Objektorientierter Entwurf eines Applikationsservers

Diplomarbeit
im Fach Informatik

am
Willhelm-Schickard-Institut für Informatik
der Eberhard-Karls-Universität Tübingen
31. August 1998

Volker Simonis
(simonis@@informatik.uni-tuebingen.de)

Wilhelm Schickard-Institut für Informatik

Ich versichere, die Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

St. Petersburg, den 31. August 1998

Dekan:	Professor Dr. U. Güntzer
1. Berichterstatter:	Professor Dr. W. Küchlin

Betreuer bei der debis Systemhaus Engineering GmbH:
Diplom–Informatiker J. Höfler
Dr. W. Suworow

Zusammenfassung

Noch vor wenigen Jahren war es üblich, daß Client-Server Systeme nach dem 2-Schichten Modell realisiert wurden. Dabei handelte es sich meistens um Datenbankbasierte Anwendungen, bei denen sich die Daten samt Datenbank auf einem ausgezeichneten Rechner befanden, die Anwendungslogik jedoch auf Clientseite implementiert wurde. Es hat sich jedoch gezeigt, daß dieser Ansatz zu immer größerer, komplizierterer und kaum noch zu wartender Client-Software, sogenannten *fat clients*, führt. Ein weiteres Problem bei diesen *fat clients* ist der ungewöhnlich hohe Portierungsaufwand auf andere Hard- oder Softwarearchitekturen.

Aus diesem Grund hat sich in letzer Zeit im Client-Server Bereich immer mehr eine 3-Schichten (*three-tier*) Architektur durchgesetzt. Dabei wird die grafische Benutzeroberfläche von der Applikationslogik entkoppelt, indem zwischen Datenbank und Client ein sogenannter Applikationsserver dazwischengeschaltet wird.

Die Aufgabe dieses Applikationsservers besteht darin, das eigentliche Datenmodell vor dem Client zu verbergen. Stattdessen stellt er der Anwendung Dienste auf einer hohen Ebene zur Verfügung, er implementiert jetzt also die Anwendungslogik und stellt sie dem Client über eine spezifizierte Schnittstelle zur Verfügung.

Der Vorteil dieses Vorgehens liegt in der Tatsache, daß die Clientsoftware viel kleiner sein kann als im 2-Schichten Modell. Im Prinzip handelt es sich dabei nur noch um eine grafische Benutzeroberfläche. Neben der geringeren Komplexität des Clients entfallen auch ein Großteil der Wartungsarbeiten an ihm, da die Anwendungslogik im Applikationsserver realisiert ist und hier auch verbessert werden kann, solange die Schnittstelle zum Client nicht geändert wird.

Durch die neue Architektur entstehen jedoch auch neue Schwierigkeiten. So gibt es jetzt zwei Kommunikationskanäle. Erstens vom Applikationsserver zur Datenbank und zweitens vom Applikationsserver zum Client. Während die Kommunikation mit der Datenbank meistens über spezielle, herstellerspezifische Bibliotheken abgewickelt wird, liegt die Implementierung der Kommunikation zwischen Client und Applikationsserver meistens in der Hand des Programmierers. Bisher wurden hier meistens unstrukturierte, streambasierte Protokolle verwendet. Diese skalieren aber schlecht mit der wachsenden Komplexität von Client/Server Applikationen und führen ebenfalls zu immer komplizierterer und schwerer zu verstehender Software.

Das Hauptaugenmerk dieser Diplomarbeit soll deswegen auf die Verbesserung der Kommunikation zwischen Applikationsserver und Client gelegt werden. Insbesondere soll versucht werden, den bestehenden Paradigmenbruch zwischen objektorientierten Anwendungen und zeichenorientierter Kommunikation zu überwinden und die Kommunikationsschnittstelle auf einer möglichst hohen Ebene zu realisieren. Dabei sollen neue Technologien wie entfernte Funktionsaufrufe (*rpc, rmi*), Objekt-Serialisierung (*object serialisation*) und Objekt-Kommunikation (*CORBA*) getestet und eingesetzt werden.

Inhaltsverzeichnis

2. Objektkommunikation mittels Serialisierung

Außer den im letzten Kapitel besprochenen Unzulänglichkeiten hat das NetRequest Protokoll noch einen wesentlichen Nachteil. Es ist nicht möglich, mit ihm komplexe Datentypen der Sprache selber, also Strukturen oder Klassenobjekte direkt zu versenden oder zu empfangen.

Dies würde aber die Client-Server Kommunikation wesentlich vereinfachen, da das für objektorientierte Systeme notwendige, aber fehlerträchtige Ein- und Auspacken der Daten in Objekte automatisiert würde. Desweiteren wäre es wünschenswert, die Kommunikation möglichst Sprach- und Systemunabhängig zu definieren, da sich zum Beispiel der Einsatz von Java im Clientbereich immer mehr abzeichnet, während Applikations- und Datenbankserver aus Performancegründen nach wie vor in C/C++ und auf unterschiedlichen Plattformen wie Unix oder Windows NT implementiert werden.

Für die genannten Anforderungen scheint nun ein System wie CORBA[1] wie geschaffen, ist es doch gerade auf System- und Sprachunabhängigkeit ausgelegt. In der Praxis zeigt es sich jedoch, daß die CORBA-Spezifikation [CORBA] viele Mehrdeutigkeiten enthält und daß die ORB's[2] der unterschiedlichen Hersteller nicht immer zueinander kompatibel sind. Außerdem gibt es bis jetzt noch keine offizielle Sprachabbildung für Java, so daß auch hier die unterschiedlichen Hersteller verschiedene Wege gehen.

Aus diesen Gründen soll hier eine andere Alternative untersucht werden. Und zwar definiert Java, seit der Version 1.1 einen eigenen Mechanismus für entfernte Funktionsaufrufe, die sogenannte *remote method invocation* (RMI [RMI]). Es handelt sich dabei, ähnlich wie bei *remote procedure calls* (RPC [RFC1831, Bloo91]) in C, um die Möglichkeit, Funktionen auf einem entfernten Rechner aufzurufen.

Ähnlich wie RPC die XDR-Spezifikation [3] [RFC1832] verwendet, um komplexe Datentypen in einen Bytestrom zu verpacken, so hat auch Java einen Mechanismus, um Objekte zu serialisieren. Als Serialisierung wird der Vorgang des derartigen Umwandelns eines komplexen Objektes in einen linearen Bytestrom bezeichnet, daß dieses daraus wieder in seinen ursprünglichen Zustand rekonstruiert werden kann.

Das urspünglich ebenfalls von Sun entwicklte RPC erreicht seine Funktionalität, ähnlich wie CORBA, mittels eines Precompilers. In einer eigenen, an C angelehnten Sprache, werden Datentypen und Funktionen definiert. Daraufhin erzeugt ein Precompiler für Funktio-

[1] CORBA : Common Object Request Broker Architekture, ein von der Open Management Group (OMG) definiertes Protokoll zur Objektkommunikation in verteilten Systemen

[2] ORB : Object Request Broker

[3] XDR : Extetrnal Data Representation. Definitionssprache für Datentypen, die in RPC Aufrufen als Argumente verwendet werden können

nen sogenannte *stubs* also leere Stellvertreterfunktionen, die auf Serverseite ausprogrammiert werden müssen, damit sie vom Client aus aufgerufen werden können. Desweiteren werden für alle definierten Datentypen Funktionen generiert, die diesen Datentyp serialisieren und deserialisieren können. Dabei ist es im Prinzip möglich, beliebige C-Verbunde (`struct`) zu serialisieren. Es gibt jedoch zwei Einschränkungen. Erstens handelt es sich hierbei nur um Verbunde und nicht um Objekte, da RPC nur eine Sprachanbindung an C nicht aber an C++ besitzt. Desweiteren können zwar verzeigerte Strukturen generiert und serialisiert werden. Diese werden jedoch nach einer Wertsematik (siehe dazu auch Kapitel 2.1) übertragen, so daß es zum Beispiel unmöglich ist, Selbstreferenzen oder Zyklen korrekt zu behandeln.

2.1 Java Objektserialisierung

Anders als bei C und RPC, wo zu serialisierende Objekte explizit deklariert werden müssen, können in Java im Prinzip alle Objekte standardmäßig serialisiert werden, wobei dies hier nur heißt, daß diese Objekte das Interface `Serializable` oder `Externalizable` implementieren müssen. Bei `Serializable` (Listing 3) handelt es sich dabei nur um ein leeres Interface, es hat nur semantische Bedeutung, indem es signalisiert, daß das Objekt seine eigene Serialisierung zuläßt.

Listing 3 : interface Serializable

```
package java.io;
public interface Serializable {};
```

Des weiteren bildet Java automatisch die transitive Hülle eines zu serialisierenden Objektes bezüglich seiner Referenzen. Das heißt, daß alle von dem gerade serialisierten Objekt aus erreichbaren (also referenzierten) Objekte rekursiv ebenfalls serialisiert werden. Dies kann in Anlehnung an die Kopiersemantik von Objekten und im Gegensatz zum flachen Serialisieren (engl. *shallow serialisation*) als tiefes Serialisieren (engl. *deep serialisation*) bezeichnet werden. Es wird also gegebenenfalls nicht nur ein Objekt alleine, sondern möglicherweise ein ganzer Baum oder Zyklus von Objekten abgespeichert beziehungsweise übertragen. Dies sollte beim Serialisieren großer Strukturen, aus Performancegründen, immer beachtet werden und gegebenenfalls für den Objektzustand unerhebliche Referenzen mit dem Schlüsselwort `transient` als nicht serialisierbar markiert werden.

Dieses Vorgehen hat andererseits aber auch große Vorteile. Wenn zum Beispiel mit baumartigen Strukturen gearbeitet wird, reicht es, die Wurzel zu serialisieren. Beim Deserialisieren wird automatisch der gesamte Baum wieder hergestellt.

Ein weiterer Vorteil des Serialisierungsprotokolles ist, daß Objekte oder Klassendefinitionen nur einmal abgespeichert werden, wobei jedem dieser Objekte eine eindeutige, fortlaufende Nummer[4] (engl. *handle*) zugeordnet wird. Taucht das selbe Objekt im gleichen Serialisierungsstrom (`ObjectOutputStream`) noch einmal auf, dann wird nicht mehr das Objekt selber, sondern nur noch dessen Nummer abgespeichert. Dadurch kann beim

[4] Java unterstützt nur vorzeichenbehaftete Zahlen. Im Serialisierungprotokoll werden jedoch an einigen Stellen (zum Beispiel für Handles und Stringlängen) vorzeichenlose Ganzzahlen verwendet (32bit bzw. 16bit), die es in Java eigentlich gar nicht gibt.

Deserialisieren zum Beispiel aus einem Zyklus wieder ein Zyklus werden.

Allgemein gesagt unterstützt das Protokoll die Referenzsemantik von Java[5]. Im Gegensatz dazu gibt es in C++ sowohl eine Referenzsemantik (beim Einsatz von Zeigern oder Referenzen) als auch eine Wertsemantik bei der Verwendung normaler Objektinstanzen. Die Beschränkung von Java auf Referenzen ist auch der Grund dafür, daß in Java keine Copy-Konstruktoren oder Zuweisungsoperatoren überladen werden müssen. Bei Kopier- oder Zuweisungsoperationen wird immer nur ein Zeiger, aber nicht der Wert auf den er zeigt, übertragen (*shallow copy*).

Listing 4 : Java Referenzsemantik

```
A a;
A b;
a = new A("a");
b = new A("b");
a = b;
// a und b referenzieren das
// selbe Objekt (A("b")), w"ahrend
// A("a") vom garbage collector
// gel"oscht wird.
```

Listing 5 : C++ Wertsemantik

```
A a;
A b;
a = A("a");
b = A("b");
a = b;
// b wird "by value", also byteweise
// nach a kopiert, und dessen Inhalt
// ueberschrieben. a und b enthalten
// untersch. Objekte mit gleichem Wert
```

Listing 4 und 5 verdeutlichen noch einmal die Unterschiede zwischen Werte- und Referenzsemantik. Sie zeigen auch, daß eine vernünftige Abbildung von Java-Objekten nur auf Zeigertypen in C/C++ sinvoll ist, da nur dadurch die Beziehung von Java-Objekten zueinander in C/C++ korrekt widergespiegelt werden kann. Eine Abbildung auf C/C++-Objekte ("*by value*"-Abbildung) ist nur bei den primitiven Datentypen sinnvoll, oder vielleicht noch bei Objekten, die selber nur primitive Datentyen und keine weiteren Objektreferenzen mehr enthalten. Aber schon hier kann durch die Abbildung die Semantik geändert werden, etwa in dem Fall, in dem zwei solcher Java-Objekte, den selben Wert referenzieren, wobei dies danach in C++ nicht mehr der Fall ist. Die Sprachabbildung von Java nach C++ wird in Kapitel 3.2 ausführlich besprochen werden.

Außer der bisher besprochenen impliziten Serialisierung sieht die Java Objektserialisierung aber auch noch die Möglichkeit vor, daß ein Objekt selber Einfluß auf die Art und Weise nehmen kann, auf die es (de)serialisiert wird.

Dafür gibt es zwei Möglichkeiten. Erstens, die Klasse implementiert, wie oben schon beschrieben, das Interface `Serializable` und definiert zusätzlich die beiden Funktionen `writeObject()` und `readObject()`. Listing 6 zeigt die genaue Signatur dieser beiden Funktionen. Sie können dazu verwendet werden, optionale Daten, die den Zustand des Objektes betreffen, zusätzlich in den Serialisierungsstrom zu schreiben oder daraus zu lesen. Sie sind dabei selber für das verwendete Datenformat verantwortlich und im allgemeinen symmetrisch zueinander.

[5] Bei primitiven Datentypen verwendet auch Java eine Wertsemantik

Listing 6 : die Funktionen readObject und writeObject

```
private void writeObject(ObjectOutputStream) throws IOException;
private void readObject(ObjectInputStream) throws IOException, ClassNotFoundException;
```

Die zweite Möglichkeit, ihre Serialisierung zu beinflussen, hat die Klasse, indem sie nicht `Serializable`, sondern das Interface `Externalizable` (Listing 7) implementiert. Dazu muß sie die beiden Funktionen `writeExternal()` und `readExternal()` ausprogrammieren, und ist damit alleine für ihr Abspeichern und Wiederherstellen verantwortlich.

Listing 7 : interface Externalizable

```
public interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput) throws IOException;
  public void readExternal(ObjectInput) throws IOException, ClassNotFoundException;
}
```

2.1.1 Das Java Objektmodell

In Java wird der als *object serialisation* bezeichnete Serialisierungsvorgang dadurch erleichtert, daß alle nichtelementaren Datentypen von der Basisklasse `Object` abgeleitet sind. Desweiteren werden alle Java Programme in einer virtuellen Maschine, der sogenannten *Java Virtual Machine* (JVM) ausgeführt, wobei dieser der Aufbau, also die Variablennamen und -typen, sowie die Funktionssignaturen und -implementierungen aller in ihr gerade aktiven Objekte bekannt ist. Diese Information können auch vom Benutzer abgefragt werden. Mittels der in der Basisklasse `Object` definierten Funktion `getClass()`, kann zu jedem Objekt das entsprechende `Class` Objekt angefordert werden, das gerade in der JVM aktiv ist. Über dieses `Class` Objekt ist dann wiederum neben den oben erwähnten Daten auch zum Beispiel der die Klasse definierende Bytecode erreichbar. Diese Tatsache vereinfacht es natürlich erheblich, eine Objektinstanz in Java zu serialisieren, und ihr Fehlen auf C++-Seite wird die Hauptschwierigkeit bei der Abbildung der Java- auf C++-Objekte sein.

Daß Strings in Java Objekte sind, wurde schon in Kapitel 1.3.2 erwähnt. Aber selbst bei Reihungen (engl. *arrays*) handelt es sich in Java um Objekte die von der Basisklasse `Object` abgeleitet sind. Das scheint zwar auf den ersten Blick widersinnig, gibt es doch unendlich viele Typen von Arrays, wenn bedacht wird, daß sowohl die Dimension als auch der Typ der Einträge variiert werden kann. Somit handelt es sich bei Arrays eigentlich um parametrisierte Typen, vergleichbar etwa mit den Vektoren der STL, die aber wegen des fehlenden Templatemechanismus in Java, einer Spezialbehandlung bedürfen. Deswegen sind Arraytypen in Java auch nicht fest eingebaute Typen, sondern ihr Objekttyp wird zur Übersetzungszeit vom Compiler festgelegt. Dabei gehört die Arraygröße nicht zum Typ des Arrayobjektes. Desweiteren werden Array-Variablen nach der gleichen Referenzsemantik wie alle anderen Objekte behandelt.

Daß Arrays von `Object` abgeleitete Objekte sind hat mehrere Vorteile. Erstens können sie überall dort eingesetzt werden, wo Objektreferenzen verlangt werden, also zum Beispiel in der Java Klasse `Hashtable`. Zweitens können Arrays, da sie Objekte sind, Bereichsprüfun-

gen durchführen, was unumgänglich für Javas Sicherheitskonzepte ist, oder sie können über öffentliche Klassenvariablen wie zum Beispiel `length` nützliche Informationen zur Verfügung stellen.

2.2 Beispiel : Java Objektserialisierung

Bevor das eigentliche Serialisierungsprotokoll genauer analysiert wird, soll ein kleines Beispiel hier erst einmal einen Eindruck von der Funktionsweise der Objektserialisierung vermitteln. Es soll gezeigt werden, daß sowohl Basisdatentypen als auch Objektdatentypen serialisiert werden können und daß die Serialisierung die Topologie verzeigerter Objektstrukturen erhält. Dafür wird in Java eine einfache Klasse `List` definiert, und davon die Klasse `Mist` abgeleitet. Diese Klassen sind in Listing 8 dargestellt.

Listing 8 : ../C++Ser/java1/List.java [Zeile 9 bis 129]

```
class Mist extends List {
// ...
  boolean value_B;
  byte value_b;
  char value_c;
  String sField[] = {"the", "ultimate", "test"};

  public String toString() {
// ...
  }
}

class List implements java.io.Serializable {
// ...
  short value_s;
  int value_i;
  long value_l;
  float value_f;
  double value_d;
  String value_str;

  byte field[][] = { { 0, 1, 2}, {7, 8, 9} };
  List oField[] = new List[3];

  List next;
  public String toString() {
// ...
  }
// ...
}
```

Mit Hilfe dieser Klassen wird nun eine heterogene, zu einem Zyklus verzeigerte Liste konstruiert. Danach werden zwei assoziative Arrays angelegt und mit Werten gefüllt (Listing 9). Dafür wird die Java Klass `Hashtable` verwendet. Im ersten Fall werden Zeichenketten mit Zeichenketten indiziert, im letzteren Objekte der Klasse `List` ebenfalls mit Zeichenketten.

Listing 9 : ../C++Ser/java1/List.java [Zeile 120 bis 166]

```
List list1 = new List();
List list2 = new List();
List list3 = new Mist();

list1.value_i = -177;
list1.value_str = "VHS";
 ...
list1.next = list2;
list2.value_i = 288;
list2.value_str = null;
 ...
list2.next = list3;
list3.value_i = -399;
list3.value_str = "Zenit";
 ...
list3.next = list1;

Hashtable hta = new Hashtable(133, (float)0.8);
hta.put("hallo", "wie geht's");
hta.put("hy", "wie geht's");
hta.put("super", "super");
Hashtable hta1 = new Hashtable(177, (float)0.7);
hta1.put("hallo", list1);
hta1.put("hy", list2);
hta1.put("super", list3);
```

Als nächstes wird die ursprüngliche Wurzel der verketteten Liste und dann deren zweites Element in einen mit einem Byte-Feld initialisierten Serialisierungsstrom geschrieben (Listing 10). Danach folgen der Reihe nach das erste `Hashtable` Objekt, eine Zeichenkette, das zweite `Hashtable` Objekt und schließlich eine Ganz- und eine Fließkommazahl.

Listing 10 : ../C++Ser/java1/List.java [Zeile 169 bis 178]

```
ByteArrayOutputStream o = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(o);
out.writeObject(list1);
out.writeObject(list2);
out.writeObject(hta);
out.writeObject("THIS IS A STRING");
```

Listing 10 : ../C++Ser/java1/List.java [Zeile 169 bis 178] (Fortsetzung)

```
out.writeObject(hta1);
out.writeInt(256);
out.writeDouble(999.999);
out.flush();
```

Listing 11 zeigt schließlich, wie diese Daten wieder aus dem Serialisierungsstrom gelesen werden können. Hier ist die Tatsache interessant, daß der Programmierer selber die Typumwandlung in den gewünschten Zieltyp vornehmen muß (`(Hashtable)in.readObject()`). Dabei handelt es sich jedoch keineswegs um eine statische Typumwandlung (engl. : *static cast*) , wie es die Syntax in Anlehnung an C/C++ vermuten ließe. Java kennt keine *static cast*, stattdessen ist diese Syntax äquivalent zu :

```
Object obj = in.readObject();
if (obj instanceof Hashtable)
  return (Hashtable)obj;
else
  throw new ClassCastException();
```

Das heißt, es wird zur Laufzeit überprüft, ob die gewünschte Typumwandlung möglich ist, und dann und nur dann wird sie auch durchgeführt. In jedem anderen Fall wird eine Ausnahme geworfen. Dieses Verhalten ist zwar langsamer, da erst zur Laufzeit und nicht wie im Falle eines *static cast* bereits zur Übersetzungszeit über die Typumwandlung entschieden wird, es ist jedoch unabdingbar, um die von Java gegebenen Sicherheitsgarantien einhalten zu können. Weil Java allerdings keine parametrisierten Datentypen (engl. : *templates*) unterstützt, konnte die Funktion `readObject()` nur so deklariert werden, daß sie nur einen einzigen Datentyp, nämlich `Object` zurückliefert. Die JVM wäre aber eigentlich in der Lage den richtigen Typ aus dem Kontext zu erkennen und könnte die richtige Typumwandlung selber vornehmen.

Solch eine Möglichkeit zur dynamischen Typumwandlung gibt es in C++ seit der Einführung des Operators `dynamic_cast<Typ>(object)` auch. In Kapitel 4.5 auf Seite 53 wird näher darauf eingegangen werden und es wird sich zeigen, daß er in Wirklichkeit genau einer “normalen” Java Typumwandlung entspricht.

Listing 11 : ../C++Ser/java1/List.java [Zeile 190 bis 207]

```
ByteArrayInputStream i = new ByteArrayInputStream(o.toByteArray());
ObjectInputStream in = new ObjectInputStream(i);
list1 = (List)in.readObject();
list2 = (List)in.readObject();
for (int x=0; x < 2; x++) {
  List rootList = list1;
  while (list1 ≠ null) {
    System.out.println(list1);
    list1 = list1.next;
                                                              ➥
```

Listing 11 : ../C++Ser/java1/List.java [Zeile 190 bis 207] (Fortsetzung)

```
    if (list1 == rootList)
      { System.out.println(" Cycle detected"); break; }
  }
  list1 = list2;
}
System.out.println((Hashtable)in.readObject());
System.out.println((String)in.readObject());
System.out.println((Hashtable)in.readObject());
System.out.println(in.readInt() + "," + in.readDouble());
```

Die Ausgabe des Programmes ist in Abbildung 2.1 zu sehen. Wichtig dabei ist, daß die Liste wieder zu einem Zyklus verzeigert ist. Daß die Daten nur in ein Byte-Feld geschrieben wurden, spielt dabei keine Rolle, der Serialisierungsstrom hätte genausogut mit einem Socket oder einer Datei verbunden werden können.

```
>java List
 List(80cdd5f) : -177,VHS,80cdd75
 List(80cdd75) : 288,null,80cdd92
 Mist(80cdd92) : -399,Zenit,80cdd5f
        Cycle detected
 List(80cdd75) : 288,null,80cdd92
 Mist(80cdd92) : -399,Zenit,80cdd5f
 List(80cdd5f) : -177,VHS,80cdd75
        Cycle detected
 {hallo=wie geht's, hy=wie geht's, super=super}
 THIS IS A STRING
 {hy=List(80cdd75) : 288,null,80cdd92,
  hallo=List(80cdd5f) : -177,VHS,80cdd75,
  super=Mist(80cdd92) : -399,Zenit,80cdd5f}
 256,999.999
```

Abbildung 2.1: Die Ausgabe des Programmes aus Listing 11. Von den `List` Objekten wird dabei jeweils nur ihr Hashwert, ihr `int` und `String` Feld sowie der Hashwert des `next` Feldes durch die überladene `toString()` Funktion ausgegeben.

2.3 Das Serialisierungsprotokoll

In [SerSp] hat Sun den genauen Aufbau des Serialisierungsprotokolles in Form einer Grammatik angegeben. Dies verleitet einen jeden Informatiker dazu, die Arbeit des Dekodierens eines auf diese Weise erzeugten Datenstromes einem Parsergenerator zu überlassen. Dieser Ansatz wird auch hier gewählt werden. Allerdings ergeben sich in diesem speziellen Fall einige Komplikationen und auf die Gründe dafür soll in den nächsten Kapiteln näher eingegangen werden.

2.3.1 Die Serialisierungsgrammatik

In Anhang A wird die gesamte Serialisierungsgrammatik wiedergegeben. Da diese aber ziemlich lang und unübersichtlich ist, werden hier die wichtigsten Produktionen vorgestellt und erläutert.

Zanächst einmal besteht jeder Serialisierungsstrom aus zwei Ganzzahlen[6], einer Identifikationsnummer und einer Versionsnummer.

```
stream:
        magic version contents
magic:
        STREAM_MAGIC
version :
        STREAM_VERSION
```

Danach folgt der eigentliche Inhalt.

```
contents:
        content
        contents content
content:
        object
        blockdata
```

Die Regel `contents` mußte dabei zum Beispiel Linksfaktorisiert werden, um von einem LL(k)-Parsergenerator wie ANTLR[7] [Parr97] akzeptiert zu werden. ANTLR wird in Kapitel 2.4 ausführlicher vorstellt werden, hier nur soviel : es handelt sich dabei um einen Parsergenerator für LL(k) Grammatiken, die syntaktische und semantische Prädikate enthalten können. Er erzeugt daraus gut lesbare, *recursive-descent* Parser [Wirth86] in C oder C++. Die Regel `contents` wurde dabei zu :

```
contents:
        content contents | ()      // () = epsilon
```

Weiterhin besteht der Inhalt des Streams entweder aus einem Objekt (Produktion `object`) oder aus rohen Daten (Produktion `blockdata`). Bei letzteren handelt es sich dabei entweder um primitive Datentypen, die byteweise abgespeichert werden, oder um Daten, die vom Objekt selber mittels der überschriebenen Funktionen des Interfaces `Externalizable` geschrieben wurden.

