL und einer API zu SLIDL

SLIDL muß Möglichkeiten zur Definition der drei im letzten Abschnitt benannten Typen von *Konnektionsoperanden* stellen. Ferner muss es hinblicklich *externer Operanden* die Angabe eines beliebigen Konverters unterstützen, der unabhängig von SLIDL ist, und ein Nicht-SLIDL-Dokument in ein Dokument übersetzt, welches ausschließlich *Konnektionsoperanden* enthält. Die API von SLIDL soll diese Angabe beim Laden einer Instanz eines Dokuments einer SLIDL-definierten Sprache auswerten, den Konverter mit dem Nicht-SLIDL-Dokument aufrufen, und die resultierende Datei auf *Konnektionsoperanden* parsen.

#### 5.3.4.2 Beschaffenheit des Konverters

Die im letzten Abschnitt eingeführte Idee eines Konverters soll näher erläutert werden: Der Konverter ist gedacht, als eigenes Programm oder Skript, welches absolut unabhängig von SLIDL oder dessen API ist. Dies gewährleistet die höchtmögliche Flexibilität bezüglich der Inklusion *externer Konnektionsoperanden*, die aus Dateien unbekannten Typs stammen. Es wäre falsch das Quell-Dateiformat des Konverters festzulegen, da es z.B. denkbar ist, dass es sich bei diesen Dateien z.B. um Bilder, Sound etc. handelt, und die Funktion des *Konverters* darin besteht *Merkmale* zu extrahieren und *Konnektionsoperanden* daraus zu bilden.

### 5.3.5 Konnektion

Die Konnektions–Schnittstelle eines Containers repräsentiert das Analogon zum Konzept der Klasse *OptionContainer*, jedoch bezogen auf SLIDL–*Klassen*. Sie ist folglich, als der Behälter der nach außen definierten Konnektionsoperanden einer Klasse zu sehen. Daraus abgeleitet soll eine *Erzeugungsregel* verstanden werden, als eine *Abbildung*[11] über zwei *Konnektionsoperanden* auf ein *konfiguriertes* Konnektions–Objekt.
Zum Verständnis des Begriffes *Konfiguration*, sei ein *unkonfiguriertes* parametrisiertes

[11] im mathematischen Sinn

Konnektionsobjekt $K_{\mathsf{U}}$, ein Container in Abhängigkeit von $n > 0$ Platzhaltern[12] $p_1,\ldots,p_n$, welche in Abhängigkeit von zwei Operanden $O_{\mathsf{SRC}}$ und $O_{\mathsf{DST}}$ substituiert werden müssen, um das *konfigurierte* Konnektionsobjekt $K_{\mathsf{K}}$ zu erzeugen. Es gilt also:

$$K_{\mathsf{K}} \quad = \quad K_{\mathsf{U}}(p_1(O_{\mathsf{SRC}}, O_{\mathsf{DST}}), \ldots, p_n(O_{\mathsf{SRC}}, O_{\mathsf{DST}}))$$

In Kapitel 7 soll dargelegt werden, welche speziellen Eigenschaften den Konnektionsoperanden und dem Konfigurationsobjekt, innerhalb der SLIDL Definition zuzuordnen sind, um aus ihnen Parameter für unkonfigurierte Konnektionsobjekte zu erzeugen.

### 5.3.6 Inverse Konnektion

Die Bezeichnung *inverse Konnektion* ist nicht mit der Aufhebung einer Konnektion zu verwechseln. Sie beschreibt die Prüfung der Existenz zweier Operanden in Bezug zu einem gegebenen Konnektionsobjekt. Die Konnektion beschreibt den Vorgang, der bei der Bearbeitung einer DRAS –Szenedatei hinblicklich des folgenden Speichervorgangs der Datei von Bedeutung ist. Dagegen repräsentiert die *inverse Konnektion* den Prüfvorgang, während des Ladens einer Szenedatei, hinsichtlich der Feststellung ihrer syntaktischen und semantischen Korrektheit. Hierbei gilt:

$$K_{\mathsf{K}}^{-1} \quad = \quad \begin{cases} wahr & \Leftrightarrow \quad \exists O_{\mathsf{SRC}}(K_{\mathsf{K}}) \,\wedge\, \exists O_{\mathsf{DST}}(K_{\mathsf{K}}) \\ falsch & \Leftrightarrow \quad \nexists O_{\mathsf{SRC}}(K_{\mathsf{K}}) \,\vee\, \nexists O_{\mathsf{DST}}(K_{\mathsf{K}}) \end{cases}$$

## 5.4 Über persistente Objekte

Das Problem des Fehlens eines einheitlichen Konzeptes zum Laden und Speichern einer DRAS–Konfiguration wurde bereits angesprochen. Das Ziel dieses Abschnittes soll es sein ein XML–basiertes objektorientiertes Konzept der *Persistenz*[13] zu definieren. In Teilabschnitt 4.2.1 wurden die beiden gängigen APIs zur Verarbeitung von XML vorgestellt: SAX und DOM. Das DRAS orientiert sich aufgrund seiner Konzeption am DOM, unter Verwendung der Bibliothek *Xerces*, die eine „*XML-Datei in eine DOM-Struktur überträgt*“

[12] ähnlich den C++–Templates

[13] Ein *persistentes* Objekt hat die Fähigkeit seine Daten über das Ende eines Programmes hinaus zu *sichern* (vgl. (Eck00, Kap. 6)). Allgemein etwas mit dauerhafter Beschaffenheit, Beharrlichkeit oder Ausdauer.

(Rau05, S. 41–42).[14]

### 5.4.1 C++ und Xerces

Der Einsatz von Xerces in C++ gestaltet sich problematisch, da die Allokation und Freigabe des Speichers von extrahierten Zeichenketten dem Verwender der Bibliothek überlassen bleibt. Das Problem: Bezüglich XML sind Zeichenketten die hauptsächlichen Daten der Begierde. Ferner ist Xerces so konzipiert, dass dessen Schnittstelle grundsätzlich keine C++-Standard-Strings kennt. Dieser Umstand veranlasste bereits den Entwickler des DRAS dazu, *template*-basierte *Wrapper*-Funktionen[15] zur Verwendung dieser Bibliothek zu definieren. Diese sind so gebaut, dass sie die Inhalte von Attributen und Elementen, abhängig von einem erwarteten Typ, liefern. Der Overhead[16] der direkten Nutzung von Xerces und der Datentyp-Prüfung erhaltener Inhalte innerhalb der Implementierung der eigentlichen Klassen des DRAS konnte so drastisch reduziert werden.

### 5.4.2 XML Persistenz

Das DOM eignet sich sehr für den schnellen und unkomplizierten Einsatz. Das bedeutet: Es ist für die Entwicklung eines Prototypen einer generalisierten Persistenz–Klasse zunächst das Mittel der Wahl[17]. Aufbauend auf der Funktionalität einer erweiterten Form von XMLTools[18], soll eine Persistenzklasse definiert werden. Ableitungen dieser Klasse erben dabei Schreib- und Lesefunktionalität auf DOM-Knoten. Sie müssen lediglich die *Registrierung* ihrer Memberdaten abhängig vom jeweiligen Datentyp und einer XML-Struktur implementieren.

## 5.5 Die Softwarearchitektur

Dieser Abschnitt widmet sich der Umsetzung aller hier formulierten Anforderungen an eine SLIDL-API zu einer Softwarearchitektur. Die explizite Definition von SLIDL selbst

[14] Xerces implementiert weitaus mehr, als die schlichte Überführung von XML–Dokumenten in die DOM–Struktur. Im Rahmen des DRAS wurde jedoch nur diese Funktionalität genutzt.

[15] gemeint: eine *statische* Klasse namens XMLTools

[16] gemeint: *Overhead* bezüglich der Anzahl nötiger Zeilen

[17] Diese ließe sich zur Optimierung der Speichernutzung und der Ausführungsgeschwindigkeit sicherlich auch mit SAX umsetzen.

[18] siehe Fußnote 15

erfolgt unter Teilabschnitt 7.1.1. Ferner wird hier der Bezug der Anforderungen an eine graphische Schnittstelle aus Kapitel 3 zu SLIDL herausgearbeitet werden.

### 5.5.1 Verbund von Schichten

Die Architektur des Gesamtsystems ist in mehrere Schichten gegliedert. Wie auf Abbildung 5.1 dargestellt ergeben sich nach der Anforderungsanalyse die folgenden drei Schichten:

1. **XML-Persistenz-Schicht:** Diese Schicht implementiert eine Schnittstelle zu *Xerces* zur Gewährleistung der Prüfbarkeit der *Wohlgeformtheit* von XML-Dokumenten. Darauf fußend implementiert sie die Möglichkeit der *Registrierung* von Memberdaten bezüglich Datenypen und beliebigen XML-Strukturen. Die *Gültigkeit* eines Dokumentes wird nicht eigens geprüft, sondern ergibt sich implizit durch die Registrierung. Eine Klasse, die diesen *Layer* verwendet, trägt selbst Sorge für die Korrektheit ihrer *Standardinitialisierung*. Auf diese Weise ist die Gültigkeitsprüfung vernachlässigbar, da im Falle der Ungültigkeit des XML-Dokumentes die initialen Einstellungen einer Klasse zum Tragen kommen. Der korrekte Ablauf eines Programmes bleibt dadurch erhalten.

2. **SLIDL-Definitions-Schicht:** Diese Schicht ist verantwortlich für das Einlesen von SLIDL-Dokumenten. Sie ist ferner zuständig für den Aufbau einer internen Repräsentation eines Dokumentes. Eine besondere Aufgabe dieser Schicht ist die Prüfung der Richtigkeit aller *Typ*- und *Klassenabhängigkeiten* eines SLIDL-Dokumentes.

3. **SLIDL-Instanz-Schicht:** Diese Schicht entpricht der Schnittstelle SLIDLs für ein potientiell aufsetzendes GUI. Jegliche Instanz einer SLIDL-Klasse, sowie die Abhängigkeiten und Zusammenhänge zwischen allen Instanzen von SLIDL-Klassen werden von dieser Schicht verwaltet. Diese Schicht kommuniziert mit der SLIDL-Definitions-Schicht. Ferner ist sie zuständig für das Einlesen von Instanzen von SLIDL-Dokumenten, sowie der Repräsentation derselben.

4. **GUI-Schicht:** Diese Schicht ist völlig entkoppelt von allen Definitions- und Instanzabhängigkeiten einer SLIDL-Definition. Ihr obliegt lediglich die Verantwortung der Darstellung von SLIDL-Instanzen, deren Abhängigkeiten und der Manipulation derselben. Sie kommuniziert nur mit der SLIDL-Instanz-Schicht.

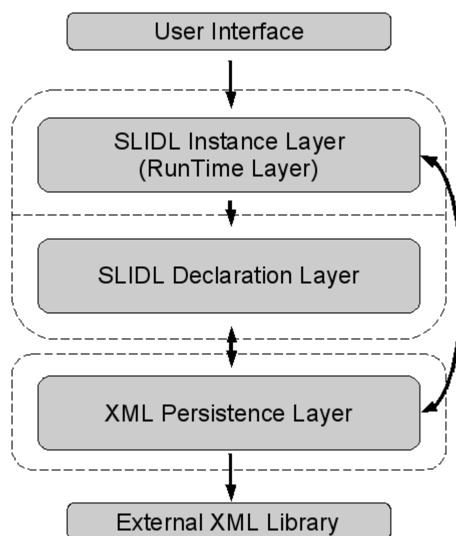


Abbildung 5.1: Die grobe Architektur

### 5.5.2 Die Persistenz Schicht

Die Persistenzschicht ist folgendermaßen gegliedert:

1. eine erweiterte Schnittstelle zur DOM-Implementierung von *Xerces*: *XMLTools*[19]
2. eine Schnittstelle zur Dokument und Speicherverwaltung von *Xerces*: *XMLSysEnvironment*[20]
3. ein Wrapper zur komfortablen Verwendung von *Xerces*-Zeichenketten: *XMLStr*[21]
4. mehrere typisierte Registrierungsklassen in Abhängigkeit von möglichen XML-Strukturen (siehe Unterabschnitt 5.5.2.1)
5. eine vollständige Registrierungsklasse *XMLRegistry*, die alle typisierten *Registrierungsklassen* zur Verfügung stellt
6. eine abstrakte Klasse zur Erzeugung beliebiger C++–Objekte: *XMLObjectGenerator*

[19] siehe UML-Diagramm: Abbildung 5.4
[20] siehe UML-Diagramm: Abbildung 5.2
[21] siehe UML-Diagramm: Abbildung 5.2

7. einer Klasse *XMLReadWriteable*, abgeleitet von *XMLRegistry* und *XMLObjectGenerator*, welche die eigentliche Infrastruktur der Persistenz implementiert, und die folgenden *virtuellen* Methoden bereitstellt:

   (a) **Lesen:** *preRead()*, *postRead()* und *read()*

   (b) **Schreiben:** *preWrite()*, *postWrite()* und *write()*

   (c) **Objekt-Erzeugung:** *buildObject(string name)*

   Dabei implementieren *read()* und *write()* die Auswertung der Registrierungseinträge der *XMLRegistry*. Diese wird danach umgesetzt in Form von Lese- und Schreibzugriffen über die Erweiterung von *XMLTools*. Die *pre*()*-Methoden werden, abhängig von der Art der Zugriffsoperation, als erste aufgerufen. Innerhalb der Oberklasse ist *preRead() leer* implementiert, während *preWrite()* einfach *preRead()* aufruft. Vorbereitungen des *Zugriffs* sollen in einer beliebigen Ableitung von *XMLReadWriteable* hier erfolgen.
   Entsprechend sind die *post*()*-Methoden für abschließende Arbeiten nach dem *Zugriff* auf DOM-Daten gedacht. Deren Implementierung in *XMLReadWriteable* ist vorhanden, aber leer. Die Implementierung der *pre*()* und *post*()*-Methoden, in der genannten Art und Weise, ermöglicht die komfortable Ableitung von *XMLReadWriteable* ohne jede dieser Methoden innerhalb einer Unterklasse überschreiben zu müssen. Erst im Zusammenhang mit Mehrfachableitungen wird dieser Umstand problematisch, da die *Kreuzdelegation*[22] zum Tragen kommt.

8. einem Präprozessor-Template, welches als Wrapper zu den eigentlichen Registrierungsfunktionen von *XMLReadWriteable* fungiert. Dieser Wrapper erwies sich als hilfreich, bei der Umstellung der Persistenz-Architektur, da keine der Ableitung von *XMLReadWriteable* abgeändert werden musste. Darum ist seine Verwendung auch in Hinblick auf zukünftige Änderungen dieses Layers durchaus berechtigt.

#### 5.5.2.1 Syntaktische XML-Strukturen

Die *erweiterte* Fassung der Klasse *XMLTools* stellt, abhängig von einem gegebenen DOM-Knoten oder Element, Funktionen zum Lesen *und* Schreiben von Inhalten der folgenden XML Strukturen zur Verfügung:

[22] siehe Teilabschnitt 6.2.3

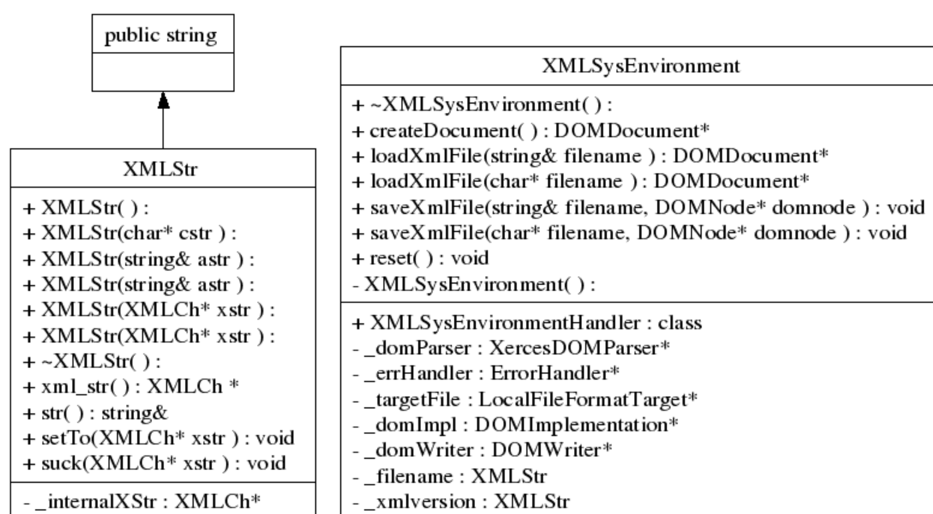


Abbildung 5.2: UML Diagramm: **XMLStr** und **XMLSysEnvironment**

1. Attributen
2. Attributen von Unterelementen
3. Elementen
4. Unterelementen

Auf Abbildung 5.4 ist das UML-Diagramm zu XMLTools dargestellt. Den wesentlichen Kern dieser *statischen* Klasse bilden folgende Funktionen:

1. *getAttribute(DOMElement* . . . )* und *setAttribute(DOMElement* . . . )*
2. *getElementContent(DOMNode* . . . )* und *setElementContent(DOMNode* . . . )*

Diese Funktionen existieren jeweils zwei Mal. Eine Variante bezieht sich auf Inhalte des Typs $T$, während die andere Variante sich auf Inhalte des Typs $vector < T >$ bezieht. Ein Großteil der übrigen Funktionen baut schließlich nur auf diesen *acht* elementaren Funktionen auf.

