Generierung von GEF Editoren für Modellgetriebene Entwicklung

Bernd Kolb, *bernd@kolbware.de*, *www.kolbware.de*
Markus Völter, *voelter@acm.org*, *www.voelter.de*

Modellgetriebene Softwareentwicklung ist eines der Themen die derzeit in aller Munde sind. Von MDSD spricht man wenn die Software teilweise oder vollständig aus Modellen generiert wird (Details siehe [SV05]). Diese Modelle sind auf eine spezielle Domäne abgestimmt. Um ein Modelle beschreiben zu können wird eine Domain-Specific Language entworfen. Diese DSL drückt die Sachverhalte der Domäne kürzer und prägnanter aus als dies eine der herkömmlichen 3GL Sprachen könnte. Die DSL kann sowohl eine textuelle als auch eine graphische Notation haben. Als graphische Notation bietet sich auf den ersten Blick UML an. Allerdings ist UML oft ungeeignet, da die UML-Syntax nicht oder nur schwer in der Lage ist die Sachverhalte der betreffenden Domäne in geeigneter Form auszudrücken. Wenn also ein UML-Tool ungeeignet ist stellt sich die Frage was dann zur Modellierung grafischer DSLs verwendet werden kann.

Eine Möglichkeit ist es die Editoren mit Hilfe des Eclipse Graphical Editor Framework, kurz GEF [GEF], selbst zu erstellen. Die eigens erstellen Editoren bieten eine Reihe von Vorteilen:

- Die Syntax (Symbolik) kann exakt auf die Bedürfnisse der Domäne abgestimmt werden
- Eine Echtzeitprüfung des gesamten Modells gegenüber dem Metamodell der Domäne ist mit wenig zusätzlichem Aufwand möglich
- Durch die Integration in Eclipse lässt sich eine nahtlose Integration in den Entwicklungsprozess erreichen

Leider jedoch ist die Erstellung der Editoren recht aufwändig; GEF und Draw2D sind nicht einfach zu benutzen. In diesem Artikel wollen wir GEF kurz vorstellen und dann zeigen wie die für eine bestimmte DSL nötigen Editoren mittels eines Generators einfach generiert werden können. Das hier gezeigte Vorgehen basiert auf dem openArchitectureWare Framework [oAW].

GEF

GEF ist ein Projekt der Eclipse-Community. Es erlaubt die – verhältnismäßig – einfache Erstellung von Grafischen Editoren für bestimmte Anwendungsmodelle. Das SWT-basierte Draw2D-Grafikframwork wird als zugrunde liegende Graphik-API benutzt. GEF kann dabei als eine Abstraktionsschicht über Draw2D angesehen werden. Es folgt streng dem Model–View–Controller Pattern. Änderungen am Modell werden direkt im View sichtbar. Umgekehrt wirkt sich das Editieren des Views unmittelbar auf das zugrunde liegende

Modell aus. GEF ist völlig anwendungsunabhängig und kann somit für beliebige Modelle eingesetzt werden, die sich als Graph darstellen lassen.

Modelle und Metamodelle

Bevor wir uns der Generierung der Editoren widmen, möchten wir kurz eine Übersicht über die verschieden Modelle und Metamodelle geben mit welchen wir uns gleich beschäftigen werden, sowie kurz die Funktion des Generators erläutern.

Bei einem Generator handelt es sich um ein Stück Software, welches aus einem Eingabemodell mit Hilfe von Generierungs-Templates Code generiert. Das Eingabemodell ist eine Instanz des Metamodells; letzteres wird dem Generator auch als Input zur Verfügung gestellt, es liegt in Form von Javaklassen vor; das Modell wird durch Instanzen dieser Klassen repräsentiert.

Ein Modell kann in verschieden Repräsentationen vorliegen. So kann ein solches Modell z.B. als XML File auf Platte gespeichert sein. Für die Generierung ist es jedoch notwendig, das Modell in Instanzen der Metaklassen zu überführen. Dies übernimmt ein so genannter Instantiator. Abbildung 1 zeigt dies. Die benötigten Modelle, Metamodelle und ihre Verarbeitung zeigt Abbildung 2, die im Folgenden beschrieben wird.