Rohe Daten werden einfach als Blöcke fester Länge in den Datenstrom geschrieben, und können somit gegebenenfalls beim Deserialisieren übersprungen werden, ohne daß ihr Format bekannt ist.

```
blockdata:
        blockdatashort
        blockdatalong
blockdatashort:
```

[6] Der Java Datentyp ist `final static short`

[7] ANTLR : ANother Tool for Language Recognition. ANTLR ist Teil des PCCTS (Purdue Compiler Compiler Tool Set) von John T. Parr

```
        TC_BLOCKDATA (unsigned byte)<size> (byte)[size]
blockdatalong:
        TC_BLOCKDATALONG (int)<size> (byte)[size]
```

Als nächstes folgt die Beschreibung eines Java-Objekts im serialisierten Bytestrom :

```
object:
        newObject
        newClass
        newArray
        newString
        newClassDesc
        prevObject
        nullReference
        exception
        TC_RESET
```

Dabei handelt es sich bei `TC_RESET` um ein Terminalsymbol. Es versetzt den gesamten Serialisierungsstrom wieder in den Anfangszustand. `exception` setzt den Serialisierungsstrom erst einmal zurück, schreibt das Objekt, das als Ausnahme geworfen wurde und setzt den Strom noch einmal zurück.

```
exception:
        TC_EXCEPTION reset (Throwable)object     reset
```

Eine `nullReference` ist ebenfalls ein Terminalsymbol, da :

```
nullReference   :
        TC_NULL
```

Sie wird überall dort eingesetzt, wo in Java `null` verwendet wird. `prevObject` ist eine Referenz (*handle*) auf ein zuvor schon im Serialisierungsstrom vorgekommenes Objekt. Dies muß aber nicht notwendigerweise ein Java Objekt sein, es kann sich dabei auch um eine Klassenbeschreibung (`newClassDesc`) handeln. Damit wird die in Kapitel 2.1 erwähnte Referenzsemantik ermöglicht, die zum Beispiel Zyklen in der serialisierten Datenstruktur erlaubt. Nebenbei ist es auch noch eine sehr platzsparende Methode, Objekte abzupeichern, da das selbe Objekt nur einmal serialisiert wird, jede weitere Referenz auf es ist einfach eine `prevObject` Produktion.

```
prevObject      :
        TC_REFERENCE (int)handle
```

Ein `newString` ist einfach ein Java String Objekt. Er wird in der Java UTF Kodierung [Flan96] abgespeichert. Interessant dabei ist, daß `newHandle` eine leere Produktion ist, was die Grammatik betrifft, das heißt sie konsumiert keine Zeichen aus dem Eingabestrom, oder schreibt keine in den Ausgabestrom. Dafür hat sie sehr wohl semantische Bedeutung, indem sie dem gerade bearbeiteten Objekt die aktuelle Handlenumer zuweist und diese erhöht. Danach kann zum Beispiel in einer `prevObject` Produktion, sowohl beim Serialisieren als auch beim Deserialisieren, dieses Objekt referenziert werden. Mit `newHandle` wird also nur der internen Referenzzähler eines Objekt-Ein/Ausgabestromes manipuliert, während sie mit `prevObject` tatsächlich geschrieben oder gelesen werden.

```
newString:
        TC_STRING newHandle (utf)
```

Die Produktion `newClassDesc` enthält alle für die Beschreibung einer Klasse notwendigen Daten. Es handelt sich dabei um den Klassennamen, eine die Klasse eindeutig charakterisierende Ganzzahl und die eigentlichen Daten. Die SUID (*Stream Unique IDentifier*) genannte Ganzzahl[8] wird entweder vom Compiler als Hashwert der Klassensignatur selbst erzeugt, oder kann vom Programmierer selber in die Klassendefinition aufgenommen werden. Sie dient der Versionskontrolle von sich ändernden Klassendefinitionen.

```
newClassDesc:
        TC_CLASSDESC className serialVersionUID newHandle classDescInfo
classDescInfo:
        classDescFlags fields classAnnotation superClassDesc
```

Die eigentliche Klassendefinition verbirgt sich hinter der Produktion `fields`. Sie enthält für jedes serialisierte Datenfeld der Klasse einen Eintrag, der den Typ des Feldes als Bytekonstante enthält, dann den Feldnamen und für Objektdatentypen, also für Objekte und Arrays, den Klassennamen. Die Feldnamen sind dabei in alphabetischer Reihenfolge geordnet, wobei erst alle primitiven und danach die Objektdatentypen geschrieben werden. `classAnnotation` sind Daten, die die Klasse in Eigenregie, mittels der Funktion `annotateClass()` schreiben kann. Zum Beispiel wäre es für eine Klasse denkbar, den sie definierenden Bytecode abzuspeichern. Gewöhnlich besteht dieses Feld jedoch aus einer Nullreferenz. Sollte die Klasse serialisierbare Vaterklassen besitzen, so folgt in `superClassDesc` rekursiv deren Definition, ansonsten steht auch hier eine Nullreferenz.

```
fields:
        (short)<count>  fieldDesc[count]
fieldDesc:
        primitiveDesc
        objectDesc
primitiveDesc:
        prim_typecode fieldName
objectDesc:
        obj_typecode fieldName className
```

`newObject` und `newArray` entsprechen schließlich den eigentlich in den Objektstrom geschriebenen oder den aus ihnen gelesenen Objekten. Sie enthalten jeweils ihre eigene Definition im oben beschriebenen Format, und dann die Klassendaten. Das Format und die Anzahl der Daten ergibt sich aus dem `fields` Feld der Klassenbeschreibung.

```
newArray:
        TC_ARRAY classDesc newHandle (int)<size> values[size]
newObject:
        TC_OBJECT classDesc newHandle classdata[]
```

[8] Der Java Datentyp ist `long`

2.3.2 Charakterisierung der Grammatik

In diesem Abschnitt sollen die Schwierigkeiten erläutert werden, die die Serialisierungsgrammatik in sich birgt.

Das erste Problem ergibt sich aus der Tatsache, daß es sich bei den geparsten Daten um einen binären Bytestrom handelt. Daß heißt, daß es unmöglich ist, darauf eine lexikalische Analyse durchzuführen, da keine Metazeichen zur Verfügung stehen. Wird zum Beispiel eine gewöhnliche Programmiersprache geparst, so enthält diese gewöhnlich nur druckbare Textzeichen. Leerzeichen oder Zeilenumbrüche sind Metazeichen, die nicht zur Sprache selber gehören, sondern nur zur Trennung der Sprachelemente (*tokens*) dienen. Sie kommen in den Sprachtokens normalerweise nicht vor. Wenn sie darin vorkommen, wie zum Beispiel Leerzeichen in einer Stringkonstanten, so nur derart, daß dieses Vorkommen durch einen regulären Ausdruck beschrieben und somit durch einen DEA (Deterministischen Endlichen Automaten)[HopUll] erkannt werden kann. Eine lexikalische Analyse ist schließlich nichts anderes, als das Erkennen und Klassifizieren von regulären Ausdrücken durch einen DEA.

Bei dem hier vorhandenen Datenstrom ist es jedoch nicht möglich Tokens zu definieren, da er wie gesagt binäre Daten, also zum Beispiel Fließkommazahlen, enthalten kann. Wird zum Beispiel die Konstante `STREAM_MAGIC = 0xaced` als zwei Byte langes Token definiert, dann kann es vorkommen, daß die Binärkodierung einer Ganzzahl zufälligerweise diese Bytefolge enthält, und die lexikalische Analyse fälschlicherweise `STREAM_MAGIC` erkennt.

Und die Möglichkeit des Vorkommens der Bytefolge von `STREAM_MAGIC` in einem Teil des Datenstroms hat eine ganz andere Qualität als das Auftreten eines Leerzeichens in einer Stringkonstante im oberen Beispiel. Letzteres ist durch einen DEA auflösbar, ersteres aber nicht, da die Information, wann und wie viele binäre Daten im Eingabestrom folgen, kontextsensitiv und somit nicht durch einen regulären Ausdruck beschreibbar ist. Sie kann zum Beispiel eine beliebige Anzahl von Bytes entfernt, in der Klassenbeschreibung eines Objektes enthalten sein. Eine Stringkonstante beginnt dagegen immer mit dem Metazeichen "`"`" und wird von diesem wieder abgeschlossen. Ein DEA benötigt also nur einen neuen Zustand, um Leerzeichen in Zeichenketten anders zu behandeln als außerhalb von ihnen.

Wird, nach obigen Ausführungen, die Grammatik auf ein einziges Token beschränkt (in diesem Fall entspräche also das einzige Token einem Byte), dann ist sie auf jeden Fall nicht mehr LL(1), da die meisten Produktionen eine Vorausschau (*look-ahead*) von mindestens 2 Token benötigen.

Desweiteren ist die Grammatik an sich schon inhärent kontextsensitiv, da eine Produktion wie zum Beispiel :

```
newObject:
        TC_OBJECT classDesc newHandle classdata[]
```

nichts anderes bedeutet, als daß beim Parsen von `classdata[]` Informationen benötigt werden, die beim Parsen von `classDesc`, eine beliebige Anzahl von Eingabezeichen zuvor, erhalten wurden. Die Verwendung einer Symboltabelle wird also unerläßlich.

Völlig unmöglich wird das grammatikgesteuerte Parsen bei den vom Objekt in Eigenregie mittels `writeObject()` geschriebenene Daten. Ihr Inhalt ist nur dem Object selbst bekannt und kann demnach auch nur von ihm gelesen werden. Gerade hier bietet ANTLR jedoch interessante Möglichkeiten, wie das nächste Kapitel zeigen wird.

2.4 Der Parsergenerator ANTLR

Es kann nun gefragt werden, warum der serialisierte Bytestrom überhaupt mittels eines durch einen Parsergenerator generierten Übersetzers erkannt werden soll. Es wäre schließlich auch denkbar, solch ein Programm von Hand zu schreiben. Tatsächlich würde dann aber nichts anderes getan werden, als einen *recursive descent* Parser selber zu schreiben.

Ab einer gewissen Größe wird das aber schnell unübersichtlich. Desweiteren hat die Verwendung eines Parsergenerators noch den weiteren Vorteil, daß es eine saubere Trennung zwischen der "Struktur" der Sprache, also ihrer Syntax und ihrer Implementierung, also ihrer Semantik gibt. Änderungen des Protokolls können somit schneller eingebaut werden , und das ganze Paket wird übersichtlicher und leichter zu warten.

Aus diesen Gründen wird hier der Parsergenerator ANTLR [Parr97] verwendet. Im Vergleich zu YACC [AhoSetUl] handelt es sich dabei aber nicht um einen LR(*1*) sondern um einen LL(*k*) Parser [Brooksh]. Das heißt unter anderem, daß ANTLR eine Vorausschau von *k* Eingabesymbolen erlaubt und im Gegensatz zu YACC Linksableitungen erzeugt. Desweiteren bietet er aber noch weitere Vorteile, auf die hier kurz im einzelnen eingegangen werden soll.

Wie schon in Kapitel 2.3.2 erwähnt, kann für die Grammatik nur ein einziges Zeichen als Token verwendet werden, da es sich bei den Eingabedaten um binäre Daten handelt. Dies erfordert jedoch beim Parsen eine Vorausschau von mehreren Zeichen, um auch aus mehreren Zeichen bestehende Terminale erkennen zu können und somit auch einen Parser, der dies zuläßt.

Stroustrup schreibt in [Strou94] "I tried to use YACC (an LALR(1) parser generator) for the grammer work, and was defeated by C's syntax." Mit ANTLR wurde jedoch von NeXT ein C/C++ Compiler auf Basis der ANSI C++ Grammatik [ANSI-CPP] entwickelt[9]. Dies ist vor allem der Fähigkeit von ANTLR zu verdanken, sowohl syntaktische als auch semantische Prädikate zu unterstützen. Bei semantischen Prädikaten handelt es sich um Ausdrücke, die mit einer beliebigen Produktion assoziiert werden können. Sie werden erst dynamisch zur Laufzeit ausgewertet und entscheiden dann darüber, ob eine Produktion ausgeführt wird oder nicht. Typischerweise handelt es sich dabei um Zugriffe auf eine globale Symboltabelle. Beim Parsen serialisierter Objekte werden semantische Prädikate intensiv eingesetzt, um entsprechend des Wertes der nächsten Token, die ja alle den gleichen Typ haben, weiterzuparsen.

Syntaktische Prädikate erlauben es, als Bedingung für die Ausführung einer bestimmten Produktion beliebig viele Folgezeichen im Eingabestrom vorauszulesen und zu überprüfen. Dies ermöglicht es einem LL(*k*) Parser, auch Produktionen zu erkennen, die eine beliebig weite Vorausschau benötigen. Dies Merkmal wird hier jedoch nicht eingesetzt, da sowieso nur mit einem einzigen Token gearbeitet wird und somit syntaktisch nicht viel entschieden werden kann.

Neben diesen sprachtheoretischen Vorteilen bietet ANTLR aber auch noch viele praktische Vorteile. So erlaubt er für jede Regel beliebig viele Argumente und Rückgabewerte, wobei es sich dabei wiederum um beliebige C++ Datentypen handeln kann. Weiterhin erzeugt er als Ergebnis eine C++ Klasse die für jede in der Grammatik angegebene Produktion eine Methode enthält. Dabei können die auszuführenden Aktionen in der Grammatik selber in C++ programmiert werden.

[9] http://www.empathy.com/pccts/index.html

Dadurch ist es möglich, von der zu einer Produktion gehörigen Aktion aus eine andere Produktion aufzurufen, indem einfach der Name dieser Produktion als Funktionsname verwendet wird. Dies ändert natürlich die Grammatik selber, da die Produktionen nicht mehr fest vorgegeben sind, sondern zur Laufzeit dynamisch geändert werden können. Andererseits ermöglicht es das elegante Parsen von solch inhärent kontextsensitiven Sprachen, wie zum Beispiel dem vom Serialisierungsprotokoll erzeugten Datenstrom.

Als Beispiel hierfür kann wieder die Definition eines Objektes herangezogen werden. Sie lautet, etwas vereinfacht :

```
object:
        newObject
        . . .

newObject:
        TC_OBJECT classDesc newHandle values

values: // The size and types are described by the
        // classDesc for the current object
```

Über den Inhalt von `values` wird in der Grammatik keine Aussage gemacht. Die Spezifikation enthält nur den Hinweis, daß hier die Daten eines Objektes in der alphabetischen Reihenfolge ihrer Bezeichner folgen, wobei die Objektdatentypen vor den primitiven Datentypen geschrieben wurden. Es ist zu beachten, daß es sich dabei wieder um komplette Objekte mit Klassendefinitionen und wiederum eigenen Daten handeln kann. Diese unscheinbare Produktion enthält also den eigentlichen Rekursionspunkt des Protokolls, der allerdings kontextfrei nicht beschrieben werden kann.

An dieser Stelle können nun die Vorzüge des erzeugten *recursive descent* Parsers ausgenutzt werden. In der ANTLR Definitionsdatei handelt es sich bei `values` um die weitaus längste Produktion. Und sie enthält je nach den ihr übergebenen Argumenten, wobei es sich um Referenzen auf zuvor aufgebaute Datenstrukturen handelt, Funktionsaufrufe wie `object()` oder `values()`. Die Produktion ist also keineswegs leer, ihre rechte Seite wird aber erst dynamisch zur Laufzeit zusammengesetzt. Es wird also in Wirklichkeit eine Grammatik für eine kontextsensitive Sprache definiert, das ganze allerdings im Gerüst einer LL(k) Grammatik.

Insbesondere wird es dadurch auch möglich, an Stellen, an denen die Grammatik gar nicht definiert ist, wie zum Beispiel bei den von `writeObject()` geschriebenen Daten, die Kontrolle an das lesende Objekt selber zu übergeben, welches im Gegenzug natürlich eine entsprechende `readObject()` Function definiert haben muß. Von dieser Funktion aus kann das Object wiederum beliebige Funktionen und damit Produktionen des Parsers aufrufen (siehe dazu Kapitel 4.6.1, auf Seite 57). Es muß noch einmal verdeutlicht werden : Die Syntax mit der der Serialisierungsstrom gelesen wird steht nicht fest. Dieser besteht zwar aus wohldefinierten Abschnitten von Klassenbeschreibungen und Daten. Die Reihenfolge und Anordung dieser Abschnitte läßt sich jedoch nicht immer aus dem Bytestrom selber ablesen und kann darum auch nicht vollständig durch eine kontextfreie Grammatik beschrieben werden.

Weitere nützliche Eigenschaften, auf die hier nicht weiter eingegangen werden soll sind unter anderem die eingebaute lexikalische Analyse, die Möglichkeit Produktionen in EBNF anzugeben und die Fehlerbehandlung mittels C++ Exeption-Handling.

3. Die Sprachabbildung

In diesem Kapitel soll auf die Sprachabbildung von Java auf C++ und von C++ auf Java eingegangen werden. Dabei handelt es sich durchaus um zwei verschiedene Dinge. Während die Abbildung in eine Richtung trivial sein kann, ist sie in die andere vielleicht überhaupt nicht möglich. Um aber die Objektkommunikation mittels Serialisierung effektiv einsetzten zu können, ist hier nur eine Beschränkung auf den kleinsten gemeinsamen Nenner sinnvoll, da im allgemeinen eine symmetrische Kommunikation benötigt wird. Das heißt, daß es zum Beispiel nicht ausreicht, wenn in C++ zwar Arrays gelesen werden können, die von Javaseite aus serialisiert wurden, jedoch C++-Arrays sich nicht so serialisieren lassen, daß sie auch auf Javaseite gelesen werden können.

Dieser Serialisierungsmechanismus wurde jedoch nicht hier entworfen und er war auch nicht für die heterogene Kommunikation zwischen Java und C++ gedacht. Deswegen ist es klar, daß die Hauptschwierigkeiten, sowohl beim Lesen als auch beim Schreiben auf der C++ Seite liegen werden. Die nächsten beiden Abschnitte werden sich demnach mit diesen spezifischen Problemen befassen.

3.1 Von Java nach C++ - ein Beispiel

Bevor die Sprachabbildung von Java- auf C++-Objekte im Einzelnen erkläutert wird, zunächst einmal ein Beispiel, wie das Lesen serialisierter Daten von C++ aus in der Praxis funktioniert. Dafür werden wieder die in Listing 8 dargestellten Klassen verwendet. In das Programm werden außerdem die in Listing 12 gezeigten Zeilen hinzugefügt, die bewirken, daß die serialisierten Daten in eine Datei geschrieben werden, deren Name dem Programm als optionaler Parameter übergeben werden kann.

Listing 12 : ../C++Ser/java1/List.java [Zeile 181 bis 187]

```
if (args.length == 1) {
  try {
    FileOutputStream file = new FileOutputStream(args[0]);
    file.write(o.toByteArray());
    file.close();
  } catch (Exception ex) { ex.printStackTrace(); }
}
```

Es wäre dabei genausogut möglich, die Daten in einen von einem Socket kreierten Ausgabestrom zu schreiben, indem dieser von Anfang an anstatt des Bytefeldes benutzt wird,

oder indem einfach der Inhalt des Bytefeldes in diesen geschrieben wird.

Als nächstes folgt jetzt in Listing 13 ein C++ Programm, das die Daten aus dieser Datei lesen und korrekt interpretieren kann. Wiederum ist zu beachten, daß der `ObjectInputStream` anstatt mit der Standardeingabe `cin` auch mit einem mit einer Netzwerkverbindung gekoppelten Eingabestrom verbunden werden könnte.

Listing 13 : ../C++Ser/serReadWrite.cpp [Zeile 11 bis 52]

```
ObjectInputStream objStream(cin);

List *list, *list2, *rootList;
objStream.readObject(list);
List *writeList1 = rootList = list;
objStream.readObject(list2);
List *writeList2 = list2;

for (int x=0; x < 2; x++) {
  while (list) {
    if (Mist *m = dynamic_cast<Mist*>(list)) cerr << *m << endl;
    else cerr << *list << endl;
    list = list->next;
    if (list == rootList) { cerr << " Cycle detected\n"; break; }
  }
  list = list2;
  rootList = list2;
}

typedef map<string, string*> strStrMap;
Init__Object<GenericHTWrapper<strStrMap> > strStrHT;
strStrMap *hta;
objStream.readObject(hta);
for (strStrMap::iterator pos = hta->begin(); pos != hta->end(); ++pos) {
  if (pos == hta->begin()) cerr << "{"; else cerr << ", ";
  cerr << pos->first << "=" << *pos->second;
}
cerr << "}\n";

string *str;
cerr << *objStream.readObject(str) << endl;

typedef map<string, List*> strListMap;
Init__Object<GenericHTWrapper<strListMap> > strListHT;
strListMap *hta1;
objStream.readObject(hta1);
for (strListMap::iterator pos1 = hta1->begin(); pos1 != hta1->end(); ++pos1){
  if (pos1 == hta1->begin()) cerr << "{"; else cerr << ", ";
                                                                    ➥
```

Listing 13 : ../C++Ser/serReadWrite.cpp [Zeile 11 bis 52] (Fortsetzung)

```
  cerr ≪ pos1→first ≪ "=" ≪ *pos1→second;
}
cerr ≪ "}\n";
cerr ≪ objStream.readDouble() ≪ "," ≪ objStream.readInt() ≪ endl;
```

Dazu mußten in C++ folgende Klassen definiert werden :

Listing 14 : ../C++Ser/List.hpp [Zeile 11 bis 54]

```
class List : public Serializable {
public:
  short value_s;
  int value_i;
  Long value_l;
  float value_f;
  double value_d;
  string *value_str;

  vector<vector<char> > *field;
  vector<List*> *oField;

  List *next;

// ...
};
ostream& operator≪(ostream &out, List&);
// ...

class Mist : public List {
public:
  bool value_B;
  char value_b;
  char value_c;
  vector<string*> *sField;
// ...
};
ostream& operator≪(ostream &out, Mist&);
```

Bei dem Vergleich der Ausgabe des Programmes in Abbildung 3.1 mit denen des Java-Gegenstückes aus Abbildung 2.1 auf Seite 16 fällt auf, daß sowohl auf Java-Seite als auch auf C++-Seite wieder zwei Zyklen entstehen. Bemerkenswert ist weiterhin, daß die Hashtable auf C++ Seite, genau wie diejenige auf Java Seite, Referenzen auf die zuvor geschriebenen `List` Objekte enthält. Es wurden also keine neuen Objekte angelegt und mittels Kopierkonstruktor initialisiert.

```
>serReadWrite.exe < java1/serial.out
 List(0x80a6f20) : -177,VHS,0x80a6f70
 List(0x80a6f70) : 288,(nil),0x80a6e08
 Mist(0x80a6e08) : -399,Zenit,0x80a6f20
        Cycle detected
 List(0x80a6f70) : 288,(nil),0x80a6e08
 Mist(0x80a6e08) : -399,Zenit,0x80a6f20
 List(0x80a6f20) : -177,VHS,0x80a6f70
        Cycle detected
 {hallo=wie geht's, hy=wie geht's, super=super}
 THIS IS A STRING
 {hallo=List(0x80a6f20) : -177,VHS,0x80a6f70,
  hy=List(0x80a6f70)    : 288,(nil),0x80a6e08,
  super=List(0x80a6e08) : -399,Zenit,0x80a6f20}
 999.999,256
```

Abbildung 3.1: Die Ausgabe des Programmes aus Listing 13. Von den `List` Objekten wird dabei jeweils nur ihre Adresse, sowie ihr `int`, `String` und `next` Feld durch die überladenen Ausgabeoperatoren ausgegeben.

3.1.1 Auswertungsreihenfolge von Postfixausdrücken und Funktionsargumenten

Ein weiterer interessanter Punkt in dem obigen Beispiel ist die Tatsache, daß die beiden Zahlen in dem C++ Programm "scheinbar" in der falschen Reihenfolge gelesen werden.

```
cout ≪ objStream.readDouble() ≪ ", " ≪ objStream.readInt() ≪ endl;
```

Es werden aber die richtigen Ergebnisse ausgegeben. Dies liegt daran, daß trotz der Linksassoziativität des "<<" Operators die Ausdrücke im obigen Befehl wegen der Operatorüberladung in umgekehrter Reihenfolge ausgewertet werden. Für den Serialisierungsstrom ist es jedoch entscheidend, daß die Daten in der umgekehrten Reihenfolge ausgelesen werden, in der sie geschrieben wurden, also erst die Ganzzahl und dann die Fließkommazahl. Genau das bewirkt obiger Befehl auch, nur daß er die Zahlen in umgekehrter Reihenfolge ausgibt. Um die gleiche Ausgabe wie in der Java-Version zu erhalten, muß obige Programmzeile in zwei Anweisungen aufgebrochen werden.

Dies empfielt sich ohnehin, da die obigen Probleme nur daher rühren, daß in C++ die Reihenfolge der Auswertung von Funktionsargumenten nicht spezifiziert ist. Das gleiche gilt für die Auswertung von Postfixausdrücken, in die die obige Zeile dank Operatorüberladung umgewandelt wird. Der Übersetzer erzeugt also einen Ausdruck der Form :

```
cout.operator≪(objStream.readDouble()).operator≪(", ").operator≪(
  objStream.readInt()).operator≪(endl);
```

der schematisch als

```
obj-expr.function(arg1, arg2).function(arg3, arg4)
```

dargestellt werden kann. Dabei wird vorausgesetzt, daß die Mitgliedsfunktion `function` ein Objekt oder eine Objektreferenz des gleichen Types wie `obj-expr` zurückliefert. Dies ist zum Beispiel bei den Ein-/Ausgabeoperatoren der Standard C++-Bibliotheksklassen `i/ostream` der Fall.

Jetzt ist jedenfalls nach dem C++ Standard [ANSI-CPP](5.2.2) nicht festgelegt, ob zum Beispiel `arg3` vor oder nach `arg4` ausgewertet wird und desweiteren, und das ist in diesem Fall besonders wichtig, ob für den Postfixausdruck `obj-expr.function(arg1, arg2)` erst `obj-expr` oder erst die Argumente von `function` berechnet werden. Und eben diese Tatsache führt in dem obigen Beispiel mit dem Ausgabeoperator dazu, daß zwar zuerst die Fließkommazahl und danach die Ganzzahl ausgegeben wird, daß aber trotzdem der Funktionsaufruf zum Lesen der Ganzzahl vor dem zum Lesen der Fließkommazahl stattfindet. Da aber beide Aufrufe voneinander abhängig sind, da sie das gleiche Objekt ändern, ist die Reihenfolge entscheidend.