#### 5.5.2.2 Besonderheit: Bool, XBool und InvXBool

Grundsätzlich sind *Elementarobjekte* mit boolschem Typ denkbar. Innerhalb einer DRAS-Konfiguration existieren jedoch zusätzlich *Elementarobjekte*, genauer XML-Elemente, deren Existenz allein als boolsches *wahr* interpretiert wird. Aus dieser Motivation heraus

enthält die Persistenz-Schicht zwei Konzepte zur Handhabung und Registrierung boolscher Elementarobjekte:

1. Die Klasse *XMLBooleanStrings* übersetzt Zeichenketten nach *Bool* und umgekehrt. *False* ergibt sich hierbei aus den Zeichenketten *false*, *no* und *0*. Die Zeichenketten *true*, *yes* und *1* werden demnach mit *true* assoziiert.
2. Es existieren zwei weitere Registrierungsklassen-Instanzen für boolsche XML-Elemente innerhalb der *XMLRegistry*[23]: *RegWithSubElementsXBool* und *RegWithSubElementsInvXBool*.

Die beiden Registrierungsobjekte *XBool* und *InvXBool* sind dabei ausschließlich für XML-Elemente zu verwenden. Ersteres liefert bei Existenz eines XML-Elements *wahr*. Das zweite verhält sich *invers* dazu.

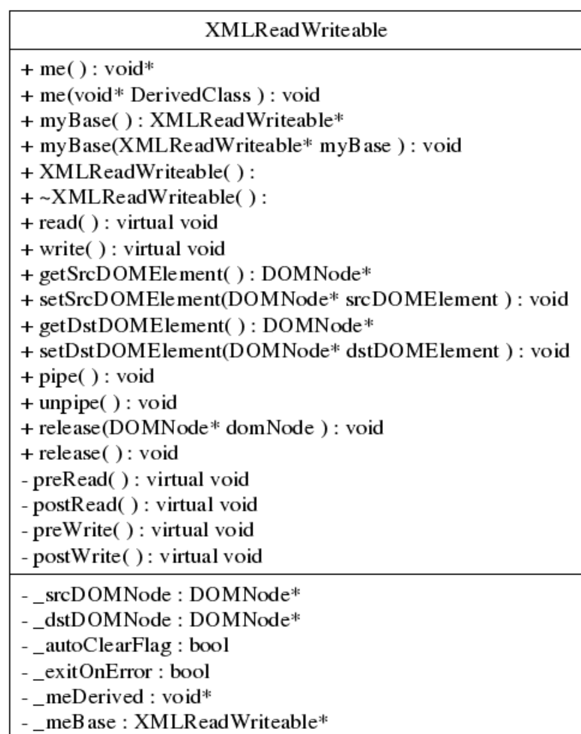


Abbildung 5.3: UML Diagramm: **XMLReadWriteable**

[23] siehe Abbildung 5.5

| XMLTools |
| --- |
| + hasElement(DOMElement* element, char* name ) : static bool |
| + hasElement(DOMNode* node, char* name ) : static bool |
| + parseElementToString(DOMElement* element, char* name, string& target ) : static bool |
| + parseElementToString(DOMNode* node, char* name, string& target ) : static bool |
| + getContentToString(DOMElement* element, string& target ) : static bool |
| + getContentToString(DOMNode* node, string& target ) : static bool |
| + getElement(DOMElement* element, string& name ) : static DOMElement* |
| + getElement(DOMElement* element, char* name ) : static DOMElement* |
| + getElement(DOMNode* node, string& name ) : static DOMElement* |
| + getElement(DOMNode* node, char* name ) : static DOMElement* |
| + getNode(DOMNode* node, DOMNode* findnode ) : static DOMNode* |
| + setSubElementFromString(DOMElement* element, char* name, string& anyString ) : static void |
| + setSubElementFromString(DOMNode* node, char* name, string& anyString ) : static void |
| + createElement(DOMElement* element, char* name ) : static DOMElement* |
| + createElement(DOMNode* node, char* name ) : static DOMElement* |
| + getAttribute(DOMElement* element, char* name, bool& val ) : static bool |
| + parseElement(DOMElement* element, char* name, bool& val ) : static bool |
| + getAttribute(DOMNode* node, char* name, bool& val ) : static bool |
| + parseElement(DOMNode* node, char* name, bool& val ) : static bool |
| + setAttribute(DOMElement* element, char* name, bool& val ) : static void |
| + setAttribute(DOMNode* node, char* name, bool& val ) : static void |
| + setElementContent(DOMElement* element, bool& val ) : static void |
| + setElementContent(DOMNode* node, bool& val ) : static void |
| + getAttribute(DOMElement* element, char* name, T& val ) : static bool |
| + getAttribute(DOMNode* node, char* name, T& val ) : static bool |
| + getAttribute(DOMElement* element, char* name, TVector& val ) : static bool |
| + getAttribute(DOMNode* node, char* name, TVector& val ) : static bool |
| + getElementAttribute(DOMElement* element, char* elemName, char* attrName, T& val ) : static bool |
| + getElementAttribute(DOMNode* node, char* elemName, char* attrName, T& val ) : static bool |
| + getElementContent(DOMElement* element, T& val ) : static bool |
| + getElementContent(DOMNode* node, T& val ) : static bool |
| + getElementContent(DOMElement* element, TVector& val ) : static bool |
| + getElementContent(DOMNode* node, TVector& val ) : static bool |
| + parseElement(DOMElement* element, char* name, T& val ) : static bool |
| + parseElement(DOMNode* node, char* name, T& val ) : static bool |
| + setAttribute(DOMElement* element, char* name, T& val ) : static void |
| + setAttribute(DOMNode* node, char* name, T& val ) : static void |
| + setAttribute(DOMElement* element, char* name, TVector& val ) : static void |
| + setAttribute(DOMNode* node, char* name, TVector& val ) : static void |
| + setElementAttribute(DOMElement* element, char* elemName, char* attrName, T& val ) : static void |
| + setElementAttribute(DOMNode* node, char* elemName, char* attrName, T& val ) : static void |
| + setElementContent(DOMElement* element, T& val ) : static void |
| + setElementContent(DOMNode* node, T& val ) : static void |
| + setElementContent(DOMElement* element, TVector& val ) : static void |
| + setElementContent(DOMNode* node, TVector& val ) : static void |
| + setSubElementContent(DOMElement* element, char* name, T& val ) : static void |
| + setSubElementContent(DOMNode* node, char* name, T& val ) : static void |
| + setSubElementContent(DOMElement* element, char* name, TVector& val ) : static void |
| + setSubElementContent(DOMNode* node, char* name, TVector& val ) : static void |
| + parseElement(DOMElement* element, char* name, TVector& val ) : static bool |
| + parseElement(DOMNode* node, char* name, TVector& val ) : static bool |
| + parseElement(DOMElement* element, char* name, T0& val0, T1& val1 ) : static bool |
| + parseElement(DOMNode* node, char* name, T0& val0, T1& val1 ) : static bool |
| + parseElement(DOMElement* element, char* name, T0& val0, T1& val1, T2& val2 ) : static bool |
| + parseElement(DOMNode* node, char* name, T0& val0, T1& val1, T2& val2 ) : static bool |

Abbildung 5.4: UML Diagramm: **XMLTools**

```
template <class T>
class XMLRegistry {
 public:
  XMLRegistry(){}
  ~XMLRegistry(){}

  XMLWithAttributesRegistry        <int>               RegWithAttributesInt;
  XMLWithAttributesRegistry        <long int>          RegWithAttributesLongInt;
  XMLWithAttributesRegistry        <double>            RegWithAttributesDouble;
  XMLWithAttributesRegistry        <float>             RegWithAttributesFloat;
  XMLWithAttributesRegistry        <string>            RegWithAttributesString;
  XMLWithAttributesRegistry        <bool>              RegWithAttributesBool;
  XMLWithAttributesRegistry        <vector<int> >      RegWithAttributesVecInt;
  XMLWithAttributesRegistry        <vector<long int> > RegWithAttributesVecLongInt;
  XMLWithAttributesRegistry        <vector<double> >   RegWithAttributesVecDouble;
  XMLWithAttributesRegistry        <vector<float> >    RegWithAttributesVecFloat;
  XMLWithAttributesRegistry        <vector<string> >   RegWithAttributesVecString;

  XMLSubElementAttributeRegistry <int>                 RegSubElementAttributeInt;
  XMLSubElementAttributeRegistry <long int>            RegSubElementAttributeLongInt;
  XMLSubElementAttributeRegistry <double>              RegSubElementAttributeDouble;
  XMLSubElementAttributeRegistry <float>               RegSubElementAttributeFloat;
  XMLSubElementAttributeRegistry <string>              RegSubElementAttributeString;

  XMLWithSubElementsRegistry     <int>                 RegWithSubElementsInt;
  XMLWithSubElementsRegistry     <long int>            RegWithSubElementsLongInt;
  XMLWithSubElementsRegistry     <double>              RegWithSubElementsDouble;
  XMLWithSubElementsRegistry     <float>               RegWithSubElementsFloat;
  XMLWithSubElementsRegistry     <string>              RegWithSubElementsString;
  XMLWithSubElementsRegistry     <bool>                RegWithSubElementsBool;
  XMLWithSubElementsRegistry     <bool>                RegWithSubElementsXBool;
  XMLWithSubElementsRegistry     <bool>                RegWithSubElementsInvXBool;
  XMLWithSubElementsRegistry     <vector<int> >        RegWithSubElementsVecInt;
  XMLWithSubElementsRegistry     <vector<long int> >   RegWithSubElementsVecLongInt;
  XMLWithSubElementsRegistry     <vector<double> >     RegWithSubElementsVecDouble;
  XMLWithSubElementsRegistry     <vector<float> >      RegWithSubElementsVecFloat;
  XMLWithSubElementsRegistry     <vector<string> >     RegWithSubElementsVecString;

  XMLWithContentRegistry         <int>                 RegWithContentInt;
  XMLWithContentRegistry         <long int>            RegWithContentLongInt;
  XMLWithContentRegistry         <double>              RegWithContentDouble;
  XMLWithContentRegistry         <float>               RegWithContentFloat;
  XMLWithContentRegistry         <string>              RegWithContentString;
  XMLWithContentRegistry         <vector<int> >        RegWithContentVecInt;
  XMLWithContentRegistry         <vector<long int> >   RegWithContentVecLongInt;
  XMLWithContentRegistry         <vector<double> >     RegWithContentVecDouble;
  XMLWithContentRegistry         <vector<float> >      RegWithContentVecFloat;
  XMLWithContentRegistry         <vector<string> >     RegWithContentVecString;

  XMLWithStringElementsRegistry                        RegWithStringElements;
  XMLWithElementsRegistry        <T>                   RegWithElements;
  XMLWithElementRegistry         <T>                   RegWithElement;
  void clear();
};
```

Abbildung 5.5: C++ Interface: **XMLRegistry**

#### 5.5.2.3 Bewertung der Persistenzklasse

Die Generalisierung von Persistenz ist aus softwaretechnischem Standpunkt wünschenswert. Sie ermöglicht eine *einheitliche* Form der Sicherung und Wiederherstellung von objektspezifischen Laufzeitdaten. Die wesentliche Vorgehensweise bei der Verwendung von *XMLReadWriteable* vollzieht sich in etwa so:

1. Leite Klasse $X$ ab von *XMLReadWriteable*.
2. Soll $X$ Daten lesen ?
   (a) Implementiere *preRead()*
      i. *Upcast* von dynamisch erzeugten, mehrfach abgeleiteten persistenten Unterobjekten.[24]
      ii. Registrierung von Memberdaten anhand von Zeichenketten.
   (b) Implementiere *buildObject(...)* für die Erzeugung persistenter Unterobjekte anhand von Zeichenketten.
   (c) Implementiere *postRead()*
      i. *Downcast* von dynamisch erzeugten, mehrfach abgeleiteten persistenten Unterobjekten.
      ii. Erledige Arbeiten, die nach dem Lesen erforderlich sind.
3. Soll $X$ Daten schreiben ?
   (a) Implementiere *preWrite()*
      i. *Upcast* von dynamisch erzeugten, mehrfach abgeleiteten persistenten Unterobjekten.
      ii. Registrierung von Memberdaten anhand von Zeichenketten.
   (b) Implementiere *postWrite()*
      i. *Downcast* von dynamisch erzeugten, mehrfach abgeleiteten persistenten Unterobjekten.
      ii. Erledige Arbeiten, die nach dem Schreiben erforderlich sind.

Dem aufmerksamen Leser wird auffallen, dass der beschriebene Ablauf vor jedem Lese- oder Schreibzugriff die Registrierung von Memberdaten vorsieht. Dies ist jedoch nicht immer notwendig, aber sinnvoll für einen geringeren Speicherbedarf. Bei einer großen Menge über Zeichenketten registrierter Daten, kann dieser andernfalls[25] rapide ansteigen. Allerdings ist die Optimierung der Persistenzschicht nicht Teil dieser Arbeit. Im Sinne der SLIDL-API reicht allein die Existenz dieser Schicht zunächst vollkommen aus.

[24] Der Grund für *Upcasts* und *Downcasts* wird später in Unterabschnitt 6.2.4.1 erläutert
[25] gemeint: bei einmaliger initialer Registrierung

### 5.5.3 Die Deklarations Schicht

Zur Rückbesinnung: SLIDL ist eine *konnektionsmächtige* Beschreibungssprache für XML. Die eigentlichen Sprachanteile werden in Kapitel 7 vorgestellt. Grundsätzlich muss eine API für SLIDL jeden deklarativen Sprachanteil als Datenstruktur abbilden. Dies erfolgt im Rahmen der Deklarationsschicht. Sie gliedert sich ihrerseits auf in die *Deklarations-Verwaltung* und die *Deklarations-Objekte*. Die *objektorientierte* Modellierung der API ist ebenfalls Teil von Kapitel 7, und wird dort ausgiebig erörtert.

#### 5.5.3.1 Die Deklarations-Verwaltung

Das Einlesen eines SLIDL-Dokumentes erfolgt über *persistente* SLIDL Definitionsobjekte. Als zentrale Sammelstelle existierender Definitionen fungiert dementsprechend die *Deklarations-Verwaltung*. Sie ist zugleich als zeichenkettenbasierte Schnittstelle für den Zugriff auf Definitionsdaten gedacht. Dies soll die Deklarationsschicht abgrenzen von der *Instanz-Verwaltung*.

### 5.5.4 Die Instanz Schicht

#### 5.5.4.1 Die Instanz-Verwaltung

Es ist anzunehmen, dass zur Laufzeit eines GUI mit Zugriff auf die SLIDL-API, abhängig von Definitionen der Deklarationsschicht, Instanzen von Klassen eines SLIDL-Dokuments erzeugt werden müssen. Die *Instanz-Verwaltung* übernimmt dabei die Aufgaben der Abfrage der Deklarationsschicht, der Zwischenspeicherung und Synchronisation von Definitionsobjekten und der zentralen Verwaltung von Instanzen, die während der Laufzeit einer aufsetzenden GUI benötigt werden. Sie ist die Schnittstelle für eine GUI.

#### 5.5.4.2 Abhängigkeiten von Daten zur Laufzeit

Die Möglichkeiten der Definition von SLIDL-Objekten[26] sehen vor, dass z.B. der Typ eines *Elementarobjekts* erst zu Laufzeit bekannt werden kann. Ferner ist es möglich *Elementarobjekte* in Abhängigkeit der Existenz oder Nicht-Existenz anderer *Elementarobjekte* instanziierbar zu machen. Dabei soll es dem Verwender von SLIDL freigestellt bleiben, derartige Abhängigkeiten auch zwischen *klassenfremden Elementarobjekten* zu definieren.

[26] gemeint: Klassen, Elemente und Attribute

Diese Beispiele und Vorgaben fallen unter *Abhängigkeiten von Daten zur Laufzeit*. Eine minimale zeichenkettenbasierte Zugriffssprache für SLIDL-Instanzen muss also ebenfalls Teil der SLIDL Instanz-Schicht sein.

