INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwurf und CORBA-basierte Implementierung eines Objektmodells für das Management von ATM-Switches

Bearbeiter: Gregor Hölscher

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Alexander Keller
Jan Hickisch, Siemens AG

INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwurf und CORBA-basierte Implementierung eines Objektmodells für das Management von ATM-Switches

Bearbeiter:	Gregor Hölscher
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Alexander Keller Jan Hickisch, Siemens AG
Abgabetermin:	15. November 1997

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. November 1997

....
(Unterschrift des Kandidaten)

Zusammenfassung

Mit den Fortschritten in der Tele- und Datenkommunikation werden die Ansprüche an die vorhandenen Netze immer höher. Zusätzlich steigen auch die Forderungen nach Netzkomponenten, die diesen Anforderungen gerecht werden können. Die ATM-Technologie spielt dabei eine wichtige Rolle, da sie in der Lage ist, hohe Bandbreiten zur Verfügung zu stellen. Auf Basis dieser Technologie hat Siemens, Bereich Private Netze, den ATM-Switch Hicom Xpress entwickelt. Dieser kann zur Zeit aber nur über ein proprietäres Managementtool administriert werden.
Um für Weiter- oder Neuentwicklungen gerüstet zu sein, soll in dieser Arbeit ein Objektmodell für das Management des ATM-Switches entworfen und auch implementiert werden. Dabei wird für die Modellierung des Modells die *Object Modeling Technique* (OMT) verwendet. Denn sie stellt eine gute Möglichkeit dar, um die Analyse- und Designphase bei diesem Projekt zu unterstützen.
Ist mit Hilfe dieser Technik ein Objektmodell für die Administration des Hicom Xpress entworfen, so kann dies im zweiten Schritt unter der Verwendung von *Use Cases*, die ebenfalls in der OMT enthalten sind, zu einem generischen Objektmodell weiterentwickelt werden. Dies ist möglich, da mit den Use Cases die Anforderungen an Managementfunktionalität und -information bezüglich der verwendeten Technologie dargestellt werden können.
Nach Fertigstellung des generischen Objektmodells wird auf dieser Basis unter Verwendung der CORBA-Technologie ein CORBA-Agent für das Management von ATM-Switches implementiert.
Damit die Administration nicht über die Kommandozeile durchgeführt werden muß, wird zusätzlich noch eine neue graphische Oberfläche entworfen. Auch diese wird mittels der CORBA-Technologie an den implementierten Managementagenten angebunden, da man mit der CORBA-Technologie u.a. unabhängig von Betriebssystemplattformen ist.
Am Ende der Arbeit steht ein Prototyp zur Verfügung, der das Management des ATM-Switches Hicom Xpress über eine graphische Oberfläche ermöglicht. Durch die Entwicklung des generischen Objektmodells ist man aber auch in der Lage, ohne sehr großen Aufwand einen anderen ATM-Switch zu verwalten. Dazu müssen im Objektmodell die Klassen, Attribute und Methoden bezüglich des gewünschten Switches überprüft und gegebenenfalls geändert oder ergänzt werden. Diese Veränderungen müssen dann natürlich noch im Agenten und in der Oberfläche implementiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Im ersten Teil des Kapitels wird das Thema der Arbeit in das Themenspektrum eingeordnet und motiviert. Die gestellte Aufgabe und die für die Lösung notwendige Vorgehensweise wird im Kapitel 1.2 vorgestellt. Im letzten Teil des ersten Kapitels werden dann die Ergebnisse der vorliegenden Arbeit formuliert.

1.1 Motivation

Die Entwicklung in der Tele- bzw. Datenkommunikation hat in den letzten Jahren einige neue Wege eingeschlagen. So wurden neben den Diensten wie Telefon, Kabelfernsehen und schmalbandige Datenübertragungen neue Dienste entwickelt. Diese Dienste, wie z.B. Videokonferenzen, Video on Demand oder breitbandige Datenübertragungen, führen jedoch zu anderen Anforderungen an ein Kommunikationsnetz. Eine weitere Entwicklung, die höhere Bandbreite erfordert, ist der Trend weg von Zentralrechnern hin zu Client-Server-Strukturen. Der Zusammenhang zwischen Bandbreite und Geschwindigkeit und die Entwicklung in diesem Bereich ist in der Abbildung 1-1 dargestellt.

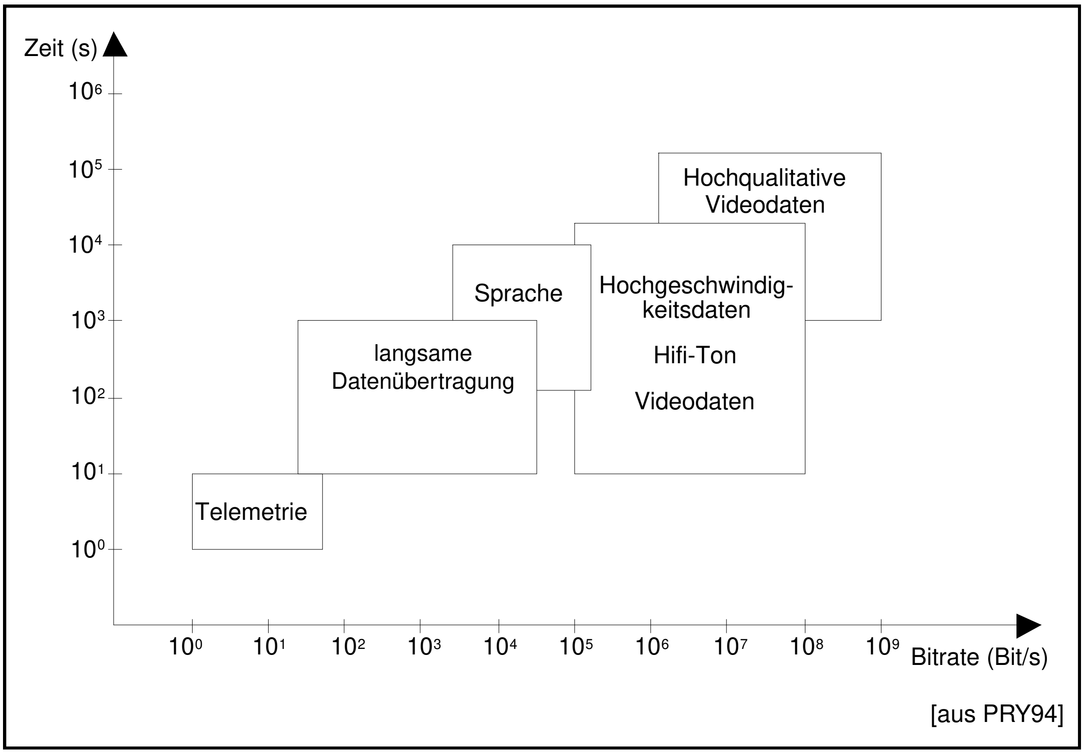


Abbildung 1-1: Dienstanforderungen in Netzen

Um diese neuen Anforderungen an ein Netz erfüllen zu können, müssen geeignete Übertragungsverfahren für die unterschiedlichen Anwendungen gefunden werden.
Vergleicht man unterschiedliche Verfahren miteinander, so stellt man sehr schnell fest, daß ein großes Spektrum vorhanden ist. Auf der einen Seite steht die Leitungsvermittlung, die während der Übertragung eine dedizierte Leitung verwendet und damit Ressourcen belegt. Jedoch ist durch die dedizierte Leitung ein schnellerer und sicherer Datentransport möglich, wodurch isochrone Anwendungen besser unterstützt werden können. Auf der anderen Seite steht die Paketvermittlung, die die jeweiligen Nachrichten in Pakete zerlegt und diese entweder verbindungslos oder verbindungsorientiert überträgt. Um auch bei diesem Verfahren isochrone Dienste zur Verfügung stellen zu können, ist größerer Rechenaufwand nötig. Deshalb ist die Komplexität gegenüber der Leitungsvermittlung auch höher.
Ungefähr in der Mitte dieser Skala befindet sich das Vermittlungsverfahren, das uns im weiteren begleiten wird, der *Asynchronous Transfer Modus*, kurz ATM genannt. ATM ist dabei ein paketorientierter Übertragungsmodus, der Zellen fester Länge verwendet und das statistische Zeitmultiplexverfahren für die Übertragung einsetzt. Für eine genauere Einführung in ATM wird auf Kapitel 2.1 verwiesen.
Auf dieser Technik basierend hat die Firma Siemens, Bereich Private Netze, einen ATM-Switch entwickelt, der HICOM Xpress genannt wird. Neben der normalen Datenübertragung werden auch Features wie Videokonferenzen oder Telefonie unterstützt. Damit wird auch klar, wieso die Firma Siemens auf die ATM-Technologie gesetzt hat, denn ATM ist zur Zeit eine der wenigen Technologien, die die nötige Bandbreite für isochrone Anwendungen wie Videokonferenzen zur Verfügung stellen kann.

1.2 Aufgabenstellung und Vorgehen

Mit der Entwicklung eines solchen ATM-Switches, der in Kapitel 3.1 vorgestellt wird, entstehen aber auch Forderungen nach einem geeigneten Management der Komponenten. Zum jetzigen Zeitpunkt wird der Switch über eine proprietäre Schnittstelle mit einem dedizierten Managementtool verwaltet. Da dieses Tool aber sehr stark auf den spezifischen Switch abgestimmt ist, ist die Verwendung für Weiter- oder Neuentwicklungen nicht so leicht möglich.

Aufgabe der vorliegenden Arbeit ist es, einen Prototypen für das Management dieses ATM-Switches zu entwickeln. Dabei soll die Kommunikation mit dem Switch über einen CORBA-konformen *Object Request Broker* (ORB) erfolgen. Um diesen Prototypen zukünftig auch ohne großen Änderungsaufwand bei anderen Netzkomponenten einsetzen zu können, soll ein generisches Objektmodell als Grundlage verwendet werden. Ebenso wird die proprietäre Benutzeroberfläche überarbeitet, da sonst Probleme beim Einsatz mit anderen Komponenten auftreten, d.h. ein Einsatz der neuen Managementanwendung wäre wieder nur für spezielle Komponenten möglich. Ziel soll aber eine Anwendung sein, die auf mehreren Plattformen und unabhängig vom Betriebssystem arbeitet. Die Anbindung dieser überarbeiteten Benutzeroberfläche an den ORB wird mittels Java realisiert.

Der Ausgangspunkt bzw. die Basis für diese Arbeit ist die *Hicom Xpress Administration* (XAD), die in Kapitel 3.2 vorgestellt wird. Aus diesem proprietären Managementtool wird nun im ersten Schritt ein Objektmodell entwickelt, das die Managementfunktionalität und -information des Switches mit Klassen, Attributen und Methoden beschreibt. Hat man dieses proprietäre Objektmodell fertiggestellt, wird dieses in ein generisches Objektmodell überführt. Dazu müssen die allgemeinen Anforderungen an einen ATM-Switch untersucht werden. Um diese Aufgabe zu erleichtern, wird auf der einen Seite die ATM-Technologie und auf der anderen Seite die Switch-Technologie betrachtet. Die beste Darstellungsform für die Szenarien im Zusammenhang mit einem ATM-Switch stellen die *Use Cases* (siehe Kapitel 4.6.1) zur Verfügung. Denn mit Hilfe dieser Szenarien werden die einzelnen Features für einen ATM-Switch definiert. Verbindet man diese *Use Cases* mit dem proprietären Objektmodell, so wird man Gemeinsamkeiten, aber auch Unterschiede erkennen können. Durch diese Verbindung entsteht ein Objektmodell mit einem generischen und einem proprietären Anteil. Durch den generischen Teil des Modells ist man nun in der Lage, zukünftige Entwicklungen leichter in das Objektmodell aufnehmen zu können. Der Grund liegt darin, daß man nur den proprietären bzw. spezifischen Teil im Modell ändern muß und der generische Anteil unverändert übernommen werden kann.
Nach Fertigstellung dieses Modells wird für jede Objektklasse der zugehörige IDL-Code generiert.

Der IDL-Code ist eine Schnittstellensprache, die *Interface Definition Language*, die eine Anbindung von Clients und Servern an den ORB ermöglicht.
Die einzelnen umgesetzten Klassen aus dem Objektmodell, die jetzt in IDL-Code vorliegen, werden mit einem IDL-Compiler compiliert. Dadurch entstehen neben Header-Dateien und Client- bzw. Server-Stubs auch Dateien, die die Funktionsrümpfe der jeweiligen Attribute und Methoden enthalten. Diese müssen noch mit dem jeweiligen Code für die Systemaufrufe gefüllt werden. Hierbei kann z.B. C++, Java oder Ada95 verwendet werden. Werden diese ergänzten Funktionsrümpfe übersetzt, spricht man bei CORBA vom sogenannten *Language Mapping*, da CORBA wegen der einheitlichen IDL in der Lage ist, unterschiedliche Sprachen für die Implementierung zu verwenden. Somit ist man in der Lage, programmiersprachen- und plattformunabhängig auf unterschiedliche Objekte zuzugreifen und dabei den Kommunikationsmechanismus des ORB zu benutzen.

Damit liegt ein Prototyp vor, um einen ATM-Switch über einen ORB zu managen. Das dabei auftretende Problem ist die fehlende Benutzeroberfläche (GUI). Zur Realisierung dieser wird die bisherige Oberfläche überarbeitet bzw. an den vorliegenden Prototypen angepaßt. Dabei ist wichtig, daß das GUI über den ORB mit dem Agenten für das Management kommuniziert. Um dies zu gewährleisten, wird die Anbindung an den ORB mittels Java realisiert. Die Erstellung der Oberfläche wird dabei mit einem GUI-Builder durchgeführt. Das Vorgehensmodell dieser Arbeit von der Analyse, über das Design bis zur Implementierung ist in der Abbildung 1-2 graphisch dargestellt.

Um die Schritte auch zufriedenstellend ausführen zu können, ist es wichtig, geeignete Softwaretools zu finden.
Für die ersten Schritte, d.h. den Entwurf des ersten Objektmodells, die Formulierung der *Use Cases* und den Entwurf eines optimierten Objektmodells, bietet sich das CASE-Tool *Software through Picture* (StP, siehe Kapitel 4.1) an. Auch die Generierung des IDL-Codes für die jeweiligen Klassen ist damit möglich.
Für die anschließende Compilierung des IDL-Codes wird der IDL-Compiler der Firma IONA verwendet. Dieser wird zusammen mit einem CORBA-konformen ORB, genannt *Orbix*, geliefert. Dieser ORB (siehe Kapitel 5.2) wird in diesem Projekt auch als Kommunikationsmedium für die Implementierung des Managementagenten für den ATM-Switch verwendet. Für die Anbindung des GUI wird ein weiterer ORB der Firma IONA verwendet, *OrbixWeb* (siehe Kapitel 6.3), da er die Java-Anbindung der Oberfläche an den ORB unterstützt. Zum Entwurf und Aufbau der Oberfläche wird der *JBuilder* (siehe Kapitel 6.3) der Firma Borland verwendet, der auch die Anbindung an den ORB mittels Java unterstützt.

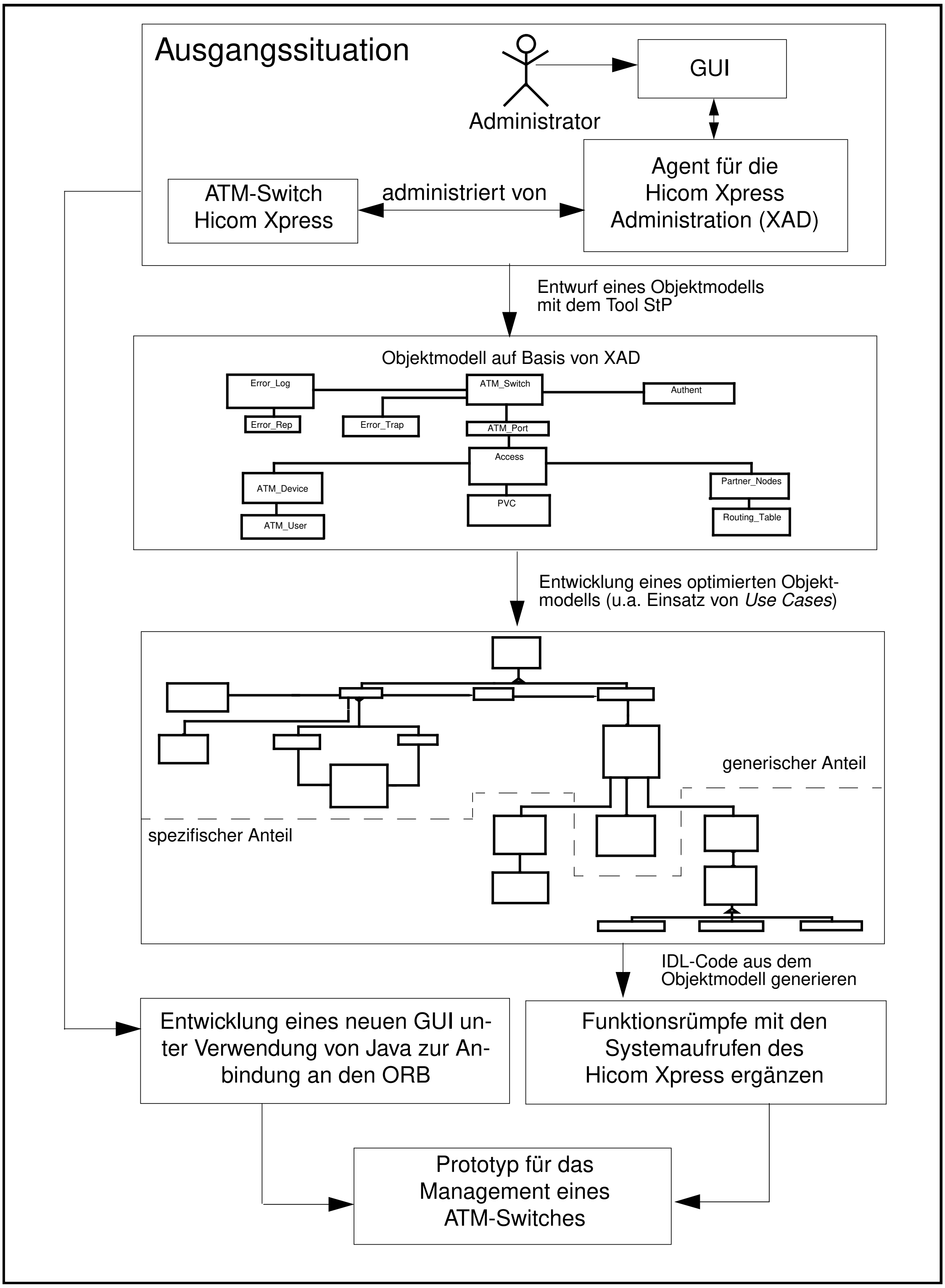


Abbildung 1-2: Aufbau der Arbeit

1.3 Ergebnisse der Arbeit

Die Aufgabenstellung der vorliegenden Arbeit berührt mehrere unterschiedliche Themenschwerpunkte. Diese Aufteilung spiegelt sich natürlich auch in den gewünschten bzw. erwarteten Ergebnissen der Arbeit wieder.

Wie schon im Titel der Arbeit festgehalten, soll ein Objektmodell zum Management von ATM-Switches entworfen werden. Die Basis für diese Entwicklung bildet das Datenmodell des Hicom Xpress, das während der Arbeit in mehreren Schritten zu einem Objektmodell ausgebaut wird. Dieses Objektmodell entspricht dabei der *Object Modeling Technique* von Rumbaugh.
Um das entstandene Modell auch für andere ATM-Switches einsetzen zu können, wird das Objektmodell mit Hilfe der *Use Cases*, die mit dem Case Tool *Software through Picture* definiert wurden, zu einem generischen Objektmodell ausgeweitet. Mit dessen Fertigstellung ist die Entwurfs- und die Designphase für das Objektmodell abgeschlossen und die Einsatzfähigkeit des Case Tools *Software through Picture*, das die OMT-Technologie zur Verfügung stellt, erfolgreich getestet worden.

Im nächsten Schritt entstehen die IDL-Dateien, die für die Implementierung des Managementagenten notwendig sind. Damit werden die im Objektmodell enthaltene Informationen zusammengefaßt und stehen somit für die weiteren Implementierungsschritte zur Verfügung. In diesem Zusammenhang wird die Schnittstelle zwischen dem Case Tool *Software through Picture* und dem *Orbix*, d.h. der ORB-Umgebung auf Verwendbarkeit überprüft, da sonst weitere Schritte nicht sinnvoll wären.

Nachdem diese Generierung erfolgreich abgeschlossen ist, beginnt im Anschluß daran die Implementierung der Agentenlösung. Dazu werden die von IONA bereitgestellten CORBA-Tools, wie der IDL-Compiler oder der ORB eingesetzt.
Am Ende der erfolgreichen Implementierung steht ein CORBA-Agent zur Verfügung, der das Management des Hicom Xpress ermöglicht (siehe Abbildung 1-3).
Mit dieser implementierten Agentenlösung kann der Administrator aber nur über ein Kommandozeileninterface den gewünschten ATM-Switch administrieren.
Um aber die Benutzerfreundlichkeit des Managementagenten für ATM-Switches zu steigern, wird im nächsten Schritt noch eine graphische Benutzeroberfläche entworfen und implementiert.
Dadurch entsteht dann das in Abbildung 1-3 dargestellte Zielszenario, in dem ein oder mehrere ATM-Switches über eine graphische Benutzeroberfläche administriert werden.
Auch bei der Implementierung der graphischen Oberfläche werden die aus dem Objektmodell generierten IDL-Dateien als Basis verwendet, da die Oberfläche über einen ORB mit dem CORBA-Agenten kommunizieren muß (siehe Abbildung 1-3).

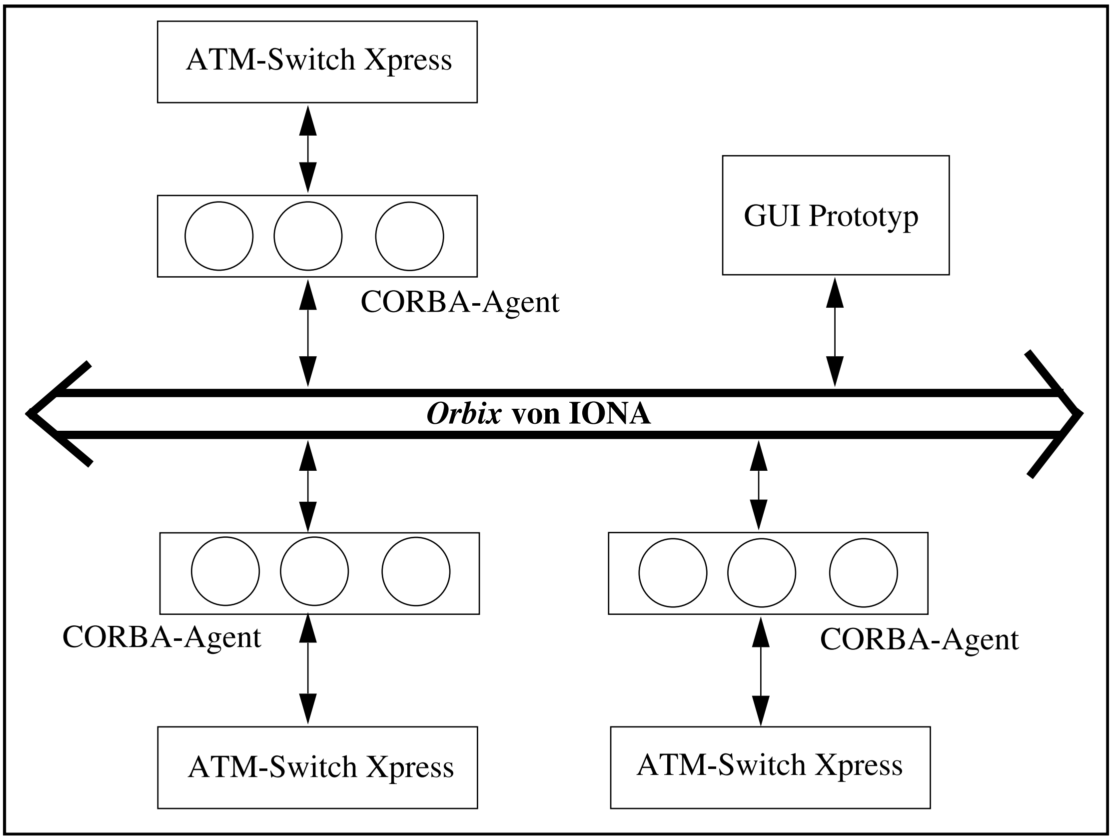


Abbildung 1-3: Zielszenario

Dazu wird für die Implementierung der Anbindung der *OrbixWeb*, ein weiteres Produkt von IONA eingesetzt und getestet. Dabei stellt man fest, daß die Benutzerfreundlichkeit auf dem Betriebssystem *Windows NT 4.0* besser ist, als auf *Unix*, da unter *Windows NT 4.0* für die meisten Arbeiten graphische Oberflächen zur Verfügung stehen. Durch den Einsatz von *OrbixWeb* auf *Windows NT 4.0* konnte gleichzeitig mit Erfolg getestet werden, ob die Kommunikation zwischen ORBs auf unterschiedlichen Plattformen möglich ist.
Der *OrbixWeb* wurde aber auch deshalb eingesetzt, da er einen IDL-Compiler zur Verfügung stellt, der Java-Code generiert. Dies ist notwendig, da für die Gestaltung der graphischen Oberfläche der *JBuilder* von Borland verwendet wird und dieser mit der Programmiersprache Java arbeitet.

Nach erfolgreicher Fertigstellung der Oberfläche mit dem *JBuilder* und der Anbindung an den ORB, steht ein CORBA-Agent mit graphischer Oberfläche für das Management von ATM-Switches zur Verfügung und die Zielgerade für die vorliegende Arbeit ist damit erfolgreich überquert worden.

2 State of the Art

In diesem Kapitel werden die eingesetzten bzw. verwendeten Technologien der vorliegenden Arbeit vorgestellt. Neben einem kurzem geschichtlichen Überblick wird die jeweilige Technologie, wie z.B. ATM oder CORBA, basierend auf den jeweiligen Grundlagen eingeführt. Dabei wird vor allem am Ende des zweiten und dritten Abschnitts des Kapitels 2.1 eine Zusammenstellung der vorkommenden Managementfunktionalität und -information der jeweiligen Technologie gegeben. Diese und alle anderen Informationen des zweiten Kapitels bilden die Basis für die nächsten Schritte.

2.1 Überblick über ATM und Switching

In den letzten Jahren haben sich ATM und vor allem auch die Switchingtechniken immer weiter entwickelt. Um diese beiden Technologien näher kennenzulernen, wird ATM im Kapitel 2.1.1 geschichtlich beleuchtet, bevor im Kapitel 2.1.2 die Grundlagen von ATM vorgestellt werden. Im Kapitel 2.1.3 wird dann zum Abschluß die zur Zeit vorhandenen und verwendeten Swichtingtechniken erläutert.

2.1.1 Allgemeines zu ATM

Die Grundlagen für den *Asynchronous Transfer Modus* (ATM) wurden bereits im Jahre 1983 von den zwei größten Forschungszentren in den USA, dem CNET und dem AT&T Bell Lab, veröffentlicht. Zu diesem Zeitpunkt waren die Überlegungen jedoch nur soweit vorangeschritten, um ATM im WAN einzusetzen.
Im Jahre 1986 wurde dann von der CCITT (Comité Consultatif International Télégraphique et Téléphonique), die heutige ITU (International Telecommunication Union), vorgeschlagen, die ATM-Technologie für das Breitband-ISDN (B-ISDN) einzusetzen. Jedoch vergingen drei weitere Jahre, bevor zwischen Amerika und Europa ein Kompromiß bezüglich der Zellenlänge gefunden wurde. Durch die Entscheidung, daß ATM-Zellen eine einheitliche Länge von 53 Byte haben sollten, wurde der Weg zur weltweiten Verbreitung von ATM geebnet.
Im November 1990 verabschiedete die CCITT Study Group XVIII die ersten 13 Normen. Diese bildeten die Grundlage für die weitere Entwicklung der ATM-Technologie. Diese ITU-T Weiterentwicklungen wurden 1992 in ersten Normen festgelegt.
Dies war möglich geworden, da regionale Institutionen, wie ETSI aus Europa, ANSI-T1 aus Nordamerika und TTC aus Japan, die CCITT-Vorschläge akzeptiert hatten.
Ein großer Sprung in der Entwicklung der ATM-Technologie konnte am Ende des Jahres 1991 verzeichnet werden.

Ursache war die Gründung des ATM-Forums im Oktober 1991. Das ATM-Forum ist ein Zusammenschluß von Herstellern und Anwendern, die die Entwicklung von ATM unterstützen wollen. Heute umfaßt dieses Gremium mehr als 700 Mitgliederorganisationen. Die Mitglieder sehen ihre Aufgabe darin, die technischen Interessen der LAN-Industrie zu erfassen und mit Ausarbeitungen über diese Themen die Normierung zu verbessern und zu beschleunigen.

Daraus folgte im Juli 1992 die erste UNI (User Network Interface)-Spezifikation. Heute liegt bereits die UNI-Version 4.0 vor. Im November 1992 wurde in Paris das EAC (European Activities Committee) gegründet, um zu zeigen, daß auch Europa an der Entwicklung interessiert ist und es keine reine amerikanische Angelegenheit ist. Daneben beschäftigt sich das EMAC (European Market Awareness & Education Committee) mit den Anliegen der europäischen Hersteller und Anwender.

In den darauffolgenden Jahren wurden die Spezifikationen immer weiter entwickelt bzw. neue Spezifikationen herausgegeben. So wurde z.B. ein *Private Network/Node to Network Interface* entwickelt, genannt PNNI 1.0, um mit diesem Protokoll unterschiedliche ATM-Switches zu koppeln. Ebenso wurde eine neue Version des *User Network Interface* (UNI 4.0) veröffentlicht, das verbesserten QoS, virtuelle UNIs und Video on Demand bietet. Um die Kommunikation herkömmlicher Netze über ATM zu ermöglichen, wurde im Januar 1995 die „LAN-Emulation over ATM Version 1.0“ veröffentlicht, die auch schon in einigen ATM-Switches realisiert ist. Für weitergehende Informationen bzgl. der Historie bzw. der einzelnen Spezifikationen wird auf das Literaturverzeichnis verwiesen.

Glaubt man den Marktforschern und den Systemspezialisten, so ist ATM die Technik der Zukunft. Dabei steht nicht nur die breitbandige Datenübertragung im Vordergrund, sondern auch die Übertragung von isochronen Kommunikationsformen wie Sprache und Video. Dies wird durch spezielle ATM-Dienstklassen (Classes of Service) und die prioritätenorientierte Verarbeitung ermöglicht. Durch ATM wird auch die Grenze zwischen lokalen Netzen und Weitverkehrsnetzen verwischt, obwohl immer noch Probleme, z.B. bei der Integration eines ATM-Backbones in eine traditionelle LAN-Welt zu lösen sind.

Mit der ATM-Technik ist man aber auch in der Lage, ein Multiservice-Netz auf der Grundlage von ATM aufzubauen. Dadurch können über ein einziges Netz viele unterschiedliche Verkehrstypen, wie X.25, Frame Relay, bestehende LANs oder PBX (Private Branch Exchange) Networking realisiert werden. Daneben wird durch das Multiservice-Netz ein Bandbreiten-Management ermöglicht und für jede einzelne Verbindung ein QoS garantiert.

2.1.2 Grundlagen von ATM

ATM, der *Asynchronous Transfer Modus*, stellt eine zeitkritische Multiplextechnik zur Übertragung von Daten, Video und Sprache dar, wobei die Bandbreite der Übertragung skalierbar ist.
Wesentliches Merkmal dieses verbindungsorientierten Übertragungsverfahrens sind Pakete mit konstanter Länge von 53 Byte. Sie bestehen aus einem 5 Byte Header, der vor allem die Informationen über den Weg im Netz enthält, und 48 Byte Nutzdaten, die am Netzzugang in diese Größe zerlegt wurden.
Wird mit diesen entstandenen Zellen die Bandbreite nicht vollständig ausgenutzt, so werden zusätzlich noch leere Zellen (idle cells) versendet. Dadurch ist der Anwender in der Lage, seine Datenrate solange zu erhöhen, bis keine weiteren Zellen mehr frei sind. Wird damit anderen Anwendern die Bandbreite entzogen, kann es bei ihm selber zu Zellverlusten durch Staus oder Bandbreitenengpässe kommen.
Dabei ist interessant, daß es für die Nutzdaten während der Übertragung keine Sicherung gibt, d.h. ATM verhält sich wie ein leitungsvermittelndes Netz und führt die Sicherung mit End-to-End-Protokollen durch. Daran erkennt man, daß der Anwender den Datenverlust selber erkennen und ein wiederholtes Senden anstoßen muß. Im Netzwerk dagegen wird nur die Fehlerentdeckung und -korrektur im Zellkopf durch den CRC (Cyclic Redundancy Check) unterstützt. Somit wird bei ATM im Gegensatz zu paketvermittelten Netzwerken keine Fehlererkennung oder Flußsteuerung auf Verbindungsbasis realisiert. Auch der Header hat im Vergleich zu paketvermittelten Netzen geringere Funktionalität, da Funktionen wie Quell- und Zieladresse oder Sendefolgenummer nicht benötigt werden. Die Adressierung geschieht bei ATM mittels virtueller Verbindungen, die weiter unten näher erläutert werden.
ATM unterstützt mehrere unterschiedliche Betriebsarten, wie verbindungsorientiert, verbindungslos, konstante Bitrate (CBR), variable Bitrate (VBR) und Burst-Betrieb (LANE).
Für die Übertragung und Vermittlung der Signale sind eine Reihe von Funktionen notwendig, um die Infrastruktur für den Transport der Daten zur Verfügung zu stellen. Hierfür wird auf den ATM-Referenzwürfel (Abbildung 2-1) zurückgegriffen, da man damit eine logische Gliederung aufbauen kann.

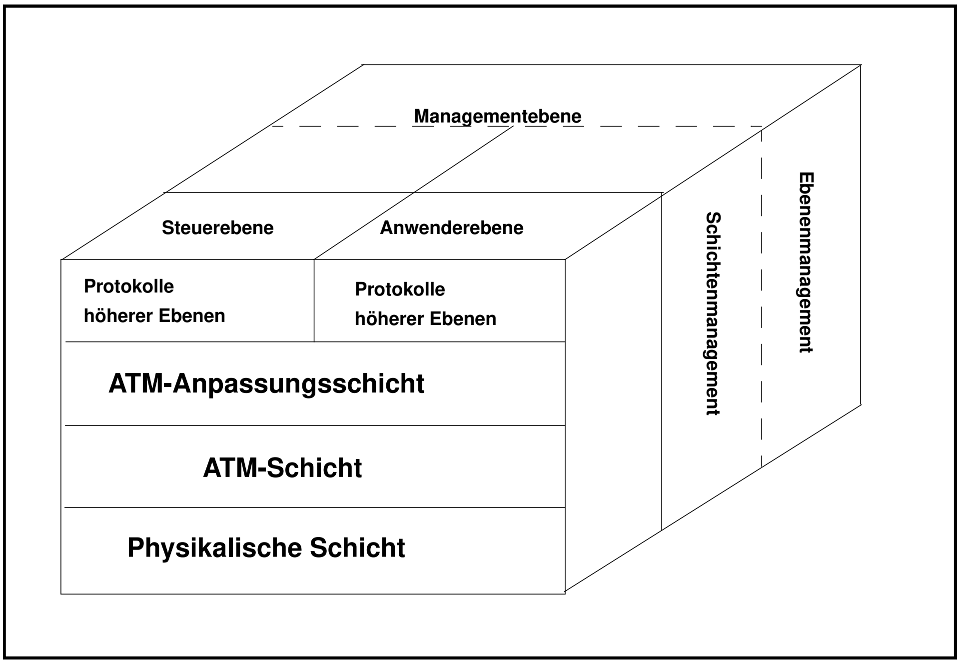


Abbildung 2-1: ATM-Referenzwürfel

- **Physikalische Schicht**:
  Die physikalische Schicht ist die unterste Schicht im Referenzwürfel, die die Funktionen für die Übertragung zur Verfügung stellt. Die Schicht selber ist nochmals in den *Physical Medium Sublayer* (PM), der die Bits überträgt und den *Transmission Convergence Sublayer* (TC), der die Zellen überträgt, unterteilt. Neben der Bitübertragung ist die PM-Schicht für die Umwandlung der Signale zwischen den unterschiedlichen Medien zuständig. Die TC-Schicht nutzt die Dienste der PM-Schicht, um die Zellen zu übertragen, die Synchronität zwischen den jeweiligen ATM-Knoten zu schützen und die empfangenen Zellen auf Korrektheit zu überprüfen. Wichtig ist noch, daß alle Schichten ab der TC-Schicht unabhängig vom Übertragungsmedium sind.

- **ATM-Schicht**:
  Die ATM-Transportschicht, die auf die physikalische Schicht aufsetzt, stellt vier Funktionen zur Verfügung:
  1) Multiplexen von Zellen eines *Virtual Paths* (VP) oder *Virtual Channel* (VC) in den Zellstrom (Erklärung siehe unten)
  2) die Kennungen der empfangenen VP- und VC-Zellen in entsprechende VP- oder VC-Kennungen auf der Sendeseite abbilden
  3) Demultiplexen der Zellen in ihre VPs oder VCs
  4) ATM Header erzeugen und entfernen

Mit dem VC wird eine unidirektionale Transportverbindung für ATM-Zellen zur Verfügung gestellt. Dabei enthält jeder Header einer ATM-Zelle einen VCI (Virtual Channel Identifier), der einen VC eindeutig einer ATM-Zelle zuordnet. Neben dem VC gibt es noch den VP, der den unidirektionalen Transport von ATM-Zellen mit verschiedenen VCs beschreibt. Diese werden unter einem VPI (Virtual Path Identifier) zusammengefaßt. Auch dieser Identifier ist im Header jeder ATM-Zelle enthalten. Der Zusammenhang zwischen VC und VP ist in der Abbildung 2-2 graphisch erläutert.

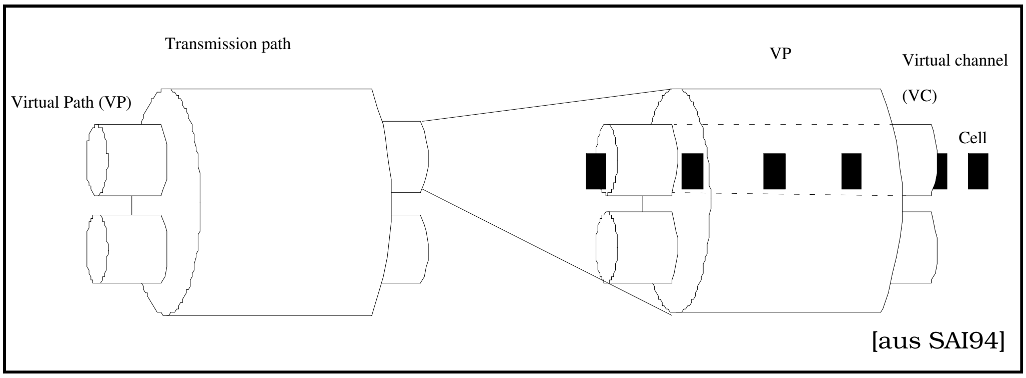


Abbildung 2-2: Hierarchie im Netzwerk

- **ATM-Anpassungsschicht**:
  Die ATM-Anpassungsschicht (ATM Adaption Layer, AAL), die auf der ATM-Schicht aufsetzt, unterstützt die Anpassung an die dienstspezifischen Anforderungen und liefert die dazu nötigen Funktionen.
  Die AAL ist wie die Physikalische Schicht in zwei Teilschichten gegliedert:
  1) *Segmentation and Reassembly Sublayer* (SAR):
     Zerlegt die PDUs der höheren Schichten in das ATM-Format und setzt umgekehrt aus den Informationen der unteren Schichten PDUs zusammen.
  2) *Convergence Sublayer* (CS):
     Bildet die Anpassung der verschiedenen Services an den SAR.

  Wichtig ist beim AAL, daß die Protokollinformationen nicht im ATM-Header der ATM-Zelle transportiert werden, sondern im Payload-Field. Mit der AAL ist auch die Fehlererkennung und -korrektur, sowie eine Zeitüberwachung möglich. Insgesamt unterscheidet man vier verschiedene AAL-Typen, die sich nach Übertragungsverfahren oder den Anwendungen unterscheiden:

  AAL 1: leitungsvermittelte Dienste mit konstanter Bitrate (CBR) und fester Zeitrelation, z.B. Primärmultiplexanschluß bei ISDN

  AAL 2: Anwendungen mit variabler Bitrate (VBR), mit exakter Zeitsynchronität, z.B. Sprach-, Ton- und Bildübertragungen (ist aber noch nicht vollständig standardisiert)

AAL 3/4: Dienste mit variabler Bitrate, aber ohne Isochronität. Man kann verbindungslose und verbindungsorientierte Dienste realisieren, z.B. Datenkommunikation

AAL 5: vereinfachte Implementierung von AAL 3/4 mit variabler Bitrate, die meist für alle Datenanwendungen verwendet wird

Im Vergleich dieser AALs stellt man fest, daß AAL 5 am meisten verwendet wird, da er recht einfach ist, einen geringen Overhead und bessere Fehlererkennung hat. Bevor wir uns noch mit den anderen Schichten des ATM-Referenzwürfels (Abbildung 2-1) befassen, soll eine genauere Betrachtung des Aufbaus einer ATM-Zelle (Abbildung 2-3) durchgeführt werden.

Eine ATM-Zelle besteht grundsätzlich aus einem 5 Byte Header und einem 48 Byte langen Informationsfeld. Der Aufbau des Headers ist bei allen ATM-Zellen bis auf die ersten vier Bit gleich. Befinden wir uns an der Schnittstelle zwischen einem ATM-User und dem ATM-Netzwerk, auch UNI genannt, so stehen im ersten Byte an den Bitpositionen vier bis sieben die **GFC-Bits** (Generic Flow Control). Damit wird der Netzzugang gesteuert, der von der aktuellen Netzauslastung abhängig ist.
Somit ist der GFC kein Mechanismus der Ende-zu-Ende-Steuerung, sondern er dient zur Staukontrolle.

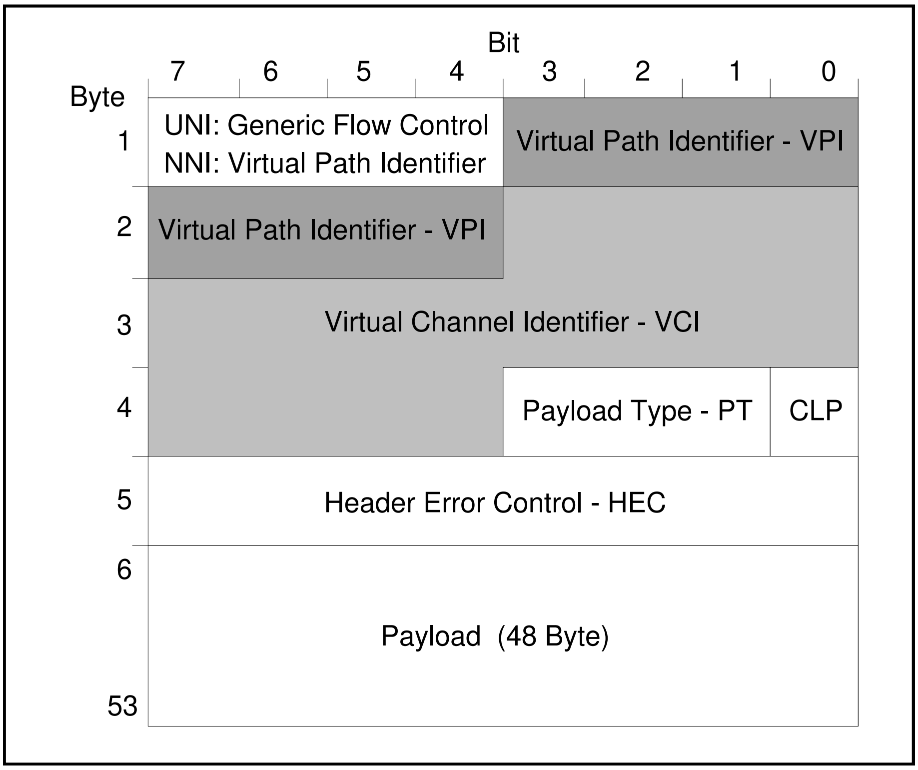


Abbildung 2-3: Aufbau einer ATM-Zelle

Befinden wir uns am NNI (Network Network Interface), so gehören die ersten vier Bits zum VPI. Das **VPI-** und **VCI-Feld** wird auch als Routing-Feld bezeichnet, da mit dem VPI und VCI die nötigen Informationen geliefert werden, um die Zellen durch die Knoten des ATM-Netzes an ihr Ziel zu bringen.
Das **"Payload type"-Feld** gibt an, welche Art von Information in der Zelle enthalten ist. Es können entweder Benutzerinformationen oder Informationen für Operationen und zur Steuerung enthalten sein.
Das **CLP-Bit** (Cell Loss Priority) gibt an, ob eine Zelle bei Überlastung des Switches oder der Leitung verworfen werden darf oder nicht. Ist der Wert des CLP Eins, so kann bei auftretenden Problemen die Zelle gelöscht werden, bei Wert Null dagegen nicht. Der Wert des CLP-Bit ist jeweils vom vereinbarten QoS beim Verbindungsaufbau abhängig.
Das **HEC-Feld** (Header Error Control) wird von der physikalischen Schicht zur Korrektur des Headers verwendet.
Im **Informationsfeld** (Payload) sind neben den reinen Nutzdaten zusätzlich noch die Protokollinformationen der ATM-Anpassungsschicht enthalten.
Bei der Betrachtung des ATM-Referenzwürfels fehlen jetzt noch drei Ebenen, die auch *Planes* genannt werden. Diese sind die *Management Plane*, die *Control Plane* und die *User Plane*.
Die Anwenderebene (User Plane) stellt die Übertragung der Nutzinformationen zur Verfügung und setzt dabei auf die ATM-Anpassungsschicht auf. Ebenso werden anwendungsbezogene Funktionen wie Flußkontrolle und Überwachung realisiert.
Die Steuerebene (Control Plane) liefert Funktionen zum Austausch von Informationen, die die ATM-Verbindung zwischen den Kommunikationspartnern steuern.
Die Managementebene (Management Plane) stellt dagegen die Schnittstelle zu Netzmanagementsystemen bereit. Sie ist nochmals in das *Plane* und das *Layer Management* unterteilt. Das *Plane Management* umfaßt dabei alle Managementfunktionen, die das gesamte System betreffen. Das *Layer Management* dagegen enthält die *Operation, Administration* und *Maintenance* (OAM-) Funktionen für die jeweiligen Schichten.
Versucht man jetzt die XAD in den ATM-Referenzwürfel einzuordnen, so stellt man fest, daß es unterschiedliche Sichtweisen für die Einordnung gibt. Betrachtet man die Oberfläche der XAD und seiner Funktionalität, so befindet man sich in der Steuerungs- bzw. Anwenderebene. Nähert man sich dagegen mehr von der Komponentenseite, d.h. vom Switch selber, so kann man durchaus sagen, daß man sich in der Managementebene befindet.
In der nun folgenden Tabelle werden zum Abschluß die ATM-Gesichtspunkte bezüglich Managementfunktionalität und -information nochmals herausgegriffen.

Managementfunktionalität	Managementinformation
Identifikation von virtuellen Kanälen und Pfaden zur Kennzeichnung der virtuellen Verbindungen	VCI und VPI legen die Kanäle und Pfade fest
Zugriffssteuerung bei Anschluß mehrerer Endgeräte an einer Teilnehmerschnittstelle	im GFC-Feld (4 Bits) wird festgelegt, wer auf das Netz zugreifen darf; bei GFC = 0000 ist die Funktion deaktiviert
Kennzeichnung des Nutzlasttyps	PLT-Feld (3 Bits) gibt Auskunft über die Art des Zelleninhalts
Fehlersicherung der Steuerinformation	HEC-Feld dient zur Korrektur des Headers, nicht des Informationsfeldes
Paket bei Überlast verwerfen	CLP-Bit = 1: Paket darf bei Überlast verworfen werden CLP-Bit = 0: hohe Priorität, Ressourcen sind für diese Zelle zu reservieren
Abbilden von Nutzinformationen auf ATM-Zellen und umgekehrt	abhängig vom verwendeten AAL-Protokoll

2.1.3 Switchingtechniken

Ein wichtiger Aufgabenbereich in der Telekommunikation war die Möglichkeit, unterschiedliche Benutzer zu verbinden und zwischen diesen umzuschalten. Dieses Umschalten zwischen zwei Teilnehmern eines Telefongesprächs wurde früher durch einen Operator von Hand durchgeführt. Mit der Zeit wurde dieses System immer weiter verbessert, bis es sogar automatisch durchgeführt werden konnte. Dazu wurden mit der Zeit immer bessere elektrische Schaltkreise realisiert. Da sich ihre Leistung immer mehr verbesserte, wurden die Switches nicht mehr nur in der Telekommunikation, sondern zunehmend auch in der Datenkommunikation eingesetzt.
Die Aufgaben dieser Switches sind neben dem Weiterschicken von Frames, das Umsetzen von Zellen in Pakete und umgekehrt, das Aufbauen, Halten und Abbauen von Verbindungen.
Ein LAN-Switch dient z.B. dazu, die Ethernet-Busstruktur in eine Sternstruktur umzuwandeln. Dabei können Teilsegmente auch weiterhin die klassische Busstruktur aufweisen. Auf diese Weise erhält man eine feine Segmentierung des vorhandenen lokalen Netzes. Betrachtet man unterschiedliche LAN-Switches, so kann man drei verschiedene Kategorien erkennen:

- **Cut-through-Switches**:
  Bei jedem Datenpaket wird nur die Ziel- und die Quelladresse benötigt und damit sofort die Verbindung durchgeschaltet. Dies ist möglich, da die Zieladresse den Empfänger eindeutig festlegt. Mit der erhaltenen Quelladresse wird zusätzlich eine Adreßtabelle im Switch aufgebaut. Nachteil ist, daß fehlerhafte Pakete und Paketfragmente ausgeliefert werden, da eine Prüfsummenberechnung mittels CRC erst am Ende des Pakets durchgeführt wird.

- **Store-and-Forward-Switches**:
  Diese Switches verhalten sich wie Brücken oder Router, d.h. sie speichern das gesamte empfangene Datenpaket in einem internen Puffer und beginnen erst dann mit der Fehlererkennung und Paketbearbeitung. Damit wird die Durchlaufzeit der Pakete durch den Switch wesentlich erhöht, vor allem bei sehr großen Paketen. Dies führt zu Problemen bei Sprach- oder Videoübertragungen, da ein kontinuierlicher Datenfluß nicht gewährleistet ist. Ein Vorteil dieser Art ist aber, daß defekte Frames erkannt und aussortiert werden.

- **Cell-Oriented-Switches**:
  Bei dieser Variante werden die Vorteile der beiden vorherigen Techniken kombiniert. Beim Empfang werden die Datenpakete in Zellen von 48 bis 64 Byte zerlegt und ohne Verzögerung weitertransportiert.