Zwar werten alle hier getesteten Compiler (Visual C++ 5.0, Borland C++ 3.5, egcs1.02/g++, EDG[1]-C++) die Funktionsargumente von rechts nach links aus, dagegen gibt es bei den Postfixausdrücken schon Unterschiede. Während EDG-C++ erst `obj-expr` und danach die Funktionsargumente auswertet, machen dies alle anderen Übersetzer in genau der umgekehrten Reihenfolge. Spätere Tests haben sogar gezeigt, daß diese Reihenfolge nicht nur vom Compiler sondern auch von der Plattform abhängig sein kann. So erzeugt egcs1.03 unterschiedlichen Code für Linux und HPUX 10.20. Das heißt also, daß nicht mehrere Aufrufe von `ObjectInputStream` in einem Ausdruck oder einem Funktionsaufruf verwendet werden sollten.

3.2 Die Abbildung von Java- auf C++ Datentypen

3.2.1 Die primitiven Datentypen

Die Abbildung der primitiven Java-Datentypen auf C++-Datentypen erfolgt auf kanonische Weise. Da die Java-Syntax sehr von C/C++ inspiriert ist, verwundert es nicht, daß auch die primitiven Datentypen größtenteils identisch sind. Dabei stellt Java bis auf `char` nur vorzeichenbehaftete Typen zur Verfügung. Den einzigen größeren Unterschied gibt es bei `char`, bei dem es sich in Java um einen 16-bit Wert handelt, der Unicode-Zeichen kodiert. Vorläufig verwendet aber noch praktisch jede Java-Software nur die ersten 255 Zeichen des Unicode-Zeichensatzes, so daß eine Abbildung von Java-`char` nach C++-`unsigned char` vernünftig erschien. Später könnte jedoch hier die Umstellung auf den neuen C++-Datentyp `wchar_t` in Erwägung gezogen werden.

Tabelle 3.1 zeigt die genaue Abbildung. Schwierigkeiten gibt es noch mit dem Typ `long`, der nicht auf allen C++-Systemen 64 Bit lang ist. Gewöhnlich ist aber ein 64-bit Typ mit anderem Namen vorhanden, so daß man sich mit einigen `typdef`-Anweisungen behelfen kann.

3.2.2 Die Objektdatentypen

In Listing 13 auf Seite 24 fällt zunächst einmal auf, daß ausschließlich Zeigerdatentypen verwendet werden. Dies hat, wie schon in Kapitel 2.1 erwähnt, gute Gründe. Danach handelt es sich schließlich bei allen Java Objekttypen um Referenzen, und diese lassen sich, soll die Topologie des Objektnetzwerkes beibehalten werden, nur auf C++ Zeiger abbilden.

Die Verwendung von Zeigern ist aber auch aus anderen Gründen sinnvoll und notwendig. Zum Beispiel wird mittels der Anweisung `objStream.readObject(list)` wobei `list` ein

[1] EDG : Edison Design Group, Hersteller eines C++ Compilerfrontends für C++, (www.edg.com)

Abbildung prim. Datentypen von Java nach C++				
Java		C++		
Typ	Größe	Typ	Größe	Alternative
boolean	1 Bit	bool	--	
char	16 Bit	unsigned char	8 Bit	wchar_t
byte	8 Bit	char	8 Bit	
short	16 Bit	short int	--	
int	32 Bit	int	--	
long	64 Bit	long	--	long long[2] _int64[3]
float	32 Bit	float	--	
double	64 Bit	double	--	long double

Tabelle 3.1: Primitive Datentypen in Java und C++

Zeiger auf ein Objekt vom Typ `List` ist, ein Objekt aus dem Eingabestrom gelesen, danach ein diesem gelesenen Objekt entsprechendes neues C++ Objekt auf der Halde erzeugt und schließlich ein Zeiger dieses neu erzeugten Objektes an `list` zugewiesen. Wichtig dabei ist, daß das so gelesene Objekt zum Beispiel auch vom Typ `Mist` sein kann. Über einen entsprechenden `dynamic_cast<Mist*>(list)` kann jederzeit ein Zeiger auf das vollständige Objekt erhalten werden.

Dieses Verhalten ist im übrigen absolut identisch zu dem in Java, wo die Funktion `read.Object()` immer eine Referenz auf ein Objekt vom Typ `Object` zurückliefert, das danach erst einmal in den gewünschten Zieltyp umgewandelt werden muß. Siehe dazu auch die Ausführungen zu Typumwandlungen in den Kapiteln 2.2 und 4.5. Allerdings gibt es in C++ den Vorteil, parametrisierte Funktionen verwenden zu können, und um genau solch eine handelt es sich bei der Funktion `readObject()` auch. Sie ist folgendermaßen deklariert :

```
template <class T> T* readObject(T*(&)) throw (ClassCastException);
```

Somit wird sie für jeden Argumenttyp neu instantiiert und kann als Ergebnis dementsprechend auch immer den korrekten Typ zurückliefern. Die Definition der Funktion ist in Listing 15 zu sehen. Im Prinzip wird die ganze Arbeit des Dekodierens des Bytestromes dem von ANTLR erzeugten Parsermodul überlassen und am Ende nur noch versucht, das Ergebnis mittels eines `dynamic_cast` in den gewünschten Zieltyp umzuwandeln. Ebenso wie beim Javapendant wird im Falle eines Mißerfolges eine Ausnahme geworfen.

Listing 15 : ../C++Ser/ObjectInputStream.cpp [Zeile 78 bis 85]

```
template <class T>
T* ObjectInputStream::readObject(T* (&obj)) throw (ClassCastException) {
  Serializable* tmp = serialParse→object()→getSerializable();
  if (tmp == NULL) return (T*)NULL;
                                                                    ➥
```

[3] Diesen nicht Standardkonformen Datentyp gibt es auf vielen Unixplattformen und GNU-Entwicklungsumgebungen

[3] Ein 64-Bit Typ des Microsoft Compilers

Listing 15 : ../C++Ser/ObjectInputStream.cpp [Zeile 78 bis 85] (Fortsetzung)

```
  obj = dynamic_cast<T*>(tmp);
  if (obj ≠ NULL )return obj;
  else throw ClassCastException();
}
```

Würde aber hier auf C++ Seite der `List` Zeiger dereferenziert werden und sein Wert einem List Objekt zugewiesen werden, dann wären die zusätzlichen Daten einer etwaigen weiter abgeleiteten Klasse unwiederbringlich verloren. Eine solche Funktion, die deserialisierte Objekte einem Objekt und nicht einem Zeiger zuweist, wurde zwar zu Versuchszwecken im Laufe der Entwicklung tatsächlich geschrieben (Listing 16). Sie hat aber mehrere Nachteile. Erstens kann das vom Parser erzeugte Objekt nicht wieder freigegeben werden, da es möglicherweise im Eingabestrom noch referenziert wird. Zweitens wird zum Kopieren des Objektes der Zuweisungsoperator aufgerufen. Sollte dieser nicht überladen sein, dann wird einfach eine flache Kopie des Objektes erzeugt. Danach würden zwei Instanzen existieren, die in den selben Zyklus zeigen. Alle daraufhin eingelesenen Objekte, die diesen Zyklus referenzieren, haben jedoch keine Beziehung mehr zu dem gerade gelesenen Objekt, bei dem es sich nur um eine Kopie eines Objektes aus diesem Zyklus handelt. Sollte der Zuweisungsoperator dagegen überladen sein und eine tiefe Kopie des Objektes machen, dann würde es von jeder Datenstruktur möglicherweise mehrere Kopien im Speicher geben. Da sich außerdem, wie sich weiter unten zeigen wird, noch weitere Nachteile ergeben, wird von der Verwendung dieser Eingabefunktion abgeraten.

Listing 16 : ../C++Ser/ObjectInputStream.cpp [Zeile 90 bis 98]

```
template <class T>
T& ObjectInputStream::readObject(T &obj) throw (ClassCastException) {
  T *t = dynamic_cast<T*>(serialParse→object()→getSerializable());
  if (t) {
    obj = *t;                          // Der Zuweisungsoperator wird zum
    return obj;                        // Kopieren des Objektes verwendet.
  }
  else throw ClassCastException();     // Wird auch dann geworfen wenn nur das
}                                      // gelesene Java Objekt 'null' war.
```

Hier zeigt sich auch ein weiterer Grund, weshalb Java Objekte vernünftigerweise nur auf C++ Zeiger abgebildet werden sollten. Java Referenzvariablen können wie alle Zeigertypen den ausgezeichneten Wert "`null`"[4] haben, was soviel bedeutet wie daß sie auf gar nichts zeigen. Werttypen haben aber immer einen Wert aus ihrem Wertebereich. Selbst wenn sie, wie in C++ möglich nicht initialisiert werden, haben sie doch einen Wert aus ihrem speziellen Wertebereich. Sie können nicht "keinen" Wert haben. Somit haben sie einen Freiheitsgrad weniger als Zeigertypen, was eine bijektive Abbildung eben unmöglich macht.

[4] Während "`null`" in Java ein reservierter Bezeichner für einen beliebigen Referenztyp ist, der "nichts" referiert, ist "`NULL` in C++ auch nach dem neuen Standard immer noch nur ein implementationsabhängiges Makro (z.Bsp `0` oder `0L` aber nicht `(void*)0`).

Genauso handelt es sich bei den in C++ verwendeten Referenzen um etwas völlig anderes als bei denen von Java her bekannten Referenzen. Während sie einen gemeinsamen Namen tragen, was schon genug zur Verwirrung beitragen kann, sind C++ Referenzen nur Decknamen (*engl: alias*) für tatsächliche Objekte. Deswegen müssen sie auch immer initialisiert[5] werden. Es kann in C++ keine Referenz deklariert werden, der erst später das Objekt zugewiesen wird, das es referenzieren soll. Somit sind C++ Referenzen eine Mischung aus Zeigertyp, denn sie können durchaus nacheinander mehrere Objekte referenzieren, und Werttypen, da sie immer einen Wert ihres eigenen Wertebereiches enthalten müssen. Deswegen ist es, obwohl der Name das Gegenteil suggeriert, genauso zwecklos eine Abbildung von Java Objektvariablen auf C++ Referenzen zu versuchen, wie es der Versuch der Abbildung auf C++ Objektvariablen ist.

3.2.3 Arrays

Eigentlich handelt es sich bei Arrays in Java, wie schon in Kapitel 2.1.1 erwähnt, um Objekte. Dennoch erfahren sie sowohl in der Sprache selber als auch im Serialisierungsprotokoll eine spezielle Behandlung, weswegen ihnen hier auch ein eigener Abschnitt gewidmet ist. Daß Arrays in Java von der gemeinsamen Klasse `Object` abgeleitet sind, ist hier beim Parsen jetzt von großem Vorteil. Arrayobjekte können damit auf gleiche Weise aus dem Bytestrom gelesen werden wie andere Objekte, und sie können Informationen über ihre Dimension und ihren Elementtyp enthalten.

Da wie oben beschrieben Java Objektvariablen auf C++ Zeiger abgebildet werden, bietet es sich an, Java Arrayvariablen ihrerseits auf C/C++-Arrays abzubilden. Bei diesen handelt es sich ja um nichts anderes als um Zeiger, wobei ein eindimensionales Array (z.Bsp. `int feld[];`) in C/C++ einem einfachen Zeiger auf den Elementtyp des Arrays (hier also `int *feld;`) entspricht. Leider gibt es diese hundertprozentige Übereinstimmung nur im eindimensionalen Fall. Sie rührt daher, daß C/C++ einen Arraybezeichner automatisch in einen Zeiger auf das erste Element des Feldes umwandeln kann ([ANSI-CPP] 4.2) und im eindimensionalen Fall entspricht das einem Zeiger auf den Elementtyp.

Schon im zweidimensinalen Fall gibt es jedoch Unterschiede. Wenn zum Beispiel `char feld[3][8];` vorausgesetzt wird, dann bezeichnet `feld[0]` einen Zeiger auf ein eindimensionales Array und dieser entspricht nicht ganz einem Zeiger auf `char`. Während nämlich ersterer dem letzteren zugewiesen werden kann (z.Bsp `char *pC = feld[0];`) ist der umgekehrte Weg nicht möglich. Dies liegt daran, daß es sich bei dem Arrays um einen zusammenhängenden Speicherbereich handelt und bei `feld[0]` nur um einen vom Compiler erzeugten, temporären Zeiger in dieses Array, an den aber nicht zugewiesen werden kann[6]. Beim Zeigerpendant `char **pFeld;` ist `pFeld[0]` jedoch eine Zeigervariable auf der Halde, die überhaupt erst einmal vom Programmierer mittels `pFeld = new (char*)[3];` angelegt und dann mit `pFeld[0] = new char[8];` initialisiert werden muß (siehe Abbildung 3.2).

Ein mittels Zeigern realisiertes *n*-dimensionales Array verbraucht in C++ also neben dem Speicherplatz für die Arraydaten immer noch das Produkt der ersten (*n-1*) Dimensionszahlen für zusätzliche Zeiger. Durch diese zusätzliche Indirektion wird es jedoch zum Beispiel

[5] Eine Ausnahme hiervon stellen Funktionsargumente und Klassenmitglieder dar. Bei Funktionsargumenten wird jedoch die Initalisierung bei jedem Funktionsaufruf vom Compiler vorgenommen, während bei Klassenmitgliedern im Konstruktor selbst für die Initialisierung gesorgt werden muß.

[6] Es handelt sich dabei um einen sogenannten *r-Value*.

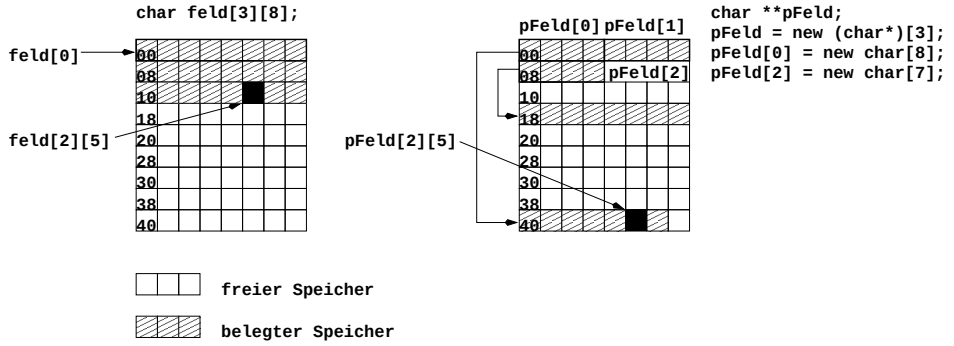

Abbildung 3.2: Der Unterschied zwischen mehrdimensionalen Arrays und Zeigern auf Zeiger in C/C++

möglich, asymmetrische Arrays zu konstruieren, was wiederum genau dem Java Modell für Arrays entspricht.

Zusammenfassend kann also gesagt werden, daß Java-Arrays am ehesten auf durch Zeiger realisierte C++-Arrays abgebildet werden könnten, da diese sowohl asymmetrisch[7] sein können als auch das Kriterium erfüllen, daß ihre Größe sich dynamisch zur Laufzeit festlegen oder ändern läßt.

Leider erweist sich dieser Weg in der Praxis als Einbahnstraße. Während der Weg von Java nach C++ wie oben beschrieben sozusagen auf natürliche Weise funktioniert, ist der Weg zurück nach Java wegen einiger Unzulänglichkeiten von C++ einfach nicht möglich. Während es in C++ noch möglich ist, von einem statischen Array mittels des `sizeof()` Operators dessen tatsächliche Größe zur Laufzeit ausfindig zu machen, ist dies bei Zeigern unmöglich. Nach einer Speicheranforderung der Form `int *pFeld = new int[7];` kann nicht mehr festgestellt werden, auf wie viele `int`-Werte `pFeld` tatsächlich zeigt. Diese Information ist jedoch sehr wohl noch im Speicherverwaltungssystem der Laufzeitumgebung vorhanden. So muß beim Freigeben dieses Speicherbereiches dem System mit der speziellen Syntax `delete[] pFeld;` mitgeteilt werden, daß es sich bei `pFeld` um den Zeiger auf ein Array und nicht nur um einen Zeiger auf ein einfaches Objekt handelt.

Aus diesen Gründen werden Java-Arrays, wie schon in Listing 14 auf Seite 25 zu sehen war, in C++ auf die in der STL[8] definierte Klasse `vector` abgebildet. Bei der STL handelt es sich um eine generische Bibliothek. Daß heißt, daß sie im Gegensatz zu herkömmlichen C++ Bibliotheken nicht aus einer tiefen Hierarchie voneinander abgeleiteter Klassen besteht. Statt Vererbung wird in ihr dafür ein anderes Sprachmittel von C++, nämlich Parametrisierung intensiv eingesetzt. Von einem Mathematiker entworfen, konzentriert sie sich hauptsächlich auf effiziente Algorithmen, die als Templatefunktionen generisch geschrieben werden und mit unterschiedlicher Effizienz auf Daten verschiedener, ebenfalls parametrisierter Containerklassen angewendet werden können. Dabei werden jeweils Garantien für das zu erwartende Zeitverhalten in der in der Informatik üblichen O-Notation gegeben.

Diese ursprünglich bei Hewlett Packard entwickelte Bibliothek ist heute schon weit verbreitet und praktisch für jeden Compiler verfügbar. Außerdem wurde sie in den neuen C++

[7] Ein Beispiel einer asymmetrischen Matrix ist zum Beispiel eine trianguläre Matrix

[8] STL : Standard Template Library (siehe z.Bsp. : [ANSI-CPP], [Brey96], [Jos96], [Lee-Stepanov], [Musser-Saini])

Standard [ANSI-CPP] aufgenommen, was auch für die Zukunft eine weitestgehende Kompatiblität auf unterschiedlichsten Hard- und Softwareplattformen garantiert.

Beim Erstellen dieser Serialisierungsbibliothek wurde die STL ebenso wie die C++ Standardbibliothek intensiv eingesetzt. Ohne sie hätte das Projekt nicht in dieser Zeit fertiggestellt werden können und ohne sie hätte es vielleicht den doppelten oder dreifachen Umfang an Code generiert. In den folgenden Kapiteln, in denen näher auf die Implementieurung eingegangen werden wird, werden nach und nach auch die eingesetzten Klassen und Algorithmen vorgestellt werden.

Jetzt jedoch zurück zur Klasse `vector`, auf die die Java-Arrays abgebildet werden. Sie ist folgendermaßen definiert[9] :

```
template <class T> vector;
```

Sie implementiert ein dynamisches Array. Dabei können Vektoren natürlich beliebig geschachtelt werden. Ein zweidimensionales Array von Ganzzahlen kann somit folgendermaßen realisiert werden :

```
vector<vector<int> > field;
```

Bei der Initialisierung solcher mehrdimensionaler Arrays ist jedoch Vorsicht geboten. Das oben deklarierte Array könnte etwa folgendermassen angelegt werden :

```
field = vector<vector<int> >(7, vector<int>(9)) ;
```

Damit wird ein Vektor mit sieben Elementen erzeugt, wobei jedes dieser Elemente wiederum aus einem neunelementigen Vektor besteht. Insgesamt also ein Array von 7x9 Ganzzahlen.

Wenn auch in der Initialisierung etwas komplizierter als die normalen C/C++-Arrays, unterscheidet sich der spätere Zugriff auf sie dank des für `vector` überladenen Zugriffsoperators "`[]`" nicht mehr. Es ist zum Beispiel möglich, die zu der von den Java-Arrays her bekannten Mitgliedsvariablen `length` äquivalente Funktion `size()` zu verwenden, um die aktuelle Größe eines Vektors zu erfahren. Desweiteren lassen sich mit geschachtelten Vektoren auch asymmetrische Arrays realisieren. Es muß jedoch darauf geachtet werden, daß bei Zugriffen mittels des Subskriptionsoperators, genauso wie in C keine Bereichsüberprüfung stattfindet. Soll das gleiche Verhalten wie in Java simuliert werden, dann muß die Mitgliedsfunktion `at()` verwendet werden. Sie wirft im Fehlerfalle eine `out_of_range` Ausnahme.

Ein Nachteil bei der Verwendung von STL Containern als Gegenstücke zu Java Klassen sollte jedoch nicht verschwiegen werden. Alle STL Container sind darauf ausgelegt, eine Wertsemantik zu unterstützen. Das heißt, sie legen grundsätzlich Kopien von allen ihnen übergebenen Elementen an und liefern auch Kopien dieser Elemente zurück. Dies hat den Vorteil, daß es keine Probleme mit Zeigern auf nicht existierende Elemente oder rekursiven Datenstrukturen gibt. Es steht allerdings im Gegensatz zu der in Java verwendeten Refe-

[9] Normalerweise haben alle Container der STL noch einen weiteren Templateparameter, der das Speichermodell festlegt. Dieser ist mit einem Templatedefaultparameter vorbelegt. Auf Systemen die dies nicht unterstützen, fehlt er typischerweise ganz. In dem hier verwendeten Zusammenhang ist er uninteressant und wird deswegen weggelassen

renzsemantik. Deswegen mußten bei der Sprachabbildung einige Kompromisse gemacht werden.

Die derzeitige Lösung sieht folgendermaßen aus. Eindimensionale Arrays werden als Zeiger auf einen Vektor des entsprechenden Typs realisiert. Enthält das Java-Array primitive Datentypen, handelt es sich bei dem Templateparameter des C++ Vektors um den entsprechenden Datentyp (siehe Tabelle 3.1 auf Seite 28). Enthält das Java-Array einen Objekttyp, so handelt es sich bei dem Templateparameter um einen Zeiger auf diesen Typ, ganz im Einklang mit der in 3.2.2 beschriebenen Abbildung von Objekten. Probleme gibt es bei mehrdimensionalen Arrays. Hier enthält jeder Eintrag eines *n*-dimensionalen Java-Arrays ein *n-1*-dimensionales Array. Dabei handelt es sich in Java natürlich wieder um eine Referenz, die in C++, der bisherigen Philosophie folgend, eigentlich auf einen Zeiger abgebildet werden müßte. Diese Lösung hätte allerdings die unschöne Eigenschaft, daß damit nicht mehr auf kanonische Weise mittels "`[x][y]`" auf ein Element zugegriffen werden könnte, sondern die etwas ungewöhnliche Schreibweise "`(*[x])[y]`" verwendet werden müßte, da der erste Eintrag erst dereferenziert werden muß.

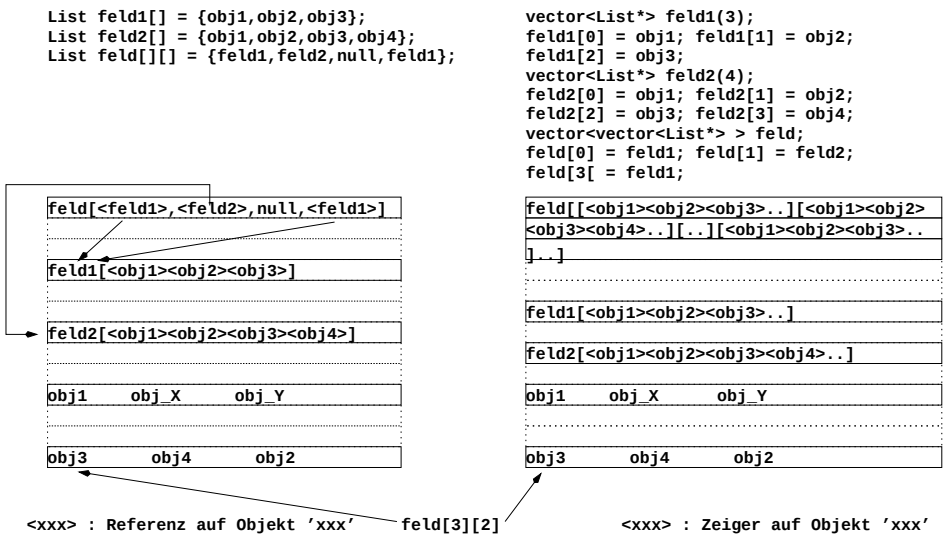

Abbildung 3.3: Der Unterschied von Java Arrays und mittels `vector` erzeugten C/C++ Arrays. Zu beachte ist, daß der Zugriff auf `feld[3][2]` zwar jeweils das gleiche Objekt zurückliefert, daß dagegen die Zuweisung eines neuen Wertes an diesen Arrayeintrag in Java gleichzeitig auch den Wert von `feld[0][2]` ändert, was in C++ nicht der Fall ist, da dort die Einträge der eindimensionalen Arrays beim Initialisieren des zweidimensionalen Arrays kopiert wurden.

Dadurch läßt sich die Semantik von Java nicht ganz nachbilden. Dort ist es zum Beispiel möglich, zwei zweidimensionale Arrays zu haben, die als Elemente die selben eindimensinalen Arrays aber in unterschiedlicher Reihenfolge enthalten oder ein zweidimensionales Array, welches mehrfach das selbe eindimensionales Array enthält (Abbildung 3.3). Dies ist jetzt in C++ nicht mehr möglich, da die eindimensionalen Arrays hier als Wertparameter kopiert werden. Bei dem hier beschriebenen Szenario handelt es sich jedoch um eine eher exotische Anwendung von Java-Arrays, die ihre ungenaue Nachbildung in C++, in Anbetracht einer einfacheren und gebräuchlicheren Syntax gerechtfertig erscheinen läßt. Es ist allerdings zu beachten, daß die Topologie etwaiger Objekte des Arrays erhalten bleibt, da

es sich bei ihnen nach wie vor nur um Zeiger handelt (Abbildung 3.3).

3.2.4 Zeichenketten

Zum Abschluß dieses Kapitels sollen die Zeichenketten betrachtet werden. Sie sind beim Programmieren von großer Bedeutung und haben deswegen in den meisten Programmiersprachen wie zum Beispiel in C/C++ oder in Pascal eine Sonderbehandlung erfahren. Das ist auch in Java nicht anders. Zwar handelt es sich bei dem Datentyp `String` um eine ganz normale, von `Object` abgeleitete und im Paket `java.lang` deklarierte Klasse. Aber im Serialisierungsprotokoll wird sie ebenso wie ein Array als spezielles Objekt übertragen (siehe dazu die Produktion `object` in Kapitel 2.3.1 auf Seite 17).