Abbildung 5.6 zeigt auf Grundlage der Ergebnisse dieses Kapitels eine differenziertere Form der Softwarearchitektur einer SLIDL-API und eines darauf aufsetzenden GUI.

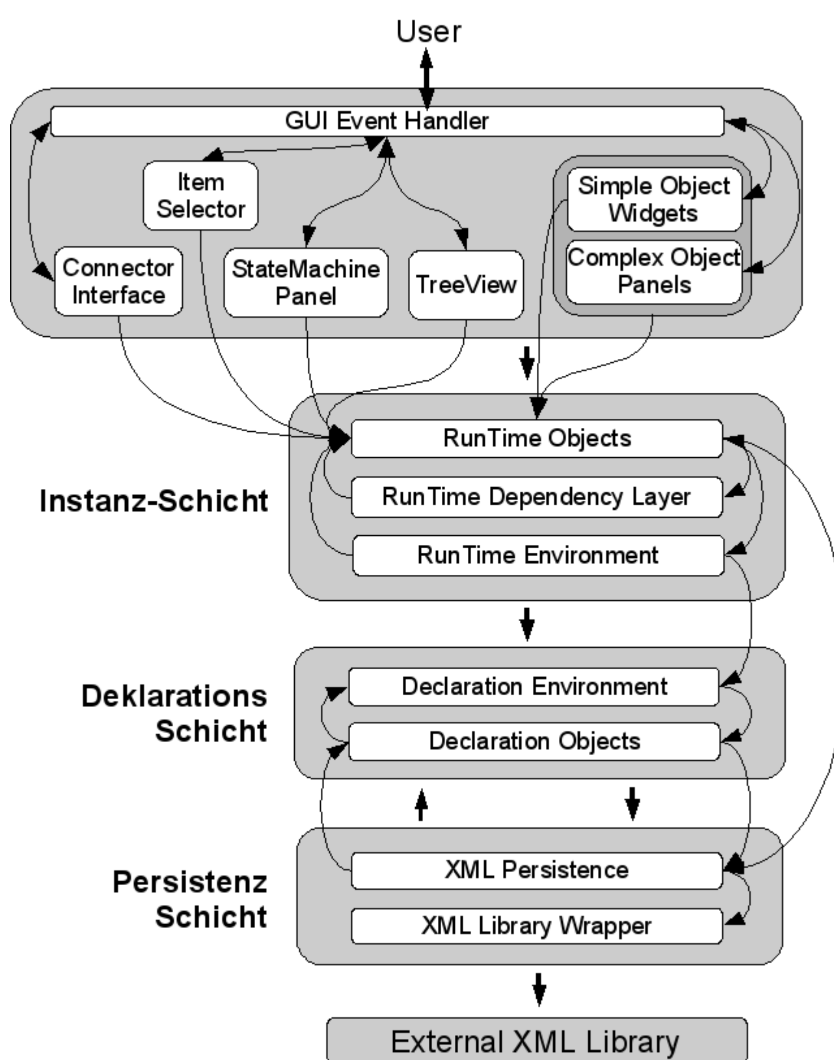


Abbildung 5.6: Die Architektur

# Kapitel 6

# Werkzeuge der Programmierung

Dieses kleine Kapitel ähnelt einem Exkurs von der Thematik weg. Dennoch wäre keine Positionierung an einer anderen Stelle des roten Fadens dieser Arbeit besser geeignet, diesen Exkurs zu vollziehen. Alle vorangegangenen Kapitel, einschließlich Kapitel 5, bereiteten den Weg hin zu Kapitel 7, systematisch aufbauend, nach den Vorgaben der Liste von Phasen des Softwareentstehungsprozesses (vgl. Teilabschnitt 2.6.1).
Im Kontrast dazu werden in diesem Kapitel zwei handwerkliche Möglichkeiten und Konzepte[1] erörtert, deren Verwendung scheinbar ungern gesehen wird. *Ungern* bezüglich einer Reihe begründeter Einwände für den Fall des unüberlegten und unstrukturierten Einsatzes[2].

Zum einen wird das Augenmerk auf Möglichkeiten und Gefahren bei der Verwendung des C/C++–Präprozessors gerichtet. Bezüglich der aufgeworfenen Probleme werden einerseits gängige konzeptionelle Gegenvorschläge vorgestellt, andererseits eine systematische Vorgehensweise zur Vermeidung unnötiger Fehler *und* des unnötigen Einsatzes dieses mächtigen, aber weise einzusetzenden Werkzeuges.
Auf der anderen Seite sollen drei Besonderheiten und Fallen[3], bei der Verwendung von Mehrfachableitungen einem reinen C++–Konzept dargestellt werden. Auch hierbei wird deutlich werden, wie wichtig ein richtiges Verständnis dieses Konzeptes, für dessen Einsatz ist, da ohne selbiges einige Phänomene auftreten können, die unerklärbar erscheinen können.

[1] des Programmierens mit C++
[2] allerdings liegt die eigentliche Ursache potentieller Probleme, dann eher an der Randbedingung
[3] wie sich zeigen wird, sogar beides gleichzeitig

## 6.1 Der C/C++–Präprozessor (Cpp)

Ein Präprozessor ermöglicht die *Rekonfiguration* des Quellcode–Textes zur Kompilierzeit. In einem Papier von 1966 unterscheidet T.E. Cheatham drei mögliche Formen von Makros von Präprozessoren (Che66):

1. **Text Makros** dienen der *textuellen* Substitution benutzerdefinierter Zeichenketten durch andere Zeichenketten. Dies entspricht der Arbeitsweise des C/C++–Präprozessors (Cpp).
2. **Berechnete Makros (Computational Macros)** ersetzen Zeichenketten durch eine Berechnung. Als Beispiele können *inline* Funktionen und *templates* herangezogen werden.
3. **Syntax Makros** ersetzen Zeichenketten unter vollständiger Kenntnis des *Syntaxbaumes* und repräsentieren ein sprachlich konsistentes Konstrukt.

Cpp überlebte den Übergang von C nach C++ ohne Veränderungen seiner Arbeitsweise. Obwohl schon früh die konzeptionellen Schwächen deutlich wurden die mit Präprozessoren einhergehen, welche unabhängig von der Sprache einer zu verarbeitenden Datei, reine textuell–orientierte Substitutionen[4] durchführen, bedingungsgebunden aktive Textblöcke auswählen können, und die Inklusion externer Dateien ermöglichen.

### 6.1.1 Traditionelle Einsatzgebiete des Präprozessors

Der letzte Absatz deutete bereits die Fähigkeiten des Cpp an. Dieser Abschnitt dient der Darlegung der traditionellen Einsatzgebiete des Cpp. (vgl. (EM00, S. 49–51)).

1. **Lexikalische Substitutionen** sind in C ein Mittel, welches analog zum Kontrukt typedef in C++ eingesetzt werden kann. Allerdings sind *typedefs* nicht analog zu allen Möglichkeiten des Einsatzes lexikalischer Substitutionen (siehe Teilabschnitt 6.1.3). Die Hauptprobleme beim Einsatz dieses Werkzeuges liegen im grundsätzlich globalen Gültigkeitsbereich von Präprozessor-Makro–Definitionen.
2. **Konkatenation von Bezeichner-Namen** können in C/C++ eingesetzt werden, um einen neuen Namen von einem gegebenen *Stamm* abzuleiten.
3. **Konvertierung von Namen zu Zeichenketten** in manchen Fällen kann es nützlich sein Namen von Klassen, Datentypen und andere sprachinterne Bezeichner als Zeichenkette vorliegen zu haben, zum Beispiel für etwaige Debug–Ausgaben.

[4] aber auch *parametrisierte* textuelle Substitutionen beherrschen

4. **textuelle Substitutionen** existieren in parametrisierter und nicht parametrisierter Form. Erstere werden in C häufig analog zum Konstrukt inline in C++ verwendet. Letztere bilden in C eine Möglichkeit zur Definition von Konstanten. C++ ermöglicht dagegen den Einsatz des Schlüsselwortes const.
5. **Bedingte Kompilierung** ist die Möglichkeit über Kontrollstrukturen auf der Ebene des Cpp je nach Bedarf auf verschiedene Quellcode–Variationen zuzugreifen. Dies ist heute noch der einzige Weg zur Kompilierzeit bedingungsabhängig Code–Teile zu rekonfigurieren; sowohl in C, als auch in C++.
6. **Datei–Inklusionen** werden sowohl in C, als auch in C++ noch heute ausschließlich von Cpp getätigt.

### 6.1.2 Das traditionelle Kompiliermodell

Aus Punkt 6 im Teilabschnitt 6.1.1 wird ersichtlich: Das mit Cpp/C++ verbundene Kompiliermodell erfordert den Aufruf von Cpp vor der Kompilierung einer jeden Quellcode–Datei. Es handelt sich um ein zweistufiges Kompiliermodell, dessen zwei Stufen wie folgt verstanden werden können (EM00, S. 52):

1. die sequentielle Kompilierung aller Cpp-vorverarbeiteten Quellcode–Dateien
2. das Verknüpfen der entstandenen Objekt–Dateien zu einer *binären Einheit*, durch den *Linker*

Die Betrachtung dieses Aspektes soll in Teilabschnitt 6.1.5 wieder aufgegriffen und etwas vertieft werden.

### 6.1.3 Cpp Templates und C++ Templates

In diesem Teilabschnitt soll auf die Möglichkeit hingewiesen werden, dass Cpp zur Erzeugung von *Templates* verwendet werden kann. Der eigentliche Vorteil von C++ - *Template*–Konstruktionen, als *typparametrisierten* Code–Einheiten, liegt in der Möglichkeit des Compilers Aussagen über Schwierigkeiten während der Kompilierzeit zu treffen. Dies innerhalb des semantischen Rahmens eines Fehlers. Teilweise ermöglicht diese Voraussetzung dem Compiler auch die Ableitung einer Hilfestellung.
Wer allerdings die Ausgaben des Compilers im Rahmen von Syntaxfehler–Meldungen bei falscher Verwendung von STL–Containern gesehen hat, kann leichten bis mittelschweren

Zweifel der Annahme gegenüber entwickeln, dass die im letzten Absatz als Vorteil angepriesene Eigenschaft von C++–Templates tatsächlich einen Vorteil darstellt. Viel wesentlicher, als die wenig hilfreichen, zuweilen sogar irreführenden und kaum lesbaren Augaben des Compilers hinblicklich der falschen Verwendung von Templates, ist die Tatsache ihrer Parametrisierbarkeit anhand eines oder mehrerer *Typ*–Parameter. Dies spart Entwicklungszeit und verkleinert den Aufwand der Wartung und Pflege von syntax–strukturell identischen, aber in der Verwendung interner Typen unterscheidbaren Klassen und Verbundtypen[5].

Je nach Anforderung und Tiefe der Verschachtelung und Rückbezüglichkeit von Template–Parameter-Abhängigkeiten, kann stattdessen ebenso auf Cpp–Templates zurückgegriffen werden. Dabei handle es sich um Dateien die folgende Eigenschaften erfüllen:

1. Sie enthalten vollständige und konsistente C++–Deklarationen und/oder Definitionen.
2. Sie definieren eine Schnittstelle der von ihrem Benutzer zu definierenden Cpp–Template - Parameter.
3. Sie prüfen beim Inklusionseintritt die Verwendung aller benötigten Parameter, und geben entsprechende Fehler bei Inkonsistenzen aus.

Wenn ferner der Benutzer eines Cpp–Templates zusätzlich dafür Sorge trägt, nach der Inklusion, alle Template–Parameter wieder ungültig zu machen, so stellt die Erstellung und Verwendung eines Templates auf Basis des Präprozessors keine Schwierigkeit mehr dar.

### 6.1.4 Der Gültigkeitsbereich von *define* Ausdrücken

Eines der größten Probleme bei der Verwendung des Cpp[6] ist mit Sicherheit, die fehlende Möglichkeit C++–Namensräumen Makros unterzuordnen. Allgemeiner, das Fehlen von Strukturen der Definition von Gültigkeitsbereichen für Makros des Cpp. Obwohl dieses Problem mit den Jahren nicht an Relevanz verliert, sind erst wenige ernstzunehmende Bestrebungen zu dessen Lösung eingeleitet worden.

[5] C/C++: struct

[6] abgesehen von der auf einzeiligen Statements beruhenden Syntax, die aber im Rahmen grundsätzlicher Überlegungen und Ableitungen zur Verwendung eines Präprozessors zu vernachlässigen ist

Ein recht aktueller Ansatz ist dem Paper (siehe (unk04)) zu entnehmen. Dort werden Forderungen an ein entsprechendes Konzept erörtert, mittels der Ausdrücke *scope* und *endscope*, Gültigkeitsbereiche für Makros zu markieren. Jede Definition innerhalb eines *Scopes* soll dabei nur innerhalb des *Scopes* gelten und den globalen Namensraum nicht *verschmutzen*. Ferner sollen mittels der Makros *import* und *export* der Import und Export von *Scope*-intern definierten Makros ermöglicht werden. Die Menge aller *export*-Kommandos eines *Scopes* bildet somit dessen öffentlich zugängliche Schnittstelle.

Die Erweiterung des Cpp, unter Verwendung des hier beschriebenen Vorschlages stellt sicher eine erhebliche Verbesserung des jetzigen Zustandes dar. Die grundsätzlichen Schwächen seiner Konzeption, als einem rein auf *textuellen* Dateimerkmalen arbeitenden Präprozessor sind damit allerdings noch nicht behoben. Ferner eröffnet sich die Frage warum die genannten *Scopes* anonym gehalten und nicht in ein Ordnungssystem hierarchiesierender Namensraumstrukturen eingebunden werden.

### 6.1.5 Objektorientierte Präprozessoren

Die wesentliche Problematik rein textuell operierender Präprozessoren konnte der Vorschlag von Teilabschnitt 6.1.4 nicht lösen. In diesem Teilabschnitt soll sich eine Regel als wahr erweisen, welche unabhängig von der Ausrichtung einer Problemstellung Gültigkeit hat: *Wenn eine Sache im Wesen krankt, sollten wir sie in Respekt*[7] *sterben lassen.*
Der auf *Syntax Makros* basierende Präprozessor **FOG** (**F**lexible **O**bject **G**enerator)(EM00) soll hier vorgestellt werden. Dieser berücksichtigt die objektorientierten Ansprüche und die syntaktische Struktur der Sprache C++. Die Tatsache seiner Kenntnis über die Sprache der vorzuverarbeitenden Dateien, macht ihn zu einem mächtigen Werkzeug. Er setzt dabei Konzepte um, wie die Nutzung der C++–Namespaces, die Möglichkeit zur Definition präprozessorbezogener *Meta-Variablen*, sowie typloser, als auch typisierter *Meta-Funktionen*. Die gesamte Syntax von FOG, als auch die innerhalb von FOG bereitgestellten Datentypen, haben ihre Entsprechung in C++.

Interessanter Weise führt die Konzeption von FOG die C++–Compiler unverträgliche Regel ein, dass Mehrfach–Definitionen erlaubt sind, solange sie sich nicht widersprechen. Auf

[7] Erläuterung: Natürlich ist Respekt auf Dinge bezogen, die diesen verdienen. Der Cpp stellt sich als ein solches dar. Das deutsche Steuerrecht oder repräsentative Demokratiesysteme hingegen nicht.

diese Weise kann, wie unter Teilabschnitt 6.1.2 angedeutet, das *Kompiliermodell* von C++ umgestellt werden. Mit dessen Erweiterung, um die Stufe des FOG, auf ein dreischichtiges Modell, können zunächst alle Quellcode–Dateien vorverarbeitet, und dabei *Implementierungen* von den *Interfaces* getrennt werden. Diese getrennten Einheiten werden schließlich von doppelten Definitionen befreit, und der zweiten Schicht (der des Compilers) übergeben. Die unter C/C++ oft benötigte Trennung von Header- und Codedateien wird auf diese Weise hinfällig. Die Autoren schlagen hinsichtlich einer derartigen Kenntnis der vorzuverarbeitenden Sprache die Bezeichnung *Meta–Compiler* vor.