Obwohl auch bei dieser Technik Filterfunktionen verwendet werden, kann durch die konstante Länge der Zellen ein kontinuierlicher Datenfluß gewährleistet werden. Die Verzögerungszeit bleibt dabei in einem akzeptablen Bereich.

Hat ein LAN-Switch noch zusätzlich eine ATM-Schnittstelle, so spricht man von einem LAN/ATM-Switch.
ATM-Switches arbeiten allgemein verbindungsorientiert, d.h. es wird nicht jedes Paket einzeln verschickt, sondern es werden zwischen zwei Endpunkten einer Netzwerkverbindung mehrere virtuelle Kanäle aufgebaut, über die dann die Pakete geschickt werden. Bei dieser Übertragung kann man zwischen einer ATM-Pfadvermittlung und einer ATM-Kanalvermittlung unterscheiden.
Bei der ATM-Pfadvermittlung, auch ATM-Cross-Connect genannt, werden die ankommenden Pfade beendet und in einen neuen Pfad umgeleitet.
Dieser Pfad wird dann zum Weitertransport verwendet, wobei die im Pfad enthaltenen Kanäle von dieser Veränderung unberührt bleiben.
Betrachtet man die ATM-Kanalvermittlung, auch ATM-Switches genannt, so werden bei dieser Variante alle ankommenden Pfade und auch Kanäle beendet und in neue Pfade und Kanäle umgeleitet.
Für den Transport der ATM-Zellen werden innerhalb des ATM-Switches sogenannte *Switching Fabrics* verwendet, die den Weg zwischen dem Eingangs- und dem Ausgangsport zur Verfügung stellen. Eine *Switching Fabric* besteht selber wieder aus mehreren Schaltelementen (Switching Elements), wobei man dabei vier Elementtypen unterscheiden kann:

- **Matrix-Switching-Element**:
  Bei dieser Variante werden alle ATM-Zellen parallel über ein Netzwerkgitter übertragen. Alle Eingangs- und Ausgangsports sind somit über das Gitter verbunden. Um Blockierungen durch die synchrone Weiterleitung der Zellen zu vermeiden, ist an jedem Eingangs- und Ausgangsport ein Pufferspeicher eingerichtet.

- **Bus-Switching-Element**:
  In diesem Fall werden die ATM-Zellen über einen Hochgeschwindigkeitsbus übertragen. Um angemessene Ergebnisse erzielen zu können, muß die Übertragungskapazität des Busses genau so groß sein, wie die der Eingangsports. Außerdem muß am Ausgangsport ein Pufferspeicher realisiert werden, damit es zu keiner Überlastung der Ausgangsport durch die hohe Übertragungsrate des Busses kommt. Damit entspricht die Leistungsfähigkeit der Matrix-Variante.

- **Ring-Switching Element**:
  Im Gegensatz zum Bus-Switching-Element sind hier alle Inputports über einen Ring miteinander verbunden. Dadurch können mehrere Inputports den Time-Slot während eines Ringumlaufs gleichzeitig verwenden. Obwohl dadurch Overhead zur Steuerung dieses Mechanismuses nötig ist, wird eine effektive Auslastung der Ringkapazität von über 100% erreicht.

- **Central-Memory-Switching-Element**:
  Vom Eingangsport aus werden alle ATM-Zellen in einen Pufferspeicher geschrieben, von dem aus die Ausgangsports die Zellen wieder auslesen. Mit dieser zentralen Speichertopologie, die den Speicher effizient ausnützt, ist man in der Lage, viele Eingangs- und Ausgangsports zu verwenden, ohne Leistungseinbrüche erleiden zu müssen.

Betrachtet man nun die interne Architektur beim Hicom Xpress, so stellt man fest, daß er eine *Switching Fabric* mit Matrix-Switching-Elementen einsetzt.
Aus diesen einzelnen Schaltelementen wird nun durch Kopplung gleicher Switching-Elementen ein Schaltnetzwerk aufgebaut. Um die Zellen entlang eines Übertragungspfades in so einem Schaltnetzwerk zu vermitteln, gibt es zwei Verfahren:

- **Self-Routing**:
  Bei der Eigenvermittlung erhält die ATM-Zelle einen zusätzlichen Header, der den Pfad der Zelle durch das Schaltnetzwerk enthält, wobei pro Vermittlungsstufe ein Feld im Header belegt ist. Beim Verlassen des Switches wird dieser Header wieder gelöscht.

- **Table-Routing**:
  Bei diesem Verfahren wird die ATM-Zelle nicht verändert, sondern es wird aus dem VPI- und VCI-Wert ein vermittlungsspezifischer Wert berechnet, der den Ausgangsport für diese Zelle enthält. Diese in der Verbindungsaufbau- bzw. Wegewahlphase ermittelten Werte werden für die weitere Verwendung in einer Tabelle abgelegt.

Sollte nun bei der Vermittlung der einzelnen ATM-Zellen Probleme auftreten, z. B. daß der Switch seine Kapazitätsgrenze erreicht hat, so besteht die Möglichkeit, daß der Switch einzelne Zellen verwirft. Die Entscheidung, welche Zellen entfernt werden dürfen und welche nicht, wird durch das CLP-Bit im Header der ATM-Zellen angezeigt. Ein ATM-Zelle, die immer verworfen werden darf, ist z.B. die Leerzelle, die nur dazu dient, einen kontinuierlichen Datenstrom zu gewährleisten. Das CLP-Bit ist bei dieser Zelle auf eins gesetzt.

Zusammenfassend ergibt die Switchingtechnologie dann folgende Managementfunktionalität und -information.

Managementfunktionalität	**Managementinformation**
Identifizierung und Auswertung der Kanal- und Pfadidentifikation der ATM-Zelle	anhand von VCI und VPI wird eine ATM-Zelle identifiziert und mit dieser Information weitergeleitet
Transport der ATM-Zelle von einem Eingangsport zu dem zugehörigen Ausgangsport	wird mittels der VCI und VPI und bei Table-Routing anhand der aufgebauten Tabelle realisiert
Entscheidung, ob die erhaltene ATM-Zelle weitertransprotiert oder verworfen wird	CLP-Bit teilt dem Switch mit, ob die ATM-Zelle bei Überlast verworfen werden darf oder nicht
Erkennen verschiedener Zelltypen mittels der unterschiedlichen Kanal- und Pfadbelegungen	VPI=0, VCI=0, PT=0, CLP=1: Leerzelle VPI=0, VCI=0, PT=X: nicht zugewiesene Zelle VPI=0, VCI=XX (XX≠0 und XX≠1): Zelle für die Übermittlung von Schicht-1-Steuerinformationen VPI=Y (Y≠0), VCI=1: Zellen für die Meta-Signalisierung VPI=Y (Y≠0), VCI=5: Zellen für Punkt-zu-Punkt-Signalisierungsverbindung VPI=Z, VCI≠0: Zellen für die Übermittlung von OAM-Nachrichten

Hinweis: X, XX, Y und Z können jeden beliebigen Wert mit Ausnahme der oben angegebenen annehmen.

2.2 Überblick über die *Object Modeling Technique*

Seit 1987 wurde bei der Firma General Electric unter Führung von James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorenson die *Object Modeling Technique* (OMT) entwickelt. Im Sommer 1991 wurde dieses Verfahren schließlich veröffentlicht und ist bis heute eine der meist verbreiteten objektorientierten Analyse- und Designmethoden.
Mit dem OMT-Modell ist man in der Lage, die Objekte der realen Welt und ihre Beziehungen untereinander darzustellen. Dabei werden vor allem erweiterte Entity-Relationship-Diagramme, Zustandsdiagramme und Datenflußdiagramme verwendet.
Grundsätzlich besteht das OMT-Modell aus drei Komponenten:

- dem Objektmodell, das die statischen Zusammenhänge zwischen Klassen beschreibt und dabei auch Vererbung und Enthaltenseinsbeziehungen darstellt,
- dem Dynamikmodell, das die Verhaltensweisen (Aktionen, Zustände) aufzeigt und
- dem Funktionsmodell, das die Datentransformation repräsentiert.

Daneben können die OMT-Modelle auch noch Use-Case-, Event-Flow-, Event-Trace- und Kontext-Diagramme enthalten. In Abbildung 2-4 sind die Zusammenhänge der einzelnen Modelle nochmals dargestellt.

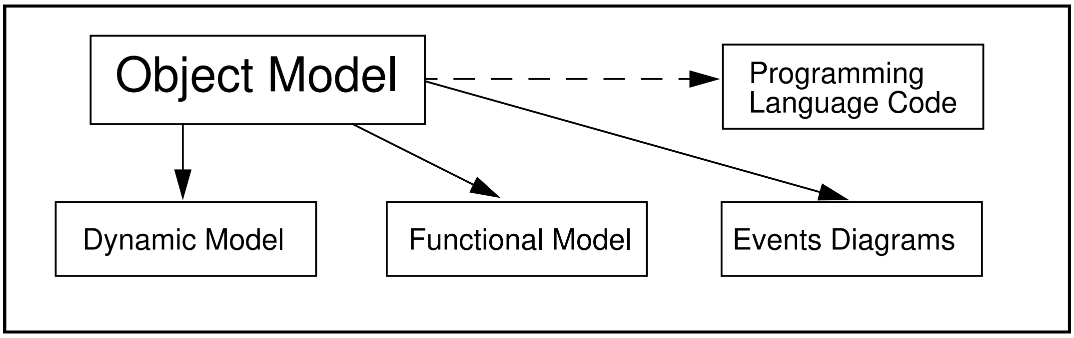


Abbildung 2-4: Übersicht des OMT-Modells

Es besteht prinzipiell die Möglichkeit, aus den entwickelten OMT-Modellen Code in verschiedenen Sprachen, wie C++, Smalltalk, IDL oder Ada95, zu generieren. Diese entwickelten OMT-Modelle werden, wie in Kapitel 4.1.1 erläutert, im Repository für die spätere Weiterverwendung abgelegt.
Im folgenden wird nun ein etwas detaillierterer Blick auf die Entwicklung eines OMT-Modells geworfen.

In der ersten Phase der Entwicklung, die auch als Analyse bezeichnet wird, soll aus einer Problembeschreibung ein **Analysemodell** entstehen, das mit einer genauen und komprimierten Problembeschreibung die Grundlage für die weiteren Schritte darstellen soll. Auch hier treten schon die Unterschiede zwischen den einzelnen Modellen auf.

So wird im **Objektmodell** die statische Struktur, im **dynamischen Modell** die Reihenfolge von Interaktionen und im **funktionalen Modell** die Transformation der Daten dargestellt. Da sich der Schwerpunkt dieser Arbeit auf die Entwicklung eines Objektmodells bezieht, werden die beiden anderen Modelle im weiteren Verlauf nicht weiter berücksichtigt.

Im Objektmodell wird versucht, eine statische Struktur zu entwickeln, die die reale Welt mittels Objekten, die ein bestimmtes Verhalten haben, abbildet.
Gibt es mehrere Objekte, die eine ähnliche Struktur aufweisen, so werden sie in **Klassen** zusammengefaßt. Diese bilden das wichtigste Element innerhalb des Objektmodells. Man kann jede Klasse als Prototypen für ein Objekt betrachten, das die in der Klasse definierten **Attribute** und **Operationen** an die anderen Objekte, auch als **Instanzen** bezeichnet, weitergibt.
Die in den Klassen enthaltenen Attribute beschreiben die Eigenschaften der Objekte, die bei jeder Instanz einer Klasse vorhanden sind, d.h. die Attribute stellen die Datenstruktur der Objekte einer Klasse dar. Die jeweiligen Attribute sind dabei in der Klasse eindeutig. Daneben gibt es noch die Operationen, die Funktionen oder Transformationen für alle Objekte einer Klasse festlegen. Zu beachten ist, daß eine Operation für mehrere Klassen gelten kann, d.h. die Operation wird in den Klassen unterschiedlich realisiert. Dieses Verhalten wird als "polymorph" bezeichnet, d.h. eine Operation mit diesem Verhalten ist "polymorph". Somit wird durch ein Attribut ein Zustand eines Objekts beschrieben, das durch die Methoden, die die Implementierung einer Operation für eine Klasse sind, verändert werden kann. Um die jeweiligen Klassen mit ihren Attributen und Methoden darstellen zu können, hat Rumbaugh eine eigene Notation entwickelt. Diese Notation ist inzwischen auch auf breiter Basis akzeptiert und sogar von anderen übernommen worden. Ein allgemeines Beispiel ist in Abbildung 2-5 dargestellt, wobei die Angaben wie Datentyp, Defaultwert, Parameter und Returnwert optional sind.

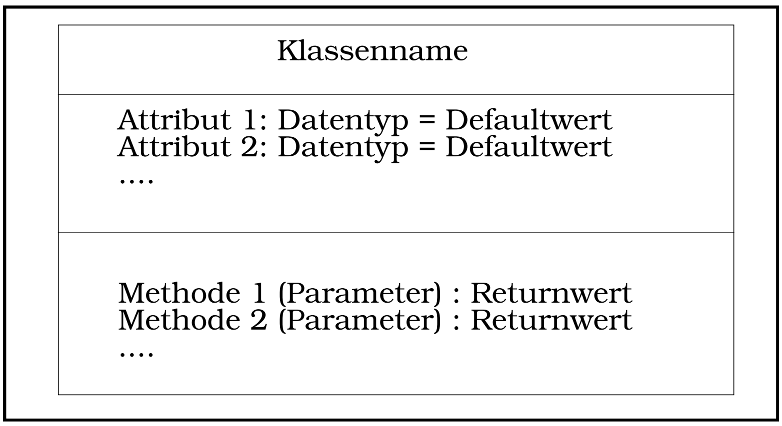


Abbildung 2-5: Klassennotation von OMT

Hat man alle Klassen aus der Problembeschreibung abstrahiert, ist es im nächsten Schritt notwendig, die Beziehungen der einzelnen Klassen untereinander zu beschreiben.
Beim OMT-Modell unterscheidet man zwischen drei unterschiedlichen Beziehungen:

- **Generalisierung**:
  Sie wird meist als “is a“- oder “kind of“-Beziehung bezeichnet und stellt das Konzept der Vererbung zur Verfügung. Man unterscheidet dabei zwischen der Oberklasse und den Unterklassen, die von der Oberklasse alle Merkmale erbt. Durch die Generalisierung entsteht eine Verfeinerung der Oberklasse in die jeweiligen Unterklassen, indem gleiche Attribute und Operationen in die Oberklasse übernommen werden. Die Generalisierung in der OMT-Notation (Abbildung 2-6) wird durch ein Dreieck dargestellt, wobei die Spitze mit der Oberklasse verbunden wird und die Unterklassen über eine waagerechte Linie mit der Grundlinie des Dreiecks. Zusätzlich besteht die Möglichkeit das Dreieck mit einem sogenannten “Diskriminator” zu ergänzen. Dies ist ein Attribut, das angibt, nach welcher Eigenschaft bei der Generalisierung unterschieden wird.

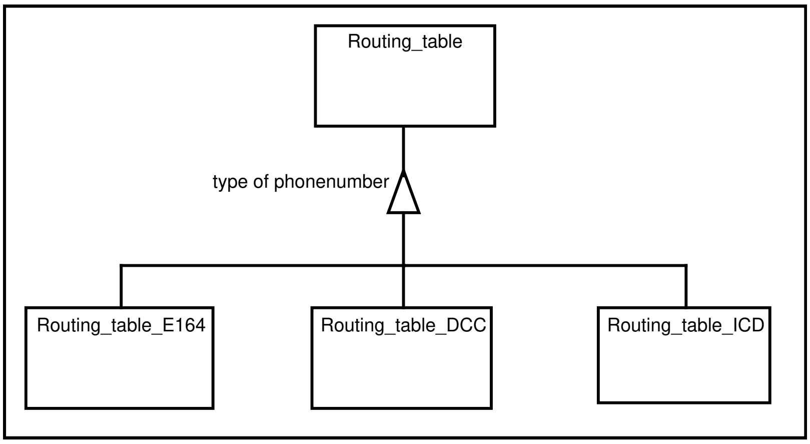


Abbildung 2-6: Beispiel für die Generalisierung

- **Assoziation**:
  Sie beschreibt eine Beziehung zwischen zwei Klassen, wobei die Assoziationen dabei inhärent bidirektional sind. Durch den Namen einer Assoziation wird die Richtung der Assoziation angegeben, wobei sie trotzdem in beide Richtungen durchlaufen werden darf. Die graphische Darstellung in der OMT-Notation für die Assoziation (Abbildung 2-7) ist eine Linie zwischen zwei Klassen. Da Assoziationen binärer, ternärer oder höherer Ordnung sein können, gibt es für allgemeine tertiäre und n-äre Assoziationen noch ein spezielles OMT-Symbol, die Raute. Von ihr gehen alle Linien aus, die die zueinander in Relation stehenden Klassen miteinander verbinden.

Man muß jedoch feststellen, daß es meistens nur binäre oder qualifizierte Assoziationen gibt. Daneben existiert noch der Begriff der Multiplizität, der angibt, wieviele Instanzen einer Klasse mit einer Instanz einer assoziierten Klasse verbunden sein können.
Dies wird im Objektmodell durch eine spezielle Notation am Ende der Assoziationslinie ausgedrückt. So ist ein fetter schwarzer Punkt das Symbol für “m”, d.h. null oder mehr. Ein transparenter Punkt dagegen steht für null oder eins. Gibt es kein besonderes Linienende, so ist es eine normale 1:1-Beziehung. Es besteht aber auch die Möglichkeit, die Multiplizität, auch manchmal Kardinalität genannt, durch einfache Angaben von Zahlen am Linienende zu spezifizieren.

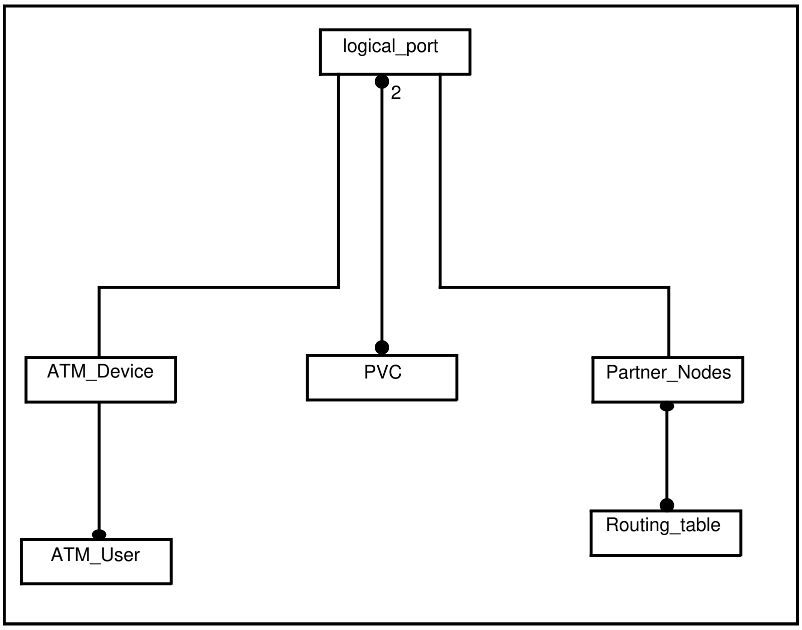


Abbildung 2-7: Beispiel für die Assoziationen

- **Aggregation**:
  Sie wird auch “part of“-Beziehung genannt und ist eigentlich ein Spezialfall einer Assoziation. Die wichtigsten Merkmale einer Assoziation sind die Transitivität und die Antisymmetrie. Bildlich gesprochen unterscheidet man zwischen einer Komponentengruppe und den einzelnen Komponenten. Die Darstellung in der OMT-Notation (Abbildung 2-8) ist ähnlich wie bei der Assoziation. Es ist eine Linie zwischen der Komponentengruppe und den einzelnen Komponenten, wobei unterhalb der Komponentengruppe eine kleine Raute gezeichnet wird.

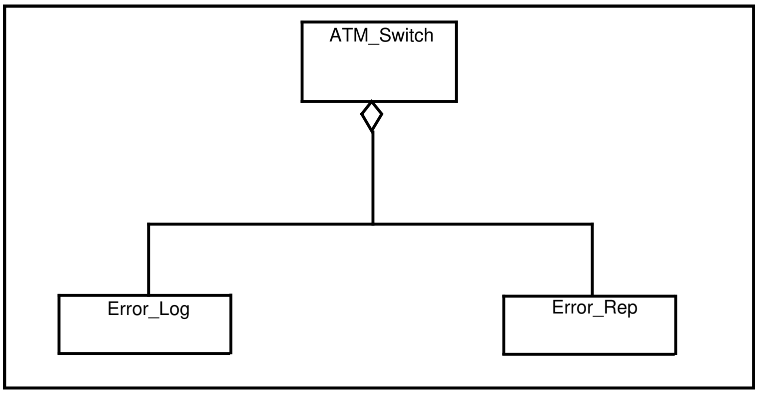


Abbildung 2-8: Beispiel für die Aggregation

Hat man nun unter Berücksichtigung all dieser Aspekte ein Modell entwickelt und die genauen Werte, Typen usw. von Attributen und Methoden festgelegt, so wird aus dem OMT-Modell Code in der gewünschten Programmiersprache generiert. Würde man die Modellierung jetzt noch weiter vorantreiben, so wäre im nächsten Schritt die Modellierung des dynamischen Modells und im Anschluß die des funktionalen Modells nötig. Für genauere Informationen über die Entwicklung eines Objektmodells wird auf Kapitel 4 verwiesen. Dort wird detailliert ein Objektmodell für das Management von ATM-Switches entwickelt.

2.3 Überblick über CORBA

Die *Common Object Request Broker Architecture* (CORBA) ist ein Standard von der *Object Management Group* (OMG) und wurde im Jahre 1991 zum erstenmal veröffentlicht. Die OMG mit Hauptsitz in Framingham, Massachusetts, USA, ist ein Zusammenschluß vieler verschiedener und internationaler Unternehmen, zu denen Softwareentwickler, Hardwarehersteller, Netzbetreiber, kommerzielle Nutzer von Computersoftware, als auch Universitäten gehören.

Gründungsmitglieder im Mai 1989 waren Christopher Stone (jetzt Präsident der OMG) und die folgenden acht Unternehmen: 3Com Corporation, American Airlines, Canon Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems und Unisys Corporation. Doch es blieb nicht bei diesen Mitgliedern. Bis heute stieg die Zahl stetig an, da die OMG im Oktober 1989 eine unabhängige und nicht profitorientierte Organisation wurde. Heute zählt die OMG circa 750 Mitglieder und ist damit zu einem starken Softwarekonsortium aufgestiegen. Ziel dieser Organisation ist es, die Interoperabilität, die Wiederverwendbarkeit und die Portabilität von Softwarekomponenten durch den objektorientierten Ansatz zu fördern. Daraus entstanden weltweite Standards wie CORBA/IIOP, *Object Services*, *CORBA Facilities* und *Domain Interfaces*.

Zur Zeit werden CORBA-Spezifikationen auf über 50 Betriebssystemen weltweit realisiert. Ebenso verwenden Firmen wie Netscape, Oracle, Sun oder IBM das Protokoll IIOP (Internet Inter-ORB Protocol) als Grundlage für ihre eigenen Produkte. Daran erkennt man, daß die Spezifikationen der OMG eine nicht zu unterschätzende Rolle bei der Entwicklung von verteilten Anwendungen spielen.

Im Mittelpunkt steht dabei die OMA (Object Management Architecture), die das Rahmenwerk für alle Aktivitäten bildet (siehe Abbildung 2-9).

Die Grundlage der OMA bilden Objekte, die die reale Welt modellieren. Das Verhalten eines Objekts kann man dabei über den Aufruf von Operationen beeinflussen. Die einzelnen Möglichkeiten dafür werden am sogenannten Interface zur Verfügung gestellt. Dort werden auch die Zugriffsmöglichkeiten dargelegt. Um dieses Interface beschreiben zu können, wurde von der OMG eine eigene Sprache definiert, die *Interface Definition Language* (IDL). Mit dieser IDL ist man in der Lage, für alle Objekte die sichtbaren Attribute, Operationen und Datentypen festzulegen.

Durch das sogenannte *Language-Mapping* werden die in IDL beschriebenen Interfaces in die verwendete Programmiersprache umgewandelt. Wichtig ist dabei, daß die Objekte in einer bekannten Programmiersprache wie z.B. C, C++ oder Java implementiert werden. Dadurch ist die Definition der Interfaces programmiersprachen-, aber auch betriebssystemunabhängig.

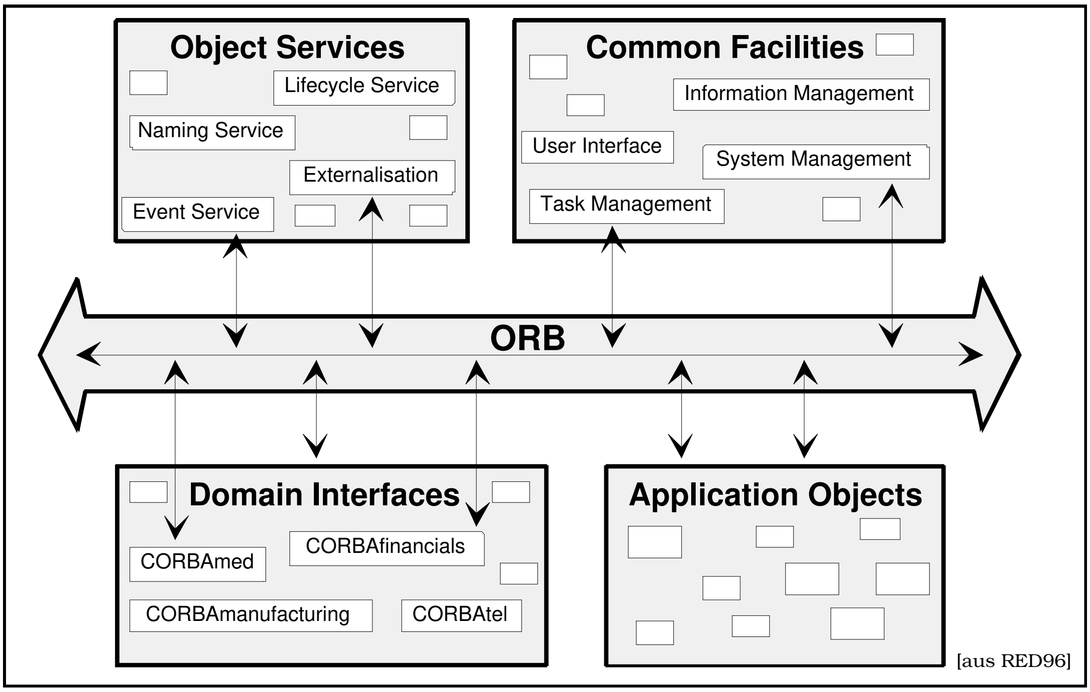


Abbildung 2-9: Komponenten der OMA

Diese nun definierten Objekte werden dann in den fünf verschiedenen Komponenten der OMA zusammengefaßt. Diese Zuordnung ist jeweils abhängig vom Dienst, der das Objekt liefert.

Die Komponente, die von allen Objekten genutzt wird, ist der **ORB** (Object Request Broker), dessen Eigenschaften durch den CORBA-Standard spezifiziert werden. Er ist das Herzstück der OMA und durch seine Arbeit als Vermittlungszentrale wird die Kommunikation zwischen den einzelnen Objekten ermöglicht. Der ORB nimmt Objektanfragen entgegen und lokalisiert bzw. übermittelt dann die Anfragen und Ergebnisse. Damit wird eine Voraussetzung geschaffen, um die Interoperabilität zwischen verschiedenen Anwendungen herzustellen.

Eine weitere Komponente bilden die ***Object Services***. Diese stellen grundlegende Funktionen zur Verfügung, die wiederum in IDL spezifiziert sind. Aufgaben, die mit den *Object Services* realisiert werden, sind z.B. das Auffinden von Objekten im Netz (Naming Service), das Umgehen mit asynchronen Ereignismeldungen (Event Service) oder die langfristige Speicherung von Objektzuständen (Persistent Object Service). Leider ist im Gegensatz zum ORB die Standardisierung in diesem Bereich noch nicht vollständig abgeschlossen.

Neben den Funktionen der *Object Services* gibt es noch die Funktionen der ***Common Facilities***. Sie liefern eine Sammlung von Anwendungsfunktionen, die auf den *Object Services* aufbauen, und die je nach Anforderungen der Anwendung konfiguriert werden können. Beispiele sind Drucker- und Dokumentenmanagement, Datenbanken und E-Mail.

Lösungen für spezielle Anwendungsgebiete werden in den ***Domain Interfaces*** zusammengefaßt, früher auch *Vertical Market Facilities* genannt. Anwendungsbeispiele sind Softwarepakete für das Finanzwesen (CORBAfinance) oder für die Telekommunikation (CORBAtel). Eine Standardisierung ist aber noch nicht vorhanden.
Die letzte Komponente in der OMA bilden die ***Application Objects***. Diese Objekte stellen die Anwendungen dar, die vom Endbenutzer als konkretes Produkt gesehen werden. Da diese Anwendungen meist nur für ein spezielles Problem eine Lösung bieten, werden die *Application Objects* von der OMG nicht standardisiert.

Nachdem nun die einzelnen Komponenten der OMA bekannt sind, soll im weiteren der ORB noch näher betrachtet werden.
Um den Kommunikationsmechanismus zwischen dem Client und dem Server zu realisieren, sind mehrere Schnittstellen notwendig, die in der Abbildung 2-10 graphisch dargestellt sind.

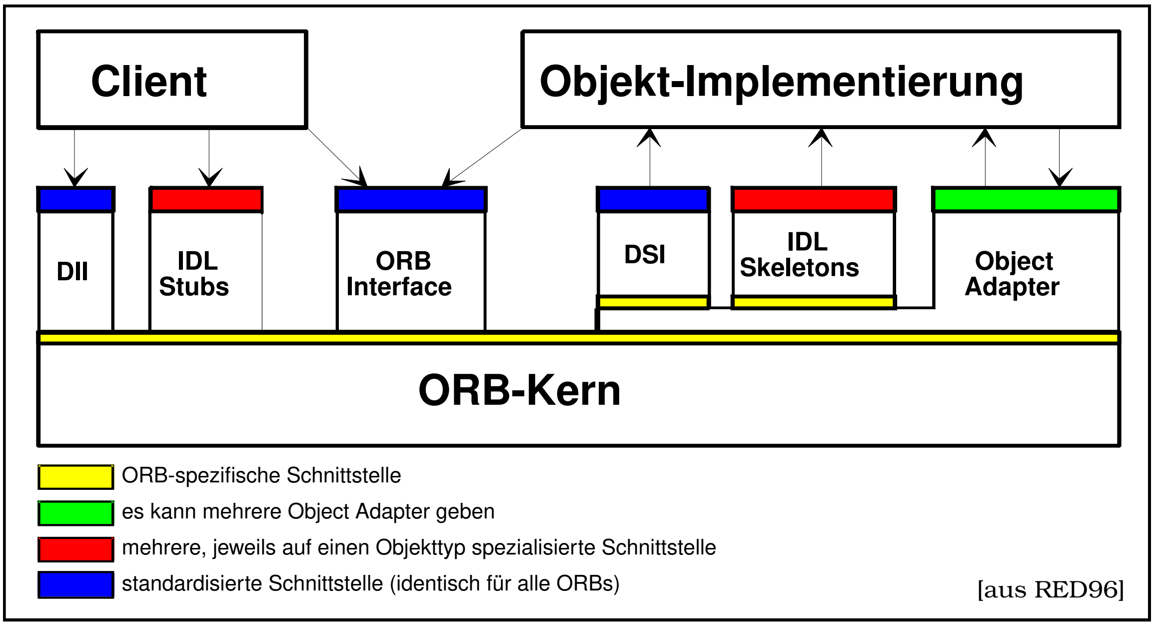


Abbildung 2-10: Schnittstellen des ORB

Der **Client** ist eine Programmeinheit, die eine Operation in einer Objekt-Implementierung aufruft. Dieser Aufruf, auch Request genannt, soll für den Client transparent sein, d.h. er kennt nur den Aufruf, die nötigen Parameter und eventuell ein Kontextobjekt. Die gesamte Implementierung des Objekts bleibt für den Aufruf bei dieser Realisierung verdeckt.
Die **Objekt-Implementierung** enthält dagegen den Programmcode für die jeweiligen Operationen, die durch den Request des Client aufgerufen werden können. Damit ist ein CORBA IDL-Interface realisiert, wobei die Objekt-Implementierung in verschiedenen Sprachen implementiert sein kann. Diese schickt nach der Bearbeitung des Requests die Anwort, auch Response genannt, an den Client zurück.

Der **ORB-Kern** liefert einen Mechanismus, um die Requests des Clients an die Objekt-Implementierung weiterzuleiten. Dabei übernimmt der ORB die Lokalisierung der Objekt-Implementierung und die Konvertierung der Daten, falls unterschiedliche Datenformate verwendet werden. Natürlich ist der ORB auch für die Versendung des Response verantwortlich. Da der ORB-Kern auf unterschiedliche Weise implementiert sein kann, ist es für die Anwendungen wichtig, die Implementierungsdetails davon abzukoppeln. Dafür wurde von der OMG eine ORB-spezifische Schnittstelle definiert.

Dieses **ORB-Interface** liefert verschiedene Hilfsfunktionen, wie die Konvertierung von Objektreferenzen in Zeichenketten und umgekehrt oder das Erstellen von Argumentenlisten für Requests, die über das DII (Dynamic Invocation Interface) geliefert werden.

Das **DII** ermöglicht einem Client den direkten Zugriff auf den ORB, wobei der Request in diesem Fall dynamisch erzeugt wird. Erst zur Laufzeit werden vom Nutzer selber oder aus dem IR (Interface Repository) die nötigen Informationen mitgeteilt.
Das IR ist dabei ein Teil des ORB und stellt die Funktionalität einer Datenbank zur Verfügung. Im IR sind alle die Informationen über ein CORBA-Objekt enthalten, die durch die IDL-Definitionen festgelegt werden. Damit steht eine Informationsbasis für die Clients und die Objekt-Implementierungen zur Verfügung, die auch von entfernten Rechnern abgefragt werden können.
Mit dem DII besteht auch die Möglichkeit *oneway-calls* oder einen *deferred synchronous request* auszuführen. Bei einem *oneway-request* muß der Aufrufer das Ende der Operation nicht abwarten und verhindert dadurch, daß der Client wie bei einer synchronen Methode blockiert wird. Dies ist aber nur möglich, wenn die Methode kein Ergebnis zurückliefert.
Im Gegensatz dazu steht der *deferred synchronous request*. Der Client wartet hier zwar ebenso das Ende nicht ab, fragt aber das Ergebnis der Methode zu einem späteren Zeitpunkt asynchron ab.

Die im Gegensatz zum DII einfachere Lösung liefern die ***IDL Stubs***. Sie dienen als Bindeglied zwischen dem Client und der Objekt-Implementierung bzw. dem ORB. Das jeweilige Interface wird bereits bei der Programmentwicklung definiert und durch den IDL-Compiler generiert und anschließend in das Programm eingebunden. Somit sind zur Laufzeit weniger Informationen nötig und die Anwendung vereinfacht sich im Vergleich zum DII. Die Objekt-Implementation erkennt übrigens nicht, ob der Request den *IDL Stub* oder das DII benutzt hat, d.h. sie sind völlig gleichwertig.

Auf der Seite der Objekt-Implementierung unterscheidet man zwischen IDL Skeletons und einem DSI (Dynamic Skeleton Interface), das weiter unten erklärt wird.

Ein ***IDL Skeleton*** ist jeweils für einen speziellen Interface-Typ reserviert und wurde aus der jeweiligen IDL-Definition mit dem IDL-Compiler generiert.
Bei der Verwendung von *Orbix* und der Programmierspache C++ ist dies die BOAImpl-Klasse. Indem der ORB mit der Hilfe des Skeletons eine Methode bzw. Funktion der Objekt-Implementierung aufruft, ist das Ziel des Requests erreicht.

Die andere Möglichkeit ist das **DSI**, das das Gegenstück zum DII darstellt. Das DSI erlaubt einem ORB, Requests an die Objekt-Implementierung zu liefern, obwohl zur Compilierungszeit der Typ des Objekts noch nicht bekannt ist. Auch in diesem Fall ist nicht erkennbar, welche Variante verwendet wurde.

Damit haben wir den Weg des Requests vom Client zur Objekt-Implementierung beschrieben. Will dagegen eine Objekt-Implementierung auf den ORB zugreifen, so verwendet sie entweder den OA (Object Adapter) oder das ORB Interface.
Der **OA** unterstützt den ORB bei der Lieferung von Requests an die Objekte und bei der Aktivierung von Objekten, um zum Objekt gehörende Daten zu ermitteln. Abhängig von den zu unterstützenden Objekten und dem Stil der Objekt-Implementierung werden unterschiedliche OA zur Verfügung gestellt. Bisher sind bei der OMG aber erst zwei OA standardisiert. Der BOA (Basic Object Adapter) wird bei normalen verteilten Anwendungen eingesetzt, damit verteilte Objekte miteinander kommunizieren können. Der OODB (Object-Oriented Database Adapter) wird dagegen bei Datenbanken eingesetzt, um z.B. ein Objekt automatisch zu rekonstruieren.

Um nun selbst ein CORBA-Objekt zu definieren, muß die Schnittstelle in IDL beschrieben werden. Das durch den IDL-Compiler entstandene IDL-Skeleton muß dann noch mit dem entsprechenden Code aufgefüllt werden. Dieses erstellte CORBA-Objekt wird als Server bezeichnet und muß nun noch beim *Implementation Repository* angemeldet werden. Dies ist ebenfalls ein Bestandteil des ORB. Es enthält Informationen zum Auffinden und Aktivieren von Objekt-Implementationen, da der Client beim Aufruf zwar eine Objektreferenz mitliefert, diese aber über den physikalischen Aufenthaltsort des Objekts nichts aussagt. Dadurch besteht die Möglichkeit, die Objekt-Implementierung jederzeit auf einen anderen Rechner zu übertragen oder sie auf mehreren Rechnern bereitzustellen. Diese Aktivitäten werden vom Client während der Kommunikation nicht wahrgenommen und werden eingesetzt, um die Belastung einzelner Rechner zu steuern bzw. zu verringern. Damit besteht die Möglichkeit, die Ausfallsicherheit eines Systems zu erhöhen.

Nachdem wir die Arbeitsweise von CORBA näher kennengelernt haben, wird noch auf den Begriff **Interoperabilität** eingegangen.

Arbeiten Hard- oder Softwarekomponenten zusammen, so spricht man von Interoperabilität. Dieser Begriff wird aber abhängig vom Umfeld unterschiedlich betrachtet.
Es gibt Interoberabilität zwischen Objekten, Programmiersprachen, Rechnern, verschiedenen ORBs, verschiedenen Kommunikationsplattformen (CORBA oder Microsoft OLE/COM) oder verschiedenen Diensten und Anwendungen.
Betrachtet man die Interoperabilität bezüglich CORBA, so ist vor allem die Zusammenarbeit von verschiedenen ORBs interessant.
Bei dieser Zusammenarbeit stehen unterschiedliche Möglichkeiten zur Verfügung:

- ***General Inter-ORB Protocol*** **(GIOP)**:
  Einheitliches Protokoll für die Kommunikation zwischen Client-seitigem ORB und Server-seitigem ORB. Eine Spezialisierung des GIOP ist das IIOP (Internet Inter-ORB Protocol), das sich auf die TCP/IP-Protokollfamilie abstützt. Daneben gibt es noch spezielle Protokolle, die gleichzeitig noch auf eine bestimmte Umgebung abgestimmt sind. Sie werden *Environment-Specific Inter-ORB-Protocols* (ESIOPs) genannt. Ein Beispiel ist das DCE-CIOP (DCE Common Inter-ORB Protocol), das auf OSF/DCE basiert.

- **Brücken (bridges)**:
  In diesen Brücken werden die jeweils eigenen Protokolle der ORBs umgewandelt, damit die ORBs miteinander kommunizieren können. Man unterscheidet zwischen Vollbrücken und Halbbrücken. Die Vollbrücken setzen unmittelbar das Protokoll des einen ORB in das Protokoll eines anderen ORB um. Die Halbbrükken dagegen wandeln das Protokoll erst in ein allgemeines, verbreitetes Protokoll, wie IIOP, um und anschließend in das Protokoll des Ziel-ORB.

Verweise auf weitere Informationen über CORBA oder einzelne Spezifikationen sind im Literaturverzeichnis enthalten. Das bereits kurz angesprochene Produkt *Orbix* von IONA auf Basis von CORBA wird in Kapitel 5.2 vorgestellt.

3 Projektvorstellung

In diesem Kapitel wird sowohl der ATM-Switch Hicom Xpress der Firma Siemens, als auch die *Hicom Xpress Administration* (XAD) eingeführt. Nach diesen Vorstellungen in den ersten zwei Teilkapiteln, wird in Kapitel 3.3 das genaue Vorgehen für den Entwurf und die Implementierung des Objektmodells beschrieben.

3.1 ATM-Switch Hicom Xpress

Die Grundlage für die vorliegende Arbeit ist ein Breitband Kommunikationssystem auf Basis der ATM Technologie mit dem Namen Hicom Xpress. Wie in Abbildung 3-1 dargestellt, besteht der ATM-Switch Hicom Xpress aus einem *Switching Block* (SB) und einem *Control Block* (CB).

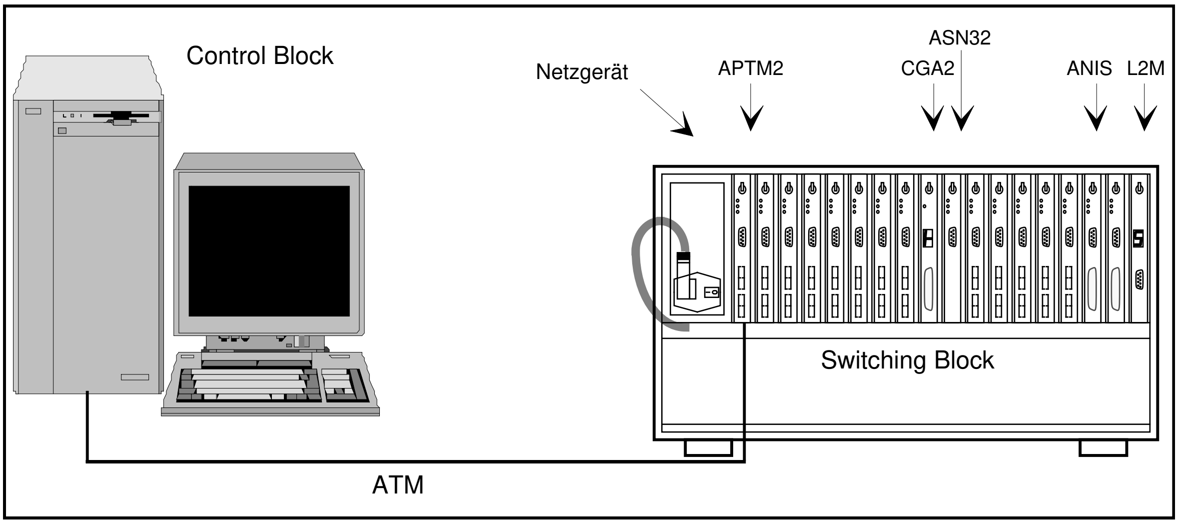


Abbildung 3-1: Hicom Xpress

Der ***Switching Block*** (SB) ist ein Baugruppenrahmen mit 18 Steckplätzen, die von links nach rechts, bei eins beginnend, durchnumeriert sind. Bei diesen vorhandenen Steckplätzen sind drei mit definierten Baugruppen belegt: auf Steckplatz 9 die Baugruppe CGA2 (siehe unten), auf Steckplatz 10 die Baugruppe ASN32 und auf Steckplatz 18 die Baugruppe L2M. Die übrigen Steckplätze können mit den Baugruppen APTM2 oder ANIS bestückt werden.
Zum besseren Verständnis folgt eine kurze Beschreibung der einzelnen Baugruppen:

- **CGA2 (Clock Generator Agent)**:
  Taktgenerator für den *Switching Block*; er kann sich als Slave auf den externen Referenztakt aufsynchronisieren oder selbst einen genauen Takt liefern.

- **ASN32 (ATM Switching Network)**:
  ATM-Koppelfeld-Baugruppe, über die die Verbindungen geschaltet werden.

- **L2M (Layer 2 Unit)**:
  Zentrale Steuerungsbaugruppe, die für die Schicht-2-Behandlung verantwortlich ist; sie kommuniziert über eine APTM2-Baugruppe mit dem *Control Block*.

- **APTM2 (ATM Port Trunk Module 2)**:
  Breitband-Anschlußbaugruppe mit zwei optischen STM1-Ports mit jeweils 155 MBit/s; als Glasfasertypen können MultiMode und SingleMode LWL eingesetzt werden. Die Baugruppe ist verantwortlich für den Anschluß anderer ATM-Endgeräte, anderer ATM-Knoten und für den Anschluß des *Control Blocks*.

- **ANIS (ATM Narrowband Interface Service)**:
  Schmalband-Anschlußbaugruppe mit einem S2-Port mit 2 MBit/s (entspricht einem ISDN-Primärmultiplexanschluß); über diesen Port ist man in der Lage, eine Hicom 300 Telefonanlage anzuschließen.

Der ***Control Block*** (CB) besteht aus einem PC mit Workstationadapter (WSA). Der PC ist ein Pentium PC mit einem Ethernet-Controller. Damit ist der CB in der Lage, mit anderen CBs sowie mit einer SNMP-Network-Management-Station zu kommunizieren. Als Betriebssystem wurde zu Beginn das UNIX System V Release 4 kompatible Echtzeit-Betriebssystem *SORIX* verwendet. Inzwischen ist bei Siemens auf das Betriebssystem *Unixware* umgestiegen worden. Der im PC eingebaute Workstationadapter dient zur Verbindung des CBs mit dem SB, um den Steuerdatenstrom untereinander auszutauschen.

Bevor im nächsten Teilkapitel auf das Tool für die *Hicom Xpress Administration* eingegangen wird, soll zunächst ein kleines Beispielszenario betrachtet werden (Abbildung 3-2), das als Testumgebung bei der Firma Siemens zur Verfügung steht.

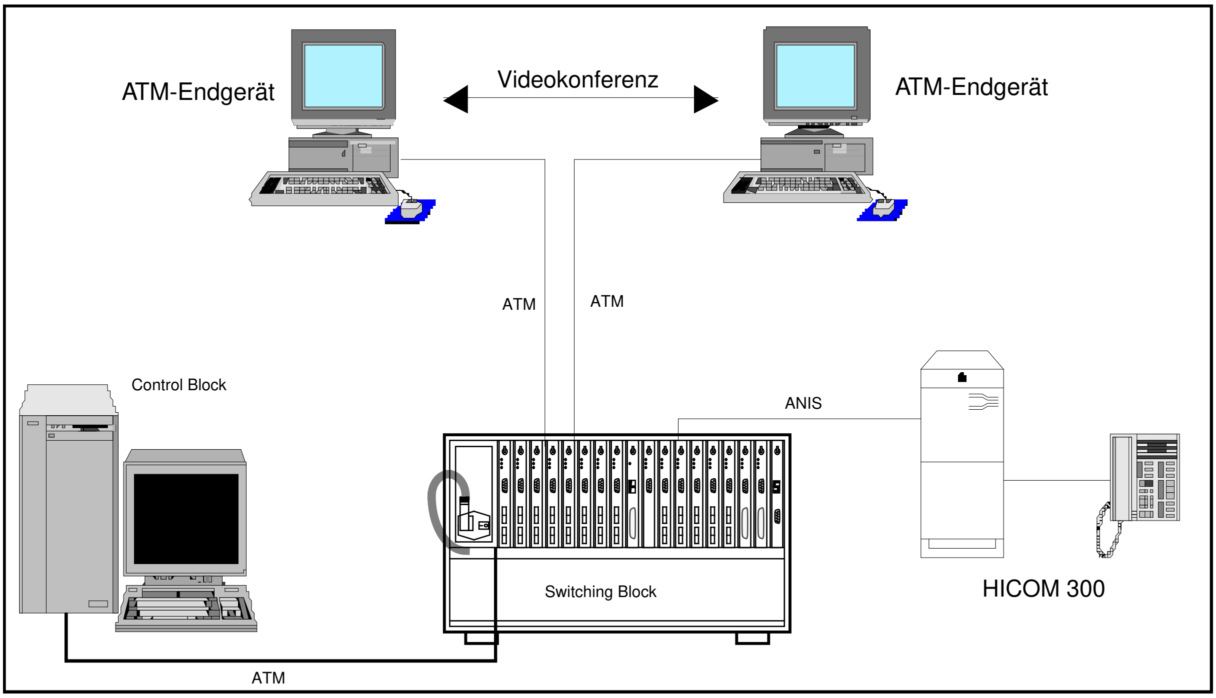


Abbildung 3-2: Beispielszenario Hicom Xpress

Mit dem ATM-Switch Hicom Xpress stehen dem Anwender einige Möglichkeiten bezüglich der Kommunikation zur Verfügung. Auf der einen Seite kann er die Datenkommunikation zwischen zwei Endgeräten mit hoher Bandbreite nutzen. Andererseits kann er aufgrund der hohen Bandbreite von bis zu 155 MBit/s auch Videoconferencing zwischen zwei oder mehreren ATM-Multimedia-PCs realisieren. Neben dieser Funktionalität besteht außerdem noch die Möglichkeit, Telefonie zu betreiben, d.h. durch den Anschluß einer HICOM 300 Telefonanlage an die ANIS Baugruppe sind Telefongespräche über den ATM-Switch möglich. Man kann an diesem kurzen Beispiel sehr gut erkennen, daß mit dem Hicom Xpress eine Hardwarekomponente mit großer Funktionalität zur Verfügung steht.

3.2 Hicom Xpress Administration

Die *Hicom Xpress Administration* (XAD) ist eine proprietäre Schnittstelle für die Administration des ATM-Switches Hicom Xpress. Sie ist als UNIX-Prozeß auf dem CB realisiert und führt über eine X-Windows/Motif-Oberfläche die Administration durch.

Die Gesamtaufgaben der XAD lassen sich in vier Gruppen aufteilen:

- Konfigurationsmanagement
- Installationsmanagement
- Fehlermanagement
- Sicherheitsmanagement

Zur **Konfiguration** zählt die Erstinbetriebnahme und Konfigurationsänderung wie z.B. das Festlegen der Sollkonfiguration des Switches. Die **Installation** umfaßt die Verwaltung der Devices, User, Partner Nodes, logischer Ports und PVCs. Beim **Fehlermanagement** werden Fehlermeldungen und defekte Einheiten angezeigt und Möglichkeiten zur Fehlerbeseitigung bereitgestellt. Das **Sicherheitsmanagement** ist nicht sehr stark ausgeprägt und dient nur zur Authentifikationsverwaltung, d.h es wird damit nur festgelegt, welche XAD-Zugriffsrechte für die jeweilige Kennung zur Verfügung stehen. Aufgaben wie die Überwachung des Systems auf Sicherheitsangriffe oder die Verschlüsselung von Informationen ist nicht realisiert.

