A d - h o c C h a t s y s t e m f ü r m o b i l e N e t z e

G r u p p e 3 (A d B e e)

Softwareentwicklungspraktikum
Sommersemester 2007

F e i n e n t w u r f

Auftraggeber:
Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Prof. Dr.-Ing. Lars Wolf
Mühlenpfordtstraße 23, 1. OG
38106 Braunschweig
Deutschland

Betreuer: Sven Lahde, Oliver Wellnitz
Hiwi: Wolf-Bastian Pöttner

Auftragnehmer: Gruppe 3(AdBee)

Name	E - Mail
Ekrem Özmen	e.oezmen@gmail.com
Celal Özyalcin	c.oezyalcin@tu-bs.de
Thorben Schulze	thorben.schulze@tu-bs.de
Danny Melching	danny.melching@tu-bs.de

Phasenverantwortlicher: Celal Özyalcin

Braunschweig, 14.05.2007

Versionsübersicht

Version	Datum	Autor	Status	Kommentar
1.0	14.05.07	Ekrem, Celal, Danny, Thorben		Erste Version des Feinentwurfs
1.5 1.6	18.06.07, 02.07.07, 12.07.07	Ekrem, Celal, Danny, Thorben		Klassendiagramm zu Nachricht aktualisiert. Klassendiagramm zu Routing aktualisiert Methodenbeschreibungen zu Nachricht aktualisiert. Methodenbeschreibungen zu Routing aktualisiert. Klassendiagramm zu Senden/Empfangen aktualisiert Methodenbeschreibungen zu Senden/Empfangen aktualisiert. Klassendiagramm zu Sicherheit aktualisiert. Methodenbeschreibungen zu Sicherheit aktualisiert.
1.7	15.07.07	Danny		Aktualisierung der Verwaltungskomponente Abschnitt 3.3 inklusive Unterpunkte.

I n h a l t s v e r z e i c h n i s

A b b i l d u n g s v e r z e i c h n i s

1 Einleitung

Das Dokument gliedert sich in Erfüllung der Kriterien, Implementierungsentwurf und Datenmodell. Im Punkt Erfüllung der Kriterien, sind die, im Pflichtenheft, ausgearbeiteten Muss-, Wunsch- und Abgrenzungskriterien aufgelistet und beschrieben. Im Punkt Implementierungsentwurf findet sich das Komponentendiagramm aus dem Grobentwurf wieder. Hier werden die einzelnen Komponenten im Detail beschrieben, mithilfe eines Klassendiagramms und Auflistung und Beschreibung der Klassen und Methoden der einzelnen Komponenten. Im Punkt Datenmodell sind kurz die langfristig zu speichernden Daten beschrieben.

Ziel dieses Dokumentes ist, die Implementierung des Projekt AdBee hinreichend genau zu beschreiben, so dass es jedem Softwareentwickler möglich ist, das Produkt zu entwickeln.

2 Erfüllung der Kriterien

2.1 Musskriterien

Die folgenden Kriterien sind unabdingbar und müssen durch das Produkt erfüllt werden:

/M10/ Benutzer können Textnachrichten sowie Binärdaten miteinander austauschen

/M20/ Der Austausch von Nachrichten erfolgt in Kommunikationskanälen

/M30/ Kommunikationskanal befinden sich Benutzer, wobei jede gesendete Nachricht von allen Benutzern im jeweiligen Kanal empfangen wird

/M40/ Jeder Kommunikationskanal besitzt einen Namen und eine eindeutige Identifizierung

/M50/ Die Kanäle sind in ihrer Anzahl nicht begrenzt

/M60/ Man kann sich eine Liste aller existierenden Kanäle anzeigen lassen

/M70/ Die Kanäle öffentlich oder geschlossen

/M80/ Jeder kann den öffentlichen Kanälen beitreten

/M90/ In den geschlossenen Kanälen muss man von einem Benutzer, der sich schon in diesem Kanal befindet, eingeladen werden

/M100/ Es muss den Benutzern des Chatsystems möglich sein, einen Kommunikationskanal zu erstellen, in diese beitreten, sie wieder verlassen und jemand anderen in einen geschlossenen Kanal einladen zu können

/M110/ Jeder Benutzer befindet sich zum Start des Programms in einem anonymen Kanal

/M120/ Der anonyme Kanal besitzt den eindeutigen Namen „Anonymous“, welcher von den anderen Kanälen nicht verwendet werden darf

/M130/ Die Übertragung in dem anonymen und den geschlossenen Kanälen erfolgt verschlüsselt

/M140/ Jeder Benutzer besitzt ein RSA-Schlüsselpaar

/M150/ Jeder geschlossene Kanal besitzt einen gemeinsamen Schlüssel

/M160/ Das RSA-Schlüsselpaar ermöglicht den sicheren Austausch des gemeinsamen Schlüssels

/M170/ Das RSA-Schlüsselpaar ermöglich die Verschleierung der Herkunft der Nachricht in den anonymen Kanälen

/M180/ Die Benutzer des Chatsystems besitzen einen Namen und müssen eindeutig identifiziert werden können

/M190/ Jeder Benutzer kann sich in mehren Kommunikationskanälen befinden

/M200/ Das RSA-Schlüsselpaar besteht aus dem öffentlichen Zertifikat, welches vor Benutzung des Programms vergeben wird, und einem privaten Schlüssel, der zum entschlüsseln und zum signieren von Nachrichten vorgesehen ist.

/M210/ Die Nachrichten besitzen eine eindeutige Identifizierung

/M220/ Die Nachrichten müssen, außer in dem anonymen Kanal, einem Sender durch Authentifizierung eindeutig zugeordnet werden können

/M230/ Es muss möglich sein, das Zertifikat des RSA-Schlüsselpaares anzufordern und zu versenden

/M240/ Ein Nachrichtenaustausch des gemeinsamen Schlüssels für die geschlossenen Kanäle muss möglich sein

/M250/ Sobald eine Nachricht verschickt wird, findet sie automatisch ihren Weg durch das Netz

/M260/ Jeder Knoten sendet eine Empfangsbestätigung an den vorherigen Knoten

/M270/ Jede Nachricht besitzt eine TTL, die angibt, wie lange sie noch im Netz verbleiben darf, bevor sie aus dem Netz entfernt wird und dem Sender eine negative Empfangsbestätigung mit dem Inhalt der Nachricht gesendet wird

/M280/ Eine nicht angekommene Nachricht wird gespeichert und später erneut gesendet

/M290/ Falls eine Nachricht endgültig die Empfänger nicht erreicht, wird dies dem Sender mitgeteilt

/M300/ Da diese Nachrichten verzögert gesendet werden und die Übertragung verbindunglos mittels UDP-Datagrammen stattfindet, muss die richtige Reihenfolge am Empfänger gewährleistet werden

/M310/ Welche Kriterien beim versenden von Nachrichten genau beachtet werden müssen, findet man in der Protokollspezifikation[1].

/M320/ Das Chatprogramm muss interoperabel mit den Chatprogrammen der anderen Projektgruppen sein

/M330/ Aufgrund der Mobilität der Teilnehmer soll eine ständige Aktualisierung der Netzstruktur, durch periodenweises Schicken von Hello-Nachrichten und Routing-Nachrichten, erfolgen

/M340/ Es sollen nicht nur einzelne Benutzer mit in eine Netzstruktur aufgenommen werden können, sondern auch eine Verschmelzung von zwei oder mehr Netzstrukturen möglich sein.

/M350/ Öffentliche Kanäle mit gleichen Namen werden zu einem Gemeinsamen zusammengeschlossen

/M360/ Geschlossene Kanäle müssen weiterhin getrennt behandelt werden

/M370/ Zum Testen der Software gibt es eine Peerverwaltung die ein- bzw. ausgeschaltet werden kann(Infrastrukturmodus)

/M380/ Das Chatprogramm muss auf den Betriebssystemen Mac OS X und Linux lauffähig sein

/M390/ Die graphische Oberfläche sollte bedienerfreundlich gestaltet sein

2.2 Wunschkriterien

Die Erfüllung folgender Kriterien für das abzugebende Produkt wird angestrebt:

/W10/ Jeder Benutzer kann eine Liste aller gerade im Netz vorhandenen Benutzer anzeigen lassen

/W20/ Neue Funktionen, die die Handhabung erleichtern oder erweitern, sollten sich möglichst einfach einbinden lassen

/W30/ Eine Portierung auf andere Betriebssysteme sollte auch ohne größere Änderungen erfolgen können

/W40/ Die Benutzeroberfläche sollte vom jeweiligen Benutzer in sinnvollem maß veränderbar sein

2.3 Abgrenzungskriterien

Folgende Funktionalitäten werden nicht durch das Produkt, sondern wie folgt beschrieben anderweitig erfüllt:

/A10/ Das Programm wird nicht mit schon vorhandenen Chatsystemen Interoperabel sein.

/A20/ Die Größe der übertragbaren Dateien ist begrenzt.

3 Implementierungsentwurf

3.1 Gesamtsystem

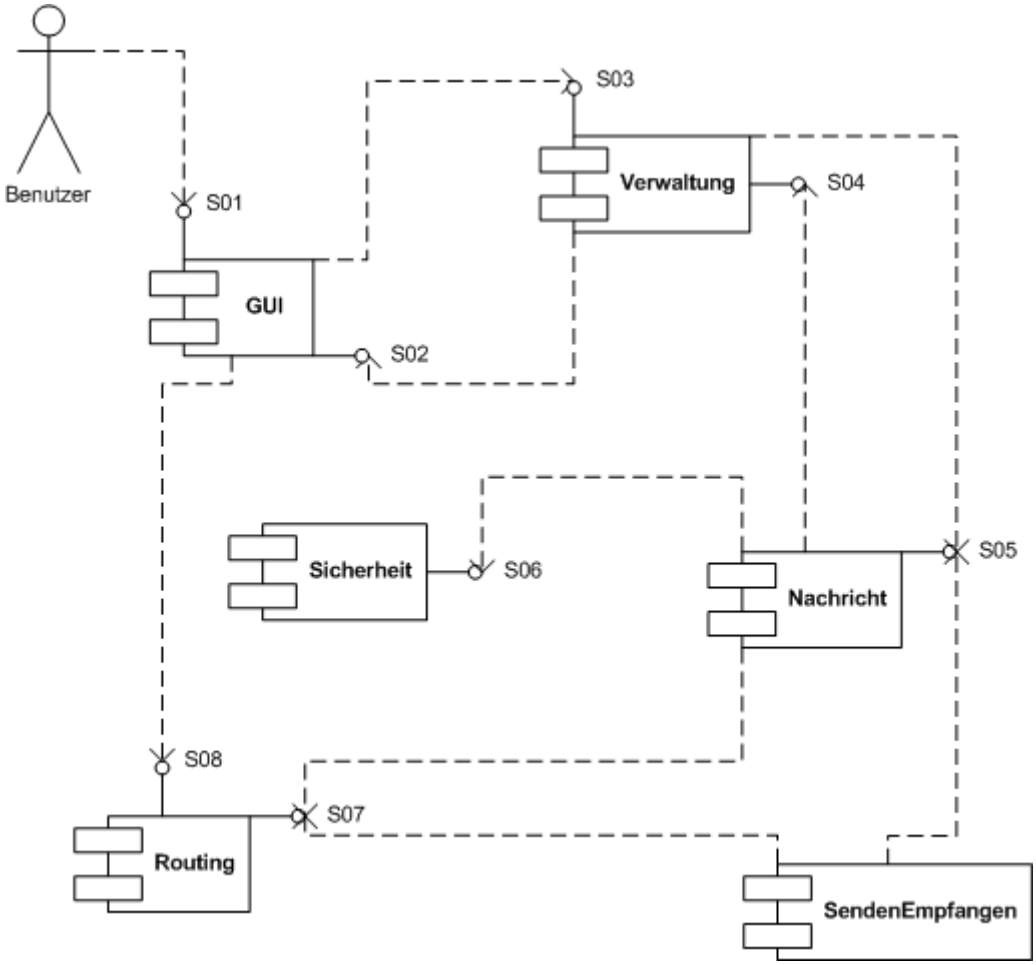

Abbildung 1: Komponentendiagramm

Das Diagramm zeigt die Komponenten die später im Programm miteinander kommunizieren. Der Benutzer steht für den menschlichen Teilnehmer, der mit dem Programm interagiert. Dieser verwendet das Programm nur über die GUI.