### 6.1.6 Zusammenfassung und Folgerungen

Die vorangegangenen Teilabschnitte konnten ausführlich darlegen, welche Möglichkeiten und Probleme mit der Verwendung des C/C++–Präprozessors und daran gebundener Techniken existieren. Im Wesentlichen konnte dargelegt werden, das Vorsicht geboten ist beim Einsatz dieses Werkzeuges, da es von der zugrunde liegenden Struktur der verwendeten Sprache nichts weiß.
Hinblicklich der Erstellung von typparametrisierten Templates auf der Basis von Präprozessor - Kommandos konnte gezeigt werden, welche grundsätzliche Vorgehensweise zu beschreiten ist, um möglichst unbeschadet und systematisch möglichen Fehlern vorbeugend entgegenzuwirken. Dabei wurde deutlich, dass die systematisierte Verwendung von Cpp–Konstruktionen, sehr stark abhängt von der Bewußtheit des Programmierers hinblicklich der Aktivierung und Deaktivierung von Gültigkeitsbereichen einzelner Gruppen von Makros.

Mit der Überleitung zu den Gültigkeitsbereichen wurden zwei konträr ansetzende Konzepte vorgestellt: das *Scoping* und der *Meta–Compiler* FOG (vgl. Teilabschnitt 6.1.4 und Teilabschnitt 6.1.5). Beide Konzepte zeigen, dass viel Potential für Verbesserungen im traditionellen Konzept des Cpp impliziert wird. Die grundsätzliche Frage nach dem Sinn und Zweck der Verwendung präprozessorabhängiger Programmierkonstruktionen ist problembezogen zu beantworten. Sie kann nicht global beantwortet werden, da zwei Dinge parallel gelten:

1. Der traditionelle Präprozessor schwächelt hinblicklich des *Scopes*. Dies kann nur durch zusätzliche Sicherheitsrichtlinien kompensiert werden. (vgl. Teilabschnitt 6.1.3)

2. Jede Form der systematischen Verkleinerung des Codestammes bei Erhaltung der Funktionalität ist als Verbesserung der Qualität zu werten.

## 6.2 Mehrfach Ableitungen in C++

Ein weiteres unter Entwicklern, mit geteilter Meinung, aufgenommenes Prinzip der Programmierung ist nahtlos integriert in das Konzept von C++: *Mehrfachableitungen*. Es repräsentiert eines von zwei zu unterscheidenden Konzepten von Objektorientierung, den polyhierarchischen Ansatz. Dieses Teilkapitel widmet sich drei Aspekten, die bei Unkenntis derselben tückische Fallstricke bilden können, und deswegen vor einem gezielten Einsatz der besagten Technik, ins Bewußtsein zu rufen sind.
Aufgrund der diesem Kapitel zugrunde liegenden recht anschaulichen Thematik, werden alle zu beschreibenden Phänomene, mittels Abbildungen verdeutlicht. Dies macht dieses Teilkapitel, im Vergleich zum vorangegangenen, zu einem leicht verdaulichen. Die folgenden Fälle sind unterschieden nach der Online Ausgabe von (Eck00, Kap. 6).

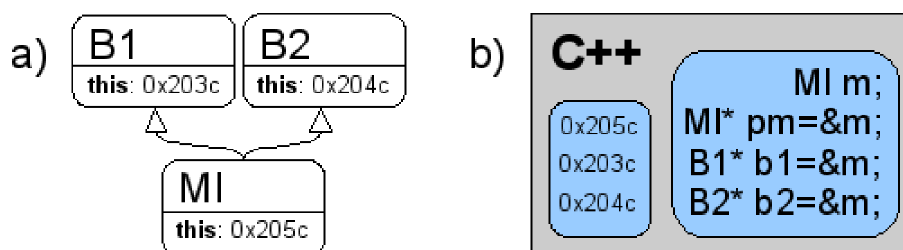


Abbildung 6.1: Schizophrenes Objekt

### 6.2.1 Schizophrene Objekte

Das Motiv des Titels des aktuellen Teilabschnittes ist nicht begründet in der Existenz einer *pathologisch* abnormalen Form von Klassendefinitionen, als vielmehr in einer Eigenschaft des *this*–Pointers und der mit ihm verknüpften Objekte, wenn diese Instanzen einer mehrfach abgeleiteten Klasse sind. Abbildung 6.1a) soll darauf hindeuten, dass bei Instanziierung einer mehrfach abgeleiteten Klasse, für jede Oberklasse, als auch fur die der erzeugte Instanz der abgeleiteten Klasse, ein eigener *this*–Pointer zur Verfügung gestellt wird.
Dieser Umstand wird allerdings erst relevant, wenn über einen *Pointer* einer der Oberklas-

sen (auf ein Objekt einer Unterklasse) auf eine ihrer geerbten Methoden zugegriffen werden soll. Eine entsprechende Situation soll Abbildung 6.1b) verbildlichen. Das *Upcasten* der Adresse einer multipel erbenden Instanz, zu der Adresse einer ihrer Oberklassen, ändert nicht nur die, mit der Instanzadresse verknüpfte *Typinformation*[8], sondern die Adresse selbst.

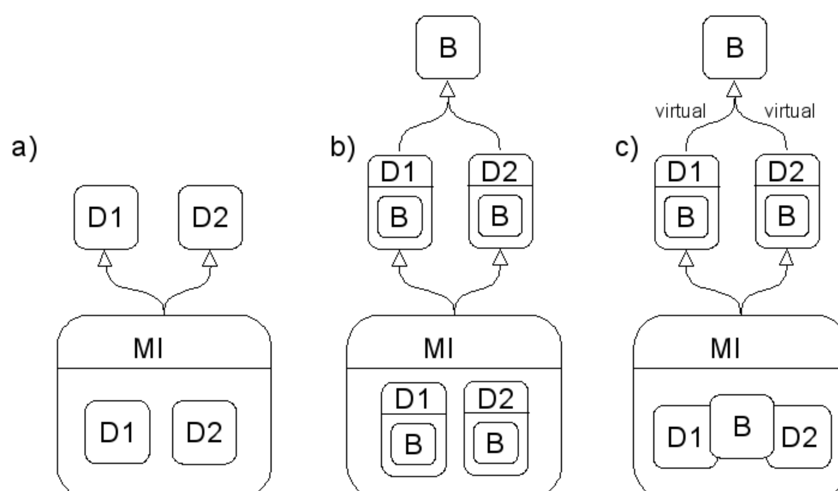


Abbildung 6.2: Virtuelle Mehrfach–Ableitungen

### 6.2.2 Virtuelle Ableitungen

Im Zusammenhang von Mehrfachableitungen muss geklärt werden, was bezogen auf die Speicherstruktur eines Objektes zu erwarten ist. Dies wird in Abbildung 6.2a) dargestellt. Angenommen beide Oberklassen enthalten jeweils eine Funktion *func*(), so würde der Compiler einen Aufruf der Instanz ***this***->*func*() als ambig werten, da er nicht eindeutig entscheiden kann welche *func*() der beiden Oberklassen ausgewählt werden soll. Dies ist in C++ nur über die vollständige Angabe des Namensraumes zu umgehen: z.B. durch den Aufruf von ***this***->*D1::func*().

Ein weiteres Phänomen, dass je nach Situation erwünscht sein kann, es aber häufig nicht ist, wird in Abbildung 6.2b) gezeigt. Diese Konstruktion positioniert B zweimal innerhalb

[8]Bemerkung: Dies wäre das übliche Verständnis des *Castings* bezogen auf singulär erbende Instanzen

von MI. Einmal über dessen Oberklasse D1, ein zweites über D2. Um diesen Effekt zu umgehen implementiert C++ das Konzept der *virtuellen Ableitung*.
Eine *virtuelle Ableitung* wird in der Definition des abzuleitenden Objektes signalisiert, durch das Schlüsselwort **virtual** vor/nach[9] dem Schlüsselwort **public** bei der Angabe der Oberklasse. In folgender Weise:

```
    class D1 : virtual public B {...}
    class D2 : virtual public B {...}
class MI : public D1, public D2 {...}
```

### 6.2.3 Kreuz–Delegation bei virtuellen Mehrfachableitungen

Die letzte Kuriosität bezüglich der Verwendung von Mehrfachableitungen ist etwas komplexer angelegt, und kann während der Planungsphase eines Projekts, als mächtiges Werkzeug eingesetzt werden. In Abbildung 6.3 sei die Ausgangssituation dargestellt. B deklariert zwei *virtuelle* und *abstrakte* Methoden. Ferner wird die eine Funktion in Bs virtueller Ableitung D1 und die andere in seiner virtuellen Ableitung D2 implementiert. Dabei sei D2s implementierte Funktion der Form, dass sie, ohne Kenntnis über eine existierende Implementierung der zweiten virtuellen Funktion von B, diese aufruft. Wie auch im Abschnitt zuvor, sei MI eine multiple Ableitung von D1 und D2.
Wenn nun die Adresse einer Instanz der Klasse MI gecastet wird, zu einer Adresse der Oberklasse D1; was passiert dann beim Aufruf der, D1 unbekannten, in D2 implementierten Funktion ?

Anders als *intuitiv* zu erwarten, offenbart das *Sequenzdiagramm* in Abbildung 6.3b), dass der Aufruf dieser Funktion völlig legitim ist. Dies ist eine bemerkenswerte Eigenschaft von C++ bezüglich der Umsetzung von virtuellen Funktionen im Rahmen der Mehrfachableitung mehrerer, von einer gemeinsamen Oberklasse virtuell abgeleiteter Klassen, zu einer neuen Unterklasse.

[9] beides ist erlaubt

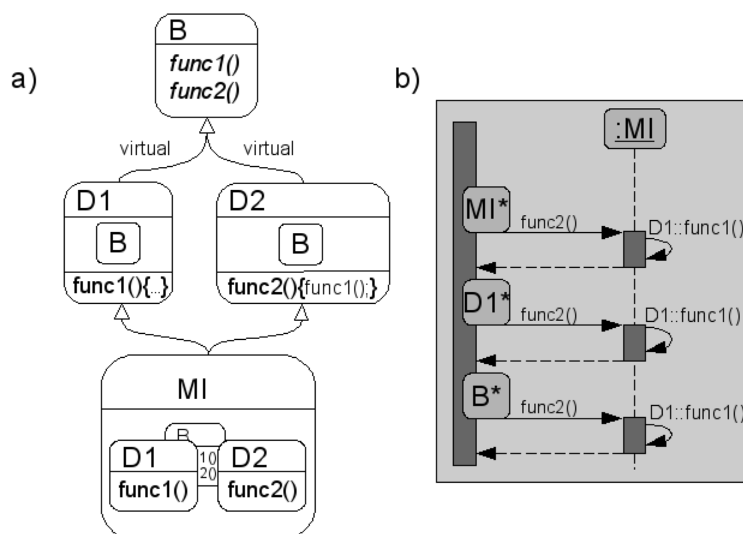


Abbildung 6.3: Kreuzdelegation bei virtueller Mehrfachableitung

### 6.2.4 Zusammenfassung und Folgerungen

Dieses Teilkapitel konnte anschaulich drei mit Mehrfachableitungen (in C++) verbundene Konzepte vorstellen:

1. **Schizophrene Objekte**
2. **Virtuelle Ableitungen**
3. **Kreuzdelegation**

Bei Unkenntis über diese Konzepte, können im Falle der Verwendung von Mehrfachableitungen Phänomene zu Tage treten, die aus der Logik über einfachen Ableitungen nicht erklärbar sind. Diese müssen tückischer Weise nicht notwendig in Speicherzugriffs- oder Compilerfehlern münden.

#### 6.2.4.1 Beispiel: Gemeiner Fehler bei schizophrenen Objekten

Gerade am Beispiel schizophrener Objekte kann man das verdeutlichen. Wie die Verknüpfung der Abbildungen 6.1a) und 6.2a) erkennen lässt, gelten bezüglich einer Mehrfachableitung *immer* zwei Dinge:

1. Instanzen einer Mehrfachableitung sind *schizophrene* Objekte.
2. Instanzen einer Mehrfachableitung enthalten vollständige Kopien ihrer Oberklassen.

Versucht man über einen Pointer der Klasse D1 (erzeugt durch *Upcasting* eines Objekts der Klasse MI) ohne vorherigen *Downcast* eine Attributinformation zu erhalten, so wird das Abfragen korrekter Daten scheitern[10]. Da D1 vor D2 im Speichersegment von MI lokalisiert ist, tritt dennoch keine Verletzung der Speichergrenzen von MI ein. Ein Fehler dieser Art ist jedoch nicht leicht aufzuspüren, wenn man ihn lediglich mit der Logik monohierarchischer Objektorientierung zu finden sucht.
Noch tückischer ist allerdings, dass im besagten Fall ein *Downcast* nicht ohne weiteres möglich ist. Zwar sind mit einem Pointer auf eine Oberklasse alle nötigen Typinformationen bekannt, nicht jedoch die *this*-Pointer der Unterklasse(n). Um einen korrekten *Downcast* zu erzielen, muss die Oberklasse also den/die Pointer auf die Instanz der Unterklasse kennen. In Abbildung 5.3, dem UML Diagramm von *XMLReadWriteable* wird dargestellt wie eine mögliche Konstruktion aussehen kann, damit ein *Downcast* problemlos vollzogen werden kann. Ein *Upcast* muss demnach verbunden werden mit der Übergabe des Pointers der Unterklasse an die Oberklasse. In der besagten Konstruktion von *XMLReadWriteable* wird dies über die Funktion *me(void* DerivedClass)* ermöglicht. Der *Downcast* kann somit auf dem Ergebnis der Funktion *me()* ausgeführt werden. Der angedeutete Fehler kann damit gezielt umgangen werden.

## 6.3 Abschließendes

Die in Teilabschnitt 6.1 und Teilabschnitt 6.2 geführten Schlussfolgerungen über den Einsatz der dort vorgestellten Werkzeuge und Konzepte ergaben, dass in beiden Fällen, nicht allein anhand des betrachteten Objektes eine grundsätzliche Eignung befürwortet oder abgelehnt werden kann oder sollte.
Es scheint sich eher am Zweck der Nutzung, sowie am Grad bewusster Auseinandersetzung eines Entwicklers mit potentiellen Schwierigkeiten und Möglichkeiten festmachen zu lassen, wie eine individuelle Bewertung im Einzelfall auszusehen hat. Generell haben sowohl der Präprozessor, als auch die Mehrfachableitung ihre Daseinsberechtigung und ermöglichen dabei einige komfortable Vorgehensweisen. Das aber mitunter schwerwiegenste Argument für den vernünftigen und verantwortungsvollen Einsatz dieser beiden Werkzeuge, ist die *qulitative* Verbesserung des Codes, gemessen an der Verkleinerung des Codestammes bei gleichzeitiger Erhaltung der Funktionalität.

[10] Diese Daten sind *relativ* zum **this**-Pointer von MI (im Speichersegment desselben) zu suchen.

Im Rahmen der Implementierung der XML–Persistenz Schicht (vgl. Teilabschnitt 5.5.2) wurde starker Gebrauch des Cpp gemacht. Dies ermöglichte dessen schnellen Umbau von der Erstimplementierung hin zu einer adäquat funktionierenden, aber vergleichsweise übersichtlicheren Zweitimplementierung, innerhalb weniger Minuten, ohne eine der davon abhängigen Klassen abändern zu müssen. Ferner konnten, durch den Einsatz parametrisierter Makros, die Anzahl verwendeter Zeilen in den *Kernfunktionen* der Persistenz–Schicht, um den Faktor 8 verringert werden, zugunsten einer übersichtlich gestalteten Implementierung. Die Einhaltung des Prinzips *lokal* gültiger Makros wurde dabei systematisch umgesetzt.

Selbiges gilt auch hinsichtlich des Einsatzes von *Mehrfachableitungen*. Das, im folgenden Kapitel vorgestellte, Konzept der *Attributklassen* ist zwar nicht zur Verkleinerung des Programmcodes eingesetzt worden, sondern zur Kapselung generalisierbarer Funktionalität; führte aber dennoch zu eben diesem Effekt.

# Kapitel 7

# Entwurf und Implementierung

Dieses Kapitel ist der Definition von sprachlichen Komponenten SLIDLs und der Modellierung und Implementierung von dessen API gewidmet. Vorweg sei angemerkt, dass alle Implementierungen von SLIDL-*Definitionsobjekten*, sowie SLIDL-*Instanzobjekten* persistent sind. Demnach sind sie alle von *XMLReadWriteable* abgeleitet.

## 7.1 SLIDL als Beschreibungssprache

Kapitel 5 bot den ausgiebigen Einblick in potentielle Sprachanteile einer *konnektionsmächtigen Beschreibungssprache* für XML. Dieser Abschnitt ist der Abbildung der beschriebenen allgemeinen Konzepte auf SLIDL gewidmet. Grundsätzlich sei darauf hingewiesen, dass die konkrete Wahl der XML-Strukturen *Geschmackssache* sind. Sehr deutlich wird dies in Hinblick auf *XMLReadWriteable*: Die Änderung eines Attributs zu einem Element oder umgekehrt entspricht der Änderung *genau* einer Zeile innerhalb der Implementierung eines SLIDL-Sprachelements.

