



# Funktionale und graphische Erweiterung des XT-Clients

**Diplomarbeit**

zur Erlangung des Grades eines
Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
**Bernhard Wolf**

Erstgutachter: Prof. Dr. Christoph Steigner, Institut für Informatik
Zweitgutachter: Dipl. Inf. Frank Bohdanowicz, Institut für Informatik

Koblenz, im Juli 2010

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

_____

Bernhard Wolf

Koblenz, 28. Juli 2010

# Inhaltsverzeichnis

# Zusammenfassung

Einer der Forschungsschwerpunkte der AG Rechnernetze ist das *Routing Information Protocol*. Eine eigene kompatible Erweiterung dieses Routingprotokolls ist der Routing Information Protocol with Metric-based Topology Investigation (RMTI [ehemals RIP-MTI]). Um dieses Protokoll testen und mit seinem Vorgänger vergleichen zu können, wird die Virtualisierungssoftware VNUML eingesetzt.

In diesen virtualisierten Netzwerken werden Router, die das RMTI-Protokoll einsetzten, mittels der *Zebra/Quagga Routing Software Suite* beobachtet. Dabei wird das Verhalten des Protokolls in unterschiedlichsten Simulationssituationen analysiert und bewertet.

Um solche Testabläufe zentral zu steuern und protokollieren zu können wurde die Anwendung RIP-XT (XTPeer) erstellt und in fortführenden Diplomarbeiten kontinuierlich erweitert. Sie ist Schnittstelle zu den *Zebra/Quagga*-Routern und kann diese steuern. Zusätzlich sammelt und analysiert sie die Routing-Informationen der Router.

Über ein GUI kann ein Benutzer diese Abläufe steuern. Um einen visuellen Überblick über eine Netzwerktopologie zu erhalten, besitzt das GUI auch eine Topologie-Anzeige. Die Anzeige repräsentiert das Gesamte Netzwerk durch Symbole, über die auch Interaktionen mit der Simulation möglich sind.

Ziel dieser Diplomarbeit war es, die bisherige Topologie-Anzeige zu überarbeiten, um sie für neue Anforderungen anzupassen. Des weiteren wurden funktionale Erweiterungen in das GUI des RIP-XTs eingebettet.

# 1 Einleitung

Routing Protokolle sind ein wichtiger Bestandteil der modernen Informationsgesellschaft. Durch ihre Fähigkeit der Selbstorganisation ermöglichen sie einen flexiblen und schnellen Transfer von Daten durch die globalen und lokalen Netzwerke weltweit. Manche dieser Protokolle sind schon seit dem Anbeginn des Internets und seines Vorläufers, dem ARPANET, im Dienst. Diese Protokolle mussten aber auf Grund des schnellen Wachstums der Netzwerke auch immer neuen Anforderungen gerecht werden. Nur durch kontinuierliche Weiterentwicklung konnten sie ihren Platz in den Netzwerken behaupten. Eines dieser Protokolle ist das *Routing Information Protocol*[1], kurz RIP.

Eine aktuelle Weiterentwicklung ist das RMTI[2] Protokoll. RMTI basiert auf dem von Andreas Schmid 1999 an der Universität Koblenz-Landau entwickelt RIP-MTI[3]. RIP-MTI ist eine zu RIP kompatible Erweiterung, um die Konvergenz-Eigenschaft von RIP zu verbessern, insbesondere wenn Topologie-Änderungen eintreten oder Fehlinformationen im Netzwerk kursieren.

Eine besondere Schwäche von RIP ist das *Count-To-Infinity* Problem, kurz CTI genannt. Zur ausführlichen Beschreibung des Problems verweise ich hier auf den »Exkurs CTI« im Anhang. Auf Grund der ständigen Weiterentwicklung des RIP-MTIs ist eine Namensänderung vorgenommen worden und das Protokoll trägt nun die Bezeichnung RMTI. Im Weiteren wird diese Bezeichnung verwendet, auch wenn in den angegeben Literaturangaben noch die damals zutreffende Bezeichnung benutzt worden ist.

Um ein solches Netzwerkprotokoll testen und analysieren zu können, benötigt es entweder eine reale Netzwerkumgebung, in der ausführliche Tests gemacht werden können, oder eine virtuelle Testumgebung, die möglichst realistisch die zu untersuchenden Details darstellen kann. In der Arbeitsgruppe Rechnernetze der Universität Koblenz wurde hierfür die Virtualisierungs- und Simulations-Software VNUML[4] ausgewählt.

VNUML baut auf dem *User Mode Linux* auf. Damit lässt sich eine Simulation eines Netzwerkes erstellen, bei der die virtuellen Computer des Netzwerks, auch *Virtual Machines* genannt, auf einem einzelnen realen Computer erzeugt werden. Der Computer, auf dem die Simulation ausgeführt wird, wird als *Host* bezeichnet. Virtuelle Netzwerke lassen sich aber auch auf mehrere reale Computer verteilen und anschließend zu einem

1 *RIP* Version 2, RFC2453, G. Malkin. November 1998.
2 *Routing Information Protocol with Metric-based Topology Investigation.*
3 *Routing Information Protocol with Minimal Topology Information.* Siehe hierzu [SCH99].
4 *VNUML*, Abkürzung für *Virtual Network User Mode Linux*. Entwickelt an dem Departamento de Ingeniería de Sistemas Telemáticos (DIT) der Universidad Politécnica de Madrid (UPM) in Spanien.

großen Netzwerk zusammenfügen. Dadurch lässt sich relativ schnell eine Netzwerktopologie aufbauen, die nur schwer in realer Hardware zu realisieren ist.

Vorteile einer solchen Simulation sind natürlich die Einsparung der Hardwarekosten und eine Flexibilität im Aufbau oder in der Konfiguration einer virtuellen Netzwerktopologie. Eine *Virtual Machine* (*VM*) besteht bei VNUML aus einem Dateisystem und einem Kernel. Beide können standardmäßig für alle simulierten *VMs* verwendet oder auch für jede *VM* explizit festgelegt werden.

Auf dem Filesystem der AG Rechnernetze ist die *Quagga Routing Suite Software*[5] vorinstalliert, welche ein Implementation eines RIP *Daemon* besitzt. Zum direkten Vergleich beider Protokolle, RIP und RMTI, wurde von Tobias Koch [KOC05] der Zebra/Quagga RIP *Daemon* um die Erweiterungen von RMTI ergänzt. Durch diese Erweiterung ließ sich nun in Szenarien das Verhalten beider Protokolle bei provozierten Ausfällen vergleichen und so Rückschlüsse auf das Konvergenzverhalten schließen. Noch bis zu dieser Zeit musste der Ablauf eines solchen Test mittels Skripten von Hand durchgeführt werden. Um einen CTI zu provozieren, musste ein günstiger Zeitpunkt in der Update-Abfolge gewählt werden. Dies war aufwändig und erforderte mehrere Terminal-Verbindungen, einerseits zu den *VMs*, um Interfaces an- und abzuschalten, andererseits zu den *Routing Daemons,* um die Routingtabelle zu beobachten. Um diese Handhabung zu vereinfachen, wurde im Rahmen der Diplomarbeit von Daniel Pähler [PAE06] an der Universität Koblenz-Landau die Software RIP-XT[6] erstellt. Das ursprüngliche Programm RIP-XT bestand aus zwei Teilen, dem XT-Server und dem XT-Client. Etwas genauer auf den Aufbau und die Funktionalität des RIP-XT wird in Kapitel 2 »RIP-XT« eingegangen, ebenso auf Erweiterungen, die durch nachfolgende Diplomarbeiten an ihm vorgenommen wurden.

Die vorliegende Diplomarbeit führt die Weiterentwicklung des RIP-XT fort. Sie hat es sich zur Aufgabe gemacht, den XT-Client mit seiner grafischen Benutzeroberfläche (GUI[7]) zu erweitern und auch die bisherigen Funktionalitäten an neue Benutzerbedürfnisse anzupassen. Dazu wurde das zur graphischen Darstellung der Netzwerktopologie verwendete Java[8] API, JGraph[9] beibehalten und neu spezialisiert[10].

5 *Quagga Routing Software Suite*, GPL licensed IPv4/IPv6 Routing Software.
6 *RIP-XT: Routing Information Protocol - e**X**ternally **T**riggered*.
7 *GUI*, engl. Abkürzung: *Graphical User Interface*.
8 *Java* und alle Java basierenden Marken sind Warenzeichen oder registriere Warenzeichen von Sun Microsystems.
9 *JGraph* untersteht dem Copyright © der JGraph Ltd. 2004-2009.
10 *Spezialisieren* im Sinne der Vererbung von Klasseneigenschaften in der objektorientierte Programmiersprache Java.

# 2 RIP-XT

Der RIP-XT, auch als XTPeer bezeichnet, wurde zu dem Zweck entwickelt, die RIP *Routing Daemons* der *Quagga Routing Suite Software*[11] *an*zusteuern, um so direkt in das Update-Verhalten eines Routers eingreifen zu können und dieses zu manipulieren. Dadurch lassen sich an einer Stelle gebündelt Update-Konstellationen provozieren, die wie eingangs beschrieben, früher nur mit großem Aufwand erzeugt werden konnten und genaues Timing voraussetzten.

## 2.1 Quagga Routing Software Suite

Bei Quagga handelt es sich um eine *Routing Software Suite*, die verschiedenste Routing Protokolle wie OSPF v2, OSPF v3, RIP v1, RIP v2, RIPng und BGP-4 beinhaltet. Quagga entstand als eigenständiges Projekt aus der Abspaltung des GNU Zebra Projekts[12] und besitzt den Anspruch, die Software einer größeren Schicht von Benutzern und Entwicklern zwecks Weiterentwicklung zugänglich zu machen. So baut sich Quagga auch nach wie vor aus dem *Core Daemon* Zebra und den *Zserv-Clients* auf.

Bei Zebra handelt es sich ist um einen IP Routing Manager, der *Kernel Routing Table* Updates verwaltet sowie Interface Lookups und das Verteilen von Routen zwischen verschiedenen Routing Protokollen bewerkstelligt. Von seiner Architektur her ist er einerseits Abstraktionsschicht zum UNIX Kernel und dessen *Routing Tables* und anderseits Schnittstelle zu den *Zserv-Clients*. Zu den einzelnen Clients verbindet er sich mittels des *Zserv API*[13], über einen UNIX oder TCP Stream und teilt so Routing Updates den einzelnen Clients mit. Ein Client kann eines der oben genannten Routing Protokolle implementiert haben.

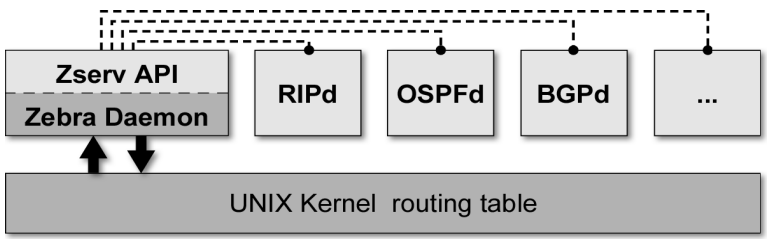


*Abbildung 1: Zebra/Quagga Architektur.*

11 *Quagga Routing Software Suite.* Offizielle Internetseite: www.quagga.net.
12 *GNU Zebra*, entwickelt von Kunihiro Ishiguro 1996. Offizielle Internetseite: www.zebra.org.
13 *API,* Application Programming Interface. Bezeichnet eine Schnittstelle, die von einem Softwaresystem zur Verfügung gestellt wird, um andere Programmen anbinden zu können.

Der Vorteil dieses modularen Aufbaus ist, dass weitere Protokolle durch Dritte implementiert und mittels des Zebra *Daemons* in die *Routing Suite* eingebettet werden können.

## 2.2 XT-Server/-Client Architektur

Für das RMTI Projekt wurde der von Tobias Koch [KOC05] durch den RMTI-Algorithmus modifizierte RIP *Daemon* noch zusätzlich um eine Update-Steuerung von Daniel Pähler [PAE06] erweitert. Um diese zentrale Steuerung der *Daemons* zu erhalten, verwendete er eine Client-Server Architektur. Hierbei verwaltet der Benutzer mittels des XT-Clients die XT-Server, welche in den RIP *Daemon* eingebettet sind und so bei deren Start automatisch mitgestartet werden. Der verwendete Client-Server-Entwurf kehrt das Client-Server-Paradigma etwas um, wie Pähler selbst feststellt[14]. Ein Client kann in diesem Entwurf mit mehreren Servern verbunden sein, aber jeder Server nur mit genau einem Client. Dies hat den Hintergrund, dass es nicht wünschenswert ist ein Szenario mit mehreren Clients zu steuern.

Das Update-Verhalten des RIP *Daemons* bezüglich eines Interfaces wird durch den Status des Interfaces geregelt. Dieser kann entweder auf AUTO, MANUAL oder AUTOTRIGGER gesetzt sein. Dadurch kann für jedes Interface einzeln das Update[15]-Verhalten festlegt werden. Das Standartverhalten von RIP, periodisch alle 30 Sekunden ein Update an alle Nachbarn zu versenden, oder die sogenannten Triggered Updates, die RIP bei Veränderungen sendet, können so für ein ganzes Netzwerk gesteuert werden.

Der Status AUTO entspricht dem Standardverhalten von RIP in Bezug auf seine Updates. In MANUAL werden alle Updates verworfen und nur durch den Benutzer veranlasste Updates werden versendet. AUTOTRIGGER ist ein von Bohdanowicz [BOH08] zusätzliche entwickelter Status und verhindert das Versenden periodischer Updates, lässt aber Triggered Updates in diesem Zustand zu.

Durch die Beeinflussung des Update-Verhaltens kann die Update-Reihenfolge gezielt gesteuert werden. Dadurch ist es möglich, Situationen zu provozieren, in welchen sich die Stärken und Schwächen eines Protokolls aufzeigen lassen. Eine bekannte Schwäche von RIP ist das sogenannte CTI Problem. Ein CTI hat zum einen den Nachteil, dass sich die Router auf Grund der Inkonsistenz des Netzwerkes durch zusätzliche Updates auf die veränderte Konstellation aufmerksam machen und versuchen wieder einen konvergenten Zustand zu erreichen. Diese Updates belasten die Bandbreiten der Netze. Ande-

14 [PAE06], Kapitel 1.2 »RIP-XT«, Seite 8f..

15 *Update Timer*, die Einstellung für die periodischen Updates von RIP kann auf beliebigen Werte gesetzt werden. Standardmäßig sind es 30 Sekunden.

rerseits wird durch die nicht hergestellte Konvergenz Netzwerkverkehr fehlgeleitet, welcher zusätzlich die Bandbreiten belastet. Beide Auswirkungen sind natürlich bei RIP oder RMTI von großem Interesse diese zu vermeiden oder möglichst gering zu halten. Zur genaueren Analyse und Auswertung des RMTI und seinen Verbesserungen gegenüber dem RIP möchte ich auf die zur Zeit aktuellste Diplomarbeit zum Thema RMTI von Frank Bohdanowicz verweisen [BOH08].

Diese Möglichkeit des Beeinflussens von außen über den RIP-XT kann mit dem *Black Box*[16]-Testverfahren verglichen werden. Es wird damit überprüft, ob ein Protokoll eine vorgegebene „Aufgabe“ innerhalb eines akzeptablen Zeitrahmens bewältigen kann. Interessant sind natürlich auch *White Box*[17]-Testverfahren, also die Möglichkeit des Blicks „in“ einen Router. Die SL[18]-Erweiterung kann als solches *White Box*-Verfahren verstanden werden, da mit dieser Erweiterung die Entscheidungen des Routers protokolliert werden. Die SL-Erweiterung versendet immer dann Nachrichten, wenn Routen ausfallen, alternativ Routen angeboten oder falsche, veraltete Routen bei dem Router eintreffen. Dadurch lassen sich Schwächen oder Fehler eines Protokolls ausfindig machen.

## 2.3 SL-Server/-Client Architektur

Um die Protokollierung der Update-Reihenfolgen zu ermöglichen, wurde mit der Diplomarbeit von Stefan Lange [LAN07] der RIP-XT und der Quagga RIP *Daemon* um den SL-Server und den SL-Client erweitert. Hier ist das Client-Server-Konzept umgekehrt zu dem vom RIP-XT. Auf jeder *Virtual Machine (VM)* ist zusätzlich zum XT-Server nun noch ein SL-Client installiert, der mit dem SL-Server auf dem *Host*[19] Rechner verbunden ist. Der SL-Client wird mit dem Quagga *Daemon* gestartet und dann über *Routing Table* Änderungen informiert. Diese Änderungen versendet er als Nachricht an den SL-Server. Der SL-Server sammelt zentral alle Änderungen der *Daemons* und analysiert diese auf CTIs, um die Daten dann dem Benutzer über das GUI des XT-Clients darzustellen. Dort kann jede Nachricht einzeln betrachtet werden, um Ereignisse genau zu untersuchen.

16 *Black Box*-Testverfahren, auch als Funktionale Testverfahren bezeichnet. Dabei wird untersucht, ob ein Programm bei Eingaben aus Äquivalenzklassen ein korrekten Ausgabewert der Klasse liefert [BAL99].
17 *White Box*-Testverfahren, auch als Strukturtestverfahren bezeichnet, siehe [BAL99].
18 *SL:* **S**ource**L**oop(Detection).
19 *Host*, bezeichnet bei VNUML den physikalischen Computer auf dem die Simulation gestartet wird.

Der Server ist als Concurrent-Server umgesetzt und erzeugt somit für jeden SL-Client, mit dem er verbunden ist, einen eigenen Thread, der parallel zu den anderen Verbindungen auf dem Server läuft. Abbildung 2 zeigt den schematischen Aufbau der XT-Server- und SL-Server-Architektur und ihr Zusammenspiel. Bezüglich der Einzelheiten der Implementierung und der verwendeten Kommunikationsprotokolle beider Server-Client Applikationen möchte ich hier nochmals auf die beiden Diplomarbeiten von Daniel Pähler [PAE06] und von Stefan Lange [LAN07] hinweisen.

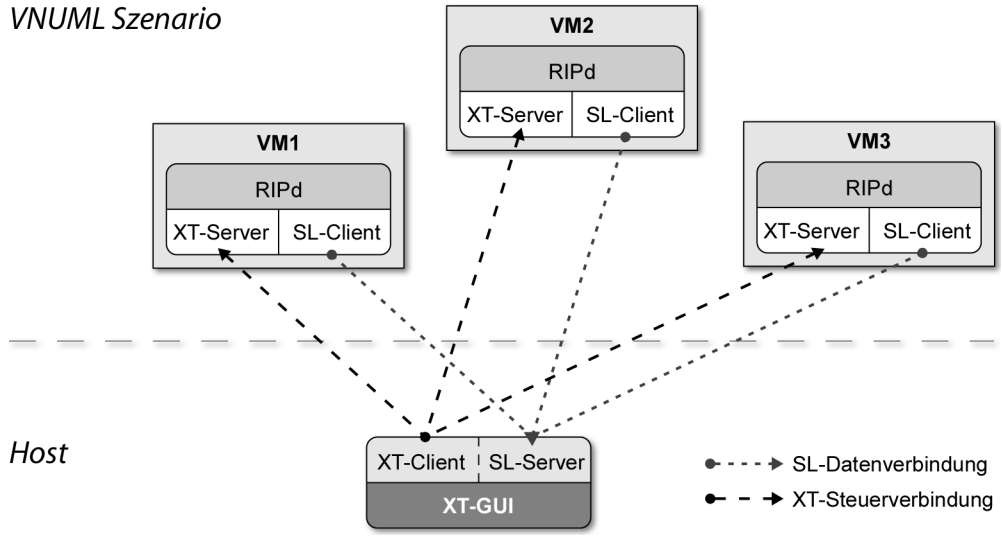


*Abbildung 2: XT-Server/-Client und SL-Server/-Client Architektur.*

## 2.4 XT-Client GUI

Das GUI, welches einen Teil des XT-Clients darstellt, ist das Kernstück des Programms. Aus ihm werden alle Threads[20] gestartet und es ist Sammelpunkt aller Informationen zur Darstellung und Auswertung für den Benutzer. Das GUI sowie der gesamte XT-Client sind in der Programmiersprache Java geschrieben. Dies ist eine Designentscheidung, die von Daniel Pähler getroffen wurde, und bei den nachfolgenden Erweiterungen beibehalten wurde.

Durch die Arbeit von Stefan Lange und den mit seinen Erweiterungen verbunden neuen Aufgaben des GUIs wurde diese modifiziert und einem neuen Layout unterworfen, siehe Abbildung 3. Dazu wurden zusätzlich zur ursprünglichen grafischen Topologieanzeige noch zwei Anzeigebereiche darunter angefügt. Im mittleren Bereich ist ein

20 *Thread*, bezeichnet eine zusätzlichen Prozess, der zum Hauptprozess eines Programms nebenläufig oder parallel verarbeitet wird.

Karteireiter, in dem die einzelnen Router aufgereiht sind. Jeder Router enthält einen Netzwerk-Karteireiter, in welchem jedes Netz mit den Routing-Informationen zu diesem Netz als Kartei aufgelistet ist.

Zuunterst befindet sich der CTI-Verlaufsgraph, der eine graphische Anzeige der Metrik eines Pfades zu einem Netzwerkes visualisiert. Nach Konvergenz eines Netzwerkes ist hier ein konstanter Wert für einen Pfad abzulesen. Erst bei Veränderung des Netzes sind an dieser Stelle Abweichungen zu erkennen. CTIs lassen sich so visuell schnell erfassen durch einen kontinuierlichen Anstieg des Metrik-Graphen bei jedem Update, bis seinem maximal Wert, der einem Ausfall eines Netzwerkes entspricht.