Die GUI leitet das weiter, was durch den Benutzer gefordert wird. Sie verwaltet die visuellen Objekte wie Kanalfenster, Eingabefeld, Teilnehmerfeld usw.

Die Verwaltung verwaltet die gesamten Informationen der Benutzer und Kanäle des Programms. Es interpretiert die Informationen, die von der Nachrichtenkomponente kommen und gibt sie in geeigneter Weise an die GUI weiter. Genauso nimmt sie Informationen von der GUI entgegen und gibt sie an die Nachrichtenkomponente weiter.

Die Nachrichtenkomponente kümmert sich um die Verwirklichung des Protokolls. Sie realisiert das Erstellen und parsen von XML-Dokumenten in Nachrichtenobjekten. Dementsprechend werden eintreffende Systemnachrichten behandelt oder an andere Komponente weitergeben. So werden z.B. ChannelNachrichten an die Verwaltung und RoutingNachrichten ans Routing weitergegeben. Die Nachrichten-Komponente besitzt einen

Sende- und einen Empfangspuffer aus denen zu sendende Nachrichten geholt bzw. empfangene Nachrichten hinzugefügt werden. Ebenfalls benutzt es die Sicherheitskomponente zum verschlüsseln, entschlüsseln, signieren, authentifizieren von Nachrichten. In der Nachrichten-Komponente laufen Threads um Periodische Nachrichten zu verschicken, um empfangene Nachrichten zu behandeln und zu sendende Nachrichten vorzubereiten. Zum Vorbereiten gehören die Sicherheitsmethoden, das, falls nötige, Splitten von Nachrichten und raussuchen des nächsten Empfängerknotens aus dem Routing.
In der Sicherheitskomponente sind die Methoden, um Nachrichtenobjekte mit bestimmten Sicherheitsverfahren zu behandeln
Die Senden/Empfangen-Komponente ist die Schnittstelle zum Netz. Über sie kommen die UDP-Datagramme an und über sie werden die Nachrichten mittels UDP versenden. Es nutzt den Dienst der Routingkomponente, um die IP und Port des nächsten Knotens zu bekommen.
Die Routingkomponente verwaltet die Routingtabellen und den Infrastrukturmodus. Die Nachrichten und Senden/Empfangen-Komponente wissen nicht, in welchem Modus sich die Komponente befindet, sondern nutzen nur deren Dienst, damit das Testen verwirklicht werden kann. Über die GUI kann der Infrastrukturmodus ein- und ausgeschaltet werden.

Die Folgenden Klassendiagramme beschreiben den Aufbau des Programms. Construkoren sind nur gesetzt, wenn sie wichtig für die Entwicklung sind. Get- und Set- Methoden sind auch nicht hinzugefügt worden, sollen aber in der Entwicklung vorhanden sein.

3.2 Implementierung von Komponente K01: GUI

Die GUI-Komponente besteht aus einem Hauptfenster(MainWindow), welches auch nach Start angezeigt wird. Andere Funktionen des Programmes, welche das Hinzufügen eines Anhangs, dem Erstellen eines Kanals, dem Einstellen der Fenster und der Behandlung des Infrastrukturmodus sind, werden in speziellen Dialogfenstern ausgeführt. Dabei werden alle Ereignisse, die durch die Fenster ausgelöst werden, durch die Eventhandling-Klasse behandelt und je nach Event bestimmte Funktionen ausgeführt.

Es werden Swing-Komponenten von Java für die GUI-Objekte benutzt.

3.2.1 Klassendiagramm

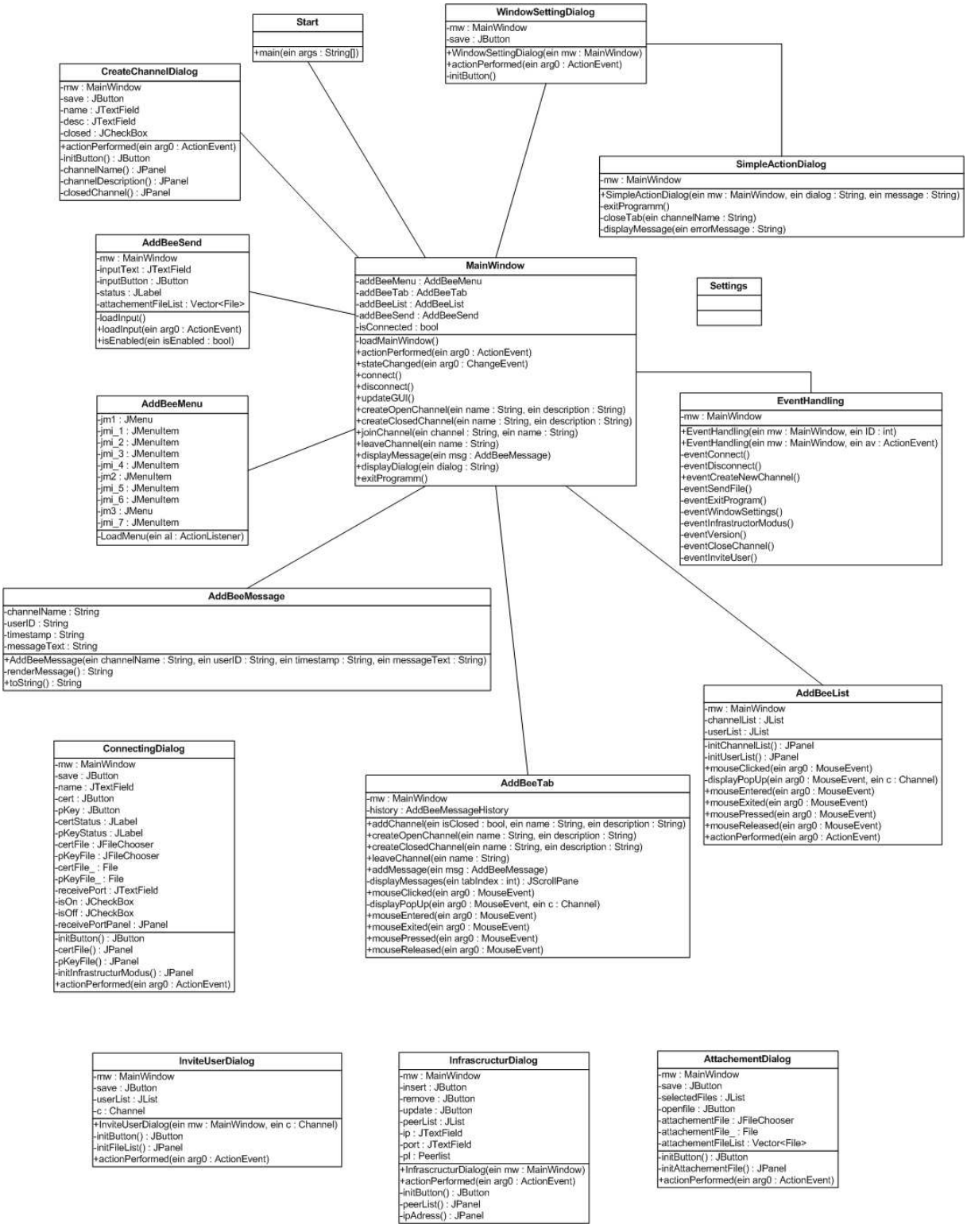

Abbildung 2: Klassendiagramm: GUI

3.2.2 Erläuterung

Klasse	Aufgabe	Attribute und Kommunikationspartner
MainWindow	Das Hauptfenster mit den Kanälen und Anzeigen für Kanäle und Benutzer.	**Attribute:** addBeeMenu : AddBeeMenu addBeeTab : AddBeeTab addBeeList : AddBeeList addBeeSend : AddBeeSend isConnected : bool **Kommunikationspartner:** GUI-, Channel, und restlichen Managment Klassen.Stellt verbindung für alle Klassen zur verfügung.
	ID: MW01 **Name:** loadMainWindow() **Beschreibung:** GUI wird nach den Vorgaben der Settings geladen.	
	ID: MW02 **Name:** actionPerformed (arg0:ActionEvent) **Beschreibung:** Ereignisse werden weitergeleitet und entsprechend neu aufgeteilt. Events von Button, Menu	
	ID: MW03 **Name:**stateChanged (arg0: Change Event) **Beschriebung:** Ereignisse werden weitergeleitet und entsprechend neu aufgeteilt.Events Tabs	
	ID: MW04 **Name:** connect() **Beschreibung:** Verbindes sich ad-hoc mit anderen Usern.	
	ID: MW05 **Name:** disconnect() **Beschreibung:** trennt alle bestehenden Verbindungen (verlässt alle Channels).	
	ID: MW06 **Name:** updateGUI() **Beschreibung:** Aktualisiert den Inhalt der einzelnen Komponenten, je nach aktiven Channel, die Mitglieder, oder wenn neue Channels hinzukommen.	
	ID: MW07 **Name:** createOpenChannel(name:String, Discreption:String) **Beschreibung:** Erstellt einen neuen offenen Channel, der in der GUI im Tab angezeigt wird.	

	ID: MW08 **Name:** createClosedChannel (name:String, Discreption:String) **Beschreibung:** String Erstellt einen geschlossenen Channel, der in der GUI im Tab angezeigt wird.	
	ID: MW09 **Name:** leaveChannel (name:String) **Beschreibung:** Nimmt einen User aus einem Channel in der GUI.	
	ID: MW10 **Name:**displayMessage (msg:AddbeeMessage) **Beschreibung:** Fügt eine Nachricht zum anzeigen in die GUI ein.	
	ID: MW11 **Name:** displayDialog (dialog:String) **Beschreibung:** Öffnen ein Benutzerdefinierten Dialog zum anzeigen, für individuelle Hinweise, die bestätigt werden müssen.	
	ID: MW12 **Name:** exitProgramm() **Beschreibung:** Beendet alle Verbindungen und verlässt das aktive Programm.	
	ID: MW13 **Name:** displayWelcomeToUser(c : Channel) **Beschreibung:** Erstellt eine Begrüßungsmessage auf dem Display	
AddBeeList	AddBeeList - Spezial Komponente zum anzeigen, von Channels und Usern	**Attribute:** mw : Mainwindow channelList : JList userlist : JList **Kommunikationspartner:** Mainwindow
	ID: AdList01 **Name:** initChannelList() **Beschreibung:** Erstellt eine neue JList und setzt oberhalb des Panels ein Label ein.	

	ID: AdList02 **Name:** initUserList () **Beschreibung:** Erstellt eine neue JList und setzt oberhalb des Panels ein Label ein.	
	ID: AdList03 **Name:** mouseClicked (arg0 : MouseEvent) **Beschreibung:** Behandlung des MouseEvent	
	ID: AdList04 **Name:** displayPopUp(arg0:MouseEvent, c:Channel) **Beschreibung:** Erstellt eine neue JPopupMenu	
	ID: AdList05 **Name:**actionPerformed(arg0:ActionEvent) **Beschreibung:** Join User in open Channel event.	
AddBeeMenu	Fügt dem Frame oberhalb eine Menüleiste hinzu, die mithilfe von Tastaturkürzeln, zusätzlich gesteuert werden kann.	**Attribute:** Jm1 : JMenu JMI_1 : JMenuItem Jmi_2 : JMenuItem Jmi_3 : JMenuItem Jmi_4 : JMenuItem Jm2 : JMenu JMI_5 : JMenuItem JMI_6 : JMenuItem Jm3 : JMenu JMI_7: JMenuItem **Kommunikationspartner:** Mainwindow
	ID: AdMenu01 **Name:** LoadMenu(al : ActionListener) **Beschreibung:** Ladet das Menü, mit den Items, die seberate Steuerungen haben**.**	

AddBeeMessage	Erweiterter Datentyp zum anzeigen einer Textnachricht in der Komponente	**Attribute:** channelName : String userID : String timestamp : String messageText : String **Kommunikationspartner:** Mainwindow
	ID: AdMessage01 **Name:** renderMessage() **Beschreibung:** Formatiert den Inhalt der Textnachricht mit User Namen und der Versandzeit in HTML um.	
	ID: AdMessage02 **Name:** toString() **Beschreibung:** formatierte HTML Textnachricht mit zusätzlichen Daten	
AddBeeSend	Erweiterter Datentyp mit Eingabe Textbox und Versendebutton und Label für Statusanzeige.	**Attribute:** Mw : MainWindow inputText : JTextField inputButton :Jbutton status : JLabel attachmentFileList : Vector<File> **Kommunikationspartner:** Mainwindow
	ID : AdSend01 **Name:** loadInput() **Beschreibung:** Laden der einzelnen Komponente innerhalb des Panels.	
	ID : AdSend02 **Name: a**ctionPerformed(arg0:ActionEvent) **Beschreibung:** Behandlung des Events, nach Eingabe des Users	
	ID : AdSend03 **Name:** isEnabled(isEnabled : boolean) **Beschreibung:** Freigeben der Kontrollen, wenn eine Verbindung zu mind. ein Channel besteht.	