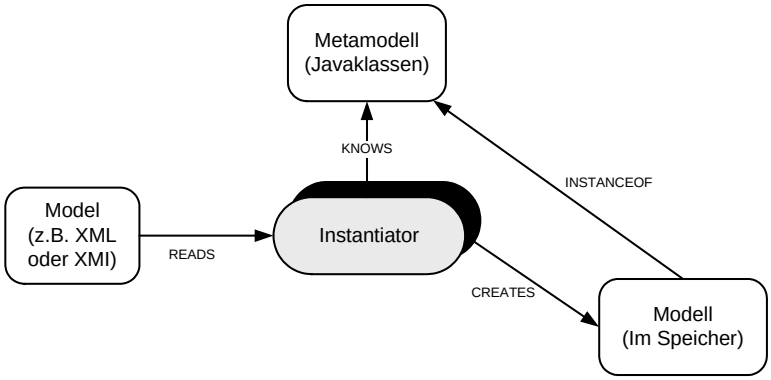

Abbildung 1: Instantiierung des Modells

Die Anwendungsentwicklung beginnt mit dem Entwurf eines Domänenmetamodells (oben links in Abbildung 2). Dies geschieht mittels eines UML-Tools. Dieses in UML beschriebene Domänenmetamodell muss nun für die Verwendung im Generator in Javaklassen umgewandelt werden. Dies wird größtenteils durch einen Generator übernommen (der Metamodellgenerator). Als Eingabe dient diesem Generator das UML Modell (als XMI Export) sowie ein Satz Templates. Als Output bekommt man zum einen eine Javarepräsentation des Domänenmetamodells und zum zweiten eine Hilfsklasse welche uns später bei der Erstellung des Modells für den GEF Editorgenerator hilfreiche Dienste leisten wird.