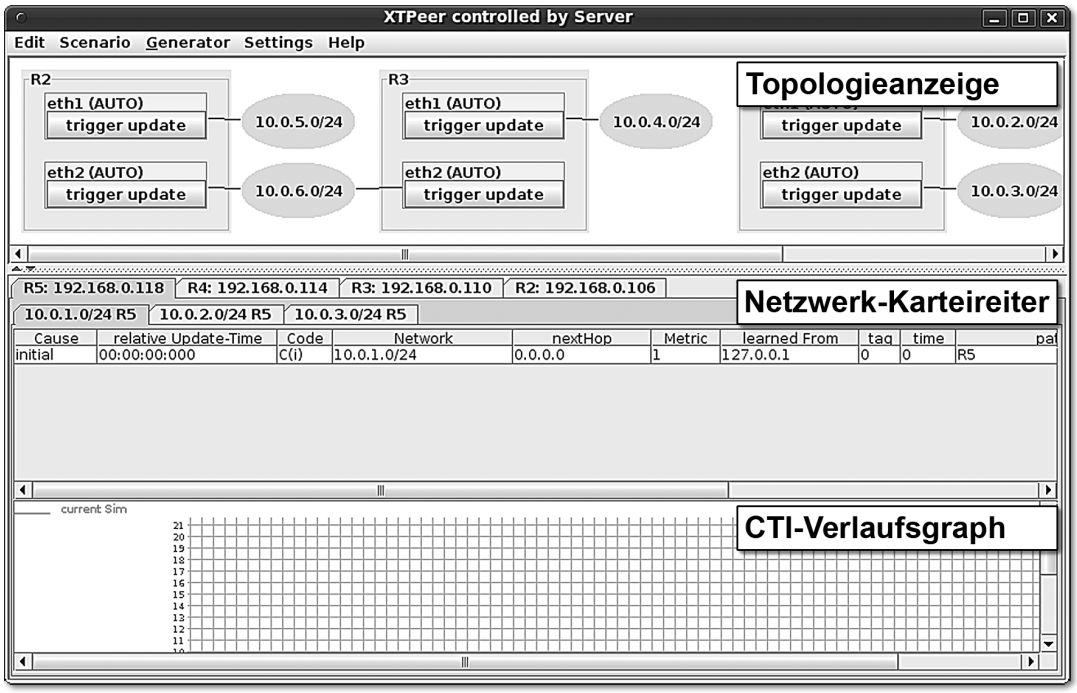


*Abbildung 3: XT-Client Aufbau.*

Das GUI setzt sich aus 18 *Packages* zusammen, durch welche die einzelnen Funktionalitäten gekapselt sind. *Packages* bei Java sind eine Sammlung von Klassen, die wegen ihrer funktionalen Zusammengehörigkeit zu einem *Package* zusammengefasst werden können. Ein *Package* wiederum kann aus weiteren *Packages* bestehen. Durch diesen modularen Aufbau lassen sich Baumstrukturen bildet, um eine umfangreiche Software übersichtlicher zu gliedern. Nach diesem Schema ist auch Java selbst aufgebaut.

Von den 18 *Packages* sind für diese Diplomarbeit aber nur zwei im Wesentlichen von Interesse. Dies ist zum einen das *Package* `backend` und zum anderen das *Package*

`frontend`. Im *Package* `backend` ist die eigentliche Programmlogik enthalten, es bildet das VNUML Netzwerk mit den XT-Servern, Interfaces und Netzwerken nach. Des Weiteren realisiert es die Kommunikation der RIP *Daemons* mit den XT-Servern. Zusätzlich beinhaltet es die veraltete `XTClient` Klasse, welche ursprünglich die XT-Server verwaltete, aber durch die Erweiterung Stefan Langes nicht mehr benötigt wird. Das Verwalten der XT-Server wird seitdem im `frontend`, in der Klasse `XTClientGUI`, gehandhabt. Eine detaillierte Beschreibung der Beziehungen zwischen den Klassen des `backend` lässt sich bei [PAE06], Seite 55ff, nachlesen und aus dem dortigen UML-Diagramm entnehmen.

Für diese Arbeit ist der Aspekt des Erzeugens der JGraph[21] Objekte von Vorrang, um die bestehende Programmstruktur zu verstehen und die entsprechenden Programmstellen zu finden, an welchen Änderungen oder Ergänzungen vorzunehmen sind. Dies bedeutet im Genaueren:

- Wo werden welche JGraph Objekt erzeugt?
- Mit welchen Methoden werden diese erzeugt?
- Wie ist die Reihenfolge der Erzeugung?

Dazu muss auch das *Package* `frontend` betrachtet werden, da das `frontend` weitestgehend das `backend` in seiner Klassenstruktur nachbildet. Die zum `backend` korrespondierenden Klassen treten in diesem *Package* immer paarweise auf, wie z.B.: die Klasse `XTServer` aus dem `backend` und die korrespondierenden Klassen im `frontend`: `XTServerVertex` und `XTServerVertexView`. Dies Verdoppelung der Klassen erklärt sich aus dem Aufbau von JGraph, der detaillierter im Kapitel 4 »JGraph« erläutert wird.

Das Sequenzdiagramm in Abbildung 4 zeigt den Programmablauf bei Erzeugung eines `XTServers` im `backend`. Dies geschieht in der Klasse `XTClientGUI` durch Aufruf der Methode `getAddServersFromXMLMenuItem()`. Die dort ausgewählte XML-Datei wird mit Aufruf der Methode `setServersFromXMLConfig()` verwendet, um ein Vektor von `XTServern` zu erhalten. Basierend auf diesem Vektor wird in der Methode `setServersFromXMLConfig()` durch den Aufruf von `addXTServer()` zuerst der `XTServer` und anschließend der JGraph `XTServerVertex` erzeugt.

21 JGraph ist eine frei erhältliche Graphen Visualisierung und Analyse Software. Eine ausführlichere Erklärung erfolgt in Kapitel 4 »JGraph«.

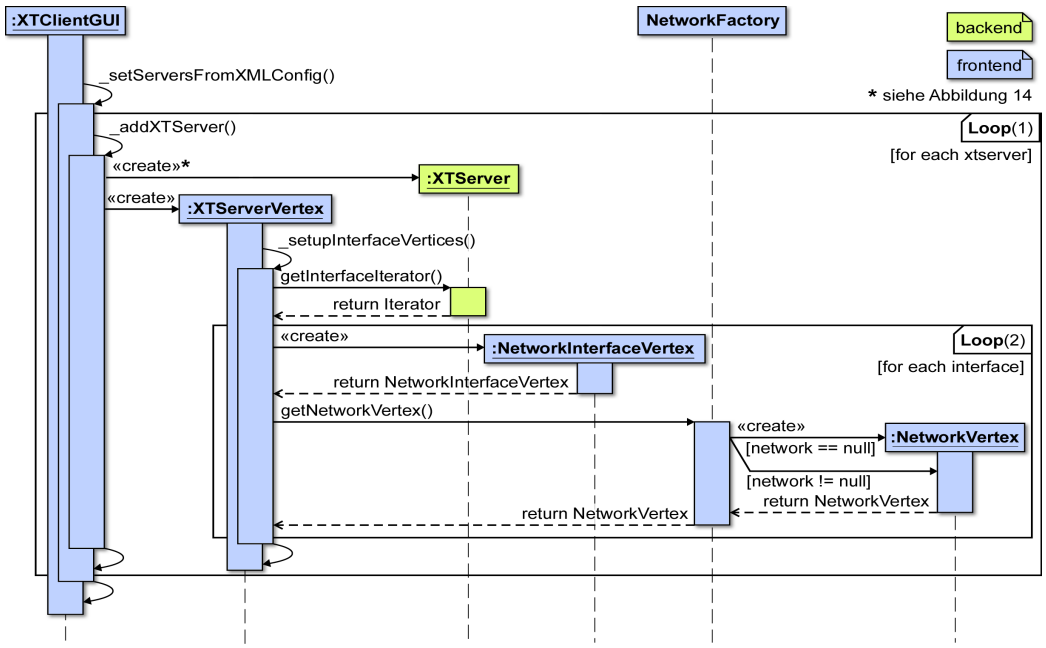


*Abbildung 4: JGraph-Objekt Erzeugungsdiagramm.*

Ein JGraph `XTServerVertex` seinerseits führt die Methode `setupInterfaces()` aus, welche die Interface-Liste des `backend XTServer` ausliest, um die korrespondierenden JGraph Interfaces zu erzeugen.

Mit jedem Interface wird dabei mittels der Klasse `NetworkFactory` auch das JGraph Netzwerk erzeugt. Wie in [PAE06], Seite 57 beschrieben, dient die Klasse `NetworkFactory` dazu, dass nur eine Netzwerkinstanz eines Netzwerkes erzeugt wird und alle Interfaces, die dieses Netzwerk bilden, damit verbunden sind. Die `NetworkFactory` verwaltet alle erzeugten Netzwerke und weiß somit, ob zu einer Netzwerkadresse bereits eine Netzwerkinstanz erzeugt wurde.

Diese oben genannten Strukturen und Abläufe gilt es bei einem Neuentwurf zu berücksichtigen und zu erhalten, damit die bisher funktionierenden Abläufe nicht erneut auf Korrektheit geprüft werden müssen.

## 2.5 Topologie-Anzeige des GUI

Die Darstellung der Topologie ist seit der ersten Version des RIP-XT von Pähler durch seinen JGraph-Entwurf festgelegt und nicht mehr verändert worden. Für die Visualisierung relevante Symbole sind: Router, die Interfaces eines Routers und Netzwerke, sowie

Kanten, die dazu dienen anzuzeigen, welches Interface mit welchem Netzwerk verbunden ist.

Der Designentwurf Pählers, siehe dazu Abbildung 5, setzt dies um, indem er für jeden Router ein Rechteck darstellt und zugehörige Interfaces innerhalb diese Rechtecks vertikal auflistet. Dadurch ergibt sich die Höhendimension eines Routers aus der Anzahl seiner Interfaces. Dies führt zu unterschiedlichen Größen für ein Router-Symbol. Die einzige darzustellende Information zu einem Router ist sein Name. Dieser wird in der rechten oberen Ecke des Router-Symbols angezeigt.

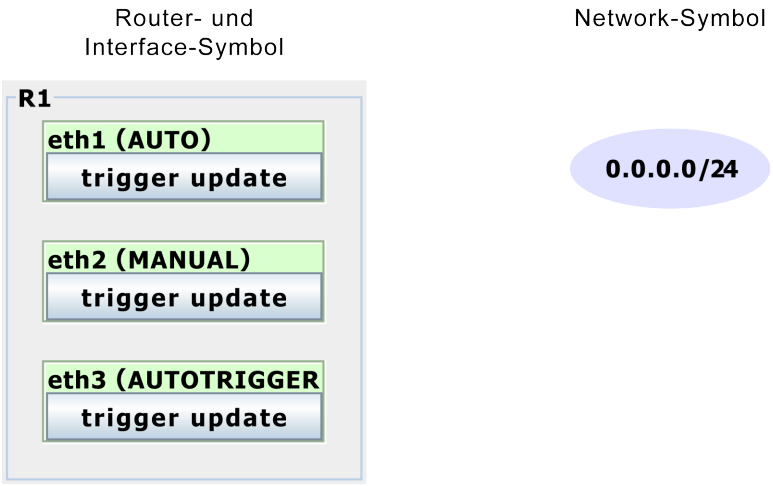


*Abbildung 5: Ursprüngliche RIP-XT Symbole.*

Ein Interface wird bei ihm durch ein Rechteck dargestellt, in das ein Button eingelassen ist. Das Interface-Symbol ist, wie oben beschrieben, wiederum in einem Router-Symbol platziert. Bei einem Interface wird der Name und der momentane Status angezeigt. Die Beschriftung eines Buttons ist für alle Interfaces gleich. Die Buttons sind funktionaler Art und dienen dazu, dem Benutzer manuell ein Update über ein Interface in ein Netzwerk zu versenden.

Netzwerke werden durch ein Oval repräsentiert und führen die Netzwerkadresse und die Netzwerkmaske in Slash-Notation als Beschriftung. Der RIP-XT zeigt alle in VNUML definierten Interfaces an, auch die Management-Interfaces. Dadurch werden auch die Netzwerke angezeigt, in denen die Management-Interfaces liegen. Diese Verbindungen sind nur für VNUML zur Steuerung der *VMs* wichtig und für die eigentlichen RIP- oder RMTI-Tests ohne Bedeutung. Deshalb kann auch auf sie verzichtet werden.

Zur Darstellung der Netzwerke ist noch anzumerken, dass sie durch eine Kante, die an der *Bounding Box*[22] des Ovales beginnt, mit einem Punkt auf der *Bounding Box* des Interface verbunden ist. Die Punkte des Kantenanfangs und des Endes sind durch die Position der beiden Elemente zueinander auf der Anzeigefläche gegeben. Dabei sind sie so angeordnet, dass die Kante die kürzeste Strecke einnimmt. Verschiebt man eines der Elemente, verschieben sich die beiden Punkte jeweils entlang ihrer *Bounding Box*, um diese kürzeste Distanz zu erhalten. Dadurch sind die Symbole auf der Topologie-Anzeige frei zu positionieren, ohne dass die Kanten selbst positioniert werden müssten.

Die Grundanordnung der Symbole wird durch einen Positionierungsalgorithmus berechnet. Dabei wird das erste Symbol, das immer ein Router ist, auf der Anzeige bei der Position (30,30) gesetzt und rechts daneben im Abstand von 40 Pixeln das erste Netzwerk, zu dem das erste Interface verbunden ist. Sollte der Router zu mehr als einem Netzwerk verbunden sein, so werden die weiteren Netzwerke mit 10 Pixel Abstand unter das erste Netzwerk gesetzt, bis alle verbundenen Netzwerke erzeugt sind. Dann wird der nächste Router platziert. Dieser wird rechts neben das erste Netzwerk gesetzt. Die bisher noch nicht erzeugten Netzwerke, zu denen er verbunden ist, werden wiederum rechts daneben als Liste von oben nach unten gesetzt.

Die so erzeugte Anzeige ist seltenst optimal oder dazu geeignet, den Benutzer schnell die Topologie erkennen zu lassen. Die einfache Platzierung ist aus gutem Grund gewählt worden, da eine aufwendige Berechnung der Platzierungen des Graphen nicht trivial ist und den Rahmen der Diplomarbeit damals gesprengt hätte. Der Benutzer kann/muss deshalb die Symbole selbst frei anordnen, um dieses Ziel zu erreichen.

Der Nachteil dabei ist, dass der Benutzer unter Umständen jedes Symbol neu setzen muss und so bei steigender Router-/Netzwerkanzahl auch mehr zu platzieren hat. Wird das GUI beendet, zeigt sich ein weiterer Nachteil, denn damit geht diese Positionierung verloren und muss vom Benutzer beim erneuten Start wieder geleistet werden. Dies ist auf lange Sicht nicht zufriedenstellend, da die Hauptaufgabe nicht im Anordnen des Graphen besteht, sondern in der Nutzung der Testfunktionen des GUI zu sehen ist. Deshalb ist der Wunsch nach einer Neugestaltung der Topologie-Anzeige gegeben, um die Benutzerfreundlichkeit des GUI in diesen Bereichen zu erhöhen.

22 *Bounding Box (2D)*, Rechteck, das eine Objekte in minimalster x- und y-Dimension umschließt.

# 3 Neukonzept der Topologie-Anzeige

Die Topologie-Anzeige wurde weitestgehend auf dem Standard-Szenario der CTI-Tests basierend entworfen, das aus fünf Routern und sechs Netzwerken besteht. Der damit verbundene Aufwand der Positionierung und die Übersicht über das gesamte Szenario sind hier noch möglich. Aktuelle Diplomarbeiten allerdings untersuchen den RMTI in großen VNUML Netzwerken mit zehn, zwanzig oder mehr Routern. Unter diesen Umständen steigt der Aufwand zur Platzierung enorm, aber auch die anschließende Übersicht leidet stark, da der Platz nicht mehr ausreicht, die gesamte Topologie innerhalb des Anzeigenbereichs vollständig anzuzeigen.

Als besonderer Nachteile der in Kapitel 2.5 »Topologie-Anzeige des GUI«, beschrieben Umsetzung werden die Symbole für die Router empfunden. Sie nehmen, je nach Interface-Anzahl, unterschiedlich viel Platz in der Anzeige ein. Ein Interface-Symbol selbst nimmt durch die Anzeige des Namens, des Status[23] und des Trigger-Buttons einen relativ großen Raum ein. Dadurch benötigt ein einzelner Router mit seinen Interfaces schon einen beträchtliche Platz auf des Anzeigenbereichs.

Der Trigger-Button besitzt aber noch einen weiteren Nachteil zusätzlich seiner Größe: Er bricht die Button-Erwartungskonformität. Das bedeutet: Da er rein Äußerlich einem Button gleicht, erwartet der Benutzer beim Klicken auf den Button eine visuelle Reaktion, die ihm signalisiert, dass der Button gedrückt wurde. Diese visuelle Reaktion bleibt jedoch aus und der Benutzer weiß somit nicht ob die von ihm gewünschte Aktion ausgeführt wurde.

Pähler, [PAE06] Seite 62f, hat indirekt schon auf dieses Problem hingewiesen. Bei dem Button handelt es sich um einen *JButton*, eine graphische Komponente der Java Swing Bibliothek, die Änderungen durch *MouseEvents*, wie *MouseOver* oder *MousePressed* abfängt und die Ereignisse visuell anzeigen kann. Dies ist hier aber nicht möglich, da JGraph eigens zum Zeichnen ein Objekt erzeugt und anschließend die Referenz auf das Objekt und damit das Objekt[24] selbst wieder löscht. Somit ist der Button, den der Benutzer sieht, kein echter *JButton*, sondern nur noch das gezeichnete Abbild eines *JButtons*, das sich dem entsprechend nicht wie ein Button verhalten kann.

Die Management-Netzwerke sind zwar ohne Relevanz, aber bedeuten einen unnötigen höheren Positionierungsaufwand, um ein Szenario zu ordnen. Außerdem steigt bei

23 *Status*: siehe, XT-Servers Interface Status, Kapitel 2.2 »XT-Server/-Client Architektur«, Seite 9.

24 Anmerkung: In Java wird ein Objekt nicht sofort gelöscht nachdem es nicht mehr referenziert wird. Das Objekt verbleibt im Speicher, es wird erst durch den automatischen Aufruf des *Garbage Collectors* endgültig aus dem Speicher entfernt und damit gelöscht.

höherer Anzahl an Routern wiederum auch der damit verbundene Platzbedarf. Aus diesem Grund wurde entschieden auf die Darstellung der Interfaces zu verzichten.

Da der Schwerpunkt der Diplomarbeit von D. Pähler nicht im Erstellen einer möglichst effizienten graphischen Topologie-Anzeige lag, sind die oben aufgeführten „Mängel“ auch nicht als Vorwurf zu verstehen, sondern als Anforderungen für einen Neuentwurf.

## 3.1 Anforderungen

Aus den oben angeführten Beobachtungen ergeben sich folgende Anforderungen bezüglich des Konzepts für den RIP-XT:

1. Die Grundeinteilung in Router, Interface und Netzwerk wird beibehalten.

2. Ein Symbol für einen Router soll möglichst minimalen Raum einnehmen und gleichzeitig die maximalen Erkennungsmerkmale aufweisen, um ihn als Router zu identifizieren.

3. Ein Interface soll minimalen Raum einnehmen und seine Zugehörigkeit zu einem Router eindeutig erkennen lassen. Für die Vorgabe des minimalen Raums müssen die Informationen eines Interfaces möglichst kompakt dargestellt werden.

4. Die Management-Interfaces sollen nicht mehr angezeigt werden.

5. Eine vom Benutzer vorgenommene Positionierung soll zu speichern sein und beim erneuten Laden berücksichtigt werden.

6. Die Positionsinformationen sollen entweder separat in eine Konfigurationsdatei oder direkt in die Szenario-Datei gespeichert werden.

Zusätzliche Anforderungen:

7. SSH-Terminal-Verbindung zu den Routern aus der Topologie-Anzeige.

8. Abspeichern der Topologie-Anzeige als PNG-Bild.

## 3.2 Entwurf

Beim Entwurf wurden verschiedene Aspekte berücksichtigt, unter anderem für welche Benutzergruppe der RIP-XT gedacht ist. Hierzu wurde schon von Pähler das Konzept stark umrissen. Die dort aufgeführten Eckpunkte behalten auch bei der Neukonzeption ihre Gültigkeit.

So wird der RIP-XT von Informatikern genutzt, die sich verstärkt mit der Thematik der Rechnernetze beschäftigen und so ein gewisses Vorwissen zu RIP und RMTI besitzen. Des Weiteren kann davon ausgegangen werden, dass der Benutzer mit Netzwerktopologie-Darstellungen aus der Fachliteratur vertraut ist. Die dort gebräuchlichste Darstellung für Netzwerkkomponenten ist durch *Cisco*[25] *Network Symbols* [CIS] geprägt.

Von diesem Standpunkt aus bietet es sich an, ein Router-Symbol durch ein entsprechendes Cisco-Symbol (Abbildung 6) darzustellen, um den Wiedererkennungswert für einen möglichst großen Kreis von Fachleuten zu gewähren. Außerdem bietet das Symbol den Vorteil, dass es auch noch in sehr kleinen Darstellung für den Benutzer erkennbar ist. Als Router-Symbol wurde eine eigene Umsetzung der Cisco-Symbole auf 60x50 Pixel erstellt.

*Abbildung 6: Offizielle Cisco Router Symbols.*

Das Cisco Router-Symbol sieht keine explizite Darstellung der Interfaces vor. Ein Interface wird implizit durch eine Verbindung zu einem anderen Element dargestellt, welches nur durch eine Kante angezeigt wird. Da es im RIP-XT aber von Bedeutung ist, jedes Interfaces eines Routers zu sehen und die zusätzlichen Informationen dazu zu erhalten, musste eine äquivalente Darstellung für diese Elemente gefunden werden.

Es bot sich an, eine Interface-Darstellung aus der Studienarbeit von Alexander Osterberg zu wählen, siehe dazu Abbildung 7. Bei dieser Darstellung ist ein Interface durch ein kleines Rechteck repräsentiert, das sich innerhalb eines festen Rahmens um ein Element verschieben lässt. Die Zugehörigkeit eines Interfaces zu einem Element wird durch eine Kante zwischen dem Interface und einem Router oder Host signalisiert.

Diese Darstellung ist für den RIP-XT überarbeitet worden. Um Kanten zwischen den Interfaces und den Routern zu vermeiden, wurde sich des Gestaltungsprinzip des „Ge-

25 *Cisco Systems,* renommierter Router und Switch Hersteller, Cisco Systems, Inc.

setzes der Nähe“ [DAH06] bedient. Dies besagt, dass selbst unterschiedliche Elemente als Einheit empfunden werden können, sofern die Distanz zwischen den Elementen gering ist. Unser Gehirn gruppiert solche Elemente automatisch. Um diese Nähe zu erreichen, wurde in der Umsetzung die Bewegungsfreiheit der Interfaces auf eine Ellipse um den zugehörigen Router eingeschränkt. Durch diese hinreichende Zuordnung konnte auf zusätzlichen Kanten in der Darstellung verzichtet werden.