Betrachtet man die Oberfläche der XAD (Abbildung 3-3), so sind mehrere Objektgruppen erkennbar. In den jeweiligen Objektgruppen sind eine oder mehreren Funktionen enthalten.

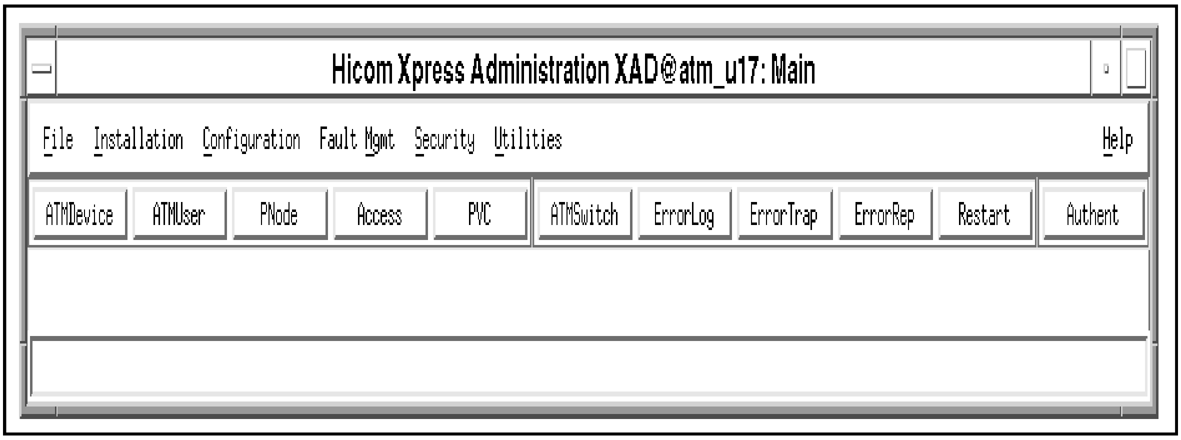


Abbildung 3-3: Hicom Xpress Administration

Außerdem lassen sich die jeweiligen Administrationsfunktionen, die jeweils einem Button im XAD-Window entsprechen, wieder einem der oben angegebenen vier Funktionsbereiche zuordnen.

Diese vier Bereiche lassen sich auch in den einzelnen Menüpunkten des Main-Window wiederfinden. Jedoch ist zu berücksichtigen, daß die Freigabe von Buttons oder Menüs von der jeweiligen Authentifizierungsberechtigung abhängig ist.
In der Statuszeile werden die jeweiligen Ergebnisse der Aufrufe oder Zusatz- bzw. Hilfsinformationen dargestellt. Im Menüpunkt "Utilities" werden noch zusätzliche Hilfsprogramme wie ein Datenbankbrowser oder Informationen zum Prozeßstatus zur Verfügung gestellt. Über den Menüpunkt "Help" gelangt man zur Online-Hilfe der XAD.

Im Folgenden sind die Features aufgeführt, die dem Administrator bei der XAD über die einzelnen Buttons zur Verfügung stehen:

Konfigurationsmanagement:
Button ATMSwitch:
- Sollkonfiguration festlegen
- Konfiguration anzeigen
- Manueller oder automatischer Konfigurationsabgleich
- In- und Außerbetriebnahme von Komponenten
- Anzeige defekter Komponenten

Button Restart:
- Restart auslösen

Installationsmanagement:
Button Access (logischer Port):
- Device Verwaltung
- Partner Node Verwaltung

Button ATMDevice:
- Device Verwaltung
- User Verwaltung

Button ATMUser:
- Device Verwaltung
- User Verwaltung

Button PNode:
- Partner Node Verwaltung

Button PVC:
- PVC Verwaltung

Fehlermanagement:

Button ErrorLog:

- Fehlerspeicher ausgeben

Button ErrorTrap:

- Bedingungen für Fehlerreports definieren

Button ErrorRep:

- Aktuelle Fehlerreports ausgeben

Bei dieser Aufstellung ist zu beachten, daß bei der XAD unter dem Konfigurationsmanagement nur die Konfigurierung des Switches mit seinen Boards und Ports realisiert ist. Damit ist zu diesem Zeitpunkt noch keine Kommunikation mit anderen Komponenten möglich. Erst durch das Installationsmanagement werden dann die Verbindungen zu Usern oder anderen Devices usw. hergestellt. Eine detailliertere Vorstellung der einzelnen Funktionen ist in Kapitel 4.2 zu finden.
Wird einer dieser Buttons aktiviert, so wird ein sogenanntes “Processing Window” geöffnet. Dieses Fenster dient zum Auflisten, Auswählen und Bearbeiten eines Objekts. Daneben gibt es noch die “View Windows”, bei denen nur die Informationen gelesen, aber nicht verändert werden dürfen. Um Daten zu verändern, werden an den entsprechenden Stellen dann sogenannte “Modify Windows” bereitgestellt. Zur Verdeutlichung ist im folgenden ein Beispiel für ein “Processing Window” (Abbildung 3-4), ein “View Window” (Abbildung 3-5) und ein “Modify Window” (Abbildung 3-6) dargestellt.

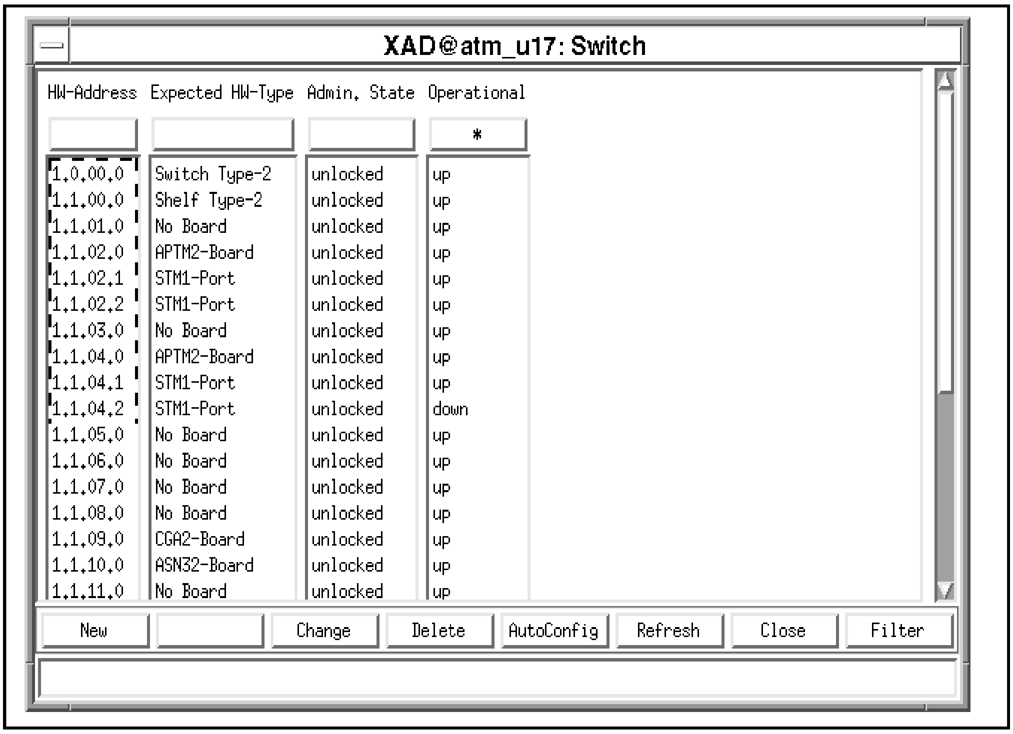


Abbildung 3-4: Processing Window “ATMSwitch”

In diesem Fenster werden nun die Hardware-Adressen der jeweiligen Komponenten, wie Switch, Board und Port angezeigt. Daneben gibt es eine Spalte für den erwarteten Hardwaretyp, die angibt, welches Board an welchem Steckplatz im Switch zur Verfügung steht.
In den beiden letzten Variablen wird der administrative Zustand der jeweiligen Hardwareeinheit dargestellt. Bei “Admin State” gibt es drei Zustände:

- locked: alle Verbindungen unterbrochen
- unlocked: normaler Betriebszustand
- ignore: wie der Zustand “locked”, wobei in diesem Fall auch keine Traps ausgesendet werden

Bei der Spalte “Operational” dagegen gibt es nur zwei Betriebszustände:

- down: Hardwareeinheit ist nicht nutzbar
- up: Hardwareeinheit ist nutzbar

Neben diesen einzelnen Informationen, die zur Verfügung stehen, werden einige Operationen angeboten. In diesem Bearbeitungsfenster sind es: “New, Change, Delete, AutoConfig und Refresh”. Damit können z.B. neue Boards eingerichtet oder gelöscht, aber auch Portdaten verändert werden. Die Funktionen “Close” und “Filter” sind Fensteroperationen, die auf den Switch keinen direkten Einfluß haben.
Wird nun in einem Processing Window ein Eintrag, in Abbildung 3-4 z.B. eine spezielle Hardwareadresse mit einem Doppelklick ausgewählt, so wird ein View Window geöffnet. Das View Window gibt die Daten des gewählten Objekts wieder. Im vorliegenden Fall (Abbildung 3-5) werden die Daten zum “Devicenamen: TD 1 (aptm 12/1)” aus der Objektgruppe “ATM Device” angezeigt.

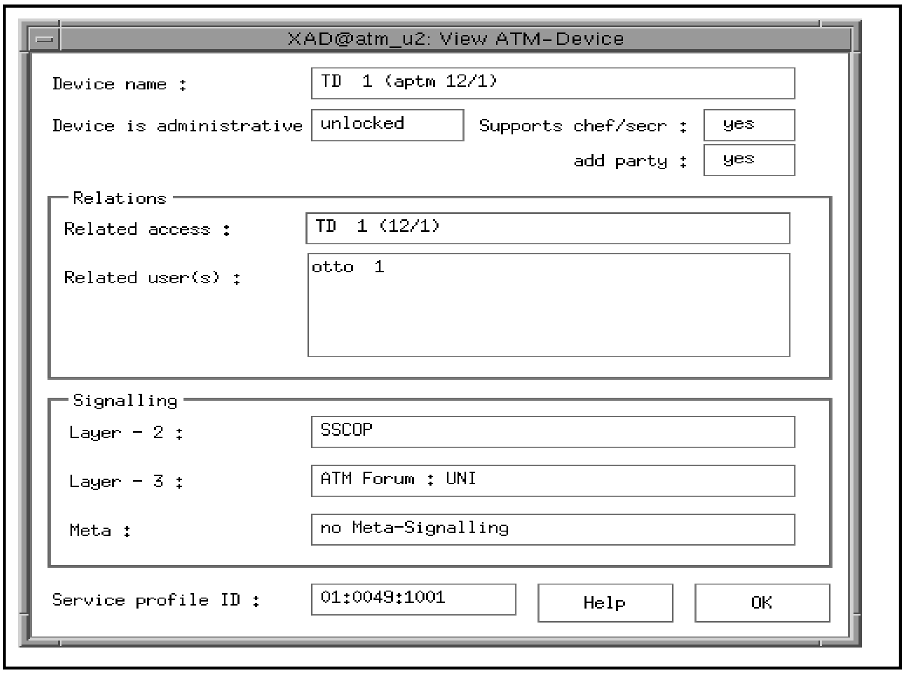


Abbildung 3-5: View Window: “ATMDevice”

Die hier dargestellten Daten können im “View Window” jedoch nur angezeigt und nicht durch eigene Eingaben verändert werden.
Ebenso wird das “View Window” auch vom System nicht automatisch aktualisiert, d.h. Veränderungen im System, wie löschen des Objekts, werden in diesem Fenster nicht angezeigt.
Dies ist bei der Arbeit mit der XAD zu beachten, da ein “View Window” jederzeit aktiv sein kann und man trotzdem im “Processing” oder “Modify Window” arbeiten kann, ohne daß die Daten aktualisiert werden.
Veränderungen an einzelnen Objekten können somit nur im “Modify Window” durchgeführt werden. In Abbildung 3-6 ist ein “Modify Window” für die Objektgruppe ATM Device dargestellt und man kann beim Vergleich mit Abbildung 3-5 sehr gut erkennen, welche Daten des Objekts verändert werden können.

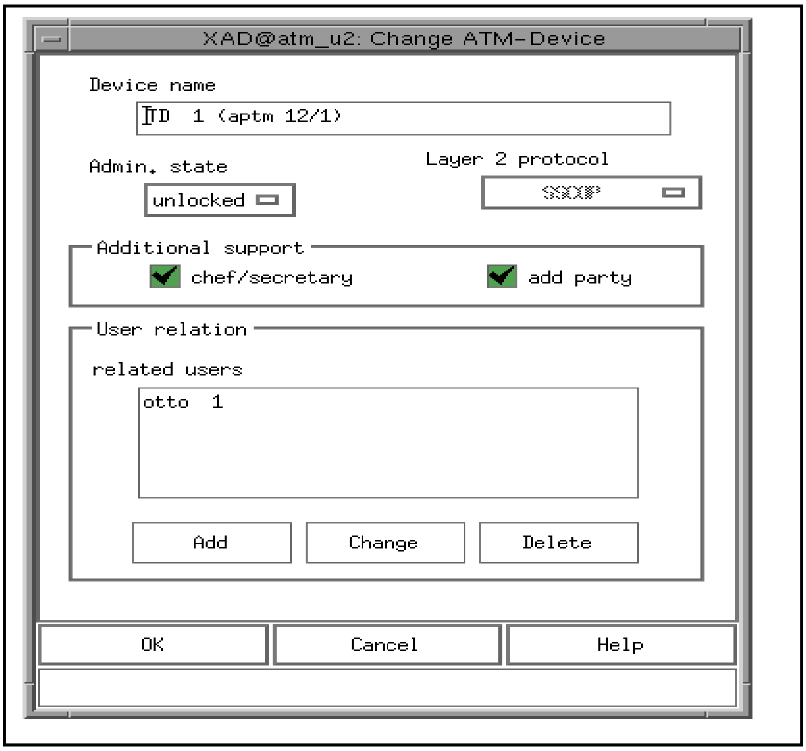


Abbildung 3-6: Modify Window: “ATMDevice”

Im vorliegenden Beispiel sind es der Devicename, der Administrationsstatus, der zusätzliche Support und der mit dem Device in Verbindung stehende User. Andere Einträge können nicht verändert werden. Bei diesem Vorgang ist zu beachten, daß parallele Eingaben in mehreren “Modify Windows” nicht erlaubt sind. Für genauere Informationen bezüglich der Funktionalität und der von der XAD zur Verfügung gestellten Parameter wird auf Kapitel 4.2 verwiesen. Auch das Servicehandbuch [SIE97] der Firma Siemens bietet weitergehende Informationen über die XAD.

Abschließend kann man sagen, daß die Anforderungen an das neue GUI doch sehr vielseitig sind. Einerseits soll dem Anwender die Arbeit durch gute Übersichtlichkeit, d.h. durch eine möglichst geringe Anzahl von eingesetzten Fenster, erleichtert werden, andererseits darf dabei aber nicht die Gefahr bestehen, daß Inkosistenzen auftreten. Somit muß gewährleistet sein, daß gleichzeitige Veränderungen am gleichen Attribut nicht möglich sind. Auch die übersichtliche Darstellung der Attribute und die intuitive Bedienbarkeit der Oberfläche sind wichtige Aspekte bei der Entwicklung. Wichtig ist auch, daß die benötigten Informationen schnell zur Verfügung stehen, d.h. der Aufbau der jeweiligen Fenster darf nicht zu lange dauern. Zusammenfassend kann man sagen, daß sich das neue GUI in den meisten Punkten doch an der XAD anlehnen kann und nur einige Änderungen bei der Gestaltung der Fenster und bei der Anzahl der Fenster nötig sind.

3.3 Vorgehensmodell

Die Basis der Arbeit ist der ATM-Switch Hicom Xpress mit dem proprietären Managementtool XAD. Die Zusammenarbeit zwischen Xpress und XAD ist wie folgt realisiert:
Der Xpress besteht, wie schon bekannt, aus einem *Switching Block* und einem *Control Block*. Der *Switching Block* (SB) führt die Signalisierung bis zur Schicht 2 durch und wickelt sicherheitstechnische Aufgaben ab. Der *Control Block* (CB) steuert mit der vermittlungs-, sicherheits- und bedientechnischen Software den *Switching Block*. Somit ist die Software der XAD auf dem CB installiert. In Abbildung 3-7 ist diese Struktur noch etwas detaillierter dargestellt. Auf der linken Seite sehen wir den Switch, der über ein Management-API von der XAD verwaltet wird. Das API wird außerdem noch von einem SNMP-Manager über einen SNMP-Agenten genutzt. Dieser ist für die weitere Arbeit aber nicht von Bedeutung. Der für die Arbeit wichtigste Teil ist einerseits das Management-API und die proprietäre GUI.

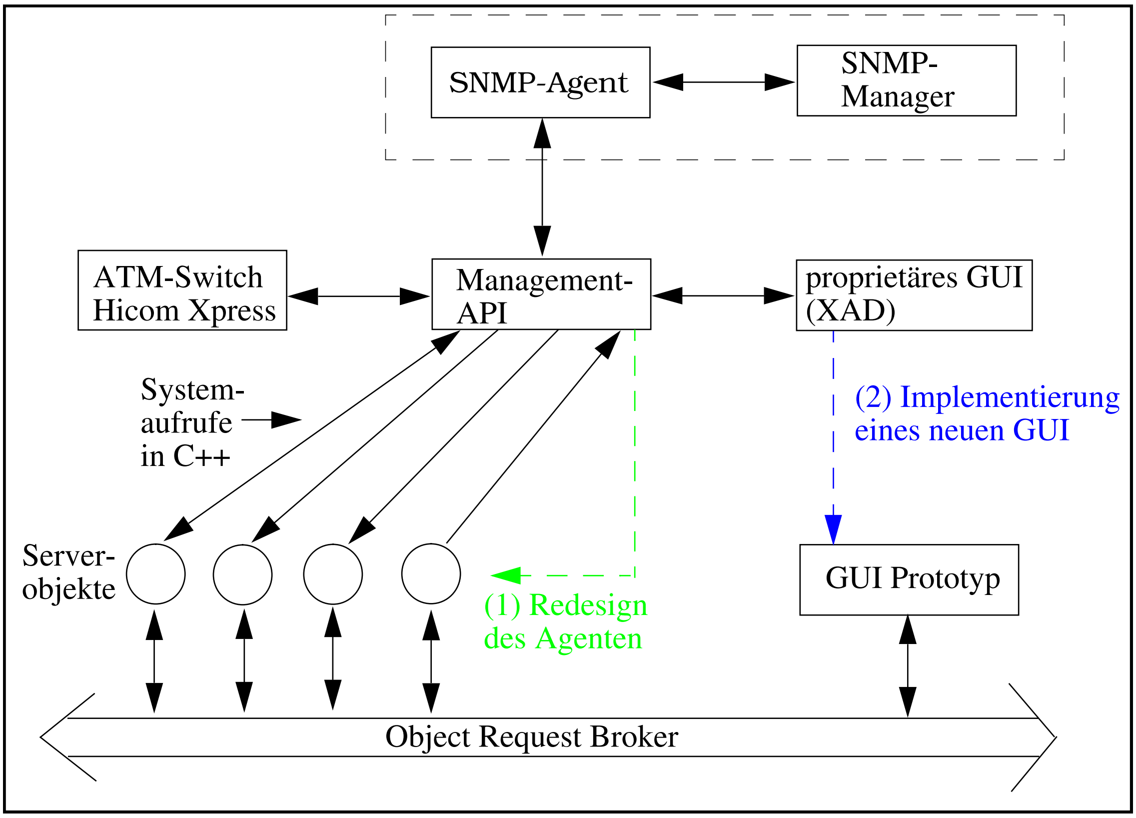


Abbildung 3-7: Ablaufszenario

Der erste Teil der Arbeit beschäftigt sich hauptsächlich mit dem Management-API, das die Grundlage für die Entwicklung des Objektmodells ist. In dem API sind alle Systemaufrufe enthalten, die zum Management des Switches nötig sind. Diese sind in C++ implementiert und stehen für die weiteren Schritte zur Verfügung.

Bei der Entwicklung des Objektmodells wird nach zwei unterschiedlichen Sichten, der Bottom-up- und der Top-down-Sicht, vorgegangen.
Durch die Verwendung dieser beiden Sichtweisen wird die Problemstellung von zwei Seiten betrachtet. Startet man mit der Bottom-up-Sicht, so nähert man sich dem Problem von der technischen Seite, d.h. man stützt sich z.B. auf eine vorhandene Komponente oder ein Objektmodell und entwickelt auf dieser Grundlage etwas Neues. Bei diesem Weg besteht aber die Gefahr, daß man sich zu stark an die vorhandenen Gegebenheiten hält und zu wenig auf die Universalität des Produkts Rücksicht nimmt. Um dieser Fehlerquelle zu begegnen, wird mit der Top-down-Sicht versucht, sich von der anderen Seite zu nähern, d. h. man versucht die Anforderungen, die an das zu entwickelnde Produkt bzw. die Technologie gestellt werden, vollständig zu erfassen. Damit entstehen Basisinformationen, auf die bei Neu- oder Weiterentwicklungen ohne Probleme zurückgegriffen werden kann. Hat man diesen Schritt abgeschlossen, werden zusätzliche produktspezifische Details ergänzt.

Im Folgenden werden diese Schritte auf die Arbeit übertragen. Auch hier beginnt man bei der Bottom-up-Sicht, d. h. man versucht aus der XAD und den zugehörigen Management-APIs die vorhandene Funktionalität herauszufiltern. Mit diesen Informationen wird ein erstes Objektmodell, das den grundsätzlichen Aufbau der XAD wiedergibt, entwickelt. Beziehungen zwischen den einzelnen Klassen sind noch nicht berücksichtigt und werden erst im zweiten Objektmodell dargestellt. Beim zweiten Objektmodell werden auch die einzelnen Klassen mit ihren Attributen und Methoden überarbeitet. Aber erst im dritten Objektmodell ist die komplette Funktionalität der XAD dargestellt und alle Veränderungen und Ergänzungen eingebaut. Da aber die XAD ein proprietäres Managmenttool ist, kann dieses entwickelte Objektmodell für andere Komponeten nicht oder nur mit eingeschränkter Funktionalität eingesetzt werden. Deshalb wird mit der Verwendung der Top-down-Sicht versucht, dieses Objektmodell noch zu optimieren. Dabei wird einerseits die verwendete Technologie, d.h. ATM, aber auch die Managementaufgaben zur Verwaltung eines ATM-Switches untersucht. Man erhält damit Managementszenarien, die die jeweiligen durchzuführenden Aufgaben eines Administrators festlegen. Die Darstellung dieser Ergebnisse wird dabei mit *Use Cases* (siehe Kapitel 4.6.1) realisiert. Mit diesen *Use Cases* wird das dritte entwickelte Objektmodell überprüft und versucht, einen generischen Anteil herauszugreifen. Dieser Anteil läßt sich auf Grund der verwendeten Technologien, ATM und Switching, nochmals in zwei Teile spalten, die aber die Fuktionalität nicht beeinflussen. Daran kann man nun sehr gut erkennen, daß durch die Verwendung der Top-down-Sicht Anteile im Objektmodell herausgegriffen wurden, die es einem Entwickler sehr leicht erlauben, Veränderungen bei den verwendeten Technologien anzupassen. Dieser generische Anteil im Objektmodell bietet aber auch die Möglichkeit, bei Neu- oder Weiterentwicklungen ohne große Probleme mit diesem zu arbeiten,

da keine komponentenspezifischen Funktionen in diesem Teil des Modells vorhanden sind. Sie stellen im Objektmodell einen eigenen Bereich, der nur die Funktionalität des generischen Anteils erbt, dar.
Verwendet man beispielsweise einen anderen ATM-Switch, so muß nur dieser spezifische Teil verändert werden.

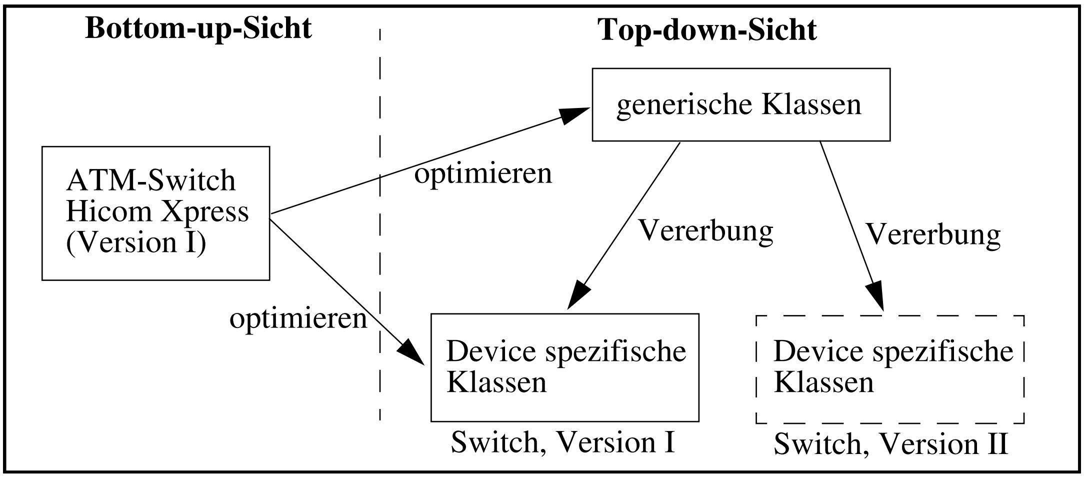


Abbildung 3-8: Redesign des Switches

Diese Vorgehensweise (Abbildung 3-8) ist in der Entwurfsphase komplizierter und arbeitsintensiver, als wenn ein Objektmodell nur für einen speziellen ATM-Switch entwickelt wird. Dieser Mehraufwand wird sich aber zu einem späteren Zeitpunkt amortisieren, da man mit dem entwickelten Modell die Basisfunktionalität jedes Switches sofort verwenden kann. Nur spezielle Attribute oder Methoden, die nicht im generischen Objektmodell enthalten sind, müssen noch eingebunden werden. Dies ist aber auch nicht sehr kompliziert, da die neuen Klassen die Funktionalität der generischen Klassen erben und sie diese damit sofort einsetzen kann.

Hat man nun dieses Redesign für das Management eines Switches abgeschlossen, gilt es, das entwickelte Objektmodell zu implementieren. Dazu wird für jede definierte Objektklasse des Objektmodells IDL-Code generiert (siehe Abbildung 3-9). Nach einer Überprüfung des IDL-Codes, ob alle Beziehungen zwischen den Klassen korrekt abgebildet wurden, wird dieser im nächsten Schritt mit einem IDL-Compiler compiliert. Als Ergebnis entstehen Header-Dateien, Client- und Server-Stubs und Implementierungsvorlagen. Diese Implementierungsvorlagen enthalten Funktionsrümpfe, die für die weitere Verwendung noch mit zusätzlichem Code ergänzt werden müssen. In diesem Fall sind es die vorhandenen Systemaufrufe des ATM-Switches, die durch das Management-API zur Verfügung stehen.
Werden diese abschließend compiliert, entstehen Server-Objekte, die von noch zu implementierenden Client-Objekten über einen ORB angesprochen werden können.

Hat man diese ersten großen Entwicklungsschritte durchgeführt, so ist ein Prototyp für das Management eines ATM-Switches entstanden (siehe Abbildung 3-7, grüner Pfeil).

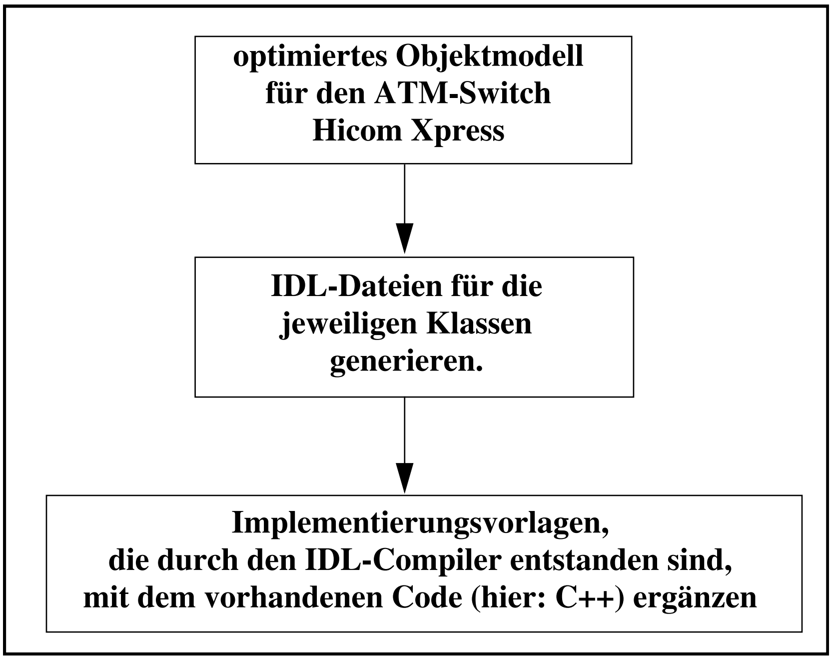


Abbildung 3-9: Implementierungsschritte

Der zweite Arbeitsschritt (Abbildung 3-7, blauer Pfeil) liegt nun darin, für den Prototypen noch eine Benutzeroberfläche zu entwerfen, da der Xpress nach der Neuimplementierung nur noch über ein Kommandozeileninterface administriert werden kann. Dies liegt daran, daß die bisherige Oberfläche auf das proprietäre Management-API zugeschnitten ist und somit für den Prototypen nicht mehr eingesetzt werden kann, da sich vor allem die Kommunikation zwischen der Oberfläche und dem Management-API geändert hat. Die neu zu entwerfende Oberfläche soll dabei ohne große Veränderungen auf unterschiedlichen Plattformen verwendbar sein. Die Basis dafür bildet die bisherige Oberfläche der XAD. Sie wird vom Aussehen grundsätzlich übernommen und nur in einigen Bereichen an die veränderten Attribute und Methoden angepaßt.

Größere Veränderungen sind aber bei der Anbindung an den Switch nötig, da sonst eine Plattformunabhängigkeit nicht gewährleistet ist. Die Kommunikation soll in Zukunft ebenfalls wie beim Prototypen des Managementagenten über einen ORB realisiert werden. Die Anbindung des GUI an den ORB wird mit Java realisiert, da auch für Java ein *Language Mapping* (genauso wie bei C++) zur Verfügung steht.

Betrachtet man diese Realisierung rückblickend, so kann man sagen, daß das GUI im CORBA-Umfeld den Client darstellt, der über den ORB auf die einzelnen Server-Objekte zugreift. Damit liegt am Ende der Arbeit ein Prototyp mit graphischer Oberfläche zum Management eines ATM-Switches vor.

4 Modellierung der Objektmodelle

Das vorliegende Kapitel 4 beschäftigt sich mit der Modellierung mehrerer Objektmodelle. Zur Einführung der Thematik wird das zur Arbeit verwendete CASE-Tool *Software through Pictures* vorgestellt. Vor der Erstellung des ersten Objektmodells wird die Funktionalität der XAD vorgestellt. Auf dieser Grundlage werden die einzelnen Klassen, Attribute und Methoden erzeugt und in einem Objektmodell zusammengefaßt. Nachdem man nun dieses spezielle, auf die XAD zugeschnittene Objektmodell entwickelt hat, wird im nächsten Abschnitt mit Hilfe von *Use Cases* ein generisches Modell entwickelt. Zum Abschluß werden diese beiden Modelle zusammengeführt und man erhält ein optimiertes Objektmodell, das aus einem generischen und einem devicespezifischen Anteil aufgebaut ist.

4.1 Das CASE-Tool *Software through Picture*s (StP)

4.1.1 Übersicht

Mit dem CASE-Tool *Software through Pictures* (StP) der Firma Aonix steht eine Sammlung von Werkzeugen für die Entwicklung von objektorientierter Software zur Verfügung.
Diese Sammlung von Tools wurde im September 1996 von IDE (jetzt: Aonix) und Thomson Software Products eingeführt und ermöglicht einen gemeinsamen Arbeitskontext und vor allem eine gemeinsame Datengrundlage in einem einheitlichen Desktop. Diese Datengrundlage ist bei der vorliegenden Arbeit das OMT-Modell von Rumbaugh et al., wobei auch die Möglichkeit besteht, andere Modelle, wie das von Booch zu verwenden. Unabhängig vom verwendeten Modell, haben alle StP Produkte eine gleiche Kernarchitektur mit folgenden Diensten:

- StP Desktop
- Object Annotation Editor (OAE)
- Diagram Filters
- Repository Browser
- Printing
- Object Management System
- Query and Reporting System

Bevor wir uns im folgenden näher mit den einzelnen Komponenten befassen, ist in der Abbildung 4-1 eine Übersicht aller Komponenten, die im StP/OMT enthalten sind, dargestellt.

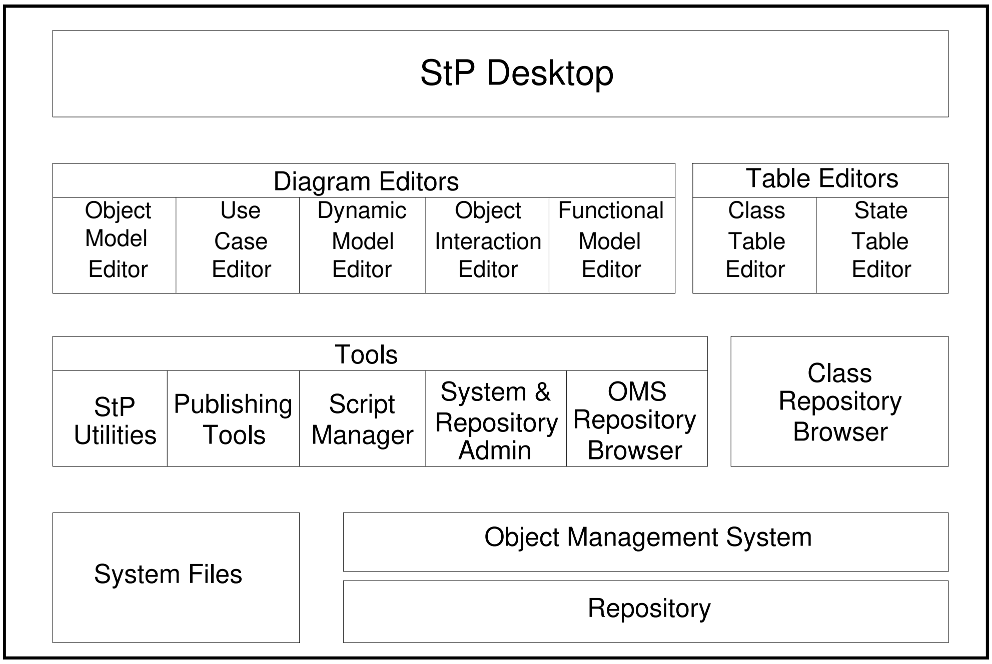


Abbildung 4-1: Übersicht über StP/OMT

StP Desktop:
Der StP Desktop ist der Schlüssel, um sowohl die jeweiligen Editoren, als auch die StP Utilities zu starten. Daneben können von dieser Oberfläche aus produktspezifische und administrative Befehle ausgeführt werden. Eine nähere Betrachtung mit kleiner Einführung folgt im Kapitel 4.1.2.

Diagram Editors:
Wie aus Abbildung 4-1 ersichtlich ist, gibt es bei StP/OMT fünf verschiedene Typen von Diagrammeditoren, die am Ende des Kapitels 4.1.2 jeweils kurz vorgestellt werden. Damit ist man in der Lage, ein OMT-Modell von Grund auf zu entwickeln. Einzelheiten zum OMT-Modell wurden im Kapitel 2.2 näher erläutert.

Table Editors:
Bei den Tabelleneditoren unterscheidet man den Klassentabelleneditor und den Zustandstabelleneditor. Sie bieten die Möglichkeit, auf knappe und geordnete Weise spezielle Informationen zu den jeweiligen OMT-Modellen darzustellen, wie z.B. den Typ eines Attributs oder die Argumente einer Operation.

Tools:
Mit den unterschiedlichen Werkzeugen werden dem User weitere Möglichkeiten angeboten, um die erstellten OMT-Modelle weiter zu bearbeiten. Daneben kann man sich mit den *Publishing Tools* z.B. die erstellten Modelle ausdrucken lassen oder sich mit dem *Skriptenmanager* Quellcode erzeugen lassen.

System Files:
Jede Objektreferenz, die in einem Objektmodell vorkommt, wird in einer ASCII-Datei gespeichert. So werden in einer Diagrammdatei die Referenzen abgespeichert, die durch das Auftreten eines Symbols im Diagramm entstehen. Kommen mehrere gleiche Symbole im Diagramm vor, so werden auch mehrere Referenzen abgespeichert. Daneben gibt es noch Tabellendateien und Dateien, die dem Systemadministrator zur Administration von StP zur Verfügung stehen.

Object Management System:
Mit dem StP *Object Management System* (OMS) werden Beziehungen zwischen den StP Produkten und dem Repository geliefert. Enthalten sind:
- *Persistent Data Model* - Schema für alle Objekte im Repository
- *Application Programming Interface* - eine Bibliothek von Funktionen, um Objekte im Repository zu manipulieren, die mit C oder C++ Programmen gelinkt wurden
- *OMS Query Language* - Abfragesprache für das Repository
- *Type Extension* - Methoden, um die Semantik von gespeicherten Daten im Repository zu erweitern

Repository:
Die Daten der jeweiligen OMT-Modelle, die von StP als *application types* bezeichnet werden, sind im Repository abgelegt, einem zentralen Datenpool. Zur Speicherung werden die *application types* in *persistent object types* umgewandelt, und die Informationen in der Datei “app.types“ abgelegt. Um die Bedeutung des Objekts und die Wechselbeziehungen zu beschreiben, wurde das *Persistent Data Model* (PDM) eingeführt. Es ist ein allgemeines statisches Datenmodell, um verschiedene Typinformationen darzustellen. Spezielle Informationen zur Darstellung des Objekts im Editor sind im PDM nicht mehr enthalten.
Bei diesem Verfahren ist jedes Objekt im Repository eindeutig. Diese Eindeutigkeit wird auch beibehalten, wenn mehrere gleiche Objekte im Modell auftreten, da jedes Objekt eigenständig betrachtet und abgespeichert wird.
Die auf diese Weise im Repository abgelegten Daten, können mit dem *Class Repository Browser* untersucht werden.

4.1.2 Der StP Desktop und die Graphik-Editoren

Um den StP Desktop auf der Oberfläche zu starten ist folgendes Kommando nötig:

stp [&]

Nach dessen Ausführung wird ein Fenster geöffnet, das der Abbildung 4-2 entspricht.

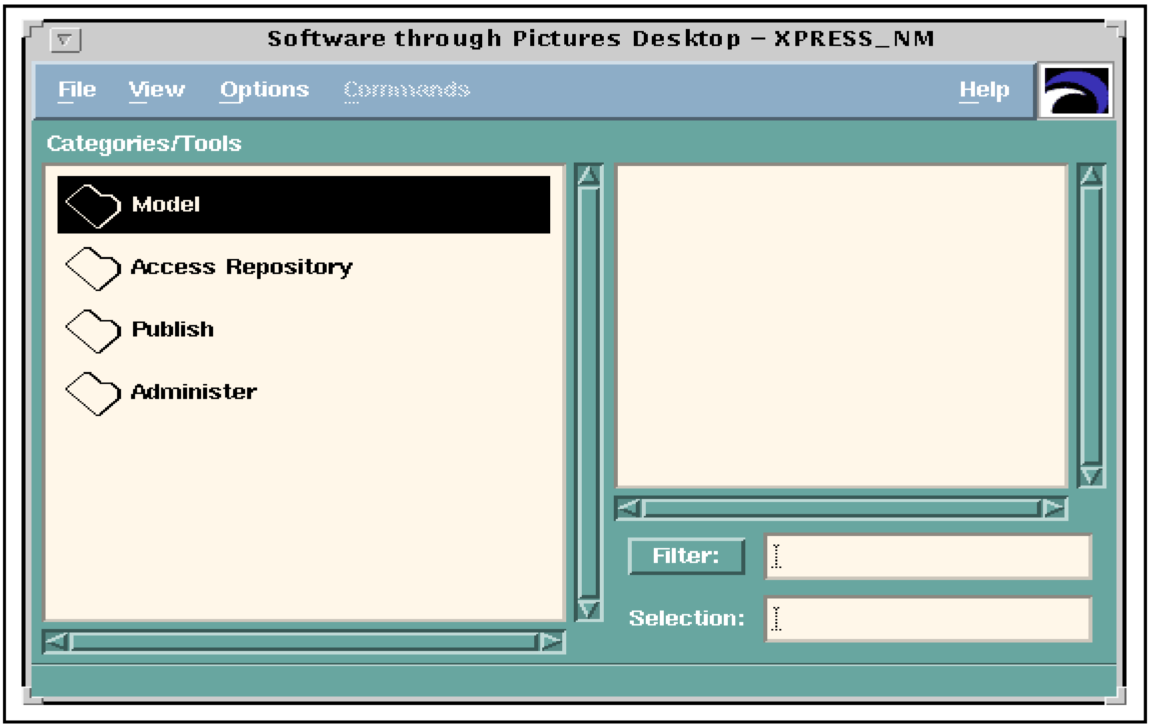


Abbildung 4-2: StP Desktop

Wie in der Abbildung 4-2 zu erkennen ist, gibt es zwei wichtige Bereiche: die Menüleiste und eine Liste von unterschiedlichen Klassen und Werkzeugen. Daneben gibt es noch eine Objektliste und einen Bereich, in dem Nachrichten des Systems, wie Fehlermeldungen, wiedergegeben werden.

In der Menüleiste gibt es folgende fünf Menüs:

- File Menu:
  Kommandos zur Auswahl von Projekten (Verzeichnissen) und Systemen (Dateien) und zum Beenden des Desktops
- View Menu:
  Aufruf der Datei “Message Log“
- Options Menu:
  Setzen von Optionen für das Desktop und die “Message Log“ Datei
- Commands Menu:
  Je nach Auswahl eines Werkzeuges Bereitstellen von unterschiedlichen Kommandos zur Ausführung
- Help Menu:
  Zugang zur Online-Dokumentation des StP

In der Klassen/Werkzeug-Liste befinden sich alle verfügbaren Werkzeuge, sowohl für StP selber als auch die Produkte wie StP/OMT. Da es zu umfangreich ist, auf alle Werkzeuge einzeln einzugehen, werden im folgenden nur einige grundlegende Desktopfunktionen benannt; weitere Informationen dazu sind in [SPF96] enthalten:

- Starten eines neuen Editors, z.B. *Object Model Editor* (OME)
- Öffnen eines Diagramms oder einer Tabelle
- Kontrolle der Semantik von Diagrammen, Tabellen oder Modellen
- Kopieren, Umbennen oder Löschen von Diagrammen und Tabellen

In der Objektliste werden dann zusätzlich zu den ausgewählten Werkzeugen noch weitere Objekte, wie Repository Objekte, angezeigt.

Der zentrale Punkt eines OMT-Modells (siehe Kapitel 2.2) ist das Objektmodell, das aus einem Diagramm und einer Klassentabelle besteht. Dabei ist das Diagramm eine Darstellung der statischen Struktur eines vorhandenen oder geplanten Systems. Zur Erstellung des Modells wird die Notation von OMT verwendet, die durch den Objektmodelleditor (Abbildung 4-3) zur Verfügung gestellt wird.

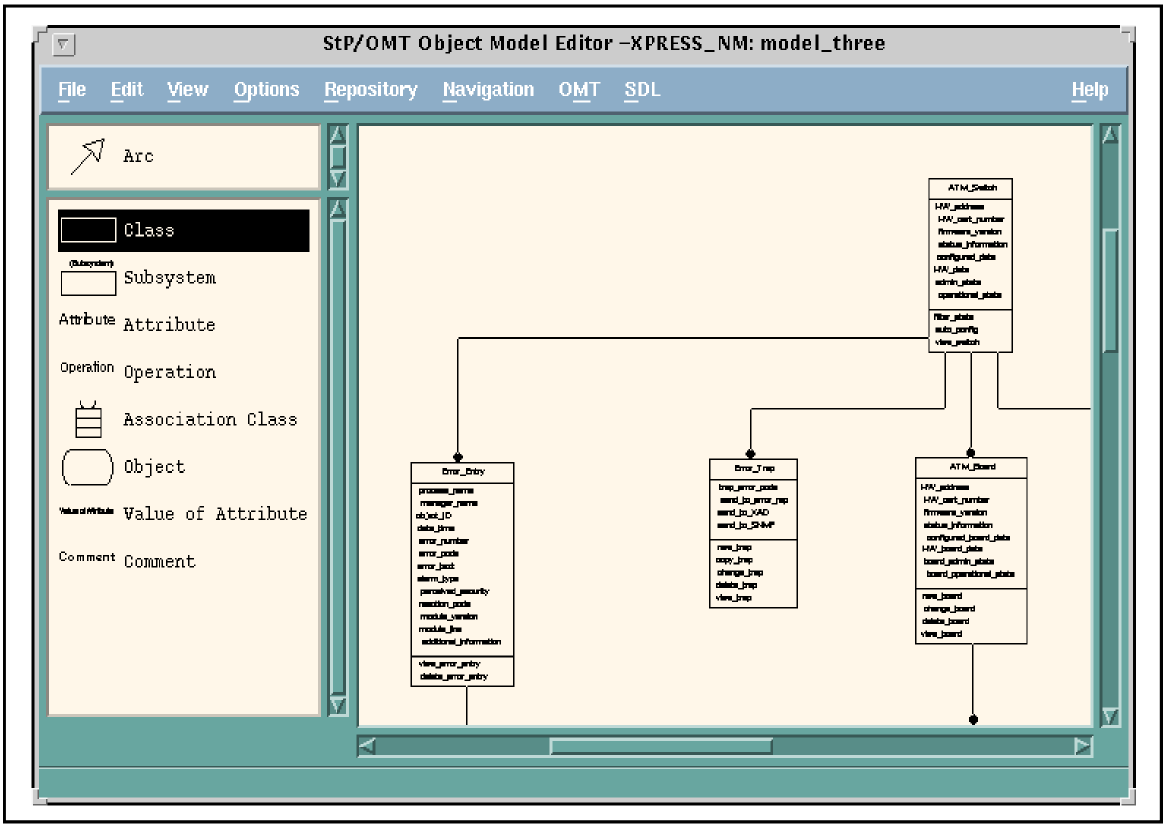


Abbildung 4-3: StP Object Model Editor

Dieser Editor ist genauso aufgebaut wie alle anderen zur Verfügung stehenden Editoren. Auf der linken Seite befindet sich eine Objektliste, die alle verwendbaren Symbole darstellt. Das Fenster auf der rechten Seite dient zur Erstellung des Diagramms.

Mit diesem Editor hat man die Möglichkeit, Klassen zu definieren, die dazugehörigen Attribute und Operationen zu ergänzen und vor allem die Beziehungen zwischen den einzelnen Klassen darzustellen.

Neben diesem Objektmodelleditor gibt es bei StP noch weitere Editoren, die im folgenden kurz aufgelistet sind:

- Use Case Editor:
  Darstellung von *Use Cases*, Ereignisfluß-, und Ereignisverfolgungsdiagrammen
- Dynamikmodelleditor:
  Beschreibung des dynamischen Verhaltens von Klassen
- Objektinteraktionseditor:
  Zeigt die Interaktionen auf, die zwischen den einzelnen Objekten bestehen; synchronisiert sich mit dem Objektmodelleditor
- Funktionsmodelleditor:
  Beschreibt die Transformation von Datenwerten
- Klassentabelleneditor:
  Hiermit werden die Attribute, Operationen und weitere spezielle Einstellungen für die Generierung von Code festgelegt
- Zustandstabelleneditor:
  Stellt die gleichen Ergebnisse wie der Dynamikmodelleditor tabellarisch dar.

4.2 Erfassen der vorhandenen Funktionalität von der XAD

Um überhaupt ein Objektmodell erstellen zu können, muß man zuerst versuchen, die komplette Funktionalität des zu realisierenden Projekts zu erfassen. Die Basis ist in diesem Fall die *Hicom Xpress Administration* (XAD). Somit besteht die erste Aufgabe darin, die jeweiligen Klassen zu identifizieren.
Betrachtet man die Bedienoberfläche der XAD, so kann man an diesem Aufbau sehr gut erkennen, welche Klassen notwendig sind. Um im weiteren Verlauf die Übersichtlichkeit nicht zu gefährden, wird eine Unterscheidung hinsichtlich der einzelnen Aufgaben wie Installation, Konfiguration, Fehlermanagement usw. gemacht.

4.2.1 Konfiguration und Installation mit der XAD

Bei der Konfiguration des ATM-Switches unterteilt man das Konfigurationsmanagement (siehe Kapitel 3.2) in Erstkonfiguration und Änderungskonfiguration, die sowohl auf den gesamten ATM-Switch als auch auf einzelne Boards angewendet werden können. Die Erstkonfiguration wird durch die "AutoConfig-Funktion" unterstützt. Sie überprüft die Hardware im Switch oder einzelne Boards und übernimmt automatisch die dabei ermittelte Konfiguration. Wird ein Board mit dieser Funktion aufgerufen, so werden automatisch auch die zugehörigen Ports konfiguriert. Bei der Änderungskonfiguration kann z.B. manuell ein neues Board angelegt werden. Zu beachten ist dabei, daß das ASN32-Board, das CGA2-Board und das L2M-Board fest vorkonfiguriert sind und somit nicht gelöscht oder verändert werden können.
Die gesamte Funktionalität für diesen Bereich ist in der XAD in sogenannten Objektgruppen zusammengefaßt. Diese Objektgruppen sind damit ohne Zweifel die richtige Wahl bei der Suche nach Klassen, da die Bezeichnung "Objektgruppe" auch allgemein der Terminologie einer Objektklasse entspricht. Zu dem Bereich Konfiguration und Installation gehören die folgenden Objektgruppen:

- ATM Switch
- ATM Device
- ATM User
- Partner Node
- Access
- Permanent Connections (PVC)
- Error Trap

Die **Objektgruppe 'ATM Switch'** dient der Verwaltung der gesamten Hardware des Switches. Darunter fällt das Festlegen und Anzeigen der Sollkonfiguration für die Boards und Ports, aber auch das Anzeigen der Istkonfiguration. Zusätzlich kann man einzelne Hardware-Einheiten wie Switch, Board und Port aktivieren oder deaktivieren. Ist eine Hardware-Einheit defekt, wird auch dies angezeigt.

Die **Objektgruppe 'ATM Device'** übernimmt die Verwaltungsaufgaben für die vorkommenden Devices. Unter einem Device versteht man ein Gerät, das einem ATM-Teilnehmer den Zugang in ein ATM-Netz ermöglicht. Die Adressierung wird über eine *Service Profile ID* realisiert. Die genauen Aufgaben sind die Konfiguration von Devices, die logische In- und Außerbetriebnahme von Devices, das Zuordnen von logischen Anschlüssen (Access), das Ändern der Zuordnung von logischen Anschlüssen, die Zuordnung eines ATM-Users zu einem ATM-Device und das Löschen von Devices. Beim Löschen von Devices ist zu beachten, daß es keine Beziehungen mehr zwischen dem Device und einem Access gibt, da die Aktion sonst nicht ausgeführt werden kann. Im Bearbeitungsfenster von ATM-Device werden folgende Attribute angezeigt:

- Device Name
- Device Typ
- Admin State (locked / unlocked)
- Access Name
- Assigned User
- Service Profile ID

Die **Objektgruppe 'ATM User'** verwaltet die einzelnen ATM-Teilnehmer, die über eine *Directory Number* (entspricht der Telefonnummer) identifiziert werden können. Darunter versteht man die Konfiguration von User-Einträgen, die Zuordnung zu Devices, die Änderung dieser Zuordnung und das Löschen von Devices. Im Bearbeitungsfenster von ATM-User wird der *User Name*, die *Phone Number*, der *Admin. State* und der *Device Name* angegeben.