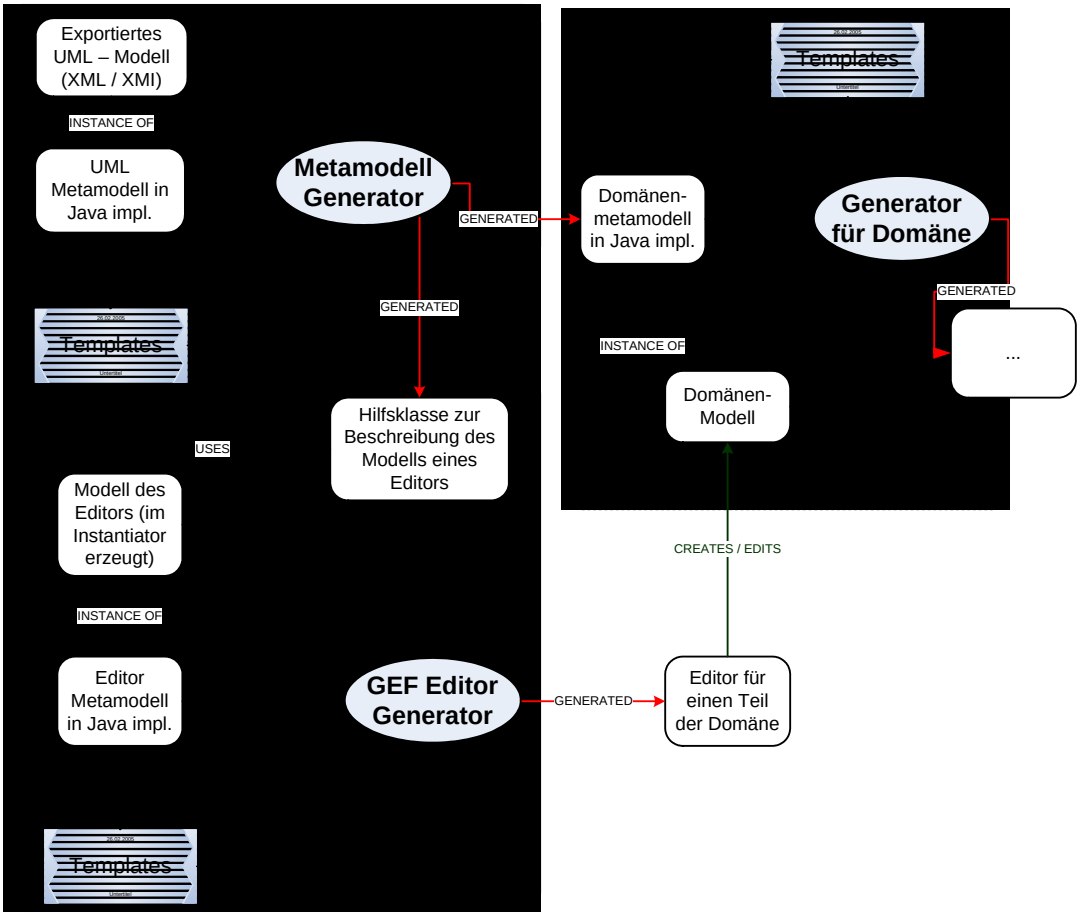

Abbildung 2: Benötigte Modelle und Metamodelle, sowie deren Verarbeitung

In Folgenden müssen wir zwischen zwei unterschiedlichen Metamodellen unterscheiden. Die Konzepte werden unten anhand eines ausführlichen Beispiels verdeutlicht.

- Das Domänenmetamodell: Dieses repräsentiert die fachliche Domäne, für welche die Editoren erstellt werden soll. Es wird in UML modelliert und durch den Metamodellgenerator in Javaklassen umgewandelt.
- Das Editormetamodell: Diese Metamodell beinhaltet Elemente die benötigt werden um einen Editor zu beschreiben; also im wesentlichen Knoten und Beziehungen zwischen Knoten. Zur Beschreibung des Modells des Editors werden Elemente aus dem Domänenmetamodell verwendet (s.u.). Aus diesem Editormodell wird dann später unser Editor generiert, welcher beliebige Instanzen, also Modelle, des Domänenmetamodells bearbeiten kann. Ein, durch die Editoren bearbeitetes Modell, wird dann später evtl. wieder als Input für einen weiteren Generator benutzt oder in einer anderen Art und Weise interpretiert (oben rechts).

Vom Konkreten Editor zum Editormetamodell

Ein graphischer Editor besteht üblicherweise aus folgenden Komponenten:

- Die Palette, mit deren Hilfe können neue Elemente im Editor angelegt und verbunden werden.
- Das Diagramm selbst
- Dem Properties-View mit dessen Hilfe die Eigenschaften eines Modellelements editierbar sind.
- Optional dem Outline-View welcher einen hierarchischen Überblick der auf dem Diagramm dargestellten Modellelemente gibt. Zusätzlich kann dieser View auch einen das Diagramm verkleinert darstellen, was vor allem bei großen Diagrammen zwecks Überblick und Navigation nützlich ist.

Da die Templates eines Generators immer für ein Metamodell erstellt werden, müssen wir erst die Stufe zwischen konkretem Modell des Editor und Editormetamodell überwinden.

- Auf einem Diagramm werden Elemente dargestellt. Jedes Element stellt eine Instanz einer bestimmten Domänenmetamodellklasse dar. Durch Regeln soll es soll möglich sein, das Erstellen sowie Löschen und Ändern in einem bestimmten Kontext erlauben oder verbieten zu können.
- Die dargestellten Elemente können Eigenschaften besitzen. Diese Eigenschaften werden als Properties bezeichnet. Properties charakterisieren ein dargestelltes Element näher und können somit Auswirkungen auf die visuelle Repräsentation eines Elements haben.
- Die dargestellten Elemente können in zwei unterschiedlichen Arten zueinander in Beziehung stehen:
    - Zum einen kann es sich um eine Verbindung handeln, welche durch eine Linie repräsentiert wird. Eine solche Linie selbst kann ebenfalls wieder ein Modellelement repräsentieren.
    - Zum zweiten ist eine Eltern-Kind-Beziehung möglich. Dies bedeutet, dass ein Element innerhalb eines anderen dargestellt wird.

Zu einem Editor gehört zusätzlich noch eine Palette:

- Eine Palette hat Gruppen.
- Eine Gruppe beinhaltet Elemente die später auf dem Editor erzeugt werden können. Es kann sich dabei sowohl um ein neues oder bestehendes Element, als auch als auch um eine Linie handeln die zwei Elemente miteinander verbindet.

