

Universität Stuttgart

Fakultät Informatik

Studienarbeit Nr. 1749

Grundlegende Überarbeitung der Teile des Java Visualisierungs- baustens für Protokolle

Torsten Brodbeck

Betreuer:

Dr. Cora Burger

Dipl.-Inform. Rolf Mecklenburg

Prüfer:

Prof. Dr. Kurt Rothermel

Begonnen am: 01.04.1999

Beendet am: 30.09.1999

CR-Klassifikation: C.2.2, H.5.1, I.3.2, I.6.8, K.3.1, K.3.2



Breitwiesenstr. 20 - 22



D-70565 Stuttgart

Tel. : +49 - 711 / 7816 - 448

Fax : +49 - 711 / 7816 - 424

Universität Stuttgart
Institut für Parallele und Verteilte
Hochleistungsrechner (IPVR)

Verteilte Systeme (VS)

Projekt HiSAP

Kurzfassung

Im HiSAP Projekt soll ein Baukasten entwickelt werden, mit dem Java Applets bzw. Applikationen erstellt werden können, die Kommunikationsprotokolle bzw. Algorithmen für Lehrzwecke visualisieren. In mehreren bereits abgeschlossenen Arbeiten wurden Teile des Java Visualisierungsbaukastens entwickelt. Im Rahmen dieser Studienarbeit wurde der Baukasten grundlegend überarbeitet. Dabei ist ein komplett neuer Baukasten entstanden, bei dem Grundkonzepte und Strukturen des alten Baukastens übernommen wurden und der zusätzlich um die Möglichkeit, Simulationsmodelle zu erstellen, erweitert wurde. Neben Entwurf und Implementation wurde die Anwendung des neuen Baukastens beschrieben und anhand eines asymmetrischen Zweiweg-Authentifikationsprotokollbeispiels veranschaulicht.

Inhaltsverzeichnis

1 Einleitung.....	1
2 Überblick über das HiSAP Projekt.....	4
2.1 Ziele des HiSAP Projektes.....	4
2.1.1 Geeignete Visualisierung.....	6
2.1.2 Interaktionsmöglichkeit.....	13
2.1.3 Wiederverwendbarkeit.....	15
2.2 Bisherige Entwicklung.....	17
2.2.1 Visualisierungsbaukasten.....	17
2.2.2 Generator für Simulationsmodelle.....	21
2.3 Fazit.....	22
3 Überarbeitung des Baukastens.....	23
3.1 Analyse.....	23
3.1.1 Anforderungen an den neuen Baukasten.....	23
3.1.2 Mögliche zukünftige Anforderungen.....	24
3.2 Entwurf.....	25
3.2.1 Grundsätzliche Entwurfsentscheidungen.....	25
3.2.2 Klassen zur Erzeugung von Simulationsmodellen.....	28
3.2.3 Schnittstellen.....	41
3.2.4 Klassen zur Visualisierung.....	42
3.3 Implementation.....	48
3.3.1 HiSAP CoreFoundationClasses.....	48
3.3.2 HiSAP ProtoVisFoundationClasses.....	52
4 Anwendung des neuen Baukastens.....	53
4.1 Grundprinzipien.....	53
4.1.1 Grundstruktur der Protokollbeispiele.....	53
4.1.2 Erstellung von Simulationsmodellen.....	54
4.1.3 Verwendung von Visualisierungskomponenten.....	56
4.2 Ein Anwendungsbeispiel.....	61
4.2.1 Asymmetrische Zweiweg-Authentifikation.....	61
4.2.2 Simulationsmodell.....	63
4.2.3 Graphischer Aufbau des Simulationsmodells.....	72
5 Zusammenfassung und Ausblick.....	80
Anhang.....	82
Literaturverzeichnis.....	88

Abbildungsverzeichnis

Abbildung 2.1: Textbasierte Darstellung.....	7
Abbildung 2.2: Entscheidungstabelle.....	7
Abbildung 2.3: Automatendarstellung.....	9
Abbildung 2.4: Topologiedarstellung.....	10
Abbildung 2.5: Time–Sequence–Diagram.....	11
Abbildung 2.6: Benutzungsschnittstelle mit Swing–Komponenten.....	12
Abbildung 2.7: RPC–Beispiel mit dem Baukasten von Schurr	18
Abbildung 2.8: Benutzungsschnittstelle von Papoulidis.....	19
Abbildung 2.9: IdleRQ–Beispiel mit dem Baukasten von Rau.....	20
Abbildung 3.1: Komponenten zur Erzeugung von Simulationsmodellen.....	28
Abbildung 3.2: FullDuplex–Connection.....	31
Abbildung 3.3: Schichtenkonzept.....	32
Abbildung 3.4: Verbindung mit mehr als zwei Endpunkten.....	33
Abbildung 3.5: Überholen von Nachrichten (Variante 1).....	33
Abbildung 3.6: Überholen von Nachrichten (Variante 2).....	34
Abbildung 3.7: Nachrichtenduplikate.....	35
Abbildung 3.8: Dynamische Zuweisung von Kommunikationsprotokollen.....	36
Abbildung 3.9: Hierarchie verbindbarer Elemente.....	37
Abbildung 3.10: LAN–Bereich (abstrakt).....	37
Abbildung 3.11: LAN–Bereich (detailliert).....	38
Abbildung 3.12: Mögliche Hierarchie einer Message.....	38
Abbildung 3.13: Entwurfsänderung der Klasse Message.....	38
Abbildung 3.14: Entkopplung von Abstraktion und Implementation.....	39
Abbildung 3.15: Schnittstellen.....	41
Abbildung 3.16: Klasse JArrowView.....	43
Abbildung 3.17: Klasse JConnectionView.....	44
Abbildung 3.18: Klasse JFullDuplexConnectionView.....	45
Abbildung 3.19: Automatische Visualisierung durch die Klasse JHistoryView... ..	46
Abbildung 3.20: Klasse JHistoryView.....	46
Abbildung 3.21: Router mit vier unabhängigen Puffern.....	49
Abbildung 3.22: Erzeugung einer hierarchischen Ack–Nachricht.....	51
Abbildung 4.1: Grundstruktur der init–Methode.....	54
Abbildung 4.2: Erzeugen von Nodes.....	55
Abbildung 4.3: Erzeugen von Connections.....	56
Abbildung 4.4: Erzeugung einer Benutzungsschnittstelle.....	57
Abbildung 4.5: Visualisierung der Knoten.....	58
Abbildung 4.6: Visualisierung von Verbindungen.....	59
Abbildung 4.7: Interne Informations– und Manipulationsfenster.....	60
Abbildung 4.8: Asymmetrische Zweiweg–Authentifikation.....	62
Abbildung 4.9: Mögliche Verbindungen des Anwendungsbeispiels.....	63
Abbildung 4.10: Mögliche Attacken.....	64
Abbildung 4.11: Verwendete Verbindungen des Anwendungsbeispiels.....	65
Abbildung 4.12: Anwendungsbeispiel mit Angreifer.....	65
Abbildung 4.13: Nachricht (M1).....	66
Abbildung 4.14: Authentifikations– und Routingprotokoll initialisieren.....	70
Abbildung 4.15: Knoten initialisieren.....	71
Abbildung 4.16: Verbindungen initialisieren.....	71
Abbildung 4.17: Erstellung mit Hilfe eines visuellen Entwicklungswerkzeugs.....	73
Abbildung 4.18: Benutzungsschnittstelle erzeugen und festlegen.....	74
Abbildung 4.19: Visualisierungskomponenten im Graphenbereich.....	75
Abbildung 4.20: Authentifikationsserver Fenster.....	76
Abbildung 4.21: Authentifikationsclient Fenster.....	77

Abbildung 4.22: Angreifer Fenster.....	77
Abbildung 4.23: Ausschnitt aus der Methode actionPerformed.....	79
Abbildung A.1: CoreFoundationClasses.....	83
Abbildung A.2: ProtoVisFoundationClasses.....	84
Abbildung A.3: Nachricht (M1).....	85
Abbildung A.4: Nachricht (M2).....	85
Abbildung A.5: Nachricht (M3).....	85
Abbildung A.6: Nachricht (M4).....	85
Abbildung A.7: Nachricht (M5).....	86
Abbildung A.8: Nachricht (M6).....	86
Abbildung A.9: Nachricht (M7).....	86
Abbildung A.10: Asymmetrisches Zweiweg-Authentifikationsprotokoll-Applet.....	87

1 Einleitung

In den Bereichen *Rechnernetze*, *Verteilte Systeme* und *Kooperation in Verteilten Systemen* basieren die Lehrinhalte hauptsächlich auf dem Austausch von Nachrichten zwischen Komponenten über einen Kommunikationskanal, wobei sich die Komponenten an ein bestimmtes Kommunikationsprotokoll halten. In den letzten Jahren haben die Kommunikationsprotokolle deutlich an Komplexität zugenommen. Die wachsende Bedeutung von Kryptographie in Kommunikationsprotokollen z.B. hat einen wesentlichen Teil dazu beigetragen. Somit fällt es Studenten zunehmend schwerer, Kommunikationsprotokolle zu verstehen.

Um das Erlernen von Kommunikationsprotokollen zu erleichtern, versucht die Abteilung *Verteilte Systeme* der Universität Stuttgart, Kommunikationsprotokolle geeignet darzustellen. Die verschiedenen Ablaufvarianten eines Kommunikationsprotokolls lassen sich z.B. sehr gut mit Hilfe einer Animation und der Möglichkeit zur Beeinflussung durch den Benutzer (z.B. Entfernen einer Nachricht vom Kanal) durchspielen. Um möglichst viele Kommunikationsprotokolle mit Hilfe von Animationen darstellen zu können, wird ein Baukasten zur Erzeugung von Kommunikationsprotokoll-Beispielen angestrebt. Dadurch entstand das Projekt HiSAP.

In mehreren bereits abgeschlossenen Arbeiten wurden Teile des Baukastens entworfen und in der Programmiersprache Java realisiert. Da die Programmiersprache Java zu Beginn dieses Projekts noch sehr jung war, erfuhr sie in der Zwischenzeit so enorme Änderungen, daß einige entwickelte Teile des Baukastens nun nicht mehr einsatzfähig sind.

Aus diesem Grund wurde diese Studienarbeit vergeben. Es gilt eine grundlegende Überarbeitung des Baukastens durchzuführen, somit den Baukasten auf den neuesten Stand der Programmiersprache Java zu bringen und eventuell Teile des vorhandenen Baukastens zu übernehmen.

Aufgabenstellung

Die Aufgabenstellung besteht aus folgenden Teilaufgaben:

1. Einarbeitung in die Themengebiete:
 - aktuelle Fähigkeiten von Java inkl. der Swing–Komponenten
 - bestehender Visualisierungsbaukasten und seine Ergänzungen
 - bestehende Werkzeuge zur Generierung von Simulationsmodellen¹
2. Beschreibung der aktuellen Fähigkeiten des Baukastens.
3. Spezifikation, Design und Realisierung eines neuen, integrierten Visualisierungsbaukastens: Hierzu gehört eine Beschreibung, wie sich die vorhandenen Arbeiten in den neuen Baukasten integrieren lassen. Außerdem soll hier eine neue Benutzungsschnittstelle beschrieben werden. Es ist nicht zwingend notwendig, daß alle Komponenten des Baukastens in dieser Studienarbeit realisiert werden.
4. Spezifikation, Design und Realisierung eines Baukastens zur Erzeugung von Simulationsmodellen: Dieser Baukasten enthält alle Grundkomponenten (zumindest in abstrakter Form) für eine Simulation von Protokollen. Außerdem muß hier beschrieben werden, wie sich die bereits existierenden Ansätze zur Erzeugung von Simulationsmodellen integrieren lassen. Zu diesem Aufgabenblock gehört auch die prototypische Implementation der oben beschriebenen Komponenten, damit ein Anwendungsbeispiel erzeugt werden kann.

¹ In der Informatik bezeichnet Simulation die Nachbildung von Vorgängen auf einer Rechenanlage auf der Basis von Simulationsmodellen. Wobei diese die wesentlichen Eigenschaften der zu simulierenden Vorgänge und ihre gegenseitige Beeinflussung widerspiegeln. Alle Ergebnisse einer Simulation beziehen sich nur auf das verwendete Simulationsmodell. Inwieweit solche Ergebnisse auf die Wirklichkeit übertragen werden können, hängt daher entscheidend davon ab, wie gut die Wirklichkeit durch das Modell nachgebildet wird [Eng93].

Überblick

Im folgenden Kapitel werden die Ziele des HiSAP Projektes erläutert und ein kurzer Überblick über existierende Arbeiten, die im Rahmen dieses Projektes entstanden, gegeben.

Der Hauptteil dieser Studienarbeit wird im Kapitel 3 behandelt, das in Analyse, Entwurf und Implementation gegliedert ist. Die Analyse befaßt sich mit den Defiziten des alten Baukastens und bestimmt die zu überarbeitenden Teile des Java Visualisierungsbaukastens. Danach folgt der konkrete Entwurf und die Implementation des neuen Baukastens.

Kapitel 4 beschreibt die Anwendung des neuen Baukastens. Zuerst wird ein Überblick über die Grundprinzipien gegeben, danach die Verwendung durch ein konkretes Anwendungsbeispiel veranschaulicht.

Schließlich werden im Kapitel 5 eine Zusammenfassung und ein Ausblick gegeben.

2 Überblick über das HiSAP Projekt

In diesem Kapitel werden die momentanen Anforderungen bzw. Ziele des HiSAP Projektes erläutert und eine kurze Beschreibung vorhandener Arbeiten gegeben.

2.1 Ziele des HiSAP Projektes

Hauptziel des *HiSAP* (**H**ighly **i**nteractive **S**imulation of **A**lgorithm and **P**rotocols) Projektes ist eine rechnergestützte interaktive Visualisierung von Algorithmen und Protokollen. Zu diesem Zweck soll ein Baukasten entwickelt werden, der Algorithmen und Protokolle geeignet darstellt und gleichzeitig **Interaktionen** des Benutzers unterstützt. Da die **geeignete Darstellung** sich nach den jeweiligen Anforderungen der Zielgruppen richtet, muß dieser Baukasten die momentan angestrebten Zielgruppen berücksichtigen. Zusätzlich soll der Baukasten eine **hohe Flexibilität** bieten, um jederzeit Erweiterungen auf andere Zielgruppen zu ermöglichen.

Im folgenden werden zwei, in diesem Projekt anvisierte, Zielgruppen beschrieben. Eine dieser Zielgruppen sind die Lehrenden und die Lernenden: Zum Einen soll der Baukasten den Lehrenden darin unterstützen, Algorithmen bzw. Protokolle so darzustellen, daß diese von den Lernenden besser und schneller verstanden werden. Dabei soll der Baukasten die Möglichkeit bieten, während des Vortrages durch Interaktionen des Lehrenden bestimmte Abläufe der Algorithmen bzw. Protokolle hervorzuheben. Zum Anderen soll der Baukasten dem Lernenden ermöglichen, die vorgestellten Algorithmen bzw. Protokolle jederzeit wieder zu betrachten, und gleichzeitig mittels interaktiver Benutzung verschiedene Abläufe durchzuspielen. Somit kann der Lernende sein Wissen vertiefen bzw. Unklarheiten beseitigen.

Die zweite Zielgruppe sind Wissenschaftler, die entweder neue Algorithmen bzw. Protokolle entwickeln und danach testen oder bereits existierende

Algorithmen bzw. Protokolle auf bestimmte Situationen testen wollen. Hier soll der Baukasten die Möglichkeit bieten, Algorithmen bzw. Protokolle mit Hilfe von Spezifikationsprachen zu beschreiben und damit ein Simulationsmodell aufzubauen, das die gewünschte Realität nachbildet. Zusätzlich soll das Simulationsmodell so visualisiert werden, daß Schwächen und Probleme der Algorithmen bzw. Protokolle erkannt werden. Dabei sollten jederzeit durch Interaktionen des Benutzers Änderungen am Simulationsmodell vorgenommen werden können.

Da die Zielgruppen möglicherweise unterschiedliche Betriebssystemplattformen verwenden, soll der Baukasten plattformunabhängig entwickelt werden. Zusätzlich bietet das Internet eine ideale Basis, um *jederzeit* und *überall* auf den Baukasten zuzugreifen. Daher wird in dem HiSAP Projekt auf die Programmiersprache und Plattform Java gesetzt.

Der Focus dieser Studienarbeit richtet sich hauptsächlich auf die erste der obengenannten Zielgruppen. Deshalb werden nun folgend die drei wesentlichen Anforderungen des HiSAP Projektes im Hinblick auf Lehrzwecke genauer betrachtet. Es wird dabei untersucht, inwieweit diese Anforderungen zu erfüllen sind; für die Anforderung *geeignete Visualisierung* wird betrachtet, welche Visualisierungsformen besonders für Lehrzwecke geeignet sind. Für die nächste Anforderung *Interaktionsmöglichkeit* wird differenziert, welche Interaktionsformen dem Verständnis der Protokollbeispiele dienen, ohne daß der Benutzer das Protokollbeispiel zu sehr verfälscht. Danach wird auf die letzte und ebenfalls wichtige Anforderung des HiSAP Projektes eingegangen, die Flexibilität bzw. *Wiederverwendbarkeit* des Baukastens.

2.1.1 Geeignete Visualisierung

Es gibt eine Vielzahl an Möglichkeiten, Kommunikationsprotokolle zu visualisieren. Um die wesentlichen Visualisierungsformen für Lehrzwecke sondieren zu können, werden zuerst typische Visualisierungsmöglichkeiten aufgezählt und darauf untersucht, für welchen Zweck diese besonders gut geeignet sind. Danach werden diese Visualisierungsmöglichkeiten im Hinblick auf die erste Zielgruppe (Lehrende und Lernende) des HiSAP Projektes bewertet.

2.1.1.1 Typische Visualisierungsmöglichkeiten

In dieser Studienarbeit werden unter Visualisierungsmöglichkeiten jegliche Darstellungsformen verstanden, die Kommunikationsprotokolle mit Hilfe eines Programmes auf dem Bildschirm beschreiben. Somit sind auch Textausgaben inbegriffen. Folglich werden die Visualisierungsmöglichkeiten grob in zwei Darstellungsformen unterteilt, nämlich in textbasierte Darstellung und in graphische Darstellung.

Textbasierte Darstellung

Die textbasierte Darstellung ist die einfachste Darstellungsform, Kommunikationsprotokolle zu beschreiben. Es bedarf jedoch einer guten Abstraktionsfähigkeit bzw. Vorstellungskraft, um vollständig textuell beschriebene Kommunikationsprotokolle zu verstehen. Der Vorteil einer textbasierten Darstellung ist die Möglichkeit, auf engem Raum die relevantesten Informationen zu liefern.

Typischerweise werden textbasierte Informationen für Kommunikationsprotokolle in drei Varianten eingesetzt. Die einfachste Variante ist die natürliche Sprache (Deutsch, Englisch usw.). Man beschreibt die relevanten Informationen in natürlichen Worten. Diese Variante ist besonders sinnvoll, wenn die Information ähnlich der normalen Kommunikation zwischen

Menschen ist. So ist es beispielsweise naheliegend, in einem eMail-Protokoll den Inhalt der Nachrichten wie in Abbildung 2.1 darzustellen.

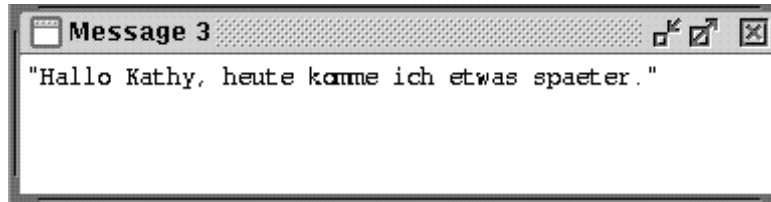


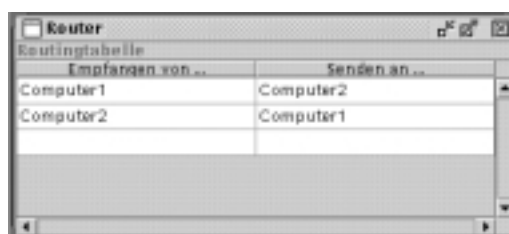
Abbildung 2.1: Textbasierte Darstellung

Eine andere Darstellungsvariante textbasierter Informationen ist die künstliche Sprache. Künstliche Sprachen werden nach Regeln aufgebaut (Syntax) und ihre Wörter und Sätze besitzen eine wohldefinierte Bedeutung (Semantik). Die Syntax wird dabei durch formale Sprachen beschrieben [Eng93].

Die künstliche Sprache hat somit gegenüber der natürlichen Sprache den Vorteil, den Wortschatz auf das Wesentliche zu beschränken und sie vermeidet durch die eindeutige Semantik unnötige Mißverständnisse. Wenn jedoch die Sprache nicht aus anderen Gründen schon erlernt wurde, bzw. weit verbreitet ist, muß sie speziell für diesen Zweck erlernt werden.

In der Informatik werden häufig künstliche Sprachen angewandt (Programmiersprachen, Specification Description Language usw.). Aus diesem Grund können in manchen Fällen künstliche Sprachen zum Lehren von Kommunikationsprotokollen eingesetzt werden.

Eine ebenfalls häufig angewandte Variante ist die Textdarstellung in tabellarischer Form. Die Tabellendarstellung zeigt viele Informationen übersichtlich auf engem Raum. In erster Linie sind die Informationen dadurch sinngemäß strukturiert. Abbildung 2.2 zeigt anhand einer Entscheidungstabelle ein Routingprotokoll.



Empfangen von...	Senden an...
Computer1	Computer2
Computer2	Computer1

Abbildung 2.2: Entscheidungstabelle

Graphische Darstellung

Graphische Darstellungsformen sind häufig hilfreich für besseres Verständnis und Einprägsamkeit von Informationen. Dies trifft jedoch nicht bei jedem Menschen zu, denn entsprechend individueller Neigung kann Information entweder textuell oder graphisch am effektivsten wahrgenommen werden [Sta94]. Es kann jedoch angenommen werden, daß die Mehrheit der Menschen bei graphischer Darstellung Informationen am effektivsten wahrnimmt.

Zur Darstellung von Kommunikationsprotokollen in graphischer Form werden typischerweise abstrakte Modelle eingesetzt, die die gewünschten Informationen hervorheben. Die in der Informatik am häufigsten genutzten Modelle für Kommunikationsprotokolle sind Graphen- und Sequenzdiagrammdarstellungen, dabei erfordern manche Modelle eine hohe Abstraktion bzw. Vorstellungskraft.

Graphendarstellung

Graphen sind anschauliche mathematische Modelle zur Beschreibung von Objekten, die untereinander in gewisser Beziehung stehen [Eng93]. Graphendarstellungen sind in der Informatik für Kommunikationsprotokolle sehr beliebt und gebräuchlich, da jedes Kommunikationsprotokoll mit Hilfe von Graphen dargestellt werden kann.

Manche Graphendarstellungen sind jedoch sehr abstrakt und werden ab einer gewissen Größe unübersichtlich, aus diesem Grund muß gesondert geprüft werden, welche Graphendarstellungen für welche Lehrzwecke besonders gut geeignet sind. Graphendarstellungen, die hohe Abstraktion erfordern, sollten nur für jene Lernenden eingesetzt werden, die mit dieser Darstellungsform vertraut sind. Sonst werden Kommunikationsprotokolle nicht effektiv wahrgenommen und die Wahrscheinlichkeit, daß die Protokolle nicht vollständig verstanden werden, steigt. Petri-Netze und endliche Automaten gehören z.B. zu diesen Graphendarstellungen, die eine höhere

Abstraktion bzw. Vorstellungskraft erfordern.

Diese Graphendarstellungen sind eher für wissenschaftliche Zwecke geeignet, da sie sehr stark an die Mathematik angelehnt sind und Kommunikationsprotokolle deshalb gut untersucht werden können. Zur Veranschaulichung zeigt Abbildung 2.3 einen Ausschnitt aus dem als endlicher Automat dargestellten "Stop-and-Wait" Protokoll, das als Anwendungsbeispiel in der Studienarbeit von Jürgen Rau für das HiSAP Projekt realisiert wurde.

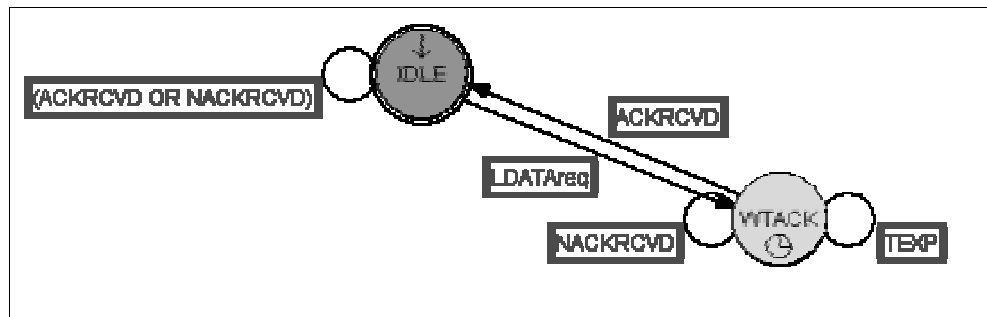


Abbildung 2.3: Automatendarstellung

Besonders gut zur Lehre von relativ vielen Kommunikationsprotokollen ist die Topologiedarstellung geeignet. Der Begriff Topologie wird allgemein als wichtiges Teilgebiet der modernen Geometrie verstanden. Sie untersucht den Zusammenhang und die gegenseitige Lage geometrischer Gebilde ohne Rücksicht auf deren im Endlichen liegende Verzerrungen, Maßverhältnisse, Winkel, Längen usw [Her87]. Dies bedeutet, daß die Verbindungsstrukturen der Rechnernetze bei der Topologiedarstellung größtenteils wie in der Realität abgebildet werden, und deshalb eine Abstraktion nur in geringem Maße erfordert ist. Zusätzlich können Abläufe von Kommunikationsprotokollen durch die in den Maßverhältnissen verzerrte Darstellung auf engem Raum visualisiert werden. Damit ermöglicht diese Darstellung eine für bestimmte Information hervorgehobene und trotzdem realitätsnahe Abbildung. So werden z.B. bei einem Token-Ring Protokoll die Computer auf engem Raum, in einem Ring dargestellt (Abbildung 2.4), um dieses Protokoll übersichtlich und anschaulich zu präsentieren. Die einzige Schwierigkeit

dieser Darstellungsform ist die Entscheidung, wo Verzerrungen angebracht sind oder die reale Abbildung beibehalten werden soll.