Zur Verwaltung von anderen angeschlossenen Netzknoten gibt es die **Objektgruppe 'Partner Node'**. Die Adressierung der Nodes ist mit einer *Service Profile ID* realisiert, die vom System automatisch vergeben wird. Als Technik wird beim S2-Port *ISDN-Backboning* und *ISDN-Interworking* und beim STM1-Port *ATM-Backboning* unterstützt. Die Funktionalität, die bereitgestellt wird, ist der Objektgruppe 'ATM-User' bzw. 'ATM-Device' entsprechend. So kann ein Partner Node nur gelöscht werden, wenn es keinen Access zwischen dem Partner Node und einem Port gibt. Im Bearbeitungsfenster werden folgende Attribute dargestellt:

- Node Name
- Network Typ (B-ISDN / N-ISDN)
- Admin State (locked / unlocked)
- Access Name
- Service Profile ID

Um logische Anschlüsse von Devices oder Partner-Nodes an das Netz, die unabhängig vom physikalischen Netz sind, zu verwalten, gibt es die **Objektgruppe 'Access'**. Sie konfiguriert oder löscht die logischen Anschlüsse und ordnet sie den Devices und Partner Nodes zu. Bei einem Access kann man zwischen drei Typen unterscheiden: Es gibt einen *exclusive access*, der die Ressourcen des Ports exklusiv nutzt. Demgegenüber steht der *shared access*, der sich die vorhanden Ressourcen mit anderen logischen Anschlüssen aufteilen muß. Der letzte Typ ist der *MC loop access*, der nur für spezielle *Multicast-Loops* eingesetzt wird.
Das Bearbeitungsfenster stellt folgende Informationen zur Verfügung:

- Access Name
- Port Address
- Port Type
- Port Operational
- Equipment Name (Device / Partner Node)

Zwischen den einzelnen logischen Anschlüssen kann man auch Festverbindungen einrichten, sogenannte PVCs, die über die **Objektgruppe 'PVC'** verwaltet werden. Bei einem PVC kann man zwischen einer Punkt-zu-Punkt und einer Punkt-zu-Mehrpunkt Verbindung unterscheiden. Definiert man *Singlecast-Verbindungen*, so werden ATM-Channels, B-Channels und ATM-Paths benötigt. Zu beachten ist jedoch, daß die B-Channels in der vorliegenden Version der XAD nicht verändert werden können. Um die Funktionalität trotzdem nicht zu gefährden, sind alle 32 möglichen Kanäle immer durchgeschaltet. Bei *Multicast-Verbindungen* werden nur Informationen über die ATM-Channels gebraucht. Neben dieser Aufgabe der Definition von PVCs gibt es noch das Löschen von PVCs, das Anzeigen der aktuellen Schaltzustände der PVCs, die logische In- und Außerbetriebnahme von PVCs und schließlich noch das Hinzufügen und Löschen von Ästen bei Multicast-PVCs.
Im Bearbeitungfenster erscheinen folgende Werte:

- PVC Name
- PVC Typ
- Multicast-ID
- Requ. State (in service / out of service)
- Actual State (in service / out of service)
- Access-A Name
- Access-B Name

Die letzte **Objektgruppe** in diesem Bereich ist **'Error Trap'**, die die Bedingungen für die Error Reports festlegt. Dabei definiert man zuerst die einzelnen Traps, d.h. man legt die Fehlercodes fest, auf die reagiert werden soll. Hat man dies gemacht, müssen noch die Aktionen definiert werden, die ausgeführt werden sollen, wenn ein festgelegter Fehlercode auftritt. Beispiele sind das Weiterleiten an SNMP oder an die XAD zur Anzeige im Error Report. Die eingestellten Werte können in dieser Gruppe angezeigt und gelöscht werden.
Betrachtet man die behandelten Objektgruppen nochmals rückblickend, so kann man die Objektgruppen 'ATM Switch' und 'Error Trap' für die Konfiguration des Switches einsetzen. Die anderen fünf Objektgruppen dienen der Installation des Switches. Weitere Informationen bezüglich einzelner Variablen in den jeweiligen Objektgruppen sind im Kapitel 4.3 enthalten.

4.2.2 Fault Management mit der XAD

Beim Fehlermanagement unterscheidet man zwischen drei Objektgruppen:

- ErrorLog
- ErrorRep
- ErrorTrap

Mit der **Objektgruppe 'ErrorLog'** wird der Fehlerspeicher des Hicom Xpress verwaltet, d.h. hier werden alle Ablaufbesonderheiten, die im System auftreten, gesammelt. Diese kann man sich entweder anzeigen lassen oder löschen. Um die Übersichtlichkeit bei der Fehlersuche zu erhöhen, kann man die gespeicherten Fehlermeldungen mit selber definierten Filtern durchsuchen.
Wichtig bei dieser Objektgruppe ist noch, daß die Anzeige der Fehlermeldungen nicht automatisch während der Sitzung aktualisiert wird. Somit muß eine Refresh-Aktion angestoßen werden, um wieder einen aktuellen Auszug aus dem Fehlerspeicher zu erhalten.

Die **Objektgruppe 'ErrorRep'** stellt, wie 'ErrorLog', die Möglichkeit bereit, eintreffende Traps und Ablaufbesonderheiten zu analysieren. Zu beachten ist dabei, daß nur die Fehlermeldungen angezeigt werden, die während der aktuellen Sitzung auftreten. Möchte man dagegen Fehlermeldungen von älteren Sitzungen erhalten, muß man in die Objektgruppe 'ErrorLog' wechseln. Denn nur dort werden alle Fehlermeldungen gespeichert. Somit stehen im 'ErrorRep' zwar nur eine begrenzte Anzahl von Fehlermeldungen zur Verfügung, aber man ist als Systemadministrator in der Lage, aktuell auftretende Meldungen sofort zu bearbeiten, da alte Meldungen nicht angezeigt werden.

Dies wird auch dadurch unterstützt, daß man in der Objektgruppe 'ErrorRep' im Gegensatz zu 'ErrorLog' die Möglichkeit hat, einzelne und schon bearbeitete Fehlermeldungen zu löschen. Dies erleichtert die Arbeit des Administrators, da er wirklich nur aktuelle Fehlermeldungen auf dem Bildschirm hat.
Sobald er einen Fehler behoben hat, kann er diesen aus dem 'ErrorRep' löschen und sich auf den nächsten Fehler konzentrieren. Zu beachten ist dabei, daß die Fehlermeldung nur im 'ErrorRep', aber nicht im 'ErrorLog' gelöscht werden, d.h. sollten wieder Probleme auftreten, kann man auf diese Informationen zurückgreifen.
Das Bearbeitungsfenster sieht bei beiden Objektgruppen gleich aus und enthält folgende Attribute:

- Error No.
- Date
- Error Code
- Alarm Type (Communication / QoS / Processing / Equipment / Environment)
- Perceived Severity (Cleared / Indeterminate / Critical / Major / Minor / Warning)
- Reaction (Signal Error / Shutdown)

Die letzte **Objektgruppe** in diesem Umfeld ist **'ErrorTrap'** und bedarf hier keiner weiteren Erläuterungen, da sie im vorherigen Abschnitt schon vorgestellt wurde.

4.2.3 Security Management mit der XAD

Im letzten Bereich gibt es nur eine einzige **Objektgruppe** und zwar die Gruppe **'Authent'**. Darin werden die Zugangsberechtigungen für die XAD verwaltet. Dazu kann man die einzelnen Administrator-Identifikationen eintragen oder löschen. Mit den jeweiligen Kennungen kann man dann den Administratoren jeweils die Zugangsebenen *Installation*, *Konfiguration*, *Fault Management* und *Supervisor* zuteilen oder sperren. Diese Information wird auch im Bearbeitungsfenster zur Verfügung gestellt.
An dieser Objektgruppe sieht man sehr schnell, daß bezüglich Sicherheit bei der XAD nicht sehr viel Funktionalität zur Verfügung steht. Dies sollte aber bei den weiteren Betrachtungen nicht vergessen werden, da durch die Öffnung der Netze die Sicherheitsaspekte verstärkt berücksichtigt werden müssen. Denn wird dieser Punkt zu stark vernachlässigt, besteht die Gefahr, daß unberechtigte Personen auf Daten oder Komponenten zugreifen können.

4.3 Erstes Objektmodell auf Basis des Xpress-Datenmodells

Für die Entwicklung eines ersten Objektmodells ist es notwendig, die einzelnen Klassen festzulegen, Attribute und Methoden den einzelnen Klassen zuzuordnen und grundlegende Beziehungen zwischen diesen zu modellieren. Der erste Schritt zur Identifizierung wurde schon im vorherigen Kapitel getan, da dort für die unterschiedlichen Managementaufgaben Objektgruppen definiert wurden. Aus jeder dieser Objektgruppen wird nun im folgenden eine eigene Objektklasse erstellt. Um eine Klasse zu erstellen, wird aus der Objektliste des OME (siehe Kapitel 4.1.2) das Klassensymbol in den Arbeitsbereich gezogen und der Klassenname ergänzt. Damit erhält man im ersten Schritt die folgenden zehn verschiedenen Klassen, die jeweils nach einer Objektgruppe der XAD benannt sind:

- `ATM_Switch`
- `Access`
- `PVC`
- `Partner_Node`
- `ATM_Device`
- `ATM_User`
- `Error_Trap`
- `Error_Log`
- `Error_Rep`
- `Authent`

Zusätzlich zu diesen zehn Klassen werden noch zwei weitere neue Klassen eingeführt. Es ist auf der einen Seite die Klasse `Routing_Table`, die im Zusammenhang mit den PVCs vorkommt und die Klasse `ATM_Port`, die neben den gleichen Informationen aus der Klasse `ATM_Switch` noch portspezifische Attribute aufweist.

Daran anschließend müssen die jeweiligen Attribute und Methoden bei den einzelnen Klassen ergänzt werden. Dazu wird aus der Objektliste ein Attribut- oder Methodensymbol in die zu bearbeitende Klasse gezogen und dort der Attributs- oder Operationsname ergänzt. Um für die jeweiligen Klassen die Attribute und Operationen zu finden, wird in der XAD die gesuchte Objektgruppe, wie z.B. ‘ATM_Switch’ aufgerufen. Dazu wird mit dem Button “ATM Switch” im Hauptfenster der XAD das *Processing Window* (PW) ATM_Switch (siehe Abbildung 3-4) geöffnet. Um nun weitere Informationen für den Switch oder die Boards zu bekommen, wird für einen Eintrag im PW das *View Window* (VW) geöffnet. Die dort angezeigten Informationen werden teilweise zusammengefaßt oder direkt als Attribute übernommen.

Auch die Operationen werden aus dem PW und VW extrahiert und als Methoden in den jeweiligen Klassen ergänzt. Im Fall der **Klasse `ATM_Switch`** entstehen so folgende Attribute und Methoden:

Attribute:

- HW_address
- HW_part_number
- firmware_version
- status_information
- configured_data
- HW_data
- switch_admin_state
- operational_state

Methoden:

- new_board
- change_board
- delete_board
- filter_state
- auto_config
- view_switch
- view_board

Bei den Attributen configured_data und HW_data der Klasse `ATM_Switch` ist zu beachten, daß zu diesem Zeitpunkt mehrere einzelne Attribute zusammengefaßt sind. Ob dies aber bei der weiteren Entwicklung so bestehen bleibt, ist noch nicht sicher. Im Moment dient es nur der Übersichtlichkeit des Modells, da weniger Attribute in den einzelnen Klassen vorkommen.

Bei der **Objektklasse Access** fällt auf, daß es keine Operation change_access gibt. Dies liegt daran, daß dies vom System nicht unterstützt wird. Um einen logischen Anschluß trotzdem zu verändern, muß man ihn erst löschen und dann neu anlegen.

Hat man die Zusammenstellung der Attribute und Methoden für alle Klassen durchgeführt, besteht die letzte Aufgabe darin, die Beziehungen zwischen den einzelnen Klassen zu modellieren. Dabei wird zunächst weder auf Vererbung, Aggregation oder Multiplizitäten Rücksicht genommen. Als Grundlage der Modellierung wird dabei das Datenmodell des Hicom Xpress (siehe Abbildung 4-4) herangezogen, da dort die jeweiligen Beziehungen gut zu erkennen sind.

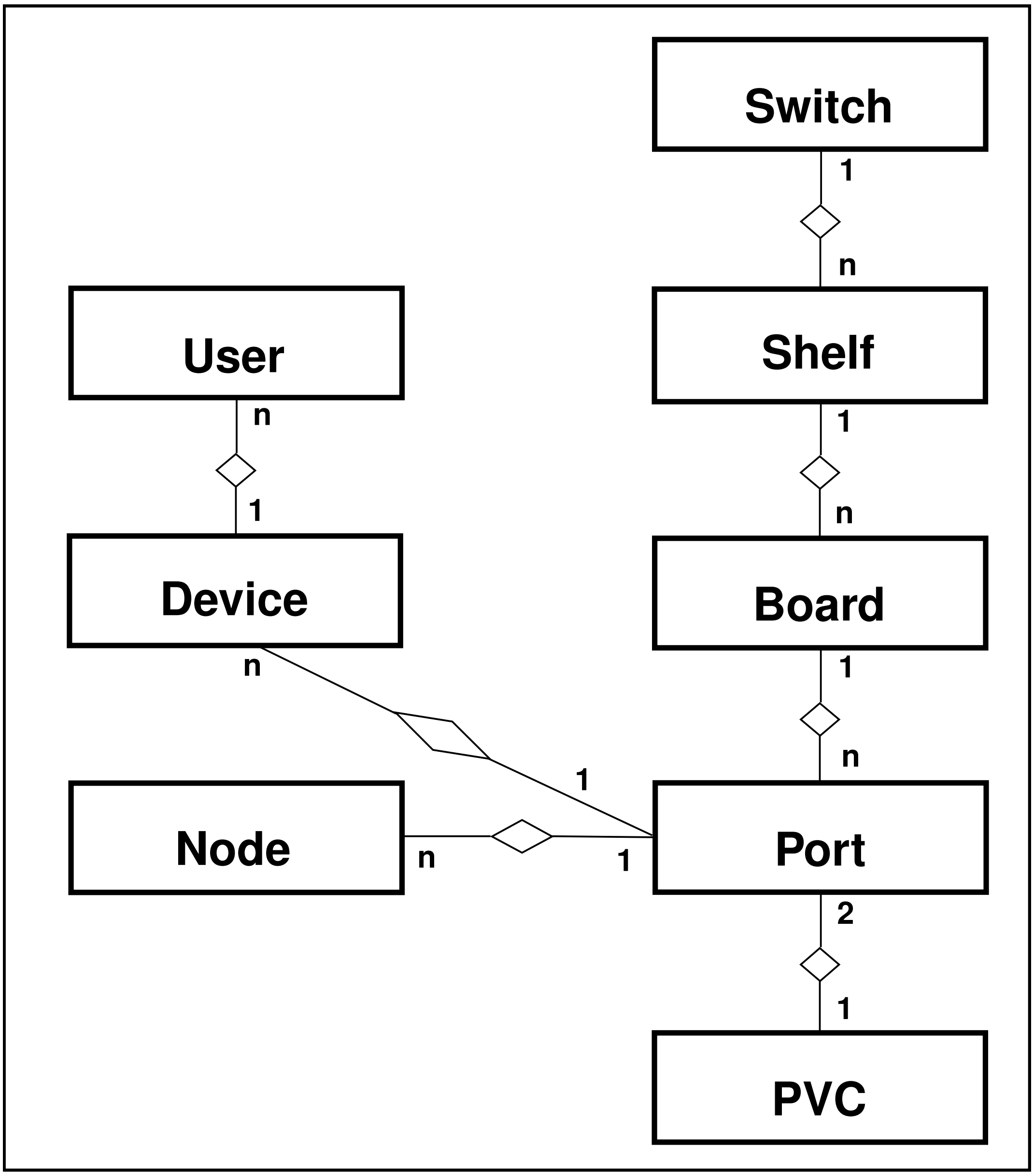


Abbildung 4-4: Datenmodell des Hicom Xpress

Nach Abschluß dieser Tätigkeit ist ein erstes Objektmodell fertiggestellt, das in Abbildung 4-5 vollständig dargestellt ist.

Wie man dort sehr gut erkennen kann, fehlen noch alle Beziehungstypen. Außerdem sind manche Klassen zu groß bzw. geben die Struktur des Hicom Xpress nicht genau und übersichtlich wieder. Um dies zu beheben, wird im folgenden eine zweite Version des Objektmodells entwickelt, das in der Abbildung 4-6 vollständig dargestellt ist.

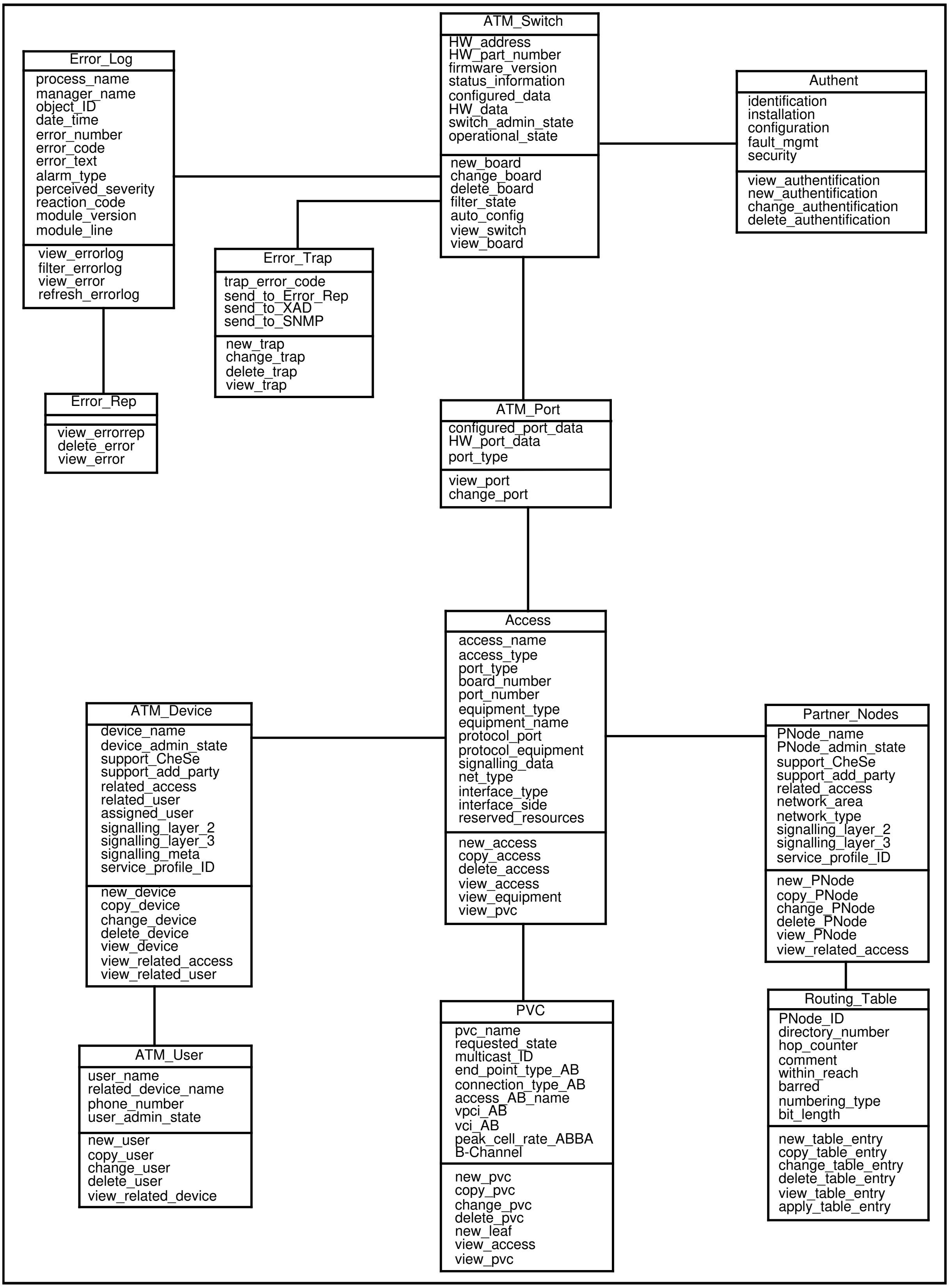


Abbildung 4-5: Objektmodell für ATM-Switches Version 1

4.4 Einbindung der Bottom-up-Analyse in das Objektmodell

In diesem Kapitel wird nun aus dem entstanden ersten Objektmodell ein zweites Objektmodell entwickelt. In dieses Modell fließen die Ergebnisse aus Kapitel 4.2 ein. Dort wurde die Funktionalität der XAD mit einer Bottom-up-Analyse erfaßt und steht nun für die Modellierung des Objektmodells zur Verfügung. Um die Übersichtlichkeit zu steigern, wird bei der Modellierung des Objektmodells die gleiche Reihenfolge wie im Kapitel 4.2 verwendet. Dadurch entsteht ein neues Objektmodell, daß die Beziehungen zwischen den einzelnen Klassen wiedergibt und somit die Struktur des Xpress verdeutlicht.

4.4.1 Konfigurations- und Installationsfunktionalität des Switches

Beginnt man bei der **Klasse `ATM_Switch`** (siehe Abbildung 4-5), so stellt man bei näherer Betrachtung fest, daß in dieser Klasse Informationen über den Switch, die Shelfs und die Boards enthalten sind. Unter einem Shelf versteht man in diesem Zusammenhang einen Baugruppenrahmen, in dem mehrere Baugruppen (Boards) enthalten sein können.
Aus einem Shelf und der dazugehörigen Software, um die Signalisierung bis zur Schicht 2 durchzuführen, besteht dann der *Switching Block* des Hicom Xpress. Zur besseren Darstellung dieser Struktur ist eine Aufteilung in mehrere Klassen wünschenswert. Somit wird die ursprüngliche Klasse `ATM_Switch` zerlegt und es entsteht zusätzlich die neue Klasse `ATM_Board`. Die Klasse `ATM_Shelf` entsteht nicht, da die Funktionalität bezüglich eines Shelf nicht mehr im Hicom Xpress unterstützt wird. Wichtig dabei ist aber, daß die Datenstruktur der Hardwarehierarchienummer nicht verändert wird, d.h. im Attribut HW_address wird weiterhin im zweiten Oktett eine Eins für einen Shelf angegeben. Eine Anpassung dieser Datenstruktur an die neue Umgebung ist jedoch von Siemens nicht geplant. Am Ende dieser Aktion liegen nun drei Klassen vor: `ATM_Switch`, `ATM_Board` und `ATM_Port`, die bei den meisten Attributen sogar übereinstimmen und den physikalischen Aufbau des Switches gut beschreiben. Die Methoden dieser Klassen sind dagegen doch etwas klassenspezifischer, was zu einem späteren Zeitpunkt der Modellierung noch zu beachten ist.
Betrachtet man bei diesen Klassen auch gleich die jeweiligen Assoziationen, so stellt man sehr leicht folgendes fest: Ein ATM-Switch kann mehrere Boards enthalten und ein Board selber wiederum mehrere Ports. Dieser Zusammenhang wird durch das Symbol für Aggregationen im OMT-Modell dargestellt. (siehe Abbildung 4-6).
Im nächsten Schritt wird die **Klasse `Access`** in die Klasse `logical_port` umbenannt, da mit diesem Namen die Funktionalität der Klasse besser ausgedrückt wird. Dementsprechend werden die jeweiligen Attributs- und Operationsnamen im Objektmodell angepaßt.

Die Beziehung zwischen ATM_Port und logical_port ist eine 1:m-Beziehung, d.h. ein physikalischer Port kann mehrere logische Ports besitzen, aber nicht umgekehrt ein logischer Port mehrere physikalische Ports.
In den **Klassen ATM_Device** und **Partner_Node** wird das Attribut support_CheSe entfernt, da auch diese Funktion zukünftig nicht mehr unterstützt wird. Sie war dafür zuständig, die Arbeit zwischen einem Chef und seiner Sekretärin zu unterstützen. Es werden nur die jeweiligen Beziehungen der einzelnen Klassen modelliert. So besteht zwischen logical_port und ATM_Device eine 1:1-Beziehung, da ein Device nur einem logischen Port zugeordnet sein kann und umgekehrt genauso. Auch zwischen logical_port und Partner_Nodes besteht diese Beziehung. Die Beziehung zwischen logical_port und PVC ist dagegen nicht so einfach.
Betrachtet man einen PVC, so sind genau zwei logische Ports einem PVC zugeordnet. Eine andere Möglichkeit besteht dabei nicht. Dreht man die Blickrichtung jedoch um, so stellt man fest, daß einem logischen Port mehrere PVCs zugewiesen sein können. Dies ist im Objektmodell (Abbildung 4-6) durch eine zusätzliche “2” neben dem ausgefüllten Punkt bei der Klasse PVC graphisch dargestellt.
Betrachtet man nun die **Klasse Routing_Table**, so findet man ein Attribut numbering_type, das den Typ der jeweiligen Telefonnummer angibt. Man unterscheidet dabei zwischen drei Typen:

- E.164
- OSI-NSAP-DCC
- OSI-NSAP-ICD

Diese Typen stellen grundsätzlich internationale Adressen dar. Unterschiede finden sich in den Dateiformaten der Typen. Die Typen *NSAP_DCC* und *NSAP_ICD* haben den gleichen Aufbau, bis auf das Adreßfeld. Bei DCC wird die Adresse in Hex-Darstellung und bei ICD in ASCII-Darstellung formuliert. Bei einem Vergleich mit E.164 stellt man fest, daß die Telefonnummer auch in ASCII dargestellt wird, aber im Vergleich zu *NSAP_DCC* und *NSAP_ICD* gibt es kein Feld über die Länge der ATM-Adresse. Dagegen gibt es ein Feld “Private”, das angibt, ob das Ziel innerhalb des privaten oder des öffentlichen Netzes erreichbar ist. Dieses Feld wird bei der XAD durch das Attribut within_reach abgebildet. Auswirkungen auf das Objektmodell sind in dieser Version noch nicht realisiert, aber diese Aufteilung sollte im weiteren nicht unberücksichtigt bleiben.
Die Beziehungen zwischen der Klasse Routing_Table und der Klasse Partner_Node sind jeweils m:n-Beziehungen. Grund dafür ist, daß jeder Partner-Node in mehreren Routingtabellen auftreten kann, aber andererseits auch jeder Routingtabelleneintrag in mehreren PartnerNodes vorkommen kann.

4.4.2 Funktionalität des Fehler- und Securitymanagements

Auch die Klassen des Fehlermanagements müssen weiter umgestellt werden, da die **Klassen Error_Log** und **Error_Rep** grundsätzlich die gleichen Informationen enthalten. Um dies zu verdeutlichen, wird eine neue Klasse `Error_Entry` eingeführt, die alle Informationen für die jeweiligen Fehlermeldungen enthält. Dadurch müssen die Attribute nur einmal bestimmt werden und können dann an die beiden Klassen weitergegeben werden.

Der Unterschied zwischen `Error_Rep` und `Error_Log` liegt darin, daß im `Error_Rep` nur die Fehlermeldungen angezeigt werden, die während der Sitzung auftreten. Möchte man dagegen alle Fehlermeldungen anschauen, muß die Klasse `Error_Log` verwendet werden. Dabei ist aber zu beachten, daß ein automatisches Aktualisieren während der Sitzung nicht durchgeführt wird.

Die Beziehungen zwischen diesen Klassen sind damit auch klar, da sowohl `Error_Log` als auch `Error_Rep` mehrere Objekte der Klasse `Error_Entries` enthalten kann.

Im Gegensatz dazu sind zwischen `ATM_Switch` und `Error_Log` bzw. `Error_Rep` verständlicherweise nur 1:1-Beziehungen realisiert, da es für jeden Switch z.B. nur einen ErrorReport geben kann.

Um diese Klassen nun mit Informationen zu versorgen gibt es die **Klasse Error_Trap**. Zwischen der Klasse `ATM_Switch` und `Error_Trap` besteht eine 1:m-Beziehung, da ein Switch für jeden Fehlercode einen Trap generiert. In der Klasse `Error_Trap` werden somit nicht die jeweiligen Traps generiert, sondern es wird in dieser Klasse festgelegt, welcher Agent einen Trap erhält. Ist es z.B. die XAD, so wird der Fehler im ErrorReport angezeigt. Diese Festlegung wird dabei für jeden vorhandenen Fehlercode durchgeführt.

Veränderungen an den Attributen werden wie bei der Klasse `Authent`, die eine 1:1-Beziehung mit der Klasse `ATM_Switch` hat, im Vergleich mit Version 1 nicht gemacht.

Betrachtet man das zweite fertiggestellte Objektmodell (siehe Abbildung 4-6) genauer, so stellt man fest, daß in einigen Klassen die gleichen Attribute oder auch Methoden enthalten sind. Ebenso gibt es noch in manchen Klassen Methoden, die schon in anderen Klassen realisiert sind. Hier ist zu überprüfen, ob es nicht besser ist, die jeweiligen Methoden zu streichen, um eine Mehrdeutigkeit und einen höheren Implementierungsaufwand zu vermeiden.

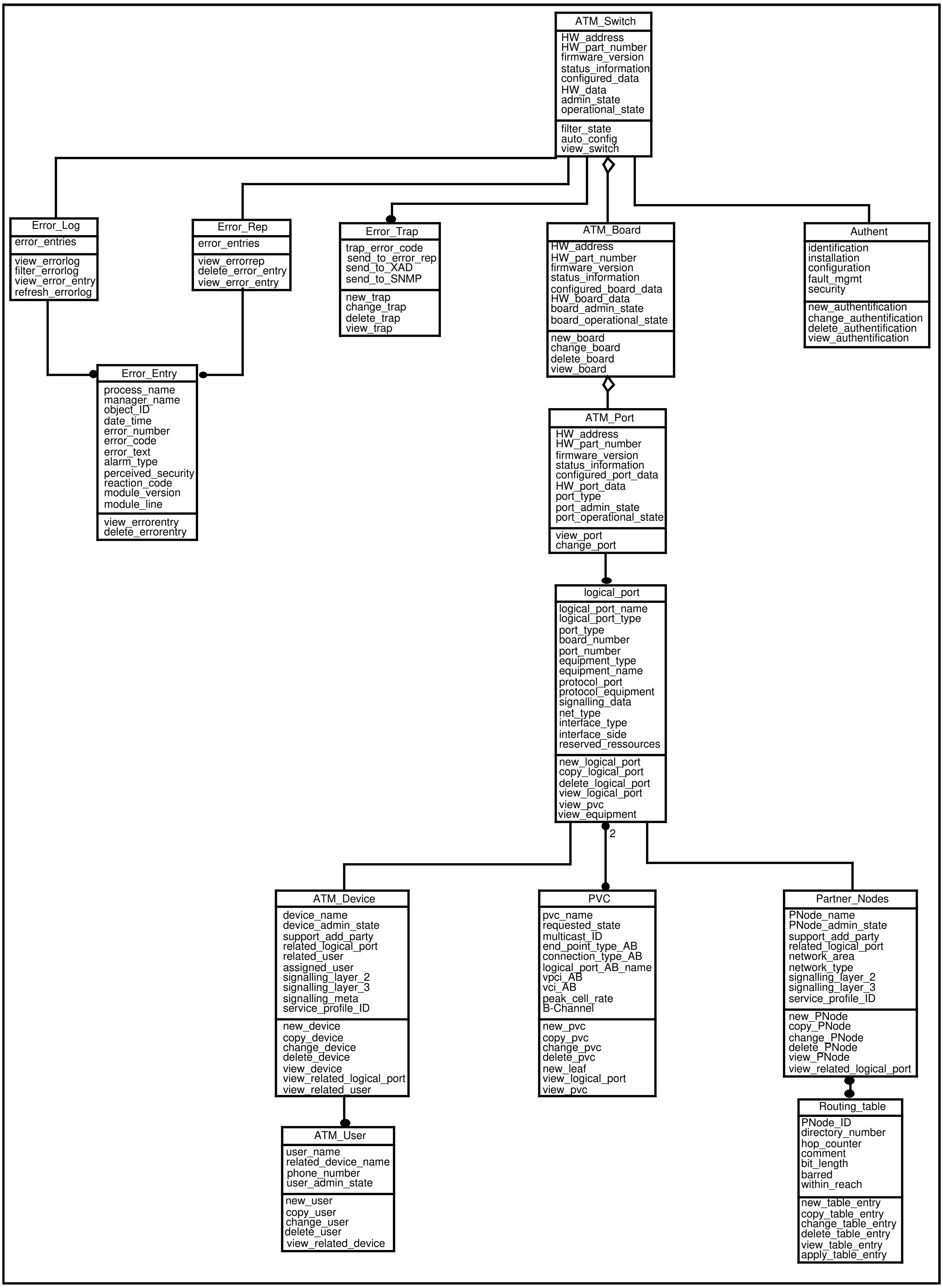


Abbildung 4-6: Objektmodell für ATM-Switches Version 2

4.5 Abschließende Überarbeitung des Objektmodells

Unter Berücksichtigung der Überlegungen am Ende des Kapitels 4.4 wird aus dem zweiten Objektmodell (siehe Abbildung 4-6) ein weiteres Objektmodell entwickelt. Dies soll aber jetzt so modifiziert werden, daß am Ende ein Objektmodell zur Verfügung steht, das die Funktionalität der XAD übersichtlich und vollständig darstellt.

Greift man als Erstes den Gesichtspunkt auf, der bei der Entwicklung des zweiten Modells am Schluß stand, so findet man einige Operationen, die in mehreren Klassen realisiert sind. Ein Beispiel ist die Operation view_related_logical_port, die sowohl in der Klasse `ATM_Device`, als auch in der Klasse `Partner_Node` auftritt. Diese wird gestrichen, da diese Methode schon in der Klasse `logical_port` durch view_logical_port realisiert ist und die nötigen Informationen in den jeweiligen Klassen zur Verfügung stellen kann. Auch die Operation view_equipment der Klasse `logical_port` wird entfernt, da man unter *Equipment* entweder ein Device oder einen PartnerNode verstehen kann, d. h. view_equipment heißt entweder view_device oder view_PNode, und diese sind schon in ihren jeweiligen Klassen vorhanden. Die beiden letzten Methoden, die noch aus dem Modell herausfallen, sind filter_state der Klasse `ATM_Switch` und filter_errorlog aus `Error_Log`. Der Grund liegt darin, daß dies Methoden sind, die nur eine Filterfunktion in einem speziellen Fenster darstellen, d.h. sie haben keinen Einfluß auf das System und dienen nur zur Verdeutlichung der Darstellung.

Es werden aber nicht nur Operationen gestrichen, sondern auch neue eingeführt. So wird in der Klasse `logical_port` die bisherige Operation new_logical_port in drei Operationen umgewandelt.

Dies ist notwendig, da drei unterschiedliche Typen von logischen Ports zur Verfügung stehen, die auch jeweils unterschiedliche Aufrufparameter besitzen.

Um dies zu verdeutlichen gibt es nun die Operationen:

- new_logical _port_exclusive
- new_logical_port_shared
- new_logical_port_mcloop

Betrachtet man im folgenden die vorher definierte Klasse bezüglich der Routingtabelle (siehe Abbildung 4-6), so stellt man sehr schnell fest, daß in diesem Bereich noch Handlungsbedarf besteht.

Wie im obigen Teil schon erläutert, kann man drei Typen von Telefonnummern unterscheiden. Um dies deutlich zu machen, ist es sinnvoll, für jeden Typ eine eigene Klasse zu bilden, die jeweils von einer Basisklasse erbt.

Diese Basisklasse wird im folgenden `Routing_table` genannt. Von dieser Klasse erben die drei neuen Klassen alle Attribute und Methoden, die in allen drei Klassen vorkommen. Die Attribute, die unterschiedlich sind, werden in die jeweiligen Unterklassen eingefügt.

Dies sind beim Typ E_164 das Attribut within_reach und bei den anderen beiden Klassen das Attribut bit_length.

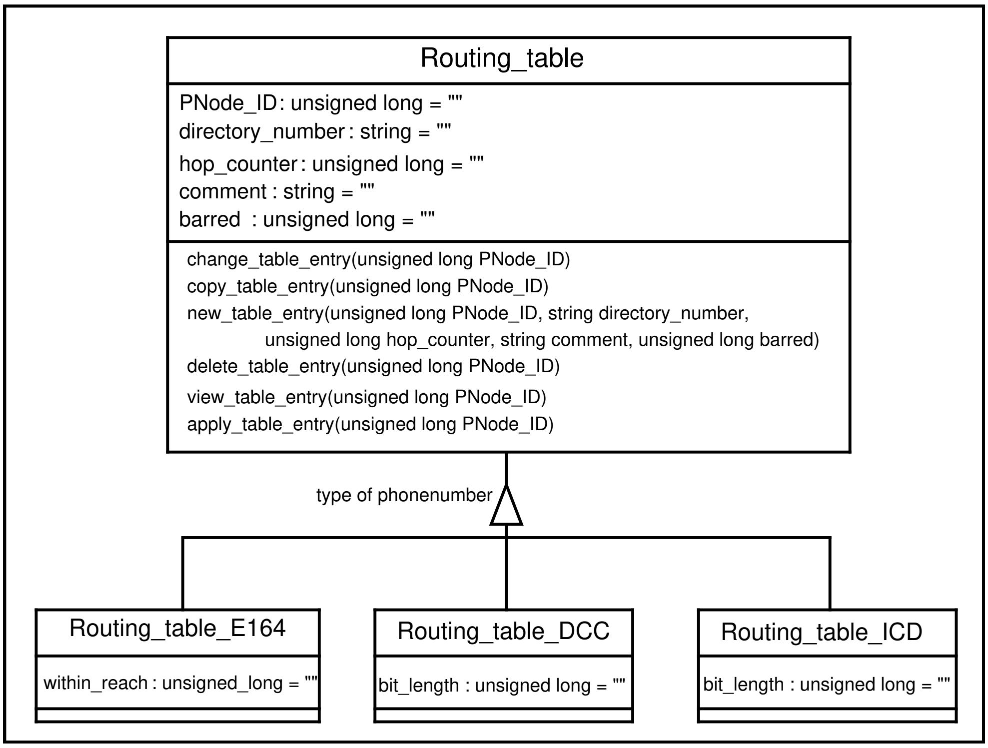


Abbildung 4-7: Neue Konstruktion der `Routing_table`

Zusätzlich wird noch ein Diskriminator eingeführt, der als weiteres Attribut zählt und damit die Bedingung für die getroffene Aufteilung angibt. In der Abbildung 4-7 ist der Ausschnitt dargestellt, der im vorliegenden Objektmodell enthalten ist.

Neben dieser Veränderung gibt es in der Version 3 des Objektmodells noch zwei weitere wichtige Änderungen.
Die erste betrifft die Klassen `ATM_Switch`, `ATM_Board` und `ATM_Port`. Betrachtet man die Klassen in Abbildung 4-8, so kann man auch hier sehr schnell erkennen, daß eine Basisklasse gebildet werden kann.
Dafür wird die **Klasse `HW_Object`** eingeführt, die die gemeinsamen Attribute übernimmt.

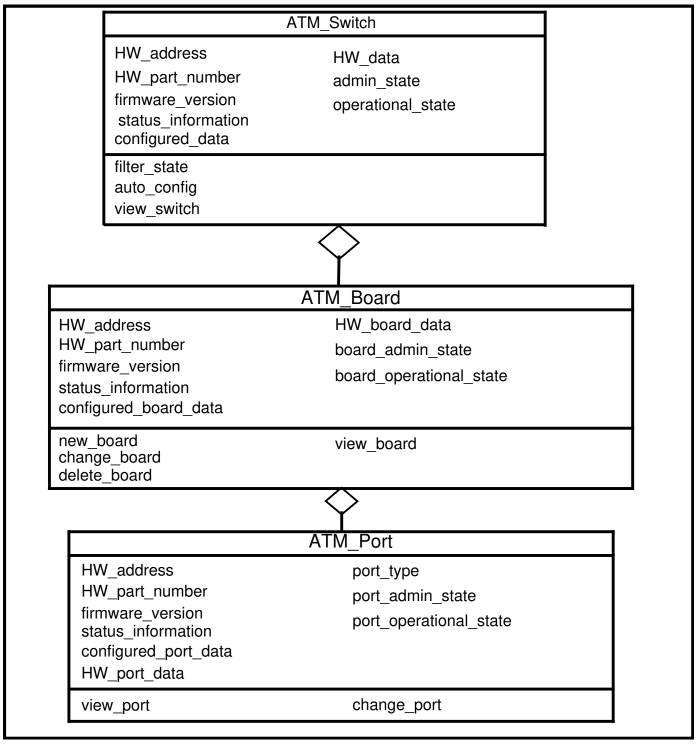


Abbildung 4-8: Auszug Objektmodell Version 2

Um die einzelnen Strukturen der Hardwarekomponenten jetzt noch besser darstellen zu können, wird in den Klassen `HW_Object` und `ATM_Port` die Zusammenfassung der Attribute aus der ersten Version des Objektmodells wieder aufgehoben und damit eine Vielzahl von neuen Attributen erstellt (siehe Abbildung 4-9). In diesem Zusammenhang werden mit dem gleichen Gedanken in der Klasse `logical_port` die zusammengefaßten Attribute aufgelöst und neue Attribute gebildet. In der Klasse `ATM_Port` ist noch zu beachten, daß die VCI-Werte des STM1-Ports den B-Kanälen des S2-Ports entsprechen. Damit ist zu erklären, wieso nur VCI-Werte bei den Attributen auftreten, obwohl auch S2-Ports eingerichtet werden können.

Bei den Operationen ist es dagegen nicht so einfach.
Grundsätzlich sind alle Operationen in der Basisklasse realisierbar, da die Funktionen bei allen Komponenten implementiert sind und sich nur die Parametrisierung für die jeweiligen Komponenten unterscheidet. Betrachtet man die Operationen view und change, so stellt man fest, daß diese für alle drei Klassen anwendbar sind. Bei den Operationen new und delete ist dies aber nicht möglich, da ein Switch bei der XAD nicht neu angelegt oder gelöscht werden darf, d.h. ruft man die XAD auf, der auf dem CB läuft, so ist grundsätzlich ein Switch vorhanden. Dies wird auch dadurch deutlich, daß der zu managende ATM-Switch aus einem SB und CB besteht.
Fehlt eine dieser Komponenten, besteht keine Möglichkeit zum Management des ATM-Switches.

Betrachtet man diese beiden Operationen in Bezug auf die Klasse `ATM_Port`, so ist auch hier zu sagen, daß diese Funktionen nicht realisierbar sind. Der Grund liegt darin, daß die Ports für ein Board automatisch angelegt werden, sobald dieses Board erzeugt wird.
Dieser Zusammenhang liegt auch bei der Operation delete vor, d.h. wird ein Board gelöscht, so werden auch die Ports gelöscht. Ein gesondertes Löschen von Ports ist in der XAD nicht möglich. Somit werden im folgenden nur die Operationen in die Basisklasse `HW_Object` aufgenommen, die sowohl für einen Switch, als auch für Boards und Ports sinnvoll sind. Dies sind die Methoden view und change. Die anderen Methoden werden weiterhin in den jeweiligen Klassen selber realisiert, da sonst Mißverständnisse bezüglich der Verwendbarkeit auftreten können. Eine Operation, die nur in der Klasse `ATM_Switch` auftritt, ist auto_config. Diese Funktion liefert automatisch die Konfiguration des Switches, wie sie zum Zeitpunkt des Aufrufs vorliegt. Die endgültige Realisierung dieser Klassen ist schließlich in Abbildung 4-9 dargestellt.

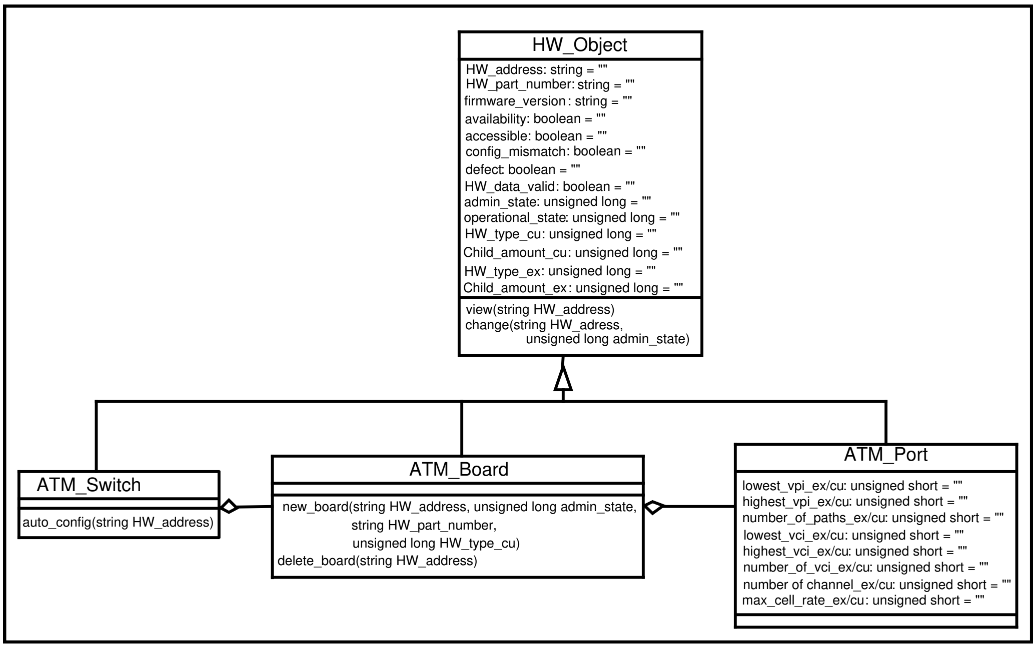


Abbildung 4-9: Neue Realisierung für `ATM_Switch`

Die letzte Veränderung zwischen Objektmodell Version 2 und Version 3 ist im Bereich des Fehlermanagements. Es betrifft neben den Klassen `Error_Log` und `Error_Rep` auch die Klasse `Error_Entry`. In der zweiten Version des Objektmodells wurden die Beziehungen (siehe Abbildung 4-6) noch nicht sehr deutlich herausgearbeitet. So wurde zwar schon herausgestellt, daß in `Error_Log` und in `Error_Rep` mehrere Objekte der Klasse `Error_Entry` enthalten sind. Diese Containment-Struktur war jedoch noch nicht sehr deutlich herausgearbeitet. Ebenso waren noch einige Operationen in den Klassen `Error_Log` und `Error_Rep` realisiert, die aber in der Klasse `Error_Entry` schon realisiert sind.

Somit bleiben nur die klassenspezifischen Funktionen in der jeweiligen Klasse bestehen. Das Ergebnis dieser Veränderungen ist in Abbildung 4-10 dargestellt.

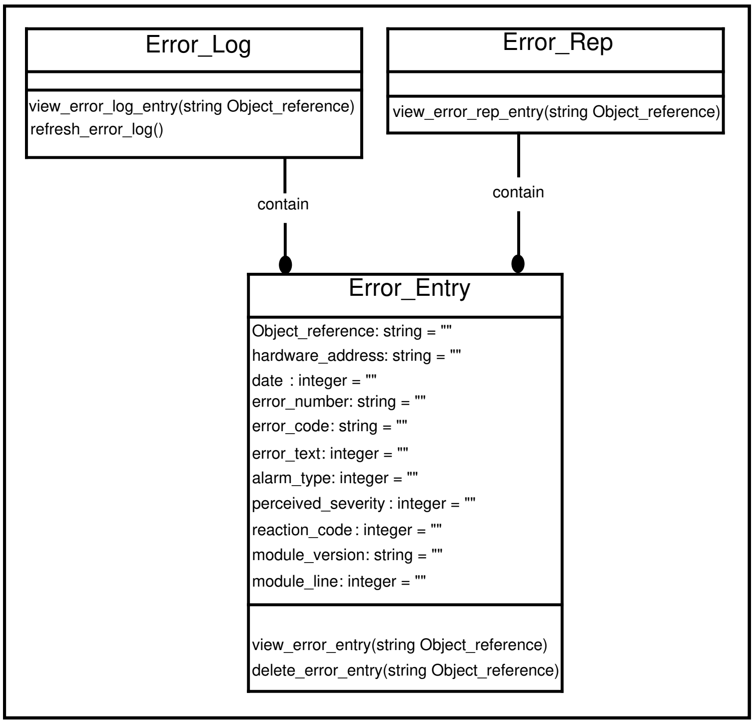


Abbildung 4-10: Veränderungen im Fehlermanagement

Vergleicht man nun abschließend das komplette Objektmodell in der Version 3 (siehe Abbildung 4-11) mit der Version 1 (Abbildung 4-5) oder Version 2 (Abbildung 4-6), so kann man durchaus größere Veränderungen erkennen. Bei dieser Betrachtung darf man nicht vergessen, daß das vorliegende Modell zum jetzigen Zeitpunkt nur für das proprietäre Tool XAD gilt. Versucht man dies zum Management eines anderen Switches zu verwenden, so wird dies mit großer Wahrscheinlichkeit nicht sofort und ohne Probleme funktionieren.
Deshalb soll nun im nächsten Abschnitt versucht werden, ein optimiertes Objektmodell zu entwerfen. Dafür werden zunächst Managementszenarien mit den sogenannten *Use Cases* definiert, die in das vorhandene Modell eingebunden werden.

Würde man diesen Schritt jedoch nicht vollziehen, müßte man jetzt mit dem fertigen Objektmodell (siehe Abbildung 4-11) noch etwas weiterarbeiten, da bisher für die einzelnen Attribute und Operationen noch keine Datentypen, Werte usw. definiert wurden. Dies wird mit dem *Class Table Editor* (CTE) gemacht, der für jede Klasse eine eigene Klassentabelle generiert, in die man die entsprechenden Werte usw. einsetzen kann. Danach ist man in der Lage, aus dem entstandenen Objektmodell den entsprechenden IDL-Code zu generieren, der dann bei der weiteren Implementierung des Objektmodells verwendet werden kann.