Folgendes Bild zeigt wie die eben beschriebenen Dinge zueinander in Beziehung stehen.

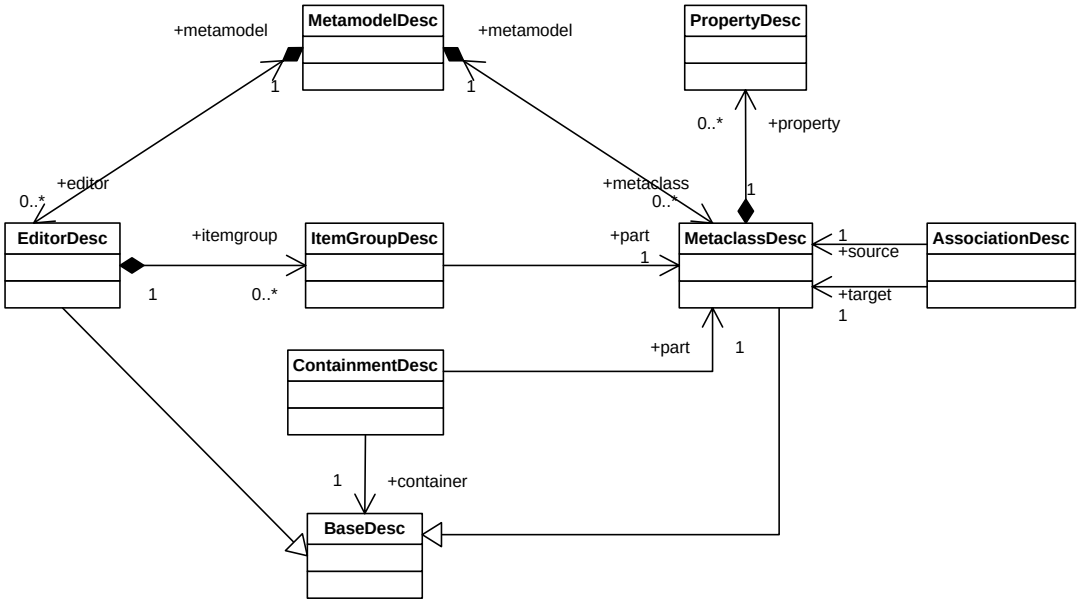

Abbildung 3: Metamodell für die Beschreibung von Editoren

Ein MetamodellDesc-Objekt dient als Container für Editorbeschreibungen und Klassenbeschreibungen (MetaclassDesc). Eine solche Klassenbeschreibung dient dazu die Klasse des Domänenmetamodells die später auf dem Editor bearbeitet werden soll dem Editor-Generator bekannt zu machen. An eine solche Beschreibung werden noch Beschreibungen für die einzelnen Properties angeheftet. Diese geben z.B. über den Typ des Property Auskunft und welche Werte eventuell in einer Auswahlliste zur Verfügung gestellt werden sollen.

Nun müssen noch die Beziehungen zwischen den Domänenmetaklassen beschrieben werden. Dies geschieht durch Subklassen von ContainmentDesc für Eltern-Kindbeziehungen bzw. AssociationDesc für Verbindungen zwischen Elementen. Die Subklassen werden verwendet um die Kardinalität der Assoziation oder Komposition anzugeben. Eine konkrete Subklasse wäre z.B. eine OneToManyAssociationDesc.

Aus diesen Angaben kann dann die nötige GEF Infrastruktur generiert werden, nur die Figures (also die konkreten grafischen Shapes) müssen von Hand implementiert werden; dazu ist ein entsprechendes Framework vorhanden. Das folgende Beispiel erläutert den Editorgenerierungsprozess.