Auf C/C++-Seite gibt es dagegen die Wahl, ob als Pendant dafür die altehrwürdigen C-Strings (`char*`) oder die in der neuen C++-Standardbibliothek definierte `string`-Klasse verwendet werden sollen. Aus mehreren Gründen wurde der `string`-Klasse der Vorzug gegeben. Erstens ist der allzu unvorsichtige Umgang mit C-Strings eine der häufigsten Fehlerursachen bei der C/C++-Programmierung. Maistens handelt es sich dabei um Speicherschutzverletzungen da ein C-String seine Kapazität nicht abspeichert und es somit keine Möglichkeit gibt, diese nachträglich festzustellen. Die dazu oft verwendete Funktion `int strlen(const char*);` offenbart auch gleich einen weiteren Nachteil der C-Strings. Sie sind per Definition immer von einem Null-Byte abgeschlossen, und die Tatsache daß praktisch alle für sie vorhandenen Bibliotheksfunktionen davon ausgehen, macht sie für die Speicherung binärer Daten unbrauchbar.

Dagegen ist der bisherige Hauptnachteil der `string`-Klasse, nämlich ihre System- und Compilerabhängigkeit, nach der Standardisierung in [ANSI-CPP] nicht mehr gegeben. Einen Nachteil im Kontext der Serialisierung sollte jedoch nicht verschwiegen werden. Die Klasse `String` ist nicht wie die Klasse `List` aus dem Beispiel in (Listing 14) von `Serializable` abgeleitet. Das heißt, daß sie die zur Serialisierung notwendigen von `Serializable` geerbten Methoden nicht zur Verfügung stellen kann. Das macht den Einsatz sogenannter Hüllenklassen (*engl.: envelope, wrapper*) notwendig. Auf den genauen Einsatz dieser Technik wird in Kapitel 4, näher eingegangen werden.

3.3 Von C++ zurück nach Java - ein Beispiel

Auch die umgekehrte Richtung von C++ nach Java soll mit einem Beispiel begonnen werden. Dazu wird wieder das schon in Listing 13 auf Seite 24 gezeigte Programm verwendet, nur daß die soeben deserialisierten Daten wieder zurück in einen Serialisierungsstrom geschrieben werden, aus dem sie daraufhin von einem Javaprogramm eingelesen werden können. Wie schon im vorletzten Abschnitt erwähnt, wird der Einfachheit halber die Standardausgabe `cout` als Ausgabestrom verwendet. Listing 17 zeigt die bisher fehlenden Zeilen des Programmes.

Listing 17 : ../C++Ser/serReadWrite.cpp [Zeile 56 bis 68]

```
ObjectOutputStream outStream(cout);
outStream.writeObject(writeList1);
outStream.writeObject(writeList2);
```

➥

Listing 17 : ../C++Ser/serReadWrite.cpp [Zeile 56 bis 68] (Fortsetzung)

```
outStream.writeObject(hta);
outStream.writeObject(str);
outStream.writeObject(hta1);
outStream.writeInt(256);
outStream.writeDouble(999.999);
(*str) = "Der selbe \"string\" ...";
outStream.writeObject(str);
(*str) = "... mit einem anderen Wert.";
outStream.writeObject(str);
outStream.flush();
```

Diese von C++ aus serialisierten Objekte werden jetzt auf Javaseite noch einmal eingelesen. Dafür wird das Java Programm aus Listing 10 und 11 von Seite 14 um die folgenden Zeilen ergänzt:

Listing 18 : ../C++Ser/java1/List.java [Zeile 210 bis 230]

```
if (System.in.available() > 0) {
  in = new ObjectInputStream(System.in);
  list1 = (List)in.readObject();
  list2 = (List)in.readObject();
  for (int x=0; x < 2; x++) {
    List rootList = list1;
    while (list1 ≠ null) {
      System.out.println(list1);
      list1 = list1.next;
      if (list1 == rootList)
        { System.out.println(" Cycle detected"); break; }
    }
    list1 = list2;
  }
  System.out.println((Hashtable)in.readObject());
  System.out.println((String)in.readObject());
  System.out.println((Hashtable)in.readObject());
  System.out.println(in.readInt() + "," + in.readDouble());
  System.out.println((String)in.readObject());
  System.out.println((String)in.readObject());
}
```

Diese unterscheiden sich nur in den ersten beiden Zeilen von Listing 11. Darin wird zuerst geprüft ob dem Programm auf der Standardeingabe `System.in` Daten zur Verfügung gestellt wurden. Wenn ja, wird ein `ObjectInputStream` mit der Standardeingabe als Datenquelle verbunden und initialisiert. Danach können die Objekte auf genau die gleiche Art und Weise gelesen werden, wie das auch aus dem `ByteArrayInputStream` in Listing 11 geschah.

Mit dem kleinen Unterschied allerdings, daß sie mittlerweile von einem C++-Programm gelesen, deserialisiert und dann wieder serialisiert wurden.

Abbildung 3.4 zeigt schließlich die Ausgabe dieses letzten Programmabschnittes. Sie gleicht derjenigen aus Abbildung 2.1 auf Seite 16 bis auf die neuen Hashwerte der ausgegebenen Objekte. Ansonsten wurden aber sowohl die Werte als auch die Topologie der Objekte erhalten.

```
>java List < serial.out
 List(80cdf1e) : -177,VHS,80cdf35
 List(80cdf35) : 288,null,80cdf64
 Mist(80cdf64) : -399,Zenit,80cdf1e
        Cycle detected
 List(80cdf35) : 288,null,80cdf64
 Mist(80cdf64) : -399,Zenit,80cdf1e
 List(80cdf1e) : -177,VHS,80cdf35
        Cycle detected
 {hy=wie geht's, hallo=wie geht's, super=super}
 THIS IS A STRING
 {hy=List(80cdf35) : 288,null,80cdf64,
  hallo=List(80cdf1e) : -177,VHS,80cdf35,
  super=Mist(80cdf64) : -399,Zenit,80cdf1e}
 256,999.999
 Der selbe "string" ...
 ... mit einem anderen Wert.
```

Abbildung 3.4: Die Ausgabe des Programmes aus Listing 18. Bis auf die Hashwerte der ausgegebenen Objekte gleicht sie Abbildung 2.1

3.4 Die Abbildung von C++ auf Java-Objekte

Die primitiven Datentypen werden wieder entsprechend der Tabelle 3.1 (s. 28) abgebildet. Wie auch in Java gibt es für jeden primitiven Java-Typ eine eigene Ausgabefunktion, die als Argument eine Variable des der Abbildung entsprechenden Typs erhält. Darüber hinaus können durch die in C/C++ automatisch durchgeführte Typumwandlung auch die in der Tabelle nicht erwähnten Typen geschrieben werden, insbesondere auch die in C/C++ üblichen `unsigned` Varianten. Für eine dadurch eventuell auftretende Wertebereichsüberschreitung ist der Programmierer verantwortlich.

C++-Objekte werden auf Java-Referenzen abgebildet. Dabei spielt es keine Rolle, ob es sich auf C++-Seite um ein Objekt oder einen Zeiger auf ein Objekt handelt. In Wirklichkeit existieren die Ausgabefunktionen nur für Zeiger. Bei Wertobjekten wird einfach die entsprechende Version mit der Adresse des Objektes aufgerufen. Genau wie in Java “merkt” sich der Serialisierungsstrom einmal ausgegebene Objekte. Wird das gleiche Objekt, als Wertparameter oder über eine Zeigerreferenz, noch einmal serialisiert, dann wird nur eine Referenz auf sein erstes Vorkommen geschrieben. Hierbei muß beachtet werden, daß es nicht reicht, sich die Adresse eines ausgegebenen Objektes zu merken, da sich der Wert des Objektes ändern kann, während seine Adresse konstant bleibt. Es muß deswegen in

C++, genauso wie in Java auch, ein Hashwert über das ganze Objekt gebildet und abgespeichert werden, um auch Änderungen des Objektwertes selber feststellen zu können. Soll dagegen wirklich des gleiche Objekt noch einmal gesendet werden, dann muß vorher der Ausgabestrom zurückgesetzt werden, was zur Folge hat, daß er neu initialisiert wird, wobei sämtliche Referenzen auf zuvor geschriebene Objekte verlorengehen.

Desweiteren sollten, wie in C/C++ üblich, nicht verwendete Zeiger mit `NULL` initialisiert werden. Sie werden dann in Java auf `null`-Referenzen abgebildet. Insgesamt können natürlich nur Objekte serialisiert werden, die entweder von `Serializable` abgeleitet sind, oder solche, für die es eine entsprechende Hüllenklasse gibt. Eine Ausnahme machen hier wieder die Arrays. Aus denen in Kapitel 3.2.3 auf Seite 30 angesprochenen Günden, ist es nicht möglich C/C++-Arrays zu serialisieren. Als Ersatz dafür werden die Vektoren der STL verwendet. Für sie gibt es keine Hüllenklassen sondern spezielle Ausgabefunktionen, die jedoch vor dem Benutzer verborgen sind. Zu ihrer Ausgabe können, wie bei anderen Objekten auch die `writeObject()` Methode verwendet werden. Wie dies alles genau funktioniert wird im nächsten Kapitel besprochen werden, in dem die C++ Serialisierungsbibliothek vorgestellt und sowohl auf deren Anwendung als auch deren Implementierung eingegangen werden wird.

4. Die C++ Serialisierungsbibliothek

In diesem Kapitel soll die schon in den Beispielen der letzten Kapitel verwendete Serialisierungsbibliothek besprochen werden. Dabei wird erst einmal ihre Klassenhierarchie vorgestellt und dann auf ihre Anwendung eingegangen. Danach wird die konkrete Implementierung anhand von Quellkodeauszügen diskutiert und die erstellte Software dokumentiert.

In C++ existiert kein Standardpersistenzmechanismus[1] für Objekte wie dies in Java der Fall ist. Trotzdem gibt es einige Bibliotheken, die dem Programmierer solch einen Mechanismus zur Verfügung stellen. Dabei muß grundsätzlich zwischen intrusiv (engl: *to intrude* = *hineindrängen*) und nonintrusiv Ansätzen unterschieden werden.

Intrusiv bedeutet in diesem Fall, daß ein Eingriff in die zu serialisierenden Klassen notwendig ist. Dafür müssen diese natürlich erst einmal im Quellkode vorhanden sein, was bei komerziellen Bibliotheken nicht immer der Fall ist. Vertreter dieser Art von Bibliotheken sind zum Beispiel die MFC (Microsoft Foundation Classes), bei der alle Klassen, die persistent gemacht werden sollen, von `CObject` abgeleitet werden müssen, oder Tools.h++ von RogueWave, bei der um den gleichen Effekt zu erzielen, von `RWCollectable` abgeleitet werden muß. Typischerweise müssen daraufhin spezielle virtuelle Funktionen dieser Basisklassen überschrieben werden.

Beim nonintrusiv Ansatz müssen demgegenüber an den zu serialisierenden Klassen keine Eingriffe vorgenommen werden. Ein typischer Vertreter davon ist zum Beispiel die Java Objektserialisierung. In dieser reinen Form wird dieser Ansatz unter C++ jedoch kaum wiederzufinden sein, da es in C++ eben keine zu der JVM äquivalente Instanz gibt, die zur Laufzeit Informationen über alle gerade aktiven Objekte bereithält. Der Grund dafür ist schlicht und einfach Effizienz. C++ hat schon eine lange Geschichte als C hinter sich, und in C werden Fragen der Typisierung grundsätzlich statisch, also zur Übersetzungszeit vom Compiler entschieden. Dieses Prinzip wurde zwar in C++ zugunsten einer größeren Funktionalität aufgeweicht, so daß ein C++-Objekt zur Laufzeit gewisse Typinformationen mit sich führen kann. Es ist jedoch noch weit von den Möglichkeiten einer Laufzeitumgebung, wie sie eine JVM bereitstellt, entfernt.

In C++ wird dieser Ansatz deswegen normalerweise über Hüllenklassen realisiert, die sowohl von einer allgemeinen Basisklasse, als auch von der zu serialisierenden Klasse erben, oder durch den Einsatz von Entwurfsmustern wie Facetten[2] (engl : *facets*) und Externem Polymorphismus [CleeSchm]. Ein Beispiel einer kommerziellen Bibliothek, die auf diesen Prinzipien beruht, ist das Systems<ToolKit> [Glass] der Firma ObjectSpace.

[1] in Corba gibt es im Vergleich dazu den Persistent Object Service (POS) (siehe [CORBA])

[2] Die Technik der Facetten wird zum Beispiel in der Standard C++ Bibliothek für die Klasse `locale` verwendet, die der Parametrisierung der Sprachumgebung dient

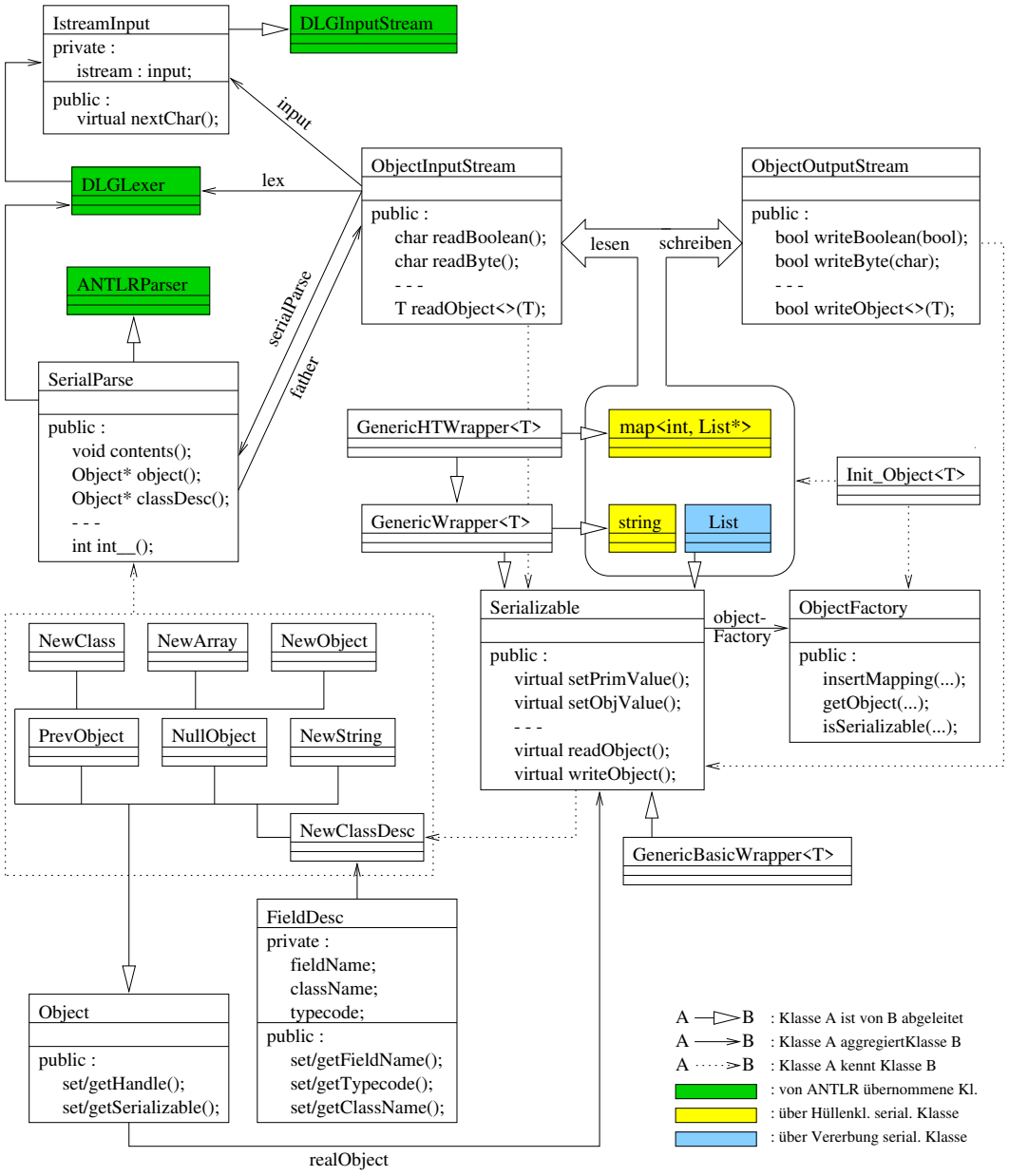

Abbildung 4.1: Das vollständige Klassendiagramm der Serialisierungsbibliothek . Es werden nur die jeweils wichtigsten Methoden und Klassenvariablen dargestellt.

Einen ganz anderen Ansatz verfolgen dagegen das schon von C bekannte RPC/XDR und neuerdings Corba. Hier werden die Strukturen beziehungsweise Klassen in einer eigenen Sprache deklariert, und aus diesen Definitionen erzeugt dann ein Precompiler Sprachabbildungen für C beziehungsweise C++. Es ist klar, daß auf diese Weise alle Klassen ohne großen Aufwand automatisch befähigt werden können, einen bestimmten Persistenzmechanismus zu unterstützen.

4.1 Serializable - Die zentrale Klasse der Bibliothek

Die hier entwickelte Bibliothek verwendet Elemente beider im letzten Abschnitt vorgestellter Ansätze. Sie ist eigentlich dafür gedacht, die Entwicklung neuer Client/Server Systeme im heterogenen Java/C++ Umfeld zu erleichtern. Demnach handelt es sich bei dem Hauptmechanismus der Klassen zur Serialisierung befähigt auch um einen Intrusiven : alle diese Klassen müssen von der abstrakten Klasse `Serializable` abgeleitet werden. Diese ist aber anders als das Java Interface `Serializable` nicht leer, sondern definiert einige reine virtuelle Funktionen, die von den erbenden Klassen implementiert werden müssen. Es wurde zwar versucht, die Anzahl dieser Funktionen möglichst gering zu halten, dennoch sind es, wie in Listing 19 zu sehen ist, mindestens 3 geworden, die für jede neue Klasse ausprogrammiert werden müssen. Dabei handelt es sich jedoch durchweg um Funktionen, die keine besondere Logik implementieren, sondern nur die Unzulänglichkeiten in Bezug auf Laufzeitunterstützung von C++ kompensieren. Sie könnten eigentlich alle automatisch aus einer bestehenden Java Klassendefinition erzeugt werden und nach einem Beispiel sollte es keine Schwierigkeiten bereiten, diese Funktionen für beliebige Klassen zu erstellen.

Außerdem wurde aber für Klassen, die nicht im Quellkode vorliegen oder für spezielle Datentypen wie zum Beispiel `string`, die Möglichkeit geschaffen, über Hüllenklassen serialisiert zu werden. Diese Hüllenklassen sind dabei sowohl von der zu serialisierenden Klasse als auch von `Serializable` abgeleitet. Auf ihre Verwendung wird in Kapitel 4.4 eingegangen werden.

Abbildung 4.1 zeigt die Struktur der Bibliothek. Sie besteht aus mehreren, eng gekoppelten Klassen, die jetzt vorgestellt werden sollen. Den Anfang macht dabei die schon erwähnte Klasse `Serializable`. Sie ist die zentrale Klasse der Bibliothek und wird von fast allen anderen Klassen verwendet. Listing 19 zeigt die vollständige Deklaration dieser Klasse. Bei den ausgelassenen Zeilen handelt es sich nur um Kommentare im Quelltext.

Listing 19 : ../C++Ser/Serializable.hpp [Zeile 81 bis 110]

```
class Serializable {
protected:
  static ObjectFactory *objectFactory; // statically initialised to NULL
  static void initBasicObjects();
public:
  typedef Serializable objType;
  virtual void setPrimValue(const string&, void*) {}
  virtual void setObjValue(const string&, Serializable*) {}
  virtual string getPrimValue(const string&) {}
  virtual Serializable* getObjValue(const string&) {}
  virtual Serializable* getNewObj() = 0;
  virtual const NewClassDesc& getClassDesc() = 0;
// ...
  virtual void* getObjAddress() { return this; }
  virtual unsigned int getObjSize() = 0; // { return sizeof(this); }
  virtual void readObject(ObjectInputStream*) {}
  virtual void writeObject(ObjectOutputStream*, string = "") {}
```

Listing 19 : ../C++Ser/Serializable.hpp [Zeile 81 bis 110] (Fortsetzung)

```
  // The next three functions only give access to the objectFactory
  static void insertMapping(const string&, const string&, Serializable*);
  static Serializable* getObjectByName(const string&);
  static Serializable* getObjectByType(const string&);
  // The next function computes a hash value for a given object
  static unsigned int hash(void*, unsigned int);
  static string getSUID(const string&);
};
```

`Serializable` enthält einen statischen Zeiger auf ein Objekt vom Typ `ObjectFactory`, der zentralen Datenstruktur der Bibliothek, und drei statische Funktionen, deren einzige Aufgabe es ist, den Zugriff auf dieses Objekt zu kapseln. Gleichzeitig sind sie dafür verantwortlich, daß `objectFactory` genau einmal angelegt wird. Da es sich bei `objectFactory` um einen Zeiger handelt wird er vom Compiler statisch mit `NULL` initialisiert, so daß keine weitere statische Variable notwendig ist, um die einmalige Initialisierung zu garantieren. Im allgemeinen passiert diese Initialisierung noch bevor das Hauptprogramm zur Ausführung kommt, von welchem aus die Bibliothek verwendet wird. Diese Problematik der Initialisierung von globalen Veriablen wird in Kapitel 4.2.1 ausführlich behandelt. Die bisher vorgestellten Methoden und Variablen sind statisch in Bezug auf die Klasse `Serializable`, das heißt, sie können auch ohne eine Instanz dieser Klasse verwendet werden. Der einzige Grund, warum sie in `Serializable` gekapselt wurden ist die Vermeidung von Bezeichnern im globalen Namensraum (engl. *namespace*). Eine andere Möglichkeit dies zu erreichen wäre die Verwendung von eigenen Namensräumen gewesen, diese Technik wird heute allerdings noch nicht von allen C++-Compilern unterstützt.

Zusammen mit dem `objectFactory` Objekt haben diese Methoden die Aufgabe, die in C++ fehlende Laufzeitumgebung zu ersetzen und für die bei ihm registrierten Klassen Informationen über deren Datenfelder und Vererbungsstrukturen zu verwalten. Innerhalb der Klasse `ObjectFactory`, deren Deklaration in Listing 20 zu sehen ist, werden diese Daten in zwei assoziativen Kontainern abgespeichert, wobei jeweils Namen den Zeiger auf ein `Serializable` Objekt indizieren.

Listing 20 : ../C++Ser/Serializable.hpp [Zeile 64 bis 77]

```
typedef map<string, Serializable* > ObjectMap;
class ObjectFactory {
friend class Serializable;
private:
  ObjectMap *objectMap;//maps object names (e.g. 'java.lang.String') to objects
  ObjectMap *typeMap;  //maps object types (e.g. 'typeid(o).name()') to objects
public:
  ObjectFactory();
  ObjectFactory(const ObjectFactory&);
  ObjectFactory& operator=(const ObjectFactory&);
  void insertMapping(const string&, const string&, Serializable*);
                                                                    ➥
```

Listing 20 : ../C++Ser/Serializable.hpp [Zeile 64 bis 77] (Fortsetzung)

```
  Serializable* getObjectByName(const string&);
  Serializable* getObjectByType(const string &);
};
```

Als nächstes sollen nun die drei von Serializable gekapselten Methoden von ObjectFactory näher beschrieben werden. Die erste, insertMapping(), wird in Listing 21 gezeigt. Sie fügt eine neue Zuordnung einer Java Klasse zu einer C++-Klasse in die Datenstrukturen der objectFactory ein.

Listing 21 : ../C++Ser/Serializable.cpp [Zeile 195 bis 204]

```
void ObjectFactory::insertMapping(const string& name,
                                  const string& typeName, Serializable* obj) {
  if (typeMap→count(typeName)) {
    delete obj; // We already have an object of this type, so delete the new one
    (*objectMap)[name] = (*typeMap)[typeName];
  } else {
   (*objectMap)[name] = obj;
   (*typeMap)[typeName] = obj;
  }
}
```

Bei ihrem ersten Argument handelt es sich dabei um den Java Klassennamen, also zum Beispiel "java.lang.String;" und bei dem Zweiten um den Klassennamen des C++-Objektes, auf das das Java Objekt abgebildet werden soll. Dabei wird jedoch nicht der Name verwendet, den der Programmierer der Klasse gegeben hat, sondern der von der Methode name() der Klasse type_info zurücklieferte Bezeichner, wobei in C++ jeder Klasse vom Compiler eine entsprechende type_info Klasse zugeordnet wird.

Bei type_info handelt es sich um eine zum neuen C++-Standard gehörige Klasse, deren Objekte von dem typeid() Operator zurückgeliefert werden. Sie enthält vom Compiler erzeugte Typinformation und ihre Mitgliedsfunktion name() liefert einen vom Compiler erzeugten, eindeutigen Klassennamen zurück[3]. Der Standard schreibt jedoch nur Eindeutigkeit innerhalb einer Applikation vor, so daß sich diese Namen auf unterschiedlichen Systemen durchaus unterscheiden können, obwohl sie die gleiche Klasse bezeichnen. Dies ist hier jedoch kein Problem, da als Schlüssel immer der Java Klassenname verwendet wird, der eindeutig ist. Dadurch ist eine Kommunikation mittels Objektserialisierung auch zwischen zwei C++-Anwendungen, die auf unterschiedlichen Systemen laufen möglich.

Das dritte Argument ist schließlich ein Zeiger auf ein Serializable Objekt. Und über

[3] Gewöhnlich handelt es sich hierbei um den gleichen Namen der auch in Objektdateien zur Identifizierung der Objektvariablen und Klassenmethoden verwendet wird. Er enthält neben dem vom Programmierer vorgegebenen Klassennamen noch andere Informationen wie zum Beispiel die Namen der parametrisierenden Typen bei Templateklassen. Diese Informationen werden nach einem bestimmten Verfahren (engl. *name mangling* = *Namensverstümmelung*) zusammengesetzt, das ebenfalls nicht im C++-Standard spezifiziert ist und zum Beispiel dazu führt, daß sich mit unterschiedlichen Compilern erzeugte Objektdateien normalerweise nicht binden (engl. *link*) lassen.

dieses Serializable Objekt ist es auch möglich, Information über seinen Typ zu erhalten. Für jede zu serialisierende Klasse, die von Serializable abgeleitet ist, muß der Programmierer nämlich die als rein virtuell deklarierte Methode

```
virtual NewClassDesc& getClassDesc() = 0;
```

ausprogrammieren. Und in dem von dieser Methode zurückgelieferten NewClassDesc Objekt (siehe Listing 33 auf Seite 59) verbirgt sich letztendlich das Wissen über die Klassenvariablen und Vererbungsstrukturen der Klasse.