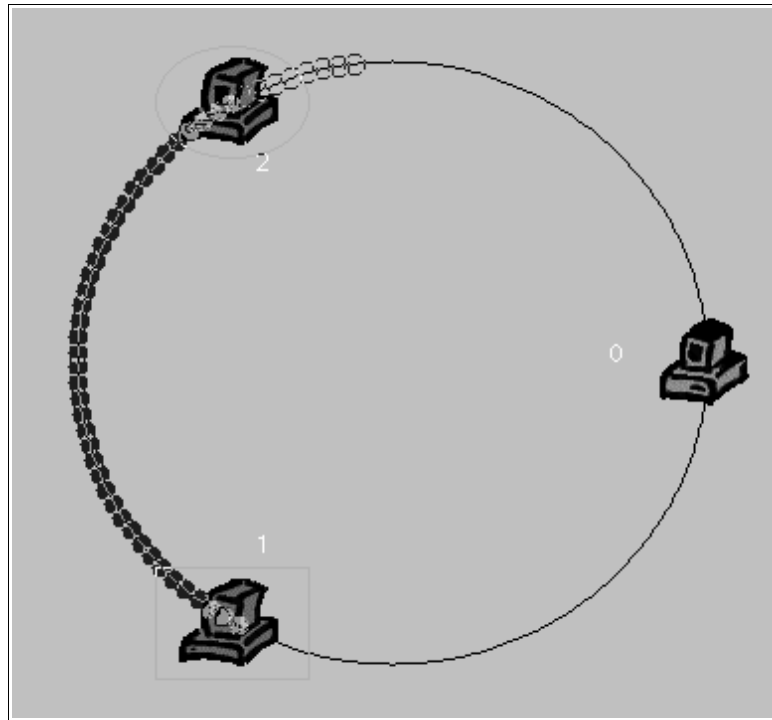


Abbildung 2.4: Topologiedarstellung

Sequenzdiagrammdarstellung

Sequenzdiagramme zeigen sehr gut die Ablaufsequenz von Kommunikationsprotokollen. Dabei wird die Sequenz entweder in Abhängigkeit der Zeit (Time-Sequence-Diagram) oder bestimmten Interaktionen (Interaction-Diagram), wie z.B. Nachrichten (Message-Sequence-Diagram), angezeigt. Sequenzdiagramme müssen nicht animiert werden und ermöglichen eine gute Darstellung von Historien. Deshalb sind Sequenzdiagramme beim Zurückverfolgen von Kommunikationsprotokollabläufen bestens geeignet. Kommunikationskanäle sind in Sequenzdiagrammen nicht ersichtlich. Das bedeutet, daß solche Diagramme keine Information über die Verbindungsstruktur der Komponenten bietet. Deswegen ist für Lehrzwecke eine alleinige Sequenzdiagrammdarstellung für Protokollbeispiele nicht immer geeignet.

Abbildung 2.5 zeigt beispielhaft mit Hilfe eines Time–Sequence–Diagrams die Historie eines Anwendungsbeispiels von Schurr[Sch97], das einen Broadcast–Algorithmus demonstriert.

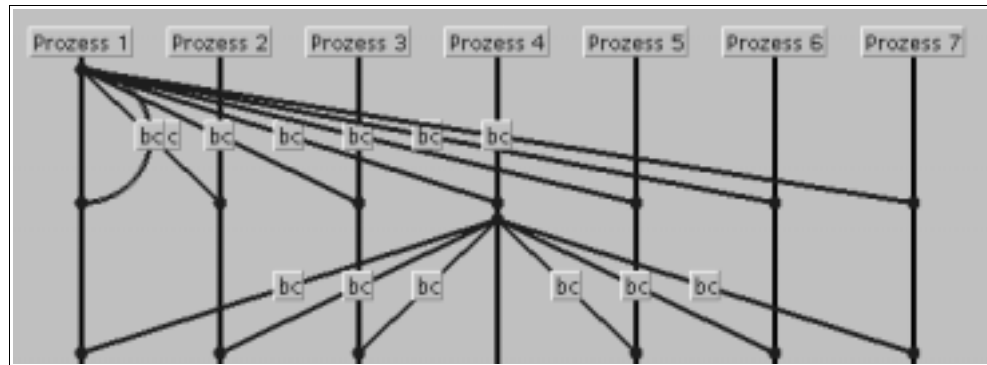


Abbildung 2.5: Time–Sequence–Diagram

2.1.1.2 Bewertung

Zusammenfassend ist festzustellen, daß alle besprochenen Visualisierungsmöglichkeiten für die Lehre geeignet sind. Wann welche Darstellung besser geeignet ist, richtet sich nach den interessierenden Aspekten eines Protokollbeispiels.

Um eine generelle Aussage treffen zu können, wäre eine weitere Studien- bzw. Diplomarbeit sinnvoll, die ausführlich alle Visualisierungsmöglichkeiten zusammen mit ihren möglichen Formen auf den effektivsten Lernerfolg untersucht um nachfolgend Empfehlungen für die Konstruktion von Protokollbeispielen geben zu können.

Generell und unabhängig vom Ergebnis einer solchen Arbeit kann gesagt werden, daß für die Lehre eine gleichzeitige Sequenzdiagramm- und Graphendarstellung für Kommunikationsprotokolle zu empfehlen ist. Weiterhin sollte die Möglichkeit bestehen, Detailinformationen zu verstecken und diese dann gezielt abrufen zu können. Ein interaktives System sollte nur jene Informationen bieten, die zum Erlernen von Kommunikationsprotokollen notwendig sind. Unnötige Informationen können sich aufgrund der Leistungsgrenzen des menschlichen Nervensystems belastend und störend auf

den Lernerfolg auswirken [Sta94].

Abschließend ist zu erwähnen, daß der Baukasten alle Visualisierungsformen unterstützen und die Entscheidung, welche eingesetzt werden, dem Entwickler des Protokollbeispiels überlassen sollte. Abbildung 2.6 zeigt z.B. eine Benutzungsschnittstelle inklusive der für ein Routing-Protokoll-Beispiel gewählten Visualisierungen.

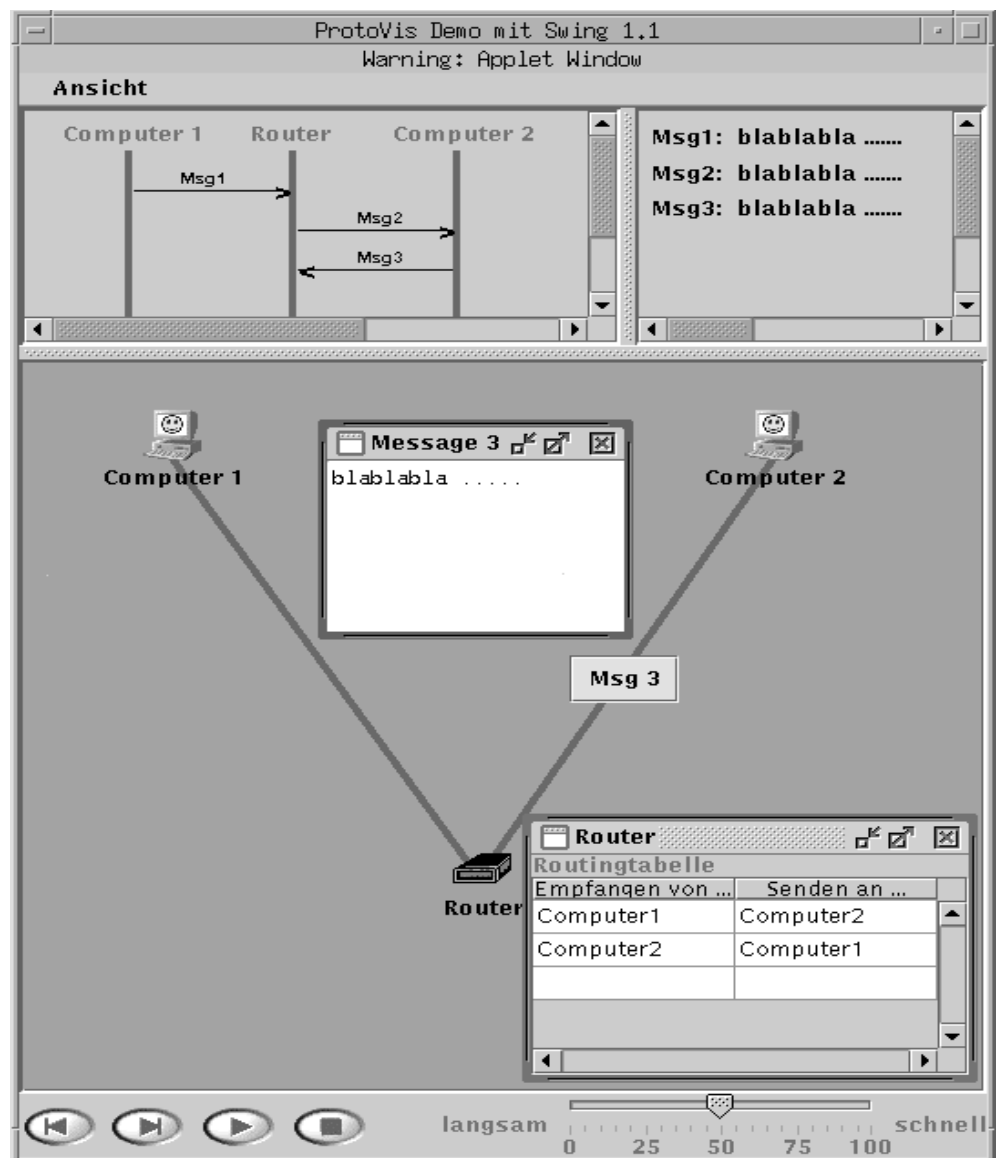


Abbildung 2.6: Benutzungsschnittstelle mit Swing-Komponenten

2.1.2 Interaktionsmöglichkeit

Systeme, die interaktiv gesteuert werden, sind heutzutage selbstverständlich. Jedoch muß vor der Programmierung eines solchen Systems abgeklärt werden, welche Interaktionsmöglichkeiten dem Benutzer zur Verfügung gestellt werden sollen.

Vor allem ist es für die Lehre besonders wichtig, dem Benutzer nur so viele Manipulationsmöglichkeiten wie notwendig zur Verfügung zu stellen, um einen besseren Lernerfolg zu erzielen. Dem Benutzer sollte auf keinen Fall gelingen, durch bestimmte Interaktionen das System zum Absturz zu bringen oder einen unkontrollierten Kommunikationsprotokollablauf hervorzurufen, der nicht mehr der Realität entspricht.

Eine Einteilung von Interaktionsmöglichkeiten wird im Paper *Interactive Protocol Simulation Applet for Distance Education* [BRM98] beschrieben. Dieses Paper setzt jedoch ein Simulationsmodell zur Visualisierung von Kommunikationsprotokollen voraus. Weil jedoch der vorhandene Baukasten zur Zeit keine Simulationsmodelle verwendet, wird in dieser Studienarbeit unabhängig vom Einsatz eines Simulationsmodells eines Baukastens die Interaktionsformen in zwei Bereiche eingeteilt. Die eine ist die *allgemeine Steuerung* ohne jegliche Manipulation des internen Kommunikationsprotokollablaufs. Die andere ist die *Manipulation des Protokollablaufs*, bei der der Benutzer Aktionen anstoßen bzw. entfernen und das eigentliche Protokoll abändern kann.

2.1.2.1 Allgemeine Steuerung

Unter *allgemeiner Steuerung* wird zum Beispiel die Geschwindigkeitssteuerung verstanden. Der Benutzer sollte die Möglichkeit erhalten, die Geschwindigkeit des Protokollablaufs von der Realität abweichen zu lassen. Zum Beispiel könnte die echte Ablaufgeschwindigkeit zum Erlernen des Protokolls zu schnell oder zu langsam sein. Es sollte auch möglich sein, den Protokollablauf jederzeit anzuhalten und wieder fortzusetzen. Ebenso wün-

schenswert ist eine Undo und Redo Funktionalität.

Diese Interaktionsmöglichkeiten sollten unabhängig vom Protokollbeispiel und zudem stets sichtbar als extra Steuerleiste zur Verfügung gestellt werden, da diese Steuerungsarten für die Lehre bei jedem Protokollbeispiel sinnvoll sind. Generell kann der Benutzer mit diesen Interaktionsmöglichkeiten auch keine unerwünschten Effekte hervorrufen.

2.1.2.2 Manipulation des Protokollablaufs

Eine weitaus schwierigere Interaktionsform als die *allgemeine Steuerung* ist die *Manipulation des Protokollablaufs*. Bei größeren Protokollbeispielen kann die Anzahl an möglichen Protokollabläufen auf ein Vielfaches anwachsen. Dennoch ist es unumgänglich, dem Benutzer für die Lehre diese Interaktionsform zur Verfügung zu stellen. Vor allem muß in Anbetracht der zweiten Zielgruppe, nämlich der Wissenschaftler, diese Interaktionsform ein elementarer Bestandteil sein, um Protokolle auf verschiedene Protokollabläufe testen und eventuell das Protokoll abändern zu können.

Es gibt für die Lehre zwei Möglichkeiten diese Interaktionsform zu realisieren, ohne daß der Benutzer eventuell unerwünschte Effekte hervorrufen kann. Entweder werden die Protokollabläufe im Voraus festgelegt und dem Benutzer nur an gewissen Ablaufzweigen die Wahl gegeben, welchen Protokollablauf er verfolgen möchte, oder der Ablauf eines Kommunikationsprotokolls wird wirklich nachsimuliert und der Benutzer bekommt über definierte Schnittstellen die Möglichkeit, jederzeit in die Simulation einzugreifen.

Die Variante der festgelegten Protokollabläufe ist zwar die am einfachsten zu realisierende und in Anbetracht von unerwünschten Effekten die robusteste Variante, jedoch schränkt sie die Flexibilität des Protokollbeispiels erheblich ein. Durch die fest im Protokollbeispiel eingebundenen und im Voraus berechneten Abläufe ist es dem Benutzer nicht mehr möglich, das eigentliche Protokoll abzuändern. Um Änderungen am Protokoll ermögli-

chen zu können, wird eine Logik-Komponente benötigt, die mittels formaler Beschreibung Protokolle simuliert.

Deshalb ist die Simulation von Kommunikationsprotokollen für die Lehre und auch in Anbetracht der Wissenschaft die sinnvollere Variante. Eine Simulation bildet die Realität ab und bietet deshalb eine hohe Flexibilität in Bezug auf Einflüsse der Umwelt. Es ist jedoch schwierig, bei einer durch den Benutzer manipulierbaren Simulation unerwünschte Effekte auszuschließen. Deshalb sollte der Baukasten eine allgemein definierte Schnittstelle zur Manipulation des Simulationsmodells anbieten, jedoch muß der Entwickler des Protokollbeispiels dann gesondert bestimmen, welche von der Schnittstelle angebotenen Manipulationsmöglichkeiten dem Benutzer zur Verfügung gestellt werden.

2.1.3 Wiederverwendbarkeit

Um eine hohe Wiederverwendbarkeit zu erlangen, muß bekanntlich der Entwurf offen und flexibel gestaltet werden. Aber wie groß muß die Flexibilität sein? Eine unendliche Flexibilität ist so gut wie unmöglich. Irgendwann muß eine Obergrenze der Flexibilität bestimmt werden, oder die Entwicklung läuft in ein unendliches Projekt.

Hier ist also ein angemessenes Maß an Flexibilität, sprich die goldene Mitte, anzustreben. Um sich dieser goldenenen Mitte anzunähern, hilft folgendes Zitat:

"Der Schlüssel zur Maximierung von Wiederverwendung liegt im Voraussehen von neuen Anforderungen und von Änderungen existierender Anforderungen sowie im Entwurf Ihrer Systeme derart, daß sie sich entsprechend entwickeln können" [GHJV96].

Um den überarbeiteten Baukasten so zu gestalten, daß er auf neue Anforderungen anpaßbar ist, werden als Hilfsmittel *Entwurfsmuster* [GHJV96] verwendet. Darauf, welche *Entwurfsmuster* für den neuen Baukasten eingesetzt

wurden, wird im folgendem Kapitel *Überarbeitung des Baukastens* im Abschnitt *Entwurf* eingegangen.

Zuletzt muß noch eine Überlegung über das Konzept des Baukastens angestellt werden. Der Baukasten kann in zwei Varianten realisiert werden, als **Klassenbibliothek** oder **Framework**.

Eine **Klassenbibliothek** besteht aus einer Menge von verwandten und wiederverwendbaren Klassen, die entworfen wurden, um nützliche und allgemeine Funktionalität zur Verfügung zu stellen. Der Entwurf von Klassenbibliotheken ist schwerer als der von Anwendungen, da sie, um nützlich zu sein, in mehreren Anwendungen verwendbar sein müssen. Weiterhin ist der Entwickler einer Klassenbibliothek nicht in der Lage zu wissen, wie diese Anwendungen oder ihre speziellen Bedürfnisse aussehen werden. Dies macht es um so wichtiger, Annahmen und Abhängigkeiten zu vermeiden, welche die Flexibilität und somit die Anwendbarkeit und Effektivität der Klassenbibliothek einschränken. Zur Erstellung eines Protokollbeispiels mit Hilfe einer Klassenbibliothek sind gute Programmierkenntnisse erforderlich, da der Hauptteil des Protokollbeispiels immer noch selbst programmiert und die Architektur bestimmt werden muß.

Ein **Framework** besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für eine bestimmte Klasse von Software darstellen. Somit bestimmt das Framework, im Gegensatz zu einer Klassenbibliothek, die Architektur der Software. Es definiert die Programmstruktur im großen, seine Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie den Kontrollfluß. Der Frameworkentwurf ist auch der komplizierteste Softwareentwurf von allen. Als Ergebnis können jedoch nicht nur Protokollbeispiele schneller entwickelt werden, sondern alle Protokollbeispiele haben auch ähnliche Programmstrukturen.

Generell wäre also ein Framework-Konzept anzustreben. Es bedarf jedoch großer Erfahrung über Gemeinsamkeiten von Protokollbeispielen, um die gemeinsame Struktur für ein Framework entwickeln zu können. Zusätzlich

benötigt ein Framework-Konzept eine lange Entwicklungszeit. Da im HiSAP Projekt momentan nur wenige lauffähige Baukästen entwickelt wurden und diese Baukästen kaum im Einsatz waren, ist es momentan nicht sinnvoll, ein Framework zu entwickeln.

Weil jederzeit möglich ist, auf der Basis einer Klassenbibliothek ein Framework zu entwickeln, wird im HiSAP Projekt vorerst für den Baukasten auf das Klassenbibliothek-Konzept gesetzt, mit dem Hintergedanken, im Laufe des Projekts die Klassenbibliothek zu einem Framework erweitern zu können.

2.2 Bisherige Entwicklung

Im Rahmen des HiSAP Projektes wurden schon viele Diplom- bzw. Studienarbeiten durchgeführt. Dieser Abschnitt gibt einen kurzen Überblick über die bereits vorhandenen Arbeiten. Ein Teil der Arbeiten befaßt sich mit der Entwicklung eines *Visualisierungsbaukastens*, der andere Teil mit der Entwicklung von *Werkzeugen zur Generierung von Simulationsmodellen*.

2.2.1 Visualisierungsbaukasten

Der Visualisierungsbaukasten ist ein Teil des HiSAP Projektes. Dieser Baukasten sollte unabhängig von den restlichen Komponenten des Projektes entwickelt werden. Unter dem Namen *ProtoVis* wurden vier Studienarbeiten durchgeführt, die im folgendem beschrieben werden.

2.2.1.1 Grundbaukasten von Schurr

Der ursprünglich im Rahmen der von Peter W. Schurr durchgeführten Studienarbeit [Sch97] erstellte Baukasten, wurde für Applets in der Java Version 1.0 geschrieben und visualisiert Kommunikationsprotokolle mit Hilfe von Time-Sequence-Diagrams. Er unterstützt nur festgelegte Protokol-

labläufe und bietet als *Interaktionsmöglichkeit* die im vorigem Abschnitt beschriebene *allgemeine Steuerung* (Geschwindigkeit, Pause–Weiter und "Aktion zurück"–"Aktion vorwärts"). Drückt der Benutzer den "Aktion zurück"–Button, wird die Aktion nicht wirklich rückgängig gemacht, sondern nur Teile der Grafik übermalt. Somit kann der Benutzer zwar den Protokollablauf in Einzelschritten betrachten, aber nicht, nach ein paar rückgängig gemachten Schritten, einen anderen Protokollablauf wählen. Eine Wahl ist erst am Ende des angehaltenen Protokollablaufs wieder möglich.

Der Grundbaukasten besteht aus den Klassen *Prozeß*, *Nachricht*, *Assistent* und *Display*. Den Hauptteil des Baukastens macht die Klasse *Assistent* aus. Diese Klasse bietet die Schnittstelle zwischen Baukasten und dem Entwickler von Protokollbeispielen und bestimmt deshalb auch die Leistungsfähigkeit des Baukastens. Mittels Methodenaufrufen ist ein einfaches Erstellen von Protokollbeispielen über den *Assistenten* möglich. Durch das monolytische Konzept des *Assistenten* wird jedoch die Auswahl verschiedener Visualisierungsmöglichkeiten von Kommunikationsprotokollen eingeschränkt. Die Erweiterbarkeit des Baukastens wird dadurch ebenfalls erschwert.

Mit Hilfe dieses Baukastens wurden mittlerweile einige Protokollbeispiele erzeugt. Darunter sind auch vier Studienarbeiten ([Baa97], [Den97], [Gez97] und [Kai97]) zur Visualisierung von Protokollen aus dem Bereich *Sicherheit*. Abbildung 2.7 zeigt eines der vielen mit dem Baukasten von Schurr erzeugten und visualisierten Protokollbeispielen.

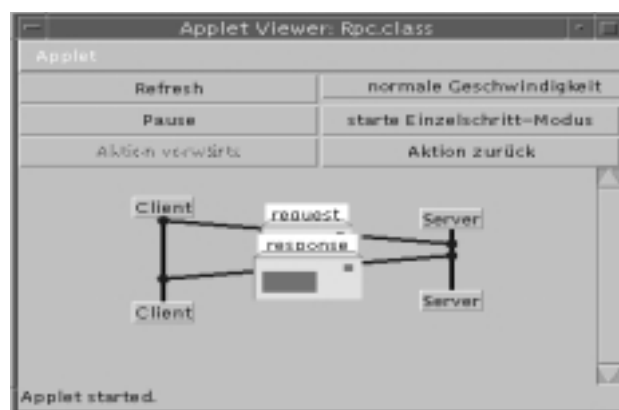


Abbildung 2.7: RPC–Beispiel mit dem Baukasten von Schurr

2.2.1.2 Erweiterung von Gorth und Papoulidis

Im Rahmen von zwei Studienarbeiten sollte der Baukasten von Schurr zur Visualisierung von SDL-Konstrukten erweitert werden. Ziel der Studienarbeit von Ralph Gorth [Gor98] war die Entwicklung einer Klassenbibliothek zur Visualisierung von Strukturen, die mittels der Protokollspezifikationsprache SDL beschrieben werden. Die Studienarbeit von Konstantinos Papoulidis [Pap98] hatte zum Ziel, eine dem Baukasten zugehörige Benutzungsschnittstelle zu realisieren und eine Undo-Funktionalität zu integrieren. Das Ergebnis dieser zwei Arbeiten war ein komplett neuer Baukasten, der Kommunikationsprotokolle hierarchisch mit Graphen darstellen kann. Eine spätere, nicht in diesen zwei Arbeiten vorgesehene, Implementierung zur Darstellung von FSM (Finite State Machines) wurde ebenfalls berücksichtigt. Die Benutzungsschnittstelle (Abbildung 2.8) wurde mit dem AWT² von Java realisiert und bietet ebenfalls wie der Grundbaukasten von Schurr die *allgemeine Steuerung* an.



Abbildung 2.8: Benutzungsschnittstelle von Papoulidis

Leider ist der Baukasten nicht komplett implementiert, so daß kein Protokollbeispiel erzeugt werden konnte. Deshalb ist eine genauere Beschreibung des Baukastens nicht möglich.

2.2.1.3 Ergänzungen von Rau

Die nachfolgende von Jürgen Rau [Rau98] durchgeführte Studienarbeit erweiterte den Grundbaukasten von Schurr zur Darstellung von Kommunikationsprotokollen als Finite State Machine (FSM, eine spezielle Graphendarstellung). Der Baukasten von Gorth und Papoulidis wurde wegen der unvollständigen Implementation nicht berücksichtigt.