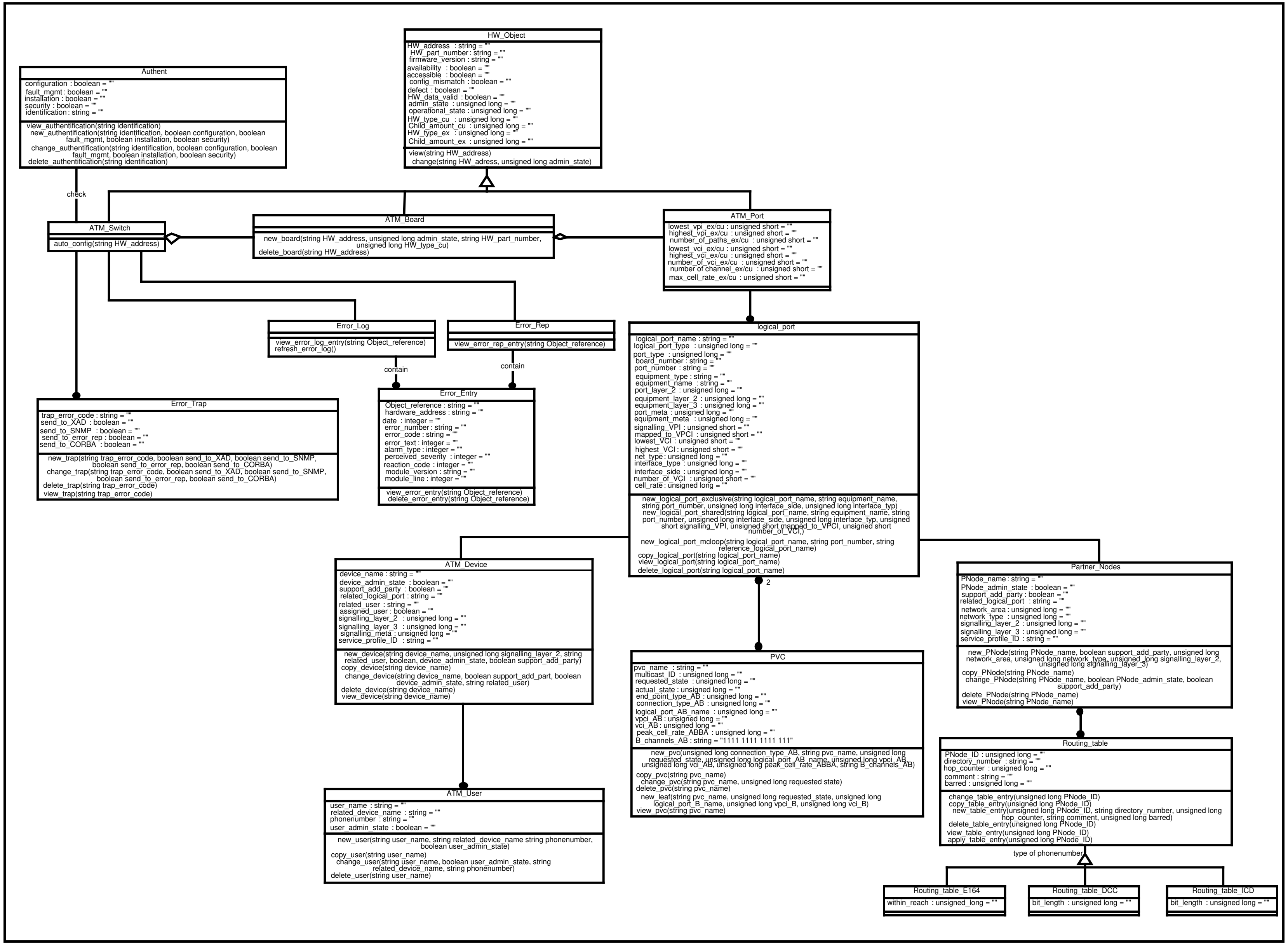


Abbildung4-11: Objektmodell für ATM-Switches Version 3

4.6 Entwicklung eines generischen Objektmodells

Am Ende dieses Kapitels soll die Grundlage dazu vorhanden sein, daß ein generisches Objektmodell für das Management eines ATM-Switches entwickelt werden kann. Um ein Verständnis für die *Use Cases* und den *Use Case Editor* von StP zu bekommen, werden sie im Kapitel 4.6.1 vorgestellt. Im Kapitel 4.6.2 werden dann mit diesen Hilfsmitteln die *Use Cases* für das Szenario entworfen.

4.6.1 Einführung in *Use Cases* und den *Use Case Editor* von StP

Während einer Systementwicklung werden eine Reihe von Modellen entwickelt, um Anforderungen, Beziehungen usw. darzustellen. Ein Modell in diesem Bereich ist das *Use Case Model (UCM)*, das ein Teil des *Requirement Model (RM)* ist. Im RM wird die gesamte Funktionalität des Systems spezifiziert und dies hauptsächlich von den *Use Cases* im UCM.
Das UCM verwendet dazu *Actors* und *Use Cases*. Die *Actors* bieten die Möglichkeit zu definieren, was außerhalb des Systems existiert. Die *Use Cases* dagegen legen fest, welche Aufgaben durch das System ausgeführt werden sollen.
Bei den *Actors* ist es wichtig, eine Abgrenzung zwischen ihm und dem Benutzer zu schaffen, da ein Benutzer das System nur mit der Rolle benutzt, wie sie vom *Actor* definiert wurde. Somit kann man den *Actor* als Klasse und die Benutzer als Instanzen dieser Klasse einordnen. Bei der Definition von *Actors* kann man zwischen menschlichen Anwendern und anderen Systemen, die z.B. mit unserem Switch kommunizieren, unterscheiden. Um bei der Modellierung die Systemstruktur nach Hauptfunktionalitäten unterscheiden zu können, wird zwischen *primary actor* und *secondary actor* unterschieden. Der *primary actor* benutzt dabei das System direkt, während der *secondary actor* das System überwacht und verwaltet, damit es der *primary actor* verwenden kann.
Wird ein System nun von einem Anwender (Actor), z.B. einem Systemadministrator, genutzt, so werden eine Reihe von Transaktionen zwischen dem Anwender und System ausgeführt. Diese Sequenz von Aktionen wird als *Use Case* bezeichnet. Damit ist jedes *Use Case* eine Möglichkeit, ein System zu benutzen. Wird ein *Use Case* ausgeführt, so wird dies als Instanz eines *Use Case*, das eine bestimmte Transaktion in Verbindung mit dem *Use Case* auslöst, betrachtet. Ist diese Ausführung abgeschlossen, wartet das *Use Case* auf weitere Eingaben. Um die Definition von *Use Cases* zu vereinfachen werden von [JAC92] folgende Fragen formuliert:

- Was sind die Hauptaufgaben für jeden *Actor*?
- Soll der *Actor* Systeminformationen lesen/schreiben/ändern können?
- Soll der *Actor* das System über Änderungen außerhalb des Systems informieren?
- Will der *Actor* über unerwartete Veränderungen informiert werden?

Hat man damit alle *Use Cases* gefunden, so hat man durch die *Actors* und *Use Cases* das System mit seiner vollen Funktionalität beschrieben. Zur Darstellung dieser einzelnen Informationen gibt es auch wieder mehrere Möglichkeiten. In der vorliegenden Arbeit wird der *Use Case Editor* (UCE) von StP verwendet, der im folgenden kurz vorgestellt wird.

Der UCE (siehe Abbildung 4-12) wird aus dem StP Desktop aufgerufen und bietet folgende Funktionen zur Bearbeitung an:

- erstellen von *Use Case* Analysediagrammen
- zerlegen von *Use Cases* in Ereigniszenarien
- erstellen von Ereignisfluß- und Ereignisverfolgungsdiagrammen
- verwenden von Realzeitbeschränkungen bei Ereignissen
- zusammenfassen aller Ereignisbeziehungen für eine spezielle Klasse

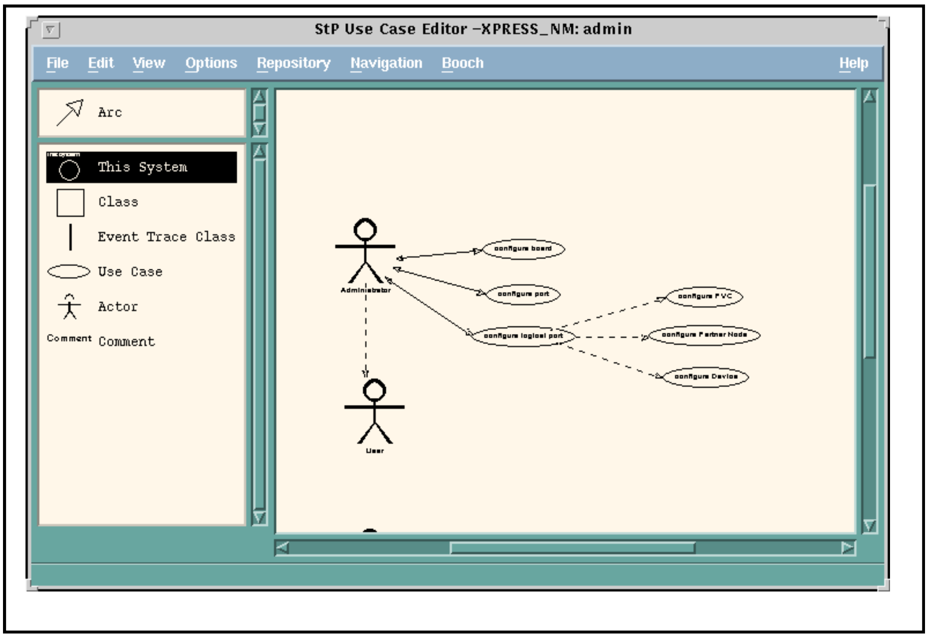


Abbildung 4-12: StP Use Case Editor

Der UCE selber bietet alle Funktionen und Menüoptionen, die in jedem Diagrammeditor von StP bereitgestellt werden. In der Objektliste sind dementsprechend alle für die Erstellung von *Use Cases* notwendigen Symbole enthalten. Diese müssen zur Verwendung nur in die rechte Seite des Fensters kopiert werden. Dort besteht dann die Möglichkeit zur Anordnung und Beschriftung der jeweiligen Symbole. Für weitere Informationen bezüglich der Verwendung des UCE wird auf [SPF96] verwiesen. Dort sind die einzelnen Symbole und Funktionen genauer beschrieben.

4.6.2 Definition der *Use Cases* für einen ATM-Switch

Im vorliegenden Kapitel soll nun versucht werden, *Actors* und *Use Cases* für einen ATM-Switch zu finden.
Bei der Betrachtung des Systems ATM-Switch von außerhalb kann man einige Akteure bestimmen. Unterscheidet man dabei noch zwischen dem *primary actor* und dem *secondary actor*, so kommt man zum folgenden Ergebnis.

primary actor	**secondary actor**
Endgerätebediener	Administrator
anderer ATM-Switch	

Nachdem die Akteure außerhalb des Systems festgelegt sind, kann mit der Modellierung der einzelnen *Use Cases* begonnen werden.

Bei den *primary actors* unterscheidet man zwischen dem Endgerätebediener bzw. User und einem anderen ATM-Switch. Beide sind Benutzer des Systems und greifen nicht zur Administration auf den Switch zu.
Bei der Betrachtung der beiden *primary actors* fällt auf, daß die jeweiligen Aufgaben bzw. Rollen sehr ähnlich sind. Dadurch wird klar, daß die *Use Cases* in einigen Fällen für beide gleich sind oder nur relativ geringe Unterschiede aufweisen. Im folgenden wollen wir den Vorgang, den ein User bei der Verwendung eines ATM-Switches durchführt, zerlegen. Dabei erhalten wir als Ergebnis die *Use Cases* für den User.
Wendet ein User A einen Switch an, so besteht sein Interesse darin, einen anderen User B zu erreichen. Um dies zu realisieren, ist es für User A notwendig, den Ort von User B zu kennen. Sollte er nicht bekannt sein, so ist eine Art Telefonbuch zur Bestimmung des Ortes von B sinnvoll. Daraus ergibt sich das erste *Use Case*: „anderen User im Netz suchen“. Hat User A seinen Partner lokalisiert, muß er eine Verbindung zu ihm aufbauen. Da auch die Möglichkeit besteht, daß User A nicht nur mit User B kommunizieren möchte, sondern gleichzeitig auch mit User C, gibt es beim Verbindungsaufbau zwei Varianten, die jeweils ein *Use Case* darstellen:

- „Punkt-zu-Punkt-Verbindung aufbauen“ oder
- „Punkt-zu-Mehrpunkt-Verbindung aufbauen“

Hat man sich für eine dieser Varianten entschieden, muß in beiden Fällen die Verbindung vom User zum Switch eingerichtet und die Verkehrsparameter festgelegt werden. Damit ist eine Verbindung vom User A zum User B und eventuell weiteren Usern aufgebaut. Daten, Sprache oder Video können nun entsprechend der festgelegten Parameter übermittelt oder empfangen werden.

Sollte User A zum gleichen Zeitpunkt einen Verbindungsaufbauwunsch von einem anderen User erhalten, so hat er die Möglichkeit, diesen entgegenzunehmen oder abzulehnen. Diese Situation entspricht einem weiteren *Use Case*.
Das letzte *Use Case* in diesem Szenario ist der Verbindungsabbauwunsch, der die eingerichtete Verbindung zwischen User und Switch wieder auflöst.
Um den gefundenen *Actor* und die *Use Cases* darzustellen, steht der schon im vorigen Abschnitt beschriebene *Use Case Editor* von StP zur Verfügung. Als Ergebnis des geschilderten Szenarios erhält man Abbildung 4-13.

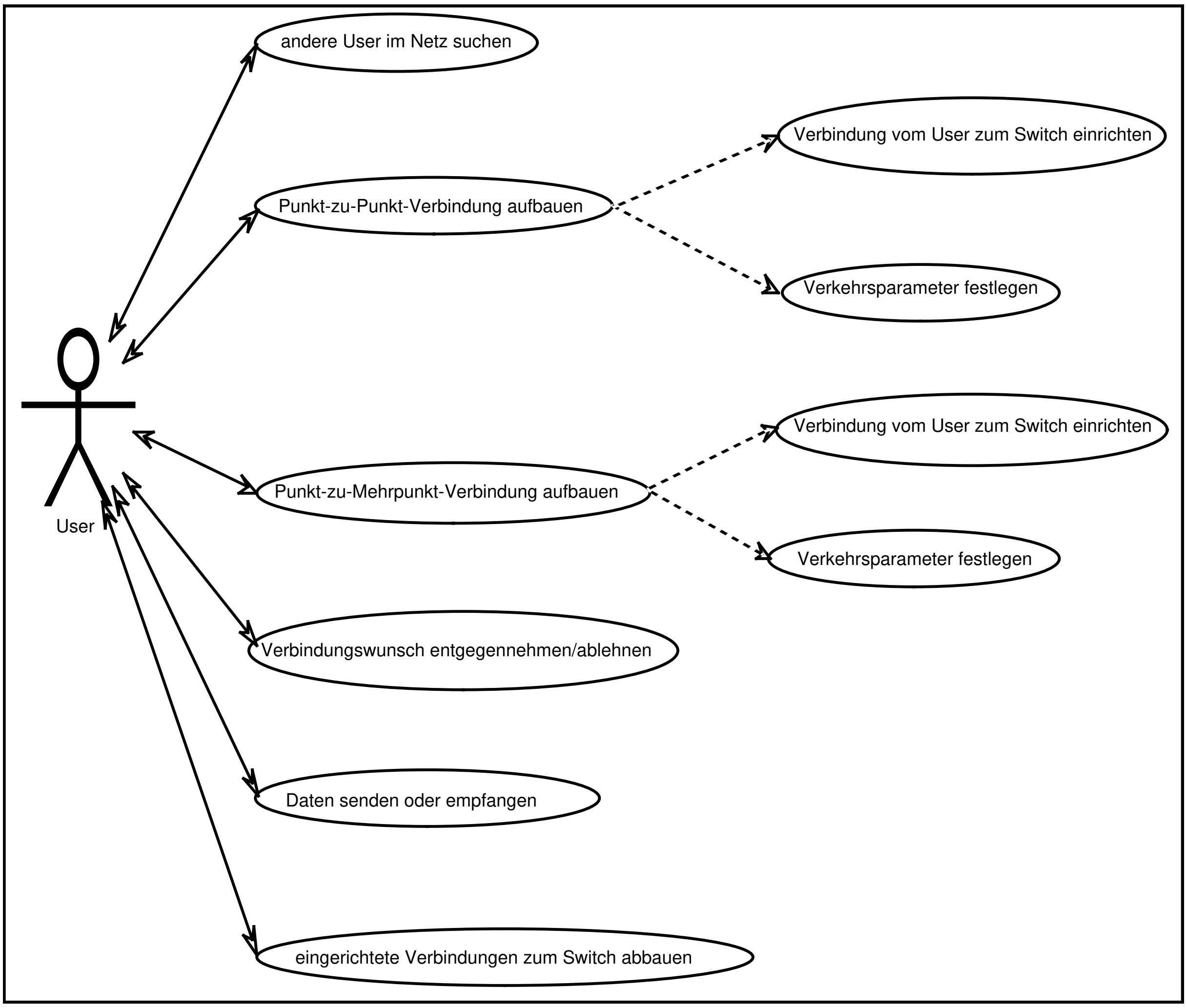


Abbildung 4-13: *Use Cases* für einen User

Vergleicht man das dargestellte Szenario mit dem anderen Fall des *primary actors*, so sind doch einige Gemeinsamkeiten zu finden. Auch für einen anderen Switch sind die ersten drei *Use Cases* des oberen Falls identisch:

- „PartnerNodes bestimmen“
- „Punkt-zu-Punkt-Verbindung einrichten“
- „Punkt-zu-Mehrpunkt-Verbindung einrichten“

Der Unterschied liegt beim Verbindungsaufbau darin, daß die Verbindung nicht zwischen einem User und einem Switch eingerichtet wird, sondern zwischen einem Eingangsport und einem oder mehreren Ausgangsports. Aber auch für diese Verbindungen müssen die jeweiligen Verkehrsparameter festgelegt werden.
Steht die Verbindung, können Daten empfangen und weitergeschickt werden. Außerdem besteht die Möglichkeit, daß Datenpakete verworfen werden. Dies ist abhängig von der Last im Switch und dem Wert des CLP-Bits im Header des jeweiligen Pakets, d. h. ist der Switch nicht mehr in der Lage, die ankommenden Pakete zu verarbeiten, besteht für ihn die Möglichkeit, Pakete zu verwerfen. Damit aber dabei keine wichtigen Pakete vernichtet werden, kontrolliert er das CLP-Bit. In diesem Feld ist die Priorität des Pakets angegeben und unterstützt damit die Entscheidung des Switches.
Der Switch ist außerdem in der Lage mit Hilfe des HEC-Feldes eine Fehlerkorrektur durchzuführen. Diese ist aber auf den Header der jeweiligen Zelle beschränkt und hat keine Auswirkungen auf das Informationsfeld der Zelle.
Das letzte *Use Case* stellt dann den Abbau der Verbindung dar. Eine graphische Zusammenfassung ist in Abbildung 4-14 dargestellt.

Der *secondary actor* steht im Gegensatz zu den beiden bisher besprochenen Fällen, da er den Switch als Administrator nicht aktiv verwendet, sondern nur Installations- und Konfigurationsaufgaben wahrnimmt. Will der Administrator dagegen auch eine Verbindung zu einem anderen User aufbauen, so verläßt er die Rolle des Administrators und wird zu einem *primary actor*. Daran erkennt man, daß die Möglichkeit besteht, die jeweiligen Rollen zu wechseln. Voraussetzung dazu ist natürlich, daß ein User die für einen Administrator notwendigen Rechte besitzt.
Zur Unterstützung bei der Formulierung der *Use Cases* für den Administrator werden die beiden Tabellen im Anschluß an die Abschnitte “Grundlagen von ATM“ und “Switchingtechniken“ herangezogen. Beide stellen Managmentfunktionalität und -information zur Verfügung, die bei einem ATM-Switch nicht fehlen sollten.
Die Suche nach den *Use Cases* wird erleichtert, wenn man das Szenario eines Administrators betrachtet. Um als Administrator Arbeiten an einem Switch durchführen zu können, muß er sich authentifizieren. Hat er sich eingeloggt, besteht die erste Aufgabe darin, die vorhandene Hardware, d.h. die Boards und Ports im Switch zu konfigurieren. Ist dieser Teil der Konfiguration abgeschlossen, so müssen für die einzelnen definierten Ports die logischen Ports festgelegt werden. Für diese logischen Ports wird dann das jeweilige Netzinterface konfiguriert. Man unterscheidet dabei zwischen zwei Schnittstellen. Auf der einen Seite muß man die Schnittstelle, auch UNI (User Network Interface) genannt, zwischen einem User und dem Switch konfigurieren. Auf der anderen Seite wird eine Schnittstelle zwischen zwei Netzwerken (Network-Network Interface, NNI) definiert.

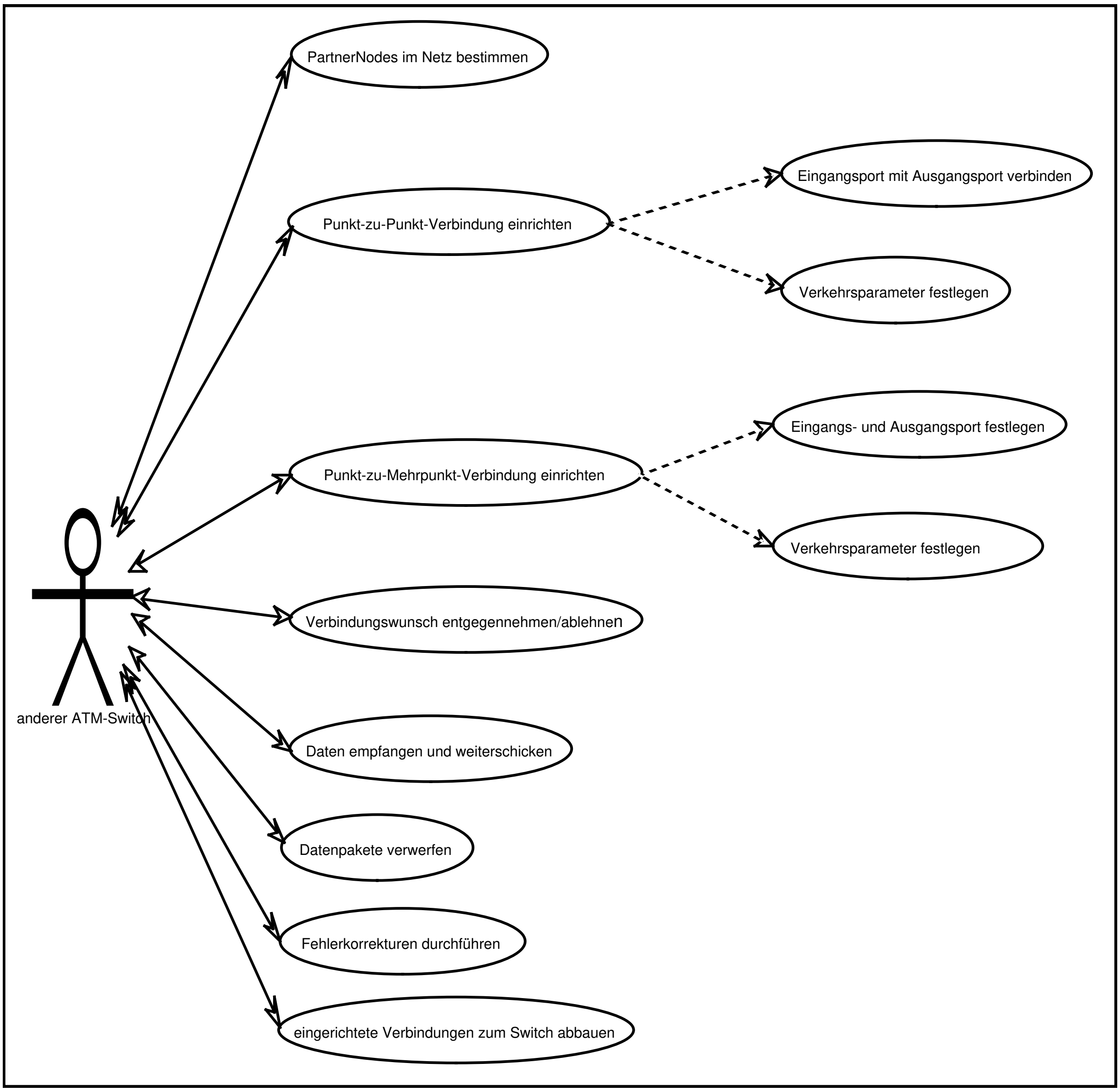


Abbildung 4-14: *Use Cases* für einen anderen Switch

Ein Beispiel für eine solche NNI ist die Verbindung von zwei ATM-Switches. Mit diesen beiden Schnittstellen ist die Arbeit am Netzinterface noch nicht abgeschlossen, da man sowohl für das UNI als auch für das NNI noch weitere Eigenschaften definieren muß.

Für das UNI richtet man die notwendigen User oder Devices ein, bei NNI konfiguriert man dagegen die Nodes und legt die dazugehörigen Routingtabellen fest. Vergleicht man diese Unterscheidung mit den beiden *primary actors* weiter oben, so kann man sagen, daß der User auf das UNI und ein anderer ATM-Switch auf das NNI zugreift.

Für die Verbindungen zwischen zwei Usern oder Switches gibt es nun zwei Möglichkeiten der Realisierung:

- den SVC (Switched Virtual Circuit) und
- den PVC (Permanent Virtual Circuit).

Im ersten Fall muß vom Administrator nichts mehr zusätzlich eingerichtet werden, da die Verbindung zur Laufzeit dynamisch geschaltet wird. Beim PVC dagegen wird vom Administrator eine feste Verbindung geschaltet, die vorher von ihm definiert wurde.
Um diese Verbindungen grundsätzlich zu schalten, werden vom Administrator die jeweiligen VPI- und VCI-Werte für den PVC festgelegt. Daran ist nun der PVC eindeutig erkennbar und der Switch weiß nun genau, daß alle Pakete mit einem bestimmten VCI- bzw. VPI-Wert auf einen bestimmten Port zu leiten sind.
Mit Abschluß dieser Aufgabe ist der Switch vom Administrator so konfiguriert, daß er in der Lage ist, als Komponente in einem Netzwerk tätig zu werden.
Jedoch sind die Tätigkeiten eines Administrators an dieser Stelle noch nicht beendet. Um die Verfügbarkeitszeiten des Switches möglichst hoch zu halten, ist eine ständige Fehlerüberwachung nötig. In diesem Zusammenhang sollte der Administrator auch von der vorliegenden Konfiguration des Switches ein Backup erstellen, damit er im Notfall die zerstörten Konfigurationsdaten wieder auf den Switch einspielen kann. Damit muß der Administrator auch in der Lage sein, einen Restart des Switches durchzuführen, da sonst ein Restore von Daten nicht sehr effektiv ist.

Eine weitere Aufgabe liegt darin, daß er Veränderungen an der Software im Switch einspielt, um Erweiterungen oder Verbesserungen schnell für die Nutzer verfügbar zu machen.

Ein Punkt, der die Performance des Switches betrifft, liegt in der Überwachung der Netzlast. Dabei soll der Administrator die Last im Switch überwachen und bei Problemen kontrollierend eingreifen, da sonst die Möglichkeit besteht, daß die Übertragung von Paketen vollständig unterbrochen wird.

Eine letzte Aufgabe, die in der letzten Zeit auch immer mehr an Bedeutung gewinnt, ist das sogenannte Accounting. Dabei soll der Administrator festhalten, wer wie lange eine bestimmte Verbindung verwendet hat und wieviele Daten über diese transportiert worden sind. Dies soll dazu dienen, eine Abrechnung zu ermöglichen, um dem Anwender wirklich nur die belegten Ressourcen in Rechnung zu stellen.
Um einen besseren Überblick über die beschriebenen Aufgaben zu erhalten, werden die *Use Cases* wieder mit dem *Use Case Editor* von StP dargestellt (Abbildung 4-15).

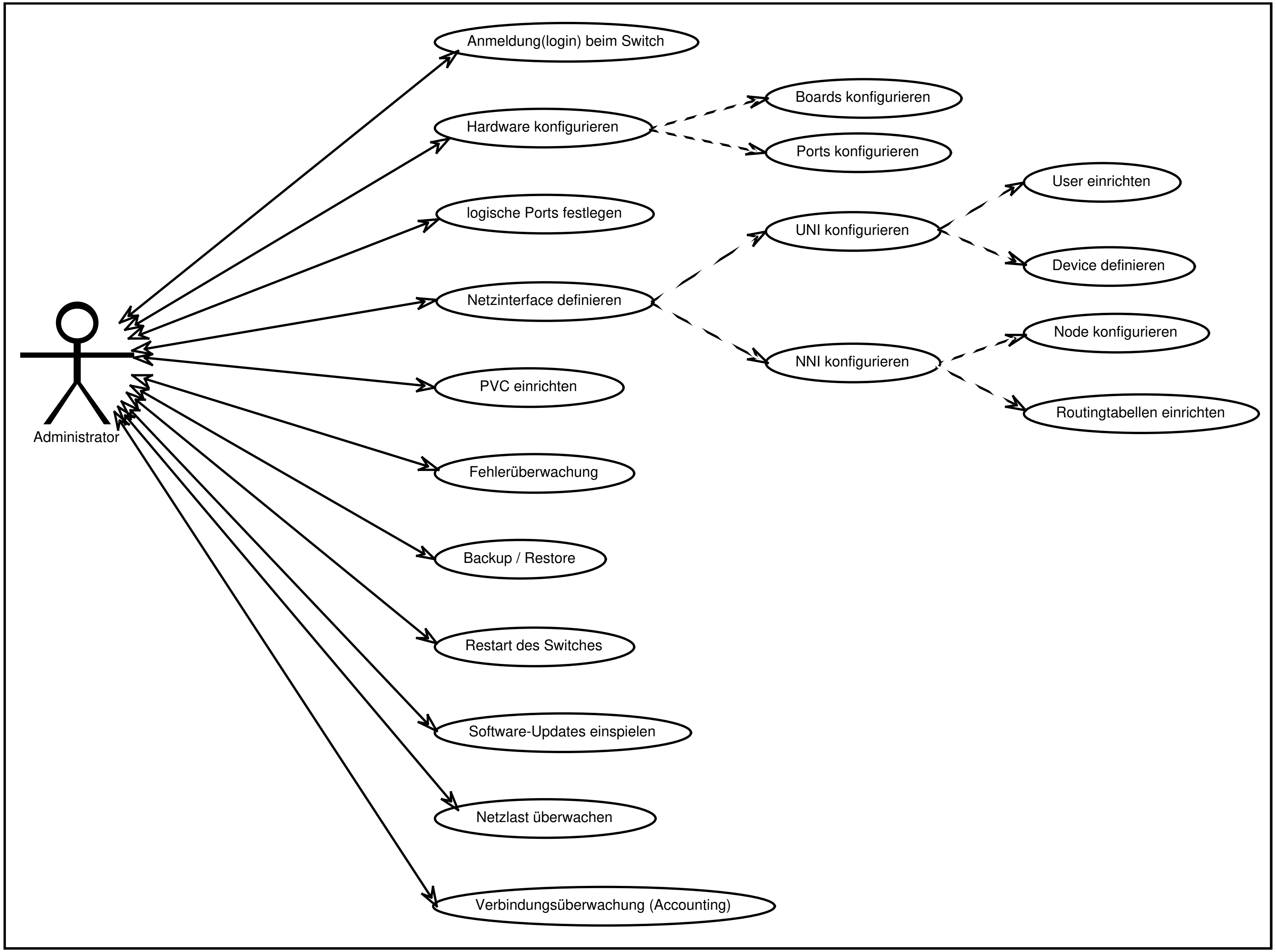


Abbildung 4-15: *Use Cases* für den Administrator

4.7 Zusammenführung der *Use Cases* und des Objektmodells

Nachdem nun die *Use Cases* vorliegen, ist im nächsten Schritt eine Kombination zwischen ihnen und dem Objektmodell der Version 3 zu realisieren. Als Ergebnis soll ein optimiertes Objektmodell entstehen, daß nicht nur für den Hicom Xpress einsatzfähig ist, sondern auch für Neu- oder Weiterentwicklungen.

Zur Durchführung dieser Aufgabe wird im folgenden jedes *Use Case Szenario* mit dem Objektmodell verglichen und versucht, die jeweiligen Klassen den *Use Cases* zuzuordnen. Dadurch werden am Ende diejenigen Klassen sichtbar, die im Objektmodell nur für den Hicom Xpress realisiert worden sind.

Szenario mit dem *Actor*: User
Beginnt man mit dem Szenario, daß ein User auf einen ATM-Switch zugreift (Abbildung 4-13), so erkennt man beim Vergleich mit dem Objektmodell, daß durch die Klassen `ATM_User` und `ATM_Device` die *Use Cases* realisierbar sind. Durch den Aufruf der Klasse `ATM_User` in der XAD werden alle User, die im System vorhanden sind, mit ihrer Telefonnummer angezeigt. Durch die anschließende Belegung eines freien Device durch einen User wird der Verbindungsaufbau realisiert. Durch die Freigabe des Device wird die Verbindung zwischen dem Switch und dem User wieder abgebaut. Bei dieser Betrachtung wird deutlich, daß die beiden Klassen für dieses Szenario nötig sind, da die in den *Use Cases* beschriebene Funktionalität bei allen ATM-Switches realisiert sein muß. Der einzige Unterschied, der auftreten kann, ist bei den Attributen zu finden. So wird ein ATM-Device beim Xpress über eine service_profile_ID addressiert, was aber bei anderen Switches durchaus anders implementiert sein kann. Daran erkennt man, daß man bei der Zusammenführung der *Use Cases* und des Objektmodells wahrscheinlich nicht nur die Klassen berücksichtigen muß, sondern auch die Attribute und eventuell die Methoden, um ein optimiertes Objektmodell zu erhalten.

Szenario mit dem *Actor*: anderer ATM-Switch
Betrachtet man nun das zweite Szenario, bei dem ein anderer ATM-Switch auf den Xpress zugreift, so wird sofort deutlich, daß die Klassen `Partner_Node` und `Routing_table` mit den zugehörigen Unterklassen von Bedeutung sind. Die Funktionalität des Szenarios für einen anderen Switch ist mit dem Szenario des Users vergleichbar. Auch die beiden Klassen haben ähnliche Aufgaben. In der Klasse `Routing_table` wird wieder die Möglichkeit bereitgestellt, andere Netzknoten zu suchen und an ihrer Telefonnummer zu identifizieren. Die Verbindung zwischen den beiden Switches wird dann über die Klasse `Partner_Node` realisiert. Auch bei diesen beiden Klassen erkennt man, daß die Funktionalität, die in den *Use Cases* beschrieben ist, vorhanden ist.

Somit gehören diese Klassen zum optimierten Objektmodell dazu. Unterschiede treten auch hier nur bei den einzelnen Attributen auf.
So wird die Adressierung wieder über das Attribut service_profile_ID abgewickelt. Außerdem besteht beim Xpress die Möglichkeit, eine Hicom 300 Telefonanlage anzuschließen und damit Telefonie über den Switch zu betreiben. Dies ist aber ein Feature, das nicht unbedingt bei anderen Switches zur Verfügung steht.
Da aber die Unterschiede für die Implementierung dieses Features nur in den Attributen zu finden ist, die auch bei einem anderen Switch verwendet werden, kann man dies nicht als eine proprietäre Objektklasse realisieren. Deshalb bleiben die beiden Klassen im optimierten Objektmodell enthalten.

Szenario mit *Actor*: Administrator
Das letzte *Use Case Szenario* beschreibt die Aufgaben eines Administrators, die für das Management eines Switches nötig sind.
Im ersten *Use Case* wird das Problem der Authentifizierung dargestellt. Betrachtet man das Objektmodell im Hinblick auf diese Tatsache, so findet man die Klasse `Authent`. In dieser Klasse wird die Authentifizierung für den Switch verwaltet, die aber aus Sicherheitsaspekten bei jedem Switch realisiert sein sollte. Somit gehört sie auch zum optimierten Objektmodell.
Die nächsten beschriebenen *Use Cases* dienen zur Konfiguration der Hardware, zum Festlegen der logischen Ports, zum Definieren des Netzinterfaces und zum Einrichten von PVCs. Dies sind Features, die für alle ATM-Switches von Bedeutung sind, da eine Kommunikation sonst nicht möglich ist. Vergleicht man die Namen der *Use Cases* mit den Objektklassen im Modell, so findet man in allen vier Fällen eine Übereinstimmung. Der einzige Unterschied liegt wiederum nur darin, daß es Attribute gibt, die in anderen Switches entweder nicht oder anders realisiert sind. Ein Beispiel ist das Attribut HW_address, da die Datenstruktur hierbei speziell für den Xpress entwickelt wurde. Damit dürfte diese Struktur bei anderen Firmen nicht vorkommen und bei der Verwendung des Objektmodells nicht funktionsfähig sein. Auch bei den Attributen HW_part_number und firmware_version liegen Datenstrukturen vor, die nicht den allgemein bekannten Datenstrukturen entsprechen und deshalb nicht ohne Probleme auf einen anderen ATM-Switch übertragbar sind.
Zurückblickend kann man bis hierher aber trotzdem sagen, daß alle bisher genannten Klassen für das optimierte Objektmodell verwendet werden müssen. Der einzige Punkt, der dabei zu beachten ist, liegt bei den Attributen und Methoden. Vor einem Einsatz dieses Objektmodells auf einem anderen Switch sollten deshalb die einzelnen Attribute auf ihre Realisierung überprüft werden. Die Möglichkeit, diese Attribute im Objektmodell einfach zu entfernen, besteht zum jetzigen Zeitpunkt nicht, da die meisten Attribute für die Realisierung des Managementtools von großer Bedeutung sind.

Das beste Beispiel in diesem Zusammenhang ist das Attribut HW_address, ohne das eine Adressierung der einzelnen Hardwarekomponeten im ATM-Switch nicht realisiert werden kann.
Die letzten Klassen, die jetzt noch nicht betrachtet wurden, sind die Klassen `Error_Log`, `Error_Rep`, `Error_Entry` und `Error_Trap`. Sie dienen beim Hicom Xpress der Fehlerüberwachung. Auch dies ist eine Funktion, die bei den *Use Cases* formuliert worden ist.
Jedoch ist dabei nicht definiert worden, in welchem Umfang die Fehler- und Netzüberwachung realisiert werden soll bzw. muß.
Grundsätzlich kann man sagen, daß bei allen Switches bestimmte Fehler, wie "Knoten nicht erreichbar" oder "Board deaktiviert", auftreten. Trotzdem gibt es mit großer Wahrscheinlichkeit Fehler, die nur bei einem bestimmten Switch auftreten. Ein anderer Switch kann mit dieser Fehlermeldung nichts anfangen. An diesem Problem erkennt man, daß auch im Bereich Fehler- und Netzüberwachung das Problem wieder bei den Attributen und den dazugehörigen Datenstrukturen liegt. Um die Fehlerüberwachung für alle Switches unterschiedlicher Hersteller gemeinsam zu realisieren, dürfen die jeweiligen Klassen nur Attribute enthalten, die für alle zutreffen. Die anderen Attribute müssen dann vom jeweiligen Hersteller in der zugehörigen Klasse ergänzt werden.

Betrachtet man nun noch die übrigen *Use Cases*, so findet man noch Funktionen, die im vorliegenden Objektmodell nicht berücksichtigt wurden. Der Grund liegt darin, daß z.B. die Funktion des Backup oder des Software-Updates nicht in der XAD realisiert sind. Dies kann bei der XAD nur über die Kommandozeile realisiert werden. Die Aufgabe der Verbindungsüberwachung ist dagegen in der XAD überhaupt nicht berücksichtigt. Für die Realisierung dieser Funktion müßte eine neue Objektklasse `Accounting` ins Objektmodell eingeführt werden, die diese Aufgaben dann übernimmt. Da dies aber in der Zukunft für den Xpress nicht geplant ist, wird diese Klasse auch im optimierten Objektmodell nicht berücksichtigt.

Versucht man nun aus diesen Erkenntnissen ein Fazit zu ziehen, so kommt man zu folgendem Ergebnis:
Die Zusammenführung der definierten *Use Cases* (Abbildungen 4-13, 4-14 und 4-15) mit dem "Objektmodell für ATM-Switches Version 3" haben zu folgendem Ergebnis geführt. Es ist nicht möglich, das Objektmodell in zwei große Teile zu spalten, einen generischen und einen devicespezifischen Anteil. Der Grund liegt darin, daß im entwickelten Objektmodell nur Klassen vorhanden sind, die für die Realisierung eines Managementtools für ATM-Switches notwendig sind. Damit ist es nicht möglich bestimmte Klassen dem generischen oder devicespezifischen Teil zuzuordnen.

Daraus darf aber nicht sofort gefolgert werden, daß das vorliegende Objektmodell einem generischen Objektmodell vollständig entspricht. Denn man kann nur sagen, daß alle vorkommenden Klassen zu einem generischen Objektmodell gehören, aber es gibt trotzdem noch weitere Klassen, wie `Accounting`, die im Objektmodell des Hicom Xpress fehlen. Zusätzlich sind im "Objektmodell für ATM-Switches Version 3" Attribute und Methoden in den Klassen enthalten, die wegen der Realisierung des Xpress vorhanden sind, d.h. sie sind devicespezifisch und müssen bei einem Switch einer anderen Firma nicht unbedingt vorkommen. Will man das so entstandene "Objektmodell für ATM-Switches Version 3" für andere ATM-Switches einsetzen, so ist eine Überprüfung der einzelnen Attribute und Methoden sinnvoll. Zusätzlich ist dann natürlich auch zu kontrollieren, ob bei diesen ATM-Switches Funktionalität vorhanden ist, die bisher im Hicom Xpress noch nicht realisiert wurde. Ist dies geschehen, kann man das Objektmodell anpassen und es als Grundlage für die Implementierung verwenden.

4.8 Zusammenfassung zum Kapitel 4

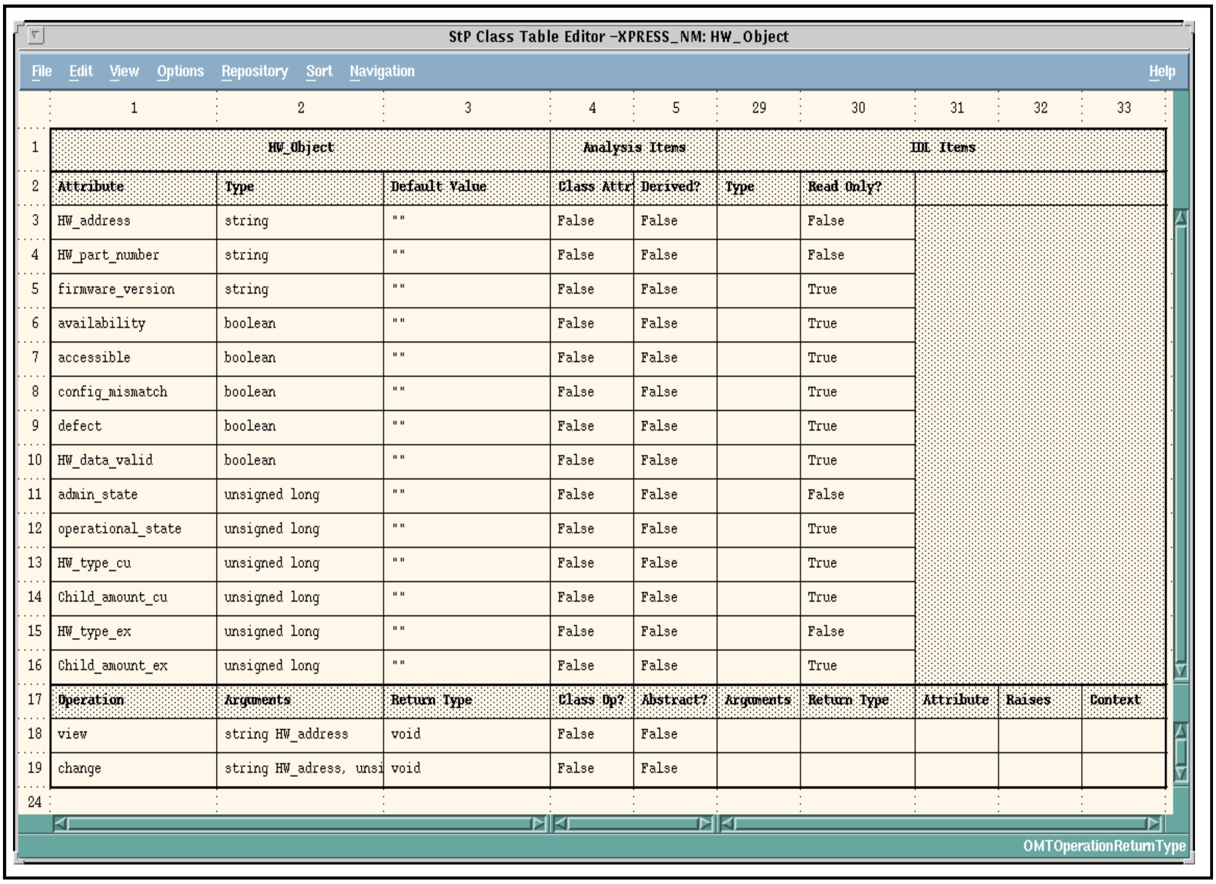
Versucht man nun am Ende des vierten Kapitels ein Fazit hinsichtlich der Modellierung zu ziehen, so kann man folgende Ergebnisse festhalten:
Die Entwicklung des Objektmodells für die XAD, beginnend beim Datenmodell ist keine leichte Aufgabe, da man versuchen muß, die Struktur des Hicom Xpress möglichst genau wiederzugeben, d.h. vor allem die Beziehungen zwischen den einzelnen vorhandenen Objektklassen. Um dieser Aufgabe gerecht zu werden, wurde die Entwicklung in mehreren Schritten durchgeführt, was dazu führte, daß drei Objektmodelle entstanden sind.
Die nächste Aufgabe bestand dann darin, aus dem dritten und letzten Objektmodell ein generisches Objektmodell zu entwickeln. Dazu betrachtete man das fertige Objektmodell im Zusammenhang mit den definierten *Use Cases* und stellte so fest, welche Veränderungen notwendig waren. Dies wurde durch die Modellierung der *Use Case Szenarios* deutlich, da mit diesen die Anforderungen an das Management erfaßt wurden. Im vorliegenden Fall sind damit sogar neue Anforderungen entstanden, die im bisherigen Objektmodell noch nicht berücksichtigt wurden.
Bei der folgenden Implementierung des Managementagenten wird aber nicht das vollständige generische Objektmodell als Grundlage verwendet, sondern das Objektmodell, das den Hicom Xpress am Besten abbildet. Der Grund liegt darin, daß die neuen Anforderungen, die im generischen Objektmodell enthalten sind, vom Hicom Xpress bisher nicht unterstützt werden können. Somit macht es keinen großen Sinn z.B. eine Klasse `Accounting` bei der Implementierung zu berücksichtigen. Jedoch muß man festhalten, daß diese Ergänzungen im generischen Objektmodell später ohne größeren Aufwand nachimplementiert werden können, vorausgesetzt der Hicom Xpress unterstützt diese Funktionalität.

5 Implementierung des Objektmodells

Nachdem nun das Objektmodell für das Management eines ATM-Switches entwickelt ist, besteht die nächste Aufgabe darin, dieses Objektmodell prototypisch zu implementieren. Die Realisierung stützt sich dabei auf CORBA mit seiner IDL-Sprache und C++. Im Kapitel 5.1 wird die Generierung des IDL-Codes aus dem Objektmodell beschrieben, bevor im Kapitel 5.2 das verwendete CORBA-Produkt vorgestellt wird. Im Anschluß daran werden die nächsten Schritte der Implementierung dargestellt, bevor im Kapitel 5.7 ein kurzer Erfahrungsbericht zur realisierten Implementierung folgt.

5.1 Generierung von IDL-Code

Nach der Erstellung des Objektmodells (siehe Version 3 im Anhang A) besteht jetzt die Möglichkeit, für die einzelnen Klassen den jeweiligen IDL-Code zu generieren. Bevor dies aber ohne Probleme durchgeführt werden kann, muß für jede Klasse eine eigene Klassentabelle erzeugt werden. Neben den Datentypen, Rückgabewerten usw. werden auch noch spezielle IDL-Parameter angegeben.
In der ersten Spalte der Klassentabelle (siehe Abbildung 5-1) werden die Attribute und Methoden angegeben, die StP selbständig aus dem Objektmodell übernimmt. Betrachtet man im folgenden nur die Attribute, so wird in den nächsten beiden Spalten der Datentyp des Attributs und der Defaultwert angegeben. Darauf folgen zwei Spalten die als “Analysis Items” zusammengefaßt sind. Diese beiden Punkte sind unabhängig von der jeweils verwendeten Programmiersprache und beziehen sich mehr auf OMT selbst. In der Spalte “Class Attr” wird festgelegt, ob nur ein einzelner Wert für das Attribut existiert, der allerdings für alle Klassen oder Objektinstanzen gemeinsam ist. In der Spalte “Derived?” wird dagegen festgelegt, ob das vorliegende Attribut von anderen Attributen berechnet wird.
Bei den Methoden wird nach dem Methodennamen in den nächsten beiden Spalten die Argumentenliste und der Rückgabewert angegeben. Unter der Spalte “Analysis Items” wird bei den Methoden bei “Class Op?” angegeben, ob die Methode besser eine Klassenmethode oder eine Methode der Klasseninstanz ist. Bei “Abstract?” wird festgelegt, ob die Methode abstrakt ist oder nicht. Wenn sie abstrakt ist, kann sie damit nicht in der Instanz einer Klasse implementiert werden.
Nachdem man nun in der Klassentabelle die allgemeinen Informationen für die jeweiligen Klassen des Objektmodells festgelegt hat, müssen im nächsten Schritt die programmiersprachenabhängigen Informationen definiert werden. Neben IDL können hier auch Informationen für C++, Ada95 und Smalltalk angezeigt werden.
Betrachtet man wieder zuerst die Attribute, so gibt es unter den “IDL Items” zwei Spalten.



	1	2	3	4	5	29	30	31	32	33
1	HW_Object			Analysis Items		IDL Items				
2	Attribute	Type	Default Value	Class Attr	Derived?	Type	Read Only?			
3	HW_address	string	""	False	False		False			
4	HW_part_number	string	""	False	False		False			
5	firmware_version	string	""	False	False		True			
6	availability	boolean	""	False	False		True			
7	accessible	boolean	""	False	False		True			
8	config_mismatch	boolean	""	False	False		True			
9	defect	boolean	""	False	False		True			
10	HW_data_valid	boolean	""	False	False		True			
11	admin_state	unsigned long	""	False	False		False			
12	operational_state	unsigned long	""	False	False		True			
13	HW_type_cu	unsigned long	""	False	False		True			
14	Child_amount_cu	unsigned long	""	False	False		True			
15	HW_type_ex	unsigned long	""	False	False		False			
16	Child_amount_ex	unsigned long	""	False	False		True			
17	Operation	Arguments	Return Type	Class Op?	Abstract?	Arguments	Return Type	Attribute	Raises	Context
18	view	string HW_address	void	False	False					
19	change	string HW_adress, unsi	void	False	False					
24										

Abbildung 5-1: Klassentabelle für die Klasse `HW_Object`

Bei “Type” wird der Attributtyp angegeben, der in IDL-Ausdrücken verwendet werden soll, d.h. es wird der urspüngliche Attributtyp überschrieben. Die Spalte “Read only?” gibt an, ob das Attribut verändert werden darf (False) oder nicht (True).