Eine andere Funktion, die ebenfalls von jeder von Serializable erbenden Klasse definiert werden muss, ist :

```
virtual Serializable* getNewObj() = 0;
```

Sie liefert einfach einen Zeiger auf ein mit new erzeugtes Objekt des eigenen Typs zurück. Sie ist notwendig, um aus dem Serialisierungsstrom gelesene Java Objekte in C++ zur Laufzeit anlegen zu können.

Bisher bezogen sich sämtliche Ausführungen der Einfachheit halber auf von Serializable abgeleitete Klassen. Bei über Hüllenklassen zu serialisiernden Klassen ist der Sachverhalt jedoch ähnlich, nur daß dort die Hüllenklasse von Serializable abgeleitet ist und somit dafür verantwortlich ist, sämtliche Informationen für die eingepackte (engl. *wrapped*) Klasse zur Verfügung zu stellen.

Mit den bisher vorgestellten Datenstrukturen und den beiden Funktionen getObjectByName() und getObjectByType() (Listing 22) ist jetzt sowohl die Abbildung von Java Klassen nach C++ als auch der umgekehrte Weg möglich.

Listing 22 : ../C++Ser/Serializable.cpp [Zeile 208 bis 223]

```
Serializable* ObjectFactory::getObjectByName(const string& name) {
  if (objectMap→count(name)) return (*objectMap)[name];
  else {
// ...
    return NULL;
  }
}

Serializable* ObjectFactory::getObjectByType(const string &type_name) {
  if (typeMap→count(type_name)) return (*typeMap)[type_name];
  else return NULL;
}
```

Dabei liefert die Funktion getObjectByName() für einen Java Klassennamen ein Objekt des ihm entsprechenden Typs auf C++ Seite, während getObjectByType() für ein C++ Objekt, dargestellt durch seinen internen Typnamen, ein C++ Objekt liefert, über das es serialisiert werden kann. Es kann sich, im einfachen Fall, bei dem zurückgelieferten Objekt um ein Objekt des gleichen Typs handeln, wie diesen das ursprüngliche Objekt hatte, wenn dieses von Serializable abgeleitet war, oder um ein Objekt der richtig parametrisierten Hüllenklasse andernfalls. Der interne, zu einem C++ Objekt obj gehörige Typname kann dabei

durch folgenden Funktionsaufruf als nullterminierte C-Zeichenkette[4] erhalten werden :

```
const char *typName = typeid(obj).name();
```

Diese beiden Funktionen werden dann auch symmetrisch zueinander in `ObjectInputStream` beziehungsweise `ObjektOutputStream` eingesetzt. Eine Instanz der Klasse `ObjectInputStream` ließt dabei Daten aus dem Serialisierungsstrom in dem zu jedem geschriebenen Objekt unter anderem dessen Klassenname vermerkt ist. Über diesen Namen und die Funktion `getObjectByName()` kann die Klasse `ObjectInputStream` dann ein diesem Objekt entsprechendes C++ Objekt erzeugen.

Bei der Ausgabe von C++ Objekten in den Serialisierungsstrom, kann die Klasse `ObjektOutputStream` dagegen über den `typeid()` Wert dieses Objektes und die Funktion `getObjectByType()` die Klasse feststellen, auf die dieses Objekt in Java abgebildet werden soll.

4.2 Init_Object : Automatisierung der Klassenregistrierung

Um den bisher besprochenen Vorgang der Registrierung einer Abbildung von C++ auf Java Objekte zu vereinfachen, wurde die parametrisierte Klasse `Init_Objekt` bereitgestellt. Sie ist in Listing 23 zu sehen.

Listing 23 : ../C++Ser/Serializable.hpp [Zeile 290 bis 325]

```
template <class T>
class Init_Object {
  static T t_VHS_t;
  static int count;
  string className;
public:
  Init_Object() {
// ...
    if (count-- > 0) {
// ...
      // istrstream name(t_VHS_t.getClassDesc().getClassName().c_str());
      // name >> className >> unused >> superClassName;
      className = t_VHS_t.getClassDesc().getClassName();
      Serializable::insertMapping(className,typeid(typename T::objType).name(),
                                  t_VHS_t.getNewObj() );
    }
// ...
  }
};
template <class T>
T Init_Object<T>::t_VHS_t; // Dynamically initialized at runtime since there
                                                                          ➥
```

[4] Es handelt sich dabei um eines der vielen Artefakte im neuen C++-Standard. Dort wird zwar eine `string`-Klasse definiert, diese wird jedoch nicht einmal in den zum Standard gehörigen Funktionen konsequent eingesetzt

Listing 23 : ../C++Ser/Serializable.hpp [Zeile 290 bis 325] (Fortsetzung)

```
template <class T>          // may be a constructor which must be called.
int Init_Object<T>::count = 1; // Statically initialized by compiler.
```

Ihre einzige Aufgabe besteht eben in der korrekten Anmeldung einer Klassenabbildung bei `Serializable::objectFactory`. Dazu muß einfach nur ein mit der anzumeldenden Klasse parametrisiertes Objekt dieser Klasse angelegt werden. Mit Hilfe des statischen Zähler `count`, der für jede Klasseninstantiierung vorhanden ist, kann das `Init_Objekt` dafür sorgen, daß es jeden Typ nur einmal anmeldet. Die Informationen, die es dazu benötigt, holt es sich aus der anzumeldenden Klasse selber, indem es eine statische Instanz davon anlegt und von dieser dann die schon erwähnte Methode `getClassDesc()` verwendet. Daraus folgt wiederum, daß sich eigentlich nur von `Serializable` abgeleitete Objekte registrieren lassen. Es funktioniert aber auch das Anmelden von Objekten eines anderen Typs, diese müssen jedoch über ihre Hüllenklassen angemeldet werden, die ja wiederum von `Serializable` abgeleitet sind.

Es reicht also aus, für jede zu serialisierende Klasse ein entsprechendes `Init_Objekt` zu instantiieren. Damit die Registrierung vor der ersten Verwendung erfolgt, genügt es zum Beispiel, nach der Klassendeklaration einer zur Serialisierung bestimmten Klasse `List` ein statisches, mit `List` parametrisiertes Initialisierungsobjekt anzulegen :

```
static Init_Object<List> foo;
```

Warum diese Lösung tatsächlich die rechtzeitige Registrierung garantiert, wird im nächsten Abschnitt erklärt werden.

4.2.1 Initialisierung globaler Objekte

Die spezielle Initilisierungsklasse `Init_Objekt` wurde deswegen definiert, um die Registrierung von Klassenabbildungen für den Anwender möglichst transparent und einfach zu gestalteten. Es ist fehlerträchtig und unbequem, wenn die korrekte Funktionsweise einer Bibliothek von dem Aufruf bestimmter Funktion, womöglich noch in einer festgelgten Reihenfolge, abhängig ist. Dies war jedoch in C oft die einzige Möglichkeit um globale Datenstrukturen mit nichtkonstanten Werten zu initialisieren. In C können globale Variablen nämlich nur statisch, also mit zur Übersetzungszeit bekannten Werten, initialisiert werden. In dem folgenden Beispiel ist die zweite Zeile ungültig, da sie zur Übersetzungszeit nicht ausgewertet werden kann :

```
double d1 = 0.123456789;
double d2 = sqrt(66.66); /* C Compile time Error */
```

Mit der Einführung von C++ und dem Ziel, Klassenobjekte genauso wie Basistypen zu behandeln traten jedoch Schwierigkeiten auf, die dazu führten, die Begrenzung der Initialisierung globaler Objekte auf konstante Werte aufzuheben. Dies ist etwa durch folgendes Beispiel (aus [Strou94]) leicht ersichtlich :

```
class Double {
                    ➥
```

```
  // ...
  Double(double);
};
Double d1 = 0.123456789;
Double d2 = sqrt(66.66);
```

Hier kann noch nicht einmal das Objekt `d1` statisch initialisiert werden, da es, trotz eines konstanten Initialisierers, den Aufruf eines Konstruktors und somit einen Funktionsaufruf erfordert, der natürlich nur dynamisch, also zur Laufzeit möglich ist.

Dies führte dazu, daß in C++ auch die dynamische Initialisierung von globalen Objekten zulässig wurde und als Folge davon aus Konsistenzgründen auch die dynamische Initialisierung von Basistypen erlaubt werden mußte.

Selbst Bjarne Stroustroup schreibt jedoch in [Strou94] daß die Auswirkungen dieser Erweiterung anfangs unterschätzt wurden. Sie führt nämlich dazu, daß unter Umständen noch vor der ersten Zeile des Hauptprogrammes im Prinzip beliebiger Programmcode zur Ausführung gelangen kann. Dies erfordert entweder einen speziellen Linker samt eines speziellen Startprogrammes, das vor dem Hauptprogramm ausgeführt wird oder den Einsatz von Zusatzprogrammen und mehreren Übersetzungsschritten. Eine interessante Diskussion hierüber und weiter sich daraus ergebende Probleme wie Inkonsistenzen im Ausnahmebehandlungsmechanismus kann in [Strou94], Kapitel 3.11.4 oder bei [Lipp96] Kapitel 6.1 nachgelesen werden.

Letztendlich hat sich die neue Semantik jedoch aufgrund ihrer Ausdrucksstärke und eleganten Mächtigkeit durchgesetzt. So müssen zum Beispiel die Standardein- und Standardausgabeströme `cin` und `cout`, bei denen es sich um globale Objekte handelt, vor ihrer ersten Verwendung mit den entsprechenden Dateideskriptoren initialisiert werden. Dieser Zeitpunkt der ersten Verwendung kann dabei durchaus schon vor dem Eintritt ins Hauptprogramm sein, etwa im Konstruktor eines anderen globalen Objektes.

Die Schwierigkeit liegt hierbei darin, daß der C++-Standard dynamische Initialisierung globaler Objekte zwar zuläßt, über die Reihenfolge der Initialisierung von solchen Objekten in unterschiedlichen Übersetzungseinheiten jedoch keine Aussage trifft ([ANSI-CPP],3.6.2). Innerhalb einer Übersetzungseinheit müssen Objekte jedoch in der Reihenfolge ihrer Deklaration initialisiert werden.

Die Lösung dieses Problemes wurde unter dem Namen Schwarzsche Zähler (engl. *Schwarz counters*) bekannt ([Schwarz]). Es handelt sich dabei um eine Klasse, die einen statischen Zähler besitzt und die bei ihrer ersten Instantiierung gewisse Initialisierungen vornimmt. Eine statische Variable dieses Typs wird in der Deklarationsdatei der Bibliothek angelegt, so daß jede Übersetzungseinheit, die diese Bibliothek verwendet, eine lokale Instanz der Klasse besitzt. Es spielt jetzt keine Rolle in welcher Reihenfolge die globalen Variablen der unterschiedlichen Übersetzungseinheiten dynamisch initialisiert werden, da jede von ihnen eine Instanz der Zählerklasse enthält, die die Bibliothek initialisiert.

Bei der im letzten Abschnitt vorgestellten `Init_Objekt` Klasse handelt es sich um nichts anderes als einen parametrisierten Schwarzschen Zähler. Richtig eingesetzt sorgt er dafür, daß die `ObjectFactory` Datenstruktur von `Serializable` automatisch noch vor dem Start des Hauptprogrammes mit allen eingebundenen Klassen initialisiert wird.

4.3 Die restlichen Methoden von `Serializable`

Neben `getNewObjekt()` und `getClassDesc()`, gibt es noch eine Reihe weiterer Methoden, die jede von `Serializable` abgeleitete Klasse implementieren sollte. Dazu zählt zum Beispiel die Funktion :

```
virtual unsigned int getObjSize() = 0;
```

die für jedes Objekt dessen Größe zurückliefert. Sie besteht also normalerweise aus dem einzigen Befehl :

```
return sizeof(*this);
```

Sie ist deswegen notwendig, um auch von über Zeiger auf die Basisklasse verwendeten Objekten deren korrekte Größe feststellen zu können. Diese Information wird von der statischen Funktion :

```
static unsigned int hash(void* obj, unsigned int size);
```

gebraucht, die für ein Objekt `obj` der Größe `size` eine Quersumme berechnet. Diese wird, zusammen mit der Adresse des Objektes, im Ausgabestrom als eindeutiger Schlüssel des Objektes betrachtet, nach dem entschieden wird, ob es selber oder nur eine Referenz auf ein zuvor schon serialisiertes Objekt geschrieben werden muß. Dabei ist es wichtig, daß die Quersumme über das vollständige Objekt und nicht nur für ein Teilobjekt gebildet wird. Die Funktion

```
static string getSUID(const string&);
```

liefert für einen Java Klassennamen dessen SUID als binär kodierte acht Byte lange Zeichenkette zurück. Diese Funktion wird hauptsächlich zum Schreiben von Arrays benötigt. Ihre Funktionsweise wird in Kapitel 4.9.2 erklärt. Über Hüllenklassen serialisierbare oder von `Serializable` abgeleitete Klassen definieren ihre eigen SUID normalerweise in ihrer `getClassDesc()` Methode.

Von Hüllenklassen, die im nächsten Abschnitt ausfürlich besprochen werden, ist zusätzlich noch die Funktion :

```
virtual unsigned int getObjAddress() { return this; }
```

so neu zu definieren, daß sie die Adresse des eingepackten Objektes anstatt der eigenen zurückliefert. Dies ist ebenfalls für die Buchhaltung geschriebener Objekte im `ObjektOutputStream` notwendig, wo ja nur das wirklich geschriebene Objekt von Bedeutung ist und nicht sein Hüllenobjekt. Als letztes bleibt noch eine Typdefinition, die von jeder von `Serializable` abgeleiteten Klasse `Obj` vorgenommen werden sollte :

```
typedef Obj objType;
```

Hüllenklassen sollten dabei anstatt ihres eigenen Typs den Typ des “eingehüllten” Objektes angeben. Der so definierte Typname `Obj::objType` wird von der `Init_Obj` Klasse verwendet, um den richtigen Typnamen für eine zu registrierende Klasse ausfindig zu machen. Dieses Typfeld und die Funktion `getObjAddress()` sind nur deswegen notwendig, um über

Hüllenklassen zu serialisierende und von Serializable abgeleitete Objekte konsistent behandeln zu können.

Neben den hier vorgestellten Methoden gibt es noch eine weitere Familie von Funktionen, die von erbenden Klassen nicht unbedingt implementiert werden müssen, da sie nicht abstrakt sind. Ob sie gebraucht werden hängt von der Art der Mitgliedsvariablen der Klasse ab. Zur Deserialisierung werden die Funktionen

```
virtual void setPrimValue(const string&, void*) {}
virtual void setObjValue(const string&, Serializable*) {}
virtual void readObject(ObjectInputStream*) {}
```

verwendet. setPrimValue() dient dabei zum Setzen einer Variablen von primitivem Datentyp. Das erste Argument enthält dabei den Namen der Mitgliedsvariablen, das zweite einen void Zeiger auf das richtig getypte Datum selber. Der Umweg über void Zeiger mußte hier deswegen gegangen werden, weil virtuelle Funktionen in C++ nicht parametrisiert werden können. Eine Lösung diese Problems wird in Kapitel 6 vorgestellt. Die gleiche Funktionalität erfüllt setObjValue() für Objektdatentypen, während die Funktion readObject() für das Lesen von mittels writeObject() auf Javaseite zusätzlich geschriebenen Daten dient (siehe dazu Abschnitt 4.6.1 auf Seite 57).

Symmetrisch zu diesen Funktionen gibt es die zum Serialisieren verwendeten Funktionen

```
virtual string getPrimValue(const string&) {}
virtual Serializable* getObjValue(const string&) {}
virtual void writeObject(ObjectOutputStream*, string = "") {}
```

Eine Besonderheit stellt hierbei die getPrimValue() Methode dar. Sie kodiert die verlangte Mitgliedsvariable gleich in ihre binäre Form und liefert sie als Zeichenkette zurück. Für diese Kodierung stehen dem Programmierer allerdings bereits entsprechende Hilfsfunktionen zur Verfügung, die nur aufgerufen werden müssen. Ihre Deklaration ist in Listing 24 zu sehen.

Listing 24 : ../C++Ser/Serializable.hpp [Zeile 44 bis 60]

```
string encodeUTF(const string&);
string encodeShort(short);
string encodeInt(int);
string encodeLong(Long);
string encodeFloat(float);
string encodeDouble(double);
string encodeBool(bool);
string encodeByte(char);
string encodeChar(unsigned char);
string encodeBasic(short s);
string encodeBasic(int i);
string encodeBasic(Long l);
string encodeBasic(float f);
string encodeBasic(double d);
string encodeBasic(bool b);
string encodeBasic(char c);
string encodeBasic(unsigned char c);
```

4.4 Hüllenklassen und -funktionen

In den letzten Kapiteln war des öfteren von Hüllenklassen die Rede. Hier soll nun erklärt werden, wie sie funktionieren und wie mit ihrer Hilfe in der Bibliothek der nonintrusiv Ansatz der Objektserialisierung realisiert wurde. Der Grund, weswegen sie überhaupt eingesetzt werden, ist die Tatsache, daß nicht alle Klassen die serialisiert werden sollen, von `Serializable` abgeleitet werden können. Als Beispiel sei hier die zur C++-Standardbibliothek gehörige `string` Klasse genannt, anhand derer auch der Einsatz der Hüllenklassen demonstriert werden soll.

Listing 25 : ../C++Ser/Serializable.hpp [Zeile 117 bis 137]

```
template <class T>
class GenericWrapper : public T, public Serializable {
private:
  void *objAddress;
public:
  GenericWrapper() {}
  GenericWrapper(T*);
  GenericWrapper(T&);
  // inherited from "Serializable"
  typedef T objType;
  virtual void setPrimValue(const string&, void*) { cerr << "ERROR1\n"; }
  virtual void setObjValue(const string&, Serializable*);
  virtual string getPrimValue(const string& s = "") { cerr << "ERROR2\n"; }
  virtual Serializable* getObjValue(const string&) { cerr << "ERROR3\n"; }
  virtual Serializable* getNewObj() { return new GenericWrapper<T>(); }
  virtual void* getObjAddress() { return objAddress; }
  virtual unsigned int getObjSize() { return sizeof(T); }
  virtual void readObject(ObjectInputStream*) { cerr << "ERROR4\n"; }
  virtual void writeObject(ObjectOutputStream*, string = "") { cerr << "ERROR5\n"; }
  virtual const NewClassDesc& getClassDesc() { cerr << "ERROR5\n"; }
};
```

In Listing 25 ist die Definition der hier verwendeten Hüllenklasse zu sehen. Es handelt sich dabei um eine parametrisierte Klasse, die somit einfach für verschiedene Typen instantiiert werden kann. Dabei ist `GenericWrapper` sowohl von `Serializable` als auch von der sie parametrisierenden Klasse abgeleitet. Dies hat den Vorteil, daß die Klasse innerhalb des `ObjektInputStream` wie eine von `Serializable` abgeleitete Klasse verwendet werden kann. Sobald das “eingehüllte” Objekt eingelesen worden ist, kann das Hüllenklassenobjekt an eine Variable des “eingehüllten” Typs zugewiesen werden, wobei die ganze Hüllenklasseninformation verlorengeht und nur noch die Daten des Zieltyps übrigbleiben.

Genauso kann im Ausgabestrom `ObjektOutputStream` ein nicht von `Serializable` abgeleitetes Objekt in sein entsprechendes Hüllenklassenobjekt umgewandelt werden, welches danach ganz normal serialisiert werden kann. Die Umwandlung kann entweder durch Zuweisung oder durch Initialisierung erfolgen. Die Hüllenklasse merkt sich dabei die Adresse des ursprünglichen Objektes und gibt sie durch die von `Serializable` geerbte virtuelle Me-

thode `getObjAddress()` aus.

Überhaupt verhält sich die Hüllenklasse völlig transparent. Sie implementiert die von `Serializable` geerbten virtuellen Funktionen derart, daß sich ihre Ergebnisse auf das eingehüllte Objekt auswirken, welches im Gegenzug so behandelt werden kann, als hätte es von `Serializable` geerbt.

Die meisten Funktionen der Hüllenklasse sind jedoch leer und müssen für jeden instantiierten Typ entsprechend ausprogrammiert werden. Dies ist in C++ durch die sogenannte Templatefunktionsspezialisierung möglich. Die leeren Methoden werden zwar vom Compiler für jeden Typparameter instantiiert, dies passiert jedoch nur, wenn es für die Funktion keine entsprechende Spezialisierung gibt, die falls vorhanden, höhere Priorität besitzt als eine implizit instantiierte Funktion. Deswegen ist es wichtig, daß die Spezialisierung vor ihrer ersten Verwendung und damit vor ihrer impliziten Instantiierung zumindest deklariert wird ([ANSI-CPP], Kap. 14.7.3). Die Funktion `getClassDesc()` für `string` wurde dabei folgendermaßen definiert :

Listing 26 : ../C++Ser/Serializable.cpp [Zeile 310 bis 322]

```
template<>  // still not supported by all compilers
const NewClassDesc& GenericWrapper<string>::getClassDesc() {
  static NewClassDesc nCD;
  static bool init = false;
  if (!init) {
    init = true;
    nCD.setClassName("java.lang.String");
    nCD.setSerialVersionUID(encodeLong(-6849794470754667710LL));
    nCD.setClassDescFlags(SC_SERIALIZABLE);
    nCD.setSuperClassDesc(NULL);
  }
  return nCD;
}
```

Sie überdeckt dabei die vom Compiler automatisch erzeugte Funktion :

```
NewClassDesc& GenericWrapper<string>::getClassDesc() { cerr << "ERROR5n"; }
```

Es soll hier nicht verschwiegen werden, daß es sicherer wäre, alle Methoden von `GenericWrapper` nur zu deklarieren und nicht zu definieren. Dann würde der Übersetzer den Anwender jedoch zwingen, auch Funktionen, die er unter Umständen gar nicht braucht auszuprogrammieren. Dafür könnte es nicht vorkommen, daß die Definition einer Funktion vergessen wird. Als Kompromiß zwischen Anwenderfreundlichkeit und Sicherheit wurden die besagten Funktionen nicht nur deklariert, sondern derart definiert, daß sie eine Fehlermeldung ausgeben. Somit läßt sich erst einmal jede Instantiierung von `GenericWrapper` übersetzen. Wenn danach die besagten Fehlermeldungen generiert werden, können die vergessenen Funktionsdefinitionen nachgeholt werden.

Neben `GenericWrapper` gibt es in der Bibliothek noch eine weitere Hüllenklasse die für die einzelnen Basistypen instantiiert wird und die jeweils für diese spezialisierte `getClassDesc()` Methoden besitzt. Sie ist nur von `Serializable` abgeleitet, da von primitiven

Datentypen nicht geerbt werden kann, und nur für den internen Gebrauch in der Bibliothek gedacht.

Listing 27 : ../C++Ser/Serializable.hpp [Zeile 197 bis 204]

```
template <class T>
class GenericBasicWrapper : public Serializable {
public:
  typedef T objType;
  virtual Serializable* getNewObj() { return new GenericBasicWrapper<T>(); }
  virtual unsigned int getObjSize() { return sizeof(T); }
  virtual const NewClassDesc& getClassDesc();
};
```

Sie wird verwendet um auf konsistente Art und Weise auch von den Basistypen die entsprechenden Java Namen erhalten zu können. Diese werden zum Beispiel benötigt, wenn die Klassennamen zu schreibender Arrays generiert werden müssen.

Außer den Hüllenklasse, gibt es in der Bibliothek auch noch eine generische, globale Funktion, die häufig verwendet wird. Ihre Hauptaufgabe ist es vor allem, den Benutzer bei der Ausgabe von Objekten zu unterstützen und diese für alle unterstützten Typen konsistent zu gestalten. Wie in Listing 17 auf Seite 34 zum Beispiel zu sehen war, konnten alle Datentypen, ohne Unterschied ob sie von Serializable abgeleitet waren, über die `writeObject()` Methode von `ObjectOutputStream` ausgegeben werden. Hinter dieser Funktionalität steckt die Funktion `wrap()` aus Listing 28. Es handelt sich dabei um eine parametrisierte Funktion, die für ein Argument beliebigen Typs einen `Serializable` Zeiger zurückliefert.

Listing 28 : ../C++Ser/Serializable.hpp [Zeile 176 bis 190]

```
template <class T>
Serializable* wrap(T* obj) throw (NotSerializableException) {
  Serializable * serObj;
  if ((serObj = Serializable::getObjectByType(typeid(*obj).name())) ≠ NULL ) {
    Serializable *ser = serObj→getNewObj();
    if (ser == ser→getObjAddress()) {
      delete ser;
      return (Serializable*)obj;
    } else {
      ser→setObjValue("", (Serializable*)obj);
      return ser;
    }
  }
  else throw NotSerializableException();
}
```

Für Objekte, die von `Serializable` abgeleitet sind, wird einfach das Objekt selber zurückgeliefert. Für Objekte eines anderen Typs wird eine Instanz der richtigen Hüllenklasse zurück-

geliefert, der das Objekt selber vorher zugewiesen wurde.

Um dies zu bewerkstelligen, verwendet wrap() die schon vorgestellte Funktion getObjectByType() aus Serializable und die Tatsache, daß die getObjAdress() Funktion für ein von Serializable abgeleitetes Objekt dessen eigene Adresse zurückliefert, während dies bei Hüllenklassenobjekten nicht der Fall ist. Bei ihnen liefert diese Funktion die Adresse des "eingehüllten" Objektes zurück.

4.5 Die Klasse ObjectInputStream

Die Klasse ObjectInputStream(Listing 29) ist der Java Klasse mit gleichem Namen nachempfunden. Sie stellt Eingabefunktionen für alle Basistypen zur Verfügung. Anders als in Java ist dagegen die Methode zum Lesen von Objekttypen parametrisiert. Das heißt, daß der Benutzer nicht wie in Java das gelesene Objekt erst in den richtigen Typ umwandeln muß, dies wird automatisch versucht und im Fehlerfalle eine Ausnahme geworfen. Zur Erinnerung hier noch einmal diese schon in Listing 15 in Kapitel 3.2.2 vorgestellte Funktion.

```
template <class T>
T* ObjectInputStream::readObject(T* (&obj)) throw (ClassCastException) {
  Serializable* tmp = serialParse→object()→getSerializable();
  if (tmp == NULL) return (T*)NULL;
  obj = dynamic_cast<T*>(tmp);
  if (obj ≠ NULL )return obj;
  else throw ClassCastException();
}
```

Der hier eingesetzte dynamic_cast<Typ>(object) ist eine im neuen C++ Standard definierte dynamische Typumwandlung. Sie entspricht genau der in Kapitel 2.2 vorgestellten Java Typumwandlung und wird mit Hilfe der schon in 4.1 erwähnten Laufzeittypinformation (engl. *runtime type information - RTTI*) realisiert. Dabei wird in die für alle Klassen mit virtuellen Funktionen angelegte virtuelle Funktionstabelle (engl. *vtable*) ein zusätzliches Feld mit Typinformationen eingefügt. Alle Instanzen einer solchen Klasse enthalten einen Zeiger auf die ihrem Typ entsprechende vtable, und damit indirekt einen Zeiger auf ihre Typbeschreibung. Diese Information ist dabei unabhängig davon, ob das Objekt über einen seinem Typ entsprechenden Zeiger oder den Zeiger auf einen Basistyps angesprochen wird. Dadurch hat der dynamic_cast Operator zur Laufzeit die Möglichkeit, diese Informationen zu vergleichen und bei Übereinstimmung die Umwandlung vorzunehmen. Daß diese Art der Typumwandlung aufwendiger ist als eine statische, zur Übersetzungszeit vorgenommene, ist klar.