Der Grundbaukasten wurde auf die Java Version 1.1 portiert und um die Möglichkeit erweitert, Protokollbeispiele zu erstellen, die als Applikation

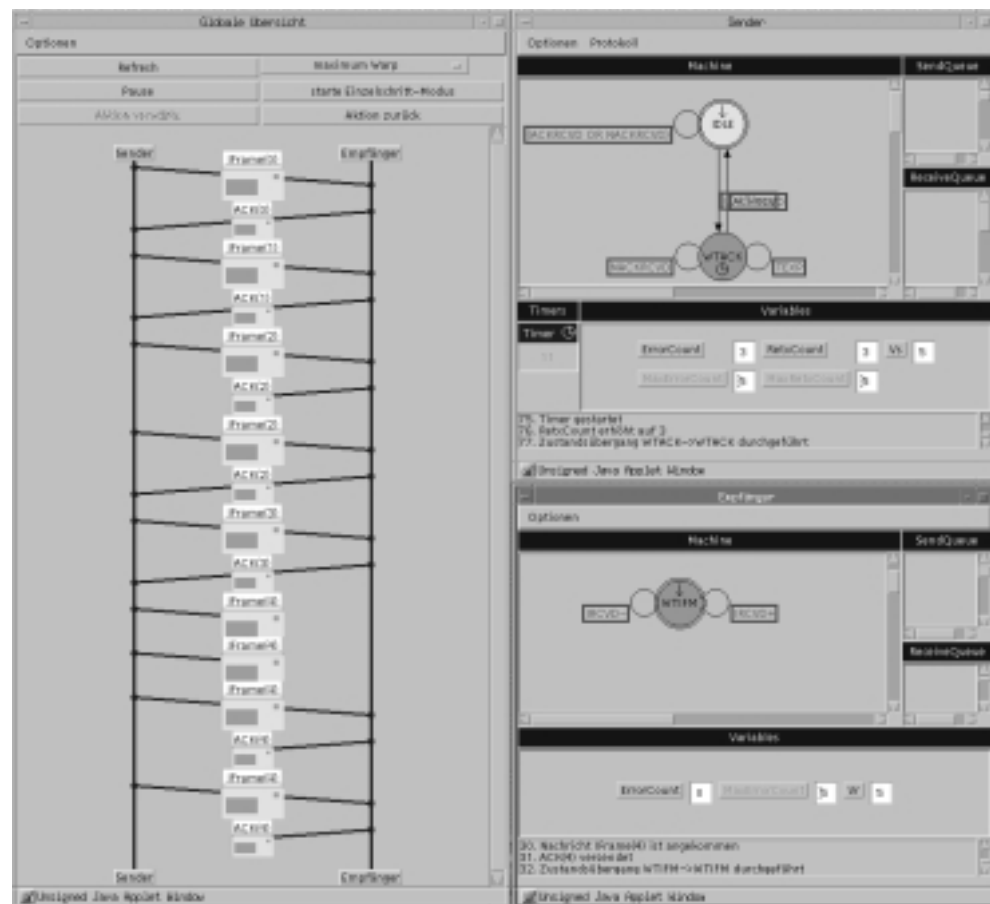


Abbildung 2.9: IdleRQ-Beispiel mit dem Baukasten von Rau

ablaufen. Zusätzlich zum Grundbaukasten entstand ein davon unabhängiger Teilbaukasten zur Visualisierung von Kommunikationsprotokollen mittels FSM. Um den Teilbaukasten vom Grundbaukasten unterscheiden zu können, wurde der Grundbaukasten als TSD-Baukasten (Time-Sequence-Diagram) und der Teilbaukasten als FSM-Baukasten benannt. Zusammen ergeben sie den *erweiterten Baukasten*.

Dieser Baukasten ist der erste implementierte Baukasten, der zwei Darstellungsformen gleichzeitig unterstützt. Mit ihm ist es nun möglich, nicht nur die Nachrichtenübermittlung zu visualisieren, sondern auch den internen Ablauf einzelner Prozesse darzustellen. Abbildung 2.9 zeigt ein mittels Time-Sequence-Diagram und FSM dargestelltes Beispiel eines "Stop-and-Wait" Protokolls.

2.2.2 Generator für Simulationsmodelle

Parallel zu den Studienarbeiten von Gorth und Papoulidis entwickelte Edgar Kogel [Kog97] in seiner Diplomarbeit einen Generator. Mit Hilfe dieses Generators ist es möglich, aus einer textuellen SDL-Spezifikation ein internes Simulationsmodell aufzubauen. Die Spezifikationsprache SDL wurde deshalb gewählt, weil sie sich zur Beschreibung von Kommunikationsprotokollen gegenüber anderen Spezifikationsprachen am besten eignet [Kai98].

Das vom Generator erzeugte interne Simulationsmodell repräsentiert allerdings lediglich die statische Struktur der Protokollumgebung. Ein lauffähiges Simulationsmodell konnte damit nicht generiert werden. Wegen fehlender Schnittstellen zu den Visualisierungsbaukästen war eine Visualisierung der internen Simulationsmodelle nicht möglich.

Die Integration von dem Generator und den Visualisierungsbaukästen zusammen mit der Entwicklung eines ablauffähigen Simulationsmodells war die Aufgabe der Diplomarbeit von Markus Mayer [May98]. Leider konnte die Integration nur theoretisch durchgeführt werden, da bis zum Ende der

Arbeit einige Teile des Baukastens nicht lauffähig waren.

In der Diplomarbeit von Oliver Baar [Baa98] wurde die Petri Netz Spezifikation um eine spezielle Spezifikation zur Beschreibung von Kommunikationsprotokollen erweitert. Zusätzlich wurde der Generator so ergänzt, daß dieser die Spezifikationssprachen SDL und die erweiterte Petri Netz Sprache interpretieren kann. Hinzu kam ein Editor für die Eingabe dieser Spezifikationssprachen.

2.3 Fazit

Durch die vielen Ergänzungen erhielt der Baukasten eine große Anzahl an Funktionalitäten. Momentan können jedoch noch keine lauffähigen Modelle zur Simulation von Kommunikationsprotokollen erzeugt werden.

Im Laufe des HiSAP Projekts kristallisierte sich heraus, daß ein solches Simulationsmodell viele Vorteile bietet. Wie in Abschnitt *Interaktionsmöglichkeit* gezeigt, kann es dadurch z.B. dem Benutzer ermöglicht werden, interaktiv Änderungen am Protokollbeispiel vorzunehmen. Soll der Baukasten später für wissenschaftliche Zwecke eingesetzt werden, wird die Erzeugung von Simulationsmodellen sogar unumgänglich.

Der momentane Visualisierungsbaukasten visualisiert festgelegte Protokollabläufe und ist für Simulationsmodelle nicht ausgelegt. Falls Simulationsmodelle im neuen Baukasten eingesetzt werden, ist eine grundlegende Überarbeitung des Visualisierungsbaukastens notwendig.

3 Überarbeitung des Baukastens

Dieses Kapitel beschreibt die grundlegende Überarbeitung des existierenden Baukastens. Zunächst wird analysiert, welche Anforderungen an den neuen Baukasten gestellt werden. Danach folgt der Entwurf des neuen überarbeiteten Baukastens. Zum Schluß werden Teile der Implementation beschrieben.

3.1 Analyse

In diesem Abschnitt wird analysiert, welche zusätzlichen Eigenschaften der neue Baukasten haben soll. Danach werden, zur Maximierung von Wiederverwendung, mögliche zukünftige Anforderungen beschrieben, damit schon beim Entwurf des Baukastens diese Anforderungen als mögliche zukünftige Erweiterungen berücksichtigt werden können.

3.1.1 Anforderungen an den neuen Baukasten

Zusammenfassend werden die an den neuen Baukasten gestellten Anforderungen aufgeführt, die bisher nicht ausreichend unterstützt wurden. Die gesamten Anforderungen sind in Kapitel 2, Abschnitt *Ziele* beschrieben.

Der überarbeitete bzw. neue Baukasten soll um die Möglichkeit ergänzt werden, alle beschriebenen *Visualisierungsmöglichkeiten* darzustellen. Aufgrund vieler, sehr verschiedener Kommunikationsprotokollbeispiele und der fehlenden Kenntnis davon, wie Kommunikationsprotokollbeispiele einheitlich geeignet visualisiert werden können, wird momentan keine automatische Visualisierung gefordert. Die Entscheidung, wie die einzelnen Kommunikationsprotokollbeispiele visualisiert werden, soll der Entwickler des Protokollbeispiels treffen. Somit kann der Entwickler Bereiche des Protokollbeispiels, die er ihm wichtig sind, nach eigenem Ermessen hervorheben bzw. visualisieren.

Bis auf die im vorigem Kapitel, Abschnitt *Interaktionsmöglichkeiten*, beschriebene *allgemeine Steuerung* ist mit der Entscheidung, welche Interaktionsmöglichkeiten dem Benutzer zur Verfügung gestellt werden sollen, ebenso zu verfahren. D.h. die Entscheidung wird wieder dem Entwickler des Protokollbeispiels überlassen. Jedoch sollen generell die Interaktionsformen der *allgemeinen Steuerung* vom Baukasten zur Verfügung gestellt werden. Die Undo-Redo Funktionalität soll, abweichend von der bisher verfolgten Strategie, auf der Basis von Simulationsmodellen realisiert werden.

Abgesehen von der Visualisierung soll der neue Baukasten die Erzeugung von lauffähigen Simulationsmodellen unterstützen. Dabei soll es möglich sein, Kommunikationsprotokolle in verschiedenen Detaillierungsstufen zu simulieren. Analog sollen die Visualisierungskomponenten den Zustand dieser Simulationsmodelle je nach Bedarf in abstrakter Form oder im Detail darstellen können.

Angestrebt ist die Simulation aller Protokolle, die für vernetzte Rechner vorgesehen sind und die in die ISO-OSI Schichten 2-7 eingeordnet werden können. D.h., bis auf die Bitübertragungsschicht (Schicht 1) sollen idealerweise für alle Kommunikationsprotokolle, zumindest aber für die, die in den Vorlesungen der Abteilung *Verteilte Systeme* am Institut für parallele und verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart behandelt werden, Simulationsmodelle durch den Baukasten erzeugt werden können.

3.1.2 Mögliche zukünftige Anforderungen

Im Sinne der Wiederverwendung werden mögliche zukünftige Anforderungen betrachtet. So wird schon beim Entwurf die Möglichkeit geschaffen, den Baukasten in Zukunft so auszubauen, daß er diesen Anforderungen gerecht wird.

Geplant ist schon der eventuelle Einsatz des Baukastens für wissenschaftliche Zwecke. Dabei muß es möglich sein, gewünschte Bereiche von Protokollen nach der Realität zu modellieren.

Unter Umständen wird ein Einsatz des Baukastens zur Simulation von kabelloser Kommunikation (Funknetze) erwünscht. Dies erfordert eventuell zusätzliche Eigenschaften der Simulationsmodelle. Wenn der Baukasten diese Eigenschaften nicht unterstützt, muß er auf diese Eigenschaften erweitert werden können, um eine spätere Unterstützung von Übertragungen mittels Funkwellen zu bieten.

Leider können nicht alle zukünftigen Anforderungen vorhergesehen werden. Deshalb werden an manchen Stellen des Entwurfs generelle Erweiterungsmöglichkeiten berücksichtigt. Im folgendem Abschnitt *Entwurf* werden diese Stellen beschrieben.

3.2 Entwurf

In diesem Abschnitt wird der Entwurf des neuen Baukastens beschrieben. Hierzu wird teilweise die formale Notation OMT³ verwendet. Bevor Erläuterungen über die Gestaltung der einzelnen Teile folgen, werden einige grundsätzliche Entwurfsentscheidungen getroffen.

3.2.1 Grundsätzliche Entwurfsentscheidungen

Es wird nun über den Umfang und die Art der Überarbeitung des Baukastens in dieser Studienarbeit entschieden. Dabei wird differenziert, welche Anforderungen berücksichtigt werden können und bei welchen auf spätere Arbeiten verwiesen werden muß.

Wegen der zukünftigen Notwendigkeit von Komponenten zur Erzeugung von lauffähigen Simulationsmodellen wird ein Entwurf von *Klassen zur Erzeugung von lauffähigen Simulationsmodellen* ein wesentlicher Bestandteil dieser Überarbeitung sein. Ein Entwurf für die in dieser Studienarbeit beschriebene *allgemeine Steuerung* von Simulationsmodellen wie z.B. Geschwindigkeitssteuerung, Undo-Redo usw., wird nicht durchgeführt.

3 Object Modelling Technique

Es wäre nun naheliegend, den vorhandenen Generator auf die entworfenen *Klassen zur Erzeugung von Simulationsmodellen* abzustimmen. Da sich der Focus dieser Studienarbeit auf die Überarbeitung des Baukastens für Lehrzwecke richtet, wird zuerst die Integration der vorhandenen Visualisierungskomponenten mit dem neuen Baukasten untersucht. Eine mit dieser Integration zusätzlich durchgeführte Anpassung des Generators, hätte den Rahmen dieser Studienarbeit gesprengt. Daher wird darauf verzichtet, so daß dies eine gute Grundlage für nachfolgende Arbeiten bietet.

Die Integration der vorhandenen Visualisierungskomponenten ist sehr schwer zu realisieren, da der momentane Visualisierungsbaukasten statt Simulationsmodellen festgelegte Protokollabläufe visualisiert.

Zudem erfuhr die Programmierumgebung Java eine so drastische Änderung, daß der Aufwand, eine Anpassung der Visualisierungskomponenten an die neue Programmierumgebung durchzuführen, enorm anstieg. Seit der Einführung von Visualisierungskomponenten namens Swing (ein Teil der Java Foundation Classes) in die Programmierumgebung Java ersetzen diese einige Visualisierungskomponenten des HiSAP-Baukastens. Somit bedarf es nur noch wenige zusätzlich programmierte Visualisierungskomponenten zur Visualisierung von Kommunikationsprotokollen. Deshalb wurde entschieden, einen Neuentwurf des Visualisierungsbaukastens auf der Grundlage von Swing-Komponenten durchzuführen.

Grundkonzepte und Strukturen des alten Visualisierungsbaukastens wurden übernommen. Eine Übernahme der existierenden und mit dem AWT⁴ von Java programmierten Visualisierungskomponenten war nicht möglich, da diese mit den neuen Swing-Komponenten schlecht zusammenarbeiten. Deshalb wurden die Visualisierungskomponenten neu programmiert.

Um den neuen Baukasten so flexibel zu gestalten, daß er zusätzliche Erweiterungen unterstützt, wie sie im Abschnitt *Mögliche zukünftige Anforderungen* beschrieben wurden, werden folgende *Entwurfsmuster* [GHJV96] beim Entwurf verwendet: *Beobachtermuster*, *Kompositionsmuster*, *Strategiemu-*

4 Abstract Window Toolkit

ster, *Prototypenmuster*, *Dekorierermuster*, *Fabrikmethodenmuster* und *Brückenmuster* (vgl. Anhang Kurzbeschreibung der Entwurfsmuster). Diese tragen zur Maximierung der Wiederverwendung des Baukastens bei. Der Grund der Verwendung dieser Muster wird in den Abschnitten erklärt, in denen sie eingesetzt werden.

Um die beiden wesentlichen Teile des neuen Baukastens austauschbar zu machen, d.h. damit das Simulationsmodell und die Visualisierungskomponenten jederzeit ausgetauscht werden können, werden sie getrennt entworfen und für ihre Zusammenarbeit bestimmte *Schnittstellen* festgelegt. Es werden also zuerst *Klassen zur Erzeugung von Simulationsmodellen*, danach *Klassen zur Visualisierung* und zum Schluß die zugehörigen *Schnittstellen* entworfen. Die Trennung zwischen Simulationsmodell und den Visualisierungskomponenten wird mittels MVC-Paradigma realisiert. Das MVC-Paradigma wird durch drei lose gekoppelte Objekte umgesetzt, Model, View und Controller Objekt. Dieses Paradigma verwendet das *Beobachter-*, *Kompositions-*, *Strategie-*, *Fabrikmethoden-* und *Dekorierermuster*. Im Detail existieren unterschiedliche Ausführungsvarianten des MVC-Paradigmas. In dieser Studienarbeit wird die in den Swing-Komponenten eingesetzte Variante verwendet. Bei dieser Variante sind die View- und Controller-Objekte etwas enger gekoppelt und werden von den Visualisierungskomponenten zur Verfügung gestellt. D.h. daß die entworfenen *Klassen zur Erzeugung von Simulationsmodellen* das Model- und die zugehörigen *Klassen zur Visualisierung* das View- und Controller-Objekt beschreiben. Eine ausführliche Dokumentation dieses MVC-Paradigmas ist auf der Java-Homepage von Sun zu finden [Sun99].

3.2.2 Klassen zur Erzeugung von Simulationsmodellen

Ziel ist, mit der richtigen Wahl der *Klassen zur Erzeugung von Simulationsmodellen* die Anzahl dieser Klassen auf ein Minimum zu beschränken (Core Foundation Classes). Dies fördert die Flexibilität auf der feinkörnigsten Ebene, indem diese Klassen in Kombination untereinander alle Komponenten beschreiben, die zur Erzeugung von Simulationsmodellen für Kommunikationsprotokolle benötigt werden.

Wie sich im HiSAP-Projekt herausstellte, sind zuerst drei Klassen zur Erzeugung von Simulationsmodellen erforderlich. Diese sind Knoten, Verbindung und Nachricht (engl. *Node*, *Connection* und *Message*). Folgend werden in Übereinstimmung mit der Implementierung nur noch die englischen Ausdrücke verwendet.

Die Klasse *Connection* ist zusätzlich in zwei Kommunikationsarten unterteilt, den Simplex- und den HalfDuplex-Betrieb. Um eine Erzeugung der Klasse *Connection* zu verhindern, ist diese abstrakt definiert. Zum Überblick zeigt Abbildung 3.1 die momentan entworfenen *Grundkomponenten zur Erzeugung von Simulationsmodellen*. Diese Grundkomponenten werden nun folgend genauer beschrieben.

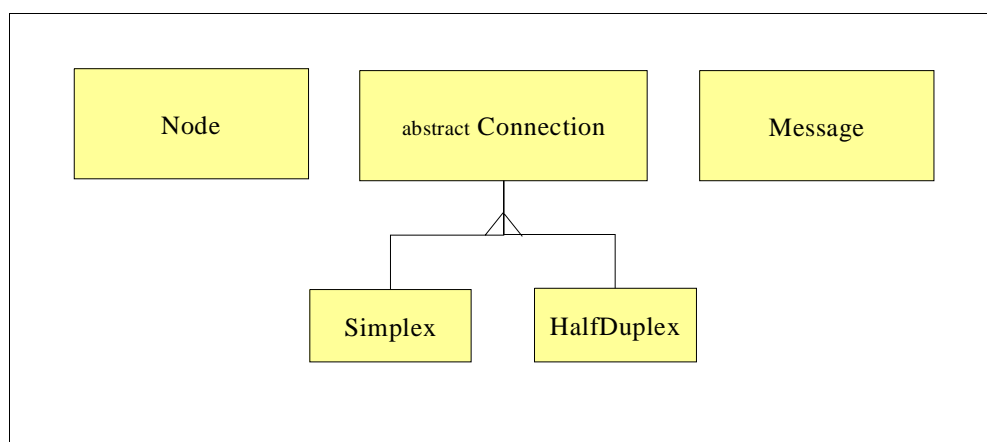


Abbildung 3.1: Komponenten zur Erzeugung von Simulationsmodellen

Node

Ein *Node* beschreibt einen Prozeß, der *Messages* versenden und empfangen kann. Wann *Messages* versendet und welche Aktionen nach Empfang einer *Message* ausgelöst werden sollen, wird durch Kommunikationsprotokolle bestimmt. Somit muß jedem neu erzeugten *Node* ein Kommunikationsprotokoll zugewiesen werden. Bevor ein *Node* mit einem anderen *Node* kommunizieren kann, müssen beide durch eine *Connection* verbunden werden. Generell kann ein *Node* über beliebig viele *Connections* mit anderen verbunden werden.

Connection

Connections übertragen *Messages* und ermöglichen somit die Kommunikation zwischen *Nodes*. Eine *Connection* kann stets nur genau zwei *Nodes* verbinden. Eine *Connection* muß entweder als eine Simplex- oder HalfDuplex-*Connection* definiert sein. Eine Simplex-*Connection* bietet einen Nachrichtenaustausch in nur eine Richtung, eine HalfDuplex-*Connection* in beide Richtungen an. Es ist jedoch nicht möglich, mit einer HalfDuplex-*Connection* gleichzeitig in beide Richtungen zu kommunizieren. Die Kommunikation in beide Richtungen muß also synchronisiert werden, sonst kollidieren die *Messages* auf der *Connection*. Wie eine gleichzeitige Kommunikation in beide Richtungen realisiert werden kann, wird im folgendem Abschnitt *Anwendungsbeispiele der Grundkomponenten* gezeigt.

Message

Durch *Messages* wird die eigentliche Information zwischen *Nodes* untereinander ausgetauscht. Die Struktur einer *Message* muß vorher von den kommunizierenden *Nodes* bestimmt werden, damit empfangene *Messages* von jeder *Node* interpretiert werden können. *Messages* können nur über *Connections* übertragen werden.

3.2.2.1 Anwendungsbeispiele der Grundkomponenten

Nachdem die Grundkomponenten beschrieben wurden, werden nun ein paar Anwendungsbeispiele gezeigt. Diese Anwendungsbeispiele demonstrieren, wie die Grundkomponenten andere Komponenten beschreiben können.

Bei allen im Rahmen dieser Studienarbeit herangezogenen Protokollbeispielen konnte jeweils mindestens ein Simulationsmodell mittels der Grundkomponenten erzeugt werden. Die Protokollbeispiele, die in den Vorlesungen der Abteilung *Verteilte Systeme* am Institut für parallele und verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart behandelt werden, waren ebenfalls in den Überlegungen inbegriffen.

Obwohl manche Komponenten sehr komplex und umständlich durch die Grundkomponenten zu beschreiben sind, ist diese Art der Beschreibung sinnvoll. Wenn für jede einzelne Komponente eine eigene Klasse programmiert werden müßte, wäre zwar die Beschreibung der Komponente einfacher (indem die zugehörige Klasse erzeugt wird), aber die Flexibilität wäre dadurch eingeschränkt. Denn es existiert eine unbegrenzte Anzahl von Komponenten und es können daher nicht alle Komponenten beim Entwurf berücksichtigt werden. Somit ist es kaum machbar, für jede einzelne Komponente eine Klasse zu entwerfen. Dies ist der Grund, warum eine Art »LEGO Baustein«-Prinzip bevorzugt wird. Dadurch wird ermöglicht, viele vom Entwickler nicht vorhergesehene Komponenten zusammenzubauen.

Nun werden einige Komponenten aufgeführt und gezeigt, wie diese Komponenten durch die Grundkomponenten beschrieben werden können. Aus Platzgründen werden nur die wichtigsten in dieser Studienarbeit behandelten Komponenten aufgeführt.

FullDuplex-Connection

Wie schon beschrieben, unterstützt die Grundkomponente *Connection* nur zwei Kommunikationsarten, den Simplex- und den HalfDuplex-Betrieb.

Ein FullDuplex-Betrieb wurde nicht umgesetzt, weil diese Betriebsart einfach durch zwei Simplex-Connections nachgebildet werden kann. Es ist somit unnötig, eigens eine FullDuplex-Connection zu programmieren. Um diese zwei Simplex-Connections vor dem Benutzer zu verstecken und eine FullDuplex-Connection darzustellen, muß eine Visualisierungskomponente existieren, die zwei Simplex-Connections als FullDuplex-Connection visualisiert. Abbildung 3.2 zeigt zwei Nodes, die mittels zweier Simplex-Connections (dicker Pfeil in eine Richtung) eine FullDuplex-Connection nachbildet und somit auch als eine FullDuplex-Connection visualisiert werden kann (gestrichelter Bereich).

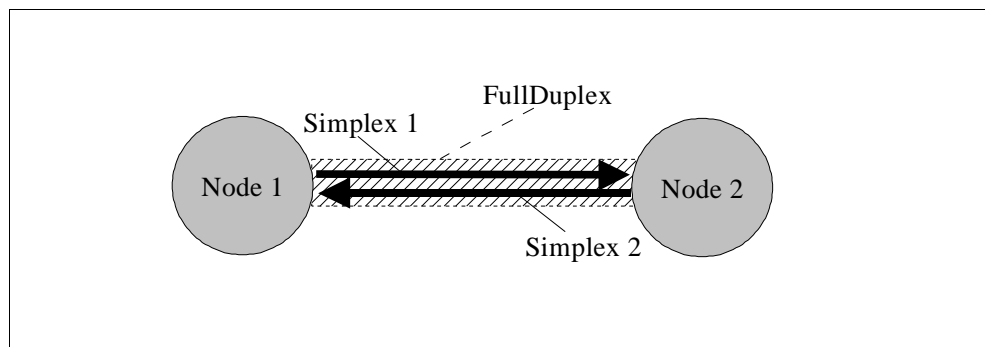


Abbildung 3.2: FullDuplex-Connection

Schichtenkonzept

Schichtenmodelle sind wichtige Modelle, um Kommunikationsprotokolle zu strukturieren. Die Grundkomponenten müssen deswegen Schichtenmodelle wie z. B. ISO-OSI- oder TCP/IP-Referenzmodelle beschreiben können. Dabei muß es möglich sein, jede einzelne Schicht zu simulieren. Dies bedeutet, daß es möglich sein muß, das Protokoll und die Service Access Points (SAPs) jeder Schicht durch die Grundkomponenten zu beschreiben. Deshalb wird versucht, eine allgemeine Schicht n durch die Grundkomponenten nachzubilden, um zu zeigen, daß alle Schichtenmodelle mit den Grundkomponenten beschrieben werden können. Abbildung 3.3 zeigt eine allgemeine Schicht n , die drei SAPs für die Schicht $n-1$ und drei SAPs für die Schicht $n+1$ besitzt. Diese Schicht wird durch sieben Nodes und mehreren HalfDuplex-Connections beschrieben, wobei die mittlere Node das Pro-

tokoll dieser Schicht ausführt und die restlichen sechs *Nodes* ein simples SAP-Protokoll, das einfach die *Messages* weiterleitet.