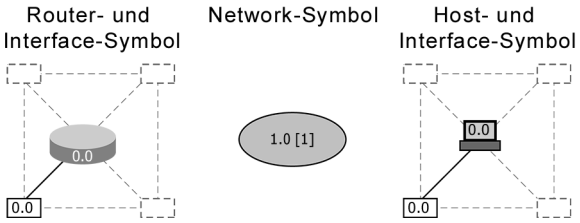


*Abbildung 7: Symbol-Darstellung beim JRoutingSimulator.*

Als Name auf dem Interface wird nur noch ein fortlaufend nummerierter Interface Bezeichner ($eth_0$ – $eth_n$) angezeigt. Um das Symbol möglichst klein zu halten, wurde die vorherige Anzeige des Status hinter dem Bezeichner weggelassen. Stattdessen wurde eine farbliche Anzeige des Status gewählt.

Die Trigger-Funktion, die ursprünglich durch einen zusätzlichen Button angezeigt wurde, ist nun durch einen Doppelklick auf das Symbol selbst realisiert. Dadurch entfällt der zusätzliche Platz für einen Button und das Symbol beschränkt sich lediglich auf die kurze Länge des Interface-Bezeichners, der sich in der Regel auf vier Schriftzeichen beläuft.

Für die farbige Status-Kodierung wurde auf Signalfarben wie Rot oder Grün verzichtet, da die Status nicht unbedingt mit den Assoziation wie „Achtung“ oder „In Ordnung“ zu verbinden sind. Eine eindeutige farbliche Unterscheidung der Status reicht aus und lässt so ein gesamtheitliches farbliches Konzept zu.

Der Status AUTO wurde farblich nicht hervorgehoben, sondern mit dem Farbton des Router-Symbols angezeigt, da dieser Status dem Normalzustand entspricht, wenn RIP oder RMTI mit den Standard-Timern arbeitet. Die Status für MANUAL und AUTO-TIGGERED wurden durch Farben hervorgehoben.

MANUAL wird mittels gelb-orange kodiert, während AUTO-TRIGGERED durch violett angezeigt wird. Die beiden Farbtöne wurden auf Grund ihrer deutlichen Unterscheidung gewählt. Zusätzlich passen sie sich in die allgemeine warme Farbgestaltung

ein. Durch warme Grautöne und cremefarbige Töne soll die sonst eher technische Darstellung etwas aufgelockert werden.

Die Abbildung 8 zeigt das neue Router-Symbol mit Interfaces-Symbolen in den drei möglichen Interface-Zuständen. Dieses Kapitel beschreibt den Aspekt des visuellen Entwurfs der Topologie-Anzeige und seiner konkreten Umsetzung in Symbole.

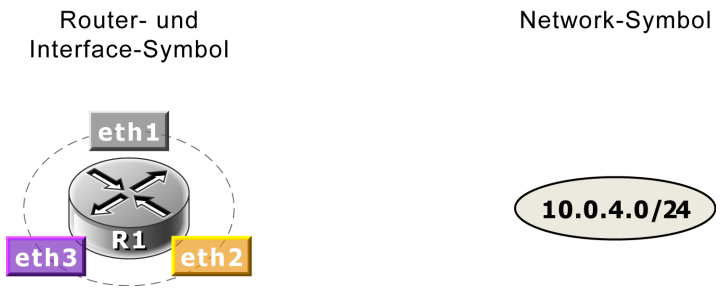


*Abbildung 8: Überarbeitete RIP-XT Symbole.*

Ein anderer Aspekt ist die Umsetzung des visuellen Entwurfs in eine Programmstruktur. Dafür wurde die Programmiersprache Java benutzt. Zusätzlich wurde das Java API JGraph verwendet, die schon in der Erstumsetzung von Pähler für die Graphendarstellung ausgewählt wurde.

Gründe für die Auswahl des APIs sind und waren einerseits die Qualität des APIs, das durch Projektreferenzen von Firmen, wie DaimlerChrysler, Oracle und viele mehr, bezeugt wird. Diese Qualität spiegelt sich aber auch in der fortlaufenden Weiterentwicklung des APIs. Sie bietet somit eine Vielseitigkeit in den Anwendungsmöglichkeiten.

Anderseits bietet das API die Möglichkeit eigene Modifikationen zu erstellen und trotzdem auf eine Basis von funktionierenden Hilfsmitteln zugreifen zu können. Im nachfolgenden Kapitel wird JGraph selbst näher vorgestellt.

# 4 JGraph

JGraph ist eine Java Bibliothek, die sich auf die Visualisierung von Graphen, Interaktion mit ihnen, automatische Layouts und Graphen basierte Performance Analysen spezialisiert hat. Abgesehen von der Visualisierung lassen sich auch Kontext oder Metadaten zu den Elementen organisieren.

JGraph war ursprünglich in zwei Versionen erhältlich: Als kostenpflichtige Bibliothek mit umfassenden Layout Möglichkeiten und als „abgespeckte“ Version ohne diese automatischen Layout-Funktionen unter der LGPL[26]. Seit der Version 5.13.0.3 sind die kostenpflichtige Layout-Anteile ebenfalls freigegeben und die komplette Bibliothek ist nun unter der *Revised BSD*[27] *Open Source License* und der *JGraph License Version 1.1* frei erhältlich.

Die Software kann unter der BSD Lizenz als Quellcode oder als Binärdateien kopiert, verändert und weiterverbreitet werden, solange der vorherige Copyright Vermerk der Software nicht entfernt wird. Für die vorliegende Diplomarbeit wurde der Quellcode nicht verändert, sondern es ist lediglich von den Binärdateien Gebrauch gemacht worden. Die Bibliothek ist einerseits zur Entwicklung benötigt worden, andererseits für das fertige RIP-XT Programm als Laufzeitbibliothek in die ausführbare JAR-Datei beigefügt. Dadurch erspart man sich die bisherige Handhabung, die JGraph-Bibliothek in das gleiche Verzeichnis wie die RIP-XT JAR-Datei zu legen, um den RIP-XT starten zu können.

Die aktuelle Version von JGraph ist 5.13.0.3 und liegt seit dem 29. Januar 2010 vor. Die Diplomarbeit selbst wurde noch in JGraph 5.10.0.0 begonnen und dann aktualisiert auf die neueste Version. Von daher wurden die umfassenden Layout Funktionen bei der Ausarbeitung nicht berücksichtigt.

## 4.1 Einstieg

Um das API nutzen zu können, sollte die aktuellste Version der JGraph 5 Bibliothek von der JGraph Webseite[28] heruntergeladen werden. Außerdem muss die Java-Version 1.4 oder höher auf dem System installiert sein.

Ebenfalls hilfreich ist das offizielle Handbuch [JGM]. Darin werden die Installation und die wichtigsten Grundlagen erklärt. Es bietet einen guten Einstieg in das Arbeiten

26 *LGPL*, Lesser General Public License. Siehe: www.gnu.org/licenses/lgpl.html.
27 *BSD*, Berkeley Software Distribution. Siehe: www.opensource.org/licenses/bsd-license.php.
28 *JGraph 5*, Quelle: www.jgraph.com/pub/jgraph-latest.jar.

mit JGraph. Einziger Nachteil an dieser Dokumentation ist, dass zwar teilweise wichtige Konzepte zur Programmierung erläutert werden, diese aber über das gesamte Handbuch verteilt sind und sich so nur schwer für einen Neueinsteiger ein gesamtheitliches Bild schaffen lässt. Aus diesem Grund wird hier der Versuch gestartet, diese Einstiegspunkte nochmals hervorzuheben und als Grundkonzept zur Erstellung eigener JGraph-Anwendungen darzustellen.

## 4.2 JGraph Grundlagen

Vorab ist folgendes zu JGraph anzumerken: Es ist eine Spezialisierung der *JComponent*-Klasse, die eine Swing[29] Basisklasse für viele grafischen Elemente in Java ist. Eine der Unterkomponenten von *JComponent* ist *JTree*. Sie dient zur Darstellung Graphen-basierter Strukturen. JGraph ist in seinem strukturellen Aufbau und seiner Klassennamensgebung stark an *JTree* angelehnt, ebenso bedient es sich des MVC[30] Pattern von Swing.

### 4.2.1 MVC Pattern

Das MVC Pattern ist eine Systemarchitektur, bei der *Model*, die Daten und *View*, die visuelle Repräsentation der Daten, getrennt sind. Interaktionen zwischen *View* und *Model* werden über den *Controller* gesteuert. Eine Benutzerinteraktion wird vom *Controller* ausgewertet. Dabei bestimmt er z.B. welche Objekte davon betroffen sind und leitet dies an den *View* weiter. Die durch die Interaktion verursachten Änderungen werden dem *Model* mitgeteilt und führen zu einer Aktualisierung der Daten. Der *View* erstellt die Objekte, die durch den Benutzer ausgewählt wurden, und ruft für jedes ausgewählte *View*-Objekt die zugehörigen Daten des *Model* auf. Die neu erstellten *View*-Objekte werden entsprechend ihrer visuellen Repräsentation gerendert. Abbildung 9 zeigt den Aufbau und die Interaktionen des Java MVC-Pattern.

Vorteil dieses Entwurfmusters ist, dass zu einem *Model* verschiedene *Views* verwendet werden können, ohne die Daten mehrfach zu halten. Die visuelle Repräsentation kann dadurch jederzeit verändert werden und so verschiedene Sichten auf ein und dieselben Daten gegeben werden. Des Weiteren ist der Speicheraufwand verringert, da die *View*-Objekte nur kurzzeitig zur Darstellung benötigt werden und anschließend wieder verworfen werden. Für die Zeichenroutine bedeutet dies, dass nur die durch Veränderung betroffenen Objekte neu gezeichnet werden müssen.

29 *Swing*, Teil der Java Foundation Classes (JFC), ein API für die Entwicklung von GUIs.

30 *MVC,* Model/View/Controller Pattern. Ein Entwurfsmuster das ursprünglich 1978 am Xerox PARC für die Programmiersprache Smalltalk-80 entwickelt wurde [MVC].

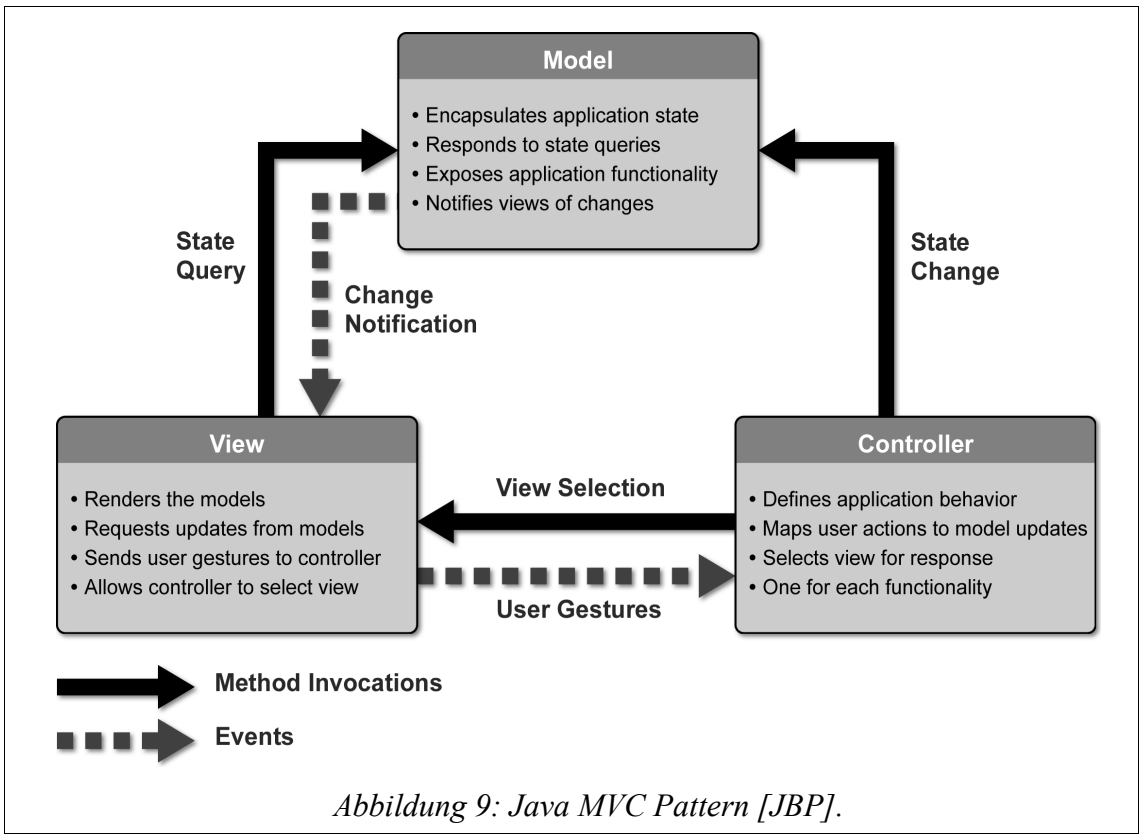


*Abbildung 9: Java MVC Pattern [JBP].*

Die Dreiteilung spiegelt sich auch im Klassenaufbau von JGraph wieder. Eine JGraph Instanz benötigt ein *GraphModel*, einen *GraphLayoutCache* und eine *GraphUI*. Der *GraphLayoutCache* entspricht dem *View*, weshalb er auch ursprünglich *GraphView* hieß. Er wurde aber in Analogie zum *AbstractLayoutCache* von *JTree* in *GraphLayoutCache* umbenannt, um auch in den Bezeichnungen dem Konzept von *JTree* zu folgen.

Die Klasse `GraphUI` hat die Funktion des Controllers. Abbildung 10 zeigt das MVC Pattern bei JGraph und die zugehörigen Klassen, mittels welcher das Pattern umgesetzt ist. In einem Punkt weicht JGraph aber von dem MVC Pattern ab, wie aus dem Klassendiagramm zu ersehen ist: Der *View* wird nicht direkt vom *Controller* referenziert, sondern nur über die JGraph Instanz. Dies geschieht, um den Zustand eines *Views* zu bewahren auch, wenn der *Controller* verändert wird.

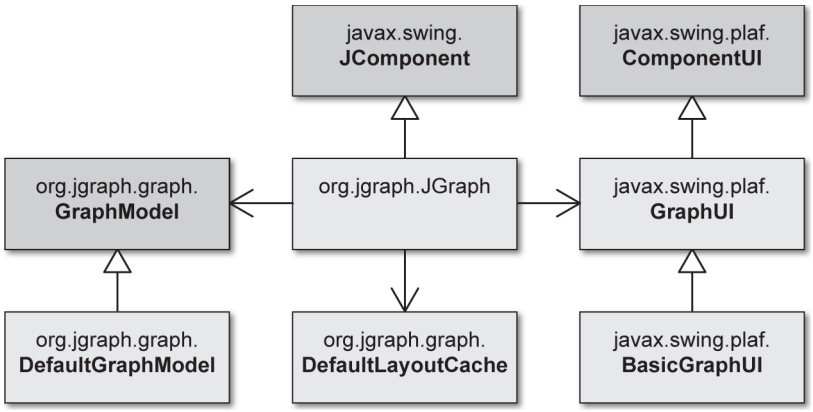


*Abbildung 10: JGraph MVC Klassenaufbau [JGM].*

In den Folgenden Unterkapiteln werden die drei Komponenten und ihre Funktionalität noch etwas näher beschrieben. Dieser Überblick soll dazu helfen die Grundstruktur von JGraph besser zu übersehen.

### 4.2.2 Das GraphModel

Das *GraphModel* ist dafür zuständig, die Graphenstruktur in irgendeiner Form umzusetzen und zu organisieren. Um die Struktur aufzubauen bedient es sich der `GraphCell`-Objekte. Die Klasse `GraphCell` ist selbst nur ein Interface in JGraph und wird in der Klasse `DefaultGraphCell` implementiert. Die beiden einzigen Methoden, die das Interface fordert, sind die `getAttributes()` und `setAttributes()`. Die Methode `getAttributes()` gibt die Attribute einer *Cell* in Form einer *AttributeMap*[31] zurück. Umgekehrt wird die gesamte *Map* durch die Methode `setAttributes()` neu gesetzt.

Die Verwendung der `DefaultGraphCell` ist nicht zwingend notwendig für das *GraphModel*. Es können allgemeine Java-`Objects`[32] zur Erstellung der Graphenstruktur durch das *GraphModel* genutzt werden. Dies ist möglich, da alle Methoden des *GraphModels* und des *GraphLayoutCaches* als Teil des Design-Vertrags von JGraph nur `Object` als Parameter oder Rückgabewert fordern. Für die meisten Anwendungen bietet es sich aber an die *DefaultGraphCells* zu benutzen und durch Spezialisierung dieser, eine Klasse zu erstellen, die den eigenen Anforderungen entspricht. Denn dadurch muss die Traversierung innerhalb des *Models* nicht selbst erstellt werden, sondern es kann auf die vorhandenen Methoden zurückgegriffen werden.

31 *AttributeMap*, besteht aus einer Reihe von *Name/Value*-Paaren, in denen die Konfiguration einer *Cell* gespeichert wird.
32 *Object*, bezieht sich hier auf die Basisklasse von Java, von der alle anderen Klassen abgeleitet sind.

Um die Graphenstruktur aufzubauen und zu verändern, besitzt das *Model* drei Methoden: `insert()`, `edit()` und `remove()`. Diese Methoden werden aber nur in bestimmten Situation verwendet, auf die Hintergründe wird im Folgenden noch eingegangen. Grundsätzlich sollte der Aufbau der Graphenstruktur mittels der Methoden des *GraphLayoutCache* durchgeführt werden.

Wie bereits erwähnt, nutzt JGraph die `DefaultGraphCells` zum Aufbau der Graphenstruktur. Um eine Graphenstruktur abzubilden wird aber mehr als nur eine Art von Elementen benötigt. Deshalb besitzt JGraph drei *Cell* Typen: *Vertex*, *Edge* und *Port*. Ein *Vertex* wird durch die Klasse `DefaultGraphCell` repräsentiert. Die beiden Klassen `DefaultEdge` und `DefaultPort` sind Spezialisierungen dieser Klasse.

Ein *Vertex*, bzw. die `DefaultGraphCell`, repräsentiert einen Knoten. Alle Eigenschaften eines Knoten, wie Größe, Position, Farbe, etc., werden in einer *AttributeMap* gespeichert. Die einzelnen Eigenschaften sind im Detail im Handbuch [JGM] hinreichend beschrieben.

Der Zugriff auf die *AttributeMap* einer *Cell* wird nicht direkt über das *Cell*-Objekt realisiert, sondern über die Klasse `GraphConstants` und der Auswahl einer Get-/Set-Methode eines Attributes erreicht. Dies soll einem möglichen Fehler in der Setzung der *Name/Value*-Paaren vorbeugen.

Zwei Knoten werden über eine *DefaultEdge* miteinander verbunden. Eine *Edge* ist intern gerichtet, dass heißt, es muss eine *Source* und ein *Target* angegeben werden. Dabei wird eine *Edge* nie direkt mit einer *Cell* verbunden, sondern über einen *DefaultPort* einer *Cell*. Der *Port* ist als Ankerpunkt einer *Cell* zu verstehen. Mit ihm können keine oder beliebige viele *Edges* verbunden sein.

Das *DefaultGraphModel* baut aus diesen Komponenten das Graphenmodell auf. Die Elemente werden in einer *ArrayList* gespeichert. Sie bildet die Wurzelebene des *Models*. Wird ein *Cell*-Objekt eingefügt, wird es an das Ende der Liste eingefügt. Die Reihenfolge des Einfügens bestimmt über die Position in der Liste und damit auch über Z[33]-Sichtbarkeit eines Elements auf der Anzeige. Auch *Edges* werden auf dieser Ebene in der Liste gespeichert. Anders sieht es mit den *Ports* aus. Sie werden immer einer *Cell* hinzugefügt und sind somit Kinder einer *Cell*. Dadurch besitzt ein Element wiederum selbst eine *ArrayList*, die seine Kindern beinhaltet. In Abbildung 11 ist eine solche mögliche Struktur abgebildet.

33 *Z-Achse*, obwohl die Zeichenfläche nur zweidimensional ist, legt JGraph jedes Objekt auf einen Layer, entsprechend ihrer Reihenfolge. Zuletzt eingefügte Objekte, liegen auf einem höheren Layer als die Vorhergehenden und verdecken somit diese.

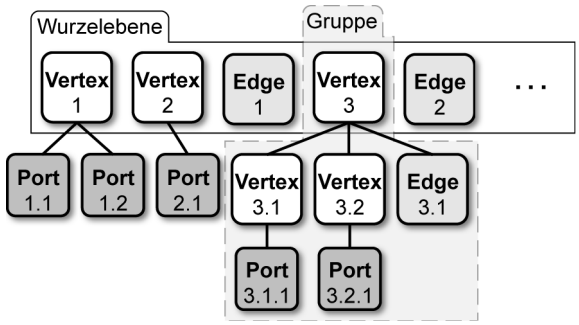


*Abbildung 11: Aufbau der Graphenstruktur im GraphModel.*

Weitere Verzweigungen in der Baumstruktur können aber auch durch Gruppierung von *Cells* entstehen. Um *Cells* in einer Gruppe zusammenzufassen, werden sie einer *Cell* als Kinder angefügt. Eine solche Gruppe kann wiederum selbst aus *Cells* bestehen, die weitere *Cell*-Gruppen beinhalten. Daraus resultieren beliebige Baumstrukturen in den Objekten der Wurzelliste. Dies ist zu berücksichtigen, wenn Objekte innerhalb des *Models* gesucht werden. Die Klasse `DefaultGraphModel` bietet Methoden zur Navigation innerhalb der Listen und zum Traversieren der Unterbäumen. Diese Methoden funktionieren mit den JGraph *Cells*. Möchte man eignen *Cell*-Objekte benutzen, müssen diese Methoden nach empfunden werden. Die Methoden des `DefaultGraphModels` sind ausführlich im JGraph-Handbuch [JGM] (Kapitel 2.4.2.4) beschrieben.