Bei den Methoden gibt es insgesamt fünf weitere Spalten für die Generierung von IDL-Code. In den Spalten “Arguments” und “Return Type” kann man die jeweiligen Argumente und Rückgabewerte für die IDL-Ausdrücke festlegen. Wird dies gemacht, werden die ursprünglichen Argumente und Rückgabewerte überschrieben. Um eine spezielle Eigenschaft oder Charakteristik für eine Methode festzulegen, gibt es die Spalte “Attribute”. Der IDL-Standard erlaubt hier aber nur den Wert “oneway”, d.h. wird dies verwendet, ist die Operation zu einem *oneway-request* geworden. Die Spalte “Raises” dient dazu, einen IDL Ausdruck zu definieren, der beschreibt, welche Ausnahmen hervortreten, wenn eine spezielle Operation aufgerufen wird. In der letzten Spalte mit dem Titel “Context” wird ein IDL Ausdruck spezifiziert, der angibt, welche Elemente im Umfeld des Clients eine Leistungsverbesserung der Operationen bewirken.

Hat man nun die Klassentabellen für alle Objektklassen fertiggestellt, wird im nächsten Schritt für die jeweiligen Klassen der IDL-Code generiert. Dazu wechselt man zum StP-Desktop und dort in die Kategorie “Access Repository”.

Dort werden unter dem Menüpunkt “Classes” alle erstellten Klassen des Objektmodells zur Verfügung gestellt. Hat man dann eine spezielle Klasse, z.B. `HW_Object` ausgewählt, werden im Menü “Commands” die Einträge aktiv und durch den Aufruf “Generate IDL for class ...” wird die “IDL Code Generation Property Dialog Box” geöffnet. Dort werden Informationen zum Ausgabeverzeichnis, zur Extension, zum Namen des Ausgabefiles usw. angegeben. Für genauere Informationen zur Dialogbox wird auf [SPG96] verwiesen. Hat man dann die Einstellungen für die Generierung des Codes abgeschlossen, wird der IDL-Code generiert. Im Fall der Klasse `HW_Object` wird folgender IDL-Code in der Datei “HW_Object.idl” erzeugt (Abbildung 5-2).

```
#ifndef _HW_Object_idl_
#define _HW_Object_idl_

// stp class declarations
interface HW_Object;
// stp class declarations end

// stp class definition 208
interface HW_Object
{
// stp class members
 readonly attribute string HW_address;
 readonly attribute string HW_part_number;
 readonly attribute string firmware_version;
 readonly attribute string availability;
 readonly attribute string accessible;
 readonly attribute string config_mismatch;
 readonly attribute string defect;
 readonly attribute string HW_data_valid;
 attribute string admin_state;
 readonly attribute string operational_state;
 readonly attribute string HW_type_cu;
 readonly attribute string Child_amount_cu;
 readonly attribute string HW_type_ex;
 readonly attribute string Child_amount_ex;
 void view(in string HW_address);
 void change(in string HW_address, in string admin_state);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Abbildung 5-2: Generierter IDL-Code für die Klasse `HW_Object`

Führt man dies nun für alle Objektklassen aus, so erhält man insgesamt 18 IDL-Files, die jeweils nach der Objektklasse benannt sind. Eine Aufstellung aller dieser Daten steht im Anhang B zur Verfügung.

Betrachtet man nochmals rückblickend das Objektmodell, so wird klar, daß auch die Beziehungen zwischen den einzelnen Klassen mit in die IDL-Files übernommen werden sollten.
Ein Beispiel für eine Vererbungsbeziehung besteht z. B. zwischen den Klassen `HW_Object`, `ATM_Switch`, `ATM_Board` und `ATM_Port`. In der Klasse `ATM_Switch` kann man an zwei Stellen eine Veränderung gegenüber der Klasse `HW_Object` feststellen. In der Abbildung 5-3 sind diese Ergänzungen farbig dargestellt.

```
#ifndef _ATM_Switch_idl_
#define _ATM_Switch_idl_

// stp class declarations
interface ATM_Switch;
// stp class declarations end

// stp auto-include 126
#include "HW_Object.idl"
// stp auto-include end

// stp class definition 126
interface ATM_Switch : HW_Object {
// stp class members
 void auto_config(in string HW_address);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Abbildung 5-3: Generierter IDL-Code für die Klasse `ATM_Switch`

Weitere Beziehungen wie Aggregationen oder Assoziationen werden bei der IDL-Code Generierung jedoch nicht im jeweiligen IDL-File abgebildet. Dadurch wird es nötig, bei der Implementierung des Objektmodells auf diese Beziehungen zu achten. Somit kann man abschließend sagen, daß man zwar mit StP das Objektmodell sehr gut darstellen kann, aber leider nicht alle Beziehungen mit in die jeweiligen IDL-Files übernehmen werden. Dadurch wird der Entwickler gefordert, um die Einschränkungen, die ja durch die Assoziationen und Aggregationen ausgedrückt werden, bei der Implementierung zu berücksichtigen.

5.2 Einführung in *Orbix* von IONA

Bevor im nächsten Abschnitt auf die weiteren Schritte zur Implementierung des Objektmodells eingegangen wird, soll in diesem Teil das Produkt vorgestellt werden, das für die Implementierung notwendig ist.

Wie im ersten Kapitel beschrieben soll ein Objektmodell zum Management eines ATM-Switches entwickelt werden, wobei sich die Implementierung dabei auf CORBA abstützen soll. Zentrale Komponente einer CORBA-Implementierung ist dabei der ORB, der die Kommunikation zwischen Client und Server steuert. Einer der führenden Anbieter solcher ORBs ist die Firma IONA-Technologies, die teilweise im Besitz der Sun Microsystems ist. Die erste Version von diesem ORB, *Orbix*, wurde im Juni 1993 veröffentlicht. Seit diesem Zeitpunkt ist er zu einem der führenden ORBs aufgestiegen. Die Implementierung von *Orbix* entspricht dem neuesten CORBA-Standard und unterstützt die Standardumwandlung von IDL in C++, die von der OMG definiert wurde. *Orbix* ist auf mehr als 20 Betriebssystemplattformen lieferbar und ist dadurch auch zwischen unterschiedlichen Systemen interoperabel.
Die Aufgabe des ORB ist es, Clients und Server zu verbinden. Damit ein Client einen Server auch finden kann, stellt IONA eine einfache Datenbasis, das *Implementation Repository*, zur Verfügung. Ist der Server dort registriert, kann er zu jeder Zeit von einem Client angesprochen werden. Bis diese Kommunikation aber realisierbar ist, sind einige Arbeitsschritte notwendig. Im folgenden werden die Schritte erläutert, die für die Implementierung des Objektmodells durchzuführen sind.

Im Makefile “orbixmtsol2s4.mk” müssen die Pfade für den *Orbix*, das Fenstersystem und den C++-Compiler eingetragen werden. Außerdem werden die Flags für das Linken und für den IDL-Compiler festgelegt. Zu beachten ist dabei, daß beim Solaris 2.5 Betriebssystem nur der SPARC Compiler 4.0.1 ohne Probleme arbeitet. Andere Compiler wie der gnu C++-Compiler sind nicht in der Lage problemlos zu übersetzen. Daneben müssen in der “.login”-Datei folgende Pfade gesetzt werden:

- ‘IT_CONFIG_PATH /etc/Orbix.cfg’ für das Konfigurationsfile von *Orbix*
- ‘IT_IMP_REP_PATH /homes/hoelsche/Orbix/Repository’ für das *Implementation Repository* von *Orbix*
- ‘LD_RUN_PATH /usr/opt/ORBIX_2.2MT/lib’ für das dynamische Linken

Hat man diese Vorarbeiten geleistet, kann man beginnen, das IDL-File “xyz” zu compilieren. Durch diesen Schritt entstehen drei neue Dateien:

- eine Header-Datei xyz.hh, die bei der Implementierung von “xyz” und allen Clients eingebunden werden muß
- eine Quelldatei xyzS.cc, die bei der Implementierung von “xyz” compiliert und eingebunden werden muß

- eine Quelldatei xyzC.cc, die in die Clients für “xyz” eingebunden und compiliert werden muß

Zusätzlich zu diesen Dateien ist es notwendig, noch vier weitere Dateien zu erzeugen. Zwei Dateien, xyz_i.h und xyz_i.cc, werden für die Implementierung der “xyz” Schnittstelle benötigt und die anderen beiden, sind das Client- und das Serverprogramm.

Hat man alle Dateien erzeugt bzw. geschrieben, müssen sie übersetzt und gebunden werden. Hat dies ohne Probleme funktioniert, sind ein lauffähiger Server und Client entstanden. Damit der Client jetzt aber den Server aufrufen kann, muß zuerst der ORB (orbixd) gestartet und der Server im *Implementation Repository* registriert werden. Diese Reihenfolge ist einzuhalten, da eine Registrierung des Servers sonst nicht möglich ist. Zur Verdeutlichung werden diese beiden Schritte in der Abbildung 5-4 für das Grid-Demo der Firma IONA dargestellt.

```
// Starten des Orbix-Daemon:
ipx14% orbixd &
[1] 586
ipx14%
[orbixd: Server "IT_daemon" is now available to the network ]
[     Configuration Orbix-TCP/1570/Orbix-XDR ]

// Registrierung des Servers im Implementation Repository:
ipx14% /usr/opt/Orbix_2.2MT/corba2/bin/putit -hipx14 grid
 /homes/hoelsche/Orbix/demos/grid/server
[orbixd: New Connection (ipx14,588,*,hoelsche,pid=588,optimised) ]
[588: New Connection (ipx14,IT_daemon,*,hoelsche,pid=586,optimised) ]
[orbixd: End of Connection (ipx14,588,*,hoelsche,pid=588) ]

// Aufruf des Clients:
ipx14% client ipx14
[orbixd: New Connection (ipx14,589,*,hoelsche,pid=589,optimised) ]
[589: New Connection (ipx14,IT_daemon,*,hoelsche,pid=586,optimised) ]
[orbixd: Processing Server Details request :  grid//_IT_PING(hoelsche) ]
[orbixd: Dynamically assigning internet port 1591 ]
[orbixd: Dynamically assigning internet port 1592 for IIOP protocol]
[grid: Server "grid" is now available to the network ]
[     Configuration tcp/1591/xdr ]
[orbixd: Server details :  xdr,tcp,1591,1592,shared ]
[589: New Connection (ipx14,grid,*,hoelsche,pid=590,optimised) ]
[grid: New Connection (ipx14,589,*,hoelsche,pid=589,optimised) ]

//Ergebnisausgabe des Clients:
height is 100
width  is 100
grid[2,4] is 123

[orbixd: End of Connection (ipx14,589,*,hoelsche,pid=589) ]
ipx14% [grid: End of Connection (ipx14,589,*,hoelsche,pid=589) ]
```

Abbildung 5-4: Beispielablauf für ein *Orbix*-Programm

Neben dem “putit”-Kommando stehen noch eine Reihe weiterer Befehle zur Verfügung, um Server zu registrieren und zu aktivieren:

- catit: Auflisten aller Einträge des Repository
- chmodit: Verändern von Start- und Aufrufrechten der Server
- chownit: Verändern der Eigentumsrechte eines Servers
- killit: Abbrechen des Serverprozesses
- lsit: Ausgeben eines speziellen Eintrags des Repository
- mkdirit: Erzeugen eines Verzeichnisses zur Strukturierung des Repository
- pingit: Kontrollieren des *Orbix Daemon* darauf, ob er noch lebt
- psit: Ausgabe einer Liste aller dem *Orbix Daemon* bekannter Server
- rmdirit: Entfernen eines Verzeichnisses aus dem Repository
- rmit: Entfernen oder Modifizieren eines Eintrags im Repository

Betrachtet man zum Abschluß dieses Kapitels noch die Realisierung des *Orbix* unter *Windows NT 4.0*, so stellt man fest, daß die grundlegenden Funktionen genauso wie bei *Unix* vorhanden sind. Um den *Orbix* einsetzen zu können waren in der gegebenen Testumgebung folgende Einstellungen nötig:

- Datei D:\MSDEV\LIB\MSVCRT.LIB ins lib-Verzeichnis von *Orbix* kopieren und in “MSVCIRT.LIB” umbenennen
- Pfadvariablen in der Datei “orbix.cfg” anpassen
- IT_CONFIG_PATH = C:\Orbix22\orbix.cfg setzen

Zur Compilierung unter *Windows NT 4.0* wird *Visual C++ Version 4.1* von Microsoft verwendet. Ob *Orbix* auch unter *Windows NT 4.0* nur diesen einen Compiler zuläßt, konnte nicht getestet werden, da kein weiterer zur Verfügung stand.
Hat man nun auch dort z.B. das Grid-Demo von IONA implementiert, so besteht die Möglichkeit, den Server auf der jeweils anderen Maschine aufzurufen. Dazu ist es nur wichtig, daß beide Server registriert sind und beim Clientaufruf der jeweils richtige Host mit angegeben wird.
Für detailliertere Informationen bezüglich der Realisierung von *Orbix* wird auf [ORP97] und [ORR97] verwiesen. Außerdem wird in den folgenden Abschnitten bei der Implementierung des Objektmodells noch näher auf die einzelnen Schritte eingegangen.

5.3 Compilierung des IDL-Codes mit dem *Orbix*-Compiler

Bevor nun mit der Compilierung des IDL-Codes begonnen werden kann, sind an den IDL-Files zwei Änderungen durchzuführen.

Die erste Änderung betrifft die Methoden. Um die Übersichtlichkeit des Objektmodells nicht noch weiter zu verschlechtern, wurden bei allen Parametern die Operatoren “in”, “out” oder “inout” weggelassen. Sie geben an, ob der Wert des Arguments der Methode übergeben oder ausgelesen werden soll. Auch beide Richtungen sind für ein Argument durch den Operator “inout” möglich. Somit müssen bei jedem IDL-File die Argumente der Methode um diese Operatoren ergänzt werden. Ist dies geschehen, sollten unter normalen Umständen kein Probleme bei der Compilierung der Dateien auftreten.

Die zweite Veränderung betrifft alle Datentypen im jeweiligen IDL-File. Bisher wurden im Objektmodell für alle Attribute und die Argumente in den Methoden die jeweiligen Datentypen festgelegt. Zur Compilierung der Dateien müssen jetzt aber alle Datentypen einheitlich auf “string” abgeändert werden.
Der Grund für diese doch ungewöhnliche Aktion liegt an der Realisierungsmöglichkeit für die Anbindung des Prototypen an den Switch. Das Betriebssystem auf dem *Control Block* (CB) des Switches ist *Unixware*. Somit besteht die Notwendigkeit, einen ORB für dieses Betriebssystem zu verwenden. Leider hat IONA unerwartet die Unterstützung von *Unixware* beendet. Somit steht kein ORB für diese Plattform zur Verfügung. Eine Portierung aller notwendigen Sourcen auf ein anderes Betriebssystem war in der kurzen Zeit leider auch nicht möglich. Um den Prototypen trotzdem zu realisieren wurde ein CORBA-Proxy entwickelt, der die Fortsetzung der Arbeit ermöglicht. Die genaue Realisierung soll zu diesem Zeitpunkt aber nicht erläutert werden, da sie zur Compilierung des IDL-Codes nicht notwendig ist.
Um die einzelnen Arbeitsschritte in den folgenden Kapiteln besser darstellen zu können, werden aus der Datei “HW_Object.idl” einige Attribute und Methoden gelöscht. Dadurch besteht die Möglichkeit, die einzelnen Abläufe für die Implementierung zu verfolgen. In der Abbildung 5-5 ist die abgespeckte IDL-Datei dargestellt.
Um diese Datei jetzt zu compilieren, muß der mitgelieferte IDL-Compiler von IONA aufgerufen werden. Er wird durch den Aufruf des Shellkommandos ***idl*** gestartet. Der einzige Parameter, der bei diesem Aufruf mitgegeben werden muß, ist der Name der IDL-Datei. Die Extension für diese Datei muß ***.idl*** sein.
Ruft man nun den IDL-Compiler für die IDL-Datei aus Abbildung 5-5 auf, so werden folgende drei Dateien erzeugt:

- HW_Object.hh
- HW_ObjectS.cc
- HW_ObjectC.cc

```
#ifndef _HW_Object_idl_
#define _HW_Object_idl_

// stp class declarations
interface HW_Object;
// stp class declarations end

// stp class definition 208
interface HW_Object
{
// stp class members
readonly attribute string HW_address;
attribute string admin_state;
void change(in string adstate);
// stp class members end
};
// stp class definition end


// stp footer
#endif
// stp footer end
```

Abbildung 5-5: Ausschnitt aus “HW_Object.idl”

Neben diesen drei generierten Dateien werden für die Realisierung noch weitere vier Dateien benötigt, die teilweise auch durch den IDL-Compiler erzeugt werden können. Verwendet man beim Aufruf des Compilers den Switch **-S**, so werden zusätzlich folgende Rumpfdateien für die Implementierungsklassen erzeugt:

- HW_Object.ih
- HW_Object.ic

Um sie für die nächsten Schritte verwenden zu können, müssen sie in “HW_Object_i.h” und “HW_Object_i.cc” umbenannt werden. Außerdem sind noch Veränderungen am Code selber durchzuführen. Da diese beiden Dateien auch eine wichtige Rolle bei der Implementierung spielen, werden die notwendigen Veränderungen im nächsten Abschnitt mit aufgegriffen.

Ein zweiter Schalter, der bei der Compilierung der IDL-Datei nötig ist, ist **-B**. Dieser wird benötigt, wenn man die *BOAImpl approach* bei der Implementierung der Schnittstelle einsetzen möchte. Denn durch Angabe dieses Schalters wird für die Schnittstelle jeder Klasse eine *BOAImpl* Klasse erstellt. Damit liegen nach Beendigung des Compilierlaufs zwei C++-Klassen vor. In unserem Fall sind es die Klassen `HW_Object` und `HW_ObjectBOAImpl`, wobei die *BOAImpl*-Klasse von der korrespondierenden C++-Klasse `HW_Object` erbt.

Um nun zu zeigen, daß eine C++-Klasse eine gegebene IDL-Schnittstelle implementiert, soll diese Klasse (`HW_Object_i`) von der *BOAImpl*-Klasse (`HW_Object-BOAImpl`) erben. Dieser Vorgang wird dann als *BOAImpl approach* bezeichnet.
Der Vorteil dieser Implementierung liegt darin, daß man für die `HW_Object_i`-Klasse z.B. eigene Konstruktoren oder Destruktoren definieren kann. In Abbildung 5-6 ist diese Konstellation nochmals graphisch erläutert.

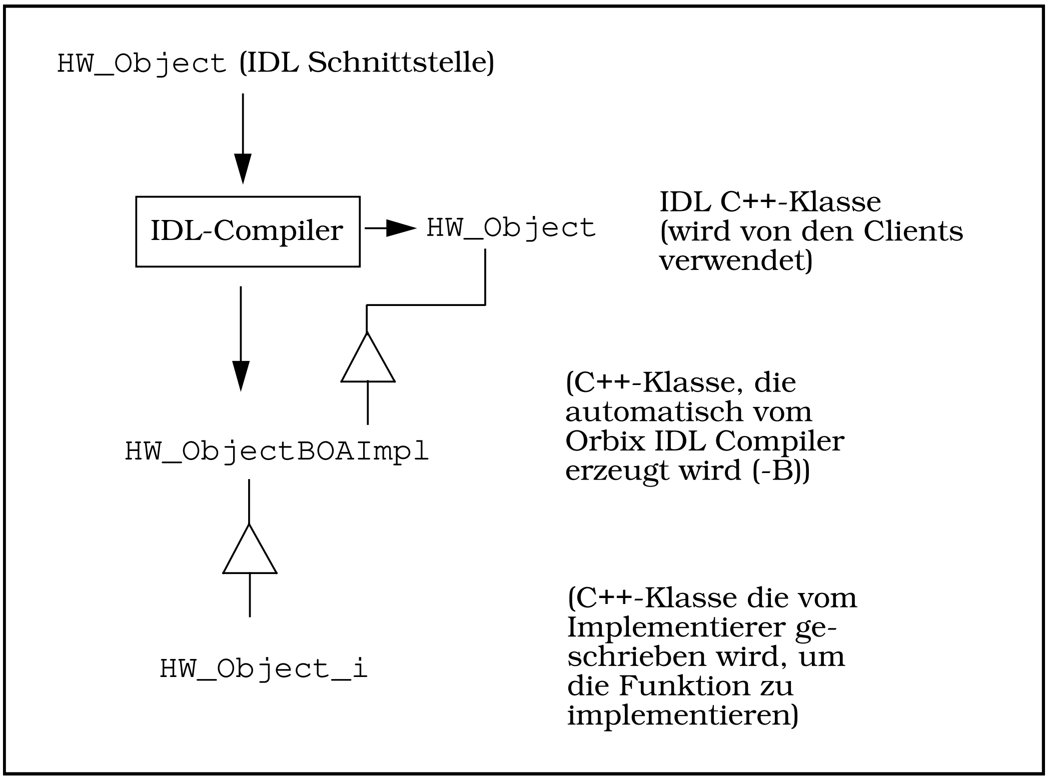


Abbildung 5-6: Erläuterung des *BOAImpl approach*

Mit dem Aufruf: **idl -B -S HW_Object.idl** werden somit fast alle für die Implementierung benötigten Dateien erzeugt. Es fehlt nur der Code für den Client und den Server selbst. Festzustellen ist dabei, daß die Implementierungsdateien “HW_Object_i.cc” und “HW_Object_i.h” für die weiteren Arbeitsschritte der Implementierung nicht ohne Veränderungen verwendet werden können. Die jeweils notwendigen Veränderungen sind in der Abbildung 5-7 und Abbildung 5-8 farbig hervorgehoben.

```
#define HW_Object_ih
#ifndef HW_Object_ih

#include "HW_Object.hh"

class HW_Object_i {
public:
 virtual char * HW_address (CORBA::Environment &IT_env=CORBA::default_environment)
                throw (CORBA::SystemException);
 virtual void admin_state (const char *admin_state, CORBA::Environment &IT_env=CORBA::
                default_environment) throw (CORBA::SystemException);
 virtual char * admin_state (CORBA::Environment &IT_env=CORBA::default_environment)
                throw (CORBA::SystemException);
 virtual void change (const char * adstate, CORBA::Environment &IT_env=CORBA::
                default_environment) throw (CORBA::SystemException);
};
#endif
```

Abbildung 5-7: Datei “HW_Object_i.h”

```
char * HW_Object_i::HW_address (CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
}

void HW_Object_i::admin_state (const char * admin_state, CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
}

char * HW_Object_i::admin_state (CORBA::Environment &IT_env) throw (CORBA::SystemException) {
}
void HW_Object_i::change (const char * adstate, CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
}
```

Abbildung 5-8: Datei “HW_Object_i.cc”

Ein Problem bestand darin, daß es in der Dokumentation leider keine Hinweise gab, welche Änderungen wirklich notwendig sind, da auch die Fehlermeldung des Compilers nicht sehr hilfreich war. Dadurch wurde der Zeitaufwand zu Beginn etwas erhöht, war aber bei der weiteren Implementierung nicht weiter von Bedeutung.
Abschließend kann man zum Bereich der Compilierung der IDL-Dateien sagen, daß er mit Ausnahme der obigen Aussage gut von IONA unterstützt wird, da man für die weitere Arbeit gute Implementierungsvorlagen erhält.

5.4 Einbinden der C++-Systemaufrufe in die C++-Rümpfe

Nachdem vom IDL-Compiler die notwendigen Dateien erzeugt wurden, besteht die nächste Aufgabe darin, in den Implementierungsdateien “HW_Object_i.h” und “HW_Object_i.cc” weitere Veränderungen bzw. Ergänzungen vorzunehmen.
Damit die Implementierungsklassen ihre Aufgaben übernehmen können, müssen C++-Systemaufrufe in der Datei “HW_Object_i.cc” eingebaut werden. Diese Arbeit ist unabhängig von der zu lösenden Implementierung, d. h. bei jeder Implementierung mit *Orbix* müssen diese Veränderungen in den Implementierungsklassen ausgeführt werden.
Um die notwendigen Änderungen besser erläutern zu können, sind in den beiden folgenden Abbildungen die jeweiligen Änderungen farbig hervorgehoben. Die Anpassungen, die durch den Compilerlauf notwendig waren und im Kapitel 5.3 angesprochen wurden, werden jetzt nicht mehr farbig hervorgehoben.

```
#define HW_Object_ih
#ifndef HW_Object_ih

#include "HW_Object.hh"

class HW_Object_i : public virtual HW_ObjectBOAImpl {
public:
 //ctor
 HW_Object_i(char *a, char *b);

 //dtor
 virtual ~HW_Object_i();

 virtual char * HW_address (CORBA::Environment &IT_env=CORBA::default_environment)
                  throw (CORBA::SystemException);
 virtual void admin_state (const char * admin_state, CORBA::Environment &IT_env=CORBA::
                  default_environment) throw (CORBA::SystemException);
 virtual char * admin_state (CORBA::Environment &IT_env=CORBA::default_environment)
                  throw (CORBA::SystemException);
 virtual void change (const char * adstate, CORBA::Environment &IT_env=CORBA::
                  default_environment) throw (CORBA::SystemException);
};
#endif
```

Abbildung 5-9: Header-Datei “HW_Object_i.h”

Betrachtet man die Veränderungen in Abbildung 5-9, so kann man zwei Dinge feststellen. Erstens werden ein Konstruktor und ein Destruktor für die Implementierungsklasse eingeführt, um ein Objekt für die Klasse erzeugen bzw. löschen zu können. Zweitens wird die Vererbungsstruktur, die in Abbildung 5-6 erläutert wurde, eingebaut.

```
#include <iostream.h>
#include "HW_Object_i.h"
#include "AttributeValue.h"

//ctor
HW_Object_i::HW_Object_i(char *a, char *b) {
}

//dtor
HW_Object_i::~HW_Object_i() {
}

char * HW_Object_i::HW_address (CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
char *wert;
wert = getAttributeStringValue(catHHN, caidSBMhwoHHNo);
return wert;
}

void HW_Object_i::admin_state (const char * admin_state, CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
}

char * HW_Object_i::admin_state (CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
char *wert;
wert = getAttributeStringValue(catINT32U, caidSBMhwoStateAdmin);
return wert;
}
void HW_Object_i::change (const char * adstate, CORBA::Environment &IT_env)
                throw (CORBA::SystemException) {
setAttributeStringValue(catINT32U, caidSBMhwoStateAdmin, adstate);
}
```

Abbildung 5-10: Implementierungs-Datei “HW_Object_i.cc”

In der Datei “HW_Object_i.cc” (Abbildung 5-10) werden dagegen zu Beginn drei Header-Dateien eingebunden. Die ersten beiden Dateien müssen nicht weiter erläutert werden, die Datei “AttributeValue.h” (Abbildung 5-11) dagegen schon.

Sie enthält die Prototypen für das Auslesen und Setzen der Attributwerte des Xpress. Zusätzlich werden einige Dateien eingebunden, die für die Realisierung des Stützprozesses (siehe Kapitel 5.5) notwendig sind.

```
#ifndef AttributeValueh
#define AttributeValueh

#include <iostream.h>
#include <iomanip.h>

#include <string.h>
#include <cabo.h>
#include <crel.h>
#include <gpid.h>
#include <gdssv.h>
#include <gamsv.h>
#include <gmid.h>
#include <csocket.h>
#include <cemap.h>
#include <cstms.h>

char* getAttributeStringValue(TyAttributeTypes attribtype, TyAttributeID attribid);

void setAttributeStringValue(TyAttributeTypes attribtype, TyAttributeID attribid,
                            const char* attribvalue);

#endif
```

Abbildung 5-11: Headerdatei “AttributeValue.h”

In der Datei “HW_Object_i.cc” (Abbildung 5-10) werden noch der Konstruktor und der Destruktor ergänzt und für die Attribute und Methoden die jeweiligen C++-Aufrufe eingebaut. Diese stellen wieder eine Besonderheit dar, da sie nicht direkt den Hicom Xpress ansprechen, sondern erst den Stützprozeß. Bei einer normalen CORBA-Realisierung, greifen diese Aufrufe dagegen sofort auf das jeweilige System zu und weitere Arbeiten hinsichtlich der Systemaufrufe sind nicht notwendig. Hätte man also einen CB mit einem lauffähigem ORB, kann das folgende Kapitel 5.5 übersprungen werden. Der Grund ist, daß im nächsten Kapitel nur die notwendigen Schritte für die Realisierung mit dem Stützprozeß dargestellt werden. In diesem Zusammenhang werden auch die Systemaufrufe, die in Abbildung 5-10 ergänzt wurden, genauer vorgestellt.

5.5 Stützprozeß für die Realisierung des CORBA-Agenten

Wie im Kapitel 5.3 schon angesprochen, gibt es für das *Unixware* Betriebssystem keine Unterstützung von der Firma IONA. Da eine Portierung der Xpress-Software auf ein anderes Betriebssystem nicht möglich ist, muß eine andere Variante gefunden werden. Grundsätzlich ist die Kommunikation innerhalb des Xpress mit 'Messages' aufgebaut, die zwischen den jeweiligen Managern verschickt werden. Die Manager sind z.B. der *SwitchManager* oder der *APTM2BoardManager*. Möchte man nun Attributwerte von diesen Managern erhalten, muß eine 'Message' mit den gewünschten Request-Daten angelegt werden und an den jeweiligen Manager geschickt werden. Der Request des Managers wird ebenfalls in einer 'Message' zurückgeschickt, die nun z.B. von der XAD ausgewertet wird. Eine zweite Möglichkeit zur Auswertung dieser 'Messages' stellt der SNMP-Manager dar, der über einen SNMP-Proxy mit dem CB des Xpress verbunden ist (siehe Abbildung 3-7).
Damit ist klar, daß für die CORBA-Realisierung ebenfalls ein Proxy nötig ist, der die gleichen Aufgaben wie der SNMP-Proxy übernimmt. Der CORBA-Proxy wird dabei auf dem CB des Xpress mittels der UNIX-Domain-Sockets realisiert. Somit ist man jetzt zumindest in der Lage an der Kommunikation innerhalb des Xpress teilzunehmen. Jedoch stehen diese Informationen bisher nur auf dem CB zur Verfügung, wo kein ORB vorhanden ist. Deshalb muß nun im zweiten Schritt noch die Übertragung der Daten vom CB auf den Rechner, der einen lauffähigen ORB hat, gelöst werden. In der Testumgebung ist dies eine Workstation mit *Solaris 2.5*.
Dazu wird ein Stützprozeß geschrieben, der über INET-Sockets, die gewünschte Kommunikation mit dem Proxy ermöglicht. Um diese Informationen jetzt im implementierten Server verwenden zu können, muß in den Implementierungsklassen des Prototyps die Umsetzung auf die Xpress-Klassen realisiert werden.
Um aber trotzdem die Fähigkeit von *Orbix* aufzeigen zu können, wird der Xpress nicht von der Workstation administriert, sondern von einem PC mit *Windows NT 4.0*. Auf diesem Rechner soll später auch die graphische Oberfläche realisiert werden. Diese neue Situation für die Implementierung ist in Abbildung 5-12 graphisch beschrieben. Um den dort dargestellten CORBA-Agenten zu realisieren, werden für den Stützprozeß zwei Methoden benötigt. Die erste Methode fragt den Wert eines Attributes und die zweite Methode weist auf der anderen Seite einem Attribut einen Wert zu. Dazu werden die Methoden "getAttributeStringValue.cc" und "setAttributeStringValue.cc" eingeführt, die als C++-Systemaufrufe auch schon in der Datei "HW_Object_i.cc" vorgekommen sind.

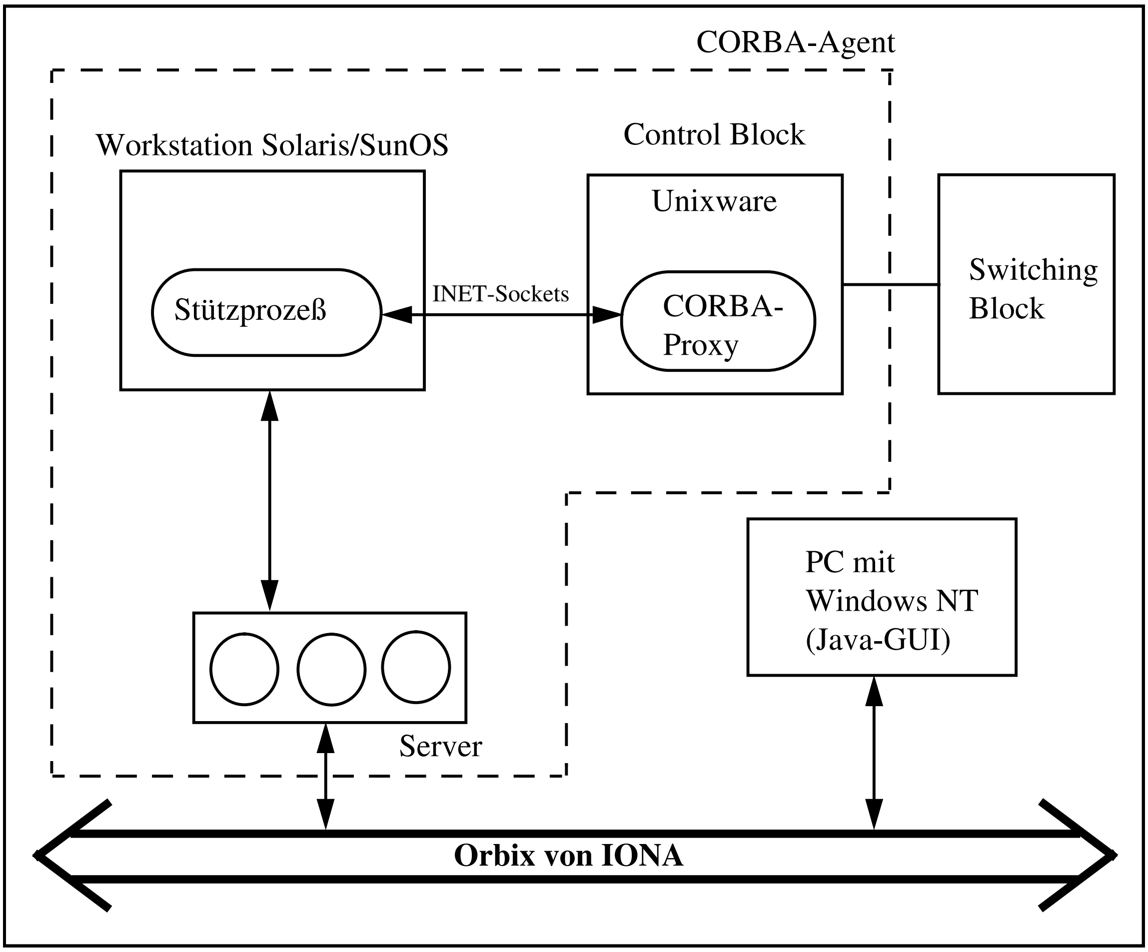


Abbildung 5-12: Realisierung des CORBA-Agenten

Die Methode "getAttributeStringValue" dient dazu, einen Attributwert vom Xpress abzufragen. Um dies realisieren zu können, werden mehrere Arbeitsschritte benötigt. Betrachtet man den Funktionsaufruf in der Datei "HW_Object_i.cc" (Abbildung 5-10), so werden dort zwei Variable an die Methode übergeben. Es ist der Typ des Attributs und die ID des Attributs. Diese beiden werden zu einem späteren Zeitpunkt innerhalb der Methode benötigt. Um diese Auswertung möglich zu machen, sind noch Vorarbeiten zu leisten.

Im ersten Schritt muß eine 'Message' angelegt werden, die eine Prozeß-ID, eine Manager-ID und eine Adressierung für die Objekte des Managers besitzt. Daneben enthält die 'Message' noch Informationen darüber, welcher Proxy angesprochen werden soll, und ob Informationen ausgelesen oder geändert werden sollen. Der Aufbau einer solchen 'Message' sieht wie folgt aus:

*CLMessage vmsgMsg(ClLogObjRef(cpidSBMCB, cmidAPTM2BoardManager,**0**),*
ClLogObjRef(cpidCORBAProxy,0,0), cevAMRequest, ***cAMsvReadNext****);*

In diesem Fall wird eine ‘Message’ an den *APTM2BoardManager* geschickt und alle Objektwerte, die im Manager enthalten sind, ausgelesen. Sollen dagegen z.B. nur die Werte des vierten *APTM2Boards* ausgelesen werden, hat die ‘Message’ folgenden Aufbau:

*CLMessage vmsgMsg(ClLogObjRef(cpidSBMCB, cmidAPTM2BoardManager,**4**),*
*ClLogObjRef(cpidCORBAProxy,0,0), cevAMRequest, **cAMsvRead**);*

Diese Adressierung der einzelnen Objekte ist nur bei *cAMsvRead* und bei *cAMsvChange* zulässig, bei *cAMvsReadNext* muß die “0” für die Adressierung verwendet werden. Im nächsten Schritt wird über diese ‘Message’ eine ‘strukturierte Message’ gelegt, durch die man in der Lage ist, die Werte der Attribute auszulesen oder zu verändern. Dies ist bei der ursprünglich erzeugten ‘Message’ nicht möglich.
Um nun die Daten beim Xpress abfragen zu können, wird die ‘Message’ mit der Funktion *fWrite2Socket(&vmsgMsg)* an den CB geschickt. Anschließend wird mit dem Aufruf *fReadMsg()* die angeforderte ‘Message’ wieder abgeholt. Bei diesem Aufruf wird nun ein Zeiger *(pmsgDummy)* auf die ‘Message’ zurückgegeben. Für den Zugriff und das Auslesen der Werte wird mit folgender Zeile eine ‘strukturierte Message’ erzeugt:

*ClStructuredMessage wstmsStms (*pmsgDummy, vecError*)

Um die Werte von dieser ‘Message’ auslesen zu können, steht folgender Funktionsaufruf zur Verfügung:

(AttributeArray[attribtype])->fGetFromStmsToASCII (attribid,
wstmsStms,
InfoPtr,
AttributeArray[attribtype]->fGetASCIISize(),
TempLength);

Dabei werden auch der Typ des Attributs und die ID des Attributs, die zuvor beim Aufruf der Funktion *getAttributeStringValue()* übergeben wurden, verwendet. Als Ergebnis erhält man den Wert des gewünschten Attributs. Wichtig ist dabei, daß er als ASCII-String vorliegt und nicht im Format des jeweiligen Attributs. Deshalb war es auch notwendig, in den IDL-Dateien die Datentypen für alle Attribute und Methoden zu verändern. Zum Schluß dieser Methode muß der Pointer auf die ‘strukturierte Message’ wieder gelöscht werden, da es sonst beim nächsten Aufruf zu Problemen kommen könnte.

Betrachtet man die zweite Methode zum Setzen von Attributwerten, so erkennt man, daß sie dem Aufbau der vorherigen Methode in etwa entspricht. Auch bei dieser werden der Typ des Attributs und die ID des Attributs, aber zusätzlich noch der neue Wert des Attributs übergeben.
Zuerst wird auch wieder eine 'Message' mit den gewünschten Werten angelegt:

*CLMessage vmsgMsg(ClLogObjRef(cpidSBMCB, cmidAPTM2BoardManager,**4**),*
*ClLogObjRef(cpidCORBAProxy,0,0), cevAMRequest, **cAMsvChange**);*

Im Anschluß daran wird eine 'strukturierte Message' erzeugt und dann der Wert des Attributs in dieser 'Message' verändert:

ClStructuredMessage vstmsStms (vmsgMsg, vecError)

(AttributeArray[attribtype])->fAddASCIIToStms(attribid, vstmsStms, attribvalue);

Zum Abschluß der Methode wird diese veränderte 'Message' mit *fWrite2Socket(&vmsgMsg)* an den CB geschickt. Hat dies ohne Probleme funktioniert, ist der Wert des angebenen Attributs im ausgewählten Manager geändert worden. Zur Kontrolle wäre eine Abfrage mit der Funktion *getAttributeStringValue* möglich.
Damit sind die Funktionsaufrufe beschrieben, die notwendig sind, um lesend oder schreibend auf den Hicom Xpress zugreifen zu können. Voraussetzung dazu ist natürlich, daß eine Verbindung zwischen dem Server und dem Stützprozeß besteht. Dafür werden noch die zwei Funktionen *connect_to()* und *disconnect_from()* geschrieben, die dann im Server des CORBA-Agenten eingebaut werden müssen (siehe Kapitel 5.6).

5.6 Realisierung eines Servers und eines Clients

Nach Abschluß dieser Arbeit muß noch ein Client und ein Server realisiert werden. Zur Vereinfachung wird ein Server und ein Client beschrieben, der als Basis den Ausschnitt aus der IDL-Datei in Abbildung 5-5 verwendet.
Der Server hat in dem vorliegenden Fall wegen der besonderen Implementierung mit dem Stützprozeß zwei Aufgaben. Einerseits die Aufgaben, die bei jeder CORBA-Implementierung vorhanden sind und andererseits die Anbindung an den Stützprozeß.
Für den Stützprozeß wird zunächst der gewünschte ATM-Switch festgelegt und anschließend mit dem Aufruf *connect_to(pcsHost)* eine Verbindung hergestellt. Ab diesem Zeitpunkt ist nun die Kommunikation zwischen dem Client und dem CB möglich. Zum Abbau der Verbindung wird die Funktion *disconnect_from(pcsHost)* verwendet. In der Abbildung 5-13 sind die notwendigen Ergänzungen wegen des Stützprozesses farbig dargestellt.

```
#include <iostream.h>
#include <stdlib.h>
#include "HW_Object_i.h"
#include <connect.h>
EXPORT ClEnumMap EnumMap;
EXPORT ClEnumMap * pEnumMap = &EnumMap;
EXPORT INT32U fGetProcID (void) {
return cpidCORBAProxy; }

int main() { // create a HWObject - using the implementation class HW_Object_i
 HW_Object_i  myHWObject("","");
 // Build the connection to the switch
 char * pcsHost="atm_u2";
 connect_to(pcsHost);
 try { // tell Orbix that we have completed the server`s initialisation
 CORBA::Orbix.impl_is_ready("HW_Object");
 }
 catch (CORBA::SystemException &sysEx) {
 cerr << "Unexpected system exception" << endl;
 cerr << &sysEx;
 exit(1);
 }
 catch (...) { // an error occured calling impl_is_ready() - output error
 cout << "Unexpected exception" << endl;
 exit(1);
 }
 // impl_is_ready() returns only when Orbix times_out an idle server or an error occurs.
 cout << "server exiting" << endl;
 disconnect_from(pcsHost);
 return 0;
}
```

Abbildung 5-13: CORBA-Server “HWSrv_Main.cc”

Der andere Teil des Servers muß bei jeder Implementierung vorhanden sein. In diesem wird zuerst mit Hilfe der Implementierungklasse ein neues Objekt der Klasse `HW_Object` erzeugt. Ist dieses Objekt initialisiert, wird mit dem Aufruf

CORBA::Orbix.impl_is_ready("HW_Object");

angezeigt, daß die Initialisierung abgeschlossen ist und der Server auf Anfragen von Clients wartet. Die weiteren Schritte im Server dienen dann nur noch der Fehlerbehandlung. Um diesen Server jetzt noch verwenden zu können, muß der compilierte und gebundene Server im *Implementation Repository* mit dem putit-Kommando registriert werden.
Um diesen registrierten Server für die Klasse `HW_Object` verwenden zu können, ist ein lauffähiger Client, der keine speziellen Realisierungen wegen des Stützprozesses mehr hat, nötig.

Dazu wird im ersten Schritt ein Zeiger auf die verwendete `HW_Object`-Klasse erzeugt. Dies ist möglich, da der IDL-Compiler für jeden Typ einer Schnittstelle einen "_var"-Typ generiert. Dieser Typ wird als eine Art abstrakter Pointer betrachtet. Danach wird mit der Funktion "_bind" *Orbix* dazu veranlaßt, ein Objekt auf dem angegebenen Host zu suchen, das eine Schnittstelle für die Klasse `HW_Object` zur Verfügung stellt. Sind dabei keine Fehler aufgetreten, wird im Beispiel-Client (Abbildung 5-14) zuerst die Methode change aufgerufen, um das Attribut admin_state zu verändern. Im Anschluß daran folgen zwei Aufrufe, die die Werte des Attributs HW_address und admin_state auslesen und nach einer Fehlerüberprüfung am Bildschirm ausgeben. Compiliert und bindet man diesen dargestellten Client mit den anderen Komponenten, kann der Client in diesem Beispiel mit dem Aufruf "*HWClient ipx14*" gestartet werden. An diesem Aufruf erkennt man, daß der Server auf dem Rechner mit dem Namen "*ipx14*" registriert ist, d.h dort befindet sich auch das *Implementation Repository* mit dem registrierten Server.
Um den Prototypen vollständig fertigstellen zu können, sind alle Klassen des Objektmodells nach dem in Kapitel 5 beschriebenen Weg implementiert worden. Zusätzlich wurde der angegebene Server so erweitert, daß er nicht nur die Klasse `HW_Object` kennt, sondern alle implementierten Klassen aus dem Objektmodell Version 3 (siehe Anhang A).
Ein Client, wie er in Kapitel 5.6 beschrieben wurde, ist aber nicht implementiert worden, da das Management des ATM-Switches über eine graphische Oberfläche erfolgen sollte und nicht über ein Kommandozeileninterface. Die Realisierung des Clients mit der graphischen Oberfläche ist im Kapitel 6 beschrieben.

```
#include <iostream.h>
#include <stdlib.h>
#include "HW_Object.hh"

int main(int argc, char **argv) {
HW_Object_var HWObjectVar;          //pointer the HWObject class that will be used
char *address;
char  *astate;

 if (argc < 2) {
     cout << "usage: " << argv[0] << " <hostname>" << endl;
     exit (-1);
}
 try { // First bind to the HWObject object
     // argv[1] has the hostname of the target HWObject object

     HWObjectVar = HW_Object::_bind("", argv[1]);
 }
catch (CORBA::SystemException &sysEx) {
     cerr << "Unexpected system exception" << endl;
     cerr << &sysEx;
     exit(1);
}
 catch (...) { // an error occured while trying to bind to the HWObject object
     cerr << "Bind to object failed" << endl;
     cerr << "Unexpected exception" << endl;
     exit(1);
}
 try {
    cout  << "try to change the admin_state of the switch" << endl;
    HWObjectVar->change("0000000001");
    cout << "try to read the attributes of the switch" << endl;
    address = HWObjectVar->HW_address();
    astate = HWObjectVar->admin_state();}
     catch (CORBA::SystemException &sysEx) {
     cerr << "Unexpected system exception" << endl;
     cerr << &sysEx;
     exit(1);
}
catch (...) { // an error occured while trying to read the attributes
     cerr << "call to attributes failed" << endl;
     cerr << "Unexpected exception" << endl;
     exit(1); }
 //no problem reading attributes
cout << "HW_address: " << address << endl;
cout << "Admin_state: " << astate << endl;
 return 0;
}
```

Abbildung 5-14: CORBA-Client “HWClient.cc”

5.7 Erfahrungsbericht zur Implementierung

Performance
Durch die Besonderheiten der Implementierung ist es nicht so einfach, eine Bewertung bzw. Beurteilung der Lösung vorzunehmen. Betrachtet man die Performance des Kommunikationsmechanismuses, so wird schnell deutlich, daß die jetzt realisierte Lösung nicht voll aussagekräftig ist. Hätte man den Prototypen ohne den zusätzlichen CORBA-Proxy und den Stützprozeß implementiert, wäre die notwendige Kommunikation direkt im CB des Xpress abgelaufen, d.h. der Client hätte sofort den Server im CB adressieren können. Da aber der ORB auf *Unixware* nicht zur Verfügung stand, war eine unmittelbare Kommunikation mit dem CB nicht möglich, sondern es mußte der Umweg über den Stützprozeß gemacht werden. Somit mußte der Client einen Server auf einer Maschine ansprechen, der einen lauffähigen ORB besitzt. Dieser hat dann den Stützprozeß aufgerufen und erst daraufhin wurde die Kommunikation mit dem CB des Xpress gestart. Damit wird klar, daß eine Bewertung nicht so einfach ist. Was man jedoch sagen kann ist, daß die im Prototypen realisierte Kommunikation durchaus akzeptabel ist und bei den Tests keine Schwachstellen aufgewiesen hat.

Codegröße und Compilierzeit
Betrachtet man die Codegrößen, so stellt man fest, daß durch den Stützprozeß ca. 20 MB zusätzlicher Code nötig war. Die Installation des verwendeten *Orbix* umfaßte ca. 60 MB, wobei allein das Unterverzeichnis "/bin" 44MB benötigte. Im Gegensatz dazu benötigt *OrbixWeb*, das auf dem Windows NT-Rechner eingesetzt wird, nur ca. 25 MB. Betrachtet man die Größen der implementierten Klassen, so kann man sagen, daß sich die Codegröße für jede Klasse abhängig von der Anzahl der Attribute und Methoden zwischen 1,5 MB und 2,5 MB bewegt.
Eine Abschätzung der Zeit für die Compilierung ist nicht so einfach, da sie im vorliegenden Fall auch durch die Belastung des Rechners immer unterschiedlich war. Man kann jedoch sagen, daß der Compiliervorgang erheblich schnell durchgeführt wurde, nachdem die notwendigen Dateien für den Stützprozeß übersetzt und auch gebunden waren. Dieser Vorgang für den Stützprozeß dauert circa sechs Minuten. Betrachtet man die einzelnen Klassen aus dem Objektmodell, so kann man sagen, daß sich die Compilier- und Bindezeit meistens zwischen einer und zwei Minuten bewegt hat.

Wiederverwendbarkeit des Codes
Die Wiederverwendbarkeit des Codes ist bei der jetzt vorliegenden Realisierung nicht ohne Probleme möglich. Jedoch dürfte sich der Änderungsaufwand in angemessenen Grenzen halten, da nur die Anbindung an den Switch geändert werden muß. Die IDL-Dateien, insgesamt wurden 18 IDL-Dateien mit StP erzeugt, und die durch den IDL-Compiler generierten Dateien (mit Ausnahme der zwei zusätzlichen Dateien für die je-

weiligen Implementierungsklassen) können ohne Veränderungen wieder eingesetzt werden. Falls trotzdem bei den IDL-Dateien Ergänzungen oder Änderungen, die durch mögliche Anpassungen im Objektmodell verursacht werden, auftreten, wird in diesem Punkt kein höherer Aufwand verursacht. Nur in den Implementierungsklassen und im Client müssen Anpassungen vorgenommen werden, falls sich Attribute oder Methoden verändert haben.
Beim implementierten Server müssen nur bei der Einführung neuer Objektklassen Anpassungen vorgenommen werden. Damit bietet diese Realisierung gute Möglichkeiten, für den Einsatz bei anderen ATM-Switches, da der Änderungsaufwand nicht sehr hoch ist. Voraussetzung dazu ist natürlich, daß die CORBA-Realisierung vollständig durchgezogen wird und nicht doch wieder proprietäre Methoden dazwischen geschaltet werden.

Probleme und Fehlersuche
Durch die Verwendung des Stützprozesse war es auch notwendig, die Makefiles für die CORBA-Implementierung und den Stützprozeß miteinander zu verbinden, um so die Compilierungsarbeit bei der Implementierung zu vereinfachen. Dies funktionierte nach anfänglichen Schwierigkeiten beim Verbinden der Makefiles auch ohne Probleme. Allgemein kann man sagen, daß sich die Fehlersuche im Rahmen der Arbeit doch in Grenzen gehalten hat. Probleme sind nur bei den notwendigen Veränderungen der Implementierungsklassen aufgetreten, da hier in der Dokumentation von *Orbix* keine Unterstützung zu finden war und bei der Anbindung des Stützprozesses, da für einen Aussenstehenden die ‘Message’-Realisierung nicht sofort durchsichtig ist und viele Kleinigkeiten zu beachten sind.