### 7.1.1 Sprachdefiniton

Es existieren zwei Arten von Sprachelementen in SLIDL:

1. deklarative Sprachanteile
2. administrative Sprachanteile

*Deklarative* Sprachanteile beschreiben dabei die *semantische Struktur* eines SLIDL-Dokuments und sind zum Teil vergleichbar mit den Sprachelementen von XSDs. *Administrative* Antei-

le sind Hilfsstrukturen, zur Vereinfachung der Formulierung *deklarativer* Anteile und der modularen Gestaltung von SLIDL-Dokumenten. Das wohl wichtigste *deklarative* Sprachelement ist *Class*. Es basiert auf nahezu allen weiteren *deklarativen* Sprachanteilen und wird auf Abbildung 7.1 dargestellt.

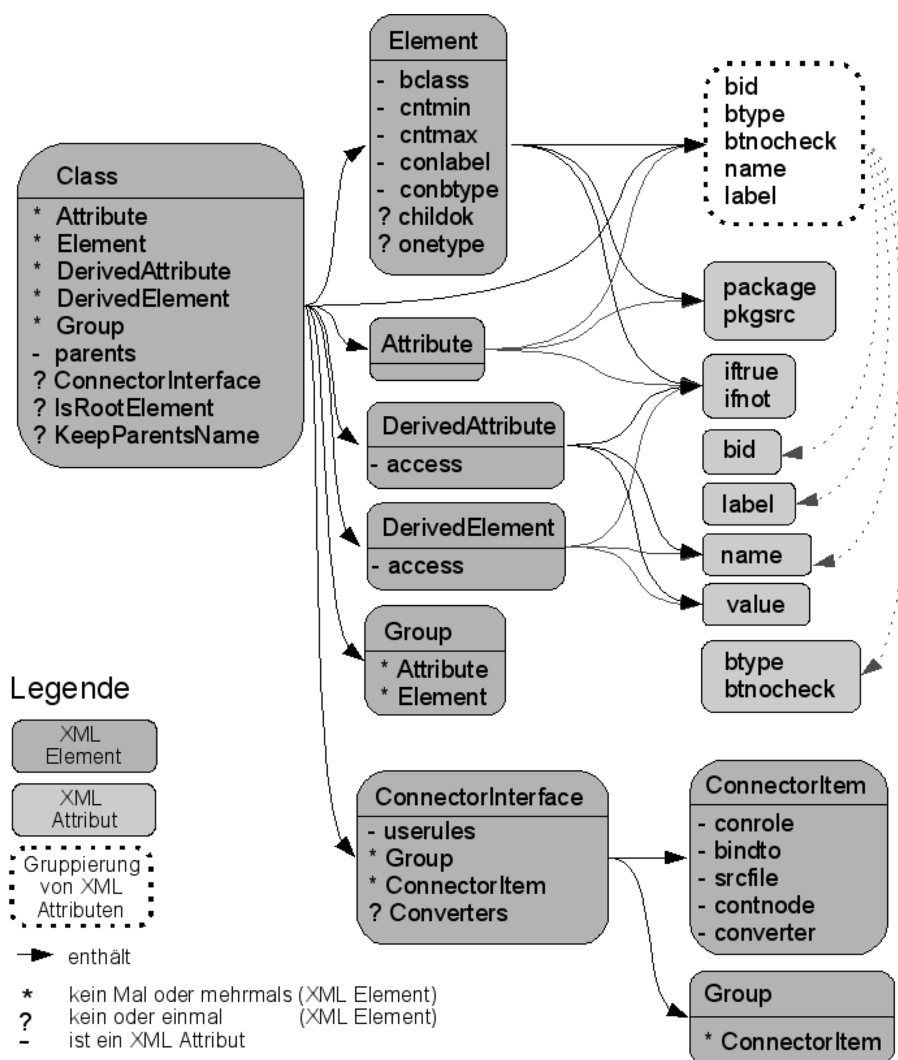


Abbildung 7.1: Die SLIDL Sprachdefinition

### 7.1.2 Class: Beschreibung untergeordneter Sprachelemente

#### 7.1.2.1 Gemeinsame Attribute

**bid:** SLIDL entstand basierend auf Szenedateien des DRAS. Dieses verzichtet auf die Verwendung von XML Namensräumen. Ebenso so verhält es sich mit SLIDL. Aus diesem Grund unterscheidet es den Namen den ein Element oder ein Attribut innerhalb einer Instanz eines SLIDL-Dokuments trägt von seiner Basis-ID *bid*. Auf diese Weise sollen Definitionsobjekte eindeutig unterscheidbar bleiben unabhänging von ihrer Gleichheit innerhalb einer Szenedatei.

**btype:** Mit *btype* kann Elementarobjekten ein Basistyp oder die Ableitung eines Basistyps zugeordnet werden.

**btnocheck:** Die API von SLIDL kann mittels des XBool-Elementes *btnocheck* angewiesen werden die Existenz eines *btypes* nicht zu prüfen. Dies kann sinnvoll sein, wenn der Typ erst zur Laufzeit endgültig bekannt wird.

**ifnot:** Elementarobjekte können mittels *ifnot* und *iftrue* in Abhängigkeit von Laufzeitdaten aktiviert bzw. deaktiviert werden.

**iftrue:** siehe *ifnot*

**label:** Das *label* enthält die Bezeichnung, die einem Sprachelement innerhalb der GUI zugeordnet werden soll.

**name:** Der Name eines Sprachelements innerhalb einer XML-Datei, z.B. einer DRAS-Szenedatei.

**package:** Zur Laufzeit kann eine Instanz einer Sprachdefiniton über eine Zeichenkette einem *package* zugeordnet werden. Dies ermöglicht die zentrale Sammlung bestimmer Instanzen, um diese an anderer Stelle zur Auswahl stellen zu können.

**pkgsrc:** Die mit *package* gesammelten Instanzen können anderen Instanzen über *pkgsrc* zur Auswahl angeboten werden.

**value:** Die Inhalte von Instanzen verschiedener Sprachelemente SLIDLs können vorinitialisiert werden. Dies geschieht mittels des Attributs *value*.

#### 7.1.2.2 Elementarobjekte

**Element:** Ein *Element* entspricht einem XML-Element elementaren Typs. Da ein Element im Gegensatz zu einem Attribut mehrfach auftreten kann, enthält es die Attribute *cntmin* und *cntmax*, welche die minimale und maximale Anzahl des Auftretens eines Elementes beinhalten. Ferner kann ein Element auch als Link auf eine Klasse

verwendet werden. Dies ist mit *bclass* zu erreichen.
In Teilabschnitt 5.3.4 war unter anderem von Konnektionsoperanden mit *Elementarobjekt-Korrespondenz* die Rede. Dies sind Konnektionsoperanden, welche direkt abhängen von Elementarobjekten einer Klasse. Das Hinzufügen eines neuen Elementes forciert in diesem Sinn das Hinzufügen eines neuen Konnektionsoperanden. Mittels der Attribute *conlabel* und *conbtype* können das Label (innerhalb des GUI) und der *Basistyp* eines solchen Konnektionsoperanden festgelegt werden.
Es kann erwünscht sein, dass der Typ eines Elementes erst *gewählt*[1] werden muss. Wenn zusätzlich erwünscht ist, dass der Typ des ersten Elementes automatisch den Typ aller Elemente mit gleicher *bid* bestimmt, kann das XBool-Element *onetype* dafür verwendet werden.
Wenn ein Element mittels *blass* auf eine Klasse verweist, so ist durch das XBool-Element *childok* festlegbar, ob auch Ableitungen dieser Klasse als Subcontainer akzeptabel sind.

**Attribute:** Ein *Attribut* entspricht einem XML-Attribut. *Attribute* sind per Definition niemals Container. Sie sind folglich immer elementaren Typs.

**DerivedElement:** Die Ableitung einer *Klasse* kann erfordern, dass bestimme Elementarobjekte der Ableitung eingeschränkt werden müssen: im Zugriff oder in ihrer Existenz. Das Attribut *access* ist hierfür vorgesehen, welches die Werte *readonly*, *readwrite* und *none* enthalten kann.

**DerivedAttribute:** siehe *DerivedElement*

#### 7.1.2.3 Container-Objekte

**Group:** Um innerhalb des GUI eine Gruppierung von Elementarobjekten zu beschreiben, ist das Containerobject *Group* vorgesehen.

**ConnectorInterface:** Jeder Konnektionsoperand einer Klasse ist innerhalb von *ConnectorInterface* als *ConnectorItem* zu definieren. Auch diese können mittels *Group* gruppiert werden.

[1] gemeint: vom Benutzer des GUI

#### 7.1.2.4 Elemente und Attribute von Klassen

**parents:** Die Oberklassen einer Klasse können durch Leerzeichen getrennt innerhalb des Attributs *parents* angegeben werden.

**IsRootElement:** Wenn eine Klasse das Wurzelelement eines XML-Dokuments repräsentieren soll, so ist das XBool-Element *IsRootElement* zu verwenden. Im DRAS würde die Klasse *Scene* demnach mit diesem Element ausgestattet, da ein *Scene*-Container das Wurzelelement einer Szenedatei ist.

**KeepParentsName:** Leitet man eine Klasse von einer anderen Klasse ab, so kann es erwünscht sein, dass beide Klassen innerhalb einer Instanz einer SLIDL-Definition den gleichen DOM-*Knotennamen* teilen. Ein Weg dies zu erreichen ist das Setzen des Attributs *name* der Unterklasse mit dem Namen der Oberklasse. Die zweite Möglichkeit ist das XBool-Element *KeepParentsName* zu verwenden. Im Falle mehrerer Oberklassen würde in diesem Fall der Name der ersten Klasse genutzt.

### 7.1.3 Typisierungs-System

SLIDL unterscheidet zwischen *Strukturtypen* und *Elementartypen*[2]. Die Definition einer Klasse entspricht der Definition eines *Strukturtyps*. Leitet man eine Klasse aus anderen ab, so entspricht dies der Definition abgeleiteter *Strukturtypen*. Eine Unterklasse erbt dabei alle Elementarobjekte der Oberklasse, sowie das *ConnectorInterface*. Über *Derived** ist die Einschränkung oder Deaktivierung geerbter Elementarobjekte möglich. Es existieren keine vordefinierten Strukturtypen.
*Elementartypen* sind gegenüber den *Strukturtypen* abhängig von vordefinierten *Basistypen*. Diese und die Möglichkeit deren Ableitung stehen in diesem Abschnitt im Vordergrund.

#### 7.1.3.1 Vordefinierte Basistypen

Vordefinierte *Basistypen* in SLIDL sind:

**STRING:** Dieser Typ entspricht beliebigen Zeichenketten. Da XML ein textuelles Format ist, kann STRING auch als beliebiger Typ aufgefasst werden. Es ist keine spezielle Typprüfung für Inhalte von elementaren SLIDL-Instanzen zu implementieren.

**REALNUM:** Mit REALNUM werden Fließkommazahlen assoziiert.

[2] siehe Teilabschnitt 5.2.3

**INTEGER:** Mit diesem Typ werden ganze Zahlen assoziiert.

**BOOL:** Dieser Typ ist abhängig von der Implementierung der Persistenzschicht[3] und beschreibt boolsche Werte.

**XBOOL:** Dieser Typ entspricht dem Typ *XBool* der Persistenzschicht.

**IXBOOL:** Dieser Typ entspricht dem Typ *InvXBool* der Persistenzschicht.

**FILEPATH:** Wenn einem Elementarobjekt ein Dateipfad zugeordnet werden und die SLIDL-API die Existenz dieser Datei prüfen soll, kann dieser Typ verwendet werden.

**DIRPATH:** Dies ist die Entsprechung von FILEPATH bezüglich Verzeichnissen.

#### 7.1.3.2 Abgeleitete Basistypen

In Abhängigkeit von den vordefinierten Basistypen können benutzerdefinierte Typen abgeleitet werden. Mit dem SLIDL-Sprachelement *DerivedType* ist dies zu erreichen. Abbildung 7.2 stellt die semantische Struktur von *DerivedType* dar. Für einen abgeleiteten Basistyp existieren folgende Möglichkeiten:

1. Für die Ableitung eines numerischen Typs ist die Einschränkung des Wertebereichs vorgesehen. In Abbildung 7.2 ist ersichtlich, dass die *Constraints* dafür in Frage kommen.
2. Die Definition *vektorieller Typen* wird über das Attribut *size* ermöglicht.
3. Die Auswahl aus einer Liste von Möglichkeiten gewährleistet der Container *Choice*.
4. Es ist denkbar, dass der Inhalt eines Elementarobjekts z.B. ein Datum, eine E-Mail Adresse oder ähnliches enthält. Es handelt sich dabei um spezielle Zeichenketten deren syntaktische Korrektheit mittels regulärer Ausdrücke prüfbar gemacht werden soll. Dafür ist *RegExp* vorgesehen.

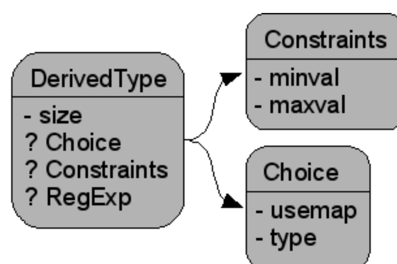


Abbildung 7.2: SLIDL: Abgeleitete Basistypen

[3] siehe Unterabschnitt 5.5.2.2

### 7.1.4 SLIDLDocument: Repräsentation der Zielsprache

Die Repräsentation der Zielsprache erfolgt über ein oder mehrere *SLIDLDocuments*. Dessen semantische Struktur ist auf Abbildung 7.3 zu sehen. *Maps* sind als assoziative Listen zu verwenden. Mittels *Includes* können andere *SLIDLDocuments* eingebunden werden. Für das DRAS könnte also gelten: Jedes Plugin und jeder Kernbaustein kann in einer seperaten Datei definiert werden.

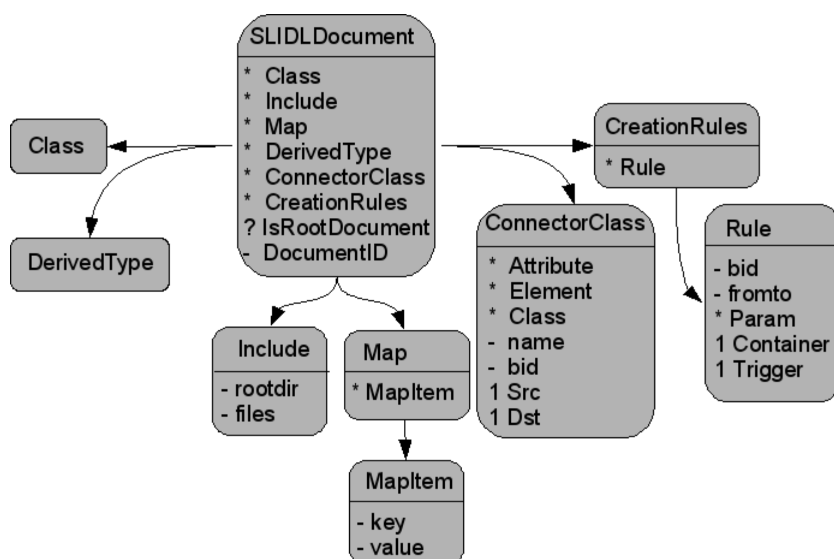


Abbildung 7.3: Das SLIDLDocument

### 7.1.5 SLIDLScript: Auflösung von Laufzeitabhängigkeiten

Es sollen Abhängigkeiten zwischen einzelnen Instanzen von SLIDL-Elementarobjekten auflösbar sein. Dies soll mittels einer minimalen Skriptsprache gewährleistet werden. Diese muss folgende Fähigkeiten bereitstellen:

1. Eine endliche Menge an Funktionen:
   (a) boolsche Funktionen: *if(), eq(), geq(), exists()* . . . etc.
   (b) Zugriffsoperationen auf *Maps*
   (c) Zugriffsoperationen auf Instanzen und deren Daten
   (d) Zugriffsoperationen auf Klassendaten einer Instanz
2. Referenzen auf die aktuelle Instanz und ihren Container: *This* und *Container*

Mit Zugriffsoperationen sind stets lesende Operationen gemeint.

### 7.1.6 ConnectorInterface, ConnectorClass und Rule

Die Abbildungen 7.1 und 7.3 beinhalten Sprachelemente deren theoretische Grundlagen in Abschnitt 5.3 eingeführt wurden. Einerseits das *ConnectorInterface*[4] und dessen *ConnectorItems*, den Konnektionsoperanden. Andererseits die *ConnectorClass*, welche dem Konnektionsobjekt entspricht. Schließlich noch die *Rules* als *Erzeugungsregeln* über Konnektionsoperanden zur Erzeugung von Konnektionsobjekten.