Ein besonders wichtiger Punkt beim Arbeiten mit dem *Model* ist, dass nach anfänglichem Erstellen, Änderungen wie erwähnt nicht mehr direkt im *Model* durchgeführt werden sollten, sondern über den *GraphLayoutCache* und seine Methoden.

Dies hat den Hintergrund, dass Änderungen, die direkt im *Model* ausgeführt werden, automatisch für jeden *View* gelten und somit der Ansatz unterlaufen wird, zu einem *Model* mehrere *Views* verwenden zu können. Ein weiterer Punkt ist: Jede Änderung über den Cache führt zu einer Zustandsänderung, die an das *Model* weitergeleitet wird. Das *Model* führt eine Aktualisierung durch und erzeugt einen Undo-Event, das durch einen Undo-Manager festgehalten werden kann. Wird direkt auf dem *Model* gearbeitet, kann keine Zustandsänderung bemerkt werden und dementsprechend kein Event erzeugt werden. Es ist immer der Weg über den Cache zu wählen und nur in Ausnahmefällen eine direkte Manipulation des *Models* zu bevorzugen.

Eine Ausnahme stellen komplexe Transaktionen dar. Sie ermöglichen es mehrere Einfüge-, Editions- oder Entfernungs-Operationen in einem Schritt auszuführen, selbst wenn diese in verschachtelten Methodenaufrufen stattfinden. Eingeleitet wird eine sol-

che Transaktion mittels einer `beginUpdate()` Anweisung. Alle Transaktionen werden erst dann durchgeführt, wenn eine `endUpdate()` Anweisung erfolgt. Zwischen ein Beginn-/End-Konstrukt können weitere Konstrukte gesetzt sein. Das *Model* wird erst aktualisiert, wenn das letzte `endUpdate()` erreicht ist.

Eine weitere Ausnahme besteht dann, wenn *Cells* unsichtbar gemacht wurde. Eine Manipulation der unsichtbaren *Cell* durch den *GraphLayoutCache* führt automatisch zu einer Änderung der Sichtbarkeit. Soll dies verhindert werden, müssen die Änderungen direkt im *Model* vorgenommen werden.

Eine solche Steuerung des Graphen ist nur über das *GraphModel* möglich, der *GraphLayoutCache* hat dieses Anweisungskonstrukt nicht. Programmcode 1 zeigt als verkürztes Beispiele solche paarweise auftretende Transaktionskonstrukte. Die *`count`*-Variable in den Kommentaren verdeutlicht, dass jedes `beginUpdate()` auch ein entsprechendes `endUpdate()` benötigt, da ansonsten das Programm auf dieses Routineende wartet, bevor es die Ausführungsschritte durchführt.

```
01 graphModel.beginUpdate();            // count = 1
02    // do some inserts, edits or removes.
03
04    graphModel.beginUpdate();         // count = 2
05    // do some inserts, edits or removes.
06
07    graphModel.endUpdate();           // count = 1
08
09 graphModel.endUpdate();              // count = 0
10 // graph will be updated now!
```

*Programmcode 1: Komplexe Transaktion im GraphModel.*

### 4.2.3 Der GraphLayoutCache

Der *GraphLayoutCache* definiert den *View* des Graphen. In ihm wird die Zuordnung eines *Cell*-Objekt zu seinem zugehörigen *CellView* gewährleistet. Dies wird mit Hilfe der *CellViewFactory* realisiert. In ihr wird die Methode `getVertexView()` implementiert. Die Methode beschreibt konkret zu welcher *Cell*-Klasse welche *CellView*-Klassen zurückgegeben werden soll. Deshalb wird einem *GraphLayoutCache* bei seiner Erzeugung sowohl das *Model* als auch die *CellViewFactory* als Parameter übergeben.

In den meisten Benutzerinteraktionen erhält man durch Selektion eines Objekts den zugehörigen *CellView* direkt. Die Instanz der Klasse `JGraph` bietet dazu Methoden, den *View* zu einer Koordinate zu ermitteln. Anders ist es, wenn es die Situation erfordert,

dass für den Benutzer im Hintergrund Objekte ausgewählt werden. Dann hat man meist eine Liste von *Cells* und benötigt die zugehörigen *Views*.

Dieses Mapping von *GraphModel* zu *GraphView* ist nur durch die Methode `getMapping()` des *GraphLayoutCaches* möglich. Dabei kann entweder zu einem einzelnen *Cell* der *View* ermittelt werden oder für eine Liste von *Cells*. Das umgekehrte Mapping, von einem *View* zu einer *Cell* ist direkt durch den *View* möglich. Er bietet dazu die Methode `getCell()` an.

Wie bereits im Kapitel 4.2.2 »Das GraphModel« erwähnt, besitzt der *GraphLayoutCache* analog zum *GraphModel* die Methoden `insert()`, `edit()` und `remove()`. In den meisten Fällen sind diese Methoden zu verwenden, um die Graphenstruktur aufzubauen oder zu verändern. Der *GraphLayoutCache* sorgt dafür, dass die Änderungen dem *Model* mitgeteilt werden. Es aktualisiert sich dementsprechend.

Mit `insert()` werden alle Objekte zunächst in das *Model* eingefügt. Dazu wird zuerst eine *Cell*- oder *Edge*-Objekt erzeugt, die Attribute werden mittels der `GraphConstants` gesetzt und anschließend wird das Objekt mit `insert()` in das *Model* eingefügt. Um nicht jedes Objekt einzeln hinzuzufügen, kann auch eine Liste von Objekten übergeben werden, die eingefügt werden sollen.

Soll ein bereits eingefügtes Objekt geändert werden, muss dies mittels eines `edit()`-Aufrufs geschehen. Die genaue Vorgehensweise wird noch im Kapitel 4.3.4 »Attributsänderungen« beschrieben, wenn die beiden Möglichkeiten des Editierrens einer *Cell* vorgestellt werden.

Mittels `remove()` lassen sich Objekte auch wieder aus dem *Model* entfernen. Die Methode arbeitet, wie `insert()`, entweder mit einer einzelnen *Cell*, die entfernt werden soll, oder einer Liste von *Cell*-Objekten. Dabei werden auch die Graphenstrukturen berücksichtigt. Wenn z.B. eine *Cell* einer Gruppe entfernt wird, dann reorganisiert JGraph die verbleibende Baumstruktur. Das Entfernen des Wurzelelements einer Gruppe entspricht der Auflösung dieser Gruppe.

### 4.2.4 Das GraphUI

Die Klasse `GraphUI` entspricht dem Controller. Sie ist lediglich ein Interface, das in der Klasse `BasicGraphUI` implementiert ist. Die Klasse stellt alle Funktionalität zur Verfügung, um Objekte in JGraph darzustellen, sie zu aktualisieren und mit ihnen zu interagieren.

Wichtiger Bestandteil hierzu ist die innere *MouseListener*-Klasse, da die gängigste Interaktion zwischen Benutzer und Graph mittels der Maus stattfinden. Der verwendete

*MouseListener* wird durch die Methode `createMouseListener()` für die `GraphUI` festgelegt. Die Standardimplementation des *MouseListener* behandelt die typischen Aktionen wie *mouseDragged*, *mouseMoved*, *mousePressed* und *mouseReleased*[34].

Analog hierzu wird auch der *KeyboardListener* der `GraphUI` durch die Methode `createKeyboardListener()` festgelegt. JGraph besitzt eine Reihe von eigenen Keyboard-Aktionen, wie z.B. das Selektieren aller Objekte durch die Tastenkombination Strg-A. Die *GraphUI* ist eine der wichtigsten Klassen, an der meist eigene Spezialisierungen vorgenommen werden müssen, damit eine eigene Anwendung in gewünschter Erwartung funktioniert. Das bedeutet, dass die verschiedenen *Listener* dem gewünschte Verhalten angepasst werden müssen.

## 4.3 Klassen-Spezialisierung

Um eine eigene JGraph-Anwendung zu erstellen, wird man es meistens nicht vermeiden können, die oben erwähnten Klassen zu erweitern und an die eigenen Bedürfnisse anzupassen. Im Folgenden werden die drei Bereiche *GraphModel*, *GraphLayoutCache* und *GraphUI* durchgegangen. Dabei soll erklärt werden, welche Spezialisierungen notwendig sind, um JGraph in seiner Grundfunktionalität zu verändern und wie die Grundstrukturen der erweiterten Klassen auszusehen haben oder welche Methode zu überschreiben von Interesse sein könnte.

### 4.3.1 Die GraphModel-Klasse

Das *Model* bildet die logische Struktur des Graphen ab, also die Relationen zwischen den Objekten, und hält Benutzerdaten in den Objekten. Die Klasse `DefaultGraphModel` ist die Grundimplementation des Interfaces `GraphModel` in JGraph und eignet sich als *Model* für den meisten Anwendungen.

Wie bereits erwähnt, benötigt das *Model* Objekte um Graphenstrukturen aufzubauen. Für den Neuentwurf der Topologie-Anzeige mussten drei eigene *Vertex*-Typen spezialisiert werden, damit diese auch gesondert behandelt werden konnten. Die Objekttypen für die *Edge* und *Port* wurden von JGraph unverändert übernommen. Für die *Vertex*-Typen wurden von der *DefaultGraphCell*-Klasse eigene Klassen abgeleitet. Anhand eines kleinen Beispiels sollen nun ganz konkret die Schritte gezeigt werden, die dazu nötig sind. Dazu legt man eine eigne Klasse an, z.B. `MyCell`, die sich von *DefaultGraphCell* ableitet. Dem Klassenkonstruktor kann optional als Parameter ein Objekt des Typs `Ob-`

34 Näherer Informationen zur Funktionalität und Gebrauch von Java MouseListener [JSE6a]: Java SE 6 API: MouseListener.

ject mitgegeben werden. Dieses Objekt ist das sogenannte *userObject*[35] und beinhaltet die Benutzerdaten. In welcher Struktur diese Daten vorliegen und wie der Zugriff auf sie erfolgt, bleibt dem Anwendungsprogrammierer überlassen. Die einzige Methode, die die Klasse implementiert haben muss, ist die Methode `toString()`. Sie gibt eine Zeichenkette zurück, welche JGraph als Label[36] der *Cell* anzeigt.

Diese Spezialisierung verändert am Verhalten der *DefaultGraphCell* erst einmal nichts Grundlegendes. Man nutzt nur den Umstand, dass sie nun jede Instanz der Klasse vom Typ `MyCell` ist. Dies spielt im Weiteren eine Rolle, wenn die Interaktion mit Objekten behandelt wird. Bei diesen Interaktionen ist es oft wünschenswert, unterschiedliche Aktionen für unterschiedliche Objekte auszuführen oder zu verhindern. Programmcode 2 zeigt den Grundaufbau einer einfachen Erweiterung der `DefaultGraphCell`-Klasse.

```
01 public class MyCell extends DefaultGraphCell{
02
03 transient protected Object userObject;
04
05    public MyCell(){
06       this(null);
07    }
08
09    public MyCell(Object userObject){
10       this.userObject = userObject;
11    }
12
13    public String toString(){
14       if(userObject != null)
15          return this.userObject.toString();
16       else
17          return “”;
18    }
19 }
```

*Programmcode 2: Grundaufbau einer eigenen Cell-Klasse.*

Die Methode `toString()` der Klasse ist nur beispielhaft implementiert. Um eine sinnvolle textuelle Repräsentation eines erzeugten Objekts zu erhalten, müsste hier das `userObject` in seiner `toString()`-Methode einen Namen oder Ähnliches als Zeichenkette zurückgeben.

35 Die Klasse GraphConstants biete den Zugriff auf das Benutzerdaten-Objekt über `getValue()` und `setValue(Object)`.

36 *Label* bezeichnet hier ein Schriftzug der innerhalb der *Cell*-Begrenzung angezeigt wird und durch Layout-Angaben positioniert werden kann.

Diese Klasse besitzt noch keine visuelle Repräsentation oder Renderer, der diese umsetzt. Das nächste Kapitel erläutert diese Schritte.

### 4.3.2 Die GraphLayoutCache-Klasse

Wie bereits erwähnt, gewährleistet der GraphLayoutCache die Zuordnung von *Model* zu *View*. Dazu muss für jeden *Cell*-Typ ein *CellView* definiert sein und ein Renderer, der basierend auf dem *CellView* diese zeichnet.

Der zur Klasse `DefaultGraphCell` zugehörige *View* ist in der Klasse `VertexView` definiert. Abbildung 12 aus dem Handbuch [JGM] zeigt die komplette `CellView` und `CellViewRenderer` Hierarchie von JGraph. Möchte man allgemein eigene *CellViews* erstellt, können diese von der Klasse `AbstractCellView` spezialisieren werden. Um den *Views* für eigene *Vertex*-Objekte zu erstellen, kann dieser direkt von der Unterklasse `VertexView` spezialisiert werden.

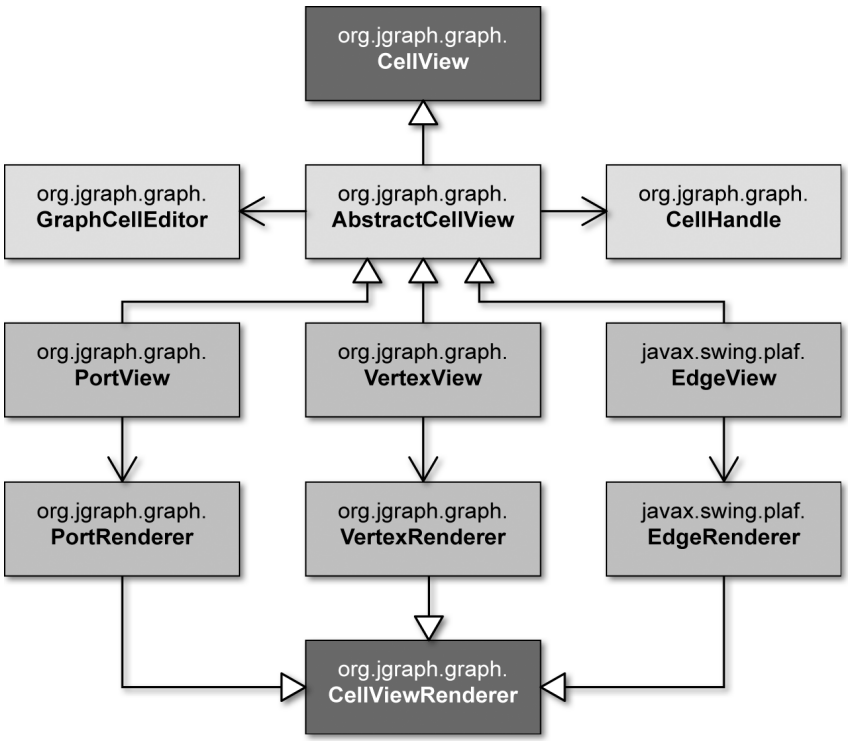


*Abbildung 12: CellView und CellViewRenderer Hierarchie.*

Um den *View* für die Beispielklasse `MyCell` zu erstellen, wählt man deshalb die Klasse `VertexView` als Oberklasse für die eigenen *View*-Klasse `MyCellView`. Wichtige Bestandteile eines *Views* sind die Festlegungen der visuellen Aspekte durch die Attribute.

Diese werden mittels der Klasse `GraphConstants` in der *AttributeMap* gesetzt, somit lassen sich schon viele Erscheinungsmerkmale einer *Cell* festlegen, siehe Programmcode 3. Reichen die Möglichkeiten von JGraph nicht aus, das Erscheinungsbild so zu gestalten, wie es erwünscht ist, dann muss ein eigener Renderer geschrieben werden, welcher diese Darstellung umsetzten kann.

```
01 public class MyCellView extends VertexView {
02
03    private static MyRenderer renderer = new MyRenderer();
04
05    public MyCellView(Object cell){
06       super(cell);
07       // Add attributes, like color, border etc...
08       GraphConstants.setBackground(getAttributes(),Color);
09       // Add more attributes...
10    }
11
12    public Point2D getPerimeterPoint(){
13       // Calculates a new perimeter point.
14       return ((MyRenderer)renderer).getPerimeterPoint();
15    }
16
17    public CellViewRenderer getRenderer(){
18       return renderer;
19    }
20 }
```

*Programmcode 3: Grundaufbau einer eigenen View-Klasse.*

Zwei Methoden sind wichtig für eine Klasse MyCellView: Zum einen die Methode `getRenderer()`, zum anderen die Methode `getPerimeterPoint()`.

Die Methode `getRenderer()` übergibt den zu verwendenden Renderer der Klasse. Der Renderer kann der *DefaultRenderer*[37] für diesen *Cell*-Typ sein oder ein eigener. Auf jeden Fall sollte er als Klassenvariable deklariert werden. Dies hat den Hintergrund, den Speicherverbrauch der Anwendung möglichst gering zu halten, indem nur ein Renderer für jede Instanz eines *Views* verwendet wird.

Die Methode `getPerimeterPoint()` ist für den Anschluss einer *Edge* an eine *Cell* wichtig. Die visuelle Repräsentation[38] einer *Edge* wird von einem *PerimeterPoint* der *Source-Cell* zum *PerimeterPoint* der *Target-Cell* gezeichnet. Standardmäßig wird hier

37 Die *DefaultRenderer* in JGraph sind: `VertexRenderer`, `EdgeRenderer` und `PortRenderer`.

38 Dies kann entweder eine Gerade sein oder z.B. eine Bézierkurve. Letztlich ist aber immer Anfangs und Endpunkt durch den *PerimeterPoint* jeder *Cell* festgelegt.

ein Punkt verwendet, der sich aus dem Schnittpunkt der direkten Verbindungslinie zwischen den Mittelpunkten zweier *Cells* und der *Bounding Box* einer *Cell* entsteht.

Weicht die Form einer *Cell* von der *Bounding Box* ab, so setzt die Kante für bestimmte Winkel mit einer Lücke an die *Cell* an. Abbildung 13 zeigt diesen Sachverhalt. Um dies zu vermeiden, muss die Methode `getPerimeterPoint()` überschrieben und geeignete Punkte für jeden Winkel berechnet werden.

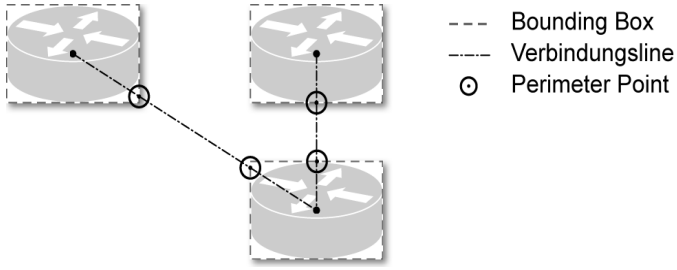


*Abbildung 13: Berechnung der Perimeter Positionen.*

Dies kann entweder in der `CellView`-Klasse geschehen oder wie in Programmcode 3 an den Renderer weitergeleitet werden, damit diese dort in der Renderer-Klasse implementiert wird. Der Renderer einer *Cell* wird im *View* bestimmt, wie in Zeile 03 und 17 des Programmcode 3 zu sehen ist. Möchte man einen eigener Renderer festlegen, dann muss dies dort definiert werden.

Um einen eigenen Renderer zu schreiben, kann man entweder allgemein eine Klasse von `JLabel` spezialisieren und das `CellViewRenderer`-Interface implementieren oder einen bestehenden *DefaultRenderer* spezialisieren. In dem `MyCellView` Beispiel oben wurde von der Klasse `VertexView` spezialisiert, deshalb bietet es sich an den `VertexRenderer` als Erweiterung des eigenen Renderer zu verwenden.

Im Folgenden werden beide Möglichkeiten umrissen. Wählt man die Spezialisierung von `JLabel`, dann muss durch das `CellViewRenderer`-Interface die Methode `getRendererComponent()` implementiert werden. Die Methode gibt eine `JComponent` zurück. Zur Erzeugung der Darstellung können deshalb alle Swing-Komponenten wie `JPanels`, `JButtons`, `JScrollPanels`, etc. verwendet werden.

Aus der vollständige Signatur der Methode in Programmcode 4, Zeile 04, ist ersichtlich, dass durch die Parameter unterschieden werden kann, in welchem Zustand die so erzeugte *Cell*-Komponente sein soll. So ist eine Unterscheidung möglich, ob die *Cell* selektiert wird, den Fokus besitzt oder ob eine Vorschau gezeichnet werden soll solange die *Cell* verschoben wird.

Der Programmcode 4 zeigt die Klassenstruktur eines solchen Renderers, der das `CellViewRenderer`-Interface implementiert. Die Methode `getRendererComponent()` besitzt fünf Parameter. Die letzten drei Booleschen Werte beschreiben, wie erwähnt, die möglichen Zustände der *Cell*. Die Variable `view` ist Klassenvariable und bietet den Zugriff auf die im *View* gesetzten Attribute.

```
01 public class MyRenderer extends JLabel implements CellViewRende-
   rer {
02
03    @Override
04    public Component getRendererComponent(JGraph graph, CellView
   view, boolean sel, boolean focus, boolean preview) {
05
06    Color bColor = GraphConstants.getBackground(view.getAttribu-
   tes());
07       // ...
08       return null;
09    }
10
11    public Point2D getPerimeterPoint(){
12       // Calculates new perimeter point.
13       return null;
14    }
15 }
```

*Programmcode 4: Renderer mit CellViewRenderer-Interface.*

Erweitert man einen bestehenden Renderer, hat man die Wahlfreiheit, ob man lieber die Methode `getRendererComponent()` oder die `paint()`-Methode oder beide der Oberklasse überschreibt, um eine eigene Darstellung zu realisieren.

Die `paint()`-Methode bietet ebenfalls die Möglichkeit, die einzelnen Zustände zu unterscheiden. Sie arbeitet aber mit einfachen geometrischen Grundelementen, wie *Line* oder *Rectangle*[39]. Aus diesen Elementen muss dann das Erscheinungsbild zusammengesetzt werden.

Der Programmcode 5 zeigt beispielhaft wie ein Renderer, der sich vom `VertexRenderer` ableitet, aufgebaut sein kann. Die Klasse `GraphConstants` und die Klassenvariable `view` bieten wieder Zugriff auf die Attribute des *View*.