Listing 29 : ../C++Ser/ObjectInputStream.hpp [Zeile 23 bis 44]

```
class ObjectInputStream {
private:
  IstreamInput *input;
  DLGLexer *lex;
  ANTLRTokenBuffer *tokenBuffer;
                                                                    ➥
```

Listing 29 : ../C++Ser/ObjectInputStream.hpp [Zeile 23 bis 44] (Fortsetzung)

```
  ANTLRToken aToken;
  SerialParse *serialParse;
public:
  ObjectInputStream(istream&);
  string readString();
  bool readBoolean();
  char readChar();
  char readByte();
  short readShort();
  int readInt();
  Long readLong();
  float readFloat();
  double readDouble();
  void* skipObject();
  template <class T> T* readObject(T*(&)) throw (ClassCastException);
  template <class T> T& readObject(T&) throw (ClassCastException);
};
```

Eine weitere nützliche Funktion ist `skipObject()`. Sie liest ein Objekt aus dem Eingabestrom, ohne es zu interpretieren. Dies ist aufgrund des Formates des Serialisierungsprotokolles möglich und kann zum Beispiel verwendet werden, um in C++ auch Objekte einzulesen, für die noch nicht alle Elemente serialisierbar gemacht wurden.

Initialisiert wird ein `ObjectInputStream` mit einem Standard C++ Eingabestrom. Dabei kann es sich sowohl um die Standardeingabe, als auch um eine Datei oder Netzwerkverbindung handeln, die jeweils die von der Klasse `istream` geerbten Funktionen implementiert. `ObjectInputStream` verwendet das ihm übergebene `istream` Objekt nur, um damit den verwendeten lexikalischen Entschlüssler (`DLGLexer`) zu initialisieren, der seinerseits nur die beiden Funktionen `istream::rdbuf()` und `istream.get()` benötigt.

4.6 Die Parserklasse `SerialParse`

Insgesamt stellt die Klasse `ObjectInputStream` eigentlich nur eine Schnittstelle für die von ANTLR erzeugte Parserklasse `SerialParse` dar. Sie soll in diesem Kapitel näher beschrieben werden.

Wie schon in Kapitel 2.4 erwähnt, wurde zum Parsen des Serialisierungsstromes der Parsergenerator ANTLR verwendet. Er erzeugt aus der ihm übergebenen Grammatik die von dem generischen Parser `ANTLRParser` abgeleitete Klasse `SerialParse`. Zur Veranschaulichung dieses Vorganges soll hier die Vorgehensweise anhand der Produktion `newObject` gezeigt werden. Die Struktur der ANTLR übergebenen Grammatik entspricht weitestgehend der von Sun angegebenen Originalgrammatik aus Anhang A. Dort ist zum Beispiel `newObject` folgendermaßen definiert :

```
newObject:
        TC_OBJECT classDesc newHandle classdata[]
```

Daraus wird in der Grammatikdefinitionsdatei folgende Definition :

Listing 30 : ../C++Ser/serProto.g [Zeile 331 bis 358]

```
newObject > [ Object *obj ]
        : << unsigned char flags; NewObject *nO;  int handle; >>
          << LT(1), TC_OBJECT() >>?
          BYTE
          <<
            nO = new NewObject;
          >>
          classDesc > [$obj]  newHandle[nO] > [handle]
          <<
// ...
            if (NewClassDesc *nCD = dynamic_cast<NewClassDesc*>($obj)) {
              nO->setClassDesc(nCD);
              nO->setSerializable(
                  Serializable::getObjectByName(nCD->getClassName())
                  ->getNewObj());
            }
            else {
              nO->setClassDesc((NewClassDesc*)NULL);
            }
            nO->setHandle(handle);
            $obj = nO;
          >>
          classdata[nO]
        ;
```

Die erste Zeile definiert dabei die Prodiktion `newObject`, die keine Argumente besitzt und einen Zeiger auf `Object`, eine selber definerte C++ Klasse, zurückliefert. In der zweiten Zeile werden die lokalen Variablen der aus dieser Produktion erzeugten Funktion deklariert. In der dritten Zeile folgt dann ein semantisches Prädikat, die Bedingung die erfüllt sein muß, damit die Produktion überhaupt ausgeführt wird. Es besagt, daß bei Vorausschau von einem Zeichen, das Prädikat `TC_OBJECT()` erfüllt sein muß, wobei es sich bei `TC_OBJECT()` um die einfache Funktion

```
int TC_OBJECT() { return (*(unsigned char*)LT(1)->getText() ==  ::TC_OBJECT); }
```

handelt, die das Ergebnis des Vergleiches des nächsten Eingabezeichens mit der `char`-Konstanten `TC_OBJECT` zurückliefert. Danach folgt der Produktionskörper, in dem Grammatikelemente und C++ Anweisungen in belieber Reihenfolge vorkommen können. C++ Code ist dabei in Klammern aus "<<" und ">>" einzuschachteln. Die Zeile

```
classDesc > [$obj]  newHandle[nO] > [handle]
```

besagt dabei zum Beispiel, daß zuerst die Produktion `classDesc` aufgerufen wird, deren Rückgabewert in `obj` gespeichert wird. Danach wird die Produktion `newHandle` mit `nO` als

Argument aufgerufen und deren Rückgabewert in `handle` abgespeichert.

Interessant ist der darauf folgende C++ Block. Hier kann die Funktion, nachdem die Klassenbeschreibung `classDesc` und somit auch der Name des im Eingabestrom folgenden Objektes gelesen wurde, mit Hilfe der statischen Methode `getObjectByName()` von `Serializable` ein dem gelesenen Java Objekt entsprechendes C++ Objekt erzeugen. Dieses neu erzeugte Objekt wird sodann der `classdata` Produktion übergeben, in der die gelesenen Klassenvariablen mit Hilfe der in Kapitel 4.3 besprochenen `Serializable`-Hilfsfunktionen gesetzt werden.

Aus dieser Definition macht ANTLR schließlich folgende C++ Funktion, die hier leicht vereinfacht dargestellt wird, da sie in Wirklichkeit viele Kommentare enthält und unübersichtlich formatiert ist.

```
Object* SerialParse::newObject(void)
{
  Object *      _retv;
  unsigned char flags;
  NewObject     *nO;
  int           handle;

  if (!(LT(1), TC_OBJECT() )) {zzfailed_pred(" LT(1), TC_OBJECT() ");}
  zzmatch(BYTE);
  consume();

  nO = new NewObject;
  _retv  = classDesc();
  handle  = newHandle( nO );

  if (NewClassDesc *nCD = dynamic_cast<NewClassDesc*>(_retv)) {
    nO→setClassDesc(nCD);
    nO→setSerializable(
    Serializable::getObjectByName(nCD→getClassName())→getNewObj());
  }
  else nO→setClassDesc((NewClassDesc*)NULL);

  nO→setHandle(handle);
  _retv = nO;
  classdata( nO );
  return _retv;
}
```

Die ersten drei Zeilen nach der Variablendeklaration prüfen das semantische Prädikat und konsumieren im Erfolgsfall das nächste Token aus dem Eingabestrom. Sämtliche Produktionsaufrufe wurden in C++ Methodenaufrufe umgewandelt.

4.6.1 Umkehrung des Kontrollflusses : Benutzerdefiniertes Parsen

Normalerweise läuft die Richtung des Kontrollflusses vom Hauptprogramm über den `ObjectInputStream`zu der Parserklasse `SerialParse` (Abbildung 4.2). In dieser wird der Grammatik entsprechend das nächste Objekt aus dem Eingabestrom gelesen, wobei es durchaus zu Rekursionen kommen kann. Dies liegt daran daß ein Objekt wiederum andere Objekte als Mitgliedsvariablen enthalten kann.

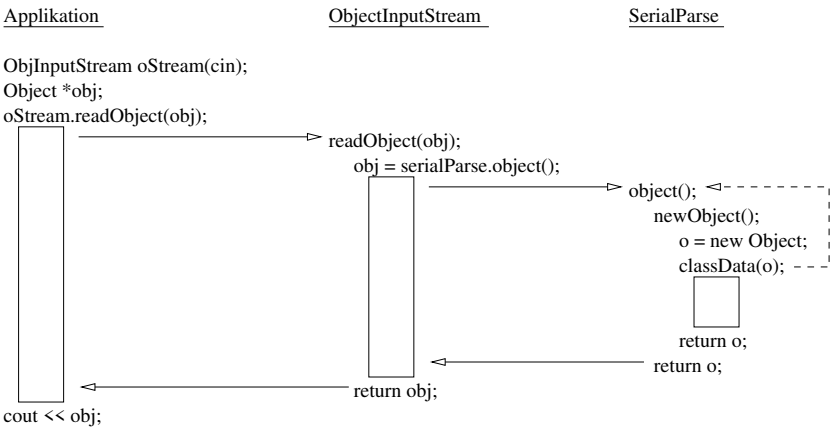

Abbildung 4.2: Der normale Kontrollfluß beim Lesen eines Objektes aus dem Serialisierungsstrom. Rekursionen gibt es nur innerhalb der Klasse `SerialParse`. Sie sind in der zugrundeliegenden Grammatik begründet.

Enthält ein Objekt jedoch mit `writeObject()`[5] geschriebene Daten, so können diese alleine mit Hilfe des Serialisierungsprotokolles nicht gelesen werden, da dieses nur Informationen über Mitgliedsvariablen und Erbschaftsbeziehungen des Objektes enthält. Nur das Objekt selber kennt das Format der von ihm zusätzlich geschriebenen Daten und nur das Objekt selber kann diese auch lesen. Es ist also notwendig, während des Parsens den Kontrollfluss an das Objekt zu übergeben, welches danach selber im gleichen Eingabestrom seine Daten einließt und die Kontrolle wieder an den Parser zurückgibt.

Dies Problem kann mit ANTLR und dem von ihm erzeugten *recursive descent* Parser elegent gelöst werden. Als Beispiel sei hier die Produktion `objectAnnotation` gezeigt :

Listing 31 : ../C++Ser/serProto.g [Zeile 392 bis 399]

```
objectAnnotation [ NewObject *nO ]
        : endBlockData                    // contents written by writeObject
        | ≪  LT(1), !TC_ENDBLOCKDATA() ≫?
          ≪
                                                                          ➥
```

[5] Es handelt sich hierbei um die `writeObject()` Methode aus `Serializable`. Sie ist nicht zu verwechslen mit der `writeObject()` Methode von `ObjectOutputStream`. Das gleiche gilt für die entsprechenden `readObjekt()` Methoden. Diese Namenskollision stammt aus Java und wurde von dort übernommen.

Listing 31 : ../C++Ser/serProto.g [Zeile 392 bis 399] (Fortsetzung)

```
        nO→getSerializable()→readObject(father);
    ≫                          // we let the Object itself parse the
    endBlockData               // 'contents'
;
```

Sie wird aufgerufen, wenn ein Objekt externe, also von seiner `writeObject()` Methode geschriebene Daten enthält. Sie ruft ihrerseits die `readObject()` Methode des gelesenen Objektes auf, der als Argument der aktuelle Eingabestrom übergeben wird. Die `readObject()` Methode des Objektes ließt wieder ganz normal aus dem ihm übergebenen `ObjectInputStream` (Abbildung 4.3). Diese Rekursion hat eine ganz andere Qualität als die Rekursion innerhalb von `SerialParse`. Während letztere nämlich in der Grammatik selber begründet ist, wird erstere von der Klassendefinition des betreffenden Objektes gesteuert.

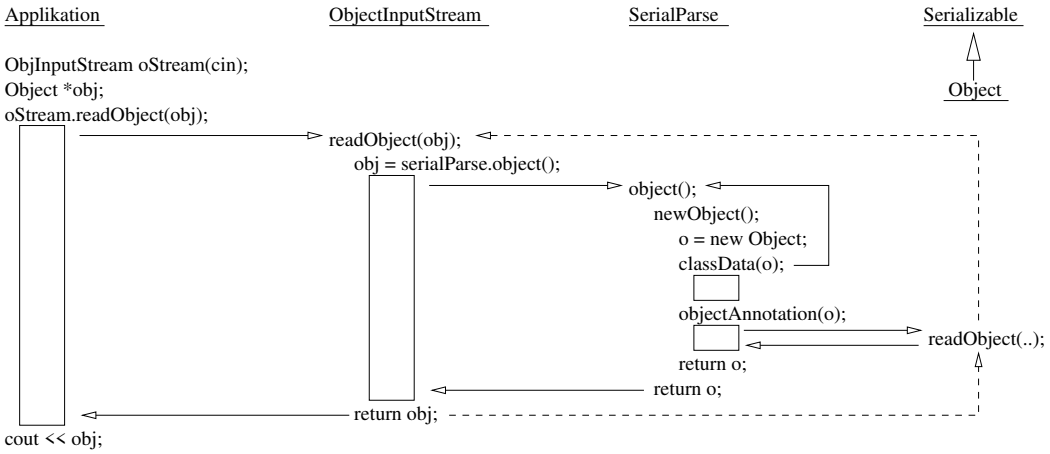

Abbildung 4.3: Der Kontrollfluß beim Lesen von mittels der `writeObject()` Methode des Objektes geschriebenen, zusätzlichen Daten. Die neue Rekursion wird mit gestrichelten Pfeilen dargestellt.

4.7 Die Hilfsklasse `Object`

Für die Klasse `SerialParse` wurden einige Hilfsklassen definiert, die hauptsächlich der Parameterübergabe zwischen den verschiedenen, auf Funktionen abgebildeten Produktionen dienen. Sie sind alle von der Klasse `Object` abgeleitet, die ihre gemeinsamen Eigenschaften zusammenfaßt.

Listing 32 : ../C++Ser/SerialObject.hpp [Zeile 38 bis 48]

```
class Object {
private :
  int myHandle;
  Serializable *realObject;
                                                              ➥
```

Listing 32 : ../C++Ser/SerialObject.hpp [Zeile 38 bis 48] (Fortsetzung)

```
public :
  int getHandle() const { return myHandle; }
  void setHandle(int h) { myHandle = h; }
  Serializable *getSerializable() const { return realObject; }
  void setSerializable(Serializable *s) { realObject = s; }
  virtual void dummy() {} // to enable RTTI
};
```

Dabei handelt es sich zum einen um ein während des Dekodierens vergebenes *handle* und zum anderen um einen Zeiger auf ein `Serializable` Objekt. Eine *handle*-Nummer wird im Serialisierungstrom nicht nur für Objekte selber, sonder zum Beispiel auch für Klassenbeschreibungen oder Zeicheketten vergeben. In denen von `Object` abgeleiteten Objekten wird während des Parsens nach und nach die ganze Information die für ein Objekt verfügbar wird gesammelt und zwischen den unterschiedlichen Produktionen weitergereicht. In Abbildung 4.1 auf Seite 40 sind die sieben von `Object` abgeleiteten Klassen zu sehen. Sie entsprechen genau den alternativen Produktion von `object` aus der Serialisierungsgrammatik.

Die virtuelle Methode wird nur deswegen definiert, um den Übersetzter zu zwingen, für Objekte dieser Klasse eine virtuelle Funktionstabelle anzulegen. Sie ist notwendig, wenn für Objekte diese Typs Laufzeittypinformationen oder der `dynamic_cast<>()` Operator verwendet werden soll (siehe Kapitel 4.1 und 4.5).

Da in diesem Kapitel schon mehrfach angesprochen, soll hier `NewClassDesc` als Beispiel der von `Object` abgeleiteten Klassen vorgestellt werden.

Listing 33 : ../C++Ser/SerialObject.hpp [Zeile 52 bis 73]

```
class NewClassDesc : public Object {
private :
  string className;
  string serialVersionUID;
  unsigned char classDescFlags;
  mutable vector<FieldDesc> fields;
  mutable bool sorted;
  const NewClassDesc *superClassDesc;
public :
  NewClassDesc() : sorted(false), superClassDesc(NULL) {}
  const string& getClassName() const { return className; }
  void setClassName(const string& s) { className = s; }
  const string& getSerialVersionUID() const { return serialVersionUID; }
  void setSerialVersionUID(const string &sUID) { serialVersionUID = sUID; }
  unsigned char getClassDescFlags() const { return classDescFlags; }
  void setClassDescFlags(unsigned char c) { classDescFlags = c; }
  const NewClassDesc *getSuperClassDesc() const { return superClassDesc; }
  void setSuperClassDesc(const NewClassDesc *o) { superClassDesc = o; }
  const FieldDesc& getFieldDesc(int) const;
```

Listing 33 : ../C++Ser/SerialObject.hpp [Zeile 52 bis 73] (Fortsetzung)

```
  void appendFieldDesc(const FieldDesc&);
  int getNrOfFields() const { return fields.size(); }
};
```

Sie enthält alle von der JVM für eine Klasse bereitgestellten Daten, wie Klassenname, Vaterklasse, SUID und die Namen und Typen allere Klassenvariablen. Diese werden in einem Vektor von `FieldDesc` Objekten verwaltet, womit sie durch Definieren des "<"-Operators für `FieldDesc` un der generischen Sortierfunktion `sort()` aus der STL in die von Java vorgeschriebene Reihenfolge gebracht werden können.

Listing 34 : ../C++Ser/SerialObject.hpp [Zeile 14 bis 34]

```
class FieldDesc {
private :
  unsigned char typecode;
  string fieldName;
  string className;
public :
// ...
  const string& getClassName() const { return className; };
  void setClassName(const string& s) { className = s; }
  int operator<(const FieldDesc&) const;
};
```

Lising 34 zeigt die `FieldDesc` Klasse. Die ausgelassenen Methoden dienen lediglich zum Lesen beziehungsweise Setzten der privaten Klassenelemente.

4.8 Die Klasse `ObjectOuputStream`

Neben dem Eingabestrom gibt es natürlich auch einen Ausgabestrom in den C++ Objekte geschrieben werden können. Ziel der Entwicklung war es, eine weitestgehende Symmetrie zwischen Ein- und Ausgabe zu erreichen. Das heißt, daß Objekte die aus einem Serialisierungsstrom gelesen wurden, auch wieder serialisierbar sein müssen. Dies ist zum Beispiel der Grund, warum die Abbildung von Java Arrays auf über Zeiger realisierte C++ Arrays aufgegeben wurde. Während die eine Richtung der Abbildung einfach implementiert werden kann, ist in diesem Fall der andere Weg, von C++ nach Java, unmöglich.

Listing 35 : ../C++Ser/ObjectOutputStream.hpp [Zeile 18 bis 63]

```
class ObjectOutputStream {
private:
  ostream out;
// ...
```

➥

Listing 35 : ../C++Ser/ObjectOutputStream.hpp [Zeile 18 bis 63] (Fortsetzung)

```
  ostrstream blockBuffer;
  int blockSize;
  void flushBuffer(bool force = false);
  unsigned int handle;
  map<string, unsigned int> classDescMap;
  map<void*, unsigned int> handleMap;
  map<void*, unsigned int> hashMap;    // hashvalues for objects in 'handleMap'
public:
  ObjectOutputStream(ostream&);
  void flush() { flushBuffer(true); }
  void reset();
  bool writeBoolean(bool);
  bool writeChar(char);
  bool writeByte(char);
  bool writeShort(short);
  bool writeInt(int);
  bool writeLong(Long);
  bool writeFloat(float);
  bool writeDouble(double);
  bool writeString(string&);
  // output functions for objects derived from 'Serializable'
  bool writeObject(Serializable*);
  bool writeObject(Serializable&);
  // output functions for vectors which are mapped to java arrays.
  // until now, only one- and two-dimensional arrays are supported.
  template <class T> bool writeObject(vector<T>*);
  template <class T> bool writeObject(vector<T> v) { return writeObject(&v); }
  template <class T> bool writeObject(vector<T*>*);
  template <class T> bool writeObject(vector<T*> v) { return writeObject(&v); }
  template <class T> bool writeObject(vector<vector<T> >*);
  template <class T> bool writeObject(vector<vector<T> > v)
  { return writeObject(&v); }
  template <class T> bool writeObject(vector<vector<T*> >*);
  template <class T> bool writeObject(vector<vector<T*> > v)
  { return writeObject(&v); }
  // general output functions for all other objects
  template <class T> bool writeObject(T *o);
  template <class T> bool writeObject(T o) { return writeObject(&o); }
};
```

In der Definition der Klasse in Listing 35, sind deutlich die Parallelen zum Eingabestrom zu erkennen. Es gibt für sämtliche Basistypen die entsprechenden Ausgabeoperationen, und genauso wie der Eingabestrom mittels eines `istream` konstruiert werden kann, benötigt die Instantiierung eines Serialisierungstroms zur Ausgabe einen Standard C++ Ausgabestrom

ostream. Dabei ist es wiederum unerheblich, ob die Objekte in eine Datei oder in eine Netzwerkverbindung geschrieben werden.

Weiterhin fällt die hohe Anzahl an parametrisierten Ausgebemethoden für Objekte auf. Dies hat damit zu tun, daß für die Ausgabe von Vektoren eigene Methoden definiert werden mußten, da es für sie aus Effizienzgründen keine Hüllenklasse gibt. Desweiteren gibt es, ähnlich wie bei der Eingabe, sowohl Funktionen für Objekte als auch für Zeiger auf diese Objekte, wobei erstere jeweils nur die entsprechende Zeigerversion mit der Adresse des Objektes aufrufen. Dies führt dazu, daß eignetlich nur die Funktion :

```
bool writeObject(Serializable*);
```

wirklich in den Ausgabestrom schreibt, alle anderen Funktionen rufen nach einiger Verwaltungsarbeit letzendlich sie auf. Als nächstes sollen die Komponenten von writeObject() näher beschrieben werden.

Listing 36 : ../C++Ser/ObjectOutputStream.cpp [Zeile 94 bis 98]

```
flushBuffer(true);
if ((obj == NULL) || (obj→getObjAddress() == NULL)) {
  out ≪ TC_NULL;
  return true;
}
```

Als erstes wird geprüft, ob es sich bei dem zu schreibenden Objekt um einen NULL Zeiger handelt, in welchem Fall einfach die Konstante TC_NULL in den Ausgabestrom geschrieben wird. Dabei kann es sich bei den hier geschriebenen Objekten nicht nur um direkt von Serializable abgeleitete Objekte handeln, sondern auch um Hüllenklassen, bei der die echte Objektadresse mit getObjektAddress() abgefragt werden muß.

Listing 37 : ../C++Ser/ObjectOutputStream.cpp [Zeile 101 bis 119]

```
bool rewriteObject = false; // True if we rewrite an object with the same
                            // addres but a new value.
unsigned int hashC; // Hold the hash value of current object, since building
                    // it can be an expensive operation.
hashC = Serializable::hash(obj→getObjAddress(), obj→getObjSize());
map<void*, unsigned int>::iterator pos1;
if ((pos1 = handleMap.find(obj→getObjAddress())) ≠ handleMap.end()) {
  // object has already been written in this stream.
  // test if his value has changed since last write.
  rewriteObject = true;
  if (hashC == hashMap[obj→getObjAddress()]) {
    // no changes => now we write only a reference.
    out ≪ TC_REFERENCE;
    string s = encodeInt(pos1→second);
    out.write(s.data(), s.length());
    return true;
                                                                    ➥
```

Listing 37 : ../C++Ser/ObjectOutputStream.cpp [Zeile 101 bis 119] (Fortsetzung)

```
  }
  // object value has changed, so continue like with new objects !
}
```

Als nächstes wird geprüft, ob das Objekt schon einmal geschrieben wurde. Dazu wird zuerst seine Adresse mit der aller bisher geschriebenen Objekte verglichen. Wird eine Übereinstimmung festgestellt, dann wird als nächstes, mittels der statischen Methode `Serializable::hash()` eine Quersumme über das gesamte Objekt gebildet und nur wenn diese mit der beim letzten Schreiben gebildeten Quersumme übereinstimmt, sich das Objekt seitdem also nicht mehr verändert hat, wird eine Referenz auf sein erstes Vorkommen geschrieben.

Listing 38 : ../C++Ser/ObjectOutputStream.cpp [Zeile 123 bis 125]

```
stack<const NewClassDesc*> hierarchy;
// get classDescription object
const NewClassDesc *nCD = &obj→getClassDesc();
```

Da es sich bei dem zu schreibenden Objekt um eine abgeleitete Klasse handeln kann, wird ein Stapel angelegt, auf den dessen Klassenbeschreibungen und die sämtlicher Vaterklassen abgelegt werden können.

Listing 39 : ../C++Ser/ObjectOutputStream.cpp [Zeile 128 bis 140]

```
// special handling for strings since they have special representation in the
if (nCD→getClassName() == "java.lang.String") { // serialisation protocol !
  if (obj→getObjAddress() == NULL) out ≪ TC_NULL;
  else {
    handleMap[obj→getObjAddress()] = handle++;
    hashMap[obj→getObjAddress()] = hashC;
    string *str = dynamic_cast<string*>(obj);
    out ≪ TC_STRING;
    string s = encodeUTF(*str);
    out.write(s.data(), s.length());
  }
  return true;
}
```

Handelt es sich bei dem auszugebenden Objekt um eine Zeichenkette, dann wird sie sofort ausgegeben, da Zeichenketten, obwohl sie in Java normale Objekte sind, im Serialisierungsstrom ein spezielles Format haben.

Bei dem in Listing 40 zu sehenden Abschnitt handelt es sich um den längsten der Funktion. In ihm werden rekursiv die Klassenbeschreibung des Objektes und seiner Vaterklassen geschrieben, wobei dazu die vom Benutzer durch die überladene Funktion `getClassDesc()` zur Verfügung gestellten Informationen verwendet werden. Dieser Teil wird wegen seiner

Länge leicht verkürzt wiedergegeben.