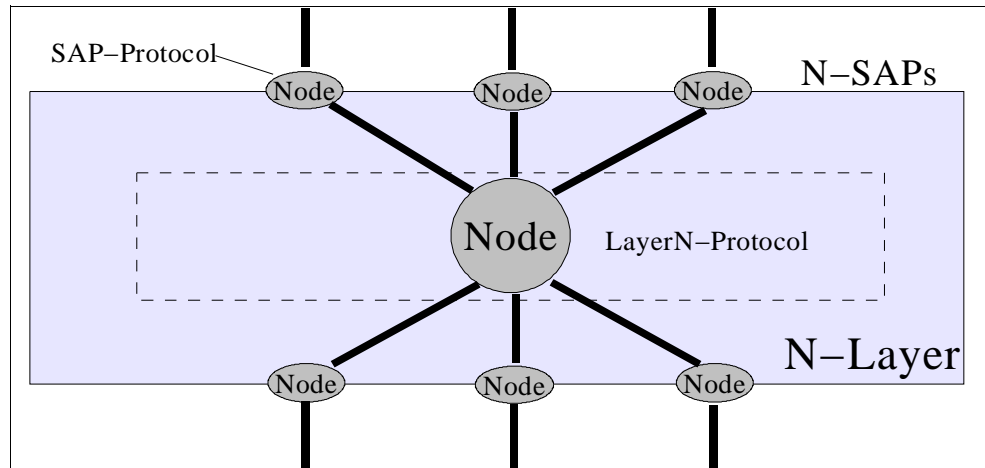


Abbildung 3.3: Schichtenkonzept

Verbindung mit mehr als zwei Endpunkten

Manchmal ist es erwünscht, eine *Connection* mit mehr als zwei *Nodes* zu verbinden. Z.B. werden mehrere Computer an ein Ethernet angeschlossen. Wenn in diesem Ethernet von einem Computer eine *Message* versendet wird, empfangen alle anderen Computer automatisch diese *Message*. Dieser Broadcast der *Message* muß von der *Connection* realisiert werden.

Solche Verbindungen müssen wegen der vorher beschriebenen Eigenschaft einer *Connection* nachgebildet und mittels abstrakter Visualisierung vor dem Benutzer und den angeschlossenen *Nodes* geeignet verborgen werden. Für eine Verbindung mit n Endpunkten, muß daher mindestens ein *Node* als Bindeglied eingeführt werden. Dieser *Node* führt ein einfaches Protokoll, wie z.B. ein Transceiver(Broadcast)-Protokoll, aus.

Abbildung 3.4 zeigt z. B. eine *Connection* mit drei Endpunkten, wobei diese mit einem *Node* in der Mitte und drei HalfDuplex-*Connections* beschrieben wird. Die *Connection* mit drei Endpunkten (gestrichelter Bereich) muß dementsprechend abstrakt von einer Visualisierungskomponente visualisiert werden.

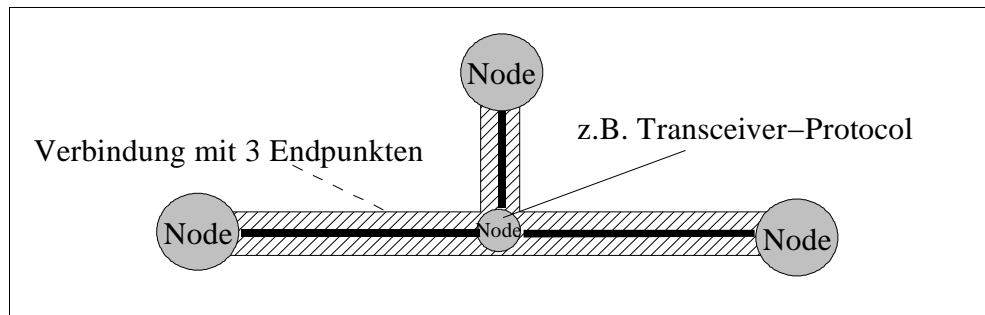


Abbildung 3.4: Verbindung mit mehr als zwei Endpunkten

Überholen von Nachrichten

Das Überholen von Nachrichten (*Messages*) ist, im Gegensatz zum alten Baukasten, in einer *Connection* nicht möglich. Dies muß nun, wie es auch in Wirklichkeit zustande kommt, entweder durch unterschiedlich lange Wege der *Messages* oder durch Vertauschen der Reihenfolge eingehender *Messages* im *Node* hervorgerufen werden.

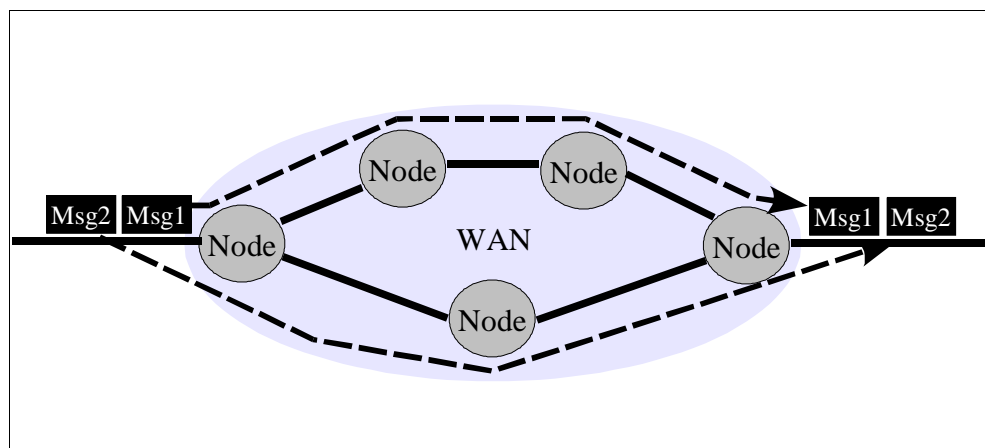


Abbildung 3.5: Überholen von Nachrichten (Variante 1)

Die unterschiedlich langen Wege werden durch Abzweigungen der Leitwege von *Messages* realisiert. Somit könnte z.B., wie in Abbildung 3.5 gezeigt, die *Msg1* über den oberen Weg geleitet werden und die *Msg2* über den unteren Weg. Da der obere Weg eine Zwischenstation mehr aufweist, wäre es möglich, daß *Messages*, die diesen Weg nehmen, eine längere Zeit

brauchen, um am rechten Ende anzukommen, als die *Messages*, die den unteren Weg nehmen.

Anstatt die Verzögerung über mehrere *Nodes* hervorzurufen wäre als Alternative eine unterschiedliche Verzögerung innerhalb der *Nodes* möglich oder eine unterschiedliche Ausbreitungsverzögerung bzw. Übertragungsgeschwindigkeit der *Connections*.

Das Vertauschen der Reihenfolge eingehender *Messages* im *Node* geschieht mittels Speichern der *Messages* im Puffer dieser *Node* und einer Verwaltung des Puffers, die die Reihenfolge der eingehenden *Messages* vertauscht. Zum Beispiel könnte ein LIFO (Last In First Out) Puffer, wie in Abbildung 3.6, gezeigt, so etwas realisieren.

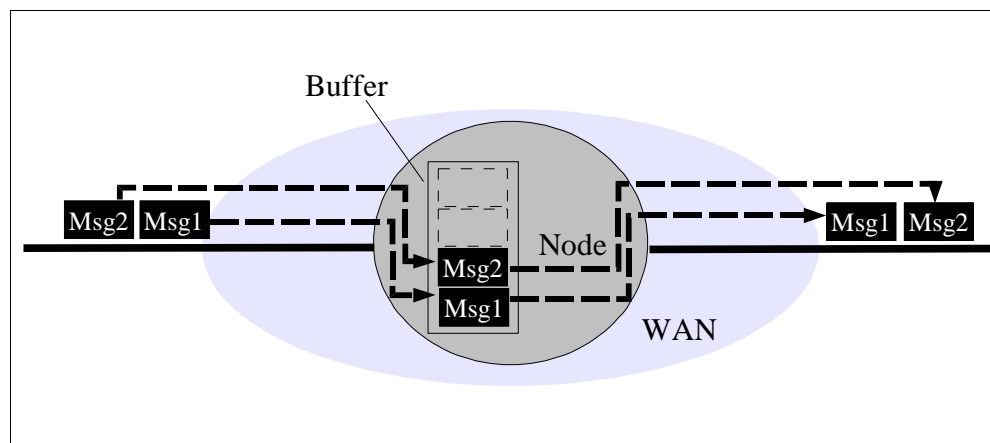


Abbildung 3.6: Überholen von Nachrichten (Variante 2)

Die Abstraktion beider Varianten könnte für den Benutzer z.B. durch eine Visualisierung als WAN (Wide Area Network) mittels eines gefärbten Kreises oder einer Wolke dargestellt werden. Somit bleibt das Detail verborgen und der Benutzer sieht nur, wie die *Messages* in den WAN Bereich verschwinden und in unterschiedlicher Reihenfolge aus dem WAN Bereich wieder auftauchen.

Nachrichtenduplikate

Ähnlich wie beim Überholen von Nachrichten wird die Komponente für die Erzeugung von Nachrichtenduplikaten beschrieben. Dabei muß mindestens ein *Node* die empfangene *Message* klonen und diese wieder versenden. Wenn beide *Messages* irgendwann einen anderen *Node* passieren, dann empfängt dieser *Node* eine identische *Message* zweimal.

Der Klonvorgang von *Messages* wird mittels *Prototypenmuster* realisiert. Wie *Messages* geklont werden, wird im Abschnitt *Implementation* erläutert.

Abbildung 3.7 zeigt beispielsweise ein vom linken *Node* erzeugtes Nachrichtenduplikat *Msg1'*.

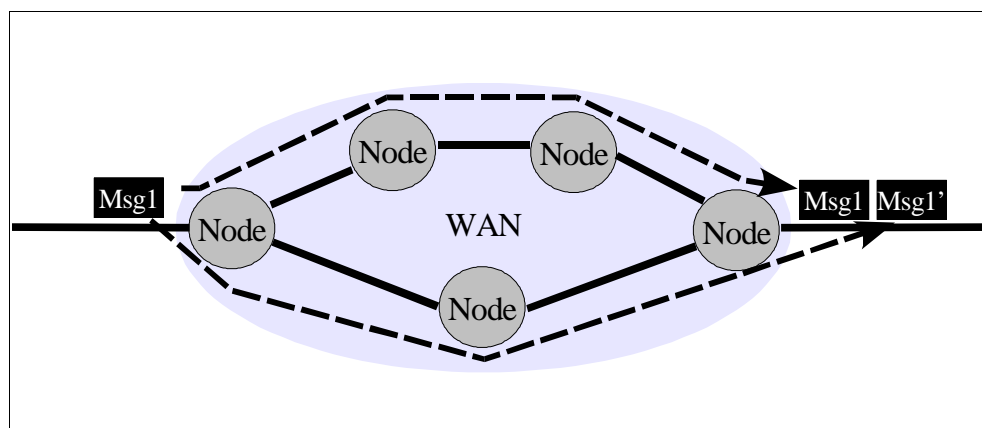


Abbildung 3.7: Nachrichtenduplikate

3.2.2.2 Zuweisung von Kommunikationsprotokollen an Nodes

Eine Möglichkeit wäre zum Beispiel die statische Zuweisung von Kommunikationsprotokollen an *Nodes*. Bei einer statischen Zuweisung werden schon beim Entwurf die Kommunikationsprotokolle an *Nodes* zugewiesen, indem konkrete Klassen die Kommunikationsprotokolle beschreiben und von der allgemeinen *Node*-Klasse erben. Dann aber kann dem *Node* kein anderes Kommunikationsprotokoll dynamisch zur Laufzeit zugewiesen werden. Dies ist jedoch für manche Zwecke erforderlich.

Deshalb wird die dynamische Zuweisung angewandt. Diese ermöglicht die Zuweisung während der Laufzeit und somit auch eine Änderung von Kommunikationsprotokollen einer *Node*. Dafür besteht die Gefahr, daß während der Laufzeit ein *Node* eine *Message* empfängt und diesem *Node* eventuell noch kein Kommunikationsprotokoll zugewiesen wurde.

Einen solchen Entwurf zur dynamischen Zuweisung von Kommunikationsprotokollen ermöglicht das *Strategiemuster*. Somit gestaltet sich der Entwurf wie in Abbildung 3.8.

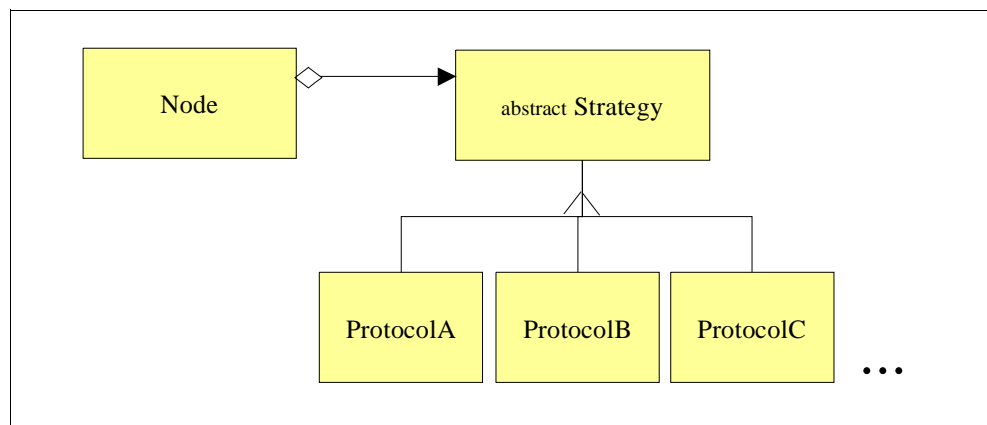


Abbildung 3.8: Dynamische Zuweisung von Kommunikationsprotokollen

3.2.2.3 Rekursive Komposition

Rekursive Komposition ist eine allgemein bekannte Technik, hierarchisch strukturierte Komponenten zu repräsentieren. Sie basiert darauf, zunehmend komplexe Komponenten aus elementaren Komponenten zusammensetzen und zu abstrahieren.

Wegen der vorhandenen Möglichkeit, mittels Visualisierungskomponenten die Simulationsmodelle bzw. ihre Komponenten abstrakt zu repräsentieren, wäre diese Technik nicht notwendig. Dennoch ist diese Technik für den Baukasten unerlässlich, da die Unterstützung der Spezifikationsprache SDL gefordert wird und diese Spezifikationsprache eine Komponente namens *Block* verwendet, die eine rekursive Komposition beschreibt. Deshalb muß ein mittels der Grundkomponenten hierarchisch strukturiertes Simulations-

modell erzeugt werden können. Deswegen werden anhand des *Kompositionsmusters* die Grundkomponenten um die Komponenten und Klassen *ConnectableObject* und *Block* zur Erzeugung von hierarchisch strukturierten Simulationsmodellen, wie in Abbildung 3.9 aufgeführt, erweitert.

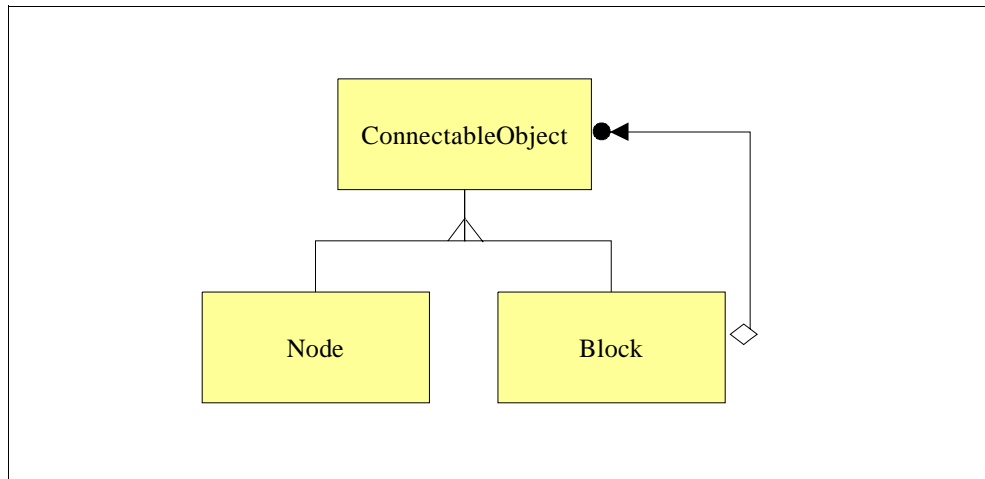


Abbildung 3.9: Hierarchie verbindbarer Elemente

Zusätzlich erleichtert diese Technik, dynamisch zur Laufzeit von einem Bereich in einem Simulationsmodell in einen abstrakteren bzw. detaillierteren Bereich überzugehen. Dies ist hilfreich, um ein Kommunikationsprotokoll je nach Bedarf detaillierter oder abstrakter zu simulieren.

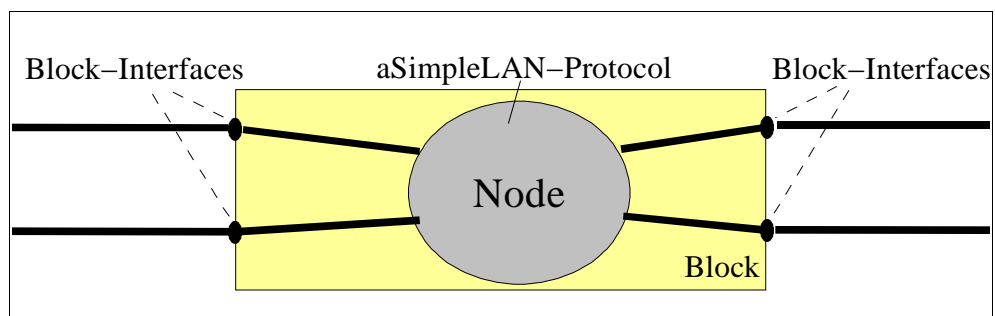


Abbildung 3.10: LAN-Bereich (abstrakt)

So kann in einem LAN-Beispiel ein abstrakter *Node*, der im *Block* (Abbildung 3.10) ein einfaches LAN simuliert, durch mehrere kleine *Nodes* ersetzt

werden, die eine detailliertere Struktur und Simulation des LAN-Beispiels abbilden (Abbildung 3.11).

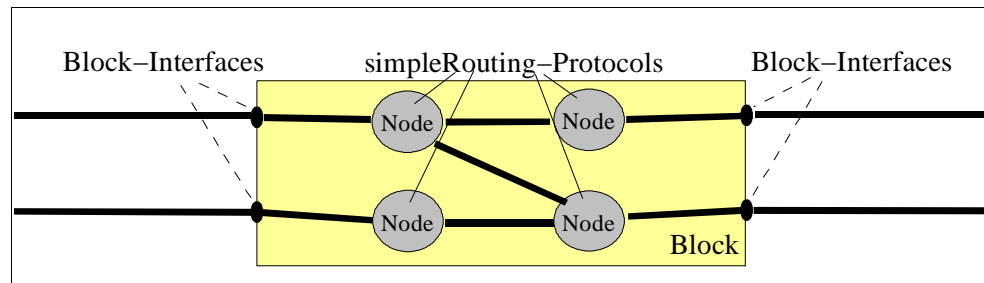


Abbildung 3.11: LAN-Bereich (detailliert)

Ebenso hilfreich ist eine hierarchisch strukturierte *Message*. Damit wird die übermittelte *Message* übersichtlicher aufgebaut und präsentiert. Durch eine hierarchische Struktur einer *Message* kann der Inhalt sinngemäß unterteilt werden. So könnte wie in Abbildung 3.12 eine *Message* zum Beispiel in Header-, Body- und Tail-Bereich unterteilt werden.

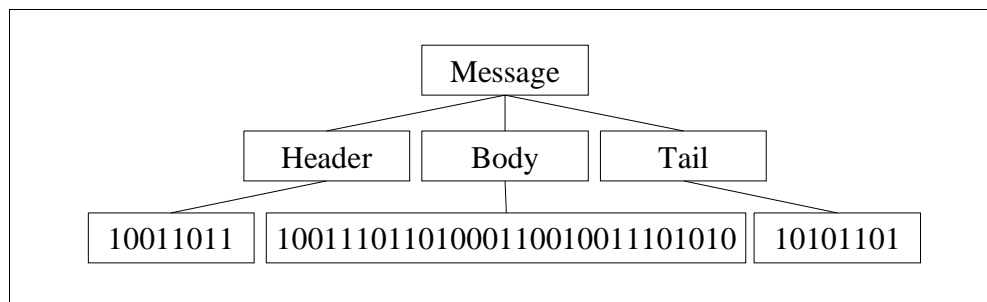


Abbildung 3.12: Mögliche Hierarchie einer Message

Die Klasse *Message* wurde deshalb wie in Abbildung 3.13 abgeändert.

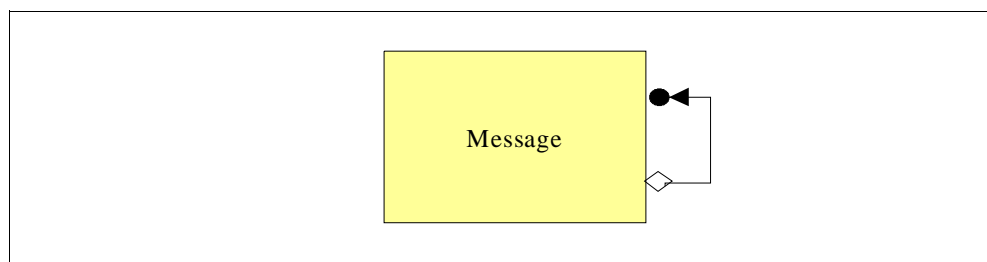


Abbildung 3.13: Entwurfsänderung der Klasse Message

3.2.2.4 Entkopplung von Abstraktion und Implementation

Die Entkopplung eines abstrakten Modells von seiner Implementation ermöglicht eine unabhängige Variation beider Bereiche. Diese Entkopplung war beim Entwurf dieser Studienarbeit bei der Klasse *Connection* sinnvoll. Somit wird eine getrennte Entwicklung zwischen den abstrakten Eigenschaften und den Eigenschaften bei einer Messageübertragung einer *Connection* erreicht.

Abstrakte Eigenschaften von *Connections* sind zum Beispiel die Kommunikationsarten (Simplex, HalfDuplex), die Einschränkung, nur zwei *Nodes* zu verbinden und die Möglichkeit, dynamisch zur Laufzeit die *Nodes* an beiden Enden zu wechseln (d.h. die *Connection* kann als Kabel gesehen werden, das beliebig umgesteckt werden kann).

Bei einer konkreten Implementation einer *Connection* werden die Eigenschaften einer Nachrichtenübertragung bestimmt. So könnte sich zum Beispiel die Signalstärke beim Übertragen in Abhängigkeit der Entfernung abschwächen.

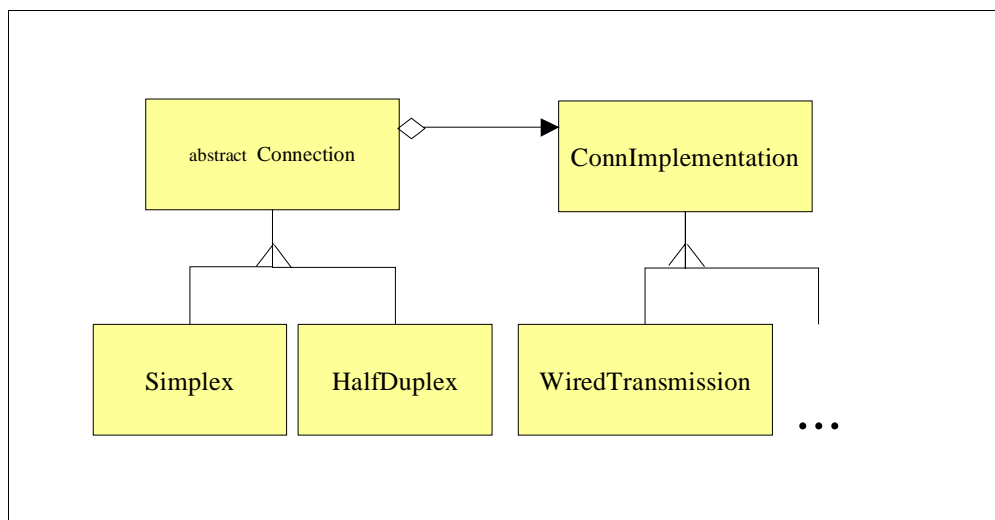


Abbildung 3.14: Entkopplung von Abstraktion und Implementation

Abbildung 3.14 zeigt den konkreten Entwurf der *Connection*, der mit Hilfe des *Brückenmusters* entworfen wurde. Dabei wurden die abstrakten Eigen-

schaften wie oben beschrieben realisiert. Der Entwurf für die konkrete Implementation der *Connection* verlief jedoch anders. In diesem Entwurf wurde hauptsächlich die Charakteristik einer Übertragung mittels Kabel berücksichtigt. Bei einer solchen Übertragung sind nur die Übertragungsgeschwindigkeit, Ausbreitungsverzögerung und eine simulierte Kollision von Nachrichten interessant. Eine Abschwächung der Signalstärke ist dabei nicht relevant und wurde vorerst ausgelassen. Diese konkret entworfene Klasse wurde *WiredTransmission* genannt.

Durch die Entkopplung der *Connection* zwischen Abstraktion und ihrer Implementation ist es aber jederzeit möglich, zusätzliche konkrete Implementationsklassen für *Connection* zu entwerfen. So wäre eine Klasse *WirelessTransmission* in einer nachfolgenden Arbeit denkbar, die dann die Möglichkeit des Baukastens auf die Simulation von Funknetzen erweitert und z.B. eine Abschwächung der Signalstärke unterstützt.