39 Java bietet in Klassenbibliothek *java.awt.Graphics* und der abgeleiteten Klasse *Graphics2D* unterschiedliche Zeichenmethoden für geometrische Figuren.

```
01 public class MyRenderer extends VertexRenderer {
02
03    @Override
04    public void paint(Graphics g){
05
06       String label = view.getCell().toString();
07       Color bColor = GraphConstants.getBackground(view.getAttri-
   butes());
08       // ...
09
10       if(selected) { /* Do something */ };
11       else if (hasFocus) { /* Do something */ };
12       else if(preview) { /* Do something */ };
13       // ...
14    }
15    public Point2D getPerimeterPoint(){
16       // Calculates new perimeter point.
17       return null;
18    }
19 }
```

*Programmcode 5: Renderer von VertexRenderer abgeleitet mit Paint-Methode.*

Nachdem der *CellView* und ein zugehöriger Renderer erstellt worden sind, muss noch die Zuordnung des *Views* zu einer *Cell* erfolgen. Hierfür ist die Klasse `CellViewFactory` zuständig, bzw. die zugehörige Standardimplementation in JGraph, die `DefaultCellViewFactory`.

Sie besitzt die Methode `createVertexView()`, welche den zu einem Objekt passenden *View* zurückgibt. Dieser *View* wird neu erzeugt und besitzt nur eine Lebensdauer bis zum Abschluss der Zeichenroutine. Anschließend wird er wieder verworfen. Dies geschieht aus Speicherplatzgründen, um bei einer großen Anzahl von *Cells* nicht die gleiche Menge an *Views* referenzieren zu müssen.

Hat man eigene *Cells* erstellt, muss die Klasse `DefaultCellViewFactory` spezialisert werden um die eigenen *Views* zuzuordnen. Dazu wird die Klasse `MyCellViewFactory` erstellt. In ihr muss die Methode `createVertexView()` implementiert sein, die ein `Object` als Eingabewert bekommt und als Rückgabewert einen `VertexView` gibt. Dieses `Object` kann jedes Objekt sein, das man in JGraph als *Cell*-Typ verwendet. Deshalb sollte in der Methode die Instanz des Objektes geprüft und der entsprechende *View* zurückgegeben werden.

Es muss nicht immer ein eigener *View* definiert sein. Wenn dies erwünscht ist könnte man auch für die Klasse `MyCell` den *DefaultCellView* zurückgeben oder jeden beliebi-

gen *View*. Programmcode 6 zeigt die Klasse `MyCellViewFactory`. Diese ist letztlich eine Auflistung von *Cell*-*View*-Paaren. Der Aufruf der Oberklassen-Methode gewährt, dass auch weiterhin die *DefaultCell*, *DefaultEdge* und der *DefaultPort* den korrekten *View* zurückgeben.

```
01 public class MyCellViewFactory extends DefaultCellViewFactory {
02    @Override
03    public VertexView createVertexView(Object cell){
04
05       if(cell instanceof MyCell)
06          return new MyCellView((MyCell)cell);
07       // Create further views for other cell types.
08
09       return super.createVertexView(cell);
10    }
11 }
```

*Programmcode 6: Aufbau der MyCellViewFactory.*

Nachdem man das *Model* und den *View* erstellt hat, ist noch der Controller an die eigenen *Cell*-Typen anzupassen, um Interaktionen oder ein *Cell*-Verhalten zu ermöglichen, welche über die von JGraph vorgegebenen Möglichkeiten hinausreichen.

### 4.3.3 Die GraphUI-Klasse

JGraph stellt mit der Klasse `GraphUI` und ihrer Implementation, der `BasicGraphUI`, die Möglichkeit *Cells* mittels der Maus zu verschieben, per Doppelklick die Labels einer *Cell* zu editieren und weitere Aktionen durchzuführen. Möchte man eigene Interaktionsmöglichkeiten per Maus oder Tastatur haben, ist der beste Weg eine eigene Klasse von der Klasse `BasicGraphUI` abzuleiten.

Die Klasse bietet ein Prinzip eigene *Listener*[40] zu registrieren. Dazu wird die Methode `createX()` implementiert. `X` kann für einen der vordefinierten *Listener* stehen, wie *MouseListerner*, *KeyboardListener*, *GraphModelListener* oder *GraphSelectionListener.* Die Zeilen 04 und 09 aus Programmcode 7 zeigen die nötigen *Create*-Methoden für die *Listener* der Klasse `MyBasicGraphUI`.

Die Methode `createX()` erzeugt jeweils eine Instanz des eigenen *Listeners* und gibt diesen zurück. Das Registrieren der *Listener* wird in der Oberklasse `BasicGraphUI` in der Methode `installListeners()` übernommen, welche genau diese *Create*-Metho-

40 Der Begriff *Listener* wird im folgenden synonym für den Begriff *Handler* benutzt. Der *MouseHandler* von JGraph ist eine Adapter-Klasse – das Interfaces *Listener* muss alle Methoden der Oberklasse implementieren, während die abstrakte Klasse *Handler* nur die benötigten Methoden [JSE6b].

den aufruft und eine Referenz auf den *Listener* setzt. Die *Listener* und ihre Funktionen selbst können als innere Klassen der `BasicGraphUI` implementiert werden.

```
01 public class MyBasicGraphUI extends BasicGraphUI{
02
03    @Override
04    protected KeyHandler createKeyListener(){
05       return new XTKeyHandler();
06    }
07
08    @Override
09    protected MouseListener createMouseListener(){
10       return new XTMouseHandler();
11    }
12
13    public class XTMouseHandler extends MouseHandler{
14
15       public void mousePressed(MouseEvent e){
16          focus = graph.getTopmostViewAt(e.getX(), e.getY(),
          reverse, leafsOnly);
17
18          if(focus instanceof MyCellView){
19             MyCell cell = (MyCell) focus.getCell();
20             // Do something intelligent.
21          }
22          super.mousePressed(e);
23       }
24
25       public void mouseDragged(MouseEvent e){
26          // Do something intelligent.
27       }
28       // ...
29    }
30
31    public class XTKeyHandler extends KeyHandler{
32
33       @Override
34       public void keyPressed(KeyEvent e) {
35          // Do something intelligent.
36       }
37    }
38 }
```

*Programmcode 7: MyBasicGraphUI-Klasse mit inneren Listener-Klassen.*

Eine wichtige Referenz, welche durch die Klasse `BasicGraphUI` gegeben ist, ist die Referenz: `graph`. Sie bietet Zugriff auf die JGraph-Instanz und damit auf Selektionsmethoden, wie `getTopmostViewAt()` oder andere, um ein *View* z.B. zu gegebenen Mauskoordinaten auszuwählen, wie im Programmcode 7, Zeile 16 dargestellt ist.

Die Referenz auf das `focus`-Objekt kann auch genutzt werden, um je nach Objekt-Klasse entsprechende Aktionen auszuführen. Zeile 18 verdeutlicht die Objektprüfung. Damit lässt sich ein Standartverhalten für eigene *Cell*-Typen definieren. Benötigt man das *Cell*-Objekt eines *Views*, um z.B. auf die Benutzer-Daten zuzugreifen, dann gelangt man mit der Methode `getCell()` des *View*-Objektes an die *Cell*, siehe Zeile 19.

Möchte man das ursprüngliche Verhalten der JGraph-*Listener* aber für Objekte erhalten, so kann die Methode der Oberklasse des *Listeners* am Ende der eigenen Objektabwicklung noch aufgerufen werden, siehe Zeile 22.

Sollte man sich dazu entschließen die *Listener* neu zu schreiben, bietet dies eine gute Möglichkeit eigene Vorstellung in JGraph umzusetzen. Mittels des Kapitels 4.3 »Klassen-Spezialisierung« und der Unterkapitel sollte es gut möglich sein, sich schnell ein eigenes Grundgerüst aufzubauen und dabei alle nötigen Methoden und Hilfsklassen zu berücksichtigen, die ein reibungsloses Funktionieren von JGraph voraussetzten.

### 4.3.4 Attributsänderungen

Wie im Kapitel 4.2.3 »Der GraphLayoutCache« beschrieben besitzt JGraph Methoden, Änderungen am *Model* oder *View* vorzunehmen und diese dem Benutzer anzuzeigen.

Damit Veränderungen korrekt angezeigt werden bzw. überhaupt sichtbar werden, muss folgender Ablauf berücksichtigt werden: Die Synchronisation zwischen *Model* und *View* wird automatisch von JGraph vorgenommen, wenn das *Model* verändert wird. Dazu ruft JGraph die Methode `refresh()` auf, welche die Attribute der *Cell* mit den dazugehörigen *CellViews* abgleicht. Dieser Schritt bewirkt noch keine visuelle Änderung der Anzeige, erst ein anschließendes `repaint()` macht diese sichtbar. Dieser Ablauf ist nötig, um wieder eine aktualisierte und korrekte Anzeige zu erhalten.

Die drei Grundeditiermethoden `insert()`, `edit()` und `remove()` führen diese beiden Schritte automatisch aus. Nimmt man aber Änderungen vor, ohne Verwendung einer der drei Methoden, dann ist man selbst dazu angehalten, die beiden Schritte auszuführen, um eine visuelle Repräsentation der Änderungen zu erhalten. Ein Grund, warum man auf die Methoden verzichten kann, ist dann gegeben, wenn Änderungen mit bestmöglicher Performance ausgeführt werden sollen und das Festhalten eines Undo-Event nicht nötig ist. Ein Beispiel für solche Fälle wäre, wenn man ein *MouseOver*-Event für einen *View* anzeigen oder bei einem Doppelklick ein Highlight-Event ausführen möchte. Bei solchen Aktionen machen Undo-Events keinen Sinn und sie sollten so schnell wie möglich dem Benutzer angezeigt werden.

Um den Überblick von JGraph abzuschließen, möchte ich noch auf die beiden Möglichkeiten eingehen, Attributsänderungen korrekt auszuführen und diese angezeigt zu bekommen. Wie bereits erwähnt, sollte Änderungen immer auf den *Views* vorgenommen werden und nicht das *Model* direkt manipuliert werden. Deshalb bezieht sich der `edit()`-Aufruf auch auf den *GraphLayoutCache* und nicht auf das *GraphModel*.

Die erste Möglichkeit ist die Synchronisation und das Neuzeichnen JGraph zu überlassen. Dies wird in den meisten Fällen der beste Weg sein um eine korrekte Anzeige zu erhalten. Dazu wird zuerst eine neue `HashMap`[41] als *AttributeMap* erzeugt, welche die beabsichtigten Änderungen aufnimmt. Dann werden mittels der `GraphConstants` die gewünschten Attribute als *Name/Value*-Paare in der `HashMap` gesetzt. Anschließend wird dem `edit()`-Aufruf das zu verändernde *Cell*-Objekt und die neue `HashMap`, welche die Änderung in sich trägt, übergeben.

JGraph vergleicht die bisherige *AttributeMap* der übergebenen *Cell* mit der neuen `HashMap`. Dabei kann entweder ein neues Attribute in die bisherige *Map* eingefügt werden, ein vorhandenes aktualisiert oder ein Attribute aus der *Map* entfernt werden. Anschließend wird der *View* mit dem *Model* synchronisiert. Zum Schluss wird dann die `repaint()`-Methode ausgeführt und die Änderungen dargestellt. Der Programmcode 8 zeigt die oben beschriebenen Schritte.

```
01  HashMap<Object, Object> map = new HashMap<Object, Object>();
02  GraphConstants.setBackground(map, new Color(128, 128, 128));
03
04  graph.getGraphLayoutCache().editCell(cell, map);
```

*Programmcode 8: Attributsänderung einer Cell mittels editCell().*

JGraph besitzt eine eigene Methode, um eine einzelne *Cell* zu editieren. Sie heißt `editCell()`. Letztlich ruft diese Methode aber die Methode `edit()` mit der *Cell* als Parameter auf. Die Signatur der Methode ist:

- `editCell(Object cell, Map attributes)`

41 *HashMap* bezieht sich auf die Java Klasse java.util.HashMap.

Besonders interessant sind zwei weitere Signaturen der `edit()`-Methode:

- `edit(Map attributes)`
- `edit(Map attributes, ConnectionSet cs,`
  `ParentMap pm, UndoableEdit[] e)`

Beide können genutzt werden, um mehrere Änderungen an *Cells* zu sammeln und diese dem *GraphLayoutCache* in einem `edit()`-Aufruf zu übergeben. Dazu wird für jede Änderung wieder eine eigene *Map* erstellt. Diese wird aber zuerst in einer sogenannten *NestedMap* gesammelt. Sind alle *Maps* erstellt und in der *NestedMap* gespeichert, dann wird nur noch die *NestedMap* an die Methode `edit()` übergeben.

```
01 HashMap<Object, Object> nested = new HashMap<Object, Object>();
02
03 HashMap<Object, Object> map1 = new HashMap<Object, Object>();
04 GraphConstants.setBackground(map1, Color.CYAN);
05 nested.put(cell1, map1);
06
07 HashMap<Object, Object> map2 = new HashMap<Object, Object>();
08 GraphConstants.setBorderColor(map2, Color.RED);
09 nested.put(cell2, map2);
10
11 // this signature
12 graph.getGraphLayoutCache().edit(nested);
13 // or...
14 graph.getGraphLayoutCache().edit(nested, null, null, null);
```

*Programmcode 9: Attributsänderung an mehreren Cells mittels NestedMap.*

Wichtig ist bei dieser Methode, dass die *Map*, welche die Änderungen beinhaltet, beim Einfügen in einer *NestedMap* einer *Cell* zugeordnet wird, für welche die Änderungen gelten sollen. Die Zeilen 05 und 09 des obigen Beispiels zeigen eine solche Zuordnung.

# 5 Implementierung

Im Folgenden Kapitel wird auf die im Rahmen der Diplomarbeit geleistete Ausarbeitung des Neukonzeptes des XT-Clients eingegangen. Dazu werden die Veränderungen an den Klassenstrukturen und den Methoden näher erläutert. Um den Leser vorab auch den Zugriff auf die Java Klassen, Java Bibliotheken und die Dokumentation des Projektes zu geben, verweise ich auf das Gitorious[42] Repository der AG Rechnernetze. Um das gesamte Projekt[43] lokal zu betrachten oder um an dem Projekt weiterzuarbeiten wird ein lokales angelegtes GIT Repository benötigt, in welches das Projekt geklont werden kann. Für eine Einführung in GIT, verweise ich auf die freie Online-Ausgabe des Buches „Pro GIT“ [PGIT].

Zunächst einmal werden die von der Implementation des Neuentwurf betroffenen *Packages* vorgestellt. Besonders im *Package* `frontend` wurde im Rahmen dieser Arbeit viele Klassen neu implementiert oder modifiziert. Auf die Details wird in Kapitel 5.1 »Implementierung des Package frontend« eingegangen.

Das *Package* `backend` ist wichtiger Bestandteil für die Funktionalität des *Packages* `frontend`, musste aber für die Arbeit nicht modifiziert werden und wird deshalb nicht weiter erläutert. Es ist bereits schon von seinem Autor Daniel Pähler [PAE06] näher beschrieben worden .

Ein weiteres wichtiges *Package* ist `jgraph2xml`, das neu in den RIP-XT eingefügt wurde. Es beinhaltet zwei Klassen, welche die Aufgabe der Positionssicherung und Positionsextraktion erfüllen. Eine näherer Beschreibung dieses *Packages* wird in Kapitel 5.2 »Implementierung des Package jgraph2xml« gegeben.

Auf jede einzelne Klasse und ihre Änderungen oder Ergänzungen einzugehen ist wenig sinnvoll. Um ein allgemeinen Überblick über die Arbeit zu geben, werden wichtige Aspekte ausgewählt, um sie näher zu kommentieren. Abgesehen davon sind alle Komponenten aber auch mit Javadoc-Kommentaren versehen, so dass eine API-Dokumentation für die Klassen und Methoden existiert, wie oben erwähnt. Einzelnen Bestandteile innerhalb des Quellcodes sind durch die Javadoc-Angabe: `@author` gekennzeichnet. Dadurch ist leicht zuzuordnen, welche Teile von mir implementiert worden sind. Als Autorenname wurde meine Universitätskennung „*bernwolf*“ verwendet. Im Falle von Modifikationen ist die ursprüngliche Autorenangabe erhalten und um ein „*modified by bernwolf*“ ergänzt. Sind Methoden eins zu eins aus der ursprünglichen Implementation

42 *Gitorious,* ist ein *Distributed Version Control Systems*, Offizielle Internetseite: gitorious.org.
43 Projektseite: git.uni-koblenz.de/xtpeerredesigned.

übernommen worden, wurde der Autor ergänzt, wenn er vorher nicht vorhanden war. Einzige Ausnahme dieser Regel ist die Klasse `XTCientGUI`. Sie beinhaltet die Graphische Benutzeroberfläche und ist zum größten Teil von Pähler, Lang, Keupen und Bohdanowicz erstellt worden. Eine genau Zuordnung der Teile zum jeweiligen Autor ist nicht immer möglich, wenn Autorenangaben schon fehlten.

Die Klassen des *Package* `frontend` ist maßgeblich für die visuelle Neugestaltung des XT-Clients verantwortlich, weshalb dieser Teil zuerst näher beschrieben wird.

## 5.1 Implementierung des Package frontend

Zunächst einmal gebe ich ein tabellarischen Überblick über die Klassennamen der ursprünglichen Klassen des *Packages* `frontend` und den Namen der neuen Klassen, damit eine klarere Zuordnung besteht. Außerdem ist die Anzahl an neu erstellten, modifizierten oder beibehalten Methoden für jede Klasse angegeben.

| **Package: frontend** | | | | |
|---|---|---|---|---|
| Alte Nomenklatur | Neue Nomenklatur | neu | modifiziert | beibehalten |
| `NetworkInterfaceVertex` | `InterfaceCell` | 3 | 1 | 2 |
| `NetworkInterfaceVertexView` | `InterfaceCellView` | 5 | 0 | 0 |
| `JGraphEllispeRenderer` | `NetworkCellRenderer` | 1 | 1 | 0 |
| `NetworkVertex` | `NetworkCell` | 1 | 0 | 4 |
| `NetworkVertexView` | `NetworkCellView` | 3 | 0 | 0 |
| `NetworkVertexFactory` | `NetworkVertexFactory` | 0 | 0 | 2 |
| `XTServerRenderer` | `RouterCellRenderer` | 2 | 0 | 0 |
| `XTServerVertex` | `RouterCell` | 7 | 1 | 5 |
| `XTServerVertexView` | `RouterCellView` | 3 | 0 | 0 |
| `XTCellViewFactory` | `XTCellViewFactory` | 1 | 0 | 0 |
| `XTClientGui` | `XTClientGUI` | 8 | 6 | 130 |
| `XTClientUI` | `XTClientUI` | 18 | 2 | 1 |
| `Utilities` | `Utilities` | 5 | 1 | 2 |
| `Start` | `Start` | 0 | 0 | 1 |
| Nicht vorhanden | `SSH` | 2 | - | - |

*Tabelle 1: Klassenübersicht des Package fontend.*

Aus der Tabelle ist der auf das MCV Pattern resultierende Aufbau abzulesen. Jede der drei Klassen, Router, Interface und Network, besitzt eine *Model*-Klasse mit der Endung

*Cell* und eine *View*-Klasse mit der Endung *CellView*. Für die Klasse Router und Interface ist ein eigener Renderer nötig gewesen.

Dieser Renderer für die Klasse `NetworkCell` ist leicht modifiziert übernommen worden. Er stammt ursprünglich von den JGraph-Entwicklern aus deren Beispielklasse `com.jgraph.example.fastgraph.FastCircleView`[44]. Dieser Renderer wurde um die Methode `getPerimeterPoint()` erweitert, um die unschönen Lücken zwischen Kante und Ellipse an den Netzwerk-Symbolen zu beseitigen. Die Methode berechnet den exakten Schnittpunkt zwischen einer Kante und der Ellipsenhülle. Ohne diese Methode wurde ein Schnittpunkt zwischen Kante und dem Objekt berechnet, der nicht auf der Außenkante der Ellipse verläuft, sondern auf der rechteckigen *Bounding Box*.

Die Klasse `XTCellViewFactory` dient dem *GraphLayoutCache* zur Zuordnung der *Views* zu den *Model*-Objekten und besteht somit nur aus den Zuordnungspaaren von *Cell*- zu *View*-Klasse.

Die weitaus größte Klasse im *Package* stellt die Klasse `XTClientGUI` dar. Sie beinhaltet die Komponenten der graphischen Benutzeroberfläche. In dieser Klasse wird die JGraph-Instanz ebenfalls durch den Aufruf der Methode `getGraph()` erzeugt. Sollte noch keine JGraph instantiiert worden sein, dann wird ein neuer Graph mit *GraphModel* und *GraphLayoutCache* erzeugt. Dem *GraphLayoutCache* wird hierbei die benötigte `XTCellViewFactory` übergeben.

Der Graph kann durch Attribute konfiguriert werden. Sie dienen dazu das JGraph Verhalten allgemein zu definieren, wie z.B. das Lösen von Kanten von ihren Objekten und die Möglichkeit sie mit anderen Objekt zu verbinden. Für dieses Verhalten gibt es das Attribut `setDisconnectable(boolean value)`, das durch diese Methode definiert werden kann. Wird für `value` der Wert `false` übergeben, dann können Kanten nicht mehr durch Wegziehen verschoben werden. Die gesamten Eigenschaften können genauer im Handbuch [JGM] mit detaillierter Beschreibung nachgelesen werden.

Die Erzeugung der JGraph-Objekte erfolgt durch eine Methoden Aufruffolge, die mit der Methode `setServersFromXMLConfig()` beginnt. In ihr werden zuerst die Server aus dem XML-Dokument dem Benutzer zur Auswahl vorgestellt. Für alle ausgewählten Server wird durch die Methode `addXTServer()` zunächst ein `XTServer` des `backends` erstellt. Für jeden `XTServer`, der erfolgreich initialisiert wurde, wird auch ein JGraph `XTServerVertex` des `frontends` mit seinen Interfaces und den zugehörigen Netzwerken erstellt. Diesen Erzeugungsablauf der `backend`-Objekte verdeutlicht das Sequenz-

44 *FastCircleView.java*. Die Klasse liegt im examples/com/jgraph/examples/fastgraph Verzeichnis des JGraph JARs.

diagramm in Abbildung 14. Für die Erzeugung der `frontend`-Objekte verweise ich auf das bereits gezeigte Sequenzdiagramm in Abbildung 4.