Ein Beispiel

Stellen wir uns vor, wir wollten ein Tool entwickeln mit dem man Firewalls in einem Netzwerk administrieren können soll. Dazu muss zum einen die Hardwareinfrastruktur beschrieben werden: Welche Rechner gibt es, welche Router oder Accesspoints sind vorhanden und wie sind diese miteinander verbunden. Als nächstes muss die Software beschrieben werden: Welche Software kommuniziert über welche Ports mit welcher anderen Software. Die Instanzen der Software sollen dann auf die unterschiedlichen Rechner verteilt werden. Zwischen den verschiedenen Softwarekomponenten gibt es Abhängigkeiten. So kann z.B. Komponente A nur dann funktionieren wenn Komponente B ebenfalls installiert ist. Wenn A auf einem Computer installiert wird soll B automatisch ebenfalls installiert werden. Diese Logik soll zwar nicht im Editor hinterlegt werden, allerdings muss dieser auf ein solches Ereignis geeignet reagieren.

Während des Editierens des Modells soll der Konfigurator der Firewalls gewarnt werden sobald zwei Softwarekomponenten miteinander kommunizieren wollen, dies aber aus physikalischen Gründen nicht möglich ist.

Es bietet sich an für das Problem zwei Editoren zu erstellen: Ein Editor beschreibt die Hardwareinfrastruktur und die darauf installierte Software und der zweite beschreibt die Kommunikation der Softwarekomponenten untereinander. Im Beispiel hier konzentrieren wir uns aus Platzgründen auf den für die Modellierung der Hardwareinfrastruktur und der darauf installierten Software. Abbildung 4 zeigt das betreffende Domänenmetamodell.

HardwareInfrastructure ist ein Container. Er hält alle miteinander kommunizierenden Geräte sowie die Cables mit welchen die Geräte verbunden sind. Es gibt zwei unterschiedliche Arten von Geräten: Geräte auf welchen Software installiert die kommunizieren will und solche, die einfach nur anfragen weiter delegieren und Netzwerke miteinander verbinden. Jedes dieser Geräte hat eine gewisse Anzahl an HardwarePorts. Jeder HardwarePort kann 0..1 Cable haben. Ein Cable muss an jedem Ende mit einem HardwarePort verbunden sein.

Mit Hilfe des Metamodellgenerators werden aus den oben dargestellten UML Diagramm die Java-Repräsentationsklassen des dargestellten Metaklassen generiert. Diese enthalten Methoden wie z.B. ein remove<Property> add<Property> oder set<Property> sowie dispose um ein Element und alle darauf deutenden Assoziationen zu löschen. Ebenfalls werden die Elemente mit gelöscht welche durch eine Komposition mit dem Element verbunden sind.

Zusätzlich wird, wie bereits oben erwähnt, eine Hilfsklasse generiert, welche die UML Elemente bereits so aufbereitet, dass sie sofort für die Beschreibung des Editors verwendet werden können.