#### 7.1.6.1 Konnektion und inverse Konnektion

Das *fromto*-Attribut einer *Rule* definiert durch Leerzeichen getrennt die *Rolle* des Quellkonnektionsoperanden und des Zielkonnektionsoperanden. Das *conrole*-Attribut eines *ConnectorItems* ist mit einer solchen *Rolle* zu verknüpfen. Einem *ConnectorInterface* können über das Attribut *userules* die Namen von *Rules* angegeben werden, die bei einer Konnektion zur Auswertung in Frage kommen. Wenn nun zwei Konnektionsoperanden verbunden werden, prüft die SLIDL-API deren *Rolle* gegen die Regeln eines *ConnectorInterfaces*. Im Falle einer Konnektion wird beim Greifen einer der Regeln die entsprechende *ConnectorClass* erzeugt. Andernfalls, bei der *inversen Konnektion*, wird für jede passende *ConnectorClass*[5] die Existenz der Elemente *Src* und *Dst* geprüft. Diese können mittels SLIDLScript relativ zum Konnektionsobjekt angegeben werden.

## 7.2 Objektorientiertes Design

Es sei darauf hingewiesen, dass im Zuge eines rückgekoppelten Phasenmodells bezüglich der *Analyse*, des *Entwurfs* und der *Implementierung* nicht jeder Bereich vollständig abgedeckt werden konnte. Als vollständig gilt die *Analyse*, sowie große Teile des *Entwurfs* und der *Implementierung*. Beispielsweise ist die Modellierung der notwendigen Klassen für SLIDLScript ausgeblieben, die notwendige Vorgehensweise zum *Parsing* eines Kommandos oder dem konkreten Zugriff auf Laufzeitdaten wird dafür in einem späteren Abschnitt erläutert.
Nachdem nun alle Sprachkomponenten SLIDLs vorgestellt worden sind, widmet sich der aktuelle Abschnitt der objektorientierten Modellierung der SLIDL-API.

[4] in Teilabschnitt 5.3.5 als *Konnektionsschnittstelle* bezeichnetet

[5] gemeint: für jede *ConnectorClass* deren *name* mit einer gefundenen XML-Struktur übereinstimmt

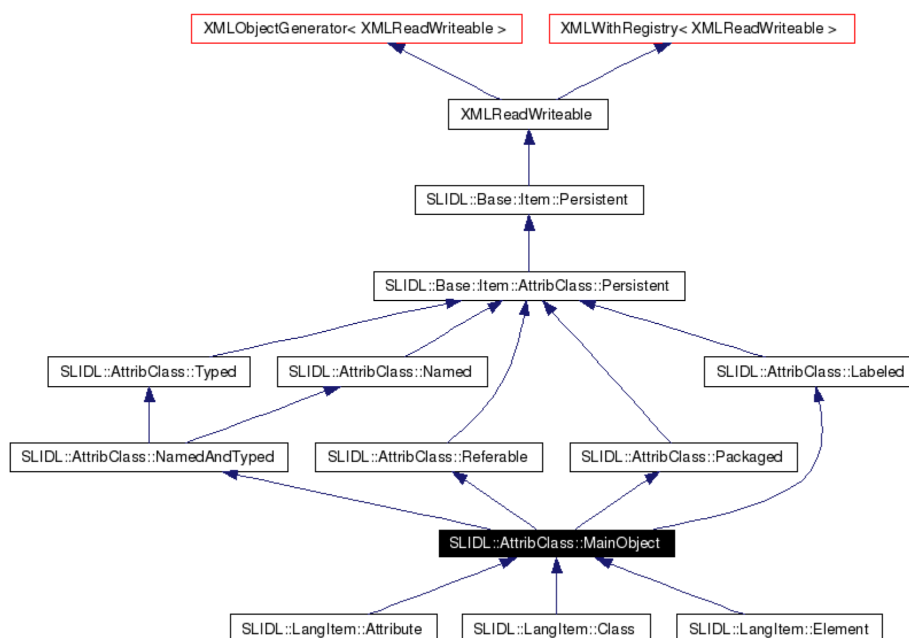


Abbildung 7.4: SLIDL-API: Ausschnitt aus der Klassenhierarchie

### 7.2.1 Attributklassen

Auf Abbildung 7.1 wurde bereits dargestellt, dass gewisse Attribute von unterschiedlichen SLIDL Sprachelementen geteilt werden. Dieser Fakt geht nahtlos in das objektorientierte Design des Deklarations-Layers von SLIDL ein. Auch unter dem Aspekt der XML-Persistenz ist es denkbar jedem gemeinsamen Attribut eines Sprachelements eine eigene Klasse zuzuordnen. Ein Sprachelement wie *Class*, *Element* oder *Attribute* kann demnach von mehreren dieser *Attributklassen* abgeleitet werden. Es erbt damit die Getter- und Setterfunktionen, sowie die persistenten Eigenschaften all seiner *Attributklassen*. Der Begriff *Attributklasse* hat somit Bedeutung in doppelter Hinsicht; einerseits aus Sicht eines SLIDL-Sprachelements in XML und andererseits aus Sicht der Objektorientierung in C++. Abbildung 7.4 verdeutlicht diesen Sachverhalt. Die Klassen *Typed*, *Named*, *Labeled*, *Referable* und *Packaged* sind *Attributklassen* im oben beschriebenen Sinn. Weitere *Attributklassen* sind auf Abbildung 7.5 zu sehen.

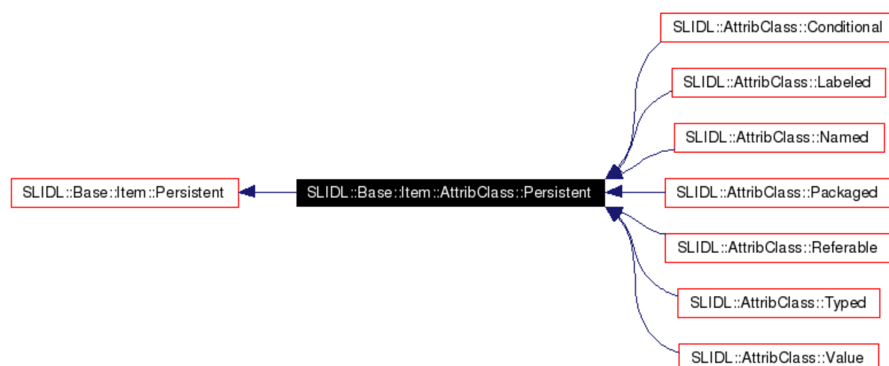


Abbildung 7.5: SLIDL-API: Die Attributklassen

### 7.2.2 Bündelnde Ordnungsklassen

Gemeinsame Attributklassen sind nicht immer Teil aller Sprachelemente. Die genauere Analyse fördert Elemente zu Tage, welche bestimme Eigenschaften teilen und andere nicht. In diesem Sinn können einzelne Attributklassen auf verschiedene Arten *gebündelt* werden. Dies entspricht prinzipiell der Umkehrung der Hierarchie der graphischen Darstellung der Sprachkomponenten. Dabei ist eine *bündelnde Ordnungsklasse* eine Klasse, abgeleitet von mehreren Attributklassen, aber ohne zusätzliche Funktionalität.
Wieder verdeutlicht Abbildung 7.4 diesen Sachverhalt. Die Klassen *NamedAndTyped* oder *MainObject* können als Beispiele für *bündelnde Ordnungsklassen* dienen.

### 7.2.3 Hypervirtuelle Ordnungsklassen

Es ist klar, dass Polyhierarchien zu relativ wirren Gesamtbildern neigen können. Die Klassenhierarchie allein ist demnach nicht mehr in der Lage die logische Struktur des Designs übersichtlich widerzuspiegeln. Darum sind weitere Ordnungsmechanismen notwendig. Eine Technik wird auf Abbildung 7.5 deutlich. Die Klasse $Attribclass :: Persistent$ erfüllen keine andere Funktion, als alle persitenten Attributklassen unter sich zu bündeln. Sie sei als *hypervirtuelle Ordnungsklasse* bezeichnet. Ihre Aufgabe besteht ausschließlich darin, während der Phase des Entwurfs und der Implementierung einen vereinfachten Überblick zu gewährleisten. Bei der abschließenden Kompilation eines Releases der API können solche Klassen durch *typedefs* ersetzt werden.

### 7.2.4 Verwendung von Namensräumen

Wie im letzten Teilabschnitt bereits dargelegt wurde, sind Techniken zur Entzerrung wirr wirkender Polyhierarchien wünschenswert. Ein hauseigenes Mittel von C++ eignet sich besonders gut einer beliebigen Klassenhierarchie mit Mehrfachableitungen ein monohierarchische Struktur aufzuzwingen. Die Rede ist von Namensräumen. In der Modellierung und Implementierung der SLIDL API wurde viel Verwendung von diesem Mittel gemacht. Auf diese Weise wurde parallel eine monohierarchische Ordnungsstruktur über mehrfach abgeleiteten Klassen erzeugt.

### 7.2.5 Initialisierung von Sprachelementen

Über das Sprachdesign und dessen Kongruenz zum objektorientierten Design der SLIDL-API erschließen sich große Teile der Arbeitsweise der Definitionsobjekte. Trotzdem soll kurz die Abfolge von Schritten bei der Initialisierung oder Reinitialisierung der Definitionsobjekte SLIDLs eingegangen werden. Diese geht einher mit der Abfolge von Schritten der Persistenzschicht beim Einlesen von Daten[6].

1. in *preRead()*:
   (a) *Upcast* dynamisch erzeugter Unterobjekte
   (b) Registriere *sprachliche* Unterobjekte bei **XMLRegistry**
2. in *postRead()*:
   (a) *Downcast* dynamisch erzeugter Unterobjekte
   (b) Registriere *this* im **SLIDL::Environment::Env** (der Deklarations-Verwaltung)

[6] siehe Unterabschnitt 5.5.2.3

| Env |
| --- |
| + ~Env( ) :<br>+ setReadInComplete(ClassDocument* rtDoc ) : void<br>+ getRootDocument( ) : ClassDocument *<br>+ getRootClass( ) : Class *<br>+ setRootDocument(ClassDocument* rtDoc ) : void<br>+ setRootClass(Class* rclass ) : void<br>+ setUnknownType(Typed* uktype ) : void<br>+ getClassDocument(string& refID ) : ClassDocument *<br>+ getClassDocument(char* refID ) : ClassDocument *<br>+ getClass(string& refID ) : Class *<br>+ getClass(char* refID ) : Class *<br>+ getClass(ID envID ) : Class *<br>+ getClassNames( ) : StrVec *<br>+ getClassNode(string& refID ) : MainObject *<br>+ getClassNode(char* refID ) : MainObject *<br>+ getType(string& name ) : DType *<br>+ getType(char* name ) : DType *<br>+ getType(ID typeID ) : DType *<br>+ getMapItem(string& mapname, string& keyname ) : string&<br>+ setClass(string& refID, Class* aClass ) : void<br>+ setClassNode(string& refID, MainObject* mObject ) : void<br>+ setClassDocument(string& refID, ClassDocument* cDoc ) : void<br>+ setType(string& name, DType* dType ) : void<br>+ setMapItem(string& mapname, string& key, string& value ) : void<br>+ setDepItem(string& refID, ID depID, Deps::Dependency* depItem ) : void<br>+ rmDepItem(ID depID, Deps::Dependency* depItem ) : void<br>+ rmClass(string& refID ) : void<br>+ rmClassNode(string& refID ) : void<br>+ rmClassDocument(string& refID ) : void<br>+ rmType(string& name ) : void<br>+ checkType(ID typeID, StrVec& valVec ) : bool<br>+ checkType(ID typeID, LIntVec& valVec ) : bool<br>+ checkType(ID typeID, DoubleVec& valVec ) : bool<br>+ checkTypeChoice(ID typeID, StrVec& choice ) : bool<br>+ getChoiceItems(int hiddenID ) : const StrVec *<br>+ getChoiceItems(ID typeID ) : const StrVec *<br>+ hasChoice(int hiddenID ) : bool<br>+ hasChoice(ID typeID ) : bool<br>+ getMObjectName(int hiddenID ) : const string &<br>+ getMObjectLabel(int hiddenID ) : const string &<br>+ getMObjectRefID(int hiddenID ) : const string &<br>+ getMObjectBType(int hiddenID ) : string<br>- Env( ) :<br>- traverseDeps(Deps::Dependency* dItem ) : bool |
| - Typed : SLIDL::AttribClass::Typed<br>+ Handler : class<br>- _rootDocument : ClassDocument*<br>- _rootClass : Class*<br>- _packages : PkgHierarchy |

Abbildung 7.6: SLIDL: UML-Diagramm **SLIDL::Environment::Env**

## 7.3 SLIDL Deklarations Verwaltung

Die Deklarations-Verwaltung ist die zentrale Sammelstelle für alle SLIDL-Sprachelemente. Sie hat ferner Kenntnis von allen SLIDL-Dokumenten, kennt das Root-Element eines Instanzdokuments und bietet eine rein zeichenkettenorientierte Schnittstelle zu allen Definitionsobjekten, deren Daten und deren Attributen. Abbildung 7.6 zeigt das UML-Diagramm der Deklarations-Verwaltung.

### 7.3.1 Auflösung von Typabhängigkeiten

Eine wesentliche Aufgabe der Deklarationsverwaltung ist auch die Auflösung von deklarativen Abhängigkeiten insbesondere bei der Ableitung von Typen oder Klassen. Eine Klasse kann von mehreren Oberklassen abgeleitet werden. Dies entspricht beginnend bei einer Klasse rückwärts aufsteigend einem monohierarchischen Abhängigkeitsbaum. Da die Ableitung von Basistypen nur die einfache Ableitung vorsieht ergibt sich hierfür ein linearer Abhängigkeitsgraph. Die Deklarationsverwaltung muss fehlerhafte Abhängigkeitsgefüge identifizieren und entsprechende Fehler zurückgeben.

Auf Abbildung 7.7 ist der Vorgang der Auflösung von Typabhängigkeiten als Datenflussdiagramm dargestellt. Dieser Vorgang soll näher erläutert werden.

Die Auflösung von Typabhängigkeiten erfolgt in vier Stufen.

1. Während des *postRead*-Prozesses werden alle definierten Struktur- und Elementartypen im Datenspeicher „Derived Types“ abgelegt. Jedes Elementarobjekt, dessen Datentyp in dieser Phase noch unbekannt ist, wird im Datenspeicher „Unknown Types“ abgelegt.

2. Innerhalb der zweiten Phase *putBasesToStack* wird für jeden abgeleiteten Typ der Abhängigkeitsbaum traversiert und auf diese Weise der Datenspeicher „DType Stack“ aufgebaut. Für jedes Stackelement wird geprüft, ob es sich bereits auf dem *Stack* befindet. Sollte dies der Fall sein wird ein *CyclicDependency*-Fehler erzeugt. Dies entspricht einer unerwünschten *zyklischen Abhängigkeit*.

3. Im dritten Schritt *solveDerivationDeps* wird der Stack wieder geleert. Jedes entfernte Objekt erbt dabei die zu erbenden Eigenschaften des zuletzt entfernten Objekts.

So erbt ein *DerivedType* die *Constraints*, die *Choice*, die *size* und/oder die *RegExp* seines Elterntyps. Eine Klasse erbt so alle *Elemente*, *Attribute* und das *ConnectorInterface* aller Oberklassen. Dabei können sowohl Elementartypen, als auch Strukturtypen geerbte Eigenschaften *überschreiben*. Fehler während dieser Phase entsprechen Definitionsfehlern im *SLIDLDocument* und lösen den *DependencyError* aus. Korrekt abgeleitete Typen werden im Datenspeicher „Independent DTypes“ gesammelt.

4. In einem letzten Schritt werden die zu Beginn ausgemusterten Elementarobjekte unbekannten Typs aus dem Datenspeicher „Unknown Types“ gegen die korrekt abgeleiteten Typen aus dem Datenspeicher „Independent DTypes“ geprüft. Alle Elementarobjekte, denen auf diese Weise eindeutig ein Typ zugeordnet werden kann, sowie deren übergeordnete Klassen gehen als Sprachelemente in das *LanguageEnvironment* ein. Sollte die Zuordnung fehlschlagen, so wird ein *UnknownType*-Fehler ausgelöst.