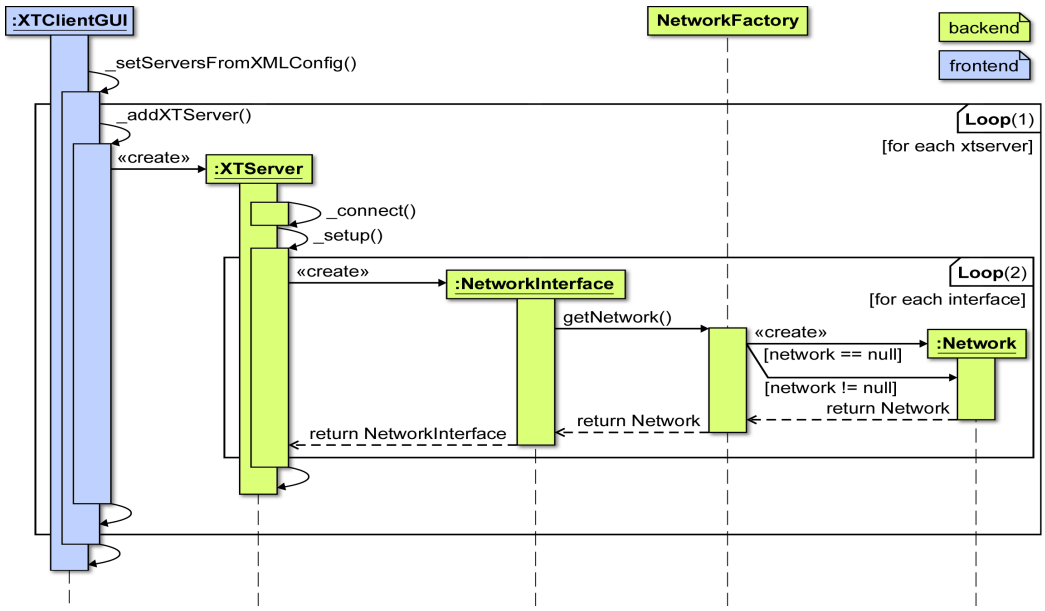


*Abbildung 14: Objekt-Erzeugungsdiagramm des backends.*

Um mit der erzeugten JGraph-Topologie interagieren zu können ist die Klasse `XTGraphUI` besonders wichtig. In dieser sind der `MouseListener`, `KeyboardListener`, `XTActionListener` und `HighlightListener` als innere Klassen implementiert. Die Methoden legen das Verhalten der Objekte während der Benutzer-Interaktion fest.

`XTGraphUI` ist eine Spezialisierung der Klasse `BasicGraphUI`. Die innere Klasse `XTMouseHandler` überschreibt die Methoden `mouseDragged()`, `mouseReleased()` und `mousePressed()` des `MouseListeners` von `BasicGraphUI`. Dies ist nötig, um die Anforderungen des Neukonzepts und seiner Interaktionsanforderungen umzusetzen.

Eine der ersten Interaktionen, die behandelt wird, ist die Methode `mousePressed()`, siehe Programmcode 10. Sie wird immer dann aufgerufen, wenn ein Benutzer mit der linken Maustaste innerhalb der JGraph-Anwendung klickt. Um die Interaktion zwischen dem Benutzer und den Objekten der Anwendung zu ermöglichen, muss erst geprüft werden, welche *Cell* betroffen ist. Dazu stehen mehrere Methoden zu Verfügung. Über die Referenz des `graph` können sie angesprochen werden. Im Wesentlichen vollziehen alle diese Methoden eine Übertragung der Klick-Koordinaten auf die *Model*-Koordinaten und prüfen welche *Cell* betroffen ist.

```
01 @Override
02 public void mousePressed(MouseEvent e){
03
04    mouseClickPos = graph.fromScreen(e.getPoint());
05    focus = graph.getTopmostViewAt(
         mouseClickPos.getX(),mouseClickPos.getY(), true, false);
06    // 1. Anweisungsblock - Distanzvektor loeschen.
07    if(focus == null || !graph.isCellSelected(focus.getCell())){
08       distances.clear();
09    }
10    // 2. Anweisungsblock - Cell-Typen behandeln.
11       if(focus != null){
12       // Calculate offset to the bounds of a cell.
13       calcBoundsOffset(new Point2D.Double(
            focus.getBounds().getX(), focus.getBounds().getY()));
14       // Show context menu on right clicks.
15       checkPopupMenu(e);
16
17       if(focus instanceof PortView){
18          focus = focus.getParentView();
19
20       } else if(focus instanceof RouterCellView){
21          Utilities.bringSelectionToFront(graph, focus);
22
23       } else if (focus instanceof InterfaceCellView){
24          newTimestamp = e.getWhen();
25          // Double click event - triggerUpdate() on interface!
26          if((newTimestamp - oldTimestamp) < 600 &
                focus == lastFocus){
27             oldTimestamp = 0;
28             try {
29                ((InterfaceCell)focus.getCell()).
                     getUserObject().triggerUpdate();
30                @SuppressWarnings("unused")
31                HighlightListener highListener =
                     new HighlightListener(focus.getCell());
32             } catch (XTServerException e1) {
33                Utilities.generateExceptionDialog(e1, graph);
34                e1.printStackTrace();
35             }
36          }
37       } else if (focus instanceof NetworkCellView &
                        e.getButton() == MouseEvent.BUTTON3){
38          NetworkCell networkCell = (NetworkCell)focus.getCell()
39
40          if(!networkCell.getUserObject().getAddress().
                        getHostAddress().startsWith("192")){

   Fortsetzung auf nächster Seite...
```

```
41              String network = networkCell.getUserObject().
                   getAddress().getHostAddress()+"/"
42              +networkCell.getUserObject().getPrefixLength();
43              new MetricPerNetworkFrame(network,xtServerTable);
44           }
45        }
46     }
47     // Don't move elements at right clicks.
48     if(!SwingUtilities.isRightMouseButton(e))
49        super.mousePressed(e);
50
51     oldTimestamp = newTimestamp;
52  }
```

*Programmcode 10: Die mousePressed-Methode.*

In `mousePressed()` wurde die Methode `getTopmostViewAt()` benutzt, um eine *Cell* zu selektieren, siehe Zeile 05. Die Methode gibt immer den obersten *View* zu einer Koordinate zurück.

Der Programmteil von `mousePressed()` besteht aus zwei Anweisungsblöcken. Der 1. Anweisungsblock wird ausgeführt, wenn der Benutzer auf eine freie Fläche klickt, um z.B. ein *Cell* bewusst zu deselektieren oder wenn auf ein andere als die momentan selektierte *Cell* geklickt wird. In diesen beiden Fällen wird der *Vector* mit den Distanzen geleert, da sie entweder nicht mehr benötigt werden, weil die Mehrfachauswahl damit aufgehoben wurde oder ein anderes Objekt ausgewählt wird, wodurch sich das Distanzreferenzobjekt geändert hat.

Der 2. Anweisungsblock dient der Unterscheidung der *Cell*-Typen, um entsprechende typenspezifische Anweisungen auszuführen. Handelt es sich bei dem `focus`-Objekt um einen Router, dann wird das Symbol mit seinen Interfaces in der Layer-Hierarchie, welche die Sichtbarkeit von Objekten bestimmt, in den Vordergrund gebracht. Die Klasse `Utilities` besitzt einige Hilfsmethoden, um die Layer-Hierarchie des Graphen um zu sortieren. Die Methode `bringSelectionToFront()` der Klasse `Utilities` verändert die Sichtbarkeit für eine Gruppe von selektierten Objekten. Es gibt außerdem die Methoden `bringInterfacesToFront()` und die Methode `bringEdgesToFront()`, welche ebenfalls die Layer-Hierarchie für die im Namen benannten Elemente verändern.

Als Teil der Anforderungen gilt, dass ein Router aus einem Router-Symbol mit einem oder mehreren Interface-Symbolen besteht. Ein solches Interface soll sich nicht beliebig weit von seinem Router entfernen lassen, bzw. sich nur auf einer festgelegten Ellipse, um den Router bewegen lassen. Ebenfalls soll sich diese Bindung der Interfaces zu ei-

nem Router auch dann nicht lösen, wenn der Router verschoben wird. Somit müssen die Interfaces eines Routers bei dessen Verschiebung relativ dazu mitbewegt werden.

Diese einfachen Bedingungen wurde in der Methoden `mouseDragged()` des Listeners definiert. Der Programmcode 11 zeigt die Methode mit verkürzten Subroutinen in den Schleifenrümpfen. Die Stellen, an denen der Code verkürzt wurde, sind durch ein „*“ im Kommentar gekennzeichnet.

Die Methode `mouseDragged()` wird jedes Mal aufgerufen, wenn der Benutzer die Maus bei gedrückter linker oder rechter Maustaste bewegt. Sie wird immer wieder durchlaufen, solange die Maus bewegt und die Taste gedrückt wird. Die Behandlung des *mouseDragged*-Events läuft folgendermaßen ab: Bevor eigene Anweisungen ausgeführt werden, wird zuerst die Methode `mouseDragged()` des `BasicMarqueeHandler` der Superklasse `BasicGraphUI` aufgerufen.

Der *MarqueeHandler* ist für eine Mehrfachauswahl zuständig. Dies geschieht immer dann, wenn der Benutzer bei gedrückter linker Maustaste die Maus weiterbewegt. Dadurch wird ein Auswahlrahmen aufgezogen, der dem Benutzer signalisiert, dass er eine Mehrfachauswahl durchführt. Der Rahmen dient zur Visualisierung der Elemente, die sich innerhalb der Auswahl befinden.

Die Methode des MarqueeHandlers wird lediglich für das Zeichnen des Auswahlrechtecks genutzt, die Festlegung welche *Cells* sich in der Auswahl befinden, musste auf Grund der gestellten Anforderungen selbst übernommen werden.

Die anschließenden zwei Zeilen, 05 und 06, dienen dazu zwei Hilfskoordinaten zu berechnen, die für die Positionierung der *CellViews* benötigt werden. Der restliche Programmcode ist in drei Anweisungsblöcke gliedern, die durch `if`/`else-if`-Anweisungen gebildet werden. Diese Anweisungsblöcke sind zuständig für die Behandeln unterschiedlicher *mouseDragged*-Interaktion.

Eine wichtige Variable für die Verarbeitung von Events ist `focus`. Sie ist eine Referenz auf einen durch Mausklick ausgewählten *CellView* einer *Cell*. Befindet sich beim Klicken keine *CellView* unter dem Mauszeiger, ist die Variable `null`. Die Variable ist wichtiger Bestandteil für die Bedingungen der Anweisungsblöcke und der darin ausgeführten Methoden.

Der 1. Anweisungsblock prüft, ob ein Benutzer beim Anklicken einer *Cell* nicht die *Cell* selbst, sondern zufällig den Port der *Cell* ausgewählt hat. Dies ist nicht offensichtlich, da sich der Port unsichtbar in der Mitte einer *Cell* befindet. Hat `focus` zufällig die Referenz auf einen *PortView*, führt das zu Problemen in der weiteren Behandlung durch die Anweisungsblöcke. Deshalb wird in diesen Fällen die Referenz von `focus` auf den

```
01 @Override
02 public void mouseDragged(MouseEvent e){
03
04    marquee.mouseDragged(e);
05    Point2D eScaled = graph.fromScreen(e.getPoint());
06    Point2D gridPoint = graph.snap(new Point(
         (int)eScaled.getX(), (int)eScaled.getY()));
07    // 1. Anweisungsblock – Port-Behandlung
08    if(focus instanceof PortView)
09       focus = focus.getParentView();
10    // 2. Anweisungsblock – Mehrfachauswahl
11    if( focus != null && graph.getSelectionCount() > 1 && !
12    (SwingUtilities.isRightMouseButton(e))){
13       // Make this only once at the beginning of a drag.
14       if(marqueeStart){
15          // Expands marqueeViews.
16          marqueeViews = checkViewSelection(
               graph.getGraphLayoutCache().getMapping(
               graph.getSelectionCells()));
17          graph.setSelectionCells(graph.getGraphLayoutCache()
               .getCells((CellView[])marqueeViews));
18          calcDistances(new Point2D.Double(
               focus.getBounds().getX(), focus.getBounds().getY()),
               graph.getGraphLayoutCache().getCells(
               (CellView[])marqueeViews));
19          Rectangle2D marqueeBound =
               GraphLayoutCache.getBounds((CellView[])marqueeViews);
20          marqOffset.setLocation(
               eScaled.getX() - marqueeBound.getX(),
               eScaled.getY() - marqueeBound.getY());
21          marqueeStart = false;
22       }
23       checkInScreenLocation(new Point2D.Double(
            eScaled.getX() - marqOffset.getX(),
            eScaled.getY() - marqOffset.getY()));
24       for (int i = 0; i < marqueeViews.length; i++) {
25          // * Translate selected cells.
26       }
27    }
28    // 3. Anweisungsblock - Einzelauswahl
29    else if(focus != null &
            !(SwingUtilities.isRightMouseButton(e))){
30       // Moving an interface around its router.
31       if(focus instanceof InterfaceCellView){
32          // * Rotate selected cell.
33       }
34       // Moving a network.
35       else if(focus instanceof NetworkCellView){
36          // * Translate selected cell.
37       }
   Fortsetzung auf nächster Seite...
```

```
38        // Moving a router and its interfaces.
39        else if (focus instanceof RouterCellView){
40           // * Expand selection if necessary.
41        }
42     }
43  }  // End of mouseDragged
```

*Programmcode 11: Verkürzter Grundaufbau der mouseDragged-Methode*

*CellView* der übergeordneten Vater-*Cell* des *PortViews* geändert. Dadurch erhält man das eigentliche Objekt der Interaktion. Der Zusammenhang, dass ein Port immer als Kind einer *Cell* in der Datenstruktur gehalten wird, ist in Kapitel 4.2.2 »Das GraphModel« beschrieben.

Der 2. Anweisungsblock behandelt die Fälle, in denen mehr als eine *Cell* ausgewählt wurde, was einer Mehrfachauswahl entspricht. Innerhalb des Blocks befindet sich ein weiterer Anweisungsblock, der nur einmal zu Beginn der Mausbewegung ausgeführt und mit einer Booleschen Variablen geprüft wird, siehe Zeile 14.

Dieser einmal auszuführende Block sorgt dafür, dass die Auswahl durch die Methode `checkViewSelection()` vollständig ist, siehe Abbildung 15. Vollständig bezeichnet hier den Zustand, wenn für jede *RouterCell* oder *InterfaceCell* geprüft wurde, ob bei einem Router alle Interfaces in der Selektion sind (Abbildung 15, 1. Selektion) oder ob für ein Interface der Router und alle weiteren Interfaces in der Selektion sind (Abbildung 15, 2. Selektion).

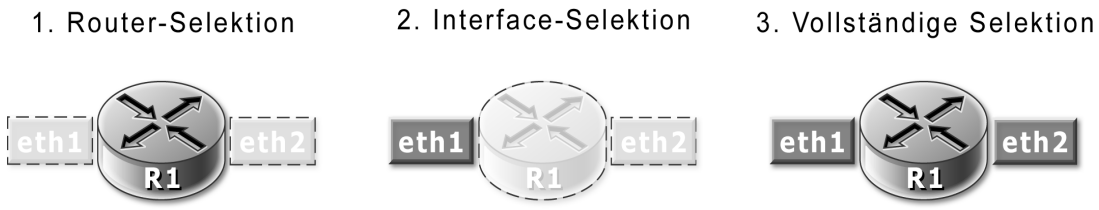


*Abbildung 15: Varianten der Selektionserweiterung.*

Die Methode erhält eine Liste von selektierten *Views* und prüft diese auf oben beschriebene Vollständigkeit und gibt die, wenn nötig ergänzte, vollständige Liste zurück. Anschließend wird in Zeile 17 die vollständige Liste der *Views* als selektiert markiert, damit dem Benutzter die überprüfte Selektion auch angezeigt wird.

Die Methode `calcDistances()` in Zeile 18 wird benötigt, um vor der eigentlichen Verschiebung die Distanzen aller beteiligten *Cells* zur Position des `focus`-Objekts zu

berechnen. Diese relativen Distanzen zwischen den Objekten verändern sich während der Verschiebung nicht und dienen dazu, die neuen globalen Positionen der Objekte relativ zur neuen Position des `focus`-Objekts zu bestimmen.

Die Variablen `marqueeBound` und `marqOffset` werden nur einmal berechnet und verändert sich für eine feste, aber beliebige Gruppe nicht mehr. Sie werden für Berechnungen der Methode `checkInScreenLocation()` benötigt.

Am Ende dieses inneren Anweisungsblocks wird noch die Boolesche Variable `marqueeStart` auf `false` gesetzt, um zu verhindern, dass der Block im nächsten Aufruf der *mouseDragged*-Methode nochmals ausgeführt wird. Die Variable `marqueeStart` wird erst wieder auf `true` gesetzt, wenn ein *mouseReleased*-Event eintritt – was einem Ende des *mouseDragged*-Event entspricht.

Vor jeder Verschiebung wird mittels der Methode `checkInScreenLocation()` geprüft, ob sich die Auswahl noch im positiven X-/Y-Bereich der Anzeige befindet. Dies soll verhindern, dass *Cells*, die sich im positiven, sichtbaren Anzeigebereich befinden, in den negativen, nicht sichtbaren Bereich, oberhalb oder links des Anzeigebereichs geschoben werden können. Nach dieser Prüfung wird jede *Cell* der Auswahl mit der Hilfsmethode `translateCell()` verschoben, jeweils in Abhängigkeit der berechneten Distanz und der momentanen Mausposition.

Der 3. Anweisungsblock behandelt die Fällen, in denen nur eine *Cell* ausgewählt ist. Dabei wird im Inneren des Blocks noch näher geprüft, um welchen Typ von *Cell* es sich handelt. Eine `InterfaceCell` wird z.B. nicht verschoben, sondern rotierten um ihren Router. Dazu werden die Hilfsmethoden `calcAngle()` und `rotateCell()` benötigt, um den entsprechenden Winkel zwischen Mausposition und Interface-Position und die daraus resultierende neue Koordinate auf der Umlaufellipse des Routers zu berechnen.

Die Behandlung einer `NetworkCell` ist relativ einfach: Sie kann direkt mit der Methode `translateCell()` verschoben werden ohne Berücksichtigung anderer Elemente oder besonderer Berechnungen vorab.

Handelt es sich um eine `RouterCell`, dann liegt implizit wieder eine Mehrfachauswahl vor, da auch jedes Interface des Routers mitbewegt werden muss. Dadurch ist die Anzahl an selektierten Elementen theoretisch immer größer als eins. Die Anzahl befindet sich aber noch auf eins, da nur der Router selektiert ist. In diesem Fall wird zuerst die Selektion erweitert. Die eigentliche Verschiebung wird dann wieder durch den 2. Anweisungsblock durchgeführt, dessen Kriterien nun durch die Selektionserweiterung erfüllt sind.

Die Methode `mouseReleased()` wird immer dann aufgerufen, wenn eine Maustaste nach dem Drücken wieder losgelassen wird, also nach einem *mousePressed*- oder *mouseDragged*-Event. Programmcode 12 zeigt den überschaubaren Methodenrumpf. Die Methode kann im Wesentlichen auf die Funktionen der Klasse `BasicGraphUI` zurück greifen. Deshalb ist der erste Aufruf auch die Methode der Oberklasse. Die eigenen Anweisungen haben die Aufgabe, Klassenvariablen der Mehrfachauswahl am Ende einer Mausinteraktion zurückzusetzen. Des Weiteren wird auch in der Variablen `lastFocus` festgehalten, welches das Objekt referenziert, das zuletzt den Fokus besaß.

```
01 @Override
02 public void mouseReleased(MouseEvent e){
03
04    super.mouseReleased(e);
05    // Reset focus and marquee variables.
06    marqOffset.setLocation(0, 0);
07    marqueeStart = true;
08    lastFocus = focus;
09    focus = null;
10 }
```

*Programmcode 12: Die mouseReleased-Methode.*

Die Methoden der Klasse `XTGraphUI` dienen zur Unterstützung des MouseListener und andere Listener und erfüllen die benötigten Berechnungen, wie die *Cell*-Selektion, Translationsvektoren-Berechnung, etc. Für die einzelnen Methoden verweise ich nochmals auf die Javadoc generierte Dokumentation. Die Funktionen sind dort alle kommentiert.

Die oben schon erwähnte Klasse `Utilities` wurde bereits von Pähler angelegt, um Methoden in einer Klasse zu sammeln, die sonst keiner Klasse funktional zugeordnet werden konnten. Die Klasse wurde um Methoden ergänzt, welche entweder die Sichtbarkeit von Objekten in JGraph verändern oder die Eigenschaften der JGraph-Instanz beeinflussen. Eine dieser Eigenschaften stellt die Skalierbarkeit des Graphen dar, wodurch die Möglichkeit besteht, die Anzeige zu verkleinern oder wieder auf ursprüngliche Größe zurückzusetzen.

Zum Abschluss der Vorstellung des *Packages* `frontend` wird noch die neu hinzugefügte Klasse `SSHTerminal` erwähnt. Sie ermöglicht das Öffnen eines Terminals aus der Topologie-Anzeige zu den einzelnen *Virtual Machines* (*VM*) des Szenarios. Die Klasse ist als eigenständiger Thread realisiert und läuft unabhängig von der RIP-XT-Anwendung. Dies bedeutet, dass die Terminals auch nach Beendigung des RIP-XTs weiterbe-

stehen und manuell geschlossen werden müssen. Dies ist auch so beabsichtigt, da die *VMs* unabhängig vom RIP-XT auf dem *Host*-System weiterlaufen und erst durch das Herunterfahren der VNUML-Simulation auch beendet werden.

Wichtige Voraussetzung für die SSH-Verbindung ist, dass die freie Terminal Emulation XTERM[45] auf den Computer installiert ist. Sie ist Bestandteil des X Window System der X.Org Foundation[46], welches auf den meisten unixartigen Betriebssystemen eingesetzt wird.