Listing 40 : ../C++Ser/ObjectOutputStream.cpp [Zeile 143 bis 223]

```
else { // write classDescription and all of its superClassDescrip recursivly
  out ≪ TC_OBJECT;
  do {
    string className;//, unused, superClassName;
...
    // istrstream name(nCD->getClassName().c_str());
    // name >> className >> unused >> superClassName;
    className = nCD→getClassName();
    // Test, if classdescription has already been written earlier ...
    map<string, unsigned int>::iterator pos2;
    if ((pos2 = classDescMap.find(className))
        ≠ classDescMap.end()) {
      // ... so we can now write only a reference to it.
      out ≪ TC_REFERENCE ≪ encodeInt(pos2→second);
...
      hierarchy.push(nCD);
      break;
    }
    // enter new classDescription into the map of written classDescriptions
    classDescMap[className] = handle++;
    // Remember this classDesc on the stack for this object,
    // so we will be able to write it's data later.
    hierarchy.push(nCD);
    // now write the classDescription
    out ≪ TC_CLASSDESC;
...
    // write all dataField specifications of this class
    short nrOfFields = nCD→getNrOfFields();
    s = encodeShort(nrOfFields);
    out.write(s.data(), s.length());
    for (int x=0; x < nrOfFields; ++x) {
...
    }
    out ≪ TC_ENDBLOCKDATA;
    // if there's a superClassDescription we will write it too.
  } while (nCD = nCD→getSuperClassDesc());
  if (nCD == NULL) out ≪ TC_NULL;
}
```

Zu guter Letzt werden schließlich die das Objekt identifizierenden Daten wie seine Adresse und seine Quersumme vermerkt, und seine Mitgliedsdaten mit seinen in Kapitel 4.3 besprochenen und von `Serializable` geerbten Hilfsfunktionen ausgegeben. Sollte das Objekt über

externe Daten verfügen, so wird am Ende noch seine `writeObjekt()` Methode aufgerufen.

Listing 41 : ../C++Ser/ObjectOutputStream.cpp [Zeile 226 bis 255]

```
// insert the object adress into the handleMap
handleMap[obj→getObjAddress()] = handle++;
// insert the object's hash value into the handleMap
hashMap[obj→getObjAddress()] = hashC;
// now write recursively the class data of all superclasses (if any at all)
// and thereafter the class data of the object itself.
while (!hierarchy.empty()) {
  const NewClassDesc *nCD = hierarchy.top();
  hierarchy.pop();
    for (int x=0; x < nCD→getNrOfFields(); ++x) {
      const FieldDesc &fD = nCD→getFieldDesc(x);
      if ((fD.getTypecode() ≠ 'L') && (fD.getTypecode() ≠ '[')) {
        string s = obj→getPrimValue(fD.getFieldName());
        out.write(s.data(), s.length());
      }
      else if (fD.getTypecode() == '[') {
        obj→writeObject(this, fD.getFieldName());
      }
      else {
        Serializable *ser = obj→getObjValue(fD.getFieldName());
        if (ser ≠ NULL) writeObject(ser);
        else out ≪ TC_NULL;
      }
    }
    // Write external data of the Class
    if (nCD→getClassDescFlags() & SC_WRITE_METHOD) {
      obj→writeObject(this);
      out ≪ TC_ENDBLOCKDATA;
    }
}
```

Es ist zu beachten, daß Arrays, also Klassenelemente mit dem Java Typcode "[" über die `writeObject()` Methode des entsprechenden Objektes ausgegeben werden. Dies liegt daran, daß die für andere Objekttypen verwendete `getObjValue()` Methode nur `Serializable` Zeiger zurückliefern kann und es aber, wie schon erwähnt, für Vektoren keine Hüllenklasse gibt. Um dennoch die richtig parametrisierte `writeObject()` Methode des Ausgabestromes aufzurufen, muß deswegen die Ausführung in einer Objektmethode selber fortgesetzt werden, da nur diese den genauen Typ des Arrays kennen kann.

4.9 Probleme beim Lesen und Schreiben von Arrays

Die größten Schwierigkeiten machen sowohl beim Lesen als auch beim Schreiben die Arrays. Die Unterschiede zwischen Java und C++ Arrays wurden bereits in Kapitel 3.2.3 ausführlich beschrieben und dort wurde auch erklärt, warum in dieser Bibliothek Java Arrays auf die C++ Klasse `vector` abgebildet werden. Aber selbst diese Lösung birgt einige Tücken, auf die hier eingegangen werden soll.

4.9.1 Lesen und Erzeugen von Arrays

Da für jedes *n*-dimensionale Array ein Vektor verwendet wird, dessen Templateparameter ein *(n-1)*-dimensionaler Vektor ist, ist es notwendig, schon zur Übersetzungszeit den richtigen Vektortyp zu kennen. Dies führt dazu, daß die Bibliothek nur ein- und zweidimensionale Arrays unterstützt. Der Teil der Methode `values()` der Klasse `SerialParse`, der Arrays von Objekten erzeugt, sieht in Pseudocode etwa folgendermaßen aus :

```
if (ONE_DIMENSIONAL_ARRAY) {
  vector<Serializable*> *oArray = new vector<Serializable*>;
  for (int x=0; x < ARRAY_SIZE; x++)
    oArray→push_back(READ_OBJECT);
}
else if (TWO_DIMENSIONAL_ARRAY) {
  vector<vector<Serializable*> > *ooArray = new vector<vector<Serializable*> >;
  for (int x=0; x < nA→getSize(); x++)
    ooArray→push_back(*(vector<Serializable*>*) READ_OBJECT);
}
```

Jede Dimension, die größer als eins ist muß gesondert behandelt werden und den von `READ_OBJECT` gelieferten Zeiger in den richtigen Vektor Typ umwandeln und dereferenzieren, da für mehrdimensionale Arrays nur Vektoren von Vektoren und nicht Vektoren von Zeigern auf Vektoren eingesetzt werden. Wegen dieser Wertsemantik der Vektoren muß hier dereferenziert werden und genau deswegen muß hier auch der genaue Typ des von `READ_OBJECT` gelieferten Zeigers bekannt sein. Dies ist der Grund, weshalb es nicht möglich ist, eine allgemeine, rekursive Behandlungsroutine für *n*-dimensionale Arrays anzugeben. Es ist dagegen möglich, beliebig viele Dimensionen zu unterstützten, sofern für jede der entsprechende Code geschrieben wird, wobei sich jeweils nur die Deklaration des Vektors und die Typumwandlung ändern. Es gilt jedoch zu beachten, daß für jede neue Dimension acht neue Vektorklassen für die Basistypen und eine für die Objekttypen hinzukommen, wobei für jede Vektorklasse vom Übersetzer neuer Code generiert wird, da es sich ja bei `vector` um eine Templateklasse handelt. Dies kann zu beträchtlichen Übersetzungszeiten und Objektdateigrößen führen.

Bei der Verwendung von Zeigern wäre dieses Problem einfacher zu lösen. Der entsprechende Pseudocode würde so aussehen :

```
if (ONE_DIMENSIONAL_ARRAY) {
  Serializable **oArray = new Serializable*[size];
  for (int x=0; x < ARRAY_SIZE; x++)
                                                    ➥
```

```
    oArray[x] = READ_OBJECT;
}
else if (MORE_THEN_ONE__DIMENSIONAL_ARRAY) {
  void **vArray = new void*[size];
  for (int x=0; x < ARRAY_SIZE; x++)
    vArray[x] = (void*)READ_OBJECT;
}
```

Da in C/C++ Zeiger und Zeiger auf Zeiger alle die gleiche Größe haben und ineinander umgewandelt werden können, ist die Konstruktion *n*-dimensionaler Arrays auf diese Weise kein Problem (siehe dazu auch Abbildung 3.2 auf Seite 31). Die Methode `readObjekt()` des Eingabestromes muß natürlich am Ende den erhaltenen Zeiger in den richtigen Zieltyp umwandel. Für ein vierdimensionales `char` Array wäre dies äquivalent zu folgender Anweisungsfolge :

```
char **dim2 = new char*[5];
char **dim3 = new char*[5];
dim2[0] = (char*)dim3;
char **dim4 = new char*[5];
dim4[0] = new char[5];
dim3[0] = (char*)dim4;
char ****feld = (char****)dim2;
```

wobei danach zum Beispiel mit `feld[0][0][0][0] = 'a';` ganz normal auf dieses Array zugegriffen werden kann.

Neben der Schwierigkeit, daß die Typen aller zu lesenden Arrays bei der Vektor-Lösung schon zur Übersetzungszeit der Bibliothek bekannt sein müssen, gibt es damit noch weitere Probleme bei Typumwandlungen. Diese sind zum Beispiel in den `setObjectValue()` Methoden der gelesenen Objekte notwendig, da die Klasse `SerialParse` nur `Serialiazable` Zeiger für Objekttypen zurückliefert.

Zwar macht es keine Schwierigkeiten, einen `vector<Serializable*>` statisch in einen `vector<List*>` umzuwandeln, vorausgesetzt `List` ist eine von `Serializable` abgeleitete Klasse. Anders sieht es dagegen bei der Typumwandlung von `vector<GenericWrapper<string>*>` nach `vector<string*>` aus, wenn `GenericWarapper<>` die in Kapitel 4.4 beschriebene Hüllenklasse ist. Sie ist zwar syntaktisch möglich, semantisch jedoch unsinnig, da diesmal, anders als im obigen Fall, `string` nur eine von mehreren Basisklassen von `GenericWrapper<string>` ist. Das heißt, `string` ist ein Teilobjekt von `GenericWrapper<string>` und ein Zeiger auf ein `GenericWrapper<string>` Objekt wird bei einer Typumwandlung zu `string` im allgemeinen seinen Wert ändern um auf das `string` Unterobjekt von `GenericWrapper<string>` zu zeigen. Dies hängt mit der Anordnung von Unterobjekten bei mehrfacher Vererbung zusammen, die vom C++ Standard nicht festgelegt ist. Eine Diskusson dieser Problematik kann zum Beispiel in [Lipp96] Kapitel 3.4 nachgelesen werden.

Wird die Typumwandlung jetzt für den ganzen Vektor vorgenommen, so ist das Ergebnis zwar ein Vektor des neuen Typs, wobei aber die Vektorelemente selber nicht einzeln in den neuen Elementyp des Vektors umgewandelt werden. Das heißt, der Vektor kann am Ende ungültige Zeiger enthalten. Die Lösung dieses Problemes besteht darin, daß der Anwender in solch einem Fall in der besagten `readObjekt()` Methode der Klasse diese Typumwand-

lung für jedes einzelne Element selber vornehmen muß.

4.9.2 Schreiben von Arrays

Genauso wie beim Lesen von Arrays sind auch beim Schreiben einige Besonderheiten zu beachten. Dies hat erstens damit zu tun, daß es für Vektoren keine Hüllenklasse gibt. Dieses Problem ist mit der Definition entsprechender Ausgabefunktionen für Vektoren in `ObjectOutputStream` (Listing 35, Seite 60) nicht ganz behoben . Es bleibt dadurch immer noch die Anomalie bestehen, daß Mitgliedsvariablen, die Arraytyp haben nicht wie normale Objekte vom Ausgabestrom selber mit Hilfe der `getObjectValue()` Methode des Objektes ausgegeben werden können, sondern stattdessen auf dessen `writeObject()` Methode zurückgegriffen werden muß (siehe dazu auch Kapitel 4.8, Seite 65).

Während es für alle Ausgabefunktionen, die Objekte in den Serialisierungsstrom schreiben jeweils eine Variante für Zeiger und eine für Wertparameter gibt, ist im Falle von Templatefunktionen zu berücksichtigen, daß auch hier von jeder Methode jeweils zwei Varianten benötigt werden. Zwar kann eine Funktion

```
template <class T> writeObject(vector<T>);
```

sowohl mit einem Vektor aus Objekten als auch mit einem Vektor aus Zeigern aufgerufen werden. Während aber im ersten Fall `T` dem Elementtyp des Vektors entsprechen würde, hätte `T` im zweiten Fall den Typ eines Zeigers auf den Elementyp des Vektors. Innerhalb der Methode selber könnte nicht entschieden werden, ob ihr Templateparameter ein Zeiger- oder ein Objekttyp ist, da es in C++ keine Möglichkeit gibt festzustellen, ob es sich bei einem Typ um einen Zeigertyp handelt[6].

Deswegen empfiehlt es sich, eine weitere parametrisierte Funktion

```
template <class T> writeObject(vector<T*>);
```

zu definieren. Sie ist spezialisierter als die erste und wird für Vektoren aus Zeigern verwendet, wobei ihr Templateparameter jetzt kein Zeigertyp mehr ist.

Das weitaus größte Problem ergibt sich bei der Ausgabe von Arrays jedoch aus der Tatsache, daß jedes Array in Java ein eigener Typ ist. Das heißt, daß beim Schreiben für jedes Array der korrekte Typname erzeugt werden muß. Während dies beim Typnamen ohne Schwierigkeiten funktioniert, da sein Format in der Dokumentation der Funktion `java.lang.Class.getName()` in [JDK1.2] definiert ist, ist es für den SUID-Wert unmöglich, da er aus einer Quersumme über die entsprechende Java-Klasse gebildet wird.

Das Problem wird hier pragmatisch gelöst, auf eine Art und Weise, die sich in Zukunft je nach Bedarf erweitern und verbessern läßt. Dazu wird die statische Funktion `getSUID` aus `Serializable` verwendet. Sie liefert zu einem Klassennamen den entsprechende SUID-Wert, indem sie mittels des Systemaufrufes `popen()`[7] das Programm `serialver` des Java Development Kit mit dem Klassennamen als Parameter aufruft, und aus dessen Ausgabe den entsprechenden SUID-Wert liest. Dies setzt aber erstens voraus, daß Java auf dem

[6] Es ist zum Beispiel möglich, dies über den durch `typeid` erhaltenen internen Klassennamen herauszufinden, der bei Zeigern meistens mit einem großen `P` beginnt. Dies ist jedoch Implementations und Systemabhängig

[7] popen() ist ein UNIX Systemaufruf. Unter Windows 95/NT gibt es die dazu äquivalente _popen() Funktion

Rechner installiert ist, auf dem die Bibliothek gerade ausgeführt wird und stellt zweitens einen nicht unerheblichen Geschwindigkeitsverlust dar. Deswegen werden einmal gelesene Werte auch in einem assoziativen Array zwischengespeichert. Für eine Produktionsversion kann dieses Array zum Beispiel schon mit den benötigten Werten initialisiert werden, oder einmal berechnete Werte können schon während der Entwicklung in eine spezielle Datei geschrieben werden, die nach dem assoziativen Array aber vor dem Aufruf von `serialver` nach dem gewünschten Eintrag durchsucht wird.

5. Der praktische Einsatz der Serialisierungsbibliothek

In diesm Abschnitt soll an zwei Beispielen gezeigt werden, wie eigene C++ Klassen mit der hier beschriebenen Bibliothek serialisiert und auf entsprechende Objekte auf Java Seite abgebildet werden können. Dabei wird als erstes auf den Fall einer neu zu entwickelnden Klassenhierarchie eingegangen, deren Klassen von `Serializable` abgeleitet werden können. Im zweiten Beispiel wird gezeigt, wie auf C++ Seite eine Hüllenklasse geschrieben werden kann, welche die Java Klasse `Hashtable` auf den STL Container `map<>` abbildet.

5.1 Beispiel 1: Von `Serializable` abgeleitete Klassen

Es seien wiedrum die schon in Lising 8 auf Seite 13 gezeigten Java Klassen `List` und `Mist` vorausgesetzt. Um sie in C++ lesen zu können müssen dafür zwei Klassen mit Mitgliedsvariablen identischen Typs definiert werden, wobei die Basistypen entsprechend der Tabelle 3.1 auf Seite 28 abzubilden sind, während Arrays zu Vektorklassen werden und alle Objektvariablen in C++ grundsätzlich durch Zeiger auf Objekte dargestellt werden. Üblicherweise werden diese Klassen die gleichen Namen wie die Java Klassen erhalten, genauso wie die Mitgliedsvariablen die Namen ihrer Pendants auf Javaseite übernehmen werden. Dies ist jedoch nicht zwingend notwendig, da die Typabbildung nicht automatisch über die Klassen- oder Variablennamen, sondern über die virtuellen, von `Serializable` geerbten Methoden geschieht.

In Listing 14 auf Seite 25 wurden zwei geeignete C++ Klassen gezeigt, allerdings ohne die Deklarationen der geerbten virtuellen Methoden. Das soll hier nachgeholt werden :

Listing 42 : ../C++Ser/List.hpp [Zeile 27 bis 36]

```
// inherited from "Serializable"
typedef List objType;
virtual void setPrimValue(const string&, void*);
virtual void setObjValue(const string&, Serializable*);
virtual string getPrimValue(const string&);
virtual Serializable* getObjValue(const string&);
virtual Serializable* getNewObj() { return new List; }
virtual const NewClassDesc& getClassDesc();
virtual unsigned int getObjSize() { return sizeof(*this); }
                                                          ➥
```

Listing 42 : ../C++Ser/List.hpp [Zeile 27 bis 36] (Fortsetzung)

```
virtual void writeObject(ObjectOutputStream*, string = "");
```

Listing 42 zeigt den bisher fehlenden Teil der Klassendeklaration von `List`. Der entsprechende Ausschnitt von `Mist` wird hier nicht gesondert gezeigt, da er sich bis auf die Funktion `getNewObjekt()`, die natürlich ein neues `Mist` Objekt zurückliefert, nicht von dem in Listing 42 gezeigten unterscheidet.

Die C++ Klassen überladen die `readObjekt()` Methode von `Serializable` nicht, da die Javaklassen auch keine externe Daten schreiben. Die `writeObjekt()` Methode ist jedoch notwendig, um das eigenen Arrayfeld schreiben zu können. Der Grund weswegen Arrays, anders als andere Objekttypen, nicht mittels der `getObjValue()` Methode serialisiert werden können wurde in Kapitel 4.8 erklärt.

Als nächstes folgt die `getClassDesc()` Methode von `List`

Listing 43 : ../C++Ser/List.cpp [Zeile 17 bis 37]

```
const NewClassDesc& List::getClassDesc() {
  static NewClassDesc nCD;
  static bool init = false;
  if (!init) {
    init = true;
    nCD.setClassName("List");
    nCD.setSerialVersionUID(encodeLong(1));
    nCD.setClassDescFlags(SC_SERIALIZABLE);
    nCD.setSuperClassDesc(NULL);
    nCD.appendFieldDesc(FieldDesc("value_s",   "short"));
    nCD.appendFieldDesc(FieldDesc("value_i",   "integer"));
    nCD.appendFieldDesc(FieldDesc("value_l",   "long"));
    nCD.appendFieldDesc(FieldDesc("value_f",   "float"));
    nCD.appendFieldDesc(FieldDesc("value_d",   "double"));
    nCD.appendFieldDesc(FieldDesc("value_str", "java.lang.String"));
    nCD.appendFieldDesc(FieldDesc("next",      "List"));
    nCD.appendFieldDesc(FieldDesc("field",     "[[byte"));
    nCD.appendFieldDesc(FieldDesc("oField",    "[List"));
  }
  return nCD;
}
```

Bei ihrem ersten Aufruf, der normalerweise aus dem Konstruktur des entsprechenden `Init_Objekt` erfolgt, initialisiert sie ihre statische Variable vom Typ `NewClassDesc`. Danach, ebenso wie bei jedem weiteren Aufruf, wird eine konstante Referenz auf dieses `NewClassDesc` Objekt zurückgeliefert. Es enthält dabei sämtliche Typinformationen die zum Lesen oder Schreiben von Objektetn dieser Klasse notwendig sind. Dies ist zum einen der Klassenname. Er muß mit dem Klassennamen in der korrespondierenden Javaklasse übereinstimmen. Als nächstes folgt die binär kodierte SUID, bei der es sich um eine Java 64-bit Vorzeichen-

behaftete Ganzzahl handelt. Bei eigenen Klassen empfiehlt es sich, diese in Java selber zu definieren, um handlichere Werte zu haben. Ansonsten kann das Programm `serialver` verwenden werden um die SUID für eine gegebene Javaklasse ausfindig zu machen, oder die statische Methode `getSUID` aufgerufen werden, die dies zur Laufzeit tut.

Mittels `setClassDescFlags()` können mehrere, mit "oder" verknüpfte Schalter gesetzt werden, die die Art und Weise der Serialisierung beinflussen. Standardmäßig ist das `SC_SERIALIZABLE` für normale Serialisierung oder (`SC_SERIALIZABLE | SC_WRITE_METHOD`), wenn die Klassenobjekte zusätzliche Informationen mit der `writeObject()` Methode schreiben. Als nächstes kann die Klassenbeschreibung einer etwaigen Vaterklasse gesetzt werden, bevor mit der Methode `appendFieldDesc()` die Klassenvariablen und deren Typ gesetzt werden, wobei diese Angaben mit denen auf Javeseite übereinstimmen müssen. Insbesondere müssen die Typen von Arrays in dem in dem Dokumentation der Funktion `java.lang.Class.getName()` in [JDK1.2] spezifizierten Format angegeben werden.

Wie schon gesagt, müssen diese Namen nicht mit den tatsächliche in C++ verwendeten Variablennamen übereinstimmen. Sie müssen jedoch mit denen in den beiden als nächstes besprochenen Methoden `setPrimValue()` und `setObjValue()` verwendetetn Bezeichnernamen konsistent sein. Die Reihenfolge in der alle diese Angaben gemacht werden ist unerheblich, da sie von der `NewClassDesc` Klasse automatisch in die von Java verlangte Reihenfolge sortiert werden.

Als nächstes folgt die `getClassDesc()` Mathode von `Mist`

Listing 44 : ../C++Ser/List.cpp [Zeile 110 bis 126]

```
const NewClassDesc& Mist::getClassDesc() {
  static NewClassDesc nCD;
  static bool init = false;
  if (!init) {
    init = true;
    nCD.setClassName("Mist");
    nCD.setSerialVersionUID(encodeLong(1));
    nCD.setClassDescFlags(SC_SERIALIZABLE);
    nCD.setSuperClassDesc(
      &Serializable::getObjectByName("List")→getClassDesc());
    nCD.appendFieldDesc(FieldDesc("value_B",   "boolean"));
    nCD.appendFieldDesc(FieldDesc("value_b",   "byte"));
    nCD.appendFieldDesc(FieldDesc("value_c",   "char"));
    nCD.appendFieldDesc(FieldDesc("sField",    "[java.lang.String"));
  }
  return nCD;
}
```

Neben den unterschiedlichen Variablenamen besteht der einzige Unterschied zu der `List` Methode darin, daß `Mist` eine Vaterklasse hat und deren Klassenbeschreibun mit Hilfe der Funktion `setSuperClassDesc()` setzt. Dafür muß die Vaterklasse schon angemeldet sein. Dies kann durch die entsperchende Anordnug der `Init_Objekt` Objekte am Ende der Deklarationsdatei gewährleistet werden.

```
static Init__Object<List> __List__registration;
static Init__Object<Mist> __Mist__registration;
```

Die beiden Methoden `setPrimValue()` und `setObjValue()` werden von der Klasse `SerialParse` während des Deserialisierens aufgerufen, um die Mitgliedsvariablen des Objektes zu setzen, wobei der erste Parameter der Methode der Name der Variablen ist und der zweite ein Zeiger auf ein Datum des zu setzenden Typs.

Listing 45 : ../C++Ser/List.cpp [Zeile 41 bis 52]

```
void List::setPrimValue(const string& var, void *val) {
// ...
  if (var == "value_s") value_s = *(short*)val;
  else if (var == "value_i") value_i = *(int*)val;
  else if (var == "value_l") value_l = *(Long*)val;
  else if (var == "value_f") value_f = *(float*)val;
  else if (var == "value_d") value_d = *(double*)val;
}
```

Bei den Basisdatentypen wird der übergebene `void` Zeiger einfach statisch in den gewünschten Zieltyp umgewandelt. Bei den Objektdatentypen muß der Fall eins Nullzeigers abgefangen werden, der für Objekte einen gültigen Wert darstellt. Desweiteren garantiert hier der `dynamic_cast<>()`, daß jede Typumwandlung zur Laufzeit auf Korrektheit überprüft wird.

Listing 46 : ../C++Ser/List.cpp [Zeile 56 bis 76]

```
void List::setObjValue(const string& var, Serializable *val) {
// ...
  if (var == "next") {
    if ((next = dynamic_cast<List*>(val)) || (val == NULL)) ;
    else cerr << "Invalid value type for field : " << var << endl;
  }
  else if (var == "value_str") {
    if ((value_str = dynamic_cast<string*>(val)) || (val == NULL)) ;
    else cerr << "Invalid value type for field : " << var << endl;
  }
  else if (var == "field") {
    field = (vector<vector<char> >*) val;
  }
  else if (var == "oField") {
    oField = (vector<List*>*) val;
  }
}
```

Bei den entsprechenden Methoden für `Mist` ist eine Besonderheit zu beachten. Es wird als erstes jeweils die Methode der Vaterklasse aufgerufen. Dies ist notwendig, da auch für ein abgeleitetes Objekt die Daten der Basisklasse gesetzt werden können müssen.

Listing 47 : ../C++Ser/List.cpp [Zeile 130 bis 142]

```
void Mist::setPrimValue(const string& var, void *val) {
// ...
  // first we check, if the variable is defined in our base class
  List::setPrimValue(var, val);
  // if not, we check our variables
  if (var == "value_B") value_B = *(bool*)val;
  else if (var == "value_b") value_b = *(char*)val;
  else if (var == "value_c") value_c = *(char*)val;
}
```

Bei der Methode `setObjValue()` erscheint der in Kapitel 4.8 auf Seite 65 besprochene Fall, eines Arrays von über Hüllenklassen serialisierten Objekten. Deswegen kann hier nicht einfach der `Serializable` Zeiger umgewandelt werden, sondern es muß eine Typumwandlung für alle Arrayelemente vorgenommen werden.