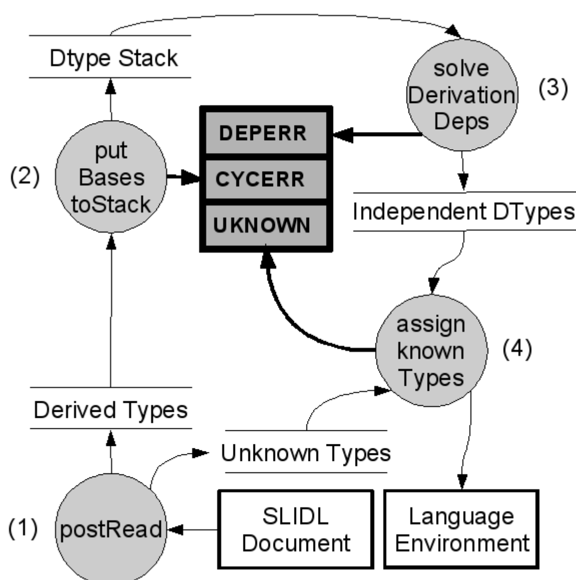


Abbildung 7.7: Auflösen von deklarativen Abhängigkeiten

### 7.3.2 Netzwerkfähigkeit

Es ist bereits dargelegt worden, dass die Deklarationsverwaltung eine zeichenkettenorientierte Schnittstelle nach Außen bereitstellt. Im ursprünglichen Entwurf der SLIDL-API war die Instanz-Schicht nocht nicht so komplex angelegt. Stattdessen war die Konstruktion der Gestalt, dass die Instanzobjekte direkten Zugriff auf Daten der Deklarationsverwaltung nehmen konnten. Der Deklarationsverwaltung oblag somit auch die Aufgabe der Instanzverwaltung. Die Instanzobjekte selbst stellten wiederum eine zeichenkettenorientierte Schnittstelle für ein aufsetzendes GUI dar.

Im Zuge der Implementierung fiel deutlich auf, dass mit geringem zusätzlichen Aufwand die Netzwerkfähigkeit der SLIDL-API gefördert werden kann. Das Ergebnis dieser Überlegung bedingt die Entzerrung des ursprünglichen Modells der Softwarearchitektur, durch die Trennung der Instanzverwaltung von der Deklarationsverwaltung. Die rein zeichenkettenorientierte Kommunikation ließe sich demnach auch über *Sockets* durchführen. Eine zusätzliche Client/Server-Schicht pro SLIDL-Verwaltung ist denkbar. Diese zwei Kommunikationsschichten könnten so über bestehende[7] Protokolle SLIDL-Verwaltungsdaten austauschen. Für eine Einzelplatzlösung mag dies zunächst übertrieben erscheinen, für einen späteren Ausbau zu einem CMS[8] sind somit jedoch bereits die wichtigsten Grundlagen vorhanden.

## 7.4 SLIDL Rückbesinnung GUI

Das GUI kann aus Sicht der SLIDL-API beliebig implementiert werden. Aufgrund des Konzepts einer zeichenkettenorientierten Schnittstelle zu den Instanzdaten und Definitionsobjektdaten ist eine desktopbasierte Lösung nun ebenso denkbar wie eine CGI-basierte Lösung. Dennoch beschränkt die vorliegende Arbeit sich primär auf die Betrachtung des Desktopansatzes.

### 7.4.1 Typabhängige Widgets auf XML Basis

Moderne GUI-APIs stellen in der Regel die Möglichkeit bereit den Aufbau von GUI-Formularen XML-basiert zu realisieren. Ferner können durch die Ableitung bestehender

[7] oder auch eigene
[8] vgl. Teilabschnitt 3.1.2

Widgets spezialisierte Steuerelemente z.B. in Abhängigkeit von Datentypen realisiert werden. Vereint man beide Konzepte, so ist es denkbar XML-konfigurierte Widgets in Bezug zu SLIDL-definierten Struktur- und Elementartypen zu setzen. Da die Implementierung der Instanzschicht nicht, sowie von Randbreichen der Deklarationsverwaltung nicht vollständig abgeschlossen werden konnte, ist es zu keiner Modellierung oder gar Implementierung dieses Vorschlages gekommen.

## 7.5 SLIDL Instanz Verwaltung

Die Instanzverwaltung dient der Speicherung und Synchronisierung von Instanzobjekten. Durch das Laden eines SLIDL-Dokuments werden innerhalb der Deklarationsverwaltung alle Definitionsobjekte und deren Beziehungen umgesetzt. Dies muss ein GUI zur Bearbeitung von SLIDL-Instanzdokumenten aber nicht wissen. Aus seiner Sicht ist auch die Aufspaltung von Elementarobjekten in Attribute und Elemente zu vernachlässigen.
Dieser Umstand geht in die Modellierung und Implementierung der Klassen der Instanzschicht ein. Aus ihrer Perspektive existieren nur *Container* und *Elementarobjekte*. Auf den Abbildungen 7.8 und 7.9 sind die UML Diagramme der entsprechenden C++-Umsetzungen zu sehen. Es sei darauf hingewiesen, dass die Implementierungen der *pre** und *post** nicht realisiert wurden. Später wird dennoch auf die grundsätzliche Vorgehensweise beim Lesen und Schreiben der Instanzdokumente eingegangen.

Die wesentlichen Aufgaben einer Instanzverwaltung sind analog zu den wesentlichen Aufgaben der Deklarationsverwaltung:

1. Speicherung von Definitionsobjekten in Form von *komplexen* oder *einfachen* Instanzobjekten[9]: Auf diese Weise können clientseitig neue Instanzobjekte durch Kopien der *gecacheten* Muster erzeugt werden.

2. Synchronisierung des Definitionsobjekt-Caches: Dies ist notwendig, um die Korrektheit neu erzeugter Instanzobjekte zu gewährleisten.

3. Interne Repräsentation der Hierarchie von Instanzobjekten eines Instanzdokuments: Dies gewährleistet den Zugriff auf Daten von Instanzobjekten durch SLIDLScript.

[9] im Sinne eines *Cache*-Speichers für Definitionsobjekte

4. Kommunikation mit der Deklarationsverwaltung: Die vorhergehenden Punkte basieren alle auf der Kommunikation zwischen Instanz- und Deklarationsverwaltung. Aus Sicht der Instanzobjekte ist die Instanzverwaltung als Schnittstelle zur Deklarationsverwaltung aufzufassen.

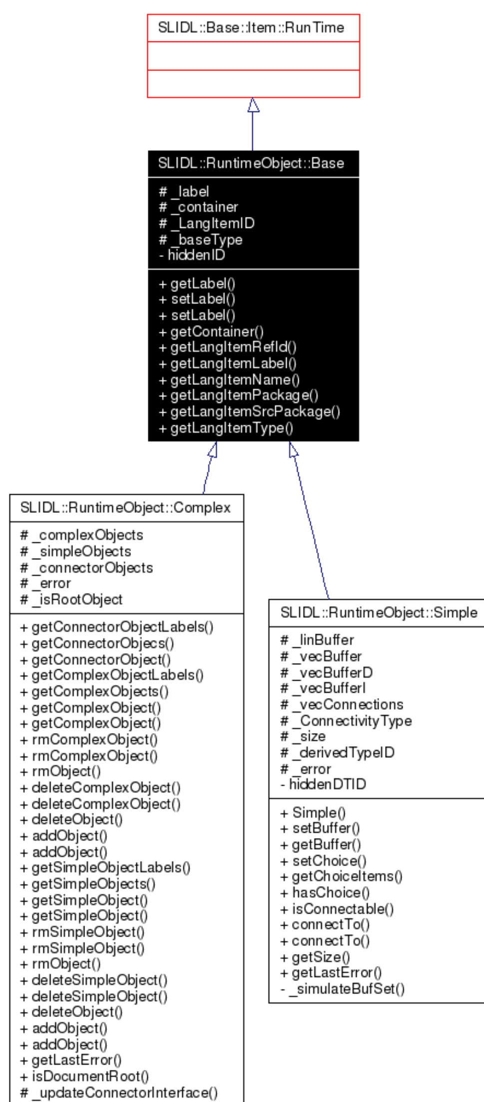


Abbildung 7.8: SLIDL: Die Laufzeit-Objekte (Pseudo-UML)

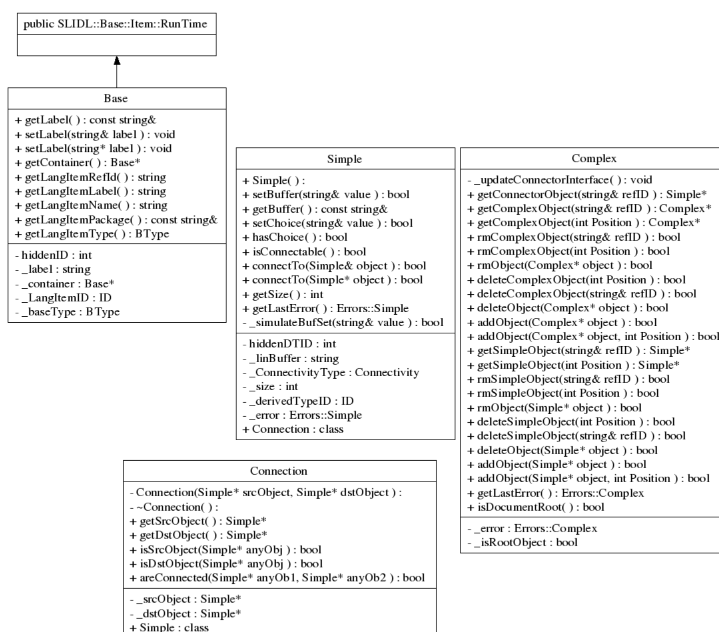


Abbildung 7.9: SLIDL: UML-Digramme: **BaseObject**, **ComplexObject** und **SimpleObject**

## 7.6 ComplexObject und SimpleObject

Die Schnittstelle des GUI zur SLIDL-API ist eine Hierarchie aus Objekten der Klassen *ComplexObject* und *SimpleObject*. Dabei übernehmen *SimpleObjects* zusätzlich die Rolle von *ConnectorItems*[10]. Die *SimpleObjects* sind so gebaut, dass sie einen Puffer für Zeichen bereitstellen. Das GUI muss nur noch mittels *setBuffer(. . . )* versuchen eine Zeichenkette in den Puffer zu schreiben. Der Rückgabewert dieses Aufrufs ist boolsch und kann dem GUI Auskunft über Erfolg oder Misserfolg des Schreibvorgangs geben. Intern leitet ein *SimpleObject* den Prüfvorgang an den entsprechenden *DerivedType* der Deklarationsverwaltung weiter, welcher alle Möglichkeiten besitzt eine Zeichenkette bezüglich seines eigenen Typs zu prüfen.

Ähnlich verhält es sich bei den *ComplexObjects*. Sie sind auch nur *Wrapper* für die Funktionalität der Strukturprüfung, welche von SLIDL-Klassen bereitgestellt wird. Will das GUI ein SimpleObject der Objekthierarchie ein neues *SimpleObject* hinzufügen, so muss es dieses mittels der Instanzverwaltung erzeugen und einem *ComplexObject* übergeben.

[10] also von *Konnektionsoperanden*

Dieses leitet den Prüfvorgang anhand der *bid* des *SimpleObjects* bis zur Deklarationsschicht weiter.

## 7.7 Abhängigkeiten von Daten zur Laufzeit

In Teilabschnitt 7.1.5 wurde bereits darauf hingewiesen, dass SLIDLScript eine Möglichkeit darstellen soll Datenabhängigkeiten zwischen Laufzeitobjekten aufzulösen. Dies soll in diesem Abschnitt vertieft werden.

### 7.7.1 SLIDLScript: Beispiel

SLIDLScript ist rein zeichenkettenorientiert zu verstehen. Bei der Definition einer Klasse können in SLIDL-Attributen *innerhalb eckiger Klammern* Kommandos dieser Sprache eingesetzt werden, welche durch die Rückgabezeichenkette ersetzt werden. Am Beispiel konditionaler Elementarobjekte[11] lässt sich das erläutern:

Sei $X$ die *bid* einer Klasse, die zwei Elementarobjekte mit den *bids* $x$ und $y$ enthalte. Es gelte zusätzlich: $x$ ist nur dann instanziierbar, wenn keine Instanz von $y$ existiert. Die SLIDL-API wertet auf Anfrage zur Generierung einer Instanz von $x$ dessen konditionale Eigenschaften aus, indem sie den Inhalt von *ifnot* und *iftrue* dem Skriptspracheninterpreter übergibt und dessen Rückgabezeichenkette boolsch interpretiert. Entsprechend der Kombination aus *wahr* und *falsch* entscheidet die API schließlich, ob $x$ instanziierbar ist. Der Inhalt des *ifnot*-Attributs von $x$ müsste demnach [exists(Container.y)] enthalten.

### 7.7.2 SLIDLScript: Syntax

Es existieren zwei syntaktisch unterscheidbare Anteile in der Skriptsprache. Einerseits Verkettungen mittels des Punktoperators zur Referenzierung von Instanzen, andererseits verschachtelbare Funktionsaufrufe. Letztere sind semantisch zweifach interpretierbar. Entweder entsprechen deren Bezeichner definierten Funktionen oder aber den Bezeichnern der SLIDL-Attribute eines Elementar- oder Containerobjekts. Der Sprachinterpreter ist als Ein-Kommando-Interpreter konzipiert. Es ist also nicht vorgesehen, dass er Sequenzen von Befehlen ausführt. Versucht man die Syntax beider Kommandoarten als reguläre Ausdrücke zusammenzufassen, so erhält man folgende Zeichenketten:

[11] siehe *ifnot* und *iftrue*

1. **Verkettete Referenzen:**
   K=(Container\.|This\.)+(Container|This|BidOfElementarObject)

2. **Definitionsdatenzugriff und Funktionen:**
   F=(FuncName|DefAttribName)\(ParamList\)

   ParamList=((F,|K,|String,)+)?(F|K|String)

### 7.7.3 Auswertung von Kommandos

Die Auswertung von Kommandos durch den Interpreter von SLIDLScript kann technisch auf Basis von erweiterten regulären Ausdrücken[12] vollzogen werden. Die wesentliche Vorgehensweise ist:

1. Beginne Auswertung...

2. $X$ ist eine verkettete Referenz ?

   (a) Spalte $X$ am Verkettungsoperator (.) in $k$ Zeichenketten.

   (b) Verfolge vom aktuellen Instanzobjekt (This) ausgehend die ersten $k-1$ Referenzen.

   (c) Ist die $k$-te Zeichenkette eine Referenz ?

      i. Ja: Gib *false* zurück.

      ii. Nein: Enthält das aktuell referenzierte Ziel ein Instanzobjekt mit der *bid* der $k$-ten Zeichenkette so gib dessen Inhalt[13] zurück.

3. $X$ ist syntaktisch ein Funktionsaufruf ?

   (a) Existiert eine definierte Funktion die zu $X$ passt ?

      i. Extrahiere alle Funktionsparameter und verfahre mit jedem nach Punkt 1[14].

      ii. Rufe die entsprechende Funktion mit den extrahierten Parameterliste auf.

[12] Es gibt diverse C++-Bibliotheken, die diese Funktionalität beherbergen. Dabei unterscheiden sich reguläre Ausdrücken verschiedener Bibliotheken meist nur wenig voneinander. Eine sehr mächtige Implementierung regulärer Ausdrücke wurde in der Skriptsprache *Perl* realisiert. Diese Abart von Ausdrücken wird von der Bibliothek **PCRE** (**P**erl **C**ompatible **R**egular **E**xpressions) auch für C/C++ bereitgestellt.

[13] als Zeichenkette

[14] Parameterexpansion

(b) Nein: Interpretiere den ersten Parameter als *verkettete Referenz* und prüfe ob der scheinbare Funktionsname des Kommandos einem Definitionsattribut des referenzierten Instanzobjekts zugeordnet werden kann.

## 7.8 Laden und Speichern von Instanzdokumenten

Dem aufmerksamen Leser wird nicht entgangen sein, dass die Modellierung der Instanzobjekte keine Information darüber enthält, ob ein Elementarobjekt ein Element oder ein Attribut ist. Dies beruht auf der Unvollständigkeit des Entwurfs. Diese Information ist speziell für das korrekte Speichern eines Instanzdokuments notwendig. Im Gegensatz zu den Definitionsobjekten können die Instanzobjekte zur Komplierzeit nicht wissen welche Attribute und Elemente sie lesen oder schreiben müssen. Das bedeutet innerhalb der *pre** Methoden muss zur Laufzeit ermittelt werden, welche Elemente oder Attribute zu lesen oder zu schreiben sind. Die Elemente und Attribute einer SLIDL-Klasse operieren dabei selbstverständlich auf dem DOM-Knoten der übergeordneten Klasse.