Jede Instanz der Klasse `SSHTerminal` erhält bei ihrer Erzeugung den Hostnamen der *VM,* zu der sie sich verbinden soll, und generiert daraus einen Kommando, welches von dem darunter liegenden Betriebssystem ausgeführt wird. Das Kommando besteht zum einem aus dem Startaufrufes eines XTERMs, zum anderen aus zwei Parametern für dieses Terminal. Der erste Parameter ist der Titel für die Konsolenleiste, welcher angezeigt wird. Der zweite Parameter ist ein Kommando, das in dem neu erzeugten Terminal ausgeführt werden soll, siehe Zeile 04 Programmcode 13.

```
01 public SSHTerminal(String hostName){
02
03    this.hostName = hostName;
04    command = "xterm -title "+hostName+" -e ssh root@"+hostName;
05 }
```

*Programmcode 13: Konstruktor der Klasse SSHTerminal.*

Das Kommando für den XTERM besteht aus dem Aufruf des SSH-Programmes, um eine Verbindung zu der *VM* als Benutzer Root aufzubauen. Der Name der *VM* reicht aus, da VNUML alle Namen und die zugehörige IP-Adressen in die hosts-Datei des Computers einträgt und so die Auflösung nach Namen ermöglicht. Über dieses Terminal können dann beliebigen Kommandos ausgeführt oder auf das Dateisystem der *VM* zugegriffen werden.

Dieser Überblick über das Package `frontend` und seine Klassen hat nicht den Anspruch auf Vollständigkeit. Um den gesamten Überblick zu erhalten, verweise ich auf die beiliegende Java Dokumentation auf der CD. Für das JGraph API möchte ich auf dessen Dokumentation hinweisen, die entweder als Online-Version[47] eingesehen werden kann oder sich bei der herunterladbaren Bibliothek befindet (siehe, Seite 22).

45 *XTERM*, Terminal Emulation für das X Window System. Das Projekt wird momentan von Thomas E. Dickey weitergeführt und ist frei erhältlich auf invisible-island.net/xterm/.
46 *X.Org Foundation*, Offizelle Webseite: www.x.org/wiki/
47 J*Graph API Dokumentation*, www.jgraph.com/doc/jgraph/.

## 5.2 Implementierung des Package jgraph2xml

Eine weitere wichtige Anforderung, die mit dem Neukonzept verbunden ist, ist die dauerhafte Speicherung der Positionen der Netzwerktopologie. Damit wird dem Benutzer ein ständiges Positionieren bei jedem Start abgenommen. Für diese Aufgabe wurde das *Package* `jgraph2xml` erstellt. Die unten stehende Tabelle zeigt das *Package* und die enthaltenen Klassen im Überblick.

| `Package: jgraph2xml` | | | |
|---|---|---|---|
| Name | neu | modifi-ziert | beibe-halten |
| `LoadSaveXML` | 13 | - | - |
| `PositionData` | 7 | - | - |

*Tabelle 2: Klassenübersicht des Package jgraph2xml.*

Für die permanente Speicherung der Positionsangaben wurde entschieden, diese Informationen direkt in eine Szenario-Datei zu schreiben. Ein Szenario in VNUML besteht aus einer XML-Datei, die Konfigurationen der einzelnen *VMs* und der zu erzeugenden Netzwerke beinhaltet. Die syntaktische Korrektheit einer VNUML-Datei wird durch die *vnuml.dtd*-Datei überprüft. Eine DTD[48]-Datei besteht aus Grammatikregeln, welche die genaue Syntax eines Dokumentes beschreiben. Die definierten Regeln geben dabei die genaue Baumstruktur der Elemente an und legen fest welche Attribute ein Element besitzt. Für ein Attribut können entweder zulässige, feste Werte definiert werden oder der Bezeichnerwert, der als Eingabe erlaubt ist. Aus diesem Grund müsste die DTD geändert werden, um eigene Positionsattribute an die relevanten XML-Elemente anzufügen. Um aber den Austausch von Szenarien nicht durch eine eigene DTD-Variante zu erschweren, wurde von dem Einfügen eigener Attribute abgesehen.

Als Ansatz der diese Problematik umgeht wurde die Verarbeitungsanweisung, die so genannte *Processing Instructions* gewählt. Eine *Processing Instruction* innerhalb eines XML-Dokumentes kennzeichnet sich durch das einleitende Tag "`<?`" und das schließende Tag "`?>`". Die *Processing Instruction* widerspricht dem Grundgedanken von XML. Da Daten eigentlich innerhalb eines XML-Dokumentes entweder durch Attribute oder als *Content* eines Elementes zu halten sind. Dadurch unterliegen die Daten den durch eine Schemataspache definierten Syntaxregeln und werden auch auf diese hin überprüft. Anders ist es bei einer *Processing Instruction*. Deren Inhalte werden vom Parser

48 *DTD*, Document Type Definition. Ist eine Schemataspache, die XML zur Erstellung von Grammatikregeln benutzt. Nachteil einer DTD ist, dass sie selbst wiederum kein XML-Dokument ist.

ignoriert. Deshalb muss eine Anwendung selbst dafür Sorge tragen, die Daten korrekt zu erstellen oder auf Korrektheit zu überprüfen. Trotz des Nachteils, die vorhandene XML-Struktur nicht nutzen zu können, wurde dies in Kauf genommen, um die Informationen möglichst ohne großen Aufwand in eine XML-Datei einzubringen. Um es mit den Worten von Charles Goldfarb[49] zu sagen:

> *„In one sense, the SGML processing instruction construct can be viewed as a throwback to that past, as it gives a user the opportunity to send system-specific markup to an application in its own language. In practice, though, processing instructions serve as a useful escape value for failures of rule-based application design or implementation. In a perfect world, they would not be needed, but, as you may have noticed, the world is not perfect.“* [GOL94]

Dieser Nachteil fällt aber gering aus, da der funktionale Ablauf des RIP-XTs nicht durch die Positionsergänzungen in irgendeiner Weise beeinträchtigt werden kann. Sie werden nur beim Laden oder Speichern benutzt. Selbst eine fehlerhafte *Processing Instructions* kann zu keinem Fehlverhalten des RIP-XT führen. Das Prüfen auf Korrektheit der Positionsinformationen in einer *Processing Instruction*, im Weiteren nur noch *PI* bezeichnet, wird in der Klasse `LoadSaveXML` erledigt.

Um mit XML-Dokumenten in Java zu arbeiten, wurde das JDOM[50] API gewählt. Das API steht zur freien Verfügung unter einer Lizenz, welche an die Apache Open Source License[51] angelehnt ist. Das API ermöglicht das Einlesen, Bearbeiten und Schreiben von XML-Dokumenten. Dazu erstellt es ein Java-basiertes *Document Object Model* des XML-Dokuments. Der Dokumentbaum baut sich dabei nicht wie bei plattformunabhängigen XML-APIs aus *Nodes*-Objekten auf, sondern jedes Element wird durch ein Objekt einer entsprechenden Java-Klasse repräsentiert. Zum Lesen und Schreiben greift JDOM auf Parser anderer APIs zurück und arbeitet dabei kompatibel zu anderen XML-Dokumentrepräsentationen, wie DOM[52] oder SAX[53]. Das API wurde wegen seiner einfache Handhabung und Vielseitigkeit beim Arbeiten mit VNUML/XML-Dokumente ausgewählt. Anwendung findet das API in der Klasse `LoadSaveXML`. Dort werden die

49 *Charles Goldfarb*, einer der Väter von *Standard Generalized Markup Language* (SGML). Der Sprache, auf der XML basiert.

50 *JDOM*, Offizielle Internetseite: www.jdom.org.

51 *The Apache Software Foundation*. Siehe: www.apache.org/licenses/

52 *DOM, Document Object Model*. Plattform und Sprachunabhängiges Interface für den Zugriff und das Ändern von XML-Dokumenten. Offizielle Internetseite: www.w3.org/DOM.

53 *SAX, Simple API for XML*. API zum sequenziellen parsen von XML-Dokumenten. Dabei werden Events erzeugt mittels derer sogenannte Rückruffunktionen aufgerufen werden, die eine Verarbeitung der Daten ermöglichten. Offizielle Internetseite: www.saxproject.org.

Positionsinformationen entweder aus einem XML-Dokument extrahiert oder in einem Dokument an die entsprechenden Elemente angefügt. Damit die Informationen zwischen verschieden Programmabläufen des RIP-XTs ausgetauscht werden können, wurde ein eigenes Datenformat erstellt. Dieses ist in der Klasse `PositionData` implementiert und soll zunächst einmal kurz vorgestellt werden, damit im Weiteren klar ist, wie die Daten verwaltet werden

### 5.2.1 Die Klasse PositionData

Alle extrahierten Positionsinformationen werden in einer eigenen Datenstruktur gehalten, um die Informationen zwischen einem JGraph und einem XML-Dokument in beide Richtungen austauschen zu können.

Ein Objekt der Klasse `PositionData` besteht aus einer Zeichenkette für den Namen, einem Point2D für die Koordinate und wiederum einem Vektor `PositionData`, siehe. Programmcode 14. Der Vektor wird über die Variable `children` referenziert, er ist aber anfänglich nicht initialisiert. Der Vektor wird erst erstellt, wenn ein Interface über die Methode `addChild()` eingefügt wird, dabei wird überprüft, ob schon eine Instanz des Vektors erzeugt wurde, siehe Zeile 16. Für ein Netzwerk-Element wird dieser Vektor nie initialisiert, da dieser keine Unterelemente besitzt. Die Klasse besitzt ansonsten noch Get-Methoden, um die Daten eines Objektes oder seiner Objektkinder auszulesen.

```
01 public class PositionData{
02
03    private String name;
04    private Point2D position;
05    private Vector<PositionData> children;
06
07    // Constructor
08    public PositionData(String name, Point2D position){
09
10       this.name = name;
11       this.position = position;
12    }
13    // Methods
14    public void addChild(String name, Point2D position){
15
16       if(children == null)
17          children = new Vector<PositionData>();
18       children.add(new PositionData(name, position));
19    }
20    // get methods...
21 }
```

*Programmcode 14: Die Klasse PositionData.*

### 5.2.2 Laden eines Szenarios

Ein Szenario kann über die `XTClientGUI` geladen werden. Dazu bietet sie mittels eines Menübefehls an, die JGraph-Topologie aus einem XML-Dokument zu erzeugen. Zusätzlich zum Erstellen der Topologie werden auch die Positionsinformation ausgelesen. Der Ablauf des Laden lässt sich mit dem Sequenzdiagramm in Abbildung 16 darstellen.

Innerhalb der Methode `getAddServersFromXMLMenuItem()`, durch das Menü aufgerufen, wird zuerst durch `setServersFromXMLConfig()` und deren Unterroutinen die JGraph Topologie erstellt, vgl. hierzu Kapitel 5.1. Anschließend werden die Positionsinformationen aus dem Dokument extrahiert. Dazu wird die Methode `getPositionsFromXMLConfig()` der Klasse `LoadSaveXML`. ausgeführt. Das Auswählen und Laden der XML-Datei wurde bereits in `getAddServersFromXMLMenuItem()` erledigt. Deshalb gibt sie die Referenz auf das Dokument an die Methode `getPositionsFromXMLConfig()` weiter. Die Klasse `LoadSaveXML` besitzt selbst die Methode `openLoadDialog()`, mit welcher eine Dateiauswahl möglich ist. Die Methode wurde aber nicht verwendet. Sie ist mit Ausblick auf eine Wiederverwendung der Klasse in anderen Anwendung angelegt.

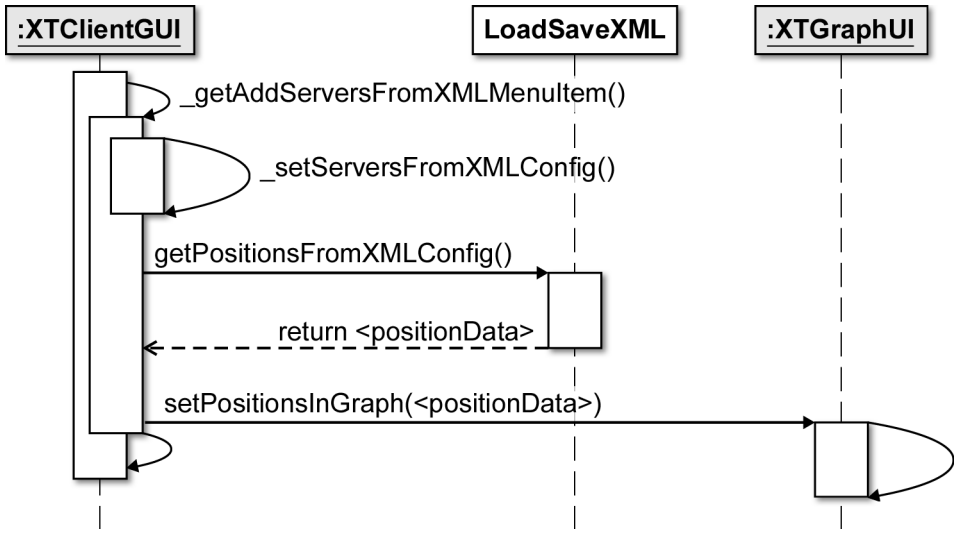


*Abbildung 16: Positionsextraktion beim Laden.*

Wie oben erwähnt, wird die Methode `getPositionsFromXMLConfig()` mit dem zuvor ausgewählten XML-Dokument aufgerufen. Mittels des SAXBuilders wird aus diesem Dokument ein JDOM-Dokument erzeugt, das alle Elemente des ursprünglichen Dokumentes enthält. Aus der Dokumentrepräsentation werden mittels Filterfunktionen des

API alle im Dokument enthaltenen *PIs* ausgelesen. Dies ergibt eine Liste von *PIs*, die auch eine leere Liste sein kann, wenn keine *PIs* im Dokument waren. Jede gefundene *PI* wird noch daraufhin überprüft, ob sie eine relevante Positionsinformation ist. Ob eine *PI* relevant ist bezüglich der Positionsextraktion, lässt sich anhand des *Target*-Parameters einer *PI* bestimmen. Alle *PIs*,die von der RIP-XT-Anwendung in ein Dokument geschrieben werden, besitzen eine Zeichenfolge als Identifikator, der dem *Target* entspricht. Der hierfür verwendete Identifikator ist die Zeichenfolge „*xtpos*“.

Allgemein ist zu dem Aufbau eines *PI*-Objekt in JDOM zu sagen: Es besteht aus zwei Parametern: einem *Target*-Parameter und einem *Data*-Parameter. Beide sind vom Typ `String`, also Zeichenketten. Der *Target*-Parameter dient als Identifikator, um *PIs* verschiedener Belange auseinander zu halten. Der *Data*-Parameter kann entweder eine Map mit *Name/Value*-Paaren sein oder aus einer beliebigen Zeichenkette bestehen.

Um die Daten aus einem solchen *Data*-Teil zu erhalten, müsste dieser mit einem eigenen Parser durchgegangen und die relevanten Information extrahiert werden. Eine Vereinfachung hierfür bietet das API an. Dazu muss die *Data*-Zeichenkette im sogenannten Pseudoattribute-Stil aufgebaut sein. Darunter versteht man, das die Zeichenkette sich aus Elemente der Struktur „*Name=Value“* zusammensetzt. In diesem Fall kann der *Value* eines Pseudoattributes explizit über Angabe des *Names* auslesen werden. Aus diesem Grund ist der *Data*-Parameter in der RIP-XT-Anwendung auch als Zeichenkette mit zwei *Name*-Werten aufgebaut. Zum einen dem *Name* x und zum anderen dem *Name* y. Der so definierte Zeichenkettenaufbau einer RIP-XT *PI* lässt sich aus der EBNF-Darstellung in Tafel 1 ablesen.

```
PI    ::= '<?', Target,' ', Data, '?>';
Target::= 'xtpos';
Data  ::= 'x=“', Double, '“', ' ', 'y=“', Double, '“';
Double::= Digit,{Digit}, {'.',Digit,{Digit}};
Digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
          | '9';
```

*Tafel 1: EBNF einer Processing Instruktion.*

Um die Positionsangabe einer *PI* mit anderen Programmteilen auszutauschen, wird ein `PositionData`-Objekt erzeugt, das die ausgelesene X- und Y-Werte als Position erhält und als Namen einen Bezeichner, der mit dem Bezeichner der JGraph-Objekten korrespondiert. Für Router und Interfaces kann dieser Name direkt aus Attributen des XML-Dokumentes gewonnen werden. So wird z.B für einen Router das *name*-Attribute des

*vm*-Elements verwendet. Bei einem Interface wird das *id*-Attribut des *if*-Elements verwendet und um das Präfix „*eth*“ ergänzt. Für ein Network existiert kein Attribut, das mit dem JGraph-Bezeichner korrespondiert. Der Bezeichner in JGraph besteht aus dem Netzwerkanteil der IP-Adresse der angeschlossenen Interfaces gefolgt von der Subnetzmaske in Slash-Notation. Hingegen ist in einem VNUML-Dokument ein Netzwerk durch ein *net*-Element repräsentiert, dessen *name*-Attribute mit dem *net*-Attribut eines Interfaces in Relation steht. Über diese Beziehung wird in VNUML festgelegt, welche Interfaces über welche Netze miteinander verbunden sind. Die VNUML-Bezeichner werden aber bei den Objekten des RIP-XT-Clients nicht verwendet, wodurch eine Zuordnung nicht möglich ist und somit auch keine Positionsinformation für Netzwerke zu speichern wäre.

Um nun den JGraph-Bezeichner zu erhalten, wird für bei der Erzeugung der Positionsangabe für ein Interface der Netzwerkanteil einer Interface-IP-Adresse berechnet. Dies ist aus der IP-Adresse und der Subnetzmaske des Interfaces möglich. Die so errechnete Netzwerkadresse wird zusätzlich mit dem Bezeichner des *net*-Attributes in eine *Lookup-Table* eingefügt. So erhält man für jedes Interface das zugehörige Netzwerk. Um doppelte Einträge zu vermeiden wird vor jeder Berechnung erst geprüft, ob der Netzwerkbezeichner schon in der *Table* ist. Falls ja, muss dieses Netzwerk nicht mehr eingefügt werden.

Durch diese Verfahren erhält man nach Behandlung aller Router und damit aller Interfaces eine komplette paarweise Liste der Netzwerke in der *Lookup-Table*, die aus der Netzwerk-Adresse, wie sie dem JGraph Namen entspricht, und dem VNUML-Namensbezeichner besteht. Durch diese Vorverarbeitung kann dann zu jeder *PI* eines Netzwerk-Elementes die Position extrahiert werden und die Netzwerkadresse als Name aus der angelegten *Lookup-Table* nachgeschaut werden.

Die extrahierten Positionen werden an die Instanz der `XTGraphUI` weitergeleitet, welche die Elemente der Positionsangaben mit den eigenen JGraph-Objekten vergleicht. Wird ein entsprechendes Objekte in der Topologie gefunden, dann werden die X/Y-Werte des Bound-Attributes entsprechend der gespeicherten Position gesetzt.

Um den Suchaufwand innerhalb der Graphen-Objekte möglichst gering zu halten, besteht eine Positionsangabe für einen Router aus seiner eigenen Position und einem Vektor aus den Positionen seiner Interfaces. Wie bereits oben erwähnt, ist die gesamte Datenstruktur mittels der Klasse `PositionData` erzeugt. Dies hat den Vorteil den Suchaufwand für Elemente zu reduziert. Im Falle eines Router-Elements besitzt dieser auch

eine Liste seiner Interfaces. Dadurch reduziert sich die Suchmenge für Positionsangaben eines Interfaces auf diese kürzere Liste.

Wurde für ein Positionselement kein JGraph-Objekt gefunden, erhöht dies einen Fehlerzähler. Nach Abschluss der Positionierung wird dem Benutzer mitgeteilt, wie viele Fehler auftraten und welche Objekte nicht gefunden wurden. Ein Fehler wirkt sich für die Positionierung folgendermaßen aus:

- Ist für einen Router keine passenden Positionsangaben gefunden worden, dann verbleiben er und seine Interfaces auf der automatisch generierten Erzeugungsposition, siehe Stichwort Positionierungsalgorithmus in Kapitel 2.5.

- Wird für ein Interface keine Information gefunden, aber für den zugehörigen Router, dann wird das Interfaces auf dem Router-Symbol positioniert.

- Für ein Netzwerke gilt das gleiche wie für ein Router, sie verbleiben auf ihrer ursprünglichen, automatisch generierten Erzeugungsposition.

Formatierungsfehler an den Positionsangaben, wie z.B. negative oder nicht als Zahl interpretierbare Informationen, werden anders behandelt. Diese Fehler werden in der Klasse `LoadSaveXML` durch das Setzen einer Default-Position abgefangen. Die Default-Position hat die Koordinate (-1, -1). Eine negativen Position hat zur Folge, dass das Objekt mit dieser Positionsinformation nicht neu gesetzt wird, sondern der Vorgang als Fehler gewertet wird und das zu setzende Objekt nach den oben genannten Kriterien weiter behandelt wird.

### 5.2.3 Speichern in einem Szenario

Das Sichern von Positionsinformationen lässt sich durch das Sequenzdiagramm in Abbildung 17 verdeutlichen. Die Klasse `XTClientGUI` bietet über ein Menübefehl die Möglichkeit, die Positionen eines Szenarios zu speichern. Dazu ruft die Methode erst `getPositionsFromGraph()` der Klasse `XTGraphUI` auf. Diese sammelt die Positionen und zugehörigen Namen von allen Routern, Interfaces und Netzwerken der JGraph Instanz. Für jedes Objekt wird ein `PositionData`-Objekt erzeugt. Die gesammelten `PositionData`-Objekte werden ihrerseits zu einem Vektor hinzugefügt. Der gesamte `PositionData`-Vektor wird von `getPositionsFromGraph()` an die `XTGraphUI` zurückgegeben.