Listing 48 : ../C++Ser/List.cpp [Zeile 146 bis 161]

```
void Mist::setObjValue(const string& var, Serializable *val) {
  // first we check, if the variable is defined in our base class
  List::setObjValue(var, val);
  // if not, we check our variables
  if (var == "sField") {
    // we get a vector of 'Serializable*' wherby in fact this 'Serializable*'
    // are 'GenericWrapper<string>*' pointer
    vector<Serializable*> *orig = (vector<Serializable*>*)val;
    sField = new vector<string*>;
    for (vector<Serializable*>::iterator pos = orig→begin();
         pos ≠ orig→end(); ++pos) {
      sField→push_back((string*)(GenericWrapper<string>*)(*pos));
    }
    delete orig;
  }
}
```

Genauso wie die `ObjectInputStream` Klasse Objektmethoden verwendet um die Daten des Objektes zu setzen, benötigt die `ObjectOutputStream` Klasse Methoden, die ihr die Klassenelemente eines zu serialisierenden Objektes in einem geeigneten Format zur Verfügung stellen. Analog zur Eingabe heißen diese Methoden hier `getPrimValue()` und `getObjectValue()`.

Listing 49 : ../C++Ser/List.cpp [Zeile 80 bis 93]

```
string List::getPrimValue(const string& name) {
  if (name == "value_s") return encodeBasic(value_s);
                                                    ➥
```

Listing 49 : ../C++Ser/List.cpp [Zeile 80 bis 93] (Fortsetzung)

```
  else if (name == "value_i") return encodeBasic(value_i);
  else if (name == "value_l") return encodeBasic(value_l);
  else if (name == "value_f") return encodeBasic(value_f);
  else if (name == "value_d") return encodeBasic(value_d);
  else return "";
}

Serializable* List::getObjValue(const string& name) {
  if (name == "next") return next;
  else if (name == "value_str") return wrap(value_str);
  else return NULL;
}
```

Für primitive Datentypen erwartet der Ausgabestrom eine Zeichenkette des binär kodierten Datums. Diese kann leicht mittels der Funktion `encodeBasic()`, die für alle primitiven Datentypen überladen ist, erzeugt werden. Für Objektdatentypen können die Objekte selber zurückgeliefert werden, sofern sie von `Serializable` abgeleitet sind, ansonsten können sie einfach mit Hilfe der Funktion `wrap()` in die ihnen zugehörigen Hüllenklasse eingepackt, zurückgegeben werden. Bei Unsicherheit über die Vererbungshierarchie, kann auch in allen Fällen `wrap()` verwendet werden, da `wrap()` von `Serializable` abgeleitete Objekte erkennt und sie unverändert zurückliefert, allerdings mit dem zusätzlichen Aufwand einer Typanfrage (RTTI) zur Laufzeit. Der Vollständigkeit halber hier noch die beiden `get-` Methoden der Klasse `Mist`.

Listing 50 : ../C++Ser/List.cpp [Zeile 165 bis 176]

```
string Mist::getPrimValue(const string& name) {
  string s;
  if ((s = List::getPrimValue(name)) ≠ "") return s;
  else if (name == "value_B") return encodeBool(value_B);
  else if (name == "value_b") return encodeByte(value_b);
  else if (name == "value_c") return encodeChar(value_c);
  else return "";
}

Serializable* Mist::getObjValue(const string& name) {
  return List::getObjValue(name);
}
```

Zum Schluß wird für diese Klassen noch jeweils eine Funktion für die Ausgabe der Mitgliedsvariablen von Arraytyp benötigt. Wie schon erwähnt, wird dazu die `writeObjekt()` Methode mißbraucht.

Listing 51 : ../C++Ser/List.cpp [Zeile 97 bis 106]

```
void List::writeObject(ObjectOutputStream *out, string name) {
  if (name == "field") out→writeObject(field);
  else if (name == "oField") out→writeObject(oField);
// ...
}
```

Deren einzige Aufgabe ist es, die `writeObject()` Methode des übergebenen `ObjectOutputStream` mit der richtigen Arrayvariablen aufzurufen. Aus den vorhandenen parametrisierten Funktionen, sucht der Überstetzer die richtige aus.

Listing 52 : ../C++Ser/List.cpp [Zeile 180 bis 191]

```
void Mist::writeObject(ObjectOutputStream *out, string name) {
  // first we check, if the variable is defined in our base class
  List::writeObject(out, name);
  // if not, we check our variables
  if (name == "sField") out→writeObject(sField);
// ...
}
```

5.2 Beispiel 2: Eine Hüllenklasse für `java.util.Hashtable`

In diesem zweiten Beispiel soll gezeigt werden, wie mit Hilfe einer Hüllenklasse eine bestehende Javaklasse auf eine bestehende C++ Klasse abgebildet werden kann, ohne eine dieser beiden Klassen zu ändern. Zur Demonstration wurde auf Javaseite die Klasse `java.util.Hashtable` und auf C++ Seite die Klasse `map` aus der Standard Template Library gewählt. Bei beiden Klassen handelt es sich um eine Implementierung assoziativer Arrays, allerdings mit unterschiedlicher Semantik. Während die Javaklasse nur Objektreferenzen speichern kann, handelt es sich bei der C++ Klasse um eine mit zwei Typargumenten parametrisierte Templateklasse. Das heißt, daß ein Typ der Klasse `map` von seinen Templateparametern abhängig ist und jede `map` Klasse mit einer neuen Parameterkombination auch einen neuen Typ besitzt. Dies ist in Java anders : unabhängig von den in ihr abgespeicherten Daten hat eine `Hashtable` immer den gleichen Typ.

Diese Unterschiede haben natürlich Implikationen auf die Art der benötigte Hüllenklasse. Da alle C++ Maps auf eine einzige Javaklasse abgebildet werden, wird nur ein `NewClassDesc` Objekt auf C++ Seite benötigt, das den entsprechenden Java Klassennamen, die SUID und die Beschreibung der Mitgliedsvariablen enthält. Trotzdem muß die Hüllenklasse parametrisiert sein, um sich mit allen `map` Typen instantiieren zu lassen.

Listing 53 : ../C++Ser/Serializable.hpp [Zeile 211 bis 220]

```
template <class T>
                                                              ➥
```

Listing 53 : ../C++Ser/Serializable.hpp [Zeile 211 bis 220] (Fortsetzung)

```
class GenericHTWrapper : public GenericWrapper<T> {
public:
  virtual void setPrimValue(const string&, void*) {}
  virtual string getPrimValue(const string&);
  virtual Serializable* getNewObj() { return new GenericHTWrapper<T>(); }
  virtual const NewClassDesc& getClassDesc();
  virtual void readObject(ObjectInputStream*);
  virtual void writeObject(ObjectOutputStream*, string = "");
};
```

Listing 53 zeigt diese `BasicHTWrapper` genannte Hüllenklasse. Sie ist von `GenericWrapper` abgeleitet und überlädt nur die von ihr selber gebrauchten virtuellen Funktionen. Es ist zu beachte, daß `GenericWrapper` nur einen Typparameter besitzt, nämlich den Typ der `map` Klasse, den sie "einhüllt". Die `map` Klasse definiert ihrerseits jedoch mittels `typedef` die von ihr verwendeten beiden Typparameter, so daß auf diese Weise auch innerhalb von `BasicHTWrapper` darauf zugegriffen werden kann.

Die schon erwänte `getClassDesc()` Methode sieht folgendermaßen aus :

Listing 54 : ../C++Ser/Serializable.hpp [Zeile 233 bis 247]

```
template <class T>
const NewClassDesc& GenericHTWrapper<T>::getClassDesc() {
  static NewClassDesc nCD;
  static bool init = false;
  if (!init) {
    init = true;
    nCD.setClassName("java.util.Hashtable");
    nCD.setSerialVersionUID("\x13\xbb\x0f\x25\x21\x4a\xe4\xb8");
    nCD.setClassDescFlags(SC_SERIALIZABLE | SC_WRITE_METHOD);
    nCD.setSuperClassDesc(NULL);
    nCD.appendFieldDesc(FieldDesc("threshold",   "integer"));
    nCD.appendFieldDesc(FieldDesc("loadFactor",   "float"));
  }
  return nCD;
}
```

Sie deklariert die beiden Mitgliedsvariablen `threshold` und `loadFactor` , wobei `loadFactor` ein Fließkommawert zwischen null und eins ist und `threshold` das Produkt aus der aktuellen Größe der `Hashtable` und `loadFactor`. Werden mehr als `threashold` Werte in die Hashtabelle eingefügt, dann wird sie vergrößert. Diese beiden Werte haben auf seiten der C++ `map` Klasse keine direkte Entsprechung, weswegen die `setPrimValue()` Methode leer bleibt. Ebenso werden die Methoden zum setzen und lesen von Objektwerten nicht gebraucht, da `Hashtable` keine entsprechenden Mitgliedsvariablen definiert. Diese Methoden werden für ein `Hashtable` Objekt niemals aufgerufen, während die `setPrimValue()` Metho-

de als leere Funktion definiert wurde, um zu vermeiden, daß während des Deserialisieren eines Hashtable Objektes die von GenericWrapper() geerbte Funktion aufgerufen wird, die eine Fehlermeldung ausgibt (siehe Kapitel 4.4, Seite 51).

Listing 55 : ../C++Ser/Serializable.hpp [Zeile 224 bis 229]

```
template <class T>
string GenericHTWrapper<T>::getPrimValue(const string& name) {
  if (name == "threshold") return encodeInt(int(2*size()*0.75));
  else if (name == "loadFactor") return encodeFloat(0.75);
  else return "";
}
```

Die Methode getPrimValue() muß dagegen implementiert werden, um ein map Objekt so serialisieren zu können, daß es auf Javaseite als Hashtable erkannt werden kann. Sie liefert einen festen loadFactor Wert von 0.75 zurück und berechnet threshold als 0.75 mal der doppelten Anzahl der Einträge der aktuellen map. Diese Werte entsprechen den Standardeinstellungen auf Java Seite.

Die ganzen Einträge des assoziativen Speichers werden jedoch mittels der writeObject() Methode geschrieben und müssen demnach auch mit der readObject() Methode des Objektes selber deserialisiert werden. Dies ist auch der Grund weswegen in der Klassenbeschreibung in Listing 54 zusätzlich zu SC_SERIALIZABLE der Schalter SC_WRITE_METHOD gesetzt wurde. Die readObjet() Methode sieht folgendermaßen aus :

Listing 56 : ../C++Ser/Serializable.hpp [Zeile 251 bis 265]

```
template <class T>
void GenericHTWrapper<T>::readObject(ObjectInputStream* in) {
  int capacity = in→readInt();
  int nrOfElems = in→readInt();
  for (int x = 0; x < nrOfElems; ++x) {
    typedef typename T::key_type key_type;
    typedef typename T::data_type data_type;
    typedef typename T::value_type value_type;
    key_type *k;
    in→readObject(k);
    data_type d;
    in→readObject(d);
    insert(value_type(*k, d));
  }
}
```

Zuerst wird die Größe der Hashtable gelesen und dann die Anzahl der Elemente die sie enthält, wobei es sich bei jedem Element um ein Schlüssel/Wert Paar handelt. Danach werden in einer Schleife alle Elemente aus dem Eingabestrom gelesen. Auf die entsprechenden Typen der C++ map kann dabei mittels des typname Schlüsselwortes zugegriffen werden.

Listing 57 : ../C++Ser/Serializable.hpp [Zeile 269 bis 282]

```
template <class T>
void GenericHTWrapper<T>::writeObject(ObjectOutputStream *out, string s) {
  out→writeInt(2*size());
  out→writeInt(size());
  typedef typename T::iterator iterator;
  typedef typename T::key_type key_type;
  typedef typename T::data_type data_type;
  iterator pos;
  for (pos = begin(); pos ≠ end(); ++pos) {
    out→writeObject(((key_type*)&(pos→first)));
    out→writeObject(((data_type)(pos→second)));
  }
  out→flush();
}
```

Ebenso wie bei den Vektoren ist auch die `map` Klasse dafür ausgelegt, eine Wertsemantik zu unterstützen. Insbesondere macht es keinen Sinn, Zeiger als Schlüssel zu verwenden, da in diesem Fall die Adresse eines Objektes einen Eintrag indiziert und nicht sein Wert. Aus diesem Grund mußte auch hier wieder ein Kompromiß gemacht werden. Während die Schlüssel immer Objekte sind, handelt es sich bei den indizierten Werten um Zeiger auf Objekte. Diese Tatsache schlägt sich sowohl in der `readObjet()` als auch in der zu ihr symmetrischen `writeObjet()` Methode nieder, wo jeweils davon ausgegangen wird, daß es sich bei dem Schlüssel-Typ um eine Klasse und bei dem Wert-Typ um einen Zeiger handelt. Natürlich müssen sowohl Wert- als auch Schlüssel-Typ serialisierbar sein.

Wie alle serialisierbaren Klassen muß auch eine Instantiierung von `GenericHTWrapper` erst einmal bei `Serializable` angemeldet werden, bevor sie verwendet werden kann. Da die Anzahl möglicher `map` Typen jedoch nicht beschränkt ist, ist es nicht sinnvoll, diese Registrierung durch statische Objekte vor dem Programmst durchzuführen. Stattdessen ist es Aufgabe des Anwenders dies an entsprechender Stelle durch anlegen eines `Init_Object` zu tun.

Danach können `map` Objekte der angemeldeten Typen problemlos serialisiert werden. Beim Lesen ist hier jedoch noch eine Besonderheit zu beachten. Da alle deserialierten assoziativen Arrays den Typ `java.util.Hashtable` haben, kann die Klasse `SerialParse` nicht wissen, auf welchen C++ `map` Typ diese abgebildet werden sollen. Standardmäßig wird dafür der letzte registrierten `map` Typ ausgewählt. Werden jedoch, wie in den folgenden Anweisungen

```
Init_Object<GenericHTWrapper<map<string, string*> > > strStrHT;
Init_Object<GenericHTWrapper<map<string, List*> > >   strListHT;
```

mehrere `map` Typen registriert, dann muß vor dem Lesen einer `Hashtable` mittels der `activate()` Methode von `Init_Object` der gewünschten `map` Typ aktiviert werden.

```
strStrHT.activate();
                  ➥
```

Die Funktion activate() der Klasse Init_Object, die in Listing 23 auf Seite 45 nicht gezeigt wurde, meldet eine einmal registrierte Klasse noch einmal an, was bei unterschiedlichen C++ Klassen die alle auf die selbe Java Klasse abgebildet werden dazu führt, daß die aktivierte Klasse beim nächsten Lesen der entsprechenden Javaklasse als ihr C++ Pendant erkannt wird.

Listing 58 : ../C++Ser/Serializable.hpp [Zeile 321 bis 330]

```
  void activate() {
    Serializable::insertMapping(className, typeid(typename T::objType).name(),
                                t_VHS_t.getNewObj());
// ...
  }
```

Sicherheitshalber sollte activate() vor jedem Lesen verwendet werden, da davon ausgegangen werden muß, daß andere Programmmodule auch Instantiierungen von GenericHT-Wrapper verwenden. Insbesondere sollte dies auch in der setPrimValue() Methode gemacht werden, wenn es ein Klassenmitglied des Typs map gibt, da die Basistypen einer Klasse immer vor deren Objekttypen gelesen werden. Gibt es in der Klasse meherere Variablen vom Typ map, dann sollte nach jedem setzen einer solchen in setObjectValue() auf den nächsten map Typ umgeschaltet werden.

7. Zusammenfassung und Ausblick

In der Diplomarbeit wurde eine sofort einsetzbare und leicht erweiterbare Bibliothek für eine Client/Server Kommunikation auf Objektebene entwickelt. Dabei wurde auf C++ Seite das von Java bekannte Serialisierungsprotokoll implementiert. Dadurch wird es möglich, die Schnittstelle zwischen Java Clients auf der einen und C++ Servern auf der anderen Seite wesentlich zu vereinfachen.

Dabei wurden die wenigen, aber dennoch grundlegenden Unterschiede zwischen Java und C++ aufgezeigt und ausführlich besprochen. Auf C++ Seite wurden die neuen Sprachkonzepte von C++ wie Laufzeittypinformationen, Parametrisierung und Ausnahmebehandlung ausgiebig verwendet um die Bibliothek sowohl verständlicher und leichter wartbar als auch leichter einsetzbar zu machen.

Außer dem Parsergenerator ANTLR, der jedoch im Quellcode vorliegt und den es für eine Vielzahl von Plattformen gibt, wurden keine speziellen Bibliotheken verwendet. Die Implentierung wurde in C++ nach Vorgaben des neuen C++ Standards und unter ausschließlicher Verwendung der darin enthaltenen Standardbibliotheken verwirklicht, so daß sie sich auf jeder Plattform mit einem Standard C++ Compiler übersetzen und einsetzen lassen sollte.

Neben dem Einsatz als Schnittstelle zwischen Java Clients und C++ Servern läßt die Bibliothek sich aber auch noch auf vielfältige andere Weise einsetzen. Als Beispiel sollen hier nur die C++/C++ Kommunikation oder das persistente Abspeichern von C++ Objekten genannt werden.

In Zukunft sollte die Bibliothek noch um weitere Hüllenklassen erweitert werden, die die Java Hilfsklassen wie `BitSet`, `Stack` oder `Vector` auf die entsprechenden STL-Container abbilden. Desweiteren wäre es nun mit der funktionierenden Serialisierung relativ einfach möglich, auf C++ Seite auch den RMI (*remote method invocation*) Mechanismus von Java zu implementieren. Damit wäre man dann nicht mehr weit entfernt von CORBA ...

A. Die Serialisierungsgrammatik

Hier also die originale Grammatik Serialisierungsgrammatik [SerSp]. Angelehnt an die EBNF, bezeichnen kleingeschriebene Namen Produktionsregeln, während Namen mit großen Anfangsbuchstaben Terminale darstellen.

```
stream:
        magic version contents

contents:
        content
        contents content

content:
        object
        blockdata

object:
        newObject
        newClass
        newArray
        newString
        newClassDesc
        prevObject
        nullReference
        exception
        TC_RESET

newClass:
        TC_CLASS classDesc newHandle

classDesc:
        newClassDesc
        nullReference
        (ClassDesc)prevObject   // an object required to be of type ClassDesc

superClassDesc:
        classDesc

newClassDesc:
        TC_CLASSDESC className serialVersionUID newHandle classDescInfo
```

```
classDescInfo:
        classDescFlags fields classAnnotation superClassDesc

className:
        (utf)

serialVersionUID:
        (long)

classDescFlags:
        (byte)  // Defined in Terminal Symbols and Constants

fields:
        (short)<count>  fieldDesc[count]

fieldDesc:
        primitiveDesc
        objectDesc

primitiveDesc:
        prim_typecode fieldName

objectDesc:
        obj_typecode fieldName className

fieldName:
        (utf)

className:
        (String)object          // String containing the field's type

classAnnotation:
        endBlockData
        contents endBlockData   // contents written by annotateClass

prim_typecode:
        'B'     // byte
        'C'     // char
        'D'     // double
        'F'     // float
        'I'     // integer
        'J'     // long
        'S'     // short
        'Z'     // boolean

obj_typecode:
        '['     // array
        'L'     // object

newArray:
        TC_ARRAY classDesc newHandle (int)<size> values[size]
```

```
newObject:
        TC_OBJECT classDesc newHandle classdata[]       // data for each class

classdata:
        nowrclass                 // SC_WRRD_METHOD & !classDescFlags
        wrclass objectAnnotation  // SC_WRRD_METHOD & classDescFlags

nowrclass:
        values  // fields in order of class descriptor

wrclass:
        nowrclass

objectAnnotation:
        endBlockData
        contents endBlockData     // contents written by writeObject

blockdata:
        blockdatashort
        blockdatalong

blockdatashort:
        TC_BLOCKDATA (unsigned byte)<size> (byte)[size]

blockdatalong:
        TC_BLOCKDATALONG (int)<size> (byte)[size]

endBlockData    :
        TC_ENDBLOCKDATA

newString:
        TC_STRING newHandle (utf)

prevObject      :
        TC_REFERENCE (int)handle

nullReference   :
        TC_NULL

exception:
        TC_EXCEPTION reset (Throwable)object     reset

resetContext:
        TC_RESET

magic:
        STREAM_MAGIC

version :
        STREAM_VERSION
```

```
values:         // The size and types are described by the
                // classDesc for the current object

newHandle:              // The next number in sequence is assigned
                        // to the object being serialized or deserialized
```

Literaturverzeichnis

[AhoSetUl] Aho, Alfred; Sethi, Ravi and Ullman, Jeffry D. *"Compilers : Principles, Techniques, and Tools"* Addison-Wesley, Reading, MA. 1986

[ANSI-CPP] ANSI/ISO Standard *"Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++"* American National Standards Institute (ANSI), Nov. 1997
z.B. : ftp://research.att.com/dist/stdc++/WP/ (only 1996 Version)

[Bloo91] Bloomer, John *"Power Programming with RPC"* O'Reilly & Associates, Inc., 1991

[Brey96] Breymann, Ullrich *"Die C++Standard Template Library"* Addison-Wesley, 1996

[Brooksh] Brookshear, J. Glenn *"Formal Languages, Automata, and Complexity"* The Benjamin/Cummings Publishing Company, Inc.

[CleeSchm] Cleeland, Chris, und Schmidt, Douglas C. *"External Polymorphism"* C++ Report, July/August 1998, SIGS Publications

[Com96] Comer, Douglas E. und Stevens, David L. (1996). *"Internetworking with TCP/IP, Vol. III."* Prentice-Hall, 1996

[CORBA] OMG *"The Common Object Request Broker : Architecture and Specification"* OMG Document, 1994 (http://www.omg.org)

[Day83] Day, J. D. und Zimmermann, H. (1983). *"The OSI Reference Model."* Proc. of the IEEE, vol. 71, pp. 1334-1340, Dec. 1983

[Flan96] Flanagan, David *"Java in a Nutshell"* O'Reilly & Associates, Inc. (1996)

[Glass] Glass, Graham *"A Universal Streaming Service"* C++ Report, April 1996, SIGS Publications

[HopUll] Hopcroft, John E.; Ullman, Jeffry D. *"Introduction to Automata Theory, Languages and Computation"* Addison Wesley (1979)

[JDK1.2] *"JDK 1.2 Documentation"* http://java.sun.com/products/jdk/1.2/

[JavaTut] Campione, Mary und Walrath, Kathy *"The Java Tutorial"* Addison-Wesley
http://java.sun.com/docs/books/tutorial/

[Jos96] Josuttis, Nicolai *"Die C++Standardbibliothek"* Addison-Wesley-Longman, 1996

[Laeng98] Längle, Frederic *"Softwaredokumentation und -entwurf in objektorientierten Systemen"*
Diplomarbeit am W. Schickard Inst. der Uni. Tübingen, 1998

[Lee-Stepanov] Lee, Meng und Stepanov, Alexander *"The Standard Template Library"*
z.B. : http://www.cs.rpi.edu/ musser/doc.ps

[Lipp96] Lippman, Stanley B. *"Inside the C++ Object Model"*
Addison-Wesley, 1996

[Musser-Saini] Musser, David R. und Saini, Atul *"STL Tutorial and Reference Guide"*
Addison-Wesley, 1996

[Parr97] Parr, John Terrence *"Language Translation Using PCCTS & C++"*
Automata Publishing Company 1997

[RFC821] Request For Comments 821. *"SMTP"* z.B. : http://sunsite.auc.dk/RFC/

[RFC959] Request For Comments 959. *"FTP"* z.B. : http://sunsite.auc.dk/RFC/

[RFC977] Request For Comments 977. *"NNTP"* z.B. : http://sunsite.auc.dk/RFC/

[RFC1094] Request For Comments 1094. *"NFS"* z.B. : http://sunsite.auc.dk/RFC/

[RFC1122] Request For Comments 1122. *"ARP, IP, ICMP, IGMP, TCP, UDP"*
z.B. : http://sunsite.auc.dk/RFC/

[RFC1123] Request For Comments 1123. *"FTP, TELNET, SMTP, DNS"*
z.B. : http://sunsite.auc.dk/RFC/

[RFC1831] Request For Comments 1831. *"RPC: Remote Procedure Call Protocol Specification Version 2"* z.B. : http://sunsite.auc.dk/RFC/

[RFC1832] Request For Comments 1832. *"XDR : External Data Representation Standard"* z.B. : http://sunsite.auc.dk/RFC/

[RFC2068] Request For Comments 2068. *"HTTP"* z.B. : http://sunsite.auc.dk/RFC/

[RMI] Sun Microsystems (1996). *"Java Remote Method Invocation Specification"* Sun Microsystems, Revision 1.2, December 2, 1996 JDK 1.1

[RBP+91] Rumbaugh, James; Blaha, Michael; Premerlani, William; Frederick, Eddy and Lorenson, William. *"Object-Oriented Modeling and Design."* Prentice-Hall, 1991

[Schimk97] Schimkat, Ralf *"Ein objektorientiertes Framework für Java-Clients in Dokumentenverwaltungssystemen"*
Diplomarbeit am W. Schickard Inst. der Uni. Tübingen, 1997

[Schwarz] Schwarz, Jerry *"Initializing Static Variables in C++ Libraries"* C++ Report, February 1989, SIGS Publications

[SerSp] Sun Microsystems (1998). *"Java Object Serialisation Specification"* Sun Microsystems, Revision 1.4.1, February 12, 1998 JDK 1.2

[Sim97] Simonis, Volker *"Normmaster Web"* debis Systemhaus EDM/D, 1997

[Stev90] Stevens, Richard W. (1990). *"UNIX Network Programming."* Prentice-Hall, 1990

[Stev92] Stevens, Richard W. (1992). *"Advanced programming in the UNIX environment."*
Addison Wesley, 1994

[Stev94] Stevens, Richard W. und Wright, G. R. (1994). *"TCP/IP Illustrated, Vol. 2."*
Addison Wesley, 1994

[Strou94] Stroustrup, Bjarne. *"The Design and Evolution of C++"*
Addison Wesley, 1994

[Wirth86] Wirth, Niklaus *"Compilerbau - Eine Einführung"*
Teubner, 1986

Colophon

Diese Diplomarbeit wurde mit LaTeX 2_ε in Times Roman der Schriftgröße 11pt gesetzt. Für die Programmlistings wurde die Schriftfamile `LetterGothic12Pitch` in 9pt verwendet.

Zur Programmdokumentation wurde ein vom Autor entwickeltes System von Skripten und Styles verwendet, das die automatische Einbindung von mittels Kommentaren gekennzeichneten Passagen aus Quellkodedateien gestattet. Die LaTeX 2_ε Bearbeitung wird durch ein Makefile gesteuert, welches bei jedem Aufruf dafür sorgt, daß in das Dokument immer die aktuellen Programmauszüge eingefügt werden. Somit enthält jedes Listing die zum Dokument relative Pfadangabe seiner Quelldatei und die tatsächlichen Zeilennummern des daraus dargestellten Abschnittes. Es spiegelt die Sichtweise des Autors von einem praxisorientierten "*literate programming*" System wieder.