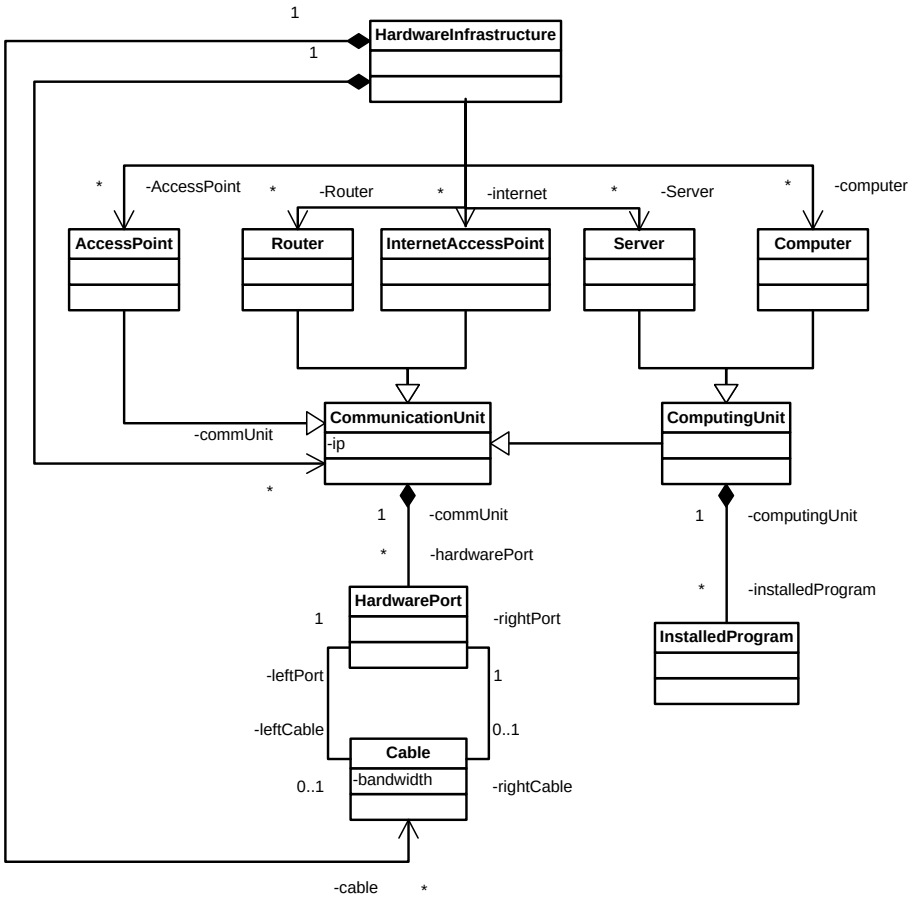

Abbildung 4: Domänenmetamodell für das Firewall-Beispiel

Der Weg zurück: Vom Metamodell zum Editor

In einem so genannten Instantiator wird das Modell für einen konkreten Editor beschrieben. Wir verwenden hier kein XML o.ä., sondern "programmieren" das Modell mit entsprechenden API Aufrufen. Zunächst wird das Domänenmetamodell bekannt gemacht, für das der Editor erstellt werden soll. Hierfür dient die Methode fillMetamodel.

```
protected void fillMetamodel(MetamodelDesc metamodel) {
  HashMap connectcmds = new HashMap();
  connectcmds.put( NWCable.class.getName(),
  "de.kolbware.mddsample.network.gef.hardware.CableConnectCmd");
  GefInstantiatorHelper.createAssocDescsAndMetaClassDescs(
     metamodel, connectcmds);
```

Hier ist schon der erste Hook zu sehen mit welchem Einfluss auf den späteren Editor genommen werden kann: Es wird der voll qualifizierte Name einer Kommandoklasse übergeben. Später wird zur Laufzeit des Editors der in dieser Klasse vorhandene Code ausgeführt. Dieses Konzept wird für viele der eingebauten Hooks verwendet. So ist es z.B. auf dieselbe Weise möglich auf Events aus dem Domänenmodell zu reagieren, oder bei Änderungen von Properties ein Command auszuführen.

Als nächstes wird die Beschreibung einer Domänenklasse angelegt, die später in dem Editor zu sehen sein soll. Danach werden einer solchen Klassenbeschreibung die Properties hinzugefügt die später bearbeitbar sein sollen. Der aus dem UML Metamodell generierte GefInstantiatorHelper enthält bereits für alle Domänenmetaklassen MetaclassDesc Objekte. Diese müssen lediglich noch "angereichert" werden.

```
accesspoint =
   GefInstantiatorHelper.getMetaClass(NWAccessPoint.class);
accesspoint.addStringProperty("Ip");
accesspoint.addStringProperty("Name").setVisibleInDiagram(true);
accesspoint.init();
metamodel.addMetaclass(accesspoint);
```

Nachdem alle Klassen welche später auf dem Editor vorkommen sollen auf diese Art und Weise beschrieben worden sind werden die Beziehungen zwischen den Klassen modelliert. Zunächst Eltern-Kind-Beziehungen:

```
new OneToManyContainmentDesc(accesspoint, hwport,
    "hardwarePort");
```

Hierbei sind die ersten beiden Parameter die jeweiligen Klassenbeschreibungen. Der letzte Parameter gibt als String den Namen der Methode an, welche auf der zuerst genannten Klassen eine Instanz der als zweites genannten Klasse annehmen kann. Diese Beschreibungen werden ebenfalls von generierten GefEditorHelper angelegt. Somit sind auch spätere Refactorings kein Problem.