AddBeeMessage Tab	Spezial Komponente zum anzeigen, von Channels und Textnachriten	**Attribute:** Mw : MainWindow History : AddBeeMessageHistory **Kommunikationspartner:** Mainwindow
	ID : AdMesTab01 **Name:** addChannel(isClosed: boolean, name: String, description: String) **Beschreibung:** Fügt dem TabbedPane einen neuen Tab ein, der zusätzlich Namen und eine Beschreibung erhält.	
	ID : AdMesTab02 **Name:** createOpenChannel(name:String, description: String) **Beschreibung:** Erstellt einen offenen Channel.	
	ID : AdMesTab03 **Name:**createClosedChannel(name:String, description: String) **Beschreibung:** Erstellt einen geschlossenen Channel.	
	ID : AdMesTab04 **Name:** leaveChannel(name: String) **Beschreibung:** Verlässt den Channel in der GUI.	
	ID : AdMesTab05 **Name:** addMessage(msg : AddBeeMessage) **Beschreibung:** Fügt eine Nachricht in ein Channel ein.	
	ID : AdMesTab06 **Name:** displayMessages(int : tabIndex) **Beschreibung:** Gibt alle Nachrichten für ein Channel frei und bindet alle im entsprechendem Channel geschriebenen Nachricht ein.	
	ID : AdMesTab07 **Name:** scrollToBottom(name : String) **Beschreibung:** Scroll die JScrollPane nach unten.	

	ID: AdMesTab07 **Name:** mouseClicked (arg0 : MouseEvent) **Beschreibung:** Behandlung des MouseEvent	
AttachementDialog	AttachementDialog - Datenversand	**Attribute:** Mw : MainWindow Save : JButton selectedFiles : JList openfile : JButton attachmentFile : JFIleChooser attachmentFile_ : File attachmentFileList : Vector<File> **Kommunikationspartner:** Mainwindow
	ID: AttDia01 **Name:** initButton() **Beschreibung:** Lädt des Save Buttons	
	ID: AttDia02 **Name:** initAttachementFile() **Beschreibung:** FileChooser für den Zertifikat mit offenen Schlüssel.	
	ID : AttDia03 **Name: a**ctionPerformed(arg0:ActionEvent) **Beschreibung:** Behandlung des Events, nach Eingabe des Users	

<table>
<tr>
<td>ConnectingDialog</td>
<td>Username und Zertifikate angeben. Es wird der zu vergebene Name des Users festgelegt und die Zertifikate, für den offenen und privaten Schlüssel werden als absoluter Pfad angegeben.

ID : ConDia01
Name: initButton()
Beschreibung:
Lädt des Save Buttons

ID : ConDia02
Name: certFile()
Beschreibung:
FileChooser für den Zertifikat mit offenen Schlüssel.

ID : ConDia03
Name: pKeyFile()
Beschreibung:
FileChooser für den privaten Schlüssel.

ID : ConDia04
Name: initInfrastructurModus()
Beschreibung:
Erstellen des Dialoges für einen Infrastructur Einstellung.

ID : ConDia05
Name: actionPerformed(arg0:ActionEvent)
Beschreibung:
Erstellen des neuen Benutzers mit Namen und starten des Anonymus Channels.</td>
<td><u>Attribute:</u>
Mw : Mainwindow
Save : JButton
Name : JTextField
Cert : JButton
pKey : JButton
certStatus : JLabel
certFile : JFileChooser
pKeyFile : JFIleChooser
certFile_ : File
pKeyFile_ : File
recievePort : JTextField
isOn : JCHeckBox
isOff : JCheckBox
recievePortPanel : JPanel
<u>Kommunikationspartner:</u>
Mainwindow, GuiManagment</td>
</tr>
</table>

CreateChannel Dialog	Anlegen eines neuen Channels. Es wird zu jedem neuen Channel ein Name, sowie die Bezeichnung benötigt.	**Attribute:** Mw : MainWindow Save : JButton Name : JtextField Desc : JTextField Closed : JCheckBox **Kommunikationspartner:** Mainwindow, Channelmanagment
	ID : CreChanDia01 **Name : a**ctionPerformed(arg0:ActionEvent) **Beschreibung:** Erstellen des neuen Benutzers mit Namen und starten des Anonymus Channels.	
	ID : CreChanDia02 **Name :** initButton() **Beschreibung:** Lädt des Save Buttons	
	ID : CreChanDia03 **Name :** channelName() **Beschreibung:** Eingabefeld für den Channel Namen. **ID :** CreChanDia04 **Name:** channelDescription() **Beschreibung:** Eingabefeld für die Beschreibung des Channels.	
	ID : CreChanDia05 **Name:** closedChannel() **Beschreibung:** Checkbox für Angabe, ob der neue Channel vom Typ closed ist	
	Verwaltung der Ereignisse Alle Ereignisse werden Zentral hier zusammengetragen und weitergeleitet.	**Attribute:** Mw : MainWindow **Kommunikationspartner:** Mainwindow
	ID : EventHan01 **Name:** eventConnect() **Beschreibung:** Verbinden	
	ID : EventHan02 **Name:** eventDisconnect() **Beschreibung:** Trennen	
	ID : EventHan03 **Name:** eventCreateNewChannel()	

EventHandling	**Beschreibung:** Neuen Channel erstellen	
	ID : EventHan04 **Name:** eventSendFile() **Beschreibung:** Daten senden	
	ID : EventHan05 **Name:** eventExitProgram() **Beschreibung:** Programm verlassen	
	ID : EventHan06 **Name:** eventWindowSettings() **Beschreibung:** Allgemeine Programm Einstellungen	
	ID : EventHan07 **Name:** eventInfrastructorModus() **Beschreibung:** Infrastruktur Modus umstellen	
	ID : EventHan08 **Name:** eventVersion() **Beschreibung:** Programm Version	
	ID : EventHan09 **Name:** eventCloseChannel() **Beschreibung:** Channel schließen/ verlassen.	
	ID : EventHan10 **Name:** eventInviteUser() **Beschreibung:** User einladen	

<table>
<tr><td rowspan="5">InfrascructurDialog</td><td>Infrastruktur Modus umstellen</td><td rowspan="5"><u>Attribute:</u>
mw : MainWindow
insert : JBotton
remove : JBotton
update : JBotton
peerList : JList
ip : JTextField
port : JTextField
pl : PeerList
<u>Kommunikationspartner:</u>
Mainwindow, Peerlist</td></tr>
<tr><td>ID : InfStruc01
Name: actionPerformed(arg0:ActionEvent)
Beschreibung:
Ereignisse verarbeiten.</td></tr>
<tr><td>ID : InfStruc02
Name: initButton()
Beschreibung:
Lädt des Insert Buttons</td></tr>
<tr><td>ID : InfStruc03
Name: peerList()
Beschreibung:
Erstellt eine Peerlist Panel</td></tr>
<tr><td>ID : InfStruc03
Name: ipAdress()
Beschreibung:
Erstellt eine Panel für IP-adress und Port nummer Eingeben.</td></tr>
<tr><td rowspan="2">InviteUserDialog</td><td>User in Channel einladen</td><td rowspan="2"><u>Attribute:</u>
mw : MainWindow
save : JButton
userList : JList
c : Channel
<u>Kommunikationspartner:</u>
Mainwindow, UserManagment</td></tr>
<tr><td>ID : InUserDia01
Name : initButton()
Beschreibung:
Lädt des Save Buttons</td></tr>
</table>

	ID : InUserDia02 **Name :** initFileList() **Beschreibung:** FileChooser für den Zertifikat mit offenen Schlüssel.	
	ID : InUserDia03 **Name: a**ctionPerformed(arg0:ActionEvent) **Beschreibung:** Ereignisse verarbeiten.	
Settings	Globale und Teilweise Einstellungen des Programm	**Attribute:** width : int heigt : int font_name : String font_size : int font_color : color tabback_color : color **Kommunikationspartner:** Mainwindow
	ID : Settings01 **Name :** writeGUIConfigFile() **Beschreibung:** Speichert die Einstellungen des MainWindow.	
	ID : Settings02 **Name :** readGUIConfigFile() **Beschreibung:** Ließt die Einstellungen aus einer Datei und stellt das MainWindow ein.	
	ID : Settings03 **Name :** existsGUIConfig() **Beschreibung:** Prüft ob die Datei, mit den Einstellungen vorhanden ist.	
SimpleActionDialog	Einfache Modale Dialoge. Dialoge, zum Verlassen des Programmes, oder schließen eines Channels mit Bestätigung.	**Attribute:** mw : MainWindow **Kommunikationspartner:** Mainwindow
	ID : SimActDia01 **Name :** exitProgramm() **Beschreibung:** Öffnet eine Dialog zum verlassen des Programmes.	
	ID : SimActDia02 **Name :** closeTab(channelName: String) **Beschreibung:** Öffnet eine Dialog zum verlassen des Channels.	

	ID : SimActDia03 **Name :** displayMessage(errorMessage: String) **Beschreibung:** Öffnet eine Dialog zum Error	
WindowSetting Dialog	Einstellungen des Programmes Hintergrundfarbe, Schriftfarbe, Schriftgröße, etc.	**Attribute:** mw : MainWindow update : JButton i_width : JTextField i_height : JTextField i_fontname : JCheckBox i_fontsize : JTextField showColorForFont : JButton showColorForTab : JButton **Kommunikationspartner:** Mainwindow
	ID : WinSetDia01 **Name :** updateWindowsSettings() **Beschreibung:** Setzt alle einstellbaren Eigenschaften von MainWindow.	
	ID : WinSetDia02 **Name :** actionPerformed(arg0:ActionEvent) **Beschreibung:** Ereignisse verarbeiten.	
	ID : WinSetDia03 **Name :** initButton() **Beschreibung:** Laden des update Button.	
	ID : WinSetDia04 **Name :** windowSettingsWidth() **Beschreibung:** Gibt ein Pannel zum einstellen der Breite von MainWindow wieder.	
	ID : WinSetDia05 **Name :** windowSettingsHeigth() **Beschreibung:** Gibt ein Pannel zum einstellen der Höhe von MainWindow wieder.	
	ID : WinSetDia06 **Name :** windowSettingsFontName() **Beschreibung:** Gibt ein Pannel zum einstellen der Schriftart von MainWindow wieder.	

	ID : WinSetDia07 **Name :** windowSettingsFontSize() **Beschreibung:** Gibt ein Pannel zum einstellen der Schriftgröße von MainWindow wieder.	
	ID : WinSetDia08 **Name :** windowSettingsFontColor() **Beschreibung:** Gibt ein Pannel zum einstellen der Schriftfarbe von MainWindow wieder.	
	ID : WinSetDia09 **Name :** windowSettingsTabColor() **Beschreibung:** Gibt ein Pannel zum einstellen der Hintergrundfarbe von MainWindow wieder.	

3.3 Implementierung von Komponente K02: Verwaltung

Die Verwaltung ist für die Verwaltung von Benutzern und Kanälen zuständig. Das UserManagement kümmert sich um die Benutzer, das ChannelManagement um die Kanäle und das GuiManagement interagiert zwischen der GUI und den anderen Klassen der Komponente. Die Channel-Klasse wird in ClosedChannel, OpenChannel und AnonymousChannel spezialisiert, um diese zu trennen. Da nur eine Verwaltung nötig ist, werden die Management-Klassen über Instanzen behandelt, ähnlich dem Entwurfsmuster Singleton.