3.2.2.5 CoreFoundationClasses

Alle Klassen zur Erzeugung von Simulationsmodellen werden als *CoreFoundationClasses* bezeichnet. Die bisher noch nicht erwähnten Klassen werden nun kurz beschrieben.

Um nicht jedem *Node* einen eigenen Thread vergeben zu müssen, wurde zu der Klasse *Node* eine Unterklasse *NodeWithThread* entworfen, so daß später eine Unterklasse *NodeWithoutThread* hinzugefügt werden kann.

Mit der Klasse *Command* wurde ansatzweise das *Befehlsmuster* realisiert, um später eine Undo-Redo-Funktionalität zu ermöglichen.

Die Klasse *MessageType* dient als Typenklasse, um Teilen der *Message* einen Typ vergeben zu können.

Ansonsten wurden die restlichen Klassen zum Puffern in *Vector*-Objekten entworfen. Im Anhang zeigt die Abbildung A.1 den gesamten Entwurf der *CoreFoundationClasses*.

3.2.3 Schnittstellen

Wie bereits in Abschnitt 3.2.1 beschrieben, wird das MVC-Paradigma wie in den Swing-Komponenten angewandt. Dabei wird das im MVC-Paradigma enthaltene *Beobachtermuster* mittels eines in Java üblichen EventListener-Modells des Typs *PropertyChangeListener* realisiert.

Die Beobachter bzw. Visualisierungskomponenten bekommen das zu beobachtende Simulationsmodell als Referenz übergeben. Um die Sicht der Visualisierungskomponenten auf das Simulationsmodell auf die benötigten Methoden und Attribute zu beschränken, müssen dazwischen definierte *Schnittstellen* eingeführt werden. Diese *Schnittstellen* werden in Java mittels *Interface* realisiert. Der Entwurf der *Interfaces* (Abbildung 3.15) gibt die Methoden und Attribute an, die die Visualisierungskomponenten vom Simulationsmodell sehen dürfen. Anstatt direkte Referenzen auf das Simulationsmodell zu bekommen, werden den Visualisierungskomponenten nur noch Referenzen über die *Schnittstelle* übergeben.

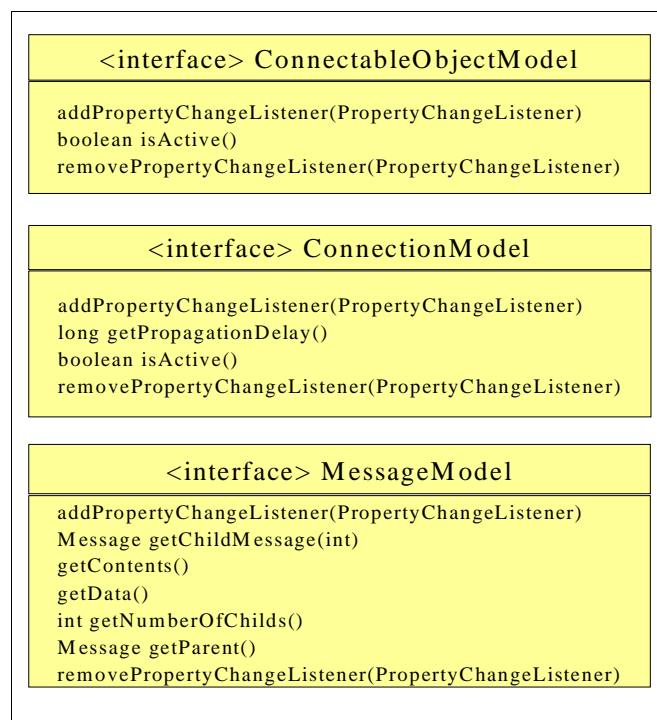


Abbildung 3.15: Schnittstellen

3.2.4 Klassen zur Visualisierung

Schon der alleinige Einsatz von Swing-Komponenten ermöglicht einigermaßen, Simulationsmodelle für Kommunikationsprotokolle zu visualisieren. Der folgende Klassenentwurf hat als Ziel, einerseits die Swing-Komponenten mit speziellen Visualisierungskomponenten für Simulationsmodelle zu erweitern; andererseits für Simulationsmodelle eingestellte Swing-Komponenten zu liefern, damit die Entwicklung von Protokollbeispielen erleichtert wird. Grundsätzlich standen zwei Entwurfsmöglichkeiten zur Wahl.

Bei der ersten Möglichkeit basiert der Entwurf auf dem *Prototypenmuster*. Dieses *Entwurfsmuster* wird für den Entwurf eines Frameworks empfohlen [GHJV96]. Wie im Kapitel zwei beschrieben wurde, ist ein Frameworkentwurf angestrebt. Deshalb würde ein frühzeitiger Einsatz des *Prototypenmusters* dem späteren Frameworkentwurf entgegenkommen.

Die zweite Entwurfsmöglichkeit ist das Beschreiben von konkreten Visualisierungen mittels eigener Klassen. Somit wird das Visualisierungsobjekt einfach durch eine Objekterzeugung der zugehörigen *konkreten Klasse* und das Setzen von wenigen Parametern erreicht.

Obwohl ein Entwurf mit Hilfe des *Prototypenmusters* im Sinne der Flexibilität und für die Erweiterung des Baukastens auf ein Framework früher oder später notwendig ist, fiel in dieser Studienarbeit die Entscheidung, die zweite Entwurfsmöglichkeit als Übergangslösung einzusetzen.

Folgende Gründe waren dafür maßgebend: Erstens ist mittels der zweiten Entwurfsmöglichkeit das Erstellen von Protokollbeispielen einfacher, so lange kein Framework mit zugehörigem Editor existiert. Zweitens wird eine visuelle Erstellung von Protokollbeispielen mit Hilfe von Java-Entwicklungswerkzeugen ermöglicht. Denn diese *konkreten Baukasten-Klassen* können als Java-Beans in vielen Java-Entwicklungswerkzeugen importiert und danach vom, im Entwicklungswerkzeug integrierten, visuellen Entwicklungseditor verwendet werden. Drittens ist ein späterer Neuentwurf wegen der strikten Trennung zwischen *Klassen zur Erzeugung von Simula-*

tionsmodellen und *Visualisierungskomponenten* und der relativ kleinen Anzahl von zusätzlich zu den Swing–Komponenten notwendigen Visualisierungskomponenten mit geringem Aufwand möglich.

Die genauere Beschreibung des Entwurfs gliedert sich in zwei Teilbereiche. Zuerst werden die Klassen erläutert, die die Swing–Komponenten ergänzen. Die restlichen aufgeführten Klassen werden in dieser Studienarbeit als *Schablonen* bezeichnet, denn sie bieten Grundeinstellungen von Swing–Klassen, um jeweils *konkrete Visualisierungsobjekte* zu beschreiben.

3.2.4.1 Zusätzliche Visualisierungskomponenten

Alle zusätzlich zu den Swing–Komponenten entworfenen Visualisierungskomponenten erben von der Swing–Klasse bzw. Swing–Komponente *JComponent*. Diese Visualisierungskomponenten unterscheiden sich hauptsächlich durch zusätzlich und unabhängig von den Swing–Komponenten gezeichneten Objekten. Folgend werden *JArrowView*, *JConnectionView*, *JFullDuplexConnectionView* und *JHistoryView* etwas näher beschrieben.

JArrowView

Die Klasse *JArrowView*, siehe Abbildung 3.16, visualisiert einen einfachen Pfeil. Ein *PropertyChangeListener* ist nicht realisiert. Dies bedeutet, daß die Darstellung des Pfeils sich nicht selbständig ändert. Die Richtung und Länge des Pfeils wird in Abhängigkeit von mehreren Parametern bestimmt, die in der

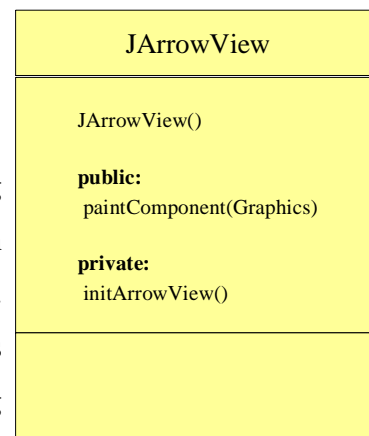


Abbildung 3.16: Klasse *JArrowView*

Vaterklasse *JComponent* definiert werden. Die Methoden zur Abänderung der Parameter sind ebenfalls in der Vaterklasse aufgeführt. Somit ist zuerst eine Instanz dieser Klasse und danach das Setzen von Parametern notwendig, um einen Pfeil darzustellen.

JConnectionView

Auf ähnliche Weise wie die Klasse *JArrowView* einen Pfeil zeichnet, zeichnet die Klasse *JConnectionView* eine Verbindung.

Wie Abbildung 3.17 zeigt, ist zusätzlich in dieser Klasse ein *PropertyChangeListener* implementiert. Mit diesem *EventListener* wird ermöglicht, einen Teil eines Simulationsmodells zu beobachten. Diese Beobachtung wird über das Interface *ConnectionModel* realisiert. Somit können Übertragungen von *Messages* auf dieser *Connection* festgelegt und dementsprechend visualisiert werden. Die *JConnectionView* visualisiert nur eine Simplex- oder HalfDuplex-Übertragung. Eine Visualisierung von FullDuplex-Übertragungen wird nicht unterstützt. Für die Visualisierung von FullDuplex-Übertragungen wurde die im nächsten Abschnitt beschriebene Klasse entworfen.

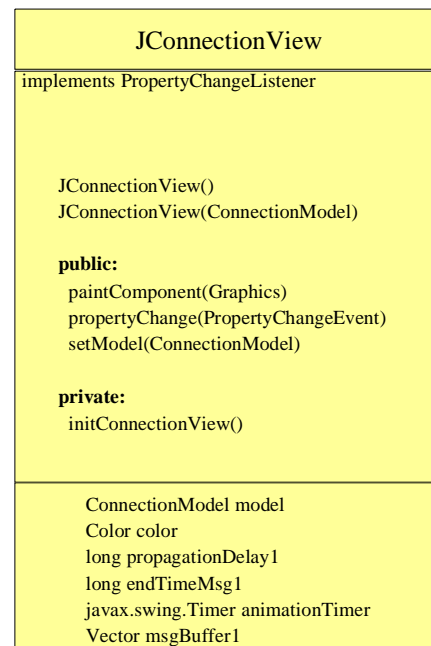


Abbildung 3.17: Klasse *JConnectionView*

Die Übertragung von *Messages* kann in den folgenden drei Varianten dargestellt werden:

- spontan vollständiges Einfärben der *Connection* bei einer Übertragung.
- schrittweises Einfärben der *Connection* bei einer Übertragung.
- Einblenden von Visualisierungskomponenten der übertragenen *Messages*.

Bei der ersten Variante ändert sich die Farbe der kompletten *Connection* für die gesamte Übertragungsdauer. So wechselt die *Connection* zum Beispiel von der Farbe Schwarz zu der Farbe Rot, sobald *Messages* übertragen werden. Wenn keine Übertragung mehr ansteht, wird wieder von Rot auf Schwarz gewechselt.

In der zweiten Variante unterscheidet sich die Darstellung von der ersten

hauptsächlich beim Einfärben der *Connection*. Das heißt, die *Connection* färbt sich in Abhängigkeit von Übertragungsrichtung, Übertragungsdauer und Ausbreitungsgeschwindigkeit.

Die letzte Variante blendet pro *Message* eine neue Visualisierungskomponente ein, die diese *Message* darstellt. Diese Visualisierungskomponente wird dann wie in Variante zwei in Abhängigkeit von Übertragungsrichtung, Übertragungsdauer und Ausbreitungsgeschwindigkeit entlang der *Connection* verschoben und danach wieder ausgeblendet.

JFullDuplexConnectionView

Die Klasse *JFullDuplexConnectionView* erbt von der Klasse *JConnectionView* und visualisiert, wie der Name schon sagt, eine FullDuplex-Übertragung. Ein Simulationsmodell kann eine FullDuplex-*Connection* nur mittels zwei Simplex- bzw. HalfDuplex-*Connections* realisieren. Dies bedeutet, daß die Klasse *JFullDuplexConnectionView* diese zwei *Connections* beobachten muß, um eine FullDuplex-Übertragung zu visualisieren. Deswegen benötigt die Klasse zwei Referenzen auf Teile des Simulationsmodells, die über das Interface *ConnectionModel* führen.

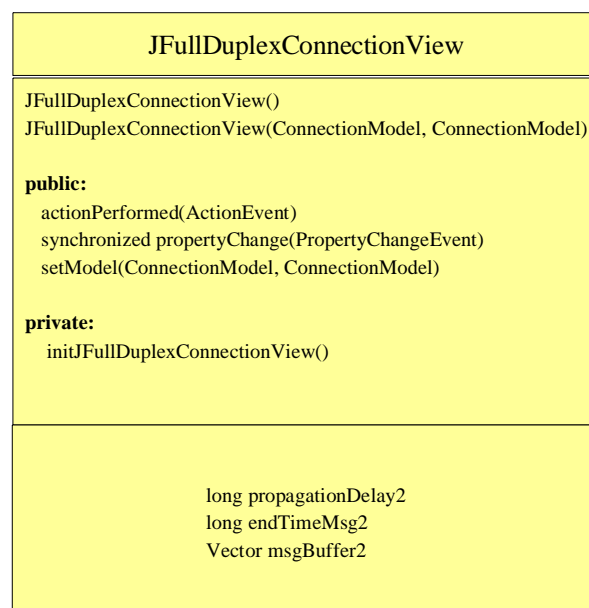


Abbildung 3.18: Klasse *JFullDuplexConnectionView*

JHistoryView

Sequenzdiagramme ermöglichen eine gute Darstellung von Historien (siehe Kapitel 2.1.1). Im Gegensatz zur Graphendarstellung kann die Positionierung von *Nodes* in Sequenzdiagrammen sinnvoll automatisiert werden.

Abbildung 3.19: Automatische Visualisierung durch die Klasse *JHistoryView*

Diese automatische Visualisierung der Historie ermöglicht die Klasse *JHistoryView*. Diese Klasse unterteilt den zugeteilten Anzeigebereich in einen Sequenzdiagramm- und einen Textausgabe-Bereich (Abbildung 3.19). Im Sequenzdiagramm Bereich werden die angegebenen *Nodes* nebeneinander angeordnet und die *Messages*, die zwischen diesen *Nodes* übertragen werden, entweder in einer Zeit- oder Interaktionsachse dargestellt. Der Textausgabe Bereich gibt jeweils den Inhalt der *Messages* aus.

Durch die automatische Visualisierung benötigt die Klasse *JHistoryView* nur eine Angabe der zu beobachtenden Komponenten. Dies wird durch die Methode *addConnectableObject(..)* ermöglicht. Wie Abbildung 3.20 zeigt, übergibt die Methode eine Referenz und den Namen der zu beobachtenden Komponente. Die Referenz führt über das Interface *ConnectableObject*.

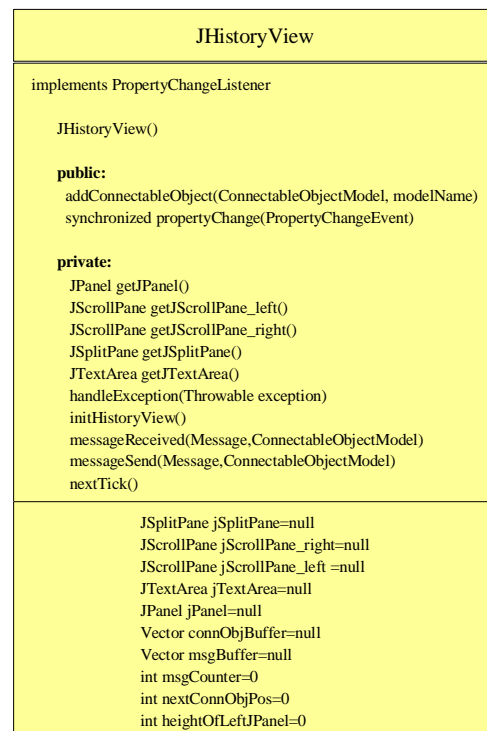


Abbildung 3.20: Klasse *JHistoryView*

3.2.4.2 Schablonen für Visualisierungsobjekte

Die im Folgenden beschriebenen Klassen sind zur *Visualisierung von Simulationsmodellen* nicht zwingend erforderlich, da sie andere Klassen verwenden, um Visualisierungsobjekte darzustellen. Zweck solcher Klassen ist, Grundeinstellungen anderer Klassen vorzugeben und zu bestimmen, welche und wie oft Klassen erzeugt werden. Zum Beispiel ist die Klasse *JComputer* nichts anderes als eine Klasse *JButton* (diese Klasse ist eine Swing-Komponente), die als Grundeinstellung einen Computer als Bild lädt und das Bild in dem Button darstellt.

Solche Klassen erleichtern das Erzeugen von Protokollbeispielen, da gewisse Einstellungen nicht jedesmal neu definiert werden müssen, sondern einfach eine Instanz dieser Klassen ausreicht. Allerdings kann die Anzahl solcher Klassen schnell anwachsen, wenn bei jeder kleinen Einstellungsvariante eine eigene Klasse entworfen wird, da dadurch die Möglichkeiten der verschiedenen Einstellungen unverhältnismäßig wachsen würden. In diesen Fällen ist es sinnvoll, eine Klasse zu wählen, die am ehesten den gewünschten Eigenschaften entspricht, und danach die Details nachzukonfigurieren.

Beispielsweise können viele Klassen zur Visualisierung von *Nodes* entworfen werden (z.B. Computer, Router, Stelle, ...), die zur Hilfe die Klasse *JButton* mit einer bestimmten Grundeinstellung verwenden. Anstatt bei jedem Bild eine Klasse zu entwerfen, ist es sinnvoller entweder die Klasse *JButton* oder z.B. *JComputer* zu verwenden und danach das neu zu ladende Bild anzugeben.

JControllBar, *JGraphArea*, *UserInterface*, *JMessageView*, *JNodeView*, *JComputer*, *JRouter*, usw sind zum Beispiel als *Schablone* verwendete Klassen, die in dieser Studienarbeit für die ersten Kommunikationsprotokollbeispiele entworfen wurden. Abschließend sei noch einmal erwähnt, daß diese Klassen als Übergangslösung gedacht sind. Sobald ein Frameworkentwurf ansteht, ist es ratsam, neue Klassen zur *Visualisierung von Simulationsmodellen* mit Hilfe des *Prototypenmusters* zu entwerfen.

3.2.4.3 *ProtoVisFoundationClasses*

Alle Klassen zur *Visualisierung von Simulationsmodellen*, ausgeschlossen die Swing-Komponenten, werden *ProtoVisFoundationClasses* genannt. Im Anhang zeigt Abbildung A.2 einen Überblick über diese Klassen.

3.3 *Implementation*

In diesem Abschnitt werden erwähnenswerte Teile der Implementation beschrieben. Zuerst sind die *CoreFoundationClasses* und danach die *Proto-visFoundationClasses* implementiert worden.

3.3.1 *HiSAP CoreFoundationClasses*

Die implementierten Klassen *NodeWithThread* und *Message* der *CoreFoundationClasses* werden in folgendem Abschnitt kurz erläutert.

3.3.1.1 *NodeWithThread*

Wie der Name schon sagt, besitzt jeder Knoten dieser Klasse einen eigenen Thread. Diese Threads übernehmen zusätzlich die Durchführung der zu den Knoten zugewiesenen Protokolle. Damit Nachrichten jederzeit bei den Knoten abgelegt werden können, selbst wenn der jeweilige Thread gerade beschäftigt ist, müssen Puffer eingeführt werden.

Jeder Knoten der Klasse *NodeWithThread* besitzt drei Puffer, *shouldSendBuffer*, *receivedBuffer* und *announcedMsgBuffer*. In dem Puffer *shouldSendBuffer* werden vom Protokoll oder vom Benutzer erzeugte Nachrichten abgelegt, die vom Knoten versendet werden sollen.

Die vom Knoten empfangene Nachrichten legen die *Connections* im Puffer *receivedBuffer* ab. Der jeweilige Thread überprüft dabei in kurzen Abständen, ob Nachrichten im Puffer sind.

Im dritten Puffer werden zusätzliche Informationen für das zugehörige Protokoll abgelegt. Wenn in diesem Puffer eine Information abgelegt wurde, wird vom Thread entweder die Methode *collisionedMessageAnnounced(...)* oder *messageCompletedSendAnnounced(...)* des Protokolls aufgerufen.

Diese Puffer dienen ausschließlich für die interne Kommunikation zwischen *CoreFoundationClasses*. Sie sollten auf keinem Fall von Protokollen verwendet werden. Wenn ein Protokoll eine Pufferverwaltung benötigt (wie z.B. *SlidingWindow*), muß in diesem Protokoll ein eigener Puffer erzeugt und evtl. mittels einen eigenes Threads verwaltet werden.

Wenn ein Knoten in der Lage sein soll, gleichzeitig über mehrere *Connections* Nachrichten zu empfangen oder zu senden, kann dies, wie Abbildung 3.21 als Beispiel gezeigt, mit mehreren Knoten realisiert werden. Der Router besitzt vier unabhängige Puffer, dadurch ist dieser Router in der Lage, vier Nachrichten gleichzeitig zu empfangen oder zu versenden.

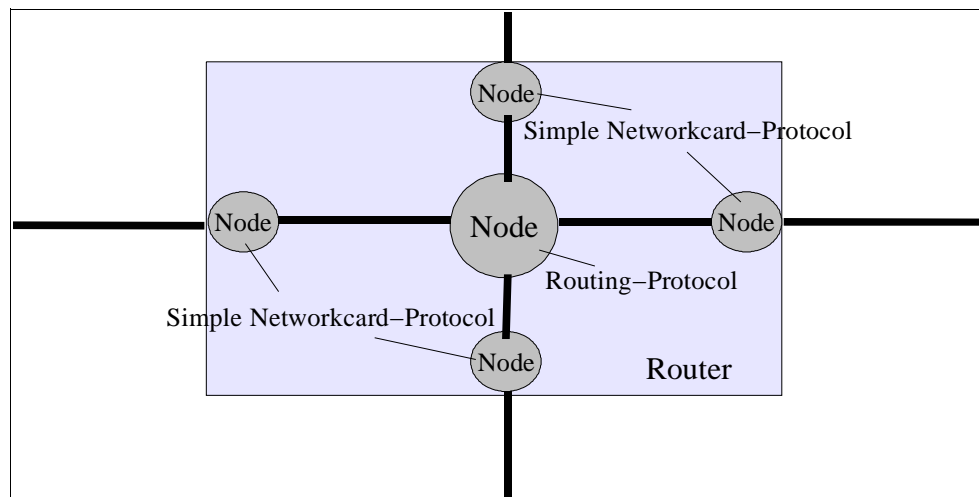


Abbildung 3.21: Router mit vier unabhängigen Puffern

Die vier äußeren Knoten simulieren sozusagen Netzwerkkarten nach, wobei jede Karte einen Puffer mit einer bestimmten Größe besitzt. Das zugehörige Protokoll leitet lediglich empfangene Nachrichten an die andere Ausgangsleitung weiter.

Das Puffern der Nachrichten übernimmt der Knoten von allein. Deshalb wird das verwendete Protokoll *Simple Network*-Protokoll genannt.

Den eigentlichen Routingvorgang des Routers übernimmt der mittlere Knoten. Deswegen muß diesem Knoten ein Routingprotokoll zugewiesen werden.

Weiterhin müssen die *Connections*, die die vier äußeren Knoten mit dem mittleren Knoten verbinden, eine geringe Länge und eine sehr große Übertragungs- und Signalausbreitungsgeschwindigkeit aufweisen. Somit werden Nachrichten zwischen den vier äußeren und dem mittleren Knoten ähnlich übertragen, wie wenn eine reale Netzwerkkarte Nachrichten über einen Bus zum Prozessor leitet.

3.3.1.2 Message

Wie im Abschnitt Entwurf beschrieben, können Nachrichten hierarchische Strukturen haben. Eine Nachricht könnte z.B. wie in Abbildung 3.22 erzeugt werden.

In diesem Beispiel wird jedem Teil dieser Ack-Nachricht ein Text übergeben. Welche Texte die eigentlichen Daten der Nachricht beschreiben und welche nicht, wird folgendermaßen entschieden:

Alle Nachrichtenteile, die keine weiteren Nachrichtenteile als Kinder besitzen, werden Blätter genannt und die restlichen Teile dagegen Zweige. Nur Texte der Blätter beschreiben die wirklichen Daten der Nachricht.

Komponenten, die Nachrichten empfangen, bekommen nur eine Referenz vom Vater Teil der Nachricht übergeben. Damit diese Komponenten nicht selber den Baum durchlaufen müssen, um die Daten der Nachricht abzufragen, wurden in der Klasse *Message* zwei Methoden realisiert.