Es folgen nun die Assoziationsbeziehungen. Im Unterschied zu einer Container-Beziehung kann einer Assoziationsbeziehung selbst wieder eine Domänenklasse zugrunde liegen, wie

es in unserem Beispiel bei Calbes der Fall ist: Die Assoziation zwischen zwei Hardwareports wird durch eine Instanz der Klasse NWCable ausgedrückt. Eine solche Assoziationsklasse kann natürlich selbst wieder Attribute besitzen, die editierbar sein sollen.

```
assoc_HwPort2HwPort = GefInstantiatorHelper.assoc_NWCableBase;
assoc_HwPort2HwPort.addStringProperty("type",
     new String[] { NWCable.LAN, NWCable.WLAN });
assoc_HwPort2HwPort.addIntProperty("bandwidth");
assoc_HwPort2HwPort.init();
```

Es ist auch möglich Assoziationen ohne eigene Domänenklasse zu modellieren. Einer Assoziationsbeschreibung können zusätzlich Layoutinformationen übergeben werden. Da es sich bei diesen Informationen nicht um einfache Strings handelt müssen diese zur Generierungszeit serialisiert werden. Zur Laufzeit des Editors werden diese Informationen dann wieder zur Verfügung. Mit diesem Trick steht dieselbe Instanz eines Objekts sowohl zur Generierungs- als auch zur Laufzeit zur Verfügung.

Als nächstes muss im Instantiator noch die Palette beschrieben werden. Dies geschieht in der Methode setupEditor:

```
protected void setupEditor(MetamodelDesc metamodel) {
     EditorDesc ed = new EditorDesc();
     ed.setName("HardwareEditor");
     ItemGroupDesc network = new ItemGroupDesc();
     ed.addItemGroup(network);
     network.setName("network");
     network.addItem(accesspoint, true);
     // …
     network.addConnectionItem(assoc_HwPort2HwPort);
      // …
}
```

Beim Hinzufügen einer Klassenbeschreibung zu einer ItemGroup kann zusätzlich über ein Boolean-flag festgelegt werden ob beim Wählen dieses Items später eine neue Instanz erzeugt werden soll oder lediglich eine im Modell bereits vorhandene ausgewählt werden kann. In diesem Fall wird zur Laufzeit des Editors nach allen Instanzen einer bestimmten Metaklasse gesucht und diese zur Auswahl gestellt.

Hiermit wäre die Beschreibung eines Editors abgeschlossen und die Generierung kann beginnen. Es werden bis auf die View-Klassen, in der GEF-Terminologie Figures genannt, alle notwendigen Klassen generiert. Für jede Klassenbeschreibung wird ein Wrapper um die Domainmodellklasse erstellt, welcher an diese weiterdelegiert soweit dies notwendig ist. Weiterhin wird für jede dieser Klassen ein Controller (in GEF Part genannt), sowie die im Controller verwendeten Strategieklassen (Policy) generiert. Dies geschieht wieder mit dem bereits oben Beschriebenen Mechanismus des Generators welcher mit Hilfe der Templates für jede Instanz des Editor-Metamodells die passenden Klassen erzeugt. Für jede Beschreibung die im Instantiator erstellt wurde wird ein Satz der für diese spezielle

Beschreibung notwendigen Templates vom Generator ausgeführt und so die dafür notwendigen Klassen erzeugt.