3.3.1 Klassendiagramm

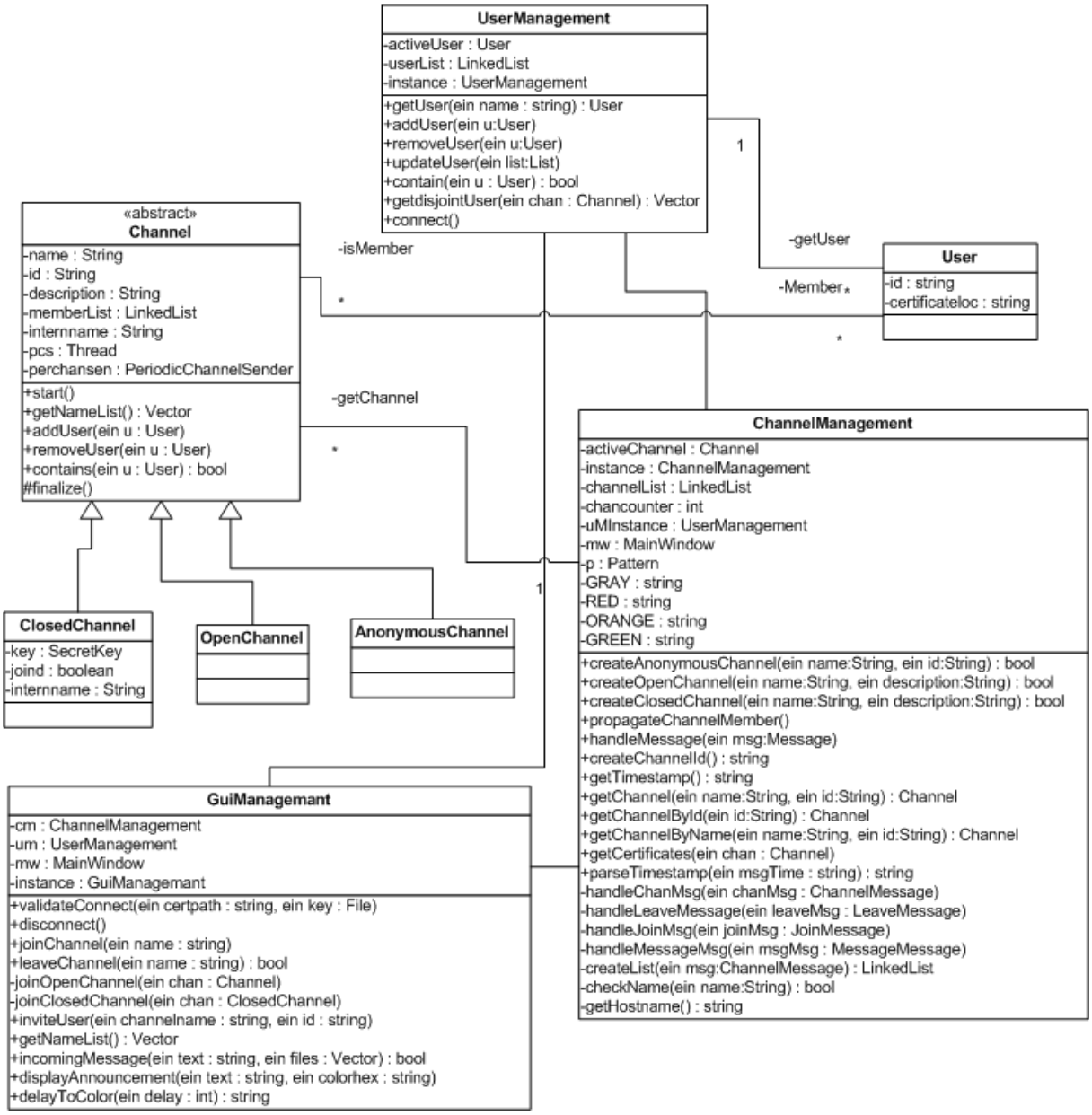

Abbildung 3: Klassendiagramm: Verwaltung

3.3.2 Erläuterung

<table>
<tr><th>Klasse</th><th>Aufgabe</th><th>Attribute und Kommunikationspartner</th></tr>
<tr><td rowspan="8">User-Management</td><td>Das Management verwaltet die User, die sich alle im Netz befinden und den User, der das Programm benutzt.</td><td rowspan="8"><u>Attribute:</u>
aktiveUser : User
userList : LinkedList

<u>Kommunikationspartner:</u>
User: Das UserManagement verwaltet die User die existieren.
Allgemein ist die Klasse Kommunikator zum Erhalt von UserInformationen</td></tr>
<tr><td>ID: UM01
Name: getUser(id:String) : User
Beschreibung: Gibt einem das Userobjekt anhand eines Namens/ID</td></tr>
<tr><td>ID: UM04
Name: addUser(u:User)
Beschreibung: Fügt einen User in die Liste aller User hinzu</td></tr>
<tr><td>ID: UM05
Name: removeUser(u:User)
Beschreibung: entfernt einen User aus der Liste</td></tr>
<tr><td>ID: UM06
Name: updateUser()
Beschreibung: Updatet die Liste anhand der Routingtabelle</td></tr>
<tr><td>ID: UM07
Name: contain(u:User)
Beschreibung: liefert „true“ falls ein Benutzer mit gleicher ID schon vorhanden ist</td></tr>
<tr><td>ID: UM08
Name: getdisjointUser(chan:Channel) : Vector
Beschreibung: liefert einen Vector mit Benutzern, die sich im Netz, aber nicht in dem Kanal befinden, der über den Parameter angegeben wird</td></tr>
<tr><td rowspan="2">User</td><td>Klasse zur Beschreibung eines Benutzers</td><td rowspan="2"><u>Attribute:</u>
id:String
certificateloc:String
<u>Kommunikationspartner:</u>
Userobjekte werden nur von anderen Klassen genutzt</td></tr>
<tr><td>ID: User01
Name: checkForCert() : String
Beschreibung: prüft ob schon ein Zertifikat zu dem zu erstellenden User vorhanden ist und liefert den Pfad zu diesem</td></tr>
</table>

<table>
<tr><td rowspan="14">Channel-Management</td><td>Die Klasse verwaltet die Kanäle.</td><td rowspan="14"><u>Attribute:</u>
activeChannel: Channel
channelList : LinkedList
chancounter : int
uMinstance : UserManagement
mw : MainWindow
p : Pattern
GRAY : string
RED : string
ORANGE : string
GREEN : string

<u>Kommunikationspartner:</u>
Channel: Das ChannelManagement verwaltet die Channel die existieren</td></tr>
<tr><td>ID: O33 (Schnittstelle)
Name: createOpenChannel(name:String)
Beschreibung: Erstellt einen offenen Kanal anhand eines Namens</td></tr>
<tr><td>ID: O33 (Schnittstelle)
Name: createClosedChannel(name:String)
Beschreibung: Erstellt einen geschlossenen Kanal anhand eines Namens.</td></tr>
<tr><td>ID: CM01
Name: propagateChannelMember(c:Channel)
Beschreibung: Wird ausgeführt, wenn eine ChannelNachricht, unabhängig von den periodischen, verschickt werden soll.</td></tr>
<tr><td>ID: O40 (Schnittstelle)
Name: handleMessage(msg:Message)
Beschreibung: Behandelt ankommende Nachrichten die für Kanäle bestimmt sind.</td></tr>
<tr><td>ID: CM02
Name: createChannelId() : String
Beschreibung: erstellt eine Kanal ID mithile von UUID</td></tr>
<tr><td>ID: O42 (Schnittstelle)
Name: getTimestamp() : String
Beschreibung: liefert einen Timestamp</td></tr>
<tr><td>ID: CM03
Name: getChannel(name: String, id:String) : Channel
Beschreibung: liefert den Kanal passend zu den Parametern</td></tr>
<tr><td>ID: CM04
Name: getChannelById(id:String) : Channel
Beschreibung: liefert den Kanal passend zum Parametern</td></tr>
<tr><td>ID: CM05
Name: getChannelByName(name:String) : Channel
Beschreibung: liefert den Kanal passend zum Parametern, hier dient der interne Name als Vergleichsobjekt</td></tr>
<tr><td>ID: CM06
Name: getCertificates(chan:Channel)
Beschreibung: verschickt GETCERTIFICATE Nachrichten für einen bestimmten Kanal</td></tr>
</table>

	ID: CM07 **Name:** parseTimestamp(msgTime:String) **Beschreibung:** addiert falls nötig UTC offset und Zeitzone zu einem Timestamp einer eingehenden Nachricht	
	ID: CM08 **Name:** createList(msg:ChannelMessage) : LinkedList **Beschreibung:** erstellt eine Liste aus Benutzern anhand der ChannelMessage	
	ID: CM09 **Name:** checkName(name:String) : bool **Beschreibung:** prüft ob schon ein Kanal mit diesem Namen vorhanden ist	
	ID: CM10 **Name:** getHostname() : String **Beschreibung:** liefert den Hostnamen des Nodes	
<<abstract>> Channel	Klasse die einen Kanal realisiert.	**Attribute:** name:String id:String description:String memberList : LinkedList internname : String pcs : Thread perchansen : PeriodicChannelSender **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
	ID: Channel01 **Name:** start() **Beschreibung:** startet den Thread für das periodische senden von Channel Announcements	
	ID: O42 (Schnittstelle) **Name:** getNameList() : Vector **Beschreibung:** liefert eine Liste mit Strings, der Namen der Teilnehmer dieses Kanals	
	ID: Channel02 **Name:** addUser(u:User) **Beschreibung:** fügt dem Kanal einen Benutzer hinzu	
	ID: Channel03 **Name:** removeUser(u:User) **Beschreibung:** entfernt einen Benutzer aus diesem Kanal	
	ID: Channel04 **Name:** contains(u:User) : bool **Beschreibung:** liefert „true“ falls der angegebene Benutzer sich im Kanal befindet	
	ID: Cannel05 **Name:** finalize() **Beschreibung:** beendet den Announcement sendenen Thread und entfernt den Kanal aus der Kanalliste des Channelmanagements	

Closed-Channel	Realisierung des geschlossenen Kanals **ID:** CChannel01 **Name:** getKeyloc() : String **Beschreibung:** Gibt einem den Ort, wo der Schlüssel gespeichert ist zurück. **ID:** CChannel02 **Name:** setKeyloc(loc:String) **Beschreibung:** Setzt den Ort, wo der Schlüssel gespeichert ist. **ID:** CChannel03 **Name:** isjoined() : boolean **Beschreibung:** Prüft, ob der Benutzer im Programm dem Channel beigetreten ist, weil im geschlossenen Kanal die Liste der Benutzer des Kanals nur darüber Auskunft gibt wer beitreten darf.	**Attribute:** keyloc:String joined:Boolean **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
OpenChannel	Benötigt keine zusätzlichen Funktionen	**Attribute:** Keine zusätzlichen **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
Anonymous-Channel	Benötigt keine zusätzlichen Funktionen	**Attribute:** Name:String=“anonymous“ Id:String=”Anonymous@Anonymous” **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
GuiManagement	Klasse zur GUI Verwaltung **ID:** GM01 **Name:** validateConnect() : bool **Beschreibung:** prüft ob das vom Benutzer angegenene Zertifikat und privater Schlüssel gültig sind und startet die Programm Threads **ID:** GM02 **Name:** disconnect() **Beschreibung:** wird beim beenden des Programms ausgeführt. Sendet LeaveMessages für alle Kanäle in denen sich der Programm Benutzer befindet	**Attribute:** cm:ChannelManagement um:UserManagemant mw:MainWindow **Kommunikationspartner:** Interagiert zwischen ChannelManagement und der GUI

	ID: O31 (Schnittstelle) **Name:** joinChannel(name:String) **Beschreibung:** fügt den aktiven Benutzer dem Kanal angegeben im Parameter hinzu, und versendet JoinMessage, handelt es sich um einen geschlossenen Kanal wird nur dem eine JoinMessage verschickt und dies dem Benutzer angezeigt	
	ID: O32 (Schnittstelle) **Name:** leaveChannel(name:String) **Beschreibung:** entfernt den aktiven Benutzer aus dem Kanal	
	ID: O42 (Schnittstelle) **Name:** getNameList() : Vector **Beschreibung:** liefert einen Vector mit den Namen aller Kanäle	
	ID: O30 (Schnittstelle) **Name:** incommingMessage(text:String, files:Vector) : bool **Beschreibung:** erstellt eine Nachricht anhand des Textes und falls vorhanden dem Anhang und gibt sie weiter an die nachricht Komponente	
	ID: GM02 **Name:** disconnect() **Beschreibung:** wird beim beenden des Programms ausgeführt. Sendet LeaveMessages für alle Kanäle in denen sich	
	ID: GM03 **Name:** delayToColor(delay:int) : String **Beschreibung:** liefert einen Hexadezimal codierten Farbwert anhand des delays. Der Farbwert wird berechnet in mehrere Stufen von grün nach rot.	
	ID: O17 (Schnittstelle) **Name:** inviteUser(channelname:String, id:String) **Beschreibung:** fügt einen Benutzer (anhand der ID) einem geschlossenen Kanal (anhand des Kanalnamens) hinzu	
	ID: O41 (Schnittstelle) **Name:** displayAnnouncement(text:String, colorhex:String) **Beschreibung:** zeigt eine Systemmeldung im aktiven Kanal an	

Implementierung von Komponente K03: Nachricht

Die Nachrichtenkomponente ist für die Verwaltung der Nachrichten zuständig. RecieveManagement und SendManagement sind die Klassen, die empfangene bzw. zu sendende Nachrichten speichern, damit sie vom MessageWorker verarbeiten und gesendet werden können. Die PeriodicMessageSender-Klasse ist dafür zuständig periodische Nachrichten zu versenden und bei gespeicherten Nachrichten, die durch ein Nack empfangen worden sind, zu prüfen, ob sie verschickt werden können. Die Message-Klasse wird in die einzelnen Unterklassen, je nach Messagetyp, spezialisiert. Dabei wird bei jeder Spezialisierung die getXML()-Funktion überlagert, um die richtige XML-Datei, wie sie im Protokoll definiert ist, zu erzeugen. Um überall das Parsen einer XML-Datei zu ermöglichen, besitzt die Messageklasse eine statische Pars-Funktion. Letztendlich sind die Managementklassen als Singleton zu implementieren.