Die Methode *getData()* durchläuft alle Kinder, sammelt die Texte der Blätter auf und gibt diese zusammengefügt als String zurück. Beispielsweise

geben folgende Aufrufe bei der erzeugten Ack-Nachricht folgendes zurück:

```
ackMsg.getData()      => 'ABBestätige Nachricht 1'
header.getData()     => 'AB'
typeSourceData.getData() => 'B'
```

```
// Message erstellen

// Teile der Message einen Typ vergeben
MessageType typeCompleteMsg      = new Compound();
MessageType typeHeader           = new Header();
MessageType typeBody             = new Body();
MessageType typeSourceData       = new Data();
MessageType typeDestinationData  = new Data();
MessageType typeBodyData        = new Data();

// Teile der Message erzeugen
Message ackMsg      = new Message(typeCompleteMsg, "Ack");
Message header     = new Message(typeHeader, "Header");
Message body       = new Message(typeBody, "Body");
Message destination = new Message(typeDestinationData, "A");
Message source     = new Message(typeSourceData, "B");
Message data       = new Message(typeBody, "Bestätige Nachricht 1.");

// Teile als komplette Message zusammenfügen
header.putChildMessage(destination);
header.putChildMessage(source);
body.putChildMessage(data);
ackMsg.putChildMessage(header);
ackMsg.putChildMessage(body);
```

Abbildung 3.22: Erzeugung einer hierarchischen Ack-Nachricht

Die zweite Methode `getContents()` liefert nur den Text des angesprochenen Nachrichtenteils zurück, unabhängig davon, ob dieses Nachrichtenteil ein Blatt oder einen Zweig darstellt. Zur Veranschaulichung werden die gleichen Aufrufe mit der Methode `getContents()` aufgeführt:

```
ackMsg.getContents()      => 'Ack'
header.getContents()     => 'Header'
typeSourceData.getContents() => 'B'
```

In manchen Fällen muß eine gleiche oder ähnliche, hierarchisch strukturierte, Nachricht neu erzeugt werden. Z.B muß bei einem Broadcast für jede Ausgangsleitung eine identische Nachricht erzeugt werden. Diese Erzeugung wird mit Hilfe des *Prototypenmusters* (vgl. Anhang Kurzbeschreibung der Entwurfsmuster) vereinfacht, denn die Klasse *Message* bietet dadurch die Methode *clone()* an. Als Rückgabeparameter dieser Methode wird eine identisch erzeugte Nachricht zurückgegeben. Ebenso kann dem Konstruktor der Klasse *Message* eine Nachricht als Vorbild übergeben werden.

Wenn beispielsweise dieselbe Ack-Nachricht, wie im obigen Beispiel erzeugt werden soll, sieht der Konstruktoraufwurf der Klasse *Message* folgendermaßen aus: `Message ackMsg2 = new Message(ackMsg);`

3.3.2 HiSAP ProtoVisFoundationClasses

Bei den implementierten *ProtoVisFoundationClasses* ist das einzig erwähnenswerte, die Entscheidung, auf welcher *Swing*-Version die *ProtoVisFoundationClasses* aufbauen sollen. Denn beim Versionswechsel von *Swing* 1.0 auf 1.1 wurde der Pfad der *Swing* Komponenten von *com.sun.java.swing.** auf *javax.swing.** umbenannt. Dies hat zur Folge, daß Klassen, die auf *Swing* 1.0 basieren, nicht mit *Swing* 1.1 funktionieren. Hinzu kam, daß zu dem Zeitpunkt der Implementation die *Java* Entwicklungswerkzeuge lediglich die Entwicklung mit *Swing* 1.0 unterstützten.

Die Wahl fiel letztendlich auf die Version 1.1, weil die mit *Swing* 1.1 aufgezählten Probleme in Zukunft verschwinden werden und durchgeführte Tests mit beiden *Swing* Versionen ergaben, daß die Version 1.1 schneller und stabiler läuft.

Abschließend wird darauf hingewiesen, daß die *Swing* Komponenten, somit auch die *ProtoVisFoundationClasses*, eine *Java Runtime Environment* 1.1.5 oder höher erfordern. Somit ist die Systemvoraussetzung des kompletten Baukastens eine *Java Runtime Environment* 1.1.5 mit *Swing* 1.1 oder höher.

4 Anwendung des neuen Baukastens

Dieses Kapitel erläutert die Anwendung des neuen Baukastens. Aufgrund des Klassenbibliothek-Konzepts benötigt der Entwickler zur Erstellung von Kommunikationsprotokollbeispielen gute Kenntnisse in der Java-Programmierung inklusive der Swing-Komponenten.

Der Einsatz von Java Entwicklungswerkzeugen, die eine visuelle Erstellung von Swing Applets bzw. Applikationen ermöglichen, ist sehr hilfreich und deshalb empfehlenswert.

Als erstes folgt eine allgemeine Beschreibung für das Erstellen von Protokollbeispielen. Danach wird die Anwendung des neuen Baukastens anhand eines Beispiels verdeutlicht.

4.1 Grundprinzipien

In den folgenden Abschnitten wird die allgemeine Anwendung und Vorgehensweise des neuen Baukastens beschrieben. Dabei gliedern sich die Abschnitte in die *Grundstrukturen*, die *Erstellung von Simulationsmodellen* und die allgemeine *Verwendung der zusätzlichen Visualisierungskomponenten*. Die Verwendung von Swing-Komponenten wird nicht näher erläutert.

4.1.1 Grundstruktur der Protokollbeispiele

Eine einheitliche Programmstruktur erhöht die Lesbarkeit der Protokollbeispiele. Deshalb wird folgend eine bestimmte Programmstruktur für Protokollbeispiele vorgeschlagen.

Jedes Protokollbeispiel-Applet bzw. -Applikation besitzt eine Methode *init()*. In dieser *init*-Methode wird das Simulationsmodell und die zugehörigen Visualisierungskomponenten aufgebaut.

Der Aufbau dieser *init*-Methode (Abbildung 4.1) wird in zwei Bereiche eingeteilt. Zuerst wird im oberen Bereich mit Hilfe des Baukastens das Simulationsmodell erzeugt danach im unteren Bereich die zugehörige Visualisierung. Die nachfolgenden Abschnitte zeigen die genauere Struktur der zwei Bereiche.

```
init()
{
// Aufbau des Simulationsmodells-----
// Knoten
...
// Verbindungen
...
// Ende: Aufbau des Simulationsmodells-----
// Visualisierung des Simulationsmodells-----
// Benutzungsschnittstelle
...
// Visualisierung der Knoten
...
// Visualisierung der Verbindungen
...
// Interne Informations- und Manipulationsfenster
...
// Ende: Visualisierung des Simulationsmodells---
System.out.println("System aufgebaut.");
}
```

Abbildung 4.1: Grundstruktur der *init*-Methode

4.1.2 Erstellung von Simulationsmodellen

Beim Erstellen eines Simulationsmodells werden zuerst die Knoten erzeugt und danach die Verbindungen.

4.1.2.1 Nodes

Jeder erzeugte *Node* muß vor dem Einsatz ein Kommunikationsprotokoll besitzen. Die Kommunikationsprotokolle gehören nicht zu den *CoreFoundationClasses* und können von jedem Protokollbeispiel-Entwickler erweitert werden. Es ist auch möglich, dynamisch zur Laufzeit einem *Node* ein anderes Kommunikationsprotokoll zuzuweisen. Jedoch sollte abgesichert werden, daß kein *Node* ohne zugewiesenes Protokoll existiert, sobald Mes-

sages versendet werden.

Abbildung 4.2 zeigt zum Beispiel eine Erzeugung von vier *Nodes*.

```
// Knoten
// Computer A
SimpleProtocol protocolA = new SimpleProtocol();
NodeWithThread computerA = new NodeWithThread(protocolA);

// Computer B
SimpleProtocol protocolB = new SimpleProtocol();
NodeWithThread computerB = new NodeWithThread(protocolB);

// Computer C
SimpleProtocol protocolC = new SimpleProtocol();
NodeWithThread computerC = new NodeWithThread(protocolC);

// Transparent Bridge
TransparentBridge transBridge = new TransparentBridge();
NodeWithThread bridgeT = new NodeWithThread(transBridge);
```

Abbildung 4.2: Erzeugen von *Nodes*

4.1.2.2 Connections

Beim Entwurf wurde die Abstraktion der *Connection* von ihrer Implementierung getrennt. Somit müssen für jede *Connection* zwei Objekte erzeugt werden: ein Objekt für die konkrete Implementation und das andere für die abstrakte Kommunikationsart. Obwohl in dieser Studienarbeit nur eine konkrete Implementation (*WiredTransmission*) existiert, ist trotzdem die getrennte Erzeugung beider Objekte erforderlich.

Abbildung 4.3 vervollständigt das obige Beispiel der vier erzeugten *Nodes* zu einem Simulationsmodell, bei dem drei Computer über eine Transparente Bridge mittels FullDuplex-Verbindungen kommunizieren können. Dabei sind die Verbindungen als *WiredTransmission* implementiert. Dieser Klasse werden drei Parameter übergeben. Der erste gibt die maximale Übertragungsgeschwindigkeit der Verbindung in Bits pro Sekunde an. Der zweite Parameter ist die Signal-Ausbreitungsgeschwindigkeit der Verbindung in Meter pro Sekunde. Mit dem dritten Parameter wird die Länge der Verbindung in Meter bestimmt. Im vorigen Beispiel bedeutet dies, daß alle drei Verbindungen 50 Bits/s übertragen, 20 Meter lang sind und jedes Signal

zwei Sekunden benötigt, um von einem Ende zum anderen Ende zu gelangen.

```
// Verbindungen
// FullDuplex-Verbindung zwischen Computer A und Bridge
WiredTransmission implAT =new WiredTransmission(50,10,20);
WiredTransmission implTA =new WiredTransmission(50,10,20);
Simplex connAT = new Simplex(computerA,bridgeT,implAT);
Simplex connTA = new Simplex(bridgeT,computerA,implTA);

// FullDuplex-Verbindung zwischen Computer B und Bridge
WiredTransmission implBT =new WiredTransmission(50,10,20);
WiredTransmission implTB =new WiredTransmission(50,10,20);
Simplex connBT = new Simplex(computerB,bridgeT,implBT);
Simplex connTB = new Simplex(bridgeT,computerB,implTB);

// FullDuplex-Verbindung zwischen Computer C und Bridge
WiredTransmission implCT =new WiredTransmission(50,10,20);
WiredTransmission implTC =new WiredTransmission(50,10,20);
Simplex connCT = new Simplex(computerC,bridgeT,implCT);
Simplex connTC = new Simplex(bridgeT,computerC,implTC);
```

Abbildung 4.3: Erzeugen von Connections

4.1.3 Verwendung von Visualisierungskomponenten

Wie im vorigem Kapitel erwähnt, ermöglichen die Swing-Komponenten, Simulationsmodelle einigermaßen zu visualisieren. Eine ausführliche Dokumentation über die Verwendung von Swing-Komponenten kann auf der Java Homepage von Sun bezogen werden [Sun99].

In diesem Abschnitt wird nur die allgemeine Verwendung von zusätzlichen, in dieser Studienarbeit implementierten, Visualisierungskomponenten dokumentiert. Die besprochenen Visualisierungskomponenten werden in der Reihenfolge aufgeführt, wie sie auch im Programmcode eines Protokollbeispiels aufgerufen werden sollen.

Zuerst wird eine grob definierte Benutzungsschnittstelle ausgewählt. Danach werden Visualisierungskomponenten für die Knoten erzeugt und den bestimmten Knoten zugewiesen. Genauso wird mit den Visualisierungskomponenten für die Verbindungen verfahren. Zum Schluß können interne Eingabefenster definiert und im Hintergrund erzeugt werden, um dem Benutzer bestimmte Interaktionen zu ermöglichen.

4.1.3.1 Benutzungsschnittstelle

Eine Auswahl einer grob definierten Benutzungsschnittstelle ist zwar nicht zwingend erforderlich, sie erleichtert jedoch erheblich, eine bestimmte Visualisierung eines Protokollbeispiels festzulegen. Eine grob definierte Benutzungsschnittstelle legt ein Hauptfenster, die Einteilung der Visualisierungskomponenten und meistens eine *allgemeine Steuerungsleiste* und die Position der *JHistoryView* fest. In dieser Studienarbeit stand bisher nur eine Benutzungsschnittstelle vom Baukasten zur Verfügung, so daß diese im folgenden Code-Beispiel Verwendung findet. Dieses Beispiel basiert auf den vorigen Beispielen, die ein Simulationsmodell mit drei Knoten erzeugt haben.

Abbildung 4.4 zeigt die Erzeugung der momentan einzigen Benutzungsschnittstelle. Dabei legt diese Benutzungsschnittstelle einen allgemeinen Visualisierungsbereich und einen Bereich für die *JHistoryView* Komponente fest. Intern wird ein *JHistoryView*- und ein *JControlBar*-Objekt erzeugt. Zu der *JHistoryView* müssen diejenigen Knoten hinzugefügt werden, die in der Historie angezeigt werden sollen. Zum Schluß wird ein *JGraphArea*-Objekt erzeugt, das später in den allgemeinen Visualisierungsbereich der Benutzungsschnittstelle eingefügt wird.

```
// Benutzungsschnittstelle
// Benutzungsschnittstelle erzeugen
UserInterface aUserInterface =new UserInterface();

// Von der Benutzungsschnittstelle erzeugte JHistoryView
// anfordern und bestimmte Knoten hinzufügen
JHistoryView aHistoryView =aUserInterface.getJHistoryView();
aHistoryView.addConnectableObject(computerA, "Computer A");
aHistoryView.addConnectableObject(computerB, "Computer B");
aHistoryView.addConnectableObject(bridgeT, "Transp-Bridge");
aHistoryView.addConnectableObject(computerC, "Computer C");

// Erzeugen eines Visualisierungsbereichs, des später zur
// Benutzungsschnittstelle hinzugefügt wird
JGraphArea aGraphArea = new JGraphArea();

// Ende: Benutzungsschnittstelle
```

Abbildung 4.4: Erzeugung einer Benutzungsschnittstelle

4.1.3.2 Visualisierung der Knoten

Ein Knoten kann von einer oder vielen Visualisierungskomponenten dargestellt werden. Die Erzeugung der Visualisierungskomponenten und Festlegung der Knoten zu den Visualisierungskomponenten wird in Abbildung 4.5 anhand eines kleinen Beispiels gezeigt.

Für den Knoten "Computer A" wird zum Beispiel eine Visualisierungskomponente *JComputer* erzeugt und zwei Parameter übergeben. Der erste Parameter dient zur Beschriftung der Visualisierungskomponente und der zweite gibt den zu beobachtenden Knoten an.

```
// Visualisierung der Knoten

// Computer A visualisieren
JComputer jCompA =new JComputer("Computer A", computerA);

// Computer B visualisieren
JComputer jCompB =new JComputer("Computer B", computerB);

// Computer C visualisieren
JComputer jCompC =new JComputer("Computer C", computerC);

// Transparent Bridge visualisieren
JBridge aBridge =new JBridge("Transparent Bridge", bridgeT);

// Ende: Visualisierung der Knoten
```

Abbildung 4.5: Visualisierung der Knoten

4.1.3.3 Visualisierung der Verbindungen

Nun muß die Darstellung der Verbindungen definiert werden. Dabei können ebenfalls mehrere Visualisierungskomponenten eine Verbindung visualisieren. Zusätzlich sind manche Visualisierungskomponenten in der Lage, zwei Verbindungen zu visualisieren.

Zum Beispiel visualisiert eine *JFullDuplexConnectionView* zwei HalfDuplex- oder Simplex-Verbindungen. Abbildung 4.6 zeigt die Erstellung von *JFullDuplexConnectionViews* mit zwei zu beobachtenden *Connections* als Parameter.

```
// Visualisierung von Verbindungen

// FullDuplex-Verbindung(computerA-bridgeT) visualisieren
JFullDuplexConnectionView jFullConnAT =new JFullDuplexCon-
nectionView(connAT, connTA);

// FullDuplex-Verbindung(computerB-bridgeT) visualisieren
JFullDuplexConnectionView jFullConnBT =new JFullDuplexCon-
nectionView(connBT, connTB);

// FullDuplex-Verbindung(computerC-bridgeT) visualisieren
JFullDuplexConnectionView jFullConnCT =new JFullDuplexCon-
nectionView(connCT, connTC);

// Ende: Visualisierung von Verbindungen
```

Abbildung 4.6: Visualisierung von Verbindungen

4.1.3.4 Interne Informations- und Manipulationsfenster

Der neue Baukasten unterstützt noch nicht die in dieser Studienarbeit beschriebene *allgemeine Steuerung*. Deshalb beschränkt sich die Interaktionsmöglichkeit des Benutzers auf die Eingabe von Daten und Steuerbefehlen in *interne Fenster*.

Diese Fenster werden durch die Swing-Komponente *InternalFrame* realisiert. Somit bestimmt der Entwickler allein durch die Festlegung, bei welchen Visualisierungskomponenten *interne Fenster* geöffnet werden und welche Interaktionen die jeweiligen Fenster bieten, die Interaktionsmöglichkeiten des Benutzers.

Weil bei der Erzeugung solcher *internen Fenster* nur Swing-Komponenten benötigt werden, wird darauf nicht näher eingegangen. Dafür wird die Einbindung von *internen Fenstern* in Protokollbeispielen genauer aufgezeigt.

```
// Interne Informations- und Manipulationsfenster

// Ein internes Fenster erzeugen
JInternalFrame aInternalFrame =new JInternalFrame();
aInternalFrame.setTitle("Transparent-Bridge");
aInternalFrame.setVisible(false);
...

// Internes Fenster dem Topologiebereich hinzufügen
aGraphArea.add(aInternalFrame);

// Zugehörige Vis-Komponente zum ActionListener hinzufügen
aBridge.addActionListener(this);

// Ende: Interne Informations- und Manipulationsfenster
```

Abbildung 4.7: Interne Informations- und Manipulationsfenster

Abbildung 4.7 zeigt ein kleines Beispiel, wie ein *internes Fenster* erzeugt und eingebunden wird.

Visualisierungskomponenten zur Visualisierung von Knoten sind in der Lage, einen *ActionEvent* auszulösen, sobald sie angeklickt werden. Wenn daraufhin ein *internes Fenster* geöffnet werden soll, muß zuvor diese Visualisierungskomponente zum *ActionListener* hinzugefügt werden. Der *ActionListener* ruft dabei die Methode *actionPerformed* auf. Der Entwickler muß in dieser Methode eine Abfrage einbauen, welche Visualisierungskomponente dieses Event auslöste und welche Aktionen durchgeführt werden sollen. So wird z.B. das Fenster geöffnet, wenn die entsprechende Visualisierungskomponente der Auslöser dieses Event war.

4.2 Ein Anwendungsbeispiel

In diesem Abschnitt wird die Anwendung des im vorangegangenen Kapitel entwickelten Baukastens anhand eines Beispiels verdeutlicht. Als Referenzbeispiel wurde das asymmetrische Zweiweg-Authentifikationsprotokoll ausgewählt. Die Vorstellung des Anwendungsbeispiels erfolgt hier in drei Schritten. Zuerst wird in kurzen Worten das asymmetrische Zweiweg-Authentifikationsprotokoll, in Anlehnung an das Vorlesungsskript *Grundlagen der Verteilten Systeme* [Rot98], wiedergegeben. Im Anschluß daran wird ein für dieses Protokoll notwendiges Simulationsmodell erzeugt. Der letzte Schritt zeigt einen möglichen graphischen Aufbau des Simulationsmodells.

Dieses Anwendungsbeispiel wurde mit Hilfe des Entwicklungswerkzeugs VisualAge für Java 2.0 erstellt. Manche Abschnitte demonstrieren, wie der neue Baukasten in einem Java Entwicklungswerkzeug eingesetzt werden kann.

4.2.1 Asymmetrische Zweiweg-Authentifikation

Das asymmetrische Zweiweg-Authentifikationsprotokoll basiert auf einem asymmetrischen Kryptosystem. Dieses System verwendet zur Verschlüsselung den Private- und Public-Key Algorithmus. Das Protokoll dient zur Authentifikation von Kommunikationspartnern untereinander. Dazu wird eine Schlüsselvergabe mittels eines Authentifikationsservers benötigt.

Wie eine Authentifikation zweier Kommunikationspartners erfolgt, wird mittels eines Kommunikationsablaufes demonstriert. In dieser Demonstration ist Knoten A der Initiator und Knoten B der Empfänger. Der Initiator besitzt schon den Public-Key vom Authentifikationsserver (PK_{AS}) und seinen eigenen Secret-Key (SK_A). Der Empfänger hat ebenfalls den Public-Key vom Authentifikationsserver (PK_{AS}) und den eigenen Secret-Key (SK_B). Die Public-Keys von Knoten A (PK_A) und B (PK_B) und den eigenen Secret-Key (SK_{AS}) verwaltet der Authentifikationsserver.

Der Kommunikationsablauf ist in Abbildung 4.8 dargestellt.

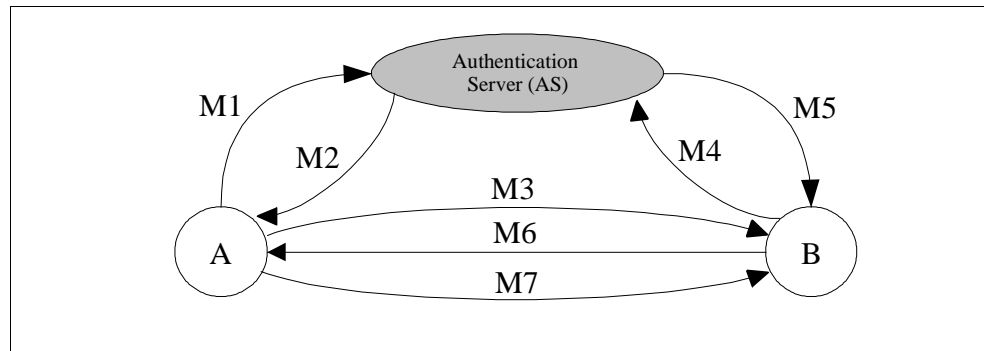


Abbildung 4.8: Asymmetrische Zweiweg-Authentifikation

Zur Veranschaulichung werden die Inhalte der in Abbildung 4.8 dargestellten Nachrichten aufgeführt.

Nachrichteninhalt:

- (M1) A → AS : A,B
- (M2) AS → A : {PK_B, B}SK_{AS}
- (M3) A → B : {R_A, A}PK_B
- (M4) B → AS : B, A
- (M5) AS → B : {PK_A, A}SK_{AS}
- (M6) B → A : {R_A, R_B}PK_A
- (M7) A → B : {R_B}PK_B

A bzw. B : Namen der Kommunikationspartner

PK_x bzw. Sk_x : Beinhalten zwei Zahlen (key: Zahl, r: Zahl)

R_A bzw. R_B : Zufallszahlen

4.2.2 Simulationsmodell

Dieses Anwendungsbeispiel wird ausschließlich für Lehrzwecke entwickelt. Um einen möglichst guten Lernerfolg zu erzielen, muß das Simulationsmodell speziell auf die Lehre abgestimmt werden. So sollte das Simulationsmodell dem Benutzer ermöglichen, bestimmte Bereiche des Protokolls zu variieren. Zum Beispiel wird ein Verschlüsselungsprotokoll vom Lernenden schneller verstanden, wenn er nach Belieben den Verschlüsselungsalgorithmus auf Nachrichten anwenden kann und dabei die daraus folgenden Auswirkungen dargestellt werden. Eine weitere Lernhilfe ist das Aufzeigen von Schwachstellen, die bei bestimmten Änderungen im Protokoll vorkommen.

Generell wird das Interesse des Benutzers, das Anwendungsbeispiel zum Erlernen des Protokolls zu benutzen, gesteigert, wenn das Anwendungsbeispiel wie ein Spiel aufgebaut ist. Deshalb verfügt das folgende Anwendungsbeispiel über verschiedene Levels. In diesen Levels wird der Benutzer in die Rolle eines Angreifers versetzt und hat zur Aufgabe, Schwachstellen der jeweiligen Protokolle aufzuspüren und auszunutzen.

4.2.2.1 Struktur der Kommunikationskanäle

Klassisch werden Beispiele des asymmetrischen Zweiweg-Authentifikationsprotokolls mit zwei Benutzern (A und B) und einem Authentifikationsserver (AS) aufgezeigt. Dabei werden diese Kommunikationspartner wie in Abbildung 4.9 verbunden.

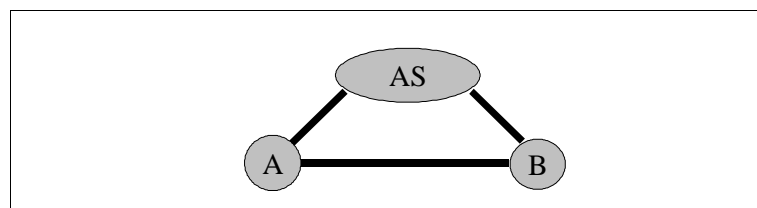


Abbildung 4.9: Mögliche Verbindungen des Anwendungsbeispiels

Es stellt sich die Frage, welche Kommunikationsarten diese Verbindungen benutzen sollen. Da eine Kollision von Nachrichten (in einer HalfDuplex-*Connection*) und eine Übertragung in nur eine Richtung (mittels eine Simplex-*Connection*) nicht erwünscht ist, ist die Kommunikationsart FullDuplex-*Connection* die naheliegendste Lösung. Diese Kommunikationsart wird durch zwei Simplex-*Connections* realisiert.

Beim Hinzufügen des Angreifers stellte sich heraus, daß durch die verschiedene Angriffspunkte insgesamt sieben Varianten von Attacken möglich sind. Aufgrund der absichtlich weggelassenen Unterstützung des neuen Baukastens, innerhalb von Verbindungen Nachrichten entfernen oder hinzufügen zu können, muß für das Entfernen bzw. Hinzufügen von Nachrichten durch den Angreifer zwischen jeder Verbindung ein Knoten geschaltet und der Knoten mit dem Angreifer verbunden werden (Abbildung 4.10).