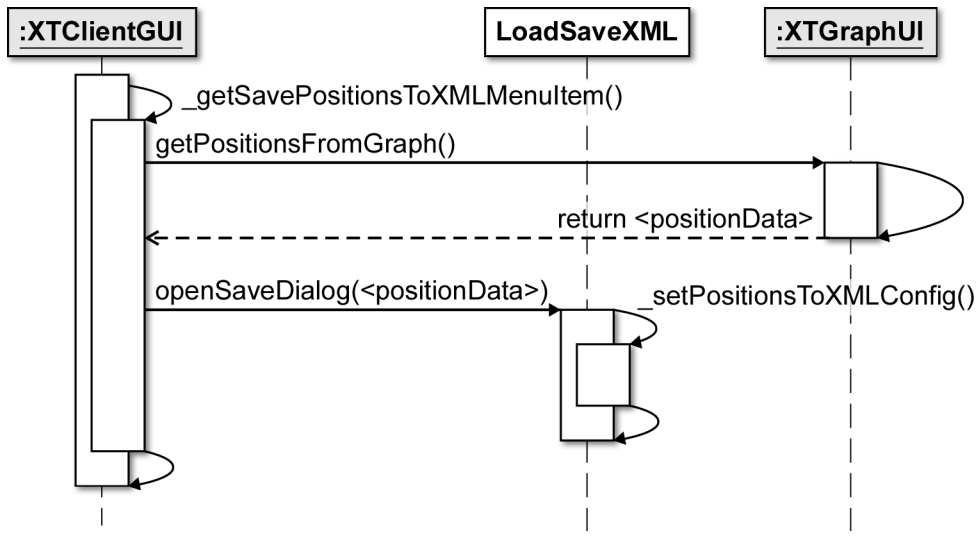


*Abbildung 17: Positionssicherung beim Speichern.*

Anders als beim Laden muss hier der Benutzer noch über einen Dialog die XML-Datei auszuwählen, in welcher die Informationen gesichert werden sollen. Theoretisch wäre es auch möglich ohne Benutzerinteraktion die Informationen in die selbe Datei zu speichern, welche auch zur Generierung des JGraph gewählt wurde. Dies hätte den Vorteil, dass der Benutzer keine Fehler in der Auswahl der Datei begehen könnte, aber andererseits würde man den Benutzer einschränken, und Speichern in Sicherheitskopien oder in eine gleiche Datei mit anderem Dateipfad unmöglich machen.

Diese unnötigen Einschränkungen wurden zu Gunsten der Sicherheit und des Komforts nicht umgesetzt, sodass an der freien Dateiauswahl festgehalten wurde.

Die Klasse `LoadSaveXML` bietet deshalb die Methode `openSaveDialog()`, die eine Benutzerauswahl zulässt, aber um die Auswahl zu vereinfachen das Verzeichnis vorschlägt, aus dem das Szenario geladen wurde. Die Methode wird von der `XTGraphUI` aufgerufen und gibt den `PositionData`-Vektor mit. Nach der Dokumentauswahl wird die eigentliche Methode aufgerufen, welche die Informationen in das ausgewählte XML-Dokument schreibt.

Die Methode `setPositionsToXMLConfig()` erledigt diese Aufgabe und parst dazu das ausgewählte XML-Dokument. Zuerst werden alle Elemente in einer Liste gesammelt, die einer *VM* entsprechen. Diese Liste von *VMs* wird durchgegangen und entsprechende korrespondierende Einträge in dem `PositionData`-Vektor gesucht. Anhand der Namen können beide miteinander verglichen werden. Wurde ein passendes XML-Element gefunden, wird aus den `PositionData`-Informationen eine *Processing Instruction* (*PI*) erzeugt und an das Element angefügt. Im Falle einer *VM* werden auch gleich die Positionen der Interfaces als *PI* gesetzt, da das `PositionData`-Objekt auch die Informationen der Interfaces des Routers als Vektor besitzt.

Nach dem Erzeugen der *PIs* der Router und Interfaces wird die Liste der Netzwerk-Elemente aus dem XML-Dokument geparst. Hier stellt sich die gleiche Problematik für die Netzwerke wie beim Laden. Die Objekte des `PositionData`-Vektors enthalten die JGraph-Bezeichner als Namen, die sich nicht einem dem VNUML-Netzwerk-Element zuzuordnen lassen. Deshalb wird auch bereits beim Speichern aus der errechneten Netzwerkadresse und denn Netzwerkbezeichner für jedes Interface ein *Lookup-Table*-Eintrag erstellt. Erst durch diese Einträge lassen sich die Netzwerk-Elemente im XML-Dokument identifizieren und den Objekten im `PositionData`-Vektor zuordnen.

Treten Fehler während dieses Prozesses auf, dadurch dass z.B. für ein `PositionData`-Eintrag kein entsprechendes XML-Element gefunden werden konnte, dann wird der Fehlerzähler erhöht und der Name des `PositionData`-Objekte in einen Fehlervektor eingetragen. Nach Abschluss des Speichervorgangs wird dem Benutzer durch ein Popup-Menü mitgeteilt ob Fehler auftraten und wenn ja, welche JGraph-Elemente nicht gespeichert werden konnten. Solange nicht zu 100% Fehler auftreten, werden alle zugeordneten Positionen in das Dokument gespeichert.

Ist für alle XML-Elemente eine `PositionData` gesucht worden, wird das neue Dokument erzeugt und gespeichert. Dem Benutzer wird zum Abschluss noch durch ein Popup-Menü das erfolgreiche Speichern mitgeteilt. Damit das Laden und Speichern funktioniert, werden noch weitere Hilfsmethoden in der Klasse `LoadSaveXML` benötigt,

die hier nicht alle im Einzelnen besprochen wurden, aber natürlich durch Dokumentation vollständig aufgeführt sind.

## 5.3 Erweiterung zum VNUML-Editor

Um den RMTI immer wieder in neuen Szenarien testen zu können, benötigt man variierende VNUML-Szenarien. Eine solche Szenario-Datei ist, wie bereits erwähnt, ein XML-Dokument, das strukturell durch die *vnuml.dtd*[54] bestimmt ist und semantisch durch den VNUML-Parser auf Korrektheit überprüft wird. Zum Erstellen des XML-Dokuments benötigt man somit nur einen Texteditors, möglichst mit XML-Syntax-Highlighting.

Durch einfaches Copy&Paste eines solchen XML-Dokuments kann so mit jedem Texteditor recht schnell eine umfangreiche Anzahl an *VM* erstellt werden. Das anschließend Konfigurieren erfordert, dann aber Konzentration und Aufwand, um z.B. Interfaces zu definieren und die Zuordnung zu den Netzwerken korrekt vorzunehmen.

Ein einfacherer Weg um neue Szenarien schnell und einfacher zu erstellen, wäre mittels eines Graphischen-Editors. Durch einfaches Drag&Drop sollte dann eine Topologie frei zusammenstellbar sein und Teile der Konfiguration mittels einfacher Maus-Interaktion vornehmenbar. Als Beispiel einer Stärke eines solchen Editors wäre die Konfiguration eines Interfaces. So muss man bei der bisherigen textuellen Konfiguration den Netzwerkbezeichner bei einem Interface richtig wählen und eine passende IP-Adresse auswählen, welche noch nicht für ein Netzwerk vergeben worden ist.

Die Konfiguration in einem graphischen Editor könnte folgendermaßen aussehen: Ein Interface wird einem Netzwerk durch Ziehen einer Kante vom Interface zu einem Netzwerk zugeordnet. Dabei erhält das Interface den Netzwerkbezeichner und bezieht die nächste frei zu vergebene IP-Adresse. Für den Benutzer werden diese Attribute automatisch gesetzt und dienen dann als Grundlage für das zu generierende XML-Dokument. Über das GUI eines Editors sollten auch die weiteren Attribute gesetzt werden können und veränderbar sein. Der Editor sollte den Benutzer auch dahingehend unterstützen, dass er mögliche Parameter für die einzelnen Elemente des Szenarios zur Auswahl stellt.

Ein solcher Editor könnte auf den JGraph-Anteilen der RIP-XT-Anwendung basierend erstellt werden, um die gesamten graphischen Darstellung und die Interaktion mit dieser zu erhalten. Zusätzlich müssten „nur noch“ die Editor-Anteile implementiert wer-

54 *vnuml.dtd*, siehe hierzu Seite 54.

den. Um dies zu realisieren müssten die JGraph-Anteile aus der RIP-XT-Anwendung heraus gelöst werden.

Diese Anteile liegen gesammelt im *Package* `frontend` und sind an fünf Stellen mit anderen Klassen des RIP-XT verknüpft. Damit eine Unabhängigkeit von der RIP-XT-Anwendung gewonnen werden kann, sind folgende Abhängigkeiten zu betrachten:

1. Die Klasse `XTGraphGUI`: Hier wird die JGraph-Instanz erzeugt und konfiguriert sowie die Objekte erzeugt.

2. Jedes `RouterCell`-Objekt bekommt beim Erzeugen ein `XTServer`-Objekt des `backends` als *userObject* übergeben, das ein Abbild einer *VM* und deren Interfaces ist.

3. Jedes `InterfaceCell`-Objekt bekommt beim Erzeugen von seinem `XTServer` ein `Network`-Objekt des `backends` als *userObject* übergeben, welches es repräsentiert.

4. In der Klasse `XTGraphUI` werden RIP-XT spezifische Methoden aufgerufen.

5. In der Klasse `Utilities` werden RIP-XT spezifische Methoden aufgerufen.

Um diese Abhängigkeiten aufzulösen können folgende Hilfestellungen berücksichtigt werden:

Zu Punkt eins: Das Erzeugen einer JGraph-Instanz kann natürlich in jeder anderen Anwendung erfolgen und erfordert keinerlei RIP-XT spezifischen Anforderungen. Das Erzeugen der JGraph-Objekte hängt eng mit den nachfolgenden Punkt zwei und drei zusammen. Sind diese erfüllt, dann verläuft die weitere Instantiierungen der Objekte automatisch ab.

Zu Punkt zwei und drei: Diese Abhängigkeit ist sehr einfach aus dem RIP-XT herauszulösen, da das *userObject* nicht zwingend notwendig ein `XTServer` oder `Network` des `backends` sein muss. Die wichtigste Methode, die JGraph von diesen Objekten benutzt, ist die `toString()`-Methode, die jedoch von jeder andere Objektklassen zu Verfügung gestellt werden kann.

Zum vierten Punkt: Hier beschränkt sich der Aufwand auf das Entfernen der inneren Klassen `XTActionListener` und `HighlightListeners`, sowie deren Aufrufstellen in der RIP-XT-Anwendung. In der Methode `mousePressed()`, der inneren Klasse `XTMouseHandler`, müssen im 2. Anweisungsblock die `else`/`if`-Anweisungsblöcke für den

`InterfaceCellView` und den `NetworkCellView` entfernt werden. All diese Methoden sind nur für die Funktionalität des RIP-XTs notwendig.

Zu Punkt fünf: In der Klasse `Utilities`, kann die Methode `generateExceptionDialog()` entfernt werden. Deren Aufrufe befinden sich in den unter Punkt vier beschrieben Event-Behandlungsklassen.

Zusätzlich können die Klassen `SSHTeminal` und `XTGraphGUI` ganz aus dem *Package* `frontend` entfernt werden. Es wird eher praktikabel sein ein neues GUI für eine neue Anwendung zu erstellen, als dass man aus der bestehenden `XTGraphUI` alle RIP-XT-Methoden entfernt und diese weiter modifiziert.

Sind diese Punkte erfüllt, können die JGraph-Klassen ohne Weiteres wiederverwendet werden. Das gesamte *Package* `frontend` kann somit in eine beliebige Anwendung eingefügt werden und trotzdem seine Funktionalität behalten. Sollen Funktionalitäten hinzugefügt werden, sollte das Kapitel 4 »JGraph« hilfreich sein, den richtigen Einstiegspunkt für Veränderungen zu finden.

# 6 Ausblick

Der Neuentwurf der Topologieanzeige bietet in seiner jetzigen Form ein einfacheres und übersichtlicheres Arbeiten mit der RIP-XT-Anwendung. Besonders in Anwendungsfällen mit größeren VNUML-Szenarien ist die Absicherung der Positionen und die Skalierung der Anzeige eine starke Erleichterung. Die abgeschlossene Arbeit bietet aber auch mit Hinblick auf eine Wiederverwertung der JGraph-Komponenten, z.B. für einen VNUML-Editor, eine gute Grundlage, um ein solches Projekt zu realisieren.

Durch die Wahl des JGraph APIs wird es auch ermöglicht sein noch weitere Aspekte, welche für eine Editor-Anwendung von Interesse sind, aus der Funktionalität des APIs einzubringen. So ist z.B. eine Undo-Funktion bereits Bestandteil von JGraph und kann einfach hinzugefügt werden, um das Arbeiten in einem Editor benutzerfreundlicher zu gestalten.

Wie in Kapitel 5.3 »Erweiterung zum VNUML-Editor« beschrieben, ist eine Wiederverwenden der bisher erstellten Komponenten mit geringem Aufwand möglich. Die Integration in eine neue Anwendung sollte auch ebenso möglich sein.

Der Hauptschwerpunkt wird auf dem Erstellen einer funktionalen und übersichtlichen Benutzeroberfläche und den Schnittstellen zur Manipulation der Objekte liegen. Im Besonderen sollten benutzerfreundliche Interaktionen zur Konfiguration der einzelnen Szenarien-Objekte gefunden werden, um die Vorteile eines graphischen Editors zu nutzen. Das anschließende Speichern eines validen XML-Dokuments sollte mittels der schon vorhanden Klasse `LoadSaveXML` und der Funktionalität des JDOM-APIs ohne Weiteres möglich sein.

Das Erstellen eines VNUML-Editors ist sicher ein spannendes und umfangreiches Projekt, das noch einer genaueren Anforderungsanalyse zu unterziehen ist, um den Nutzen für einen möglichen Anwender so groß wie möglich zu gestalten. Es wird aber aufgrund seiner Komponenten immer ein in Umfang und Funktionalität erweiterbares Projekt sein, an dem noch weitere Arbeiten teilhaben werden.

# Anhang

## A. Exkurs CTI

Das *Count-to-Infinity*-Problem (CTI, [PET03]) ist ein Fehler, der während der Konvergenzphase eines Netzwerkes.auftreten kann Man versteht darunter das kontinuierliche Ansteigen einer Routenmetrik innerhalb eines Netzwerkes bis zur Unerreichbarkeit, also dem *Infinity*-Wert des RIPs.

Damit dies Problem auftritt, muss eine bestimmte Konstellation in einem Netzwerk erfüllt sein. Um eine allgemeine Beschreibung dieses Problems zu geben, bedient man sich eines einfachen Topologie-Aufbaues, dem sogenannten Y-Szenario. Das Y-Szenario beschreibt den minimalen Aufbau einer Topologie, in welchem ein CTI erzeugt werden kann. Sie besteht aus einer Schleife von mindestens drei Routern, die entweder direkt oder über Netzwerke miteinander verbunden sind. Von mindestens einem der Schleifenrouter muss ein Pfad zu zwei weiteren in Reihe verbundenen Routern abführt. Damit besteht die gesamte Topologie aus vier Routern und fünf Netzwerken[55], siehe Abbildung 18.

In einer solchen Topologie muss nach Erreichen der Konvergenz eine bestimmte Ereignisfolge auftreten, damit ein CTI erzeugt wird:

1. Der Router `R1` verliert die Verbindung zu Router `R4` über sein Interface `4.1`. Er wird weiterhin Updates an seinen Nachbarn `R2` und `R3` über die Route zum Netzwerk `5.0` senden solange der *timeout*-Timer für dieses Netzwerk nicht abgelaufen ist (Standartwert entspricht 180 Sek.). Ist bis dahin kein Update von `R4` eingegangen, markiert `R1` die Route zu dem Netzwerk `5.0` als unerreichbar mit der Metrik 16.

2. Anschließend sendet `R1` ein Update an seinen Nachbarn mit der neuen Metrik. Einer der beiden, `R2` oder `R3`, erhält diese Update nicht und besitzt somit noch einen alten Wert in seiner Routing-Tabelle (im Beispiel wird `R2` ausgewählt, das Update nicht erhalten zu haben).

3. `R2` sendet sein periodisches Update, in der das ausgefallene Netzwerk `5.0` noch als erreichbar geführt ist, an seine Nachbarn `R3`.

55 *Netzwerk*, jedes Interface definiert hierbei ein Netzwerk. Besitzen zwei Interfaces das gleiche Präfix, dann liegen sie im gleichen Netzwerk - z.B. Interface `5.1` und `5.2` liegen somit im Netzwerk `5.0`.

4. R3 übernimmt diesen Eintrag, da das Update eine bessere Metrik besitzt als sein momentaner Eintrag für das Netzwerk. Er sendet die Aktualisierung an R1 weiter, der am Schleifenanfang sitzt.

5. R1 wird ebenfalls die scheinbar bessere Route übernehmen und wieder an der Urheber der Fehlinformation (R2) senden. R2 erhält somit ein Update mit einer schlechteren Metrik für das Netzwerk 5.0, weiß aber, dass er dieses Netz über R1 gelernt hat und nimmt es deshalb an.

6. Im Folgenden wird er die scheinbar aktualisierte Metrik wieder an seinen Nachbarn R3 senden, welcher seinerseits von einer Aktualisierung ausgehen wird. Dies läuft immer so weiter.

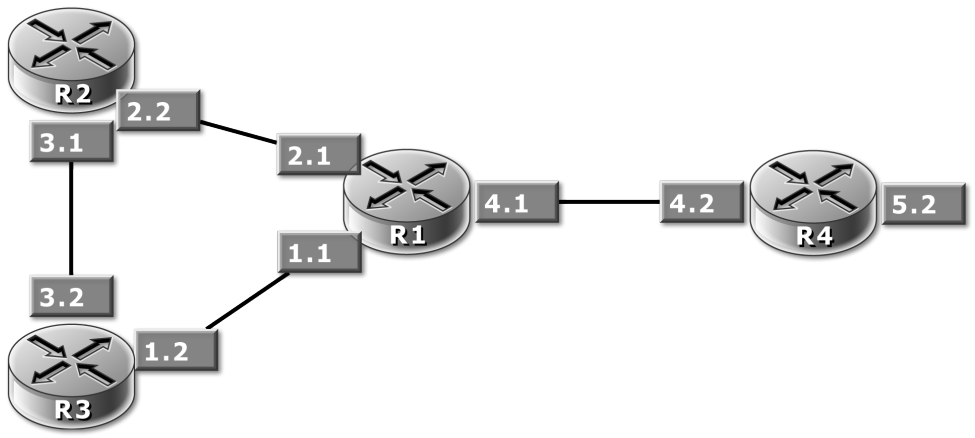


*Abbildung 18: CTI-Minimaltopologie.*

Die Route wird nun in der Schleife von einem Router zum nächsten weitergesendet und dabei jeweils die Metrik um eins erhöhen. Jeder Router glaubt aber noch eine erreichbare Route zu dem ausgefallenen Netzwerk 5.0 zu besitzen. Diese Metrikspirale geht soweit, bis die maximale Metrik 16 erreicht ist und alle beteiligten Router das Netz erneut als unerreichbar eintragen werden. Somit ist ein CTI kein bleibender Fehler, aber er belastet Netzwerke, weil er die Konvergenz dieser verhindert. Fehlende Konvergenz führt zu fehlgeleiteten Pakets innerhalb dieser betroffenen Netzwerke und kann damit zu Überlastung der Router führen.

## B. JGraph Lizenz

## C. JDOM Lizenz

*$Id: LICENSE.txt,v 1.11 2004/02/06 09:32:57 jhunter Exp $*



*Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:*

1. *Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.*
2. *Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.*
3. *The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact <request_AT_jdom_DOT_org>.*
4. *Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management <request_AT_jdom_DOT_org>.*

*In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following:*

> *"This product includes software developed by the JDOM Project (http://www.jdom.org/)."*

*Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos.*

Fortsetzung auf der nächsten Seite...

*THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILIT OF SUCH DAMAGE.*

*This software consists of voluntary contributions made by many individuals on behalf of the JDOM Project and was originally created by Jason Hunter <jhunter_AT_jdom_DOT_org> and Brett McLaughlin <brett_AT_jdom_DOT_org>. For more information on the JDOM Project, please see <http://www.jdom.org/>.*

# Literaturverzeichnis

[SCH99]: **Schmid, Andreas**
»RIP-MTI: Minimum-effort loop-free distance vector routing algorithm«
1999, Universität Koblenz-Landau.

[KOC05]: **Koch, Tobias**
»Implementation und Simulation von RIP-MTI«
2005, Universität Koblenz-Landau.

[PAE06]: **Pähler, Daniel**
»Extern steuerbare Routing-Updates im RIP-Daemon der Quagga-Programmsuite«
2006, Universität Koblenz-Landau.

[BOH08]: **Bohdanowicz, Frank**
»Weiterentwicklung und Implementierung des RIP-MTI-Routing-Daemons«
2008, Universität Koblenz-Landau.

[BAL99] : **Balzert, Helmut**
»Lehrbuch Grundlagen der Informatik«
1999, Spektrum Akademischer Verlag.

[LAN07]: **Lange, Stefan**
»Zentrale Betrachtung von Routing-Informationen zur Analyse des Konvergenzverhaltens verschiedener RIP-Algorithmen und Unterstützung des Generierens von Testfällen«
2007, Universität Koblenz-Landau.

[CIS]: **Cisco Systems Inc.**
»Network Topology Icons«
http://www.cisco.com/web/about/ac50/ac47/2.html, Abruf: 04.06.2010.

[DAH06] : **Dahm, Markus**
»Grundlagen der Mensch-Computer-Interaktion«
2006, Pearson Studium.

[JGM]: **JGraph Ltd.**
»JGraph and JGraph Layout ProUser Manual«
http://www.jgraph.com/pub/jgraphmanual.pdf, Abruf: 10.01.2010.

[MVC]: **Reenskaug, Trygve M. H.**
»MVC XEROX PARC 1978-79«
http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, Abruf: 15.05.2010.

[JBP]: **Sun Microsystems, Inc.**
»Java BluePrints: Model-View-Controller«
http://java.sun.com/blueprints/patterns/MVC-detailed.html, Abruf: 04.06.2010.

[JSE6a]: **Oracle/Sun**
»Interface: MouseListener«
http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/api/java/awt/event/MouseListener.html, Abruf: 21.07.2010.

[JSE6b]: **Oracle/Sun**
»Klasse: MouseAdapter«
http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/api/java/awt/event/MouseAdapter.html, Abruf: 21.07.2010.

[PGIT]: **Chacon, Scott**
»Pro Git«
http://progit.org/book/, Abruf: 15.07.2010.

[GOL94] : **Goldfarb, Charles F.**
»The SGML Handbook«
1994, Clarendon Press.

[PET03] : **Peterson, Larry L.; Davie, Bruce S.**
»Computernetze«
2004, dpunkt.verlag.

# Abbildungsverzeichnis

# Programmcodeverzeichnis