3.3.3 Klassendiagramm

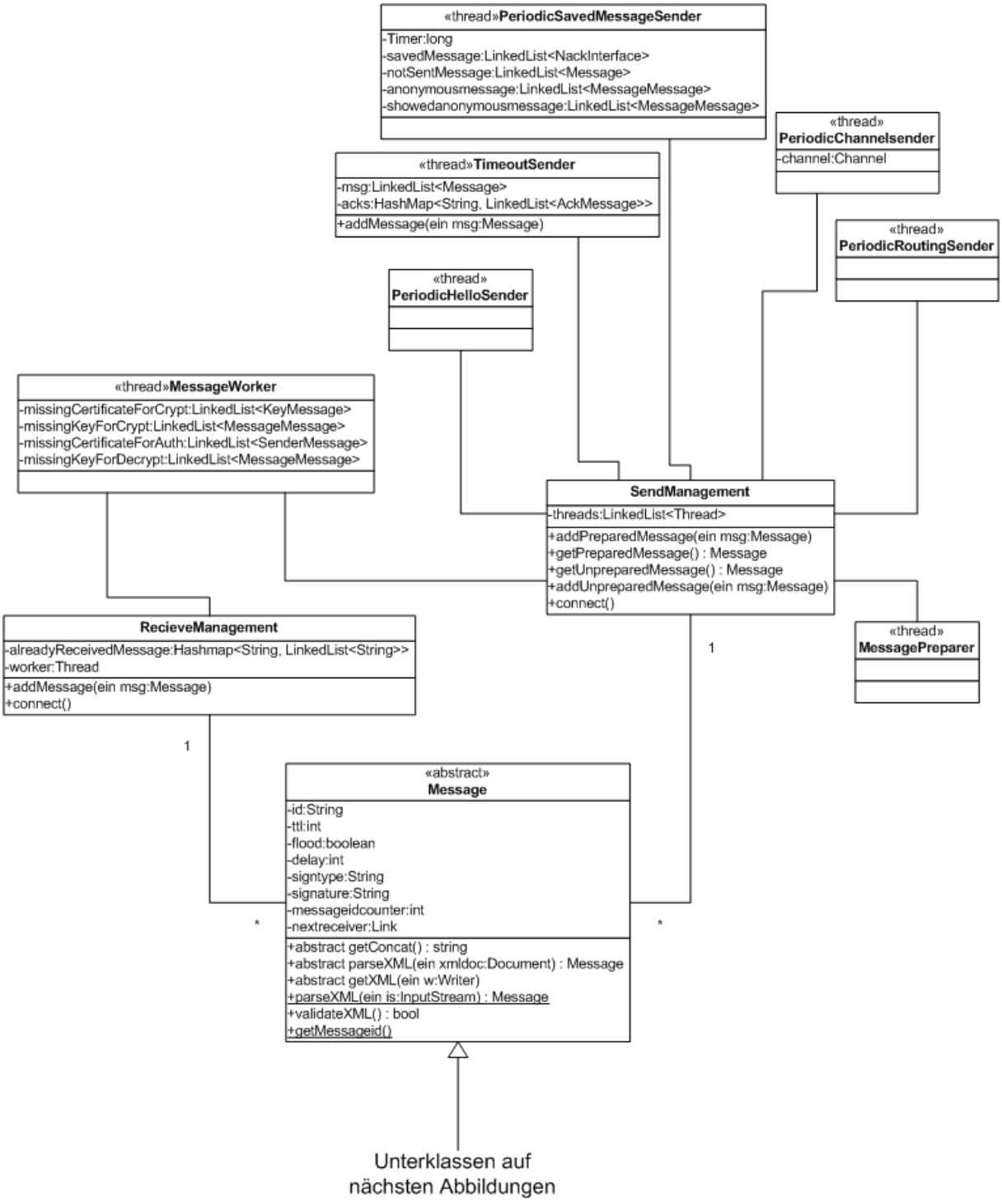

Abbildung 4: Klassendiagramm: Nachricht

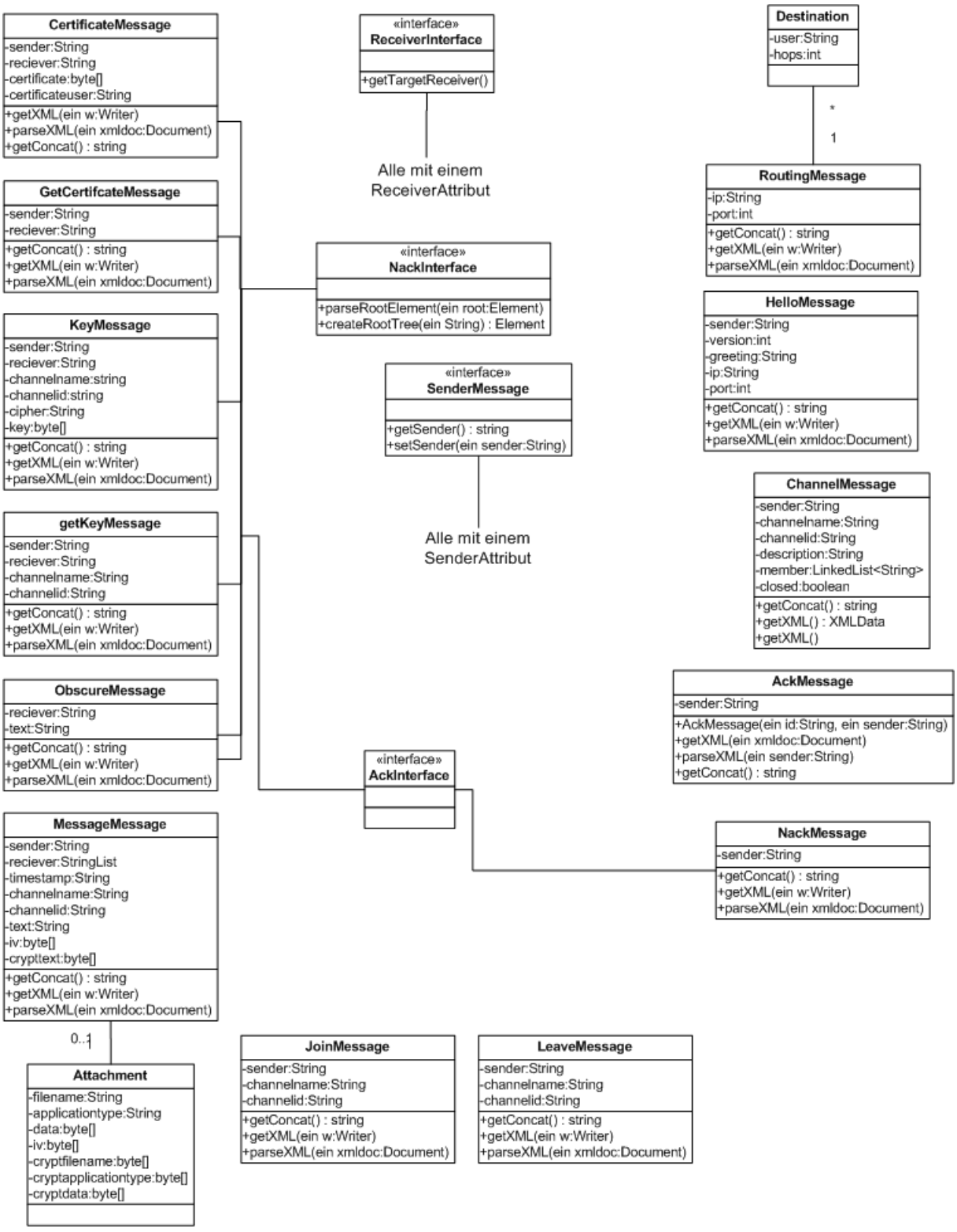

Abbildung 5: Klassendiagramm: Unterklassen der Messageklasse

3.3.4 Erläuterung

<table>
<tr><th>Klasse</th><th>Aufgabe</th><th>Attribute und Kommunikationspartner</th></tr>
<tr><td rowspan="4">Recieve-Management</td><td>Puffert die Empfangenen Nachrichten</td><td rowspan="4"><u>Attribute:</u>
alreadyReceivedMessage:HashMap
worker:Thread;

<u>Kommunikationspartner:</u>
Message: Das Management verwaltet die Messages, die empfangen worden sind.
MessageWorker: Behandelt die angekommenen Nachrichten
Allgemein erhält das RecieveManagement-objekt nur Nachrichten, von der Senden/Empfangen-Komponente.</td></tr>
<tr><td>ID: O51 (Schnittstelle)
Name: addMessage(msg:Message)
Beschreibung: Fügt eine Nachricht dem Puffer hinzu. Ebenfalls wird, geprüft ob die Nachricht schon einmal empfangen worden ist. AckMEssages werden direkt an den TimeOutSender übergeben.</td></tr>
<tr><td>ID: ReMa01
Name: getMessage() : Message
Beschreibung: Holt eine Nachricht aus dem Puffer. Wird vom MessageWorker genutzt</td></tr>
<tr><td>ID: ReMa02
Name: connect()
Beschreibung: Startet die Threads die im ReceieveManagement sind. (MessageWorker)</td></tr>
<tr><td rowspan="6">Send-Management</td><td>Puffert zu sendende Nachrichten</td><td rowspan="6"><u>Attribute:</u>
threads: LinkedList<Thread>
unpreparedlist: LinkedList<Message>
preparedlist: LinkedList<Message>

<u>Kommunikationspartner:</u>
Message: Das Management verwaltet die Messages, die zum Senden bereit sind.
MessagePreparer bereitet die Nachrichten zum senden vor.
Allgemein erhält das SendManagementobjekt nur Nachrichten von anderen Klassen die Nachrichten losschicken möchte. Die Senden/Empfangen-Komponente holt sich Nachrichten zum senden</td></tr>
<tr><td>ID: SeMa01
Name: addPreparedMessage(msg:Message)
Beschreibung: Fügt eine vorbereitete Nachricht hinzu.</td></tr>
<tr><td>ID: SeMa02
Name: getUnpreparedMessage() : Message
Beschreibung: Entfernt eine unvorbereitete Nachricht aus dem Sendepuffer.</td></tr>
<tr><td>ID: O52 (Schnittstelle)
Name: getPreparedMessage(msg:Message)
Beschreibung: Gibt eine vorbereite Nachricht zum Senden zurück</td></tr>
<tr><td>ID: O50 (Schnittstelle)
Name:
addUnpreparedMessage(msg:Message)
Beschreibung: Fügt eine unvorbereitete Nachricht hinzu.</td></tr>
<tr><td>ID: SeMa03
Name: connect()
Beschreibung: Startet die gesammten Thread aus dem SendManagement. Die periodischen Sender, den TimeoutSender und den Messagepreparer;</td></tr>
</table>