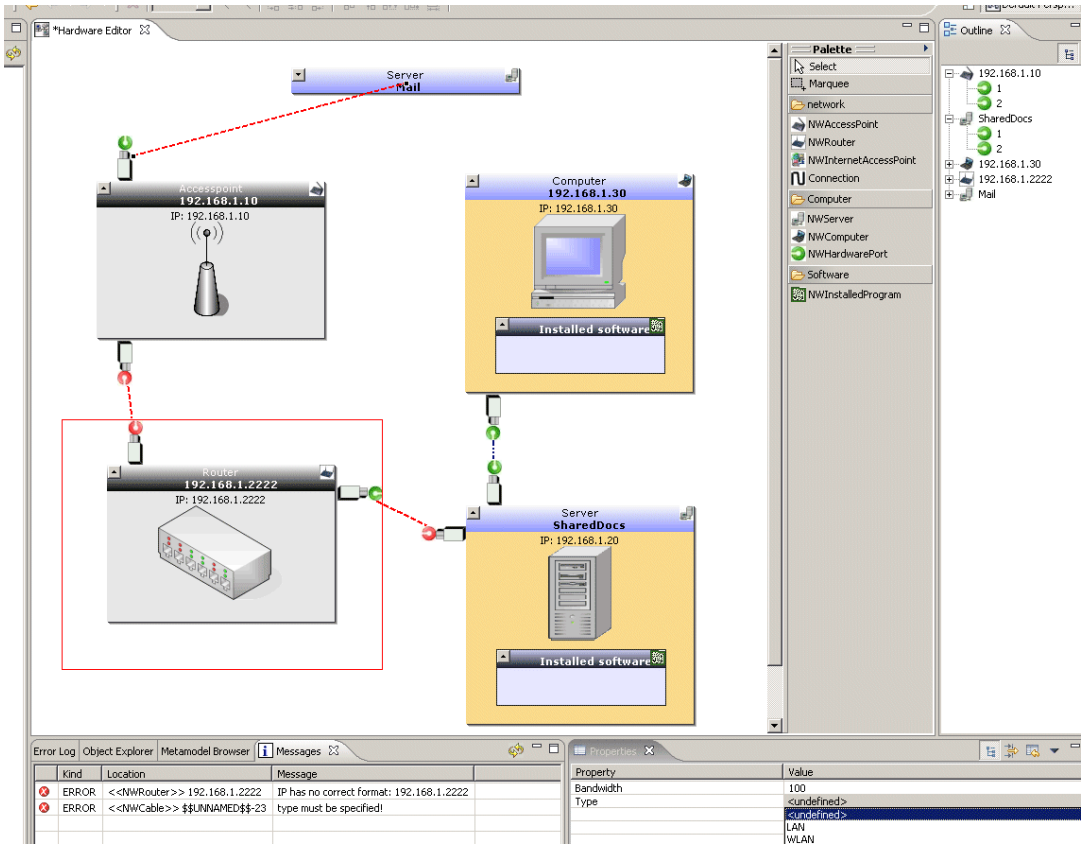

Abbildung 5: Screenshot des Beispieleditors

Um die in GEF oft ebenfalls sehr komplex zu erstellenden Views zu erzeugen entstand parallel ein Framework um diese Arbeit zu erleichtern und auch komplexe graphische Repräsentationen (incl. Collapsable Compartments, etc.) mit möglichst wenig Aufwand erstellen zu können.

IDE Integration

Änderungen am Modell die der Anwender durch einen generierten Editor herbeiführt werden in Echtzeit auf die im Domainmodell hinterlegten Constraints geprüft und eventuell auftretende Verletzungen können durch eine beliebige Veränderung der Figure dem Benutzer visuell deutlich gemacht werden. Zusätzlich werden diese Fehler im Modell in einem extra Eclipse-View dargestellt.

Jedes erstellte Diagramm kann zusammen mit den Layoutinformationen unabhängig von den Domainklassen gespeichert werden und „verschmutzt“ so nicht die Modellklassen.

Die Speicherung des Domänenmodells an sich ist mit Mitteln von openArchitectureWare standardmäßig möglich.

Fazit

Solche speziell auf eine Domäne ausgerichtete Editoren zusammen mit einer darauf abgestimmten Umgebung führen zu einer ungeahnten Produktivität und zeigen dadurch die stärken modellgetriebener Entwicklung.

Das hier beschriebene Beispiel sowie die Implementierung des hier beschriebenen Codegenerators für Editoren ist Open Source und ist zusammen mit einem Framework für domänenspezifische Entwicklungsumgebungen unter www.openarchitectureware.org verfügbar. Der beschreibe Ansatz ist natürlich nicht GEF-spezifisch, sondern kann auch für jedes andere Grafikframework so implementiert werden. Darüber hinaus gibt in unserer Community bereits die erste viel versprechende Ansätze für weitere DSL-spezifische Anpassungen. So sind gerade Anpassungen von Microsoft Visio, diverse UML-Tools sowie Featuremodellierungswerkzeuge in Arbeit oder bereits verfügbar.

Referenzen

[SV05] Stahl, Völter, Modellgetriebene Softwareentwickung, dPunkt 2005

[GEF] www.eclipse.org/gef

[oAW] www.openarchitectureware.org