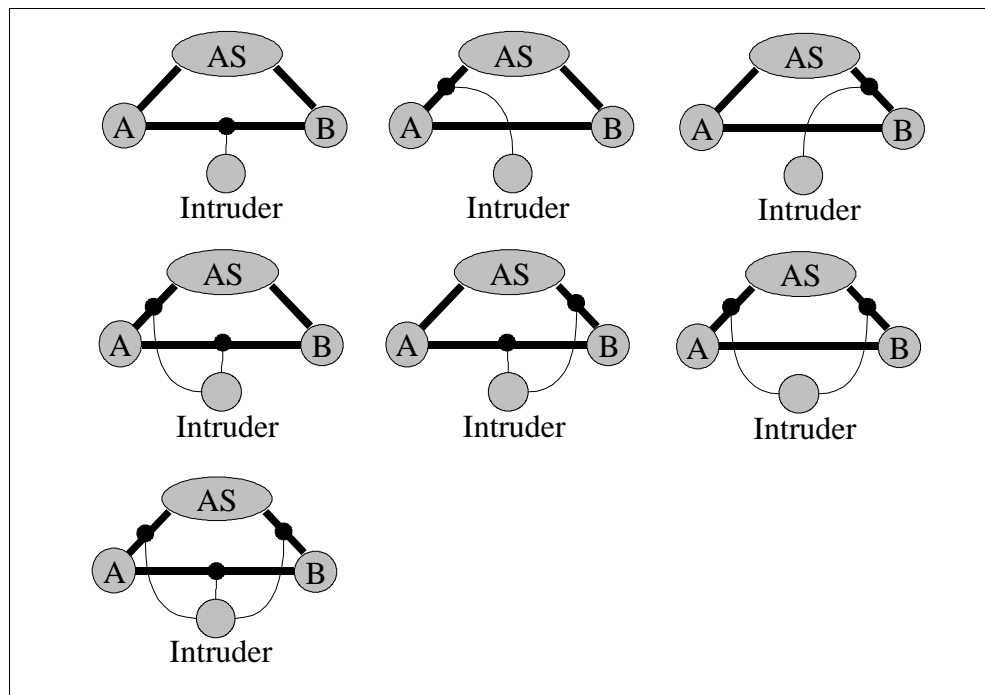


Abbildung 4.10: Mögliche Attacken

Dies ist mit dem neuen Baukasten ohne Probleme möglich, doch der Programmieraufwand wäre für diese Studienarbeit zu hoch. Deshalb wurde zur Vereinfachung eine andere Struktur der Kommunikationskanäle gewählt. In

dieser Struktur wird ein einfacher Router zur Verteilung von Nachrichten eingesetzt. Abbildung 4.11 zeigt nun die im Anwendungsbeispiel verwendeten Verbindungen.

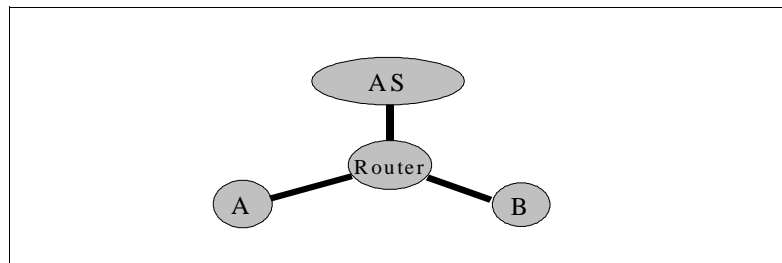


Abbildung 4.11: Verwendete Verbindungen des Anwendungsbeispiels

Diese Struktur ermöglicht dem Angreifer zu jeder Zeit, ausgetauschte Nachrichten abzuhören, zu entfernen oder neue Nachrichten an den gewünschten Kommunikationspartner zu versenden, indem er einfach wie in Abbildung 4.12 den Router *anzapft*. Wenn der Angreifer keinen Angriff während eines Kommunikationsablaufs durchführt, verhält sich dieses Protokoll genauso als wäre der Angreifer nicht vorhanden. Somit kann das Simulationsmodell gleich mit dem *anzapfenden* Angreifer aufgebaut werden.

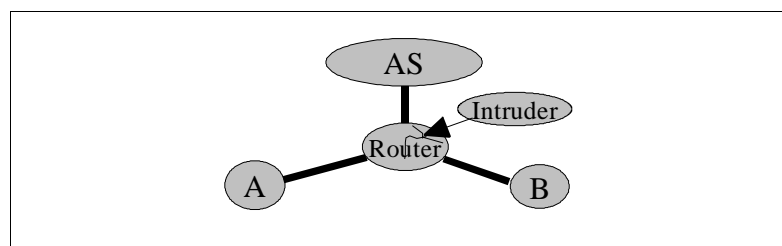


Abbildung 4.12: Anwendungsbeispiel mit Angreifer

Durch das Zusammenschmelzen von sieben auf eine Variante von Attacken, verkleinert sich der Programmieraufwand auf eine in dieser Studienarbeit durchführbare Größe.

4.2.2.2 Nachrichten

Dieser Abschnitt beschreibt den Aufbau der einzelnen Nachrichten, die im Anwendungsbeispiel verwendet werden. Alle Nachrichten haben, wie Abbildung 4.13 zeigt, folgende Grundstruktur: Header, Data.

Der Header dient als einfache Information für den dazwischen geschalteten Router. Diese Information wird vor dem Benutzer größtenteils verdeckt, damit er nicht unnötig von dieser unwichtigen Information verwirrt wird.

Die für den Benutzer interessantere Information steht im Datenbereich. Dieser Bereich kann aus beliebig vielen Hierarchieebenen bestehen, wobei die Blätter, die eigentlichen Daten, und die Zweige eventuelle Informationen in abstrakter Form enthalten. Die abstrakte Information ist nicht notwendig und wird nur zur besseren Übersicht verwendet. Jeder Datenteil einer Nachricht besteht aus drei Zeichen.

Der Angreifer sieht z.B. die Nachricht (M1) nicht wie in Abbildung 4.13 dargestellt, sondern folgendermaßen: ' A B'. Die im Anhang gezeigte Nachricht (M2), Abbildung A.4, z.B. so: '1R35T4 D'.

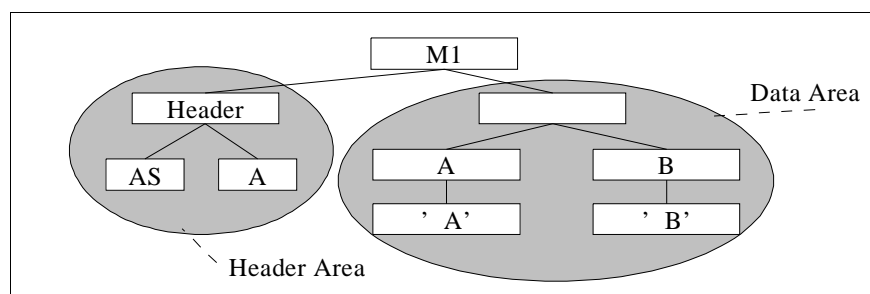


Abbildung 4.13: Nachricht (M1)

Im Anhang zeigen Abbildung A.3 bis Abbildung A.9 alle Nachrichten, die für den Kommunikationsablauf in diesem Anwendungsbeispiel benötigt werden.

4.2.2.3 Kommunikationsprotokolle und Algorithmen

Weil in dem neuen Baukasten noch keinerlei Protokolle und Algorithmen dieser Art existieren, müssen diese noch entwickelt werden. Für das Anwendungsbeispiel sind zwei Protokolle und ein Algorithmus erforderlich.

Das erste Protokoll stellt ein einfaches Routingprotokoll dar, das vom Angreifer manipuliert wurde. Durch die Manipulation kann der Angreifer bestimmen, ob die nächste vom Router empfangene Nachricht zu ihm geleitet werden soll.

Das zweite Protokoll ist das eigentliche asymmetrische Zweiweg-Authentifikationsprotokoll. Dieses Protokoll verwendet einen asymmetrischen Verschlüsselungsalgorithmus. Das Zweiweg-Authentifikationsprotokoll wird durch zwei Teilprotokolle realisiert, nämlich durch das AuthentifikationsServer-Protokoll (dieses Protokoll ist für die Schlüsselvergabe zuständig) und das AuthentifikationsClient-Protokoll (dieses Protokoll ist für den gesicherten Kommunikationsaufbau zwischen Clients zuständig).

Eine detaillierte Beschreibung der entwickelten Protokolle und Algorithmen und die komplette Abbildung des Programmcodes würde mal wieder den Rahmen dieser Studienarbeit sprengen. Aus diesem Grund werden die Kommunikationsprotokolle und Algorithmen nur im Groben beschrieben.

Asymmetrischer Verschlüsselungsalgorithmus RSA

Der RSA-Algorithmus wurde, wie in [Rot98] beschrieben, in seiner einfachsten Form implementiert. Grundprinzip des Algorithmus ist die Berechnung:

$$(Zahl)^{PK \text{ oder } Sk} \bmod(r) = \text{Ergebnis}$$

Die Zahl beschreibt den numerischen Wert eines Zeichens. Ebenso wird das Ergebnis als Zeichen wiedergegeben.

Die obige Formel zeigt, daß der Algorithmus einen Text zeichenweise ver-

schlüsselt. Um einen ständigen Aufruf des Algorithmus pro Zeichen zu vermeiden, wird als Übergabeparameter ein String erwartet. Somit kann dem Algorithmus ein kompletter Text übergeben werden. Der Algorithmus zerlegt den Text selbständig und fügt ihn verschlüsselt wieder zusammen.

AuthentifikationsServer-Protokoll

Das Protokoll des AuthentifikationsServers (*AuthenticationServer*) hat als einzige Aufgabe, die öffentlichen Schlüssel der beteiligten Kommunikationspartner zu verwalten und zu verteilen. Dabei muß garantiert sein, daß die verwalteten öffentlichen Schlüssel auch wirklich die Schlüssel der angegebenen Kommunikationspartner sind. Genauso müssen die Kommunikationspartner den wirklichen Public-Key des AuthentifikationsServers besitzen.

Für die Verwaltung der öffentlichen Schlüssel benötigt das AuthentifikationsServer-Protokoll eine Tabelle. Diese Tabelle wird beim Erzeugen des AuthentifikationsServer-Protokolls angelegt. Einträge in diese Tabelle können mit der Methode *addToPKTable(...)* vorgenommen werden.

Im Gegensatz zum AuthentifikationsClient-Protokoll muß das AuthentifikationsServer-Protokoll keine verschiedenen Zustände annehmen. Der AuthentifikationsServer muß nur Public-Key-Anfragen bearbeiten und bei Erfolg eine mit dem SK(AS) verschlüsselte Antwort zurückschicken.

AuthentifikationsClient-Protokoll

Das AuthentifikationsClient-Protokoll (*AuthenticationClient*) beschreibt beim Aufbau einer Zweiweg-Authentifikation einerseits den Initiator als auch andererseits den Empfänger. Für diese Aufgabe benötigt das Protokoll einen Boolean-Flag, das beim Initiator, bzw. beim Versenden der ersten Nachricht (M1), gesetzt wird. Dieses Flag bestimmt die Rolle des Protokolls. Entweder hat es die Rolle des Initiators oder die Rolle eines Empfän-

gers. Beide Rollen gehen in drei unterschiedliche Zustände über. Zum Vermerk der Zustände wird eine Integer Variable verwendet, die den Wert 0, 1 oder 2 annehmen kann.

Der Zustand 0 ist der Anfangszustand, bei dem die Protokolle entweder eine Nachricht (M3) empfangen oder die Anweisung bekommen können, den Kommunikationsaufbau selbst durchzuführen. Das Protokoll erzeugt beim selbständigen Kommunikationsaufbau eine Nachricht (M1) und sendet sie zum Authentifikationsserver.

Bei den anderen Zuständen erwartet das Protokoll eine bestimmte Nachricht. Wenn diese Nachricht nicht empfangen wird, wird eine Fehlermeldung ausgegeben und das Protokoll angehalten. Danach werden alle empfangenen Nachrichten ignoriert. Damit das Protokoll wieder arbeitet, ist es erforderlich, mittels der Methode *start()* den Zustand des Protokolls komplett zurückzusetzen.

Einfaches manipuliertes Routing-Protokoll

Das vom Angreifer manipulierte Routingprotokoll (*HackedSimpleRouting*) erbt von der Klasse *SimpleRouting* und erweitert diese Klasse um die Abfrage eines Boolean-Wertes. Dieser Boolean-Wert gibt an, ob der Angreifer die nächste Nachricht abfangen möchte. Wenn der Angreifer dies möchte, leitet der Router die Nachricht zum Angreifer weiter, anstatt zur nächsten Ausgangsleitung.

Das Routing-Protokoll besitzt eine Routingtabelle, die mit der Methode *addToRoutingTable(...)* konfiguriert wird. Als Parameter werden ein String und zwei Integer Werte übergeben. Der String-Wert gibt den benötigten Inhalt des Ziel-Headers an, damit die Nachricht auf die vom ersten Integer-Wert angegebene Ausgangsleitung weitergeleitet werden kann. Der optionale zweite Integer-Wert gibt die bis zum Ziel benötigten Hops⁵ an.

⁵ Hops gibt die Anzahl der zu passierenden Knoten an.

4.2.2.4 Implementation

Folgend werden einige Programmcode–Ausschnitte der `init`–Methode des Anwendungsbeispiels, die das Simulationsmodell erzeugen, aufgeführt.

Protokolle

Bevor Knoten für die Benutzer A und B, den Router und den Authentifikationsserver erzeugt werden können, müssen die zuständigen Kommunikationsprotokolle erzeugt und initialisiert werden. Abbildung 4.14 zeigt, wie die einzelnen Protokolle initialisiert werden. Die Klasse `AuthenticationClient(...)` benötigt die ersten zwei Übergabeparameter, um ihren eigenen Secret Key (SK, r), und die letzten zwei Übergabeparameter, um den Public Key (PK, r) des Authentifikationsserver zu konfigurieren. Weil die Klasse `AuthenticationServer(...)` eine Public Key Tabelle verwaltet, benötigt sie nur die Angabe ihres eigenen Secret Keys. Nachfolgend müssen die zugehörigen Tabellen konfiguriert werden. Beim Hinzufügen der Werte muß darauf geachtet werden, daß die übergebenen Strings immer drei Zeichen lang sind.

```
/ init Strategy
client1= new AuthenticationClient(305, 91, 35, 119);
client2= new AuthenticationClient(77, 119, 35, 119);
anAS    = new AuthenticationServer(11, 119);
hackedRouter = new HackedSimpleRouting();

// set routing
hackedRouter.addToRoutingTable(" A", 1);
hackedRouter.addToRoutingTable(" B", 2);
hackedRouter.addToRoutingTable(" AS", 3);

// set pkTable of AS
anAS.addToPkTable(" a", 17, 91);
anAS.addToPkTable(" b", 5, 119);
```

Abbildung 4.14: Authentifikations– und Routingprotokoll initialisieren

Knoten

Wie Abbildung 4.15 zeigt, werden die Knoten in etwa so aufgerufen, wie es im Abschnitt *Grundprinzipien* beschrieben wurde.

```
// init Node
node1= new NodeWithThread(client1);
node2= new NodeWithThread(client2);
node3= new NodeWithThread(anAS);
nodeR= new NodeWithThread(hackedRouter);
```

Abbildung 4.15: Knoten initialisieren

Connections

Die Erzeugung der in Abbildung 4.16 gezeigten Verbindungen unterscheidet sich ebenfalls kaum von der im Abschnitt *Grundprinzipien* beschriebenen Erzeugung von Verbindungen. Wie schon erwähnt, werden die Knoten mittels FullDuplex-Verbindungen verbunden, die durch zwei Simplex-*Connections* realisiert werden.

```
// init and set connections

WiredTransmission impl1R = new WiredTransmission(0, 10, 20);
WiredTransmission impl2R = new WiredTransmission(0, 10, 20);
WiredTransmission impl3R = new WiredTransmission(0, 10, 20);
WiredTransmission implR1 = new WiredTransmission(0, 10, 20);
WiredTransmission implR2 = new WiredTransmission(0, 10, 20);
WiredTransmission implR3 = new WiredTransmission(0, 10, 20);

conn1R = new Simplex(node1,nodeR,impl1R);
connR1 = new Simplex(nodeR,node1,implR1);
conn2R = new Simplex(node2,nodeR,impl2R);
connR2 = new Simplex(nodeR,node2,implR2);
conn3R = new Simplex(node3,nodeR,impl3R);
connR3 = new Simplex(nodeR,node3,implR3);
```

Abbildung 4.16: Verbindungen initialisieren

4.2.3 Graphischer Aufbau des Simulationsmodells

Das Simulationsmodell ist nun lauffähig. Die Darstellung, wie das Simulationsmodell visualisiert wird, wird extra bestimmt und in diesem Abschnitt beschrieben.

Zuerst werden die vom neuen Baukasten angebotenen Visualisierungskomponenten verwendet. Diese werden in dem Abschnitt *Benutzungsschnittstelle* erläutert. Danach werden interne Fenster allein mit den Swing-Komponenten erzeugt. Diese Fenster stellen dem Benutzer Protokollinformationen und Interaktionsmöglichkeiten zur Verfügung. Zum Schluß müssen alle Visualisierungskomponenten, die von *JButton* erben, in den *ActionListener* eingetragen und die Methode *ActionPerformed* angepaßt werden.

4.2.3.1 Benutzungsschnittstelle

In dem Anwendungsbeispiel wird als Benutzungsschnittstelle die Klasse *UserInterface* verwendet. Nach der Erzeugung dieser Klasse muß ein Objekt der Klasse *JGraphArea* hinzugefügt werden. In diesem Objekt werden die Visualisierungskomponenten für die Knoten, Verbindungen und *internen Fenster* positioniert und dargestellt.

Bevor dieses Objekt der Benutzungsschnittstelle hinzugefügt wird, wird zur Demonstration der Graphenbereich mit Hilfe des Entwicklungswerkzeugs VisualAge für Java erstellt. In die Entwicklungswerkzeuge müssen zuvor die Klassen des neuen Baukastens importiert werden. Weiterhin wird darauf hingewiesen, daß der visuelle Editor des Entwicklungswerkzeugs die Swing Komponenten 1.1 oder höher verwenden muß.

Wenn alle Klassen des neuen Baukastens im Entwicklungswerkzeug vorhanden sind, werden die *Klassen zur Visualisierung* vom Entwicklungswerkzeug wie Java-Beans behandelt. Das bedeutet, daß diese Klassen im Entwicklungswerkzeug mit kleinen Einschränkungen wie die Swing Komponenten behandelt werden. Somit kann der erste Grundaufbau sehr leicht

für die Darstellung des Anwendungsbeispiels entwickelt werden.

Abbildung 4.17 zeigt, wie das Anwendungsbeispiels visuell erstellt wurde. Zuerst ist ein Objekt der Klasse *JGraphArea* erzeugt worden (dieses Objekt hat den Namen *TopologyView1*), danach sind Visualisierungskomponenten in dieses Objekt (der quadratische Bereich) hineingezogen, positioniert und bearbeitet worden.

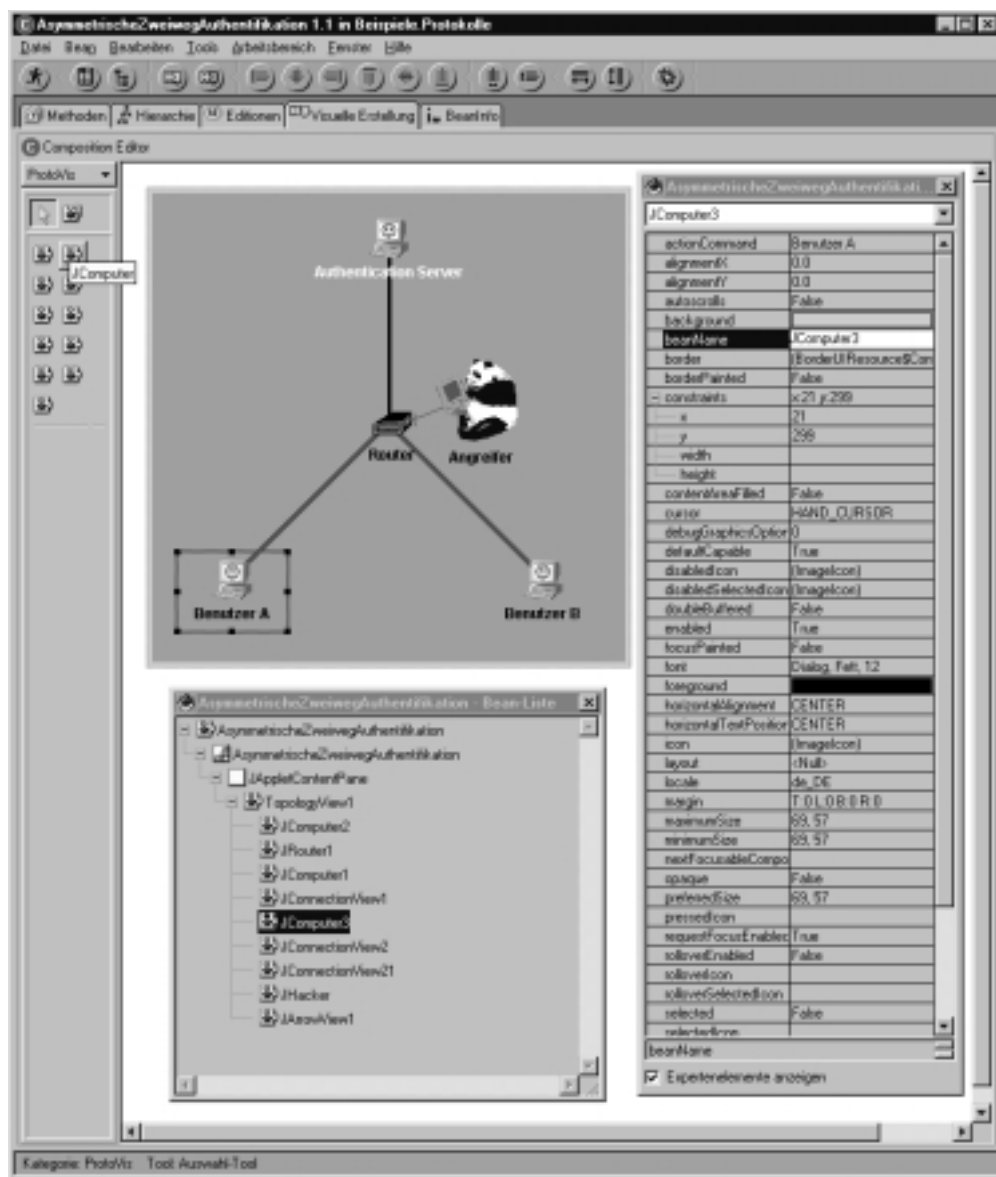


Abbildung 4.17: Erstellung mit Hilfe eines visuellen Entwicklungswerkzeugs

Abbildung 4.18 zeigt einen Ausschnitt der *init()*-Methode. In diesem Ausschnitt wird demonstriert, wie das *TopologyView1*-Objekt (ein Objekt der Klasse *JGraphArea*) der Benutzungsschnittstelle hinzugefügt und die Benutzungsschnittstelle konfiguriert wird.

```
// User Interface festlegen

// UI erzeugen
UserInterface aUI = new UserInterface();

// UI konfigurieren
aUI.getJControlBar().getJStartButton().setEnabled(true);
aUI.getJControlBar().getJStopButton().setEnabled(false);
aUI.getJControlBar().getJStepBackwardButton().setEnabled(false);
aUI.getJControlBar().getJStepForwardButton().setEnabled(false);
aUI.getJHistoryView().addConnectableObject(node1, "Benutzer A");
aUI.getJHistoryView().addConnectableObject(node2, "Benutzer B");
aUI.getJHistoryView().addConnectableObject(node3, "Authentication Server");
aUI.getJHistoryView().addConnectableObject(nodeR, "hacked Router");

// Topologiebereich hinzufügen
aUI.addJGraphArea(getTopologyView1());
```

Abbildung 4.18: Benutzungsschnittstelle erzeugen und festlegen

VisualAge für Java legt für jede Visualisierungskomponente eine zugehörige Methode mit einem ähnlichen Namen an. Beim ersten Aufruf dieser Methode wird die Visualisierungskomponente erzeugt. Ansonsten gibt diese Methode eine Referenz auf die existierende Visualisierungskomponente zurück. Deshalb wird beim Hinzufügen des *TopologyView1*-Objekts (ein Objekt der Klasse *JGraphArea*) die Methode *getTopologyView1()* aufgerufen.

Aus dem selben Grund werden die Visualisierungskomponenten für den

Graphenbereich in der *init*-Methode durch Aufruf der von *VisualAge* erstellten Methoden erzeugt. Gleichzeitig werden den Visualisierungskomponenten zugehörige Models mit Hilfe der im vorigen Kapitel entworfenen *Schnittstellen* zugewiesen. Abbildung 4.19 zeigt, wie Teile des Simulationsmodells mit den Visualisierungskomponenten gekoppelt werden.

```
// Nodes & Connections
getJComputer1().setModel(node1);
getJComputer2().setModel(node2);
getJAS().setModel(node3);
getJRouter1().setModel(nodeR);
getJConnectionView1().setModel(conn1R,connR1);
getJConnectionView2().setModel(connR2,conn2R);
getJConnectionView3().setModel(conn3R,connR3);

//InternalFrames
getTopologyView1().add(getJInternalFrame1(),0);
getTopologyView1().add(getJInternalFrame2(),0);
getTopologyView1().add(getJInternalFrame3(),0);
getTopologyView1().add(getJInternalFrame4(),0);

// set Models in Protocols to JInternalFrames
getJTable().setModel(anAS.getTableModel());
getJInternalFrameContentPanel().add(new JTextArea ( cli-
ent1.getTextAreaModel()), "Center");
getJInternalFrameContentPane2().add(new JTextArea ( cli-
ent2.getTextAreaModel()), "Center");
getJInternalFrameContentPane3().add(new JTextArea(anAS.get-
TextAreaModel()), "Center");
```

Abbildung 4.19: Visualisierungskomponenten im Graphenbereich

Die in dem Anwendungsbeispiel verwendeten Protokolle erzeugen einige Models, die von den Swing-Komponenten zur Verfügung gestellt werden. Diese Models werden, in Abbildung 4.19 im letzten Bereich ersichtlich, den *InternalFrames* übergeben. Dieser Teil wurde speziell für das Anwendungsbeispiel entwickelt und war vorher nicht im neuem Baukasten vorhanden. Deshalb wird auf die für das Anwendungsbeispiel benötigte Erstellung von *InternalFrames* im folgendem Abschnitt genauer eingegangen.