Message-Worker	Holt sich Nachrichten aus dem RecieveManagement und verarbeitet sie. Falls Zertifkate oder Key ankommen prüft es, ob Nachrichten darauf warten. Es für auch die Sicherheitsmethoden durch die beim empfangen druchgeführt werden müssen, wie authentifizieren, deobscuren und entschlüsseln. Weitergeleitete Nachrichten werden ans SendManagement übergeben.	**Attribute:** missingCertificateForCrypt : KeyMessageList missingCertificateForAuth : SenderMessage-List missingKeyForCrypt:MessageMessageList missingKeyForDeCrypt:MessageMessageList **Kommunikationspartner:** **RecieveManagement:** Dauerhaftes holen von Empfangenen Nachrichten, um diese zu verarbeiten. **SendManagement:** Je nach empfangener Nachricht müssen diese zum weiterleiten an dieses Objekt übergeben werden.
Message-Preparer	Bereitet Nachrichten zum Senden vor. Dazu gehören die Sicherheitsaspekte. Ebenfalls wird im Routing abgefragt, wohin die Message gehen soll und ob sie gesplittet werden muss.	**Attribute:** keine **Kommunikationspartner:** **SendManagement**: Holt sich daraus Nachrichten und packt sie vorbereitet wieder rein.
Periodic-HelloSender	Packt Hello Nachrichten periodisch in das SendManagement	**Attribute:** Keine **Kommunikationspartner:** **SendManagement:** Periodisch werden die Nachrichten an dieses Objekt gegeben
Periodic-Channel-Sender	Für jeden Kanal existiert ein Channelsender der die ChannelNachrichten verschickt.	**Attribute:** channel:Channel **Kommunikationspartner:** **SendManagement:** Periodisch werden die Nachrichten an dieses Objekt gegeben
Periodic-Routing-Sender	Sendet periodisch Routing-Nachrichten indem diese in das SendManagement gepackt werden.	**Attribute:** keine **Kommunikationspartner:** **SendManagement:** Periodisch werden die Nachrichten an dieses Objekt gegeben

Periodic-Saved-Message-Sender	Sende regelmäßig durch Nack empfangene Nachrichten und noch nicht gesendete Nachrichten neu, sobald eine Route existiert. Ebenfalls prüft es, ob gesendete anonyme Nachrichten auch später entschlüsselt ankommen.	**Attribute:** savedMessage:LinkedList<NackInterface> notSendMessage:LinkedList<Message> anonymousmessage : LinkedList<MessageMessage> showanonymousmessage : LinkedList<MessageMessage> **Kommunikationspartner:** **SendManagement:** Periodisch werden die Nachrichten an dieses Objekt gegeben
Timeout-Sender	Speichert die Nachrichten für die es ein Ack gibt und die gesendet werden und wartet auf ein Ack. Ansonsten sendet es die Nachricht neu.	**Attribute:** msg:LinkedList<Message> HashMap<String, LinkedList<AckMessage>> **Kommunikationspartner:** **SendManagement:** Periodisch werden die Nachrichten an dieses Objekt gegeben
	ID: TiSe01 **Name:** addMessage(msg:Message) **Beschreibung:** Fügt gesendete Nachricht hinzu für die ein Ack erwartet wird.	
	ID: TiSe02 **Name:** addAck(ack:AckMessage) **Beschreibung:** Fügt empfangene AckMessages hinzu	
	ID: TiSe02 **Name:** handleTooBigMessageSize(msg:Message) **Beschreibung:** Gibt eine entsprechende Meldung aus, wenn eine Nachricht zu groß ist.	
<<abstract>> Message	Realisiert die Nachrichten.	**Attribute:** id:String ttl:int flood:boolean delay:int
	ID: MSG01 **Name:** abstract getXML() : XMLData **Beschreibung:** Wandelt die Nachrichten in XML-Daten um.	

		signtype:String signatur:String messageidCounter:int; nextReceiver:Link **Kommunikationspartner:** Wird von den anderen Klassen benutzt um Nachrichten zu behandeln. Gleiches gilt für alle Spezialisierungen.
	ID: MSG02 **Name:** static parseXML(x:XMLData) : Message **Beschreibung:** Wandelt XML-Daten in Message-Objekte um.	
	ID: MSG03 **Name:** abstract getConcat() : String **Beschreibung:** Gibt die Verbindung von Inhalten zum Signieren zurück	
	ID: MSG04 **Name:** abstract getparseXML (xmldoc:Document) : String **Beschreibung:** Nachdem der Nachrichtentyp durch die static Methode festgestellt wird, wird sie ans Object weitergeben.	
HelloMessage	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String version:int greeting:String ip:String port:int
RoutingMessage	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** hops:int user:String ip:String port:int
KeyMessage	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String receiver:String channelname:String channeled:String cipher:String key:byte[]
GetKey-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String receiver:String channelname:String

NackMessage	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String message:NackInterface
AckMessage	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String
Certificate-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String receiver:String certificate:byte[] certificateuser:String
GetCertificate-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String receiver:String
Channel-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String channelname:String channeled:String description:String closed:boolean
JoinMessage	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String channelname:String channeled:String
Leave-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String channelname:String channeled:String
Obscure-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** receiver:String text:String

Message-Message	Überlagerte getXML()-Funktion Überlagerte parseXml(xmldoc:Document)-Funktion Überlagerte getConcatFunktion	**Attribute:** sender:String receiver:StringList timepstamp:String channelname:String channeled:String text:String iv:byte[] crypttext:byte[]; **Kommunikationspartner:** **Attachement:** Wird benutzt um Anhänge zu speichern.
Attachement	Realisiert einen Anhang	**Attribute:** filename:String apptype:String data:byte[] iv:byte cryptfilename:String cryptapptype:String cryptdata:byte[] **Kommunikationspartner:** Wird von **MessageMessage**-Objekten für Anhänge benutzt
<<Interface>> Receiver-Interface	Wird von allen Messages Implementiert die einen Empfänger haben, um einen eindeutigen empfänger zurückzugeben. Bei MessageMessage z.B. den ersten in der Empfängerliste, nachdem sie gesplittet worden ist **ID:** ReInt01 **Name:** getTargetReceiver():String **Beschreibung:** Gibt einen Receiver zurück	
<<Interface>> NackInterface	Alle Messages die als Nack verschickt werden sollen müssen dieses Interface implementieren **ID:** NaInt01 **Name:** parseRootElement(root:Element) **Beschreibung:** parst ein neues RootElement. So kann die Message in dem Nack geparst werden.	

	ID: NaInt02 **Name:** creatRootTree(s:String) **Beschreibung:** Baut einen neuen Baum auf mit dem String als rootname.	
<<Interface>> AckInterface	Alle Messages die ein Ack erwarten implementieren dieses Interface	
<<Interface>> Sender- Message	Jede Message die einen Sender hat implementiert das Interface	
	ID: SenderM01 **Name:** getSender() **Beschreibung:** gibt einem den Sender zurück	

3.4 Implementierung von Komponente K04: Sicherheit

Die Secure Komponente liegt als einzelne Klasse vor, genauer, als Singleton, die Klasse darf also nur einmal erzeugt werden und ist global verfügbar. Die Methoden der Klasse sollen die Sicherheitsvorgaben des Protokolls erfüllen. So wird in einem Geschlossenen Kanal ver- und entschlüsselt via Data Encryption Standard, im Öffentlichen und Geschlossenen Kanal wird via X.509 zertifiziert und authentifiziert um in Anonymen Kanal wird mit Hilfe des X.509 Zertifikats nach dem Zwiebelprinzip ver-/entschlüsselt.