Abbildung 7.10: SLIDL-API: Klassenhierarchie

# Kapitel 8

# Ergebnis

Das Hauptziel dieser Arbeit ist nicht erreicht worden. Die Aufteilung des Hauptproblems in Teilprobleme führte zur Entwicklung der SLIDL-API. Diese soll im Sinne der Aufgabenstellung eine flexible Definition und Verwaltung von Laufzeitdaten ermöglichen. Sie ist so gebaut worden, dass auf ihr beliebige GUI Typen aufsetzen können. Die Analyse des Kernproblems kann als vollständig angesehen werden. Die Modellierung und Implementierung hingegen sind nur teilweise vollständig.

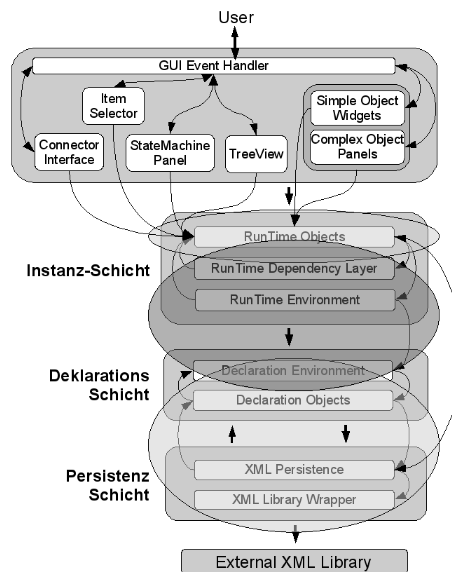


Abbildung 8.1: Implementierungsstand der Architektur

## 8.1 Entwicklung theoretischer Grundlagen

Ein wesentlicher Teil dieser Arbeit wurde der Entwicklung theoretischer Grundlagen gewidmet. Dabei sind in Hinblick auf DRAS-Konfigurationsdateien drei Aspekte besonders berücksichtigt worden:

1. der grundsätzliche Aufbau einer Metasprache zu Beschreibung der semantischen Struktur von XML-Dokumenten
2. die Verallgemeinerung der *Bindung* von DRAS-Systemkomponenten mit dem Resultat der *Konnektion* auf Konfigurationsebene
3. das Design einer *konnektionsmächtigen* Beschreibungssprache für XML

## 8.2 Vollständig abgeschlossene Systemkomponenten

Auf Abbildung 8.1 ist der aktuelle Stand der Implementierung markiert. Dabei deutet eine helle transparente Ellipse an, dass alle von ihr bedeckten Systemkomponenten implementiert sind. Zusammenfassend sind dies:

1. die Schnittstelle zu einer XML-API[1]
2. die Persistenzklasse auf Basis von Memberdatenregistrierungen
3. die Klassenhierarchie der Definitionsobjekte von SLIDL
4. die Klassenhierarchie der Instanzobjekte von SLIDL
5. weite Teile der Deklarationsverwaltung

## 8.3 Konzeptionell abgeschlossene Systemkomponenten

Die dunkle transparente Ellipse deutet an, dass die Modellierung und Implementierung darunter befindlicher Systemkomponenten nicht abgeschlossen werden konnte. Diese Bausteine sind jedoch von Seiten der theoretischen Konzeption als vollständig anzusehen. Darunter fallen:

1. die Implementierung der *pre** und *post**-Methoden der Instanzobjekte[2]

[1] gemeint: die *Xerces* DOM-API
[2] insbesondere die Speicherung und das Laden von Instanzdokumenten ist noch unvollständig

2. die SLIDLSkript Engine
3. die Instanzverwaltung
4. kleine Teile der Deklarationsverwaltung[3]

[3] speziell die Implementierung der Auflösung der Typabhängigkeiten befindet sich zum Zeitpunkt der Abgabe dieser Arbeit in einem instabilen Zustand

# Abkürzungsverzeichnis

**XSD** ........... **X**ML **S**chema **D**efinition
**API** ........... **A**dvanced **P**rogramming **I**nterface
**CGI** ........... **C**ommon **G**ateway **I**nterface
**CMS** .......... **C**ontent **M**anagment **S**ystem
**DOM** .......... **D**ocument **O**bject **M**odel
**DRAS** ......... **D**ominik **R**aus **A**nimations **S**ystem
**FOG** .......... **F**lexible **O**bject **G**enerator
**GUI** ........... **G**raphical **U**ser **I**nterface
**ML** ............ **M**arkup **L**anguage
**PCRE** ......... **P**erl **C**ompatible **R**egular **E**xpressions
**SAX** ........... **S**imple **API** for **X**ML
**SGML** ......... **S**tandard **G**eneralized **M**arkup **L**anguage
**SLIDL** ........ **SL**ang**I**tem **D**efinition **L**anguage
**STL** ........... **S**tandard **T**emplate **L**ibrary
**UML** .......... **U**nified **M**odeling **L**anguage
**URL** .......... **U**nified **R**esource **L**ocator
**XML** .......... **X**tensible **M**arkup **L**anguage

# Literaturverzeichnis

[Bal00] Helmut Balzert. *Lehrbuch der Software-Technik — Software–Entwicklung*. Spektrum Akademischer Verlag, Heidelberg/Berlin, 2 edition, 2000.

[Can95] Georg Cantor. *Beiträge zur Begründung der transfiniten Mengenlehre*. 1895.

[Che66] T.E. Cheatham. The introduction of definitional facilities into higher level languages. *AFIPS, Proceedings of the 1966 Fall Joint Computer Conference*, pages 623–637, 1966.

[Eck00] Bruce Eckel. *Thinking in C++ Volume 2*. http://www.camtp.uni-mb.si/books/Thinking-in-C++, 2 / last updated: 22.02.2000 edition, 2000.

[EM00] E.D.Willink and V.B. Muchnik. An object–oriented preprocessor fit for c++. *IEE Proceedings Software*, pages 49–57, 2000.

[Mül00] Johann-Adolf Müller. *Systems Engineering*. Manz-Verlag Schulbuch (Fortis), Wien, 2000.

[Rau05] Dominik Rau. *Echtzeitanimationssystem für Medienkunst und Infotainment mit einem interaktiven, virtuellen Aquarium als Anwendungsbeispiel*. Diplomarbeit, 2005.

[Som04] Ian Sommerville. *Software Engeneering 7*. PEARSON / Addison Wesley, Boston / New York, 7 edition, 2004.

[unk04] unknown. The scope extension for the c/c++ preprocessor. *www.open-std.org*, pages Reference: ISO/IEC IS 14882:2003(E), ISO/IEC IS 9899:1999(E), 2004.

# Stichwortverzeichnis


Abstraktion, 3, 24, 37
  Abstraktions-Layer, 6
  Daten, 25
  Generalisierung, 5
  Konnektion, 42
  Objekt, 7, 37
Animationssystem, 1
Auszeichnungssprache, 27

C/C++, 46, 77, 79
  Ableitung
    einfach, 66
    Kreuzdelegation, 65
    mehrfach, 65, 66
    mehrfach / multipel, 64
    Schizophrenes Objekt, 63, 66
    virtuell, 65, 66
  C++–Namespace, 61
  Linker, 59
  Mehrfachableitung, 63
  Namensraum, 60, 64
  Objektorientierung, 63
  Pointer, 63
  Präprozessor, 57, 58, 60, 62
    Bedingte Kompilierung, 59
    Berechnete Makros (Computational Macros), 58
    Datei–Inklusion, 59
    Konkatenation von Bezeichner-Namen, 58
    Konvertierung von Namen zu Zeichenketten, 58
    Lexikalische Substitution, 58
    Syntax Makros, 58
    Text Makros, 58
    textuelle Substitution, 59
  Template, 59, 60
    C++–Template, 60
    Cpp–Template, 60
  Typinformation, 64
  Zeichenkette, 46
C/C++–Programme
  Präprozessor, 58
C/C++-Programme, 3

Daten, 26, 28, 37
  Datenhierarchie, 25
  Datenklasse, 38
  Datenobjekt, 37
  Datenobjekt–Modul, 26
  Datentyp, 26, 61
Dokumenttyp, 27
DOM, 28, 29, 45, 46
DRAS, 5, 6, 8, 15–17, 20, 21, 29, 38, 41–43, 45, 46, 71
  Bindung, 42, 43

Event, 42
EventData, 43
Konfiguration, 45, 46
Option, 42, 43
StateValue, 42, 43
DTD, 38

FOG, 62
Meta–Compiler, 62
Meta-Funktion, 61
Meta-Variable, 61

GUI, 9, 41, 42

Hardware, 3
hinreichendes Kriterium, 3

Interaktion, 3

Komplexität, 3, 24
Kontext
Rahmenbedingungen, 3

Markup Language
Auszeichnungssprache, 27
deskriptiv, 27
prozedurale, 27
mehrfach und virtuell, 65

Ordnung
Beziehungsgraph, 7
Fokussierung, 38
Hierarchie, 8
Hierarchiestruktur
Baumstruktur, 28
Mengenlehre
Menge, 24, 25
Objekt
Eigenschaft, 24, 26
Eigenschaftsnähe, 7
Eigenschaftstyp, 7
Objektorientierung, 26
Eigenschaft, 24
Objekt, 26
Ordnungsstruktur, 3, 10, 23, 28, 37
Hierarchiestruktur, 24, 29
Oberflächenstruktur, 29
Struktur, 8, 38
Syntaxstruktur, 39
syntaktische Struktur, 61
System, 6, 24, 25
Softwaresystem, 1

Persistenz, 45, 46
postRead, 49
postWrite, 49
preRead, 49
preWrite, 49

Regulärer Ausdruck, 41
erweitert, 41

SAX, 28, 45, 46
Callback–Funktionen, 28
Ereignis, 28
Parsing, 28
SGML
Meta–Sprache, 27
SLIDL, 38, 39, 69
Attribclass, 77
Labeled, 77

Named, 77
Packaged, 77
Referable, 77
Typed, 77
Attribut, 39, 41, 71, 73, 89
bid, 71
btnocheck, 71
btype, 71
ifnot, 71
iftrue, 71
label, 71
name, 71, 73
package, 71
pkgsrc, 71
value, 71
Attributklasse, 77
Basistyp, 40, 72, 73
abgeleitet, 74
BOOL, 74
DIRPATH, 74
FILEPATH, 74
INTEGER, 74
IXBOOL, 74
REALNUM, 73
STRING, 73
vordefiniert, 73
XBOOL, 74
Bool, XBool, InvXBool, 50
Class, 73
IsRootElement, 73
KeepParentsName, 73
parents, 73
ComplexObject, 86
ConnectorClass, 76
Rule, 76
ConnectorInterface, 72, 76
ConnectorItem, 72
Container, 39, 40, 42, 44, 72, 75
Container–Objekt, 40
Containertyp, 39, 41
Datentypen, 73
Deklarationsschicht, 55
Verwaltung, 55
Deklarationsverwaltung, 81, 83, 84, 86
Definitionsobjekt, 84
DerivedAttribute, 72
DerivedElement, 72
access, 72
DerivedType, 74
Choice, 74
Constraints, 74
RegExp, 74
size, 74
Element, 39, 41, 71–73, 89
cntmax, 71, 72
cntmin, 71, 72
conbtype, 71, 72
conlabel, 71, 72
conrole, 71, 72
Elementarobjekt, 39, 40, 44, 51, 71, 87, 89
Attribut, 72
Attribute, 72
Element, 71, 72
konfiguriert, 45
unkonfiguriert, 45

Elementartyp, 39–41, 73
Ableitung, 40
skalar, 40
vektoriell, 40
Group, 72
Implementierung, 69
Instanzdokument, 89
Instanzobjekt, 89
Instanzschicht, 55
Laufzeitabhängigkeiten, 55, 75
Verwaltung, 55
Instanzverwaltung, 83, 84, 86
Aufgaben, 84
Instanzobjekt, 84
Laufzeitabhängigkeiten, 87
Klasse, 39, 41, 73, 87
Ableitung, 41, 73
Klassenhierarchie, 79
Konnektion, 42, 44, 45, 76
invers, 45, 76
Konnektionsobjekt, 42–44
Konnektionsoperand, 42–44
konnektionsmächtige Beschreibungssprache, 69
Konnektionsobjekt, 76
Konnektionsoperand, 72, 76
extern, 44
Konverter, 44
mit Elementarobjekt–Korrespondenz, 44, 72
ohne Elementarobjekt–Korrespondenz, 44
Map, 75
Namensräume, 79
Netzwerkfähigkeit, 83
Ordnungsklasse, 78
bündelnd, 78
hypervirtuell, 78
Persistenz, 45, 46, 54, 69, 77
Memberdatenregistrierung, 46
XMLBooleanStrings, 51
XMLReadWriteable, 51, 54, 69
XMLRegistry, 46, 51
XMLTools, 51
Schichtenmodell, 47–49
Definitionsschicht, 47
GUI-Schicht, 47
Instanzschicht, 47
Persistenzschicht, 47–49
SimpleObject, 86
SLIDLDocument, 75
SLIDLScript, 75, 87
Definitionsdatenzugriff, 87
Funktionsaufruf, 87
if, eq, geq, exists . . . , 75
Kommandoauswertung, 88
Verkettete Referenz, 87, 88
Sprachanteil
administrativ, 69
deklarativ, 69
Sprachdefiniton, 69, 70
Sprachelement, 77, 79
Initialisierung, 79
Strukturtyp, 39, 73
Ableitung, 41
This, 75

Typabhängigkeiten, 81
Datenflussdiagramm, 81
Software, 3, 23
API, 3, 27, 29, 40, 41
Applikation, 28
Callback–Funktionen, 28
Ereignis, 28
Parsing, 28
Plugin
Pluginarchitektur, 6
Programm, 28
Code, 28
Funktionalität, 9
Kerncode, 6
Programmcode, 4, 26
Programmier–Schnittstelle, 28
Programmieraufwand, 28
Programmierung, 25
Schnittstelle, 6
Softwarearchitektur, 38, 46, 56
Schichtenmodell, 47–49
Softwareergonomie, 8, 10
Bedienungskonzept, 9
GUI, 9
Steuerelement, 9
Softwaretechnik, 11
Software (Softwareergonomie)
GUI, 83
Benutzeroberfläche, 5, 6
Schnittstelle, 6
Widget, 83
SoftwareeInteraktion
Benutzerinteraktion, 3
Softwaretechnik
Analyse
Anforderungsanalyse, 5
Grundlage
Prinzipien, 11
Grundlagen
Methoden und Werkzeuge, 11
Werkzeug, 13
Kapselung, 26
Methoden
Sequenzdiagramm, 65
UML-Diagramm, 48
Phasen, 12
Analyse, 12
Entwurf, 12
Implementierung, 12
Test, 12
Prinzipien
Abstraktion, 11
Generalisierung, 4
Hierarchisierung, 11
Modularisierung, 4, 5, 11
Strukturierung, 11
Trennung der Belange, 5, 11
Wiederverwertbarkeit, 5
Software-System, 11
Softwarearchitektur
Schichtenmodell, 5
System
System–Design, 10
Systemelement, 12
Systemgrenze, 12
Wiederverwertbarkeit, 4

Softwaretechnik (Phasen)
  Analyse
    Anforderungsanalyse, 12, 37
    objektorientierte Analyse, 12, 13
    Systemanalyse, 12, 37
  Entwurf
    objektorientierte Design, 13
    Softwarearchitektur, 13, 37
  Implementierung
    Funktionale Programmierung, 12
    Objektorientierte Programmierung, 12
  Test
    Debug/Testing–Library, 13
    Integrationstest, 13
    Integrationstests, 12
    Low–Level–Test, 13
    Modultest, 13
    Modultests, 12
STL, 59
  STL–Container, 59
Szene, 6
  Szenedatei, 6, 15, 38
    Konfigurationseditor, 17
    Offline–Konfigurationseditor, 15
    Online–Konfigurationseditor, 15
  Szenegraph, 3, 23
  Szeneneditor, 6
  Szeneobjekt, 19

XML, 6, 22, 23, 27, 29–31, 37, 38, 45, 69, 83
  Bausteine
    Attribut, 30, 72
    Element, 30
    syntax–strukturell, 39
  Container–Perspektive, 39
  Gültigkeit, 31
  Namensraum, 31
  Persistenz, 45, 46
  Root-Element, 81
  semantische Struktur, 74
  Sprachspezifikation, 25
  Wohlgeformtheit, 31
  Xerces, 45, 46
  XML–Datei, 17
  XML–Datenobjekt, 38
  XML–Dokument, 27, 29, 38
    Dokumentvolumen, 29
  XML–Editor, 6
  XML–Hierarchie, 28, 29
  XML-Dokument, 42
XSD, 33, 38