4.2.3.2 Interne Fenster

Für das Anwendungsbeispiel wurden drei verschiedene *interne Fenster* erstellt. Diese Fenster geben Auskunft zu den jeweils zugeordneten Protokollen und bieten vereinzelt Interaktionsmöglichkeiten für den Benutzer an.

Authentifikationsserver Fenster

Von diesem Fenster wird der Benutzer über den Zustand des Authentifikationsservers informiert. Ein Eingreifen in das Protokoll selbst wird dem Benutzer in diesem Fenster nicht ermöglicht.

Dieses Fenster gliedert sich, wie Abbildung 4.20 zeigt, in drei Anzeigebereiche. Der obere Bereich zeigt tabellarisch die vom Authentifikationsserver verwalteten Public Keys an, der mittlere Bereich die von Protokoll ausgegebenen Texte und der untere Bereich den verwendeten Secret Key.

Abbildung 4.20: Authentifikationsserver Fenster

Authentifikationsclient Fenster

Das Authentifikationsclient Fenster dient ebenfalls der reinen Information über den Zustand des zugehörigen Client-Protokolls. In diesem Fenster existieren zwei Anzeigebereiche. Der obere Bereich ist für die Textausgabe und der untere für die Anzeige wichtiger Variablen des Protokolls zuständig. Im unteren Bereich werden die Variablen Zufallszahl, Public Key des Authen-

fifikationsservers, Public Key des Initiators bzw. Empfängers und der eigene verwendete Secret Key dargestellt. Abbildung 4.21 zeigt das Authentifikationsclient Fenster des Benutzers B.

Abbildung 4.21: Authentifikationsclient Fenster

Angreifer Fenster

Der Benutzer des Anwendungsbeispiels übernimmt die Rolle des Angreifers. Deshalb bietet das Angreifer Fenster (Abbildung 4.22) dem Benutzer

Abbildung 4.22: Angreifer Fenster

die Möglichkeit, Interaktionen durchzuführen. Er kann bestimmen, wann eine Nachricht abgefangen und wann diese Nachricht an wen wieder versendet werden soll. Dabei kann er diese Nachricht beliebig oft manipulieren. Der Textbereich, der die abgefangene Nachricht anzeigt, kann durch Tastatureingabe bearbeitet werden. Zudem kann auf diesen Text jederzeit der asymmetrische Verschlüsselungsalgorithmus mit einem vom Benutzer ausgewählten Schlüssel beliebig oft angewendet werden. Der Benutzer hat dabei die Möglichkeit, einen selbst definierten Schlüssel anzugeben. Dies sind die Mittel, die dem Benutzer ermöglichen, im Anwendungsbeispiel das ablaufende Protokoll anzugreifen.

4.2.3.3 ActionListener

Nachdem alle *internen Fenster* erstellt wurden, müssen noch die vom Benutzer erzeugten Aktionen verarbeitet werden. Das bedeutet, daß jeder *JButton* (Visualisierungskomponenten zur Visualisierung von Knoten sind inbegriffen) dem *ActionListener* hinzugefügt werden muß, um auf Interaktionen des Benutzers reagieren zu können.

Der Ausführungscode für die einzelnen Aktionen des Benutzer muß dabei in der Methode *actionPerformed* eingetragen werden. Diese Methode wird mit jedem Klick des Benutzers auf einen *JButton* aufgerufen.

Beim Aufruf dieser Methode wird ein Objekt vom *ActionListener* übergeben. Dieses übergebene Objekt löste den Event aus, somit kann festgestellt werden, welcher *JButton* gedrückt wurde.

Zur Veranschaulichung wird in Abbildung 4.23 ein kleiner Ausschnitt aus der *ActionPerformed* Methode gezeigt. Der Programmcode in diesem Ausschnitt öffnet ein Authentifikationsclient Fenster, sobald der Benutzer auf die Visualisierungskomponente *JComputer1* drückt.

```
ActionPerformed(ActionEvent evt)
{
...

// InternalFrame "Benutzer A" öffnen
if(evt.getSource()==getJComputer1())
{
    if (getJInternalFrame1().isIcon())
    {
        try
        {
            getJInternalFrame1().setIcon(false);
        } catch (java.beans.PropertyVetoException vetoExc)
        {
        }
    }else
    {
        getTopologyView1().add(getJInternalFrame1(), getJInternalFrame1().getName(), 0);
        try
        {
            getJInternalFrame1().setSelected(true);
        } catch (java.beans.PropertyVetoException vetoExc)
        {
        }
    }
    getJInternalFrame1().setVisible(true);
}
// Ende: InternalFrame "Benutzer A" öffnen

...
}
```

Abbildung 4.23: Ausschnitt aus der Methode `actionPerformed`

Die wesentlichen Teile des Anwendungsbeispiels wurden nun beschrieben. Im Anhang zeigt Abbildung A.10 das vollständige Applet während eines Durchlaufs des asymmetrischen Zweiweg-Authentifikationsprotokolls. Der Schnappschuß erfolgte, nachdem der Authentifikationsservers die Nachricht M5 erstellt und versendet hatte.

5 Zusammenfassung und Ausblick

In dieser Studienarbeit wurde untersucht, welche aktuellen Anforderungen an den Visualisierungsbaukasten gestellt werden und welche Teile des Baukastens daher überarbeitet werden müssen.

Wegen der rasanten Entwicklung der Programmiersprache Java und der neu hinzugekommenen Anforderungen mußte der Baukasten komplett neu erstellt werden. Der neue Baukasten wurde nach dem Klassenbibliothek-Konzept realisiert.

Der Focus dieses Neuentwurfs richtet sich hauptsächlich auf die Erweiterung, Simulationsmodelle zu erzeugen. Die entworfenen *Klassen zur Visualisierung von Simulationsmodellen* stellen nur eine Übergangslösung dar, weil eine spätere Erweiterung des neuen Baukastens auf ein Framework einen Neuentwurf dieser Klassen erfordert.

Die Systemvoraussetzung des neuen Baukastens ist eine Java Runtime Environment 1.1.5 mit Swing 1.1 oder höher. Durch den Einsatz der Swing-Komponenten wird die Visualisierung der Simulationsmodelle in vielen Varianten ermöglicht. Zusätzlich wurden weitere Klassen zur Visualisierung so entworfen, daß diese in ein Entwicklungswerkzeug als Java-Beans importiert werden können.

Zum Schluß wurden die Grundprinzipien zur Anwendung des neuen Baukastens erläutert und anhand des asymmetrischen Zweiweg-Authentifikationsprotokoll-Beispiels veranschaulicht.

Ausblick

Der neue Baukasten ermöglicht Applets bzw. Applikationen zu erzeugen, die mittels eines Simulationsmodells Kommunikationsprotokolle visualisieren. Weil jedoch einige Bereiche des Baukastens nicht implementiert wurden, ist der Funktionsumfang noch stark eingeschränkt. Dadurch existieren für den Baukasten eine Menge an Erweiterungsmöglichkeiten, die eine gute

Grundlage für nachfolgende Arbeiten bieten. Es werden kurz einige Erweiterungsvorschläge aufgeführt.

Um Simulationsmodelle mittels einer Spezifikationssprache (z.B. SDL) aufbauen zu können, ist es erforderlich, die existierenden *Generatoren zur Erzeugung von Simulationsmodellen* auf die im neuen Baukasten zur Verfügung gestellten und lauffähigen *Klassen zur Erzeugung von Simulationsmodellen* anzupassen.

Damit dem Benutzer weitere Interaktionsmöglichkeiten vom Baukasten zur Verfügung gestellt werden können, ist eine Erweiterung des Baukastens zur Unterstützung der *allgemeinen Steuerung* notwendig. Vor allem wäre ein Entwurf und die zugehörige Implementierung einer Undo–Redo Funktionalität mit Hilfe des Entwurfsmusters *Command* eine interessante Aufgabe.

Eine weitere vermißte Funktionalität, ist die Möglichkeit, Knoten, Verbindungen und Protokolle bzw. Algorithmen zur Laufzeit zurückzusetzen. Diese Funktionalität ist zwar berücksichtigt, jedoch noch nicht umgesetzt.

Der Höhepunkt des HiSAP Projektes wäre die Entwicklung eines Frameworks. Der neue Baukasten ermöglicht, Erweiterungen auf ein Framework durchzuführen.

Generell wächst die Mächtigkeit des momentan existierenden Baukastens, wenn Protokolle, Algorithmen und *Klassen zur Visualisierung von Simulationsmodellen* in den Baukasten integriert werden, die zusätzlich zu den im Baukasten erstellten Protokollbeispielen entwickelt wurden.

Anhang

Kurzbeschreibung der Entwurfsmuster

"**Beobachtermuster**

Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so daß die Änderung des Zustands eines Objekts dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Brückenmuster

Entkopple eine Abstraktion von ihrer Implementierung, so daß beide unabhängig voneinander variiert werden können.

Dekorierermuster

Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.

Fabrikmethodenmuster

Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welscher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

Kompositionsmuster

Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.

Prototypenmuster

Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars, und erzeuge neue Objekte durch Kopieren dieses Prototypen.

Strategiemuster

Definiere eine Familie von Algorithmen, kapsle jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihm nutzenden Klienten zu variieren." [GHJV96]

HISAP ProtoVisFoundationClasses Version 1.0

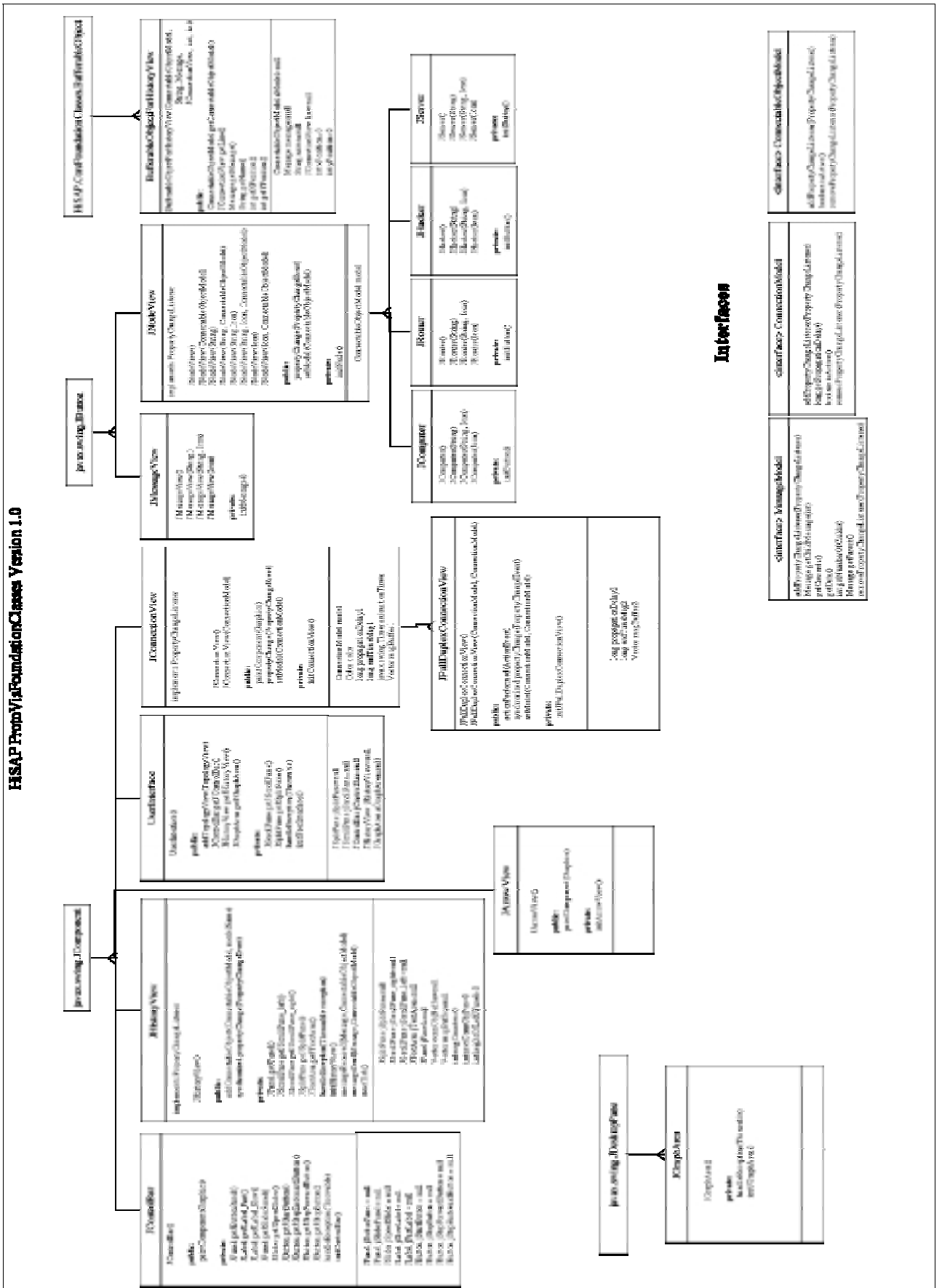


Abbildung A.2: ProtoVisFoundationClasses

Aufbau der im Anwendungsbeispiel verwendeten Nachrichten

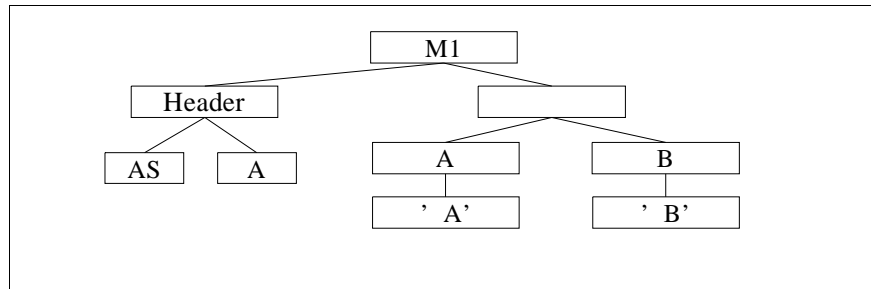


Abbildung A.3: Nachricht (M1)

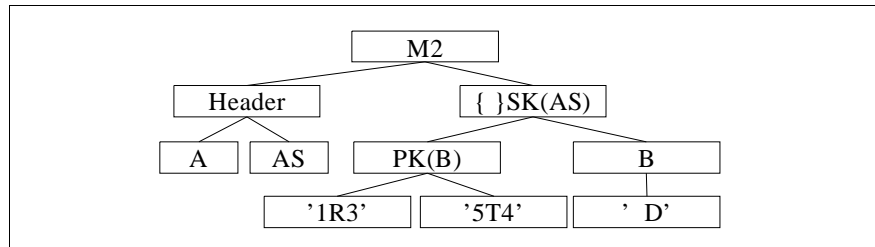


Abbildung A.4: Nachricht (M2)

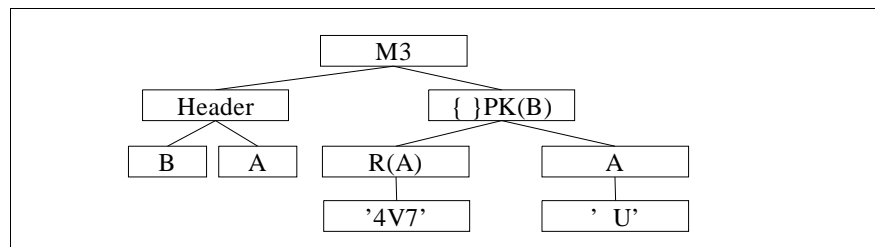


Abbildung A.5: Nachricht (M3)

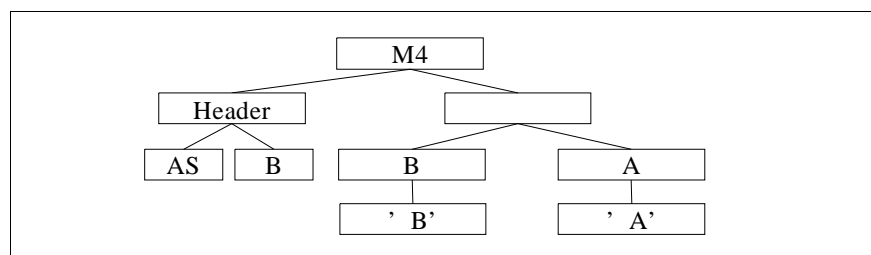


Abbildung A.6: Nachricht (M4)

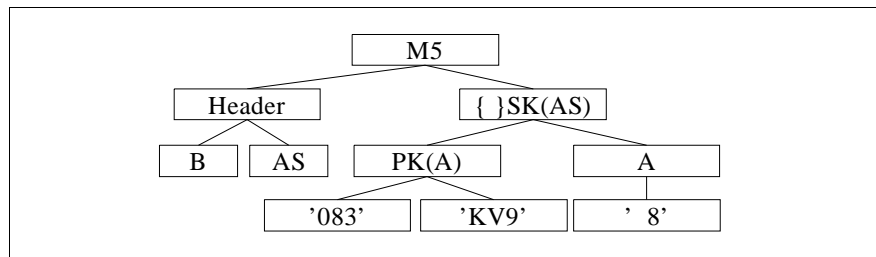


Abbildung A.7: Nachricht (M5)

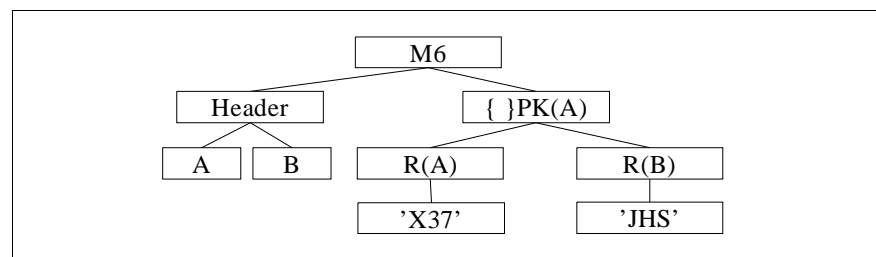


Abbildung A.8: Nachricht (M6)

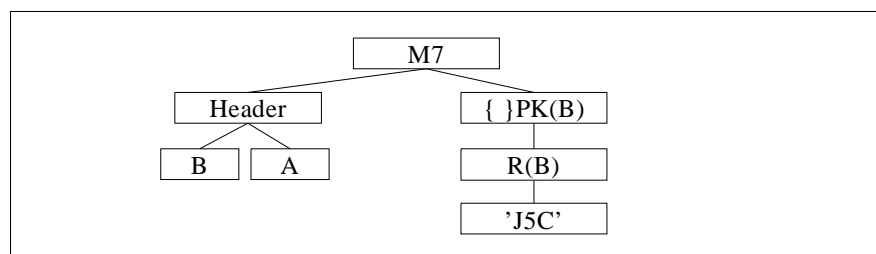


Abbildung A.9: Nachricht (M7)

Applet Viewer: Beispiele.Protokolle.AsymZweiwegAuthentifikation.class

```

(57) abstract M1: a, b J, data[as a a b]
(58) abstract M1: a, b J, data[as a a b]
(59) abstract M2: {PK(b), b}SK(as) J, data[ a as dd" :]
(60) abstract M2: {PK(b), b}SK(as) J, data[ a as dd" :]
(61) abstract M3: {R(a), a}PK(b) J, data[ b advb N]
(62) abstract M3: {R(a), a}PK(b) J, data[ b advb N]
(63) abstract M4: b, a J, data[as b b a]
(64) abstract M4: b, a J, data[as b b a]
(65) abstract M5: {PK(a), a}SK(as) J, data[ b as dG "d J

```

AS			
Benutzer	PK	r	
a	17	91	
b	5	119	

PK(b)-Anfrage von a empfangen.
Erzeuge Nachricht M2: abstract{ PK(b), b J, data[5 119 b]
M2 mit SK(as) verschlüsselt: data[dd" :]
Versende M2 an a.
PK(a)-Anfrage von b empfangen.
Erzeuge Nachricht M5: abstract{ PK(a), a J, data[17 91 a]
M5 mit SK(as) verschlüsselt: data[dG "d J]
Versende M5 an b.

Secret Key von AS = (key: 11, r:119)

Benutzer A

Statusfenster

Erzeuge Nachricht M1: abstract{ a, b J, data[a b].
Versende M1 an AS.
Erwarte M2 als nächste Nachricht: abstract{ PK(b), b}SK(as) J.
Nachricht ?, data[dd" :] empfangen.
Entschlüsselte Nachricht mit PK(as).
Entschlüsselte Nachricht: data[5 119 b].
Nachricht als M2 akzeptiert.
Erzeuge Nachricht M3: abstract{ R(a), a}PK(b) J, data[123 a].
M3 mit PK(b) verschlüsselt: data[advb N].
Versende M3 an B.
Erwarte M6 als nächste Nachricht: abstract{ R(a), R(b)}PK(a) J.

Zufallszahl von Benutzer A = 123
Public Key des Authentication Servers = (key:35, r:119)
Public Key von Benutzer B = (key: 5, r:119)
Secret Key von Benutzer A = (key:305, r: 91)

Benutzer B

Statusfenster

Nachricht ?, data[advb N], empfangen.
Entschlüsselte Nachricht mit SK(b).
Entschlüsselte Nachricht: data[123 a].
Nachricht als M3 akzeptiert.
Erzeuge Nachricht M4: abstract{ b, a J, data[b a].
Versende M4 an AS.
Erwarte M5 als nächste Nachricht: abstract{ PK(a), a}SK(as) J.

Zufallszahl von Benutzer B = 321
Public Key des Authentication Servers = (key: 35, r:119)
Public Key von Benutzer A = (key: 17, r: 91)
Secret Key von Benutzer B = (key: 77, r:119)

Applet started.

langsam 0 25 50 75 100 schnell

Abbildung A.10: Asymmetrisches Zweiweg-Authentifikationsprotokoll-Applet

Literaturverzeichnis

- [Baa97] Oliver Baar, *Visualisierung von Protokollen am Beispiel des Authentifizierungsprotokolls von Needham und Schroeder*, Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1648, 1997.
- [Baa98] Oliver Baar, *Visualisierung von Protokollen: Ein erweitertes Petri Netz-Modell zur Simulation von Kommunikationsprotokollen*, Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 1644, 1998.
- [BRM98] Cora Burger, Kurt Rothermel, Rolf Mecklenburg, *Interactive Protocol Simulation Applets for Distance Education*, Abt. Verteilte Systeme, Fakultät Informatik, Universität Stuttgart, 1998.
- [Den97] Ulrich Denneler, *Visualisierung von Protokollen am Beispiel des Protokolls für Secure Electronic Transaction (SET)*, Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1650, 1997.
- [Eng93] Hermann Engesser, Hrsg. *Duden Informatik*. Dudenverlag, Mannheim, zweite Auflage, 1993.
- [Gez97] Tolga Gezgin, *Visualisierung von Protokollen am Beispiel des Protokolls für Traceable Anonymous Cash*, Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1608, 1997.
- [GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley (Deutschland) GmbH, 1. Auflage 1996.
- [Gor98] Ralph Gorth. *Erweiterung des Java-Baukastens zur Visualisierung von Protokollen: Erweiterung des Baukastens zur Visualisierung von SDL-Konstrukten*. Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1671, 1998.
- [Her87] *Herders Grossses Handlexikon*, Verlag Herder Freiburg, Breisgau, 1987.
- [HiS99] *HiSAP Projekt des IPVR*, HiSAP-Homepage: <http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/ProroVis/protovis.html>, Universität Stuttgart, Fakultät Informatik, 1999.
- [Kai97] Michael Kaiser, *Visualisierung von Protokollen am Beispiel des (simplifizierten) Kerberos Authentifikationsprotokolls*, Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1649, 1997.
- [Kai98] Michael Kaiser, *Bewertung des Simulationsmodells des HiSAP-Werkzeugs zur interaktiven Simulation von Protokollen*. Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 1668, 1998.
- [Kog97] Edgar Kogel. *Visualisierung von Kommunikationsprotokollen mit Java*. Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 1522, 1997.
- [May98] Markus Mayer. *Konfiguration und interaktive Animation von Protokollen verteilter Systeme mit Hilfe des ProtoVis-Generators für Protokollmaschinen und des Visualisierungsbaukastens*. Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 1605, 1998.
- [Pap98] Konstantinos Papoulidis. *Erweiterung des Java-Baukastens zur Visualisierung von Protokollen: Weiterentwicklung der Benutzerschnittstelle, Undo-Funktionalität*. Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1670, 1998.

- [Rau98] Jürgen Rau. *Erweiterung des Java–Baukastens zur Visualisierung von Protokollen: Erweiterung für die Visualisierung von Endlichen Automaten*. Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1715, 1998.
- [Rot98] Prof. Dr. Kurt Rothermel, *Grundlagen der Verteilten Systeme*, Universität Stuttgart, Fakultät Informatik, Lehrstuhl »Verteilte Systeme«, Institut für Parallele und Verteilte Höchstleistungsrechner, Vorlesungsskript, SS 1998.
- [Sch97] Peter W. Schurr. *Visualisierung von Lehrinhalten mittels Java Applets*. Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 1608, 1997.
- [Sta94] Christian Stary. *Interaktive Systeme*. Friedr. Vieweg & Sohn Verlagsgesellschaft mgH, Braunschweig/Wiesbaden, 1994.
- [Sun99] Sun, Java–Homepage (<http://www.javasoft.com>), 1999.
- [Tan97] Andrew S. Tanenbaum, *Computernetzwerke*, Prentice Hall Verlag, München, dritte Ausgabe, 1997.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt
und nur die angegebenen Quellen benutzt zu haben.

(Torsten Brodbeck)