3.4.1 Klassendiagramm

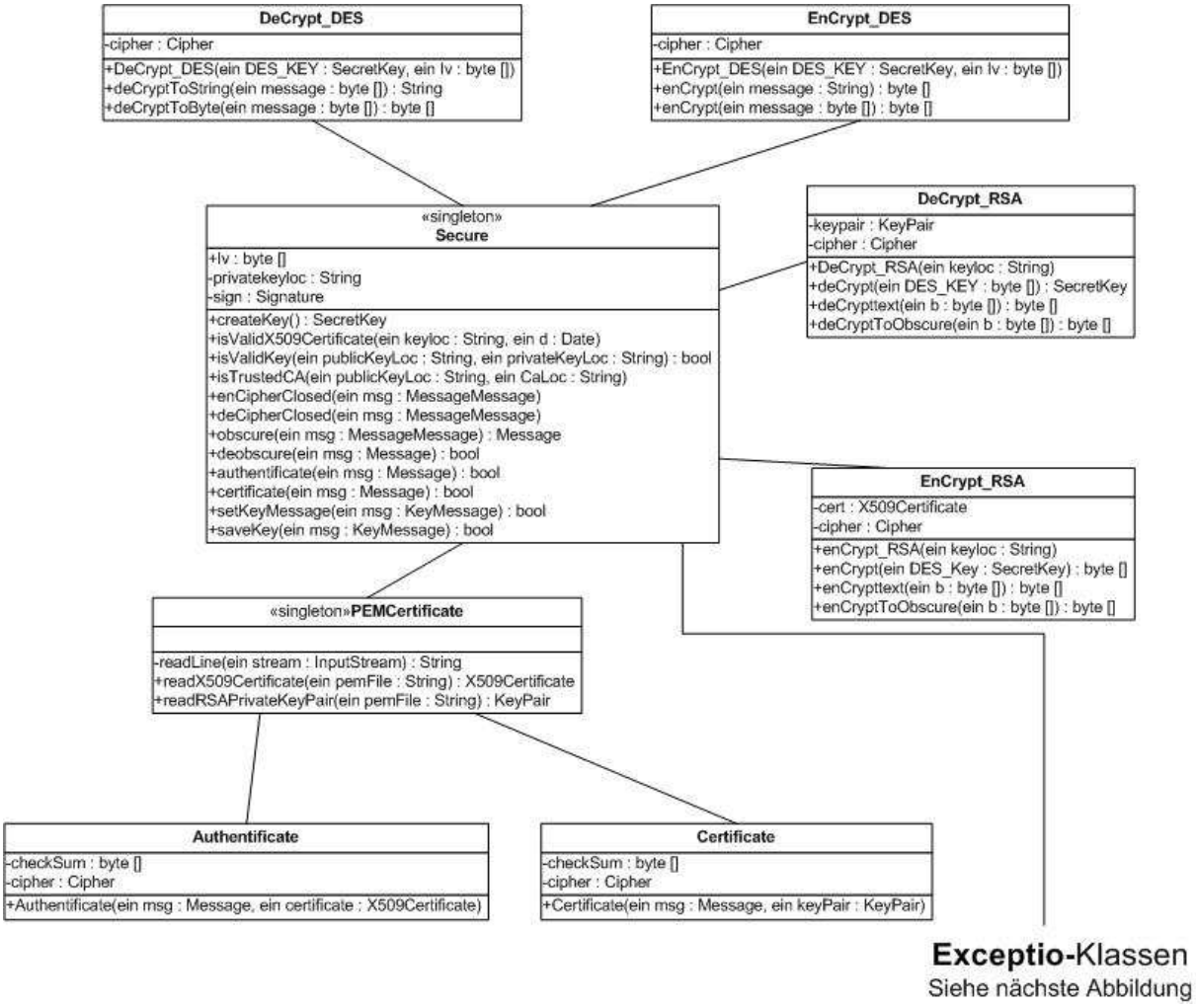

Abbildung 6: Klassendiagramm: Secure

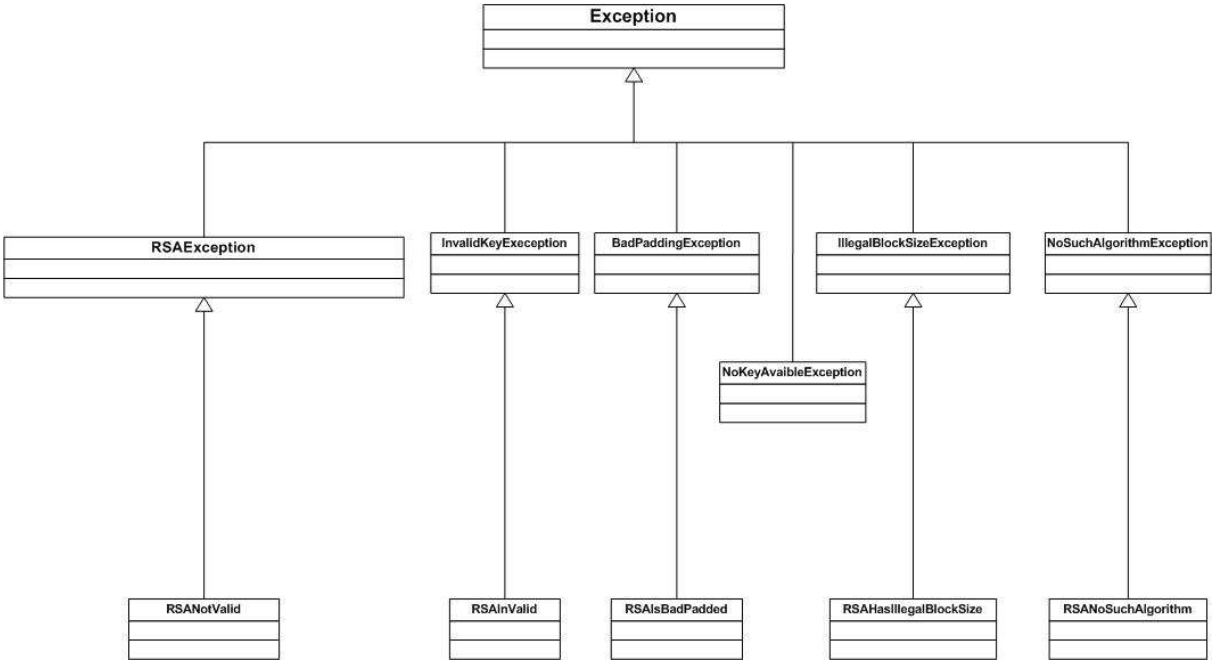

Abbildung 7:Klassendiagramm: SecureExeption

3.4.2 Erläuterung

<table>
<tr><th>Klasse</th><th>Aufgabe</th><th>Attribute und Kommunikationspartner</th></tr>
<tr><td rowspan="5"><<Singelton>>
Secure</td><td>Secure - Verschlüsseln / Entschlüsseln von Nachrichten. Sicherheit vergibt jedem geschlossenem Channel einen symmetrischen DES-Key, mit dem die Rohdaten verschlüsselt und entschlüsselt werden. Der DES-Key vom geschlossenen Channel darf niemals öffentlich zwischen den Usern ausgetauscht werden. Damit der DES-Keys weitergegeben werden darf, muss dieser auch verschlüsselt werden.
Dazu wird das asymmetrische RAS verfahren genutzt: Es müssen zunächst die Empfänger ermittelt werden. Jeder Empfänger besitzt ein Attribute, die den Pfad zu seinem Zertifikat (offener Schlüssel) angibt. Mit diesem Zertifikat, wird der DEY-KEY verschlüsselt. Nun kann der Versender den verschlüsselten DEY-KEY an die Empfänger versenden und die entsprechenden Empfänger können mit dem Privaten Schlüssel den DES-KEY entschlüsseln um anschließend die Rohdaten zu entschlüsseln.</td><td rowspan="5"><u>Attribute:</u>
Iv: byte[]
privatekeyloc: String
sign: Signature

<u>Kommunikationspartner:</u>
Allgemein wird dies zum verschlüssel, entschlüsseln, zertifizieren und authentifizieren von Messageobjekten benutzt.</td></tr>
<tr><td>ID: SEC01
Name: createKey () : SecretKey
Beschreibung:
Gibt den symmetrischen DES-KEY für einen geschlossenen Channels wieder. Mit diesem Schlüssel, werden die Rohdaten verschlüsselt und entschlüsselt. NICHT mit dem asymmetrische RAS-Schlüsselpaar.</td></tr>
<tr><td>ID: SEC02
Name: isValidX509Certificate(String keyloc, Date d)
Beschreibung:
Überprüft das Zertifikat auf seine Gültigkeit.</td></tr>
<tr><td>ID: SEC03
Name: isValidKey(String publicKeyLoc, String privateKeyLoc)
Beschreibung:
Überprüft das Key auf die Gültigkeit.</td></tr>
<tr><td>ID: SEC04
Name: isTrustedCA(String publicKeyLoc, String CaLoc)
Beschreibung:
Überprüft den Aussteller des Zertifikats und dem openKey auf die Gültigkeit.</td></tr>
</table>

	ID: SEC05 **Name:** enCipherClosed(msg: MessageMessage) **Beschreibung:** Entschlüsselt den Inhalt einer Nachricht eines Geschlossenen Kanals und falls vorhanden auch den Anhang.	
	ID: SEC06 **Name:** deCipherClosed(msg: MessageMessage) **Beschreibung:** Verschlüsselt eine Nachricht mit dem Schlüssel des jeweiligen Geschlossenen Kanals, falls vorhanden auch den Anhang.	
	ID: SEC07 **Name:** obscure (msg: MessageMessage) : Message **Beschreibung:** Erhält eine MESSAGE Nachricht, verschlüsselt diese und erstellt daraus eine OBSCURE Nachricht, so oft, wie User in der UserList stehen. Verschlüsselt wird mit den Zertifikaten der User aus der Liste.	
	ID: SEC08 **Name:** deobscure(msg: MessageMessage) : boolean **Beschreibung:** Entschlüsselt den Inhalt einer Nachricht eines Anonymen Kanals.	
	ID: SEC09 **Name:** authentificate(m: Message) : boolean **Beschreibung:** Liefert „true“, falls das Zertifikat, des Senders, der Nachricht, gültig ist.	
	ID: SEC010 **Name:** certificate(m: Message) : boolean **Beschreibung:** Zertifiziert eine Nachricht.	
	ID: SEC011 **Name:** setKeyMessage (msg: KeyMessage) : boolean **Beschreibung:** Fügt der Nachricht den verschlüsselten Channel KEY hinzu.	
	ID: SEC012	

	Name: saveKey (msg:Key Message) : boolean **Beschreibung:** Fügt den Channel KEY aus der Nachricht gelesenen Nachricht dem Channel hinzu.	
<<Singelton>> PEM-Certificate	Zertifikat mit Public-KEY, sowie Private-KEY im PKCS#8 werden eingelesen. Für das letztere Zertifikat wird auf eine externe Komponente zugegriffen, da Java in dieser Version keine entsprechenden Parser anbietet Zertifikate mit Private-Key die mit openSSL erstellt wurden im PKSC#8 Format liest.	**Attribute:** keine **Kommunikationspartner:** **Allgemein** wird dies zum Einlesen der PEM Datei benutzt.
	ID: PEMC01 **Name:** readLine(InputStream stream): String **Beschreibung:** Einlesen der PEM Datei in ein Stringbuffer.	
	ID: PEMC02 **Name:** readX509Certificate(String pemFile): X509Certificate **Beschreibung:** Erstellt aus der einzulesenden PEM Datei ein x509Certificat, aus dem der Public-Key herauszulesen ist.	
	ID: PEMC03 **Name:** readRSAPrivateKeyPair(String pemFile): KeyPair **Beschreibung:** Liest den Private-Key von der PEM Datei des jeweiligen Benutzers ein und gibt ein Paar wieder.	
Authentificate	Authentificate - Entschlüsselt die Signatur einer Message.	**Attribute:** checkSum: byte[] cipher : Cipher **Kommunikationspartner:** **Allgemein** wird dies zum Entschlüsseln der Signatur benutzt.
	ID: AUTH01 **Name:** getCheckSum(): String **Beschreibung:** Gibt die Prüfsumme der Nachricht wieder.	
Certitificate	Certificate - Verschlüsselt die Signatur einer Message	**Attribute:** checkSum: byte[] cipher : Cipher **Kommunikationspartner:** **Allgemein** wird dies zum Verschlüsseln der Signatur benutzt.
	ID: CERT01 **Name:** getCheckSum(): String **Beschreibung:** Gibt die Prüfsumme der Nachricht wieder.	
EnCrypt_DES	EnCrypt_DES - Verschlüsselung nach DES Prinzip.Daten werden mit dem symmetrischen DES Algorithmus verschlüsselt.	**Attribute:** cipher : Cipher **Kommunikationspartner:** **Allgemein** wird dies zum Verschlüsseln von Daten benutzt.
	ID: ENCDES01 **Name:** enCrypt(String message): byte[] **Beschreibung:** Verschlüsselung eines Textes nach DES-Verfahren.	

	ID: ENCDES02 **Name:** enCrypt(byte[] message): byte[] **Beschreibung:** Verschlüsselung eines Textes nach DES-Verfahren in bytes.	
DeCrypt_DES	DeCrypt_DES - Entschlüsselung nach DES Prinzip.Daten werden mit dem symmetrischen DES Algorithmus verschlüsselt.	**Attribute:** cipher : Cipher **Kommunikationspartner:** **Allgemein** wird dies zum Entschlüsseln von Daten benutzt.
	ID: DECDES01 **Name:** deCryptToString(byte[] message):String **Beschreibung:** Entschlüsselung eines Textes nach DES-Verfahren.	
	ID: DECDES02 **Name:** deCryptToByte(**byte**[] message): byte[] **Beschreibung:** Entschlüsselung eines Textes nach DES-Verfahren in bytes.	
EnCrypt_RSA	EnCrypt_RSA - Verschlüsselung nach RSA Prinzip. Daten (DES-KEY) wird nach dem RSA-Schlüsselverfahren verschlüsselt.	**Attribute:** cert: X509Certificate cipher : Cipher **Kommunikationspartner:** **Allgemein** wird dies zum Verschlüsseln von Daten benutzt.
	ID: ENCRAS01 **Name:** enCrypt(SecretKey DES_Key):byte[] **Beschreibung:** Verschlüsselt den DES-KEY mit dem öffentlichen Schlüssel des Empfängers.	
	ID: ENCRAS02 **Name:** deCryptToByte(**byte**[] message): byte[] **Beschreibung:** Entschlüsselung eines Textes nach DES-Verfahren in bytes.	
	ID: ENCRAS03 **Name:** enCryptToBObscure(**byte**[]b): byte[] **Beschreibung:** Verschlüsselung einer nach Zwiebel-Verfahren in bytes.	
DeCrypt_RSA	DeCrypt_RSA - Entschlüsselung nach RSA Prinzip. Daten (DES-KEY) wird nach dem RSA-Schlüsselverfahren entschlüsselt.	**Attribute:** keypair: KeyPair cipher : Cipher **Kommunikationspartner:** **Allgemein** wird dies zum entschlüsseln von Daten benutzt.
	ID: DECRAS01 **Name:** deCrypt(**byte**[] DES_Key):SecretKey **Beschreibung:** Entschlüsselt den DES-KEY mit dem privaten Schlüssel des Empfängers.	

	ID: DECRAS02 **Name:** deCrypttext(**byte**[] b): byte[] **Beschreibung:** Entschlüsselt den Text und gibt ihn als byte[] wieder.	
	ID:D ECRAS03 **Name: d**eCryptToBObscure(**byte**[]b): byte[] **Beschreibung:** Entschlüsselung einer nach Zwiebel-Verfahren in bytes.	
RSAException	RSAException - Fehlerklassen für RSA – Zertifikate. Ausgabe von Benutzerdefinierten Exceptions, bei ungültigen, fehlerhaften Zertifikaten.	**Attribute:** keine **Kommunikationspartner:** **Allgemein** wird dies zum Behandeln von Ausnahmen/Fehlermeldungen benutzt.

3.5 Implementierung von Komponente K05: Senden/Empfangen

Die Senden/Empfangen-Komponente übernimmt das Senden und empfangen der Nachrichten über UDP. Es holt sich Nachrichten aus der Nachrichtenkomponente um sie zu verschicken und übergibt empfangene Nachrichten. Der Port in dem die Nachrichten empfangen werden kann bei Benutzung des Infrastrukturmodus geändert werden.