Als Fazit der Implementierung kann man sagen, daß CORBA durchaus eine gute Möglichkeit zum Management eines ATM-Switches bietet. Außerdem erlaubt die Technologie eine schnellere Anpassung an Neuerungen oder Veränderungen. Zusätzlich ist man in der Lage, von jedem beliebigen Rechner in einem Netzwerk, der eine ORB installiert hat, auf den ATM-Switch zuzugreifen. Damit wird die Fehlerbehebung und Wartung meiner Meinung nach deutlich vereinfacht.

5.8 Zusammenfassung zum Kapitel 5

Betrachtet man das fünfte Kapitel rückblickend, so kann man mehrere Arbeitsschritte erkennen. Zuerst wurde aus dem in Kapitel 4 entstanden Objektmodell bzw. aus den dort modellierten Klassen, der jeweilige IDL-Code mit StP generiert.
Im Anschluß daran wurden die entstanden IDL-Dateien an die Situation bezüglich der nicht vorliegenden Software (kein *Orbix* für *Unixware*) angepaßt und schließlich mit dem IDL-Compiler von IONA übersetzt.
Damit entstanden neben Header-Dateien und Client- bzw. Server-Stubs auch sogenannte Implementierungsdateien. In diesen mußte der vorhandene Code für die Systemaufrufe eingebaut werden. Da für den CB kein ORB von der Firma IONA zur Verfügung stand, mußte zusätzlich noch ein Stützprozeß für die Anbindung an den CB geschrieben werden. Dieser Proxy ermöglichte die Kommunikation zwischen dem CB und dem Server, der im Anschluß an diese Arbeit implementiert werden mußte. Nachdem auch dies geschehen war, mußte ein Client implementiert werden, der die gewünschten Informationen vom implementierten Server abfragt. Da dies im vorliegendem Fall nur über ein Kommandozeileninterface möglich ist, wird im Kapitel 6 eine graphische Oberfläche entworfen, um so die Benutzerfreundlichkeit für die Anwender zu steigern. Diese graphische Oberfläche wird dann dabei die Rolle des Clients übernehmen, wodurch die Kommunikation zwischen der graphischen Oberfläche und dem Server ebenfalls über einen ORB realisiert werden muß.

6 Realisierung einer graphischen Oberfläche

Im vorliegenden Kapitel wird die graphische Oberfläche für den CORBA-Agenten entworfen und realisiert. Nach der Vorstellung des Konzepts im Kapitel 6.1, wird im Kapitel 6.2 das vorhandene Szenario für den Managementagenten und die Oberfläche vorgestellt. Bevor im Kapitel 6.4 näher auf die Implementierung eingegangen wird, stellt das Kapitel 6.3 die eingesetzten Softwarepakete vor.

6.1 Konzept für ein Java-GUI

Um das Management des ATM-Switches Hicom Xpress benutzerfreundlicher zu gestalten, wird eine graphische Oberfläche realisiert. Diese soll nicht völlig neu entwikkelt werden, sondern auf Basis des Tools für die Administration des Hicom Xpress entstehen. Wichtig ist dabei, daß nur die Gestaltung der Oberfläche teilweise übernommen werden soll und nicht die Realisierung der Anbindung.
Bei der graphischen Oberfläche der XAD gibt es insgesamt drei Hauptfenster, das *Processing Window* (PW), das *View Window* (VW) und das *Modify Window* (MW). Die Aufgaben der jeweiligen Fenster und ihre Funktionen sind in Kapitel 3.2 vorgestellt worden. Dabei ist aufgefallen, daß im VW keine Änderungen möglich sind und man extra in das MW wechseln muß, um Attribute zu verändern. Da dies bei vielen geöffneten Fenstern unübersichtlich wird, gibt es beim neuen Entwurf nur noch das PW in der alten Form. Das VW und das MW werden dabei in einem gemeinsamen Fenster zusammengefaßt.
Einstiegspunkt für die Administration wird wieder das XAD Main Window, wobei die Buttons an das entworfene Objektmodell angepaßt werden. Durch die Aktivierung eines Buttons wird das zugehörige Klassenfenster (PW) geöffnet.
Der Aufbau dieser Fenster wird soweit wie möglich von der XAD übernommen, d.h. es werden nur die Änderungen im Objektmodell beim Aufbau der jeweiligen Fenster berücksichtigt. Aktiviert man in einem dieser Fenster einen Tabelleneintrag, wird ein weiteres Fenster geöffnet, das zusätzliche Informationen zu dem ausgewählten Objekt enthält. Dies entspricht im Aufbau dem bisherigen VW mit den jeweiligen Anpassungen. Um in diesem Fenster erkennen zu können, welches Attribut verändert werden kann, wird dieses mit einer anderen Farbe dargestellt. Dadurch erkennt der Administrator sofort, welche Attribute verändert werden können. Ein schreibender Zugriff auf die anderen Attribute ist nicht möglich. Das Fenster zum Neuanlegen von Objekten wird dagegen nur die Attribute enthalten, die zum Anlegen des jeweiligen Objekts notwendig sind. Zu beachten ist, daß bei der Realisierung des Prototypen nicht alle Feinheiten der bisherigen XAD übernommen werden, d.h. manche Eigenschaften, wie die Statuszeile oder die Filterfunktion, werden im Prototypen nicht realisiert.

Nachdem bei der Gestaltung der Oberfläche keine großen Veränderungen gemacht werden, wird die Anbindung der Oberfläche an den Xpress auf eine andere Art und Weise realisiert.
Wie im Kapitel 5 vorgestellt, ist die Implementierung des Servers für das Management des Xpress CORBA-basiert, d.h. die Kommunikation zwischen dem Client und dem Server geschieht über einen ORB.
Da die graphische Oberfläche die Rolle des Clients übernimmt, muß die Kommunikation mit dem CB des Xpress auch jetzt mittels eines ORBs realisiert werden. Damit steht man vor dem Problem, mit welcher Programmiersprache man die graphische Oberfläche am Besten realisiert.
Denn der Aufwand für die Gestaltung und Implementierung der Oberfläche selber, als auch die Anbindung an den ORB sollte sich im Rahmen halten. Betrachtet man unter diesen Bedingungen mehrere Programmiersprachen, so kristallisiert sich Java als Lösung für dieses Problem heraus. Gründe sind das vorhandene *Language Mapping* bei CORBA und die vorhandenen Java-Klassen für die Realisierung von graphischen Oberflächen.
Damit ist nun eine Entscheidung bezüglich der eingesetzten Technologien getroffen, aber noch nicht darüber, welche Produkte dabei zur Verfügung stehen. Beginnt man bei der Anbindung an den ORB und das *Language Mapping* für Java, so findet man sehr schnell in dem Produktangebot von IONA Technologies eine Lösung.
IONA bietet unter dem Namen *OrbixWeb* eine Implementierung des *Orbix* für die Programmiersprache Java an, d.h. die Basis für die Implementierung des *OrbixWeb* bildete *Orbix*, der im Kapitel 5 für die Implementierung eingesetzt wurde. Der Unterschied liegt nur darin, daß er von IONA explizit für die Programmiersprache Java entwickelt wurde. Damit ist dieser ORB in einer Java-Umgebung voll einsetzbar und der IDL-Compiler liefert in diesem Fall auch sofort Java-Code und nicht wie bei *Orbix* C++-Code. Eine kurze Einführung zu *OrbixWeb* ist in [ORW97] enthalten. Um diesen ORB nun nutzen zu können, wird ein weiteres Entwicklungstool benötigt, das bei der Gestaltung der Oberfläche eingesetzt werden kann. Voraussetzungen für dieses Produkt sind: Verwendung von Java, möglichst einfache Verfahren zur Gestaltung der Oberfläche und die Möglichkeit zur Anbindung an CORBA. Ein Produkt, das all diese Eigenschaften zur Verfügung stellt, ist der *JBuilder* der Firma Borland. Zu beachten ist, daß dieses Produkt nur auf den Plattformen *Windows95* und *Windows NT 4.0* eingesetzt werden kann. Doch dies ist für die Realisierung des Prototypen kein Nachteil, da damit die Kommunikation über den ORB noch besser dargestellt wird.
Damit ist die Entwicklungsumgebung für die Realisierung der graphischen Oberfläche komplett. Bevor jedoch mit der Implementierung begonnen wird, soll nochmals kurz das Einsatzszenario dargestellt werden.

6.2 Szenario für ein Java-GUI

Nachdem das Konzept für die Realisierung der Oberfläche entworfen ist, wird die Situation für die Implementierung betrachtet. Damit wird verhindert, daß bei der Implementierung in die falsche Richtung gearbeitet wird, da man das gewünschte Ziel aus den Augen verloren hat.

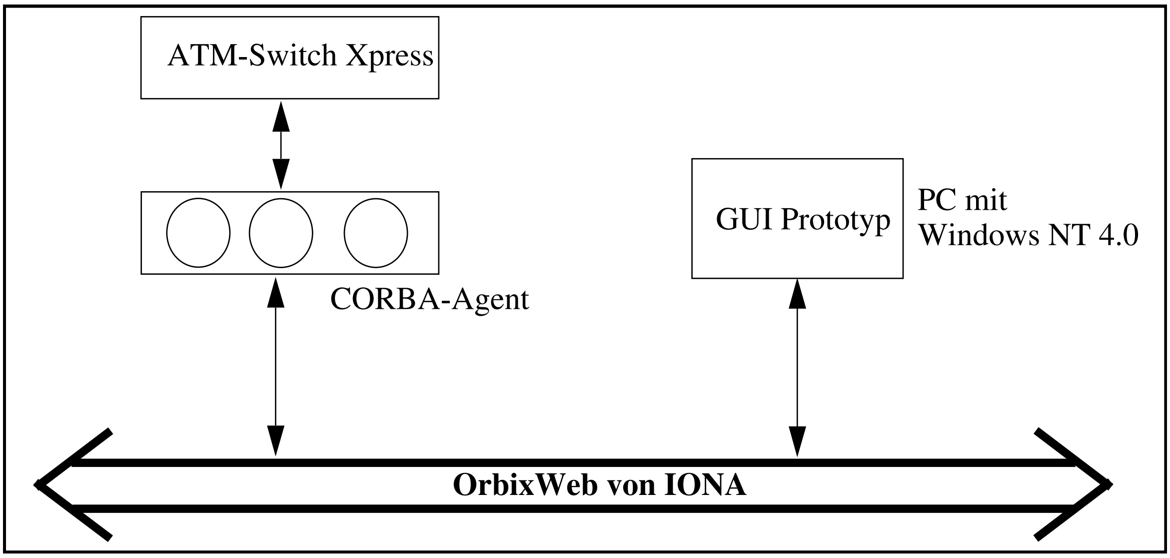


Abbildung 6-1: Einsatzszenario des GUI

Betrachtet man die Abbildung 6-1, so befinden wir uns bei der Implementierung der Oberfläche auf dem PC mit dem Betriebssystem *Windows NT 4.0*. Dort wird mit Hilfe eines Entwicklungstools ein Client entworfen und realisiert, der über *OrbixWeb* den implementierten CORBA-Agenten aufruft. Hat die Kommunikation zwischen dem GUI-Prototyp und dem CORBA-Agenten ohne Probleme stattgefunden, werden die gewünschten Informationen durch den CORBA-Agenten am Xpress abgefragt. Hat der CORBA-Agent vom Xpress alle notwendigen Daten abgefragt, werden diese wieder über den *OrbixWeb* an den Client zurückgeliefert. Der Client stellt dann diese erhaltenen Attributwerte in der entworfenen graphischen Oberfläche dar. Damit stehen sie dem Administrator zur Verfügung. Auch die Änderung von Attributen geschieht über diesen Weg.

Um bei der Realisierung der graphischen Oberfläche weitgehend von Betriebssystemen unabhängig zu sein, wird die Oberfläche als Java-Applet implementiert.

Da Java betriebssystemunabhängig eingesetzt werden kann, ist das entworfene Java-Applet auf jedem Rechner, der einen ORB installiert hat, lauffähig. Voraussetzungen dafür sind ein Java-fähiger Browser und natürlich der für das Applet notwendige Code. Zusätzlich kann es sein, daß manche Pfade zu Klassenbibliotheken angepaßt werden müssen, da die Installation nicht auf allen Rechner identisch ist.

Wird dagegen ein HTTP-Server eingesetzt, besteht sogar die Möglichkeit, von jeder beliebigen Maschine im Netz, die einen entsprechenden Java-fähigen Browser besitzt, das Applet zu starten. Damit ist der Systemadministrator in der Lage, bei entsprechender Infrastruktur sehr schnell auf den Hicom Xpress zugreifen zu können.
Dadurch wird die angestrebte Unabhängigkeit von Plattformen und Betriebssystemen, die durch die Verwendung von CORBA impliziert wurde, noch weiter unterstützt.

6.3 Werkzeuge für die Implementierung

Wie im Kapitel 6.1 erläutert, sind für die Realisierung der graphischen Oberfläche mehrere Arbeitsschritte notwendig. Nach der Installierung von *OrbixWeb* und die Einstellung der jeweiligen Pfade für die Konfigurationsdatei “Orbix.cfg” und die Klassenbibliotheken von Java, steht *OrbixWeb* zur Verfügung.
Der Start des *OrbixWeb-Daemon* und die Registrierung des Servers wird unter Windows mittels einer graphischen Oberfläche realisiert. Dadurch wird die Verwendung des *OrbixWeb* vereinfacht, da man nicht mehr wie bei *Unix* mit der Kommandozeile arbeiten muß und alle nötigen Informationen graphisch dargestellt werden. Im Fenster des *OrbixWeb-Daemon* (Abbildung 6-2) wird der Verbindungsaufbau zum ORB dargestellt und die jeweilige auftretende Kommunikation mit dem ORB. Ab dem Start des Daemon ist die Kommunikation zwischen Clients und registrierten Servern möglich.

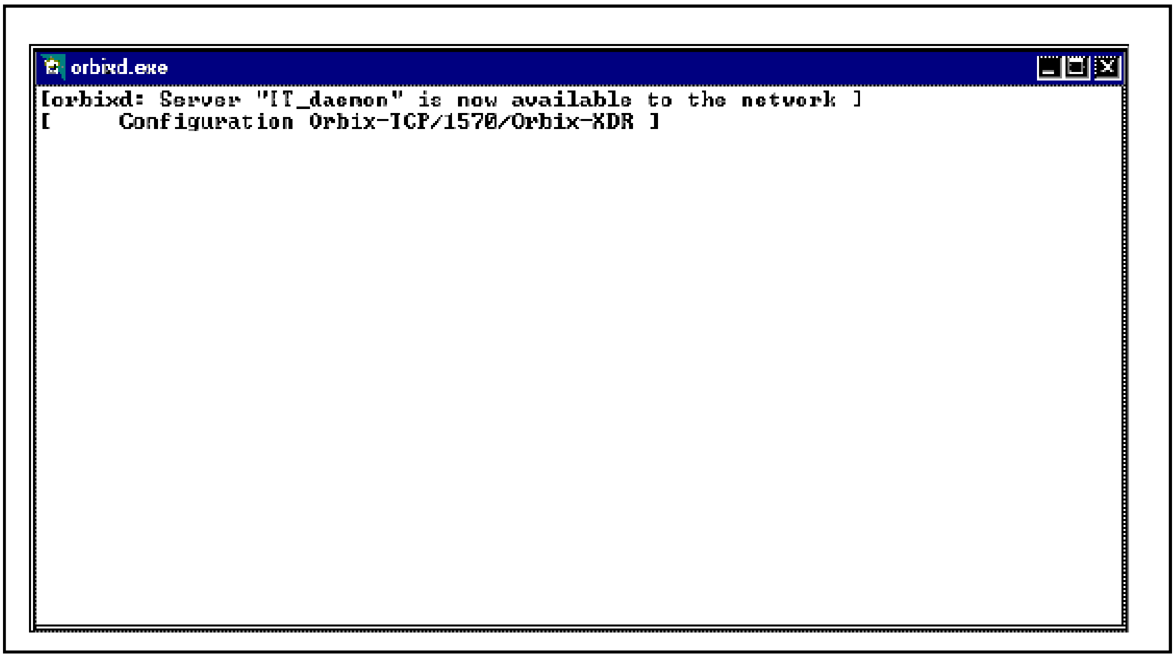


Abbildung 6-2: Fenster für den *Orbix-Daemon*

Für die Registrierung und Verwaltung der einzelnen Server steht ein eigener Server-Manager (Abbildung 6-3) zur Verfügung. Er stellt das *Implementation Repository* des *OrbixWeb* dar. Über diesen wird die benötigte Verbindung vom PC zum Unix-Rechner, auf dem der Server des CORBA-Agenten läuft, aufgebaut. Ist eine Verbindung vorhanden, wird der Server registriert und ist ab diesem Zeitpunkt von einem Client aufrufbar. Zusätzlich werden alle benötigten Informationen im *ServerManager* zur Verfügung gestellt. Vor allem die Angaben über den Namen des Servers und den Eigentümer sind wichtig, da nur so eine funktionierende Implementierung möglich ist.

Sollte der Client von einer Kennung auf einem Rechner gestartet werden, die nicht dem Eigentümer entspricht, muß in den Zeilen 'Launch' und 'Invoke' diese Kennung eingetragen werden (siehe Abbildung 6-3). Auch kann man an dem Sternsymbol in der linken Spalte neben dem Servernamen erkennen, ob der Server gestartet (drehendes Symbol) ist oder nicht.
Insgesamt kann man sagen, daß die Funktionalität, die im *ServerManager* zur Verfügung gestellt ist, die Arbeit mit dem *OrbixWeb* erheblich erleichtert.
Eine weitere verwendete *OrbixWeb*-Komponente ist der mitgelieferte IDL-Compiler. Im Gegensatz zum IDL-Compiler von *Orbix* wird in diesem Fall Java-Code erzeugt. Der Aufruf des Compilers mit **idl HW_Object.idl** erzeugt ein Unterverzeichnis "java-output", das mehrere Java-Dateien enthält. Diese werden für die weitere Implementierung der Oberfläche benötigt.

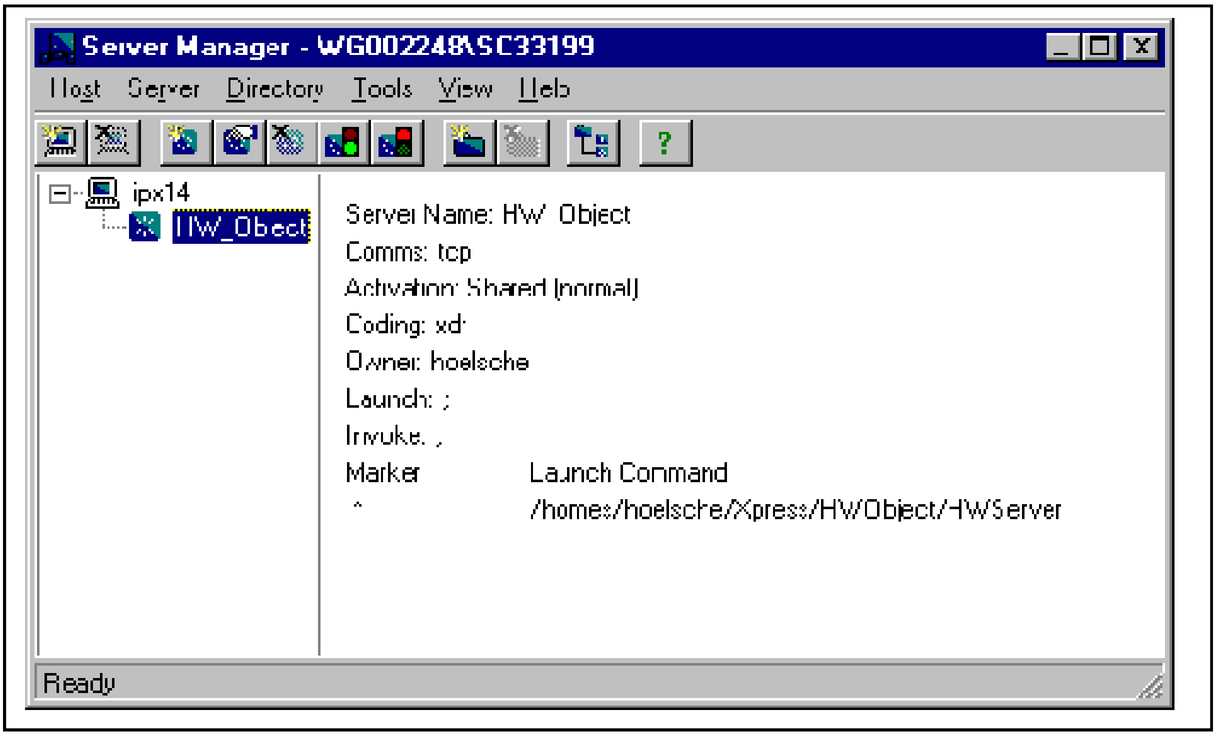


Abbildung 6-3: Fenster für den *ServerManager*

Ist die Compilierung aller IDL-Dateien abgeschlossen, kann der zweite Arbeitsschritt für die Realisierung der Oberfläche beginnen. Dazu wird nun der *JBuilder* der Firma Borland benötigt. Nach dem Aufruf des *JBuilder* wird die Entwicklungsoberfläche (Abbildung 6-4) gestartet, die die Menü- und Symbolleisten enthält.
Für die Realisierung der graphischen Oberfläche muß mit dem *JBuilder* ein neues Projekt angelegt werden, das durch den "Projekt-Experten" unterstützt wird. Dort werden die Informationen abgefragt, die vor allem für die Generierung der HTML-Seite des Projekts benötigt werden.

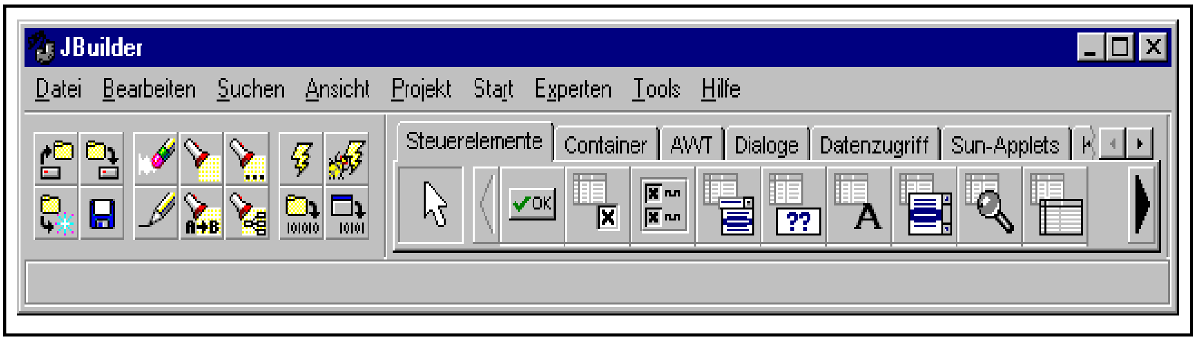


Abbildung 6-4: Entwicklungsoberfläche *JBuilder*

Danach wird dem Befehl “Neu“ des Menue “Datei“ ein neues Applet, ein Frame oder eine Anwendung erstellt. Für unsere Realisierung wählen wir das Applet und der Applet-Experte wird geöffnet. Dort werden der Name des Package und der Klassenname festgelegt und abgefragt, welche zusätzlichen Generierungsinformationen wie Headerkommentare im Quellcode eingefügt werden sollen. Außerdem können Parameter für das Applet angegeben und der Titel, der Name, die Codebase, die Breite und die Höhe für das Applet in der HTML-Seite festgelegt werden. Mit diesen Informationen generiert der *JBuilder* den Code für das Applet, und es wird das Projekt für die weitere Bearbeitung geöffnet (siehe Abbildung 6-5).

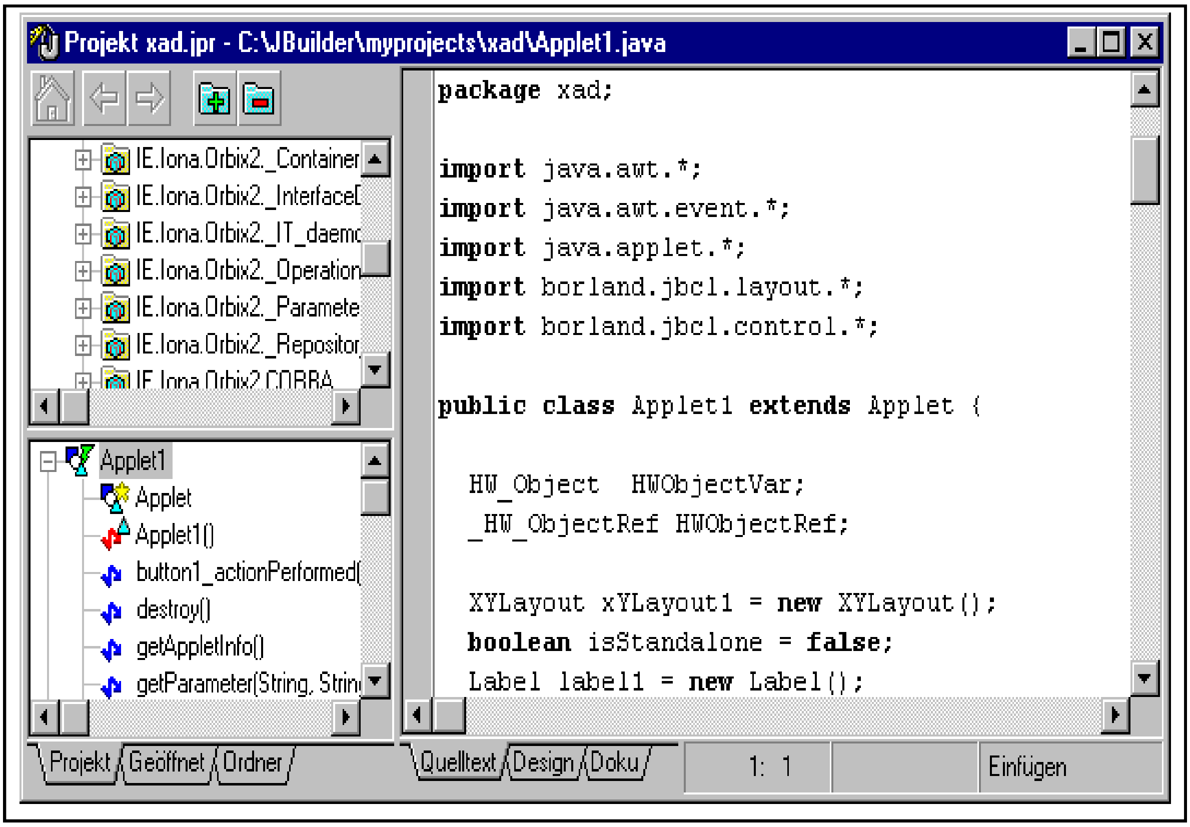


Abbildung 6-5: Projektfenster für das Applet

Bevor man jetzt mit dem *UI-Designer* die Gestaltung der Oberfläche vornimmt, muß dafür gesorgt werden, daß alle Dateien, die von den erzeugten Java-Dateien eingebunden werden, auch im Package vorhanden sind. Alle eingesetzten Dateien werden im oberen linken Fenster der Abbildung 6-5 dargestellt.
Mit den Symbolen "+" und "-" können neue Packages hinzugefügt oder nicht benötigte entfernt werden. Für die geplante Implementierung der graphischen Oberfläche müssen vor allem die Packages vom *OrbixWeb* eingebunden werden. Im Fenster darunter werden die jeweiligen im Package enthaltenen Methoden, die verwendet werden können, angezeigt. Im rechten Fenster wird schließlich der jeweilige Code der aufgerufenen Datei angezeigt. Wechselt man dort vom Register "Quelltext" auf das Register "Design", wird der *UI-Designer* gestartet (siehe Abbildung 6-6). Dort kann nun das Applet mit der in der Entwicklungsoberfläche enthaltenen Komponentenpalette erstellt werden.

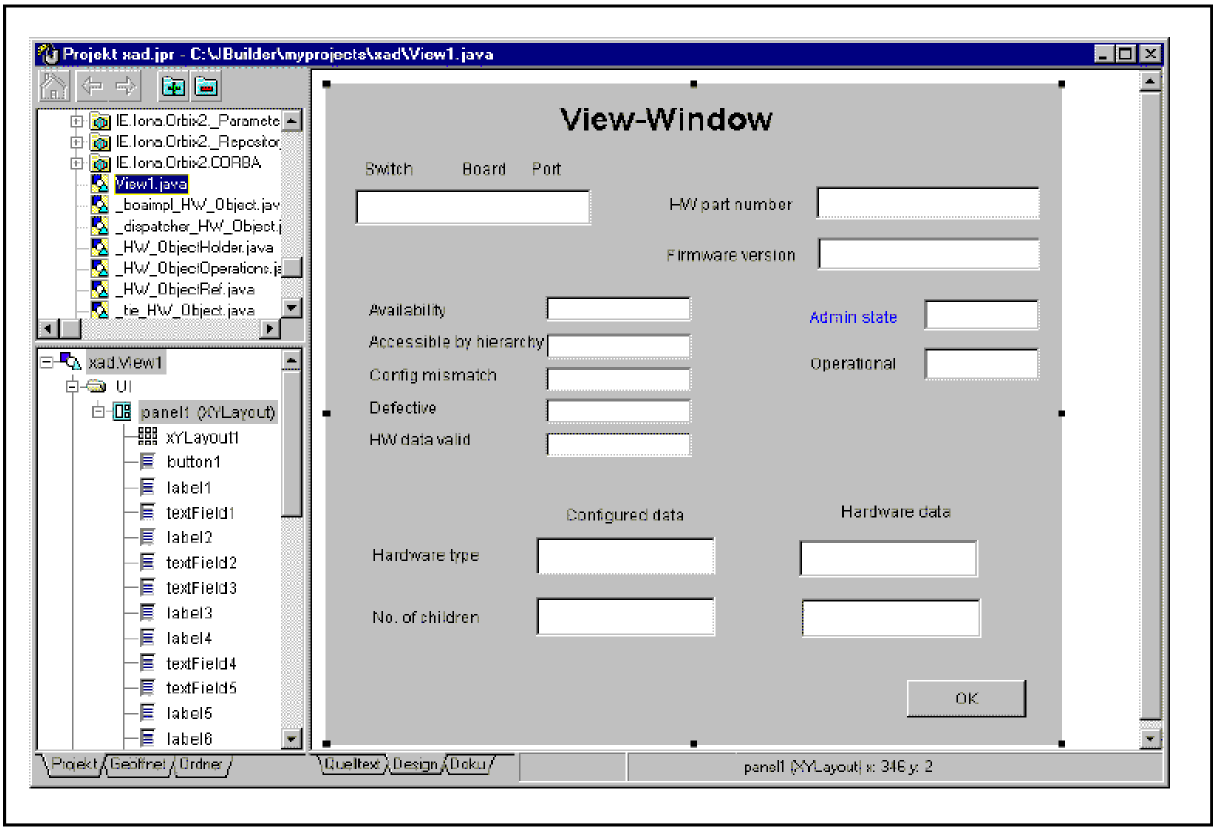


Abbildung 6-6: *UI-Designer* für das Applet

Die vorhanden Komponenten entsprechen den folgenden Klassen: `borland.jbcl.*`, `java.awt.*` und `jclass.bwt.*`.
Zusätzlich können noch weitere Klassen anderer Hersteller oder eigene in die Komponentenpalette eingebunden werden.

Möchte man z.B. einen Button erstellen, wird diese Komponente in der Palette aktiviert. Im erzeugten Applet kann jetzt mit der linken Maustaste ein Button erzeugt werden. Der dafür notwendige Quelltext wird vom *UI-Designer* automatisch in den Quelltext übernommen.
Im sogenannten *Komponenteninspektor* kann man dann für die jeweilige Komponente, wie Button oder Textfeld, bestimmte Eigenschaften (z.B. Schriftart, -und größe, Layouttyp, Farbe usw.) einstellen. Da in den meisten Fällen das Applet nicht nur aus einem Fenster besteht, müssen weitere erzeugt werden. Dafür wird z.B. über den Menuepunkt "Neu" ein Dialogfenster ausgewählt. Dieses wird dann im selben Package wie das Applet nach Festlegung des Namens eingebunden. Jetzt kann man auf die gleiche Weise, wie vorher mit dem *UI-Designer* das Fenster gestalten. Hat man schließlich mit diesem Verfahren das gewünschte Applet fertiggestellt, ist die Gestaltung der graphischen Oberfläche abgeschlossen. Jedoch müssen im nächsten Schritt alle durch die Erstellung der Oberfläche erzeugten Coderümpfe mit dem jeweils benötigten Code ergänzt werden, da sonst keine Attributwerte usw. angezeigt werden können.

6.4 Implementierung der graphischen Oberfläche

Im dritten und letzten Schritt muß der entstandene Quelltext noch nachbearbeitet werden, damit die Attributwerte usw. im jeweiligen Fenster angezeigt werden. Ebenso müssen die Aktionen für die definierten Buttons implementiert werden.
Um die Stellen, an denen Code ergänzt werden muß, zu finden, wird im *UI-Designer* z.B. ein Doppelklick auf einen definierten Button ausgeführt. Der *JBuilder* wechselt nun an die Stelle im Quelltext, die zum aktivierten Button gehört. Um im folgenden diese Implementierung beispielhaft und übersichtlich darstellen zu können, wird der Quelltext in den Abbildungen soweit reduziert, daß nur die Sequenzen dargestellt werden, die für das Verständnis notwendig sind.
Beginnt man beim Hauptfenster des Applets (siehe Abbildung 6-7), so ist für die Buttons eine Aktion zu implementieren.

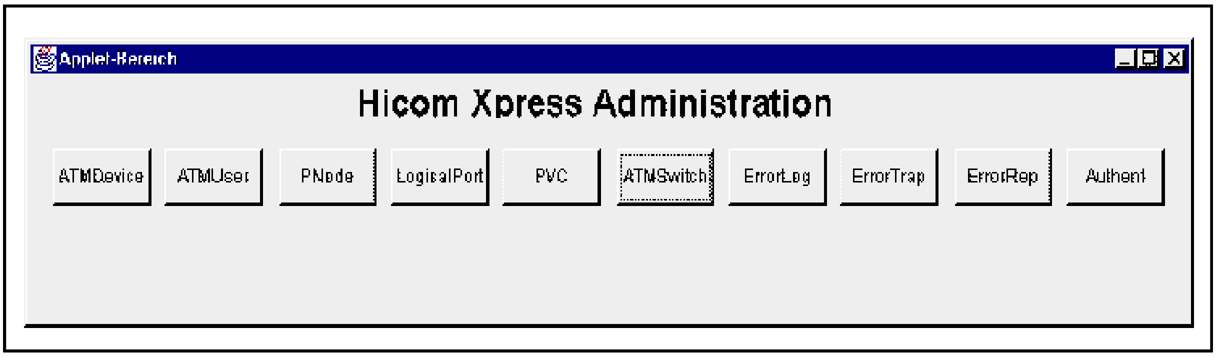


Abbildung 6-7: Startfenster des neuen Applets

Soll z.B. die Aktion für den Button “ATM-Switch“ implementiert werden, wird im *UI-Designer* ein Doppelklick auf diesen Button ausgeführt und man gelangt an die benötigte Stelle im Quelltext. Dort müssen die beiden Zeilen, die in der Abbildung 6-8 farbig dargestellt sind, ergänzt werden. Zu beachten ist, das nur ein Ausschnitt der gesamten Datei dargestellt ist.
Hat man dort die gewünschte Aktion implementiert, führt man dies für alle gewünschten Komponenten aus.
Sind alle Aktionen festgelegt, müssen noch die Attributwerte beschafft werden. Um diese vom Xpress auszulesen, werden die Methodenaufrufe verwendet, die z.B. in der Datei “HW_Object.java” durch den IDL-Compiler des *OrbixWeb* generiert wurden.
Zusätzlich verwendet man für jede eingesetzte Komponente, die in einem Fenster enthalten ist, die Methoden, die von der jeweiligen Klasse zur Verfügung gestellt werden. Betrachtet man z.B. die Klasse `java.awt.textField`, so gibt es dort die Methode setText und man kann damit einen Attributwert darstellen. So lautet ein vollständiger Aufruf für die Klasse `HW_Object`, um z.B. die Hardwareadresse auszugeben:

textField1.setText(HWObjectRef.get_HW_address());

```
package xad;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import borland.jbcl.layout.*;
import borland.jbcl.control.*;

public class Applet1 extends Applet {
XYLayout xYLayout1 = new XYLayout();
boolean isStandalone = false;
Button button6 = new Button();
Label label1 = new Label();


//Initialisierung der Komponente
public void jbInit() throws Exception{
xYLayout1.setWidth(816);
xYLayout1.setHeight(157);
button6.setLabel(„ATMSwitch“);
button6.addActionListener(new Applet1_button6_actionAdapter(this));
label1.setFont(new Font(„Dialog“, 1, 24));
label1.setLocation(new Point(40, 6));
label1.setText(„Hicom Xpress Administration“);
this.setLayout(xYLayout1);
this.add(button6, new XYConstraints(413, 47, 69, 36));
this.add(label1, new XYConstraints(230, 7, 357, 21)); }

private static DecoratedFrame frame;

//Main-Methode
static public void main(String[] args) {
Applet1 applet = new Applet1();
applet.isStandalone = true;
frame = new DecoratedFrame();
frame.setTitle(„Applet-Bereich“);
frame.add(applet, BorderLayout.CENTER);
applet.init();
applet.start();
frame.pack();
Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
frame.setLocation((d.width - frame.getSize().width) / 2, (d.height - frame.getSize().height) / 2);
frame.setVisible(true); }

void button6_actionPerformed(ActionEvent e) {
ATMSwitch1 d = new ATMSwitch1(frame,true);
d.show(); }
}

class Applet1_button6_actionAdapter implements java.awt.event.ActionListener {
 Applet1 adaptee;
 Applet1_button6_actionAdapter(Applet1 adaptee) {
 this.adaptee = adaptee; } }
```

Abbildung 6-8: Ausschnitt aus Applet1.java

Zu beachten ist, daß diese Codezeilen nicht dort ergänzt werden müssen, wohin man durch den Doppelklick im *UI-Designer* gelangt, sondern in der Methode *jbInit()*, die vom *JBuilder* im Quelltext zur Verfügung gestellt wird.
Hat man auch diese Verbindung zwischen dem *UI-Designer* und dem Quelltext hergestellt, fehlt nur noch die Verbindung zwischen der entstandenen graphischen Oberfläche und dem CORBA-Agent, damit die Attributwerte ausgelesen werden können.
Diese Anbindung wird auf einem ähnlichen Weg realisiert, wie er für einen beispielhaften Client in Kapitel 5.5 beschrieben wurde.
Zuerst wird ein Pointer für eine Klasse, z.B. `HW_Object` erzeugt:

HW_Object HWObjectVar;

Zusätzlich wird eine Objektreferenz zu dieser Klasse benötigt. Die Deklaration dazu lautet:

_HW_ObjectRef HWObjectRef;

Um nun auf die Attribute dieser Klasse im Applet zugreifen zu können, wird im Applet ein neues Objekt der Klasse, z.B. `HW_Object` erzeugt:

HWObjectVar = new HW_Object();

Damit ist man in der Lage, eine Verbindung zum dazugehörigen Server aufzubauen. Dabei muß der Rechnername mit übergeben werden:

HWObjectRef = HWObjectVar._bind("", "ipx14");

Ist die Verbindung erfolgreich aufgebaut worden, können z.B. die Attribute für die Klasse `HW_Object` abgerufen werden.
Zum besseren Verständnis dieser einzelnen Schritte wird in der Abbildung 6-9 ein Ausschnitt eines Quelltexts dargestellt, der diese einzelnen Programmschritte enthält. In diesem Zusammenhang nicht benötigte Fragmente wurden gestrichen und die relevanten Teile für die Anbindung farbig hervorgehoben.

```
package xad;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import borland.jbcl.control.*;

public class View1 extends Dialog {

HW_Object  HWObjectVar;
_HW_ObjectRef HWObjectRef;

 XYLayout xYLayout1 = new XYLayout();
 Panel panel1 = new Panel();

 public View1(Frame frame, String title, boolean modal) {
 super(frame, title, modal);
 try {
 jbInit();
 add(panel1);
 pack(); }
 catch (Exception e) {
 e.printStackTrace(); }
 }

public void jbInit() throws Exception{
xYLayout1.setWidth(542);
xYLayout1.setHeight(500);
panel1.setLayout(xYLayout1);

 HWObjectVar = new HW_Object();
 System.out.println(„binding HWObject....“);
 try {
 HWObjectRef = HWObjectVar._bind(„“,“ipx14“);

 } catch (Exception e) {
 System.out.println(„EXCEPTION“);
 e.printStackTrace();
 }
 System.out.println(„done.“);

 textField1.setText(HWObjectRef.get_HW_address());
 }
```

Abbildung 6-9: Ausschnitt aus Applet1.java

Um den Prototypen nun fertigzustellen, müssen diese beschriebenen Schritte für alle Klassen und Attribute realisiert werden. Im Anschluß daran muß das fertiggestellte Programm noch compiliert werden und kann dann ausgeführt werden.

Treten dabei Probleme auf, werden vom *JBuilder* noch einige Debuggingfunktionen zur Verfügung gestellt, die bei der Fehlersuche gute Dienste leisten. Damit kann man abschließend sagen, daß mit dem *JBuilder* ein Entwicklungswerkzeug zur Verfügung steht, das den Entwickler bei der Gestaltung und Implementierung einer graphischen Oberfläche gut unterstützt.

Zum Start des Applets wird nun ein Java-fähiger Browser benötigt, der *JDK 1.1* verarbeiten kann.

Zum Zeitpunkt der Test war dies leider nur im *JBuilder* selbst oder im *Hot Java Browser* von Sun möglich. Tests mit *Netscape 4.0* und dem *Explorer 4.0* haben leider nicht den gewünschten Erfolg geliefert, da zur Zeit noch immer nicht der volle Standard vom *JDK 1.1* bei diesen Browsern realisiert ist.
Sind diese Probleme jedoch beseitigt, dürfte es bei der Ausführung des Applets keine Probleme geben. Ein Beispiel für ein Fenster, das die Attribute für die Klasse `ATM_Switch` enthält, ist in Abbildung 6-10 dargestellt.

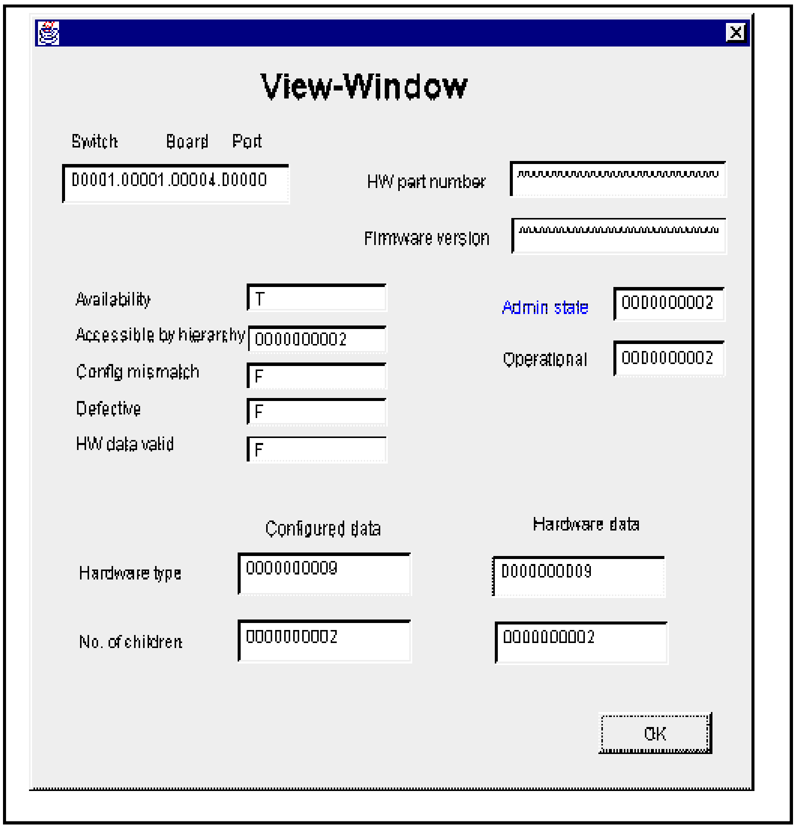


Abbildung 6-10: View-Window des neuen Applets

6.5 Zusammenfassung zum Kapitel 6

Um eine graphische Oberfläche für das Management von ATM-Switches zu erhalten, sind einige Arbeitsschritte notwendig. Nachdem man sich in der Konzeptionsphase vor allem Gedanken über die Gestaltung der Oberfläche gemacht hat, wurde in der zweiten Phase vor allem über die Möglichkeit der Realisierung nachgedacht, wobei die Voraussetzungen, die z.B. durch die bisherige Implementierung des Servers schon vorgegeben waren, nicht aus den Augen verloren werden durften. Nachdem eine Entscheidung bezüglich der Realisierung gefallen war, wurden geeignete Entwicklungstools für die Implementierung gesucht. Zum Einsatz kam schließlich der *JBuilder* der Firma Borland und der *OrbixWeb* der Firma IONA.
Unter diesen Prämissen wurde dann schließlich die graphische Oberfläche als Java-Applet entworfen. Der Grund für diese Realisierungsvariante liegt darin, daß man damit die Plattform- und Betriebssystemunabhängigkeit, die von CORBA schon vorgegeben wurde, weitergeführt hat. Denn ist die richtige Browsersoftware und ein ORB auf einem Rechner vorhanden, kann der Administrator ohne Problem auf den gewünschten ATM-Switch zugreifen und diesen administrieren.
Bei der vorliegenden Implementierung des Prototypen ist aber zu beachten, daß die Attributwerte noch so im Applet dargestellt werden, wie man sie aus der Kommunikation mit Hicom Xpress erhält. Um die Benutzerfreundlichkeit des entwickelten Applets noch zu erhöhen, sollten die vom CORBA-Agenten gelieferten Daten noch in Fließtext umgewandelt werden. Ist auch dies geschehen steht ein komfortables Tool zum Management von ATM-Switches zur Verfügung.

7 Zusammenfassung und Ausblick

Nachdem alle Arbeiten für den Entwurf und die CORBA-basierte Implementierung eines Objektmodells für das Management von ATM-Switches abgeschlossen sind, werden im folgenden nochmals die einzelnen durchgeführten Arbeitsschritte in chronologischer Abfolge zusammengefaßt.

Die Grundlage für die gesamte Arbeit war der ATM-Switch Hicom Xpress der Firma Siemens, Bereich Private Netze. Nach der Erfassung und Darstellung der vorhandenen Funktionalität des Hicom Xpress, wird zusammen mit seinem Datenmodell ein erstes Objektmodell entwickelt. Bei der Modellierung wird dabei auf die *Object Modeling Technique* von Rumbaugh zurückgegriffen, da dadurch eine einheitliche und allgemein bekannte Notation zur Verfügung steht. Um die erreichten Ergebnisse darzustellen, wird das Case Tool *Software through Picture* eingesetzt, das auch die Notation von OMT darstellen kann.

Zu beachten ist, daß in diesem ersten Objektmodell keine Beziehungen zwischen den einzelnen Klassen modelliert sind. Auch geben die definierten Klassen, Attribute und Methoden die Struktur des Hicom Xpress nicht deutlich genug wieder.
Deshalb wird im nächsten Schritt ein zweites Objektmodell entwickelt, das die Beziehungen zwischen den Klassen zum Ausdruck bringt. Zusätzlich werden neue Klassen eingeführt und die vorhanden Attribute und Methoden auf ihre Verwendbarkeit überprüft. Damit ist ein zweites Objektmodell entstanden, daß die Struktur des Hicom Xpress gut widerspiegelt.

Da aber auch zu diesem Zeitpunkt der Modellierung noch nicht alle objektorientierten Strukturen, wie Vererbung ins Modell eingeflossen sind, wird noch ein drittes Objektmodell entwickelt. Mit der Fertigstellung dieses dritten Objektmodells ist dann endgültig ein Objektmodell für das Management des Hicom Xpress vorhanden und die durchgeführte Bottom-up-Analyse abgeschlossen.

Um aber auch gewährleisten zu können, daß das Objektmodell bei anderen ATM-Switches eingesetzt werden kann, soll mittels einer Top-down-Analyse eine generisches Objektmodell entstehen. Eine gute Möglichkeit zur Unterstützung dieses Prozesses findet man in den *Use Cases*. Mit diesen kann man die Aufgaben für unterschiedliche Anwender bzw. Benutzer eines ATM-Switches definieren.

Nachdem die *Use Cases* für mehrere unterschiedliche Akteure im Case Tool StP definiert sind, werden diese zusammen mit dem dritten Objektmodell des Hicom Xpress in Verbindung gebracht.

Durch diesen Schritt entsteht ein generisches Objektmodell, das als Grundlage für das Management von ATM-Switches verwendet werden kann. Wendet man dieses generische Objektmodell auf das dritte Objektmodell des Hicom Xpress an, so kann man dabei keine genaue Trennung zwischen generischen und devicespezifischen Klassen vornehmen. Nur bei der Betrachtung von Attributen und Methoden gibt es einige, die anders zu benennen oder zu realisieren sind.
Mit diesem Schritt ist die erste große Phase der vorliegenden Arbeit abgeschlossen und es kann nach der Modellierung mit der Implementierung eines CORBA-Agenten auf Basis des entworfenen Objektmodells begonnen werden.

Dabei wird aber nicht das vollkommen generische Objektmodell verwendet, sondern das dritte Objektmodell des Hicom Xpress. Der Grund liegt darin, daß im generischen Objektmodell Klassen vorhanden sind, die von der Technik des Hicom Xpress zur Zeit und wohl auch zukünftig nicht unterstützt werden.

Im ersten Schritt der Implementierungsphase wird für jede Klasse im Objektmodell der IDL-Code generiert. Diese Funktionalität wird wiederum vom Case Tool *Software through Pictures* zur Verfügung gestellt.
Sind für alle Klassen der IDL-Code generiert, werden diese mit dem IDL-Compiler übersetzt. Der eingesetzte IDL-Compiler ist dabei eine Komponente im CORBA-Produkt *Orbix* von IONA, das für die Implementierung verwendet wird.
Durch die Übersetzung mit dem IDL-Compiler entstehen für jede Klasse Header-Dateien, Server- und Client-Stubs und sogenannte Implementierungsdateien. Für die Fortsetzung der Arbeit können dabei die Header-Dateien und die Server- und Client-Stubs ohne Modifizierungen übernommen werden. Anpassungen sind nur an den Implementierungsdateien nötig. Dabei müssen die C++-Systemaufrufe für die einzelnen Attribute und Methoden ergänzt werden und zusätzlich noch Konstruktoren und Destruktoren und Includedateien. Um diese implementierten Dateien einsetzen zu können, wird im weiteren ein Server und ein Client implementiert.