3.5.1 Klassendiagramm

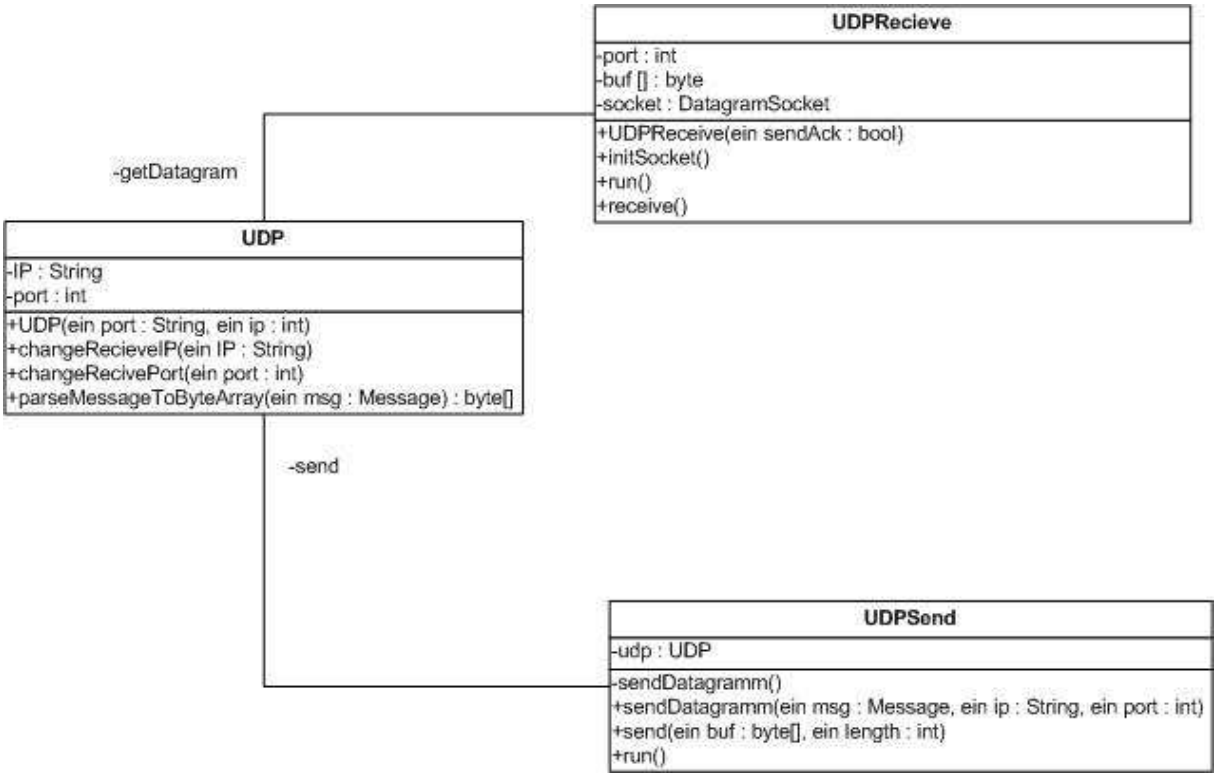

Abbildung 8: Klassendiagramm: Senden/Empfangen

3.5.2 Erläuterung

<table>
<tr><th>Klasse</th><th>Aufgabe</th><th>Attribute und Kommunikationspartner</th></tr>
<tr><td rowspan="4">UDP-Recieve</td><td>Übergibt empfangene Nachrichten an das RecieveManagement</td><td rowspan="4"><u>Attribute:</u>
Port: int
buf[]: byte
socket: DatagramSocket

<u>Kommunikationspartner:</u>
RecieveManagement: Nachrichten die ankommen müssen als Messageobjekte übergeben werden.
UDP: Zum Nachrichten aus dem Netz holen</td></tr>
<tr><td>ID: UDPR01
Name: initSocket ()
Beschreibung: Über den Port werden die neuen Socket initialisiert.</td></tr>
<tr><td>ID: UDPR03
Name: run()
Beschreibung: Starten des Thread zum empfangen von Nachrichten.</td></tr>
<tr><td>ID: UDPR02
Name: receive ()
Beschreibung: Es wird ein Port freigegeben auf dem die Nachrichten empfangen werden. Zu beachten ist, dass die zu empfange Datei nicht Größer als die Vereinbarte maximale Größe für Nachrichten, da sonst die Nachricht unvollständig ist.</td></tr>
<tr><td rowspan="5">UDPSend</td><td>Holt sich Nachrichten aus dem SendManagement und versendet sie.</td><td rowspan="5"><u>Attribute:</u>
udp: UDP

<u>Kommunikationspartner:</u>
SendManagement: Nachrichten, die losgeschickt werden sollen müssen von diesem Objekt geholt werden.
UDP: Zum Nachrichten in das Netz geben</td></tr>
<tr><td>ID: UDPS01
Name: sendDatagramm()
Beschreibung: Die zu versende(n) Nachricht(en) werden vom SendManagement abgerufen und zum versenden als byte[] geparst.</td></tr>
<tr><td>ID: UDPS02
Name: sendDatagramm(Message msg,String ip, int port)
Beschreibung: Änderung der IP über den die Datagramme gesendet und empfangen werden.</td></tr>
<tr><td>ID: UDPS03
Name: send(byte[] buf, int length)
Beschreibung: Versendet die Nachricht als DatagramPacket, an den Empfänger</td></tr>
<tr><td>ID: UDPS04
Name: run ( )
Beschreibung: Startet den Thread zum versenden der Nachricht</td></tr>
</table>

UDP	UDPKlasse die das Sendet/Empfangen realisiert.	**Attribute:** Ip:String Port: int **Kommunikationspartner:** Wird von **UDPSend** und **UDPRecieve** genutzt um Nachrichten aus dem Netz empfangen und ins Netz senden zu können.
	ID: UDP01 **Name:** changeRecieveIP() **Beschreibung:** Änderung der IP über den die Datagramme gesendet und empfangen werden.	
	ID: UDP02 **Name:** changeRecievePort() **Beschreibung:** Änderung des Ports über den die Datagramme gesendet und empfangen werden.	
	ID: UDP03 **Name:** parseMessageToByteArray (Message msg): byte[] **Beschreibung:** Parst eine Nachricht im XML Format in byte[] um.	

3.6 Implementierung von Komponente K06: Routing

Die Routingkomponente ist dafür zuständig die Netzstruktur durch Hello- und RoutingNachrichten aufzubauen. Ebenfalls kann man Routingnachrichten erstellen lassen und RoutingRichtungen abfragen.

3.6.1 Klassendiagramm

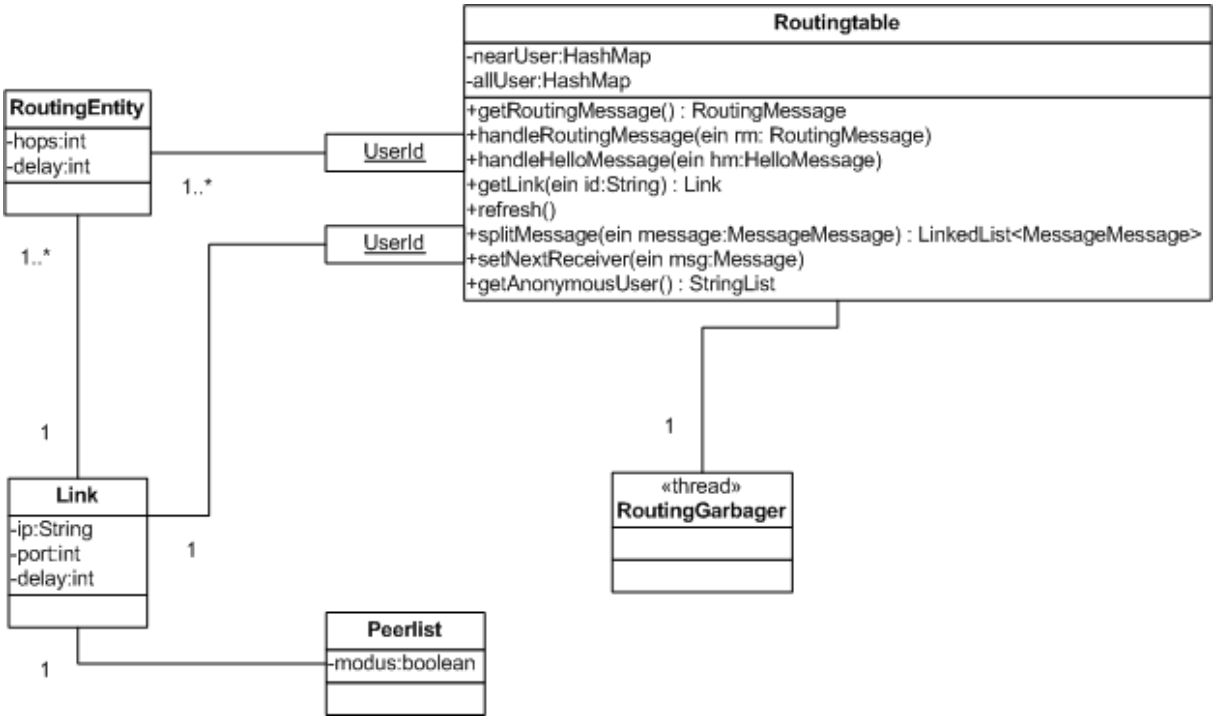

Abbildung 9: Klassendiagramm: Routing

3.6.2 Erläuterung

Klasse	Aufgabe	Attribute und Kommunikationspartner
Routingtable	Baut die Routingtabelle auf und ab. Splittet Nachrichten und setzt die nächsten Empfänger für eine Nachricht.	**Attribute:** nearUser:HasMap allUser:HashMap **Kommunikationspartner:** **MessageWorker:** Der MessageWorker übergibt die richtigen Nachrichten an das Routingtableobjekt. **PeriodicRoutingSender:** Holt sich aktuelle RoutingMessages, falls diese verschickt werden müssen.
	ID: O73 (Schnittstelle) **Name:** getRoutingMessage() : RoutingMessage **Beschreibung:** Gibt eine aktuelle Routing-Nachricht zurück, um sie anderen Senden zu können.	
	ID: O71 (Schnittstelle) **Name:** handleRoutingMessage (rmsg:RoutingMessage) **Beschreibung:** Aktualisiert die Tabelle anhand einer RoutingMessage	

	ID: O72 (Schnittstelle) **Name:** handleHelloMessage(hmsg:HelloMessage) **Beschreibung:** Aktualisiert die Tabelle.	
	ID: RT01 **Name:** refresh() **Beschreibung:** Wird vom RoutingGarbager benutzt um die Tabelle von Links aufzuräumen die nicht mehr da sind.	
	ID: RT02 **Name:** getLink(id:String) : Link **Beschreibung:** Gibt einem einen Link für einen User zurück.	
	ID: O70 (Schnittstelle) **Name:** setNextReceiver(msg:Message) : boolean **Beschreibung:** Setzt den nächsten EmpfängerLink für eine Message. Wenn keiner vorhanden ist wird false zurückgegeben.	
	ID: O75 (Schnittstelle) **Name:** splitMessage(msg:MessageMessage) : LinkedList<MessageMessage> **Beschreibung:** Splittet eine Message je nachdem welchen Weg sie gehen müssen.	
	ID: O74 (Schnittstelle) **Name:** getAnonymousUser() : StringList **Beschreibung:** Gibt einem eine Liste zufälliger Benutzer zurück.	
RoutingEntity	Realisiert die Routingeinträge	**Attribute:** hops:int link:Link delay:int **Kommunikationspartner:** Wird von der **Routingtable**-Klasse genutzt.

Link	Realisiert Links. Also direkt Verbindungen eines Knotens	**Attribute:** ip:String port:int delay:int **Kommunikationspartner:** Wird von der **Routingtable**-Klasse genutzt
<<thread>> Routing- Garbager	Ruft die refresh() Methode der Routingtable auf um alte links und Routingeinträge zu entfernen.	**Kommunikationspartner:** Ruft die **Routingtable**-Klasse auf
PeerList	Kann Links auf nehmen und entfernen. Wird für den Infrastrukturmodus benutzt.	**Attribute:** modus:Boolean peers:List<Link> **Kommunikationspartner:** Wird abgefragt wenn Infrastrukurmodus aktiviert ist.

4 Datenmodell

Für das Programm wird kein Datenmodell benötigt. Pro Programm werden lediglich eingestellte Fenstereinstellungen gespeichert, die kein Beziehungsmodel benötigen. Ebenfalls benötigen die Zertifikate und Schlüssel keine Datenbeziehungsmodelle. Sie werden neben dem Programm in einem „data“ Ordner als Dateien gespeichert.