Die Aufgabe des implementierten Servers liegt darin, die ankommenden Requests des Clients entgegenzunehmen und die gewünschten Informationen an den Client zurückzuschicken. Mit der abgeschlossen Implementierung des Servers und der Compilierung aller benötigten Dateien, ist die Implementierung des CORBA-Agenten für das Management eines ATM-Switches abgeschlossen. Zu beachten ist dabei noch, daß für diese vorliegende Implementierung ein Stützprozeß geschrieben werden mußte. Diese Proxylösung war nötig, da für das Betriebssystem des *Control Blocks* des Hicom Xpress kein ORB von der Firma IONA zur Verfügung gestanden ist.

Um diesen implementierten CORBA-Agenten auch nutzen zu können, wird ein lauffähiger Client benötigt. Da für einen Client, der direkt für *Orbix* geschrieben ist, nur ein Kommandozeileninterface zur Verfügung steht, wird im letzten Teil der Arbeit eine graphische Oberfläche entworfen und implementiert.
Dem Konzept nach sollte eine graphische Oberfläche entworfen werden, die für den Benutzer die benötigten Informationen deutlich darstellt und sich bei der Anzahl der eingesetzten Fenster auf ein Minimum beschränkt. Außerdem sollte die Bedienung der Oberfläche so implementiert werden, daß sie für den Benutzer intuitiv deutlich wird. Nach Fertigstellung des Konzepts wird über das geplante Zielszenario vor allem im Hinblick auf die eingesetzten Entwicklungstools nachgedacht.
Für die Implementierung werden als Folge dieser Betrachtung der *OrbixWeb* von IONA und der *JBuilder* von Borland eingesetzt. Mit dem beim *OrbixWeb* mitgelieferten IDL-Compiler werden dann für alle IDL-Dateien Java-Code erzeugt. Dieser wird für die weitere Entwicklung benötigt, da die Oberfläche mit Hilfe des *JBuilders* als Applet realisiert werden soll. Hat man mit dem *JBuilder* die Oberfläche entworfen und den notwendigen Code für den Aufruf des Servers ergänzt, steht eine graphische Oberfläche für das Management eines ATM-Switches zur Verfügung.

Betrachtet man diese einzelnen Schritte im Zusammenhang, so kann man sagen, daß ein Objektmodell entworfen wurde und auf dieser Basis ein CORBA-Agent mit graphischer Oberfläche für das Management eines ATM-Switches entstanden ist. Die Performance dieser vorliegenden Implementierung ist durchaus zufriedenstellend, wobei leider keine Aussage darüber gemacht werden kann, ob eine Leistungssteigerung ohne den Stützprozeß möglich ist.

Wirft man zum Abschluß der Implementierungsphase noch einen Blick auf die eingesetzten Technologien, so kann man sagen, daß mit CORBA ein Standard zur Verfügung steht, der die heute weit verbreiteten Client/Server-Strukturen gut unterstützt. Ein Vorteil dabei ist vor allem, daß CORBA bei der Implementierung u.a. von Betriebssystemen unabhängig ist. Jedoch muß für die gewünschte Plattform ein ORB zur Verfügung stehen. Ist dies nicht der Fall, so ist ein größerer Aufwand nötig, um die fehlende Funktionalität selber zu realisieren. Bei den eingesetzten Produkten *Orbix* und *OrbixWeb* von IONA konnte man feststellen, daß die Dokumentation in den meisten Fällen durchaus hilfreich und ausreichend war und auch der Support von IONA zufriedendstellend ist. Ein Punkt, der positiv aufgefallen ist, ist die bessere Unterstützung der Arbeit auf der Plattform für *Windows NT 4.0*. Dort wurde nämlich eine graphische Oberfläche zur Verfügung gestellt, die die Arbeit mit *Orbix* und *OrbixWeb* doch stark erleichtert hat, da man sich nicht die langen Kommandozeilenbefehle von *Unix* merken mußte. Insgesamt stellt IONA mit *Orbix* und *OrbixWeb* gute Produkte im Zusammenhang mit der CORBA-Technologie zur Verfügung.

Auch das in der Modellierungsphase und zu Beginn der Implementierungsphase eingesetzte CASE-Tool *Software through Pictures* ist positiv zu bewerten. Mit seinem Funktionsumfang und vor allem auch seinem Navigationssystem zwischen unterschiedlichen Modellen und Tabellen wird dem Anwender die Modellierung eines Objektmodells doch stark vereinfacht. So kann z.B. eine in dem Klassentabelleneditor erstellte Klasse ohne Probleme als graphische Darstellung in den Objektmodelleditor übernommen werden. Auch die bereitgestellte Codegenerierung für die im Objektmodell dargestellten Klassen ist nicht zu verachten. Verbesserungen sind nur im Hinblick auf die Darstellung der Ergebnisse vorzunehmen. So ist es nicht möglich, einen graphischen Auszug eines Objektmodells zu bekommen. Vielmehr muß man das komplette Objektmodell z.B. in ein *FrameMaker*-Dokument umwandeln und kann erst dort die jeweiligen Klassen herausziehen.
Auch wird bei der Übernahme eines Objektmodells in ein *FrameMaker*-Dokument die Ausrichtung der einzelnen Attribute und Methoden verschoben. Trotzdem stellt Aonix mit StP ein Produkt auf dem Markt zur Verfügung, daß mit seinem Funktionsumfang nur wenige Wünsche bei der Modellierung offen läßt.

Das für die Gestaltung und Implementierung eingesetzte Entwicklungstool JBuilder von Borland hinterläßt ebenfalls einen positiven Eindruck. Dies kommt vor allem daher, daß beim Entwurf und der Implementierung der Oberfläche nur wenige Probleme auftreten und der größte Teil der Codegenerierung vom JBuilder übernommen wird. Im vorliegenden Fall mußten z.B. nur die Codezeilen für den Aufruf des CORBA-Agenten ergänzt werden. Auch die Suche nach bestimmten Attributen oder Methoden im Code wird durch den *JBuilder* vorbildlich gelöst. Sucht man z.B. die Stelle im Code, an dem eine Aktion für einen in der Oberfläche enthaltenen Button definiert werden soll, so ist dies mit einem Doppelklick sofort gelöst. Ein Nachteil ist, daß durch die Neuheit des Produkts, vor allem bezüglich *JDK 1.1* Probleme beim Aufruf des Applets im Netscape oder Microsoft Browser auftreten. Jedoch sollten diese Probleme in zukünftigen Versionen der Browser gelöst sein.

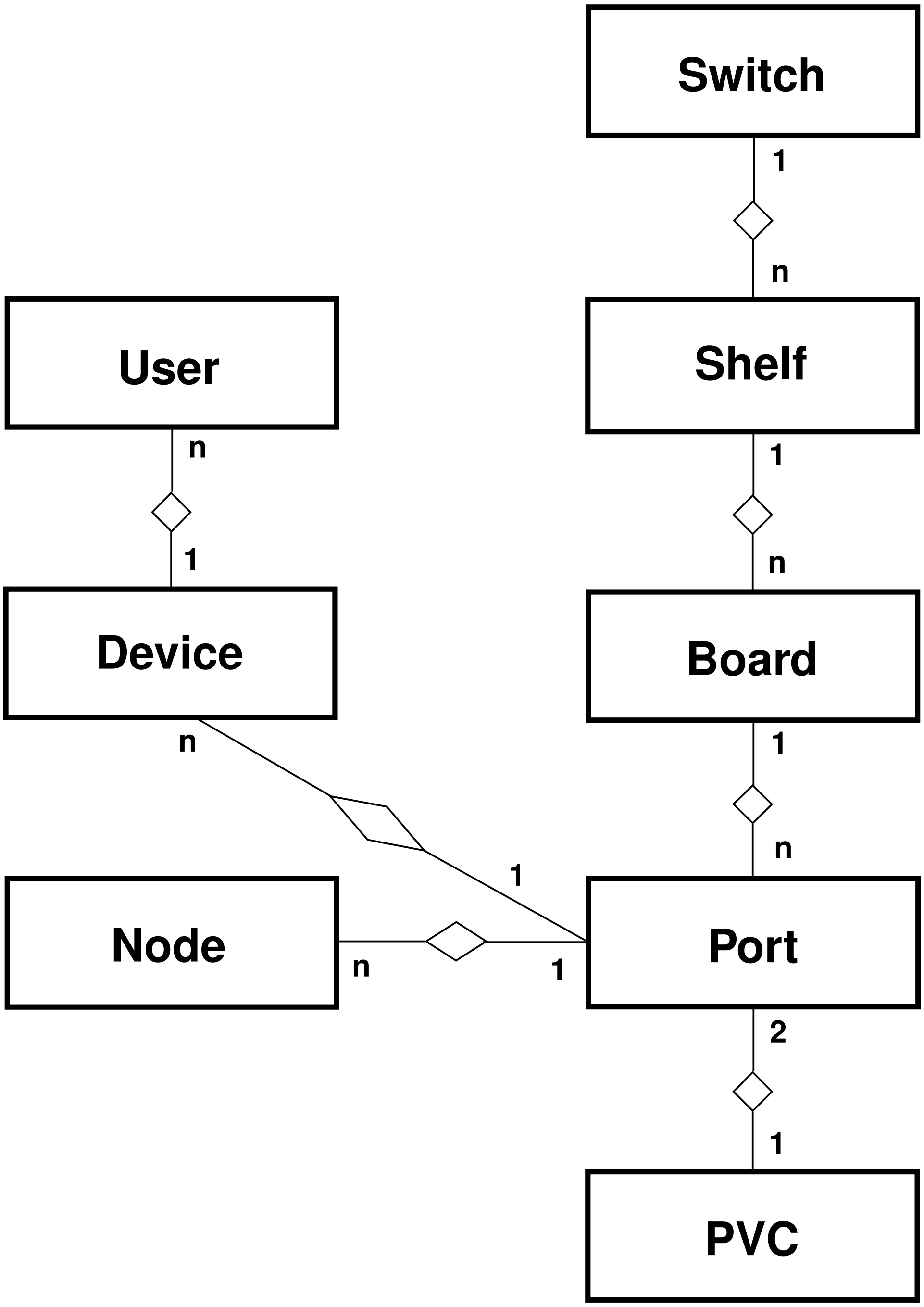
Somit kann man abschließend sagen, daß die verwendeten Entwicklungstools die Entwurfs- und Implementierungsphasen der vorliegenden Arbeit zur vollen Zufriedenheit bewältigt haben.

Mit dem Abschluß dieser Betrachtung kann man sagen, daß man im Zusammenhang mit dieser Arbeit einerseits die Einsetzbarkeit und Verwendbarkeit der CORBA-Technologie im Zusammenhang mit Managementaspekten nachgewiesen hat, aber andererseits auch den Einsatz von Case- und Entwicklungstools rechtfertigen konnte, da sie die Arbeit für den Entwickler doch erleichtern.

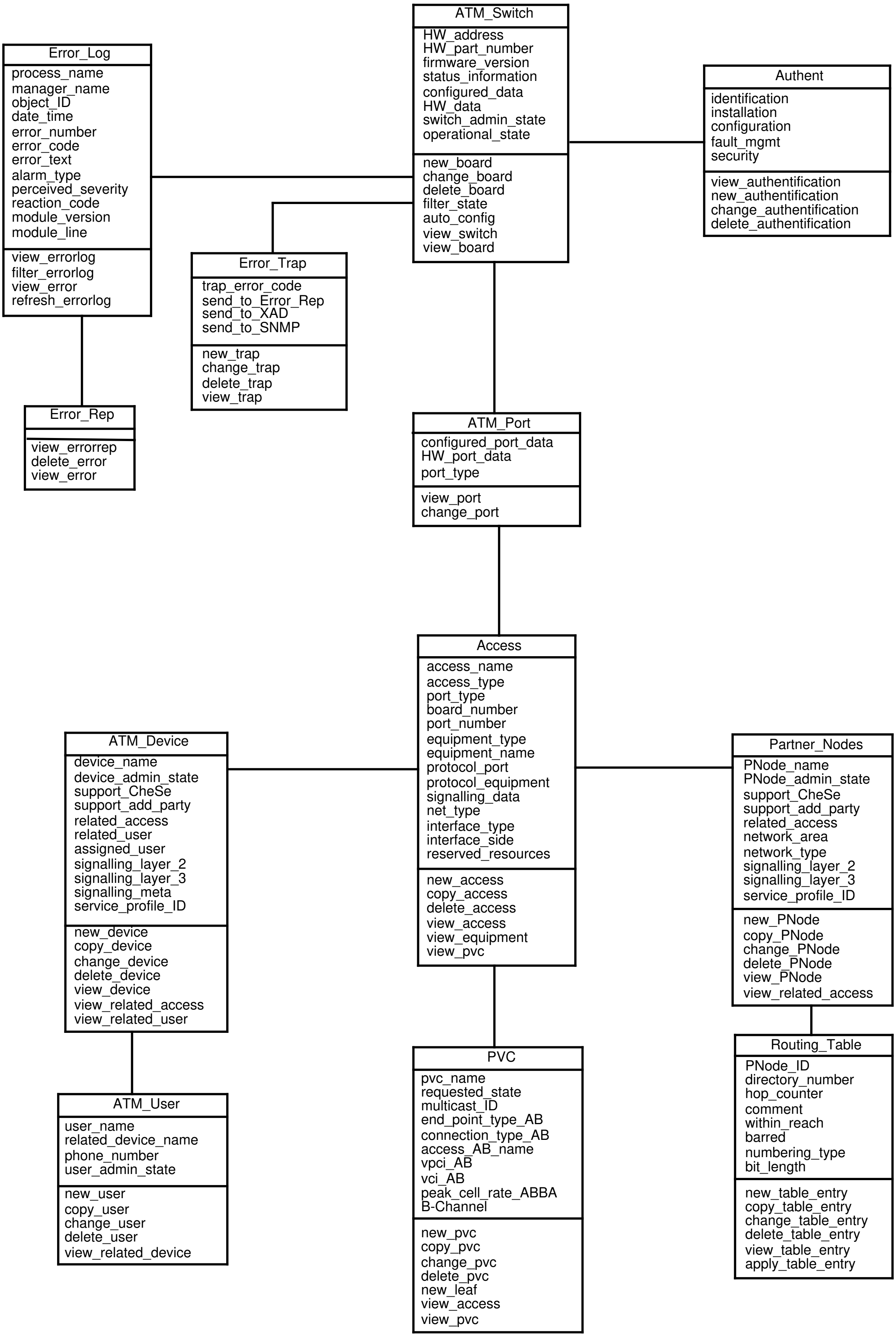
Außerdem konnten Erfahrungen darüber gesammelt werden, inwieweit ein gut modelliertes Objektmodell, die spätere Implementierung erleichtern, was im Zusammenhang mit Neu- oder Weiterentwicklungen nicht unwichtig ist. Zu beachten ist aber, daß dies nur möglich ist, wenn Technologien wie CORBA oder Java eingesetzt werden, die u.a. plattformunabhängig arbeiten können.

Wagt man in diesem Zusammenhang einen abschließenden Ausblick, so kommt man zu folgendem Ergebnis. Um z.B. die Plattformunabhängigkeit der CORBA-Technologie noch besser nachweisen zu können, sollte man versuchen einen ORB einer anderen Firma auf dem *Control Block* des Hicom Xpress zu verwenden, der dann natürlich auf *Unixware* lauffähig sein muß. Damit könnte dann auch der Einsatz des implementierten Stützprozesses beendet werden. Ebenso könnte man das entwickelte Objektmodell dazu verwenden, einen CORBA-Agenten für einen anderen ATM-Switch zu realisieren.

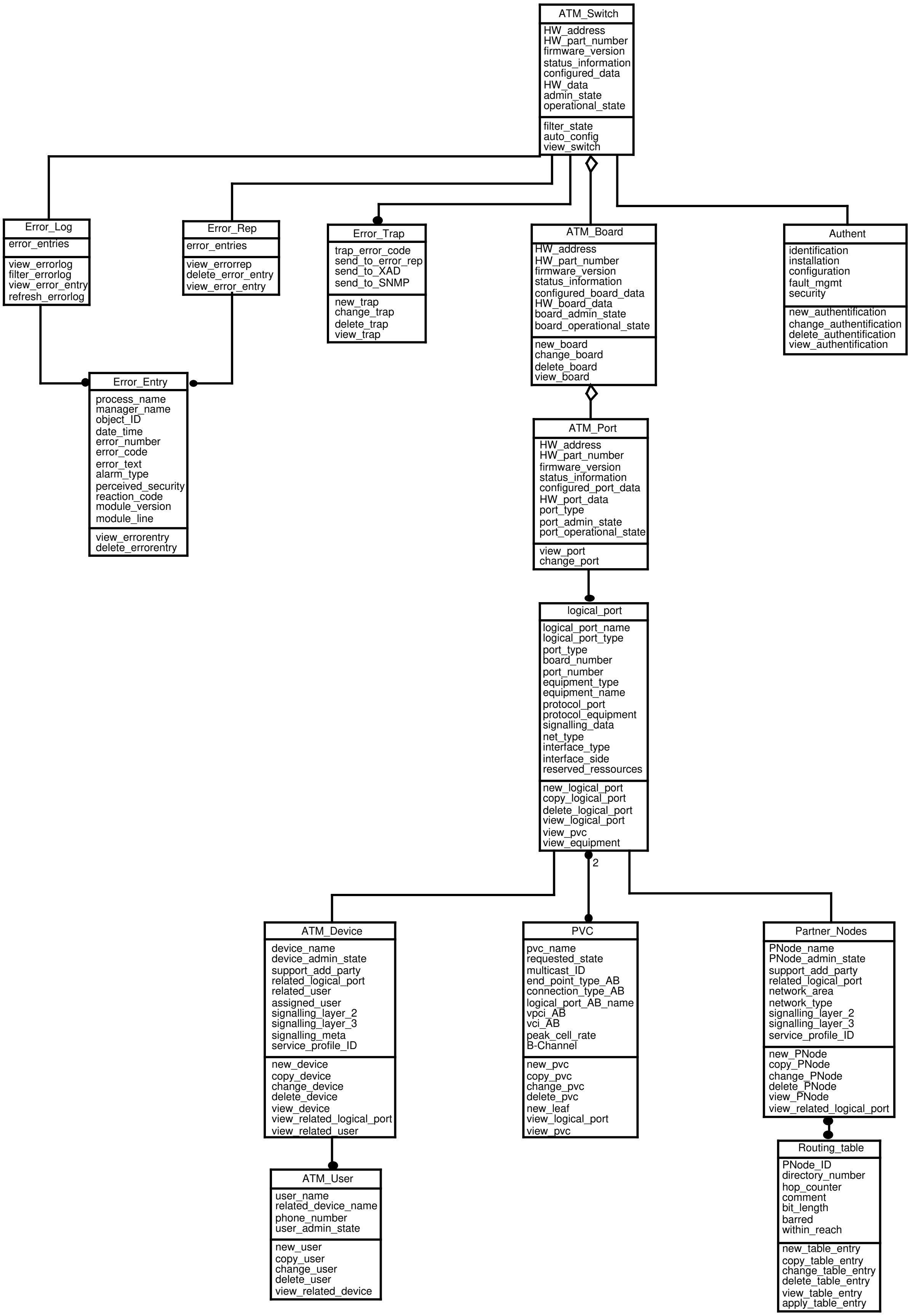
Anhang A: Daten- und Objektmodelle



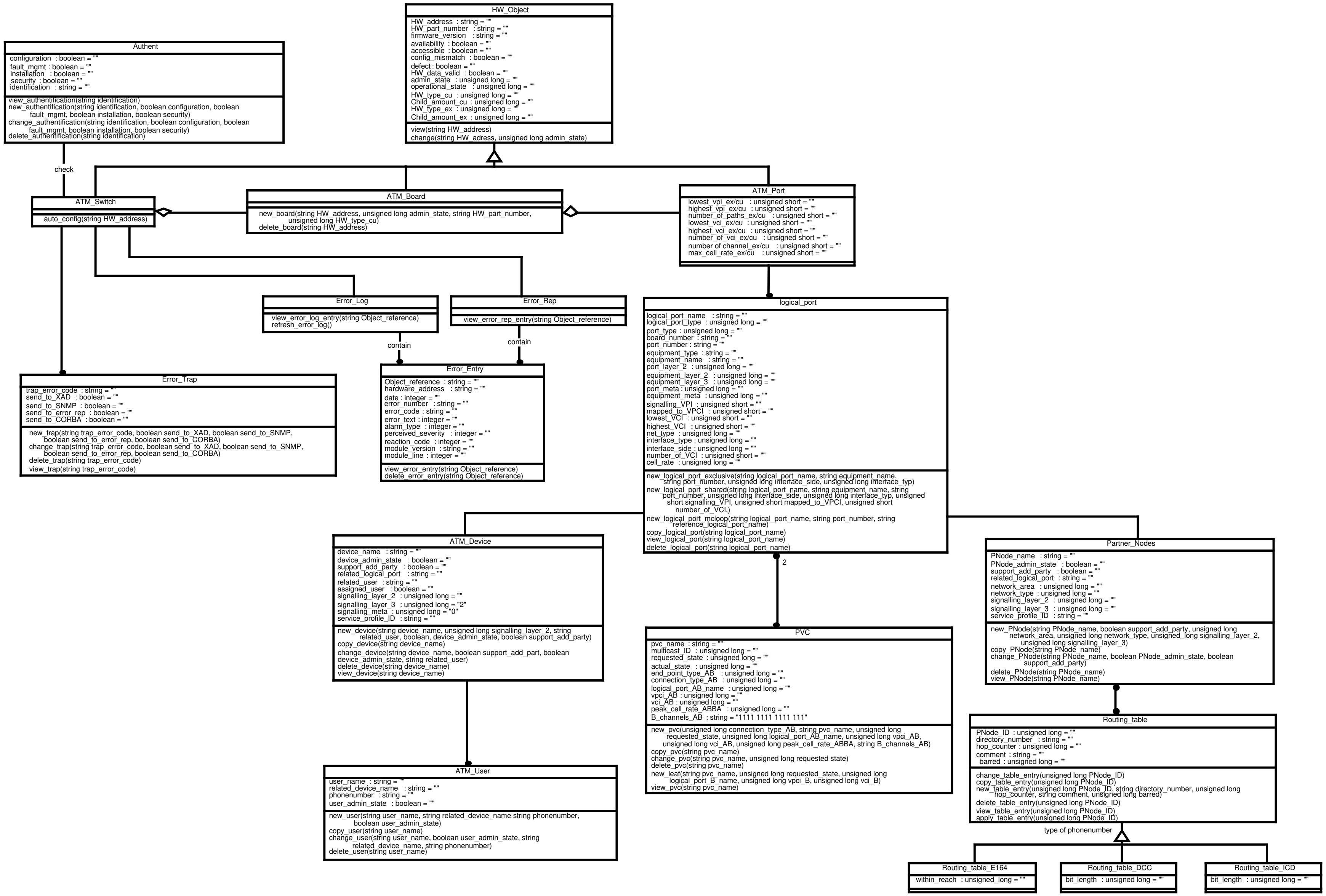
Datenmodell des Hicom Xpress



Objektmodell für ATM-Switches Version 1



Objektmodell für ATM-Switches Version 2


Authent
configuration : boolean = ""
fault_mgmt : boolean = ""
installation : boolean = ""
security : boolean = ""
identification : string = ""
view_authentification(string identification)
new_authentification(string identification, boolean configuration, boolean fault_mgmt, boolean installation, boolean security)
change_authentification(string identification, boolean configuration, boolean fault_mgmt, boolean installation, boolean security)
delete_authentification(string identification)
check
HW_Object
HW_address : string = ""
HW_part_number : string = ""
firmware_version : string = ""
availability : boolean = ""
accessible : boolean = ""
config_mismatch : boolean = ""
defect : boolean = ""
HW_data_valid : boolean = ""
admin_state : unsigned long = ""
operational_state : unsigned long = ""
HW_type_cu : unsigned long = ""
Child_amount_cu : unsigned long = ""
HW_type_ex : unsigned long = ""
Child_amount_ex : unsigned long = ""
view(string HW_address)
change(string HW_adress, unsigned long admin_state)
ATM_Switch
auto_config(string HW_address)
ATM_Board
new_board(string HW_address, unsigned long admin_state, string HW_part_number, unsigned long HW_type_cu)
delete_board(string HW_address)
ATM_Port
lowest_vpi_ex/cu : unsigned short = ""
highest_vpi_ex/cu : unsigned short = ""
number_of_paths_ex/cu : unsigned short = ""
lowest_vci_ex/cu : unsigned short = ""
highest_vci_ex/cu : unsigned short = ""
number_of_vci_ex/cu : unsigned short = ""
number of channel_ex/cu : unsigned short = ""
max_cell_rate_ex/cu : unsigned short = ""
Error_Log
view_error_log_entry(string Object_reference)
refresh_error_log()
Error_Rep
view_error_rep_entry(string Object_reference)
contain
contain
Error_Trap
trap_error_code : string = ""
send_to_XAD : boolean = ""
send_to_SNMP : boolean = ""
send_to_error_rep : boolean = ""
send_to_CORBA : boolean = ""
new_trap(string trap_error_code, boolean send_to_XAD, boolean send_to_SNMP, boolean send_to_error_rep, boolean send_to_CORBA)
change_trap(string trap_error_code, boolean send_to_XAD, boolean send_to_SNMP, boolean send_to_error_rep, boolean send_to_CORBA)
delete_trap(string trap_error_code)
view_trap(string trap_error_code)
Error_Entry
Object_reference : string = ""
hardware_address : string = ""
date : integer = ""
error_number : string = ""
error_code : string = ""
error_text : integer = ""
alarm_type : integer = ""
perceived_severity : integer = ""
reaction_code : integer = ""
module_version : string = ""
module_line : integer = ""
view_error_entry(string Object_reference)
delete_error_entry(string Object_reference)
logical_port
logical_port_name : string = ""
logical_port_type : unsigned long = ""
port_type : unsigned long = ""
board_number : string = ""
port_number : string = ""
equipment_type : string = ""
equipment_name : string = ""
port_layer_2 : unsigned long = ""
equipment_layer_2 : unsigned long = ""
equipment_layer_3 : unsigned long = ""
port_meta : unsigned long = ""
equipment_meta : unsigned long = ""
signalling_VPI : unsigned short = ""
mapped_to_VPCI : unsigned short = ""
lowest_VCI : unsigned short = ""
highest_VCI : unsigned short = ""
net_type : unsigned long = ""
interface_type : unsigned long = ""
interface_side : unsigned long = ""
number_of_VCI : unsigned short = ""
cell_rate : unsigned long = ""
new_logical_port_exclusive(string logical_port_name, string equipment_name, string port_number, unsigned long interface_side, unsigned long interface_typ)
new_logical_port_shared(string logical_port_name, string equipment_name, string port_number, unsigned long interface_side, unsigned long interface_typ, unsigned short signalling_VPI, unsigned short mapped_to_VPCI, unsigned short number_of_VCI,)
new_logical_port_mcloop(string logical_port_name, string port_number, string reference_logical_port_name)
copy_logical_port(string logical_port_name)
view_logical_port(string logical_port_name)
delete_logical_port(string logical_port_name)
2
ATM_Device
device_name : string = ""
device_admin_state : boolean = ""
support_add_party : boolean = ""
related_logical_port : string = ""
related_user : string = ""
assigned_user : boolean = ""
signalling_layer_2 : unsigned long = ""
signalling_layer_3 : unsigned long = "2"
signalling_meta : unsigned long = "0"
service_profile_ID : string = ""
new_device(string device_name, unsigned long signalling_layer_2, string related_user, boolean, device_admin_state, boolean support_add_party)
copy_device(string device_name)
change_device(string device_name, boolean support_add_part, boolean device_admin_state, string related_user)
delete_device(string device_name)
view_device(string device_name)
ATM_User
user_name : string = ""
related_device_name : string = ""
phonenumber : string = ""
user_admin_state : boolean = ""
new_user(string user_name, string related_device_name string phonenumber, boolean user_admin_state)
copy_user(string user_name)
change_user(string user_name, boolean user_admin_state, string related_device_name, string phonenumber)
delete_user(string user_name)
PVC
pvc_name : string = ""
multicast_ID : unsigned long = ""
requested_state : unsigned long = ""
actual_state : unsigned long = ""
end_point_type_AB : unsigned long = ""
connection_type_AB : unsigned long = ""
logical_port_AB_name : unsigned long = ""
vpci_AB : unsigned long = ""
vci_AB : unsigned long = ""
peak_cell_rate_ABBA : unsigned long = ""
B_channels_AB : string = "1111 1111 1111 111"
new_pvc(unsigned long connection_type_AB, string pvc_name, unsigned long requested_state, unsigned long logical_port_AB_name, unsigned long vpci_AB, unsigned long vci_AB, unsigned long peak_cell_rate_ABBA, string B_channels_AB)
copy_pvc(string pvc_name)
change_pvc(string pvc_name, unsigned long requested state)
delete_pvc(string pvc_name)
new_leaf(string pvc_name, unsigned long requested_state, unsigned long logical_port_B_name, unsigned long vpci_B, unsigned long vci_B)
view_pvc(string pvc_name)
Partner_Nodes
PNode_name : string = ""
PNode_admin_state : boolean = ""
support_add_party : boolean = ""
related_logical_port : string = ""
network_area : unsigned long = ""
network_type : unsigned long = ""
signalling_layer_2 : unsigned long = ""
signalling_layer_3 : unsigned long = ""
service_profile_ID : string = ""
new_PNode(string PNode_name, boolean support_add_party, unsigned long network_area, unsigned long network_type, unsigned_long signalling_layer_2, unsigned long signalling_layer_3)
copy_PNode(string PNode_name)
change_PNode(string PNode_name, boolean PNode_admin_state, boolean support_add_party)
delete_PNode(string PNode_name)
view_PNode(string PNode_name)
Routing_table
PNode_ID : unsigned long = ""
directory_number : string = ""
hop_counter : unsigned long = ""
comment : string = ""
barred : unsigned long = ""
change_table_entry(unsigned long PNode_ID)
copy_table_entry(unsigned long PNode_ID)
new_table_entry(unsigned long PNode_ID, string directory_number, unsigned long hop_counter, string comment, unsigned long barred)
delete_table_entry(unsigned long PNode_ID)
view_table_entry(unsigned long PNode_ID)
apply_table_entry(unsigned long PNode_ID)
type of phonenumber
Routing_table_E164
within_reach : unsigned_long = ""
Routing_table_DCC
bit_length : unsigned long = ""
Routing_table_ICD
bit_length : unsigned long = ""

Anhang B: IDL-Code der Implementierung

Klasse `HW_Object`:

```
// StP -- created on Wed Sep 17 11:07:34 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _HW_Object_idl_
#define _HW_Object_idl_

// stp class declarations
interface HW_Object;
// stp class declarations end

// stp class definition 208
interface HW_Object
{
// stp class members
    readonly attribute string HW_address;
    readonly attribute string HW_part_number;
    readonly attribute string firmware_version;
    readonly attribute boolean availability;
    readonly attribute boolean accessible;
    readonly attribute boolean config_mismatch;
    readonly attribute boolean defect;
    readonly attribute boolean HW_data_valid;
    attribute unsigned long admin_state;
    readonly attribute unsigned long operational_state;
    readonly attribute unsigned long HW_type_cu;
    readonly attribute unsigned long Child_amount_cu;
    readonly attribute unsigned long HW_type_ex;
    readonly attribute unsigned long Child_amount_ex;
    void view(in string HW_address);
    void change(string HW_address, unsigned long admin_state);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `ATM_Switch`:

```
// StP -- created on Wed Sep 17 10:35:24 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _ATM_Switch_idl_
#define _ATM_Switch_idl_

// stp class declarations
interface ATM_Switch;
// stp class declarations end

// stp auto-include 126
#include "HW_Object.idl"
// stp auto-include end

// stp class definition 126
interface ATM_Switch : HW_Object
{
// stp class members
   void auto_config(string HW_address);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `ATM_Board`:

```
// StP -- created on Wed Sep 17 10:31:00 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _ATM_Board_idl_
#define _ATM_Board_idl_

// stp class declarations
interface ATM_Board;
// stp class declarations end

// stp auto-include 123
#include "HW_Object.idl"
// stp auto-include end

// stp class definition 123
interface ATM_Board : HW_Object
{
```

```
// stp class members
   void new_board(string HW_address, unsigned long admin_state, string HW_part_number, unsi-
gned long HW_type_cu);
   void delete_board(string HW_address);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `ATM_Port`:

```
// StP -- created on Wed Sep 17 10:33:19 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _ATM_Port_idl_
#define _ATM_Port_idl_

// stp class declarations
interface ATM_Port;
// stp class declarations end

// stp auto-include 128
#include "HW_Object.idl"
// stp auto-include end

// stp class definition 128
interface ATM_Port : HW_Object
{
// stp class members
   readonly attribute unsigned short lowest_vpi_ex_cu;
   readonly attribute unsigned short highest_vpi_ex_cu;
   readonly attribute unsigned short number_of_paths_ex_cu;
   readonly attribute unsigned short lowest_vci_ex_cu;
   readonly attribute unsigned short highest_vci_ex_cu;
   readonly attribute unsigned short number_of_vci_ex_cu;
   readonly attribute unsigned short number of channel_ex_cu;
   readonly attribute unsigned short max_cell_rate_ex_cu;
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `logical_port`

```
// StP -- created on Wed Sep 17 10:27:39 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _logical_port_idl_
#define _logical_port_idl_

// stp class declarations
interface logical_port;
// stp class declarations end

// stp class definition 129
interface logical_port
{
// stp class members
    attribute string logical_port_name;
    attribute unsigned long logical_port_type;
    attribute unsigned long port_type;
    attribute string board_number;
    readonly attribute string port_number;
    attribute string equipment_type;
    attribute string equipment_name;
    readonly attribute unsigned long port_layer_2;
    readonly attribute unsigned long equipment_layer_2;
    readonly attribute unsigned long equipment_layer_3;
    readonly attribute unsigned long port_meta;
    readonly attribute unsigned long equipment_meta;
    attribute unsigned short signalling_VPI;
    attribute unsigned short mapped_to_VPCI;
    readonly attribute unsigned short lowest_VCI;
    readonly attribute unsigned short highest_VCI;
    readonly attribute unsigned long net_type;
    readonly attribute unsigned long interface_type;
    attribute unsigned long interface_side;
    attribute unsigned short number_of_VCI;
    attribute unsigned long cell_rate;
            new_logical_port_exclusive(string logical_port_name, string equipment_name, string port_number, unsigned long interface_side, unsigned long interface_typ);
        void new_logical_port_shared(string logical_port_name, string equipment_name, string port_number, unsigned long interface_side, unsigned long interface_typ, unsigned short signalling_VPI, unsigned short mapped_to_VPCI, unsigned short number_of_VCI,);
          void new_logical_port_mcloop(string logical_port_name, string port_number, string reference_logical_port_name);
    void copy_logical_port(string logical_port_name);
    void view_logical_port(string logical_port_name);
    void delete_logical_port(string logical_port_name);
// stp class members end
};
```

```
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `ATM_Device`:

```
// StP -- created on Wed Sep 17 10:31:43 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _ATM_Device_idl_
#define _ATM_Device_idl_
// stp class declarations
interface ATM_Device;
// stp class declarations end

// stp class definition 246
interface ATM_Device
{
// stp class members
   attribute string device_name;
   attribute boolean device_admin_state;
   attribute string related_logical_port;
   attribute string related_user;
   attribute unsigned long signalling_layer_2;
   readonly attribute unsigned long signalling_layer_3;
   readonly attribute unsigned long signalling_meta;
   readonly attribute string service_profile_ID;
   attribute boolean support_add_party;
   readonly attribute boolean assigned_user;
   void new_device(string device_name, unsigned long signalling_layer_2, string related_user, boo-
lean device_admin_state, boolean support_add_party);
   void copy_device(string device_name);
         void change_device(string device_name, boolean support_add_party, boolean
device_admin_state, string related_user);
   void delete_device(string device_name);
   void view_device(string device_name);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `ATM_User`:

```
// StP -- created on Wed Sep 17 10:34:37 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _ATM_User_idl_
#define _ATM_User_idl_

// stp class declarations
interface ATM_User;
// stp class declarations end

// stp class definition 154
interface ATM_User
{
// stp class members
   attribute string user_name;
   attribute string related_device_name;
   attribute string phonenumber;
   attribute boolean user_admin_state;
     void new_user(string user_name, string related_device_name string phonenumber, boolean
user_admin_state);
   void copy_user(string user_name);
     void change_user(string user_name, boolean user_admin_state, string related_device_name,
string phonenumber);
   void delete_user(string user_name);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `PVC`:

```
// StP -- created on Wed Sep 17 11:08:09 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _PVC_idl_
#define _PVC_idl_

// stp class declarations
interface PVC;
// stp class declarations end

// stp class definition 141
interface PVC
```

```
{
// stp class members
    attribute string pvc_name;
    attribute unsigned long multicast_ID;
    attribute unsigned long requested_state;
    attribute unsigned long actual_state;
    readonly attribute unsigned long end_point_type_AB;
    readonly attribute unsigned long connection_type_AB;
    attribute unsigned long logical_port_AB_name;
    attribute unsigned long vpci_AB;
    attribute unsigned long vci_AB;
    attribute unsigned long peak_cell_rate_ABBA;
    readonly attribute string B_channels_AB;
        void new_pvc(unsigned long connection_type_AB, string pvc_name, unsigned long requested_state, unsigned long logical_port_AB_name, unsigned long vpci_AB, unsigned long vci_AB, unsigned long peak_cell_rate_ABBA, string B_channels_AB);
    void copy_pvc(string pvc_name);
    void change_pvc(string pvc_name, unsigned long requested state);
    void delete_pvc(string pvc_name);
        void new_leaf(string pvc_name, unsigned long requested_state, unsigned long logical_port_B_name, unsigned long vpci_B, unsigned long vci_B);
    void view_pvc(string pvc_name);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Partner_Nodes`:

```
// StP -- created on Wed Sep 17 11:08:53 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Partner_Nodes_idl_
#define _Partner_Nodes_idl_

// stp class declarations
interface Partner_Nodes;
// stp class declarations end

// stp class definition 148
interface Partner_Nodes
{
// stp class members
    attribute string PNode_name;
```

```
    attribute boolean PNode_admin_state;
    attribute boolean support_add_party;
    attribute string related_logical_port;
    attribute unsigned long network_area;
    attribute unsigned long network_type;
    attribute unsigned long signalling_layer_2;
    readonly attribute unsigned long signalling_layer_3;
    readonly attribute string service_profile_ID;
    void new_PNode(string PNode_name, boolean support_add_party, unsigned long network_area,
unsigned long network_type, unsigned_long signalling_layer_2, unsigned long signalling_layer_3);
    void copy_PNode(string PNode_name);
    void change_PNode(string PNode_name, boolean PNode_admin_state, boolean
support_add_party);
    void delete_PNode(string PNode_name);
    void view_PNode(string PNode_name);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Routing_table`:

```
// StP -- created on Wed Sep 17 11:10:06 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Routing_table_idl_
#define _Routing_table_idl_

// stp class declarations
interface Routing_table;
// stp class declarations end

// stp class definition 179
interface Routing_table
{
// stp class members
    readonly attribute unsigned long PNode_ID;
    attribute string directory_number;
    attribute unsigned long hop_counter;
    attribute string comment;
    attribute unsigned long barred;
    void change_table_entry(unsigned long PNode_ID);
    void copy_table_entry(unsigned long PNode_ID);
```

```
    void new_table_entry(unsigned long PNode_ID, string directory_number, unsigned long hop_counter, string comment, unsigned long barred);
  void delete_table_entry(unsigned long PNode_ID);
  void view_table_entry(unsigned long PNode_ID);
  void apply_table_entry(unsigned long PNode_ID);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Routing_table_E164`:

```
// StP -- created on Wed Sep 17 11:13:06 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Routing_table_E164_idl_
#define _Routing_table_E164_idl_

// stp class declarations
interface Routing_table_E164;
// stp class declarations end

// stp auto-include 186
#include "Routing_table.idl"
// stp auto-include end

// stp class definition 186
interface Routing_table_E164 : Routing_table
{
// stp class members
  attribute unsigned_long within_reach;
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Routing_table_DCC`:

```
// StP -- created on Wed Sep 17 11:12:09 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Routing_table_DCC_idl_
#define _Routing_table_DCC_idl_

// stp class declarations
interface Routing_table_DCC;
// stp class declarations end

// stp auto-include 187
#include "Routing_table.idl"
// stp auto-include end

// stp class definition 187
interface Routing_table_DCC : Routing_table
{
// stp class members
    attribute unsigned long bit_length;
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Routing_table_ICD`:

```
// StP -- created on Wed Sep 17 11:13:30 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Routing_table_ICD_idl_
#define _Routing_table_ICD_idl_

// stp class declarations
interface Routing_table_ICD;
// stp class declarations end

// stp auto-include 188
#include "Routing_table.idl"
// stp auto-include end

// stp class definition 188
interface Routing_table_ICD : Routing_table
{
```

```
// stp class members
    attribute unsigned long bit_length;
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Error_Trap`:

```
// StP -- created on Wed Sep 17 11:07:04 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Error_Trap_idl_
#define _Error_Trap_idl_

// stp class declarations
interface Error_Trap;
// stp class declarations end

// stp class definition 164
interface Error_Trap
{
// stp class members
    attribute string trap_error_code;
    attribute boolean send_to_XAD;
    attribute boolean send_to_SNMP;
    attribute boolean send_to_error_rep;
    attribute boolean send_to_CORBA;
     void new_trap(string trap_error_code, boolean send_to_XAD, boolean send_to_SNMP, boolean
send_to_error_rep, boolean send_to_CORBA);
   void change_trap(string trap_error_code, boolean send_to_XAD, boolean send_to_SNMP, boolean
send_to_error_rep, boolean send_to_CORBA);
    void delete_trap(string trap_error_code);
    void view_trap(string trap_error_code);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Authent`:

```
// StP -- created on Wed Sep 17 10:43:49 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Authent_idl_
#define _Authent_idl_

// stp class declarations
interface Authent;
// stp class declarations end

// stp class definition 159
interface Authent
{
// stp class members
    attribute boolean configuration;
    attribute boolean fault_mgmt;
    attribute boolean installation;
    attribute boolean security;
    attribute string identification;
     view_authentification(string identification);
     new_authentification(string identification, boolean configuration, boolean fault_mgmt, boolean
installation, boolean security);
     change_authentification(string identification, boolean configuration, boolean fault_mgmt, boo-
lean installation, boolean security);
     delete_authentification(string identification);
// stp class members end
};
// stp class definition end
// stp footer
#endif
// stp footer end
```

Klasse `Error_Log`:

```
// StP -- created on Wed Sep 17 11:05:54 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Error_Log_idl_
#define _Error_Log_idl_

// stp class declarations
interface Error_Log;
// stp class declarations end
// stp class definition 170
interface Error_Log
{
```

```
// stp class members
   void view_error_log_entry(string Object_reference);
   void refresh_error_log();
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Klasse `Error_Rep`:

```
// StP -- created on Wed Sep 17 11:06:18 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Error_Rep_idl_
#define _Error_Rep_idl_

// stp class declarations
interface Error_Rep;
// stp class declarations end

// stp class definition 174
interface Error_Rep
{
// stp class members
   void view_error_rep_entry(string Object_reference);
// stp class members end
};
// stp class definition end
// stp footer
#endif
// stp footer end
```

Klasse `Error_Entry`:

```
// StP -- created on Wed Sep 17 10:44:33 1997 for hoelsche@hpp022 from system XPRESS_NM

#ifndef _Error_Entry_idl_
#define _Error_Entry_idl_

// stp class declarations
interface Error_Entry;
// stp class declarations end
```

```
// stp class definition 176
interface Error_Entry
{
// stp class members
    readonly attribute string Object_reference;
    readonly attribute string hardware_address;
    readonly attribute integer date;
    readonly attribute string error_number;
    readonly attribute string error_code;
    readonly attribute integer error_text;
    readonly attribute integer alarm_type;
    readonly attribute integer perceived_severity;
    readonly attribute integer reaction_code;
    readonly attribute string module_version;
    readonly attribute integer module_line;
    void view_error_entry(string Object_reference);
    void delete_error_entry(string Object_reference);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Abkürzungsverzeichnis

AAL	ATM Adaption Layer
ABR	Available Bitrate
ANIS	ATM Narrowband Interface S2
ANSI-T1	American National Standards Institute - Working Group T1
API	Application Programming Interface
APTM2	ATM Port Trunk Module (2 Ports)
ASN32	ATM Switching Network (32 x 32 Matrix)
ATM	Asynchronous Transfer Modus
BOA	Basic Object Adapter
B-ISDN	Broadband - Integrated services digital network
CB	Control Block
CBR	Constant Bitrate
CCITT	Comité Consultatif International Télégraphique et Téléphonique
CGA2	Clock Generator ATM (Release 2)
CLP	Cell Loss Priority
CNET	Chief of Naval Education & Training
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
CS	Convergence Sublayer
CTE	Class Table Editor
DCC	Data Country Code
DCE	Data circuit terminating equipment
DCE-CIOP	DCE Common Inter-ORB Protocol
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
EAC	European Activities Committee
EMAC	European Market Awareness & Education Comittee
ESIOP	Environment-Specific Inter-ORB Protocol
ETSI	European Telecommunication Standards Institute
GFC	Generic Flow Control
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
HEC	Header Error Control
ICC	International Code Designator
IDE	Interactive Development Environments
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IM	Implementation Repository
IR	Interface Repository
ITU	International Telecommunication Union (früher CCITT)
LAN	Local Area Network
LANE	Local Area Network Emulation

L2M	Layer 2 Unit
NNI	Network Network Interface
NSAP	Network Service Access Point
OA	Object Adapter
OAE	Object Annotation Editor
OAM	Operation, Administration and Maintenance
ODP	Open Distributed Processing
OMA	Object Management Architecture
OME	Object Model Editor
OMG	Object Management Group
OMT	Object Modeling Technique
OODB	Object-Oriented Database Adapter
ORB	Object Request Broker
OSF	Open Software Foundation
PBX	Private Branch Exchange
PM	Physical Medium Sublayer
PNNI	Private Network/Node to Network Interface
PVC	Permanent Virtual Circuit
PW	Processing Window
QoS	Quality of Service
RM	Requirement Model
RM-ODP	Reference Model of Open Distributed Processing
SAR	Segmentation and Reassembly Sublayer
SB	Switching Block
SNMP	Simple Network Management Protocol
STM1	Synchronous Transport Module Type 1 (155 Mbit/s)
StP	Software through Pictures
SVC	Switched Virtual Circuit
TC	Transmission Convergence Sublayer
TCP/IP	Transport Control Protocol / Internet Protocol
TP	Twisted Pair
TTC	Technical Telecommunication Center
UCE	Use Case Editor
UCM	Use Case Model
UNI	User Network Interface
VBR	Variable Bitrate
VC	Virtual Channel
VCC	Virtual Channel Connection
VCI	Virtual Channel Identifier
VP	Virtual Path
VPI	Virtual Path Identifier
VW	View Window
WSA	Workstation Adapter
XAD	Hicom Xpress Administration

Literaturverzeichnis

[ATMFO] Homepage des ATM Forums, http://www.atmforum.com

[BDM94] M. Boisseau, M. Demange, J.-M. Munier, High-Speed Networks, John Wiley & Sons, 1994

[COOMG] CORBA and OMG Information Resources (Spezifikationen, Bücher, Applikationen usw.), http://www.acl.lanl.gov/CORBA

[EVB96] Edward V. Berard, Be Careful with “Use-Cases”, The Object Agency, Inc., May 1996, http://www.toa.com/pub/html/use-case.html

[HAR96] Rüdiger Hartmann, Netze ohne Grenzen, Artikel aus der c’t, S. 346 - 354, Oktober 1996

[HHS94] Rainer Händel, Manfred N. Huber, Stefan Schröder, ATM-Networks: Concepts, Protocols, Applications, Addison-Wesley, 1994

[HOAON] Homepage von Aonix mit Produktübersicht über StP, http://www.aonix.com/Products/StP/stp.html

[HOBOR] Homepage von Borland mit Übersicht über den JBuilder, http://www.borland.com/jbuilder/jbcsfact/

[HOITU] Homepage der ITU, http://www.itu.int

[HOOMG] Homepage der OMG, http://www.omg.org

[HVL97] Mathias Hein, Nickolaus von der Lancken, ATM: Konzepte - Trends - Migration, International Thomson Publishing, 1997

[ICO96] Working with OMT-Part I: Objects From Use-Cases, ICON Computing, 1994 - 1996, http://www.iconcomp.com/papers/omt1-usecases/OMT1-UseCases.frm.html

[ION96] IONA Technologies, The Orbix Architecture, Articel, November 1996, http://www.iona.com/Products/Orbix/Architecture/index.html

[JAC92] Ivar Jacobsen et al., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992

[ORP97] IONA Technologies, Orbix 2: Programming Guide, March 1997

[ORR97] IONA-Technologies, Orbix 2: Reference Guide, March 1997

[ORW97] IONA-Technologies, OrbixWeb V2.0.1, Product Overview, February 1997, http://www.iona.com/Products/Orbix/OrbixWeb/Overview.html

[PRY94] Martin de Prycker, Asynchronous Transfer Mode: Die Lösung für Breitband-ISDN, Prentice Hall, 1994

[RED96] Jens-Peter Redlich, CORBA 2.0: Praktische Einführung für C++ und Java, Addison-Wesley, 1996

[RUM93] Rumbaugh et al., Objektorientiertes Modellieren und Entwerfen, Hanser-Verlag, 1993

[SAI94] Hiroshi Saito, Teletraffic Technologies in ATM Networks, Artech House Inc., 1994

[SCM97] Douglas C. Schmidt, Overview of CORBA, Articel, July 1997, http://siesta.cs.wustl.edu/~schmidt/corba-overview.html

[SIE95] Siemens AG, HicomXpress Administration (XAD), Komplexspezifikation, Version 2, November 1995

[SIE97] Siemens AG, ATM-Switch Hicom Xpress, Servicehandbuch, März 1997, P30122-X5501-X-9-19

[SIE95] Siemens AG, HicomXpress Administration, Bedienungshandbuch, Version 4, 1995

[SNATM] Standards und Normen bezüglich ATM (Drafts, RFCs), http://www.lirmm.fr/atm/normes.html

[SPC96] Interactive Development Environments, Inc., Software through Pictures / Object Modeling Technique - Creating OMT Models, Chapter 1 - 6 and 11 - 13, Release 3, February 1996

[SPF96] Software through Pictures / StP Core - Fundamentals of StP, Chapter 1 - 5 and 11, Release 2, February 1996

[SPG96] Interactive Development Environments, Inc., Software through Pictures / Object Modeling Technique - Generating Code, Chapter 7, Release 3, February 1996

[SPO96] Interactive Development Environments, Inc., Software through Pictures / Object Modeling Technique - OMT Handbook, Chapter 1 - 5, Release 3, November 1996

[SPS96] Interactive Development Environments, Inc., Software through Pictures / Object Modeling Technique - Getting Started with StP/OMT, Chapter 1 - 4, Release 3, February 1996

[SSC94] Steffen Schäfer, Objektorientierte Entwurfsmethoden: Verfahren zum objektorientierten Softwareentwurf im Überblick, Addison-Wesley, 1994