Zimmermann, Frank

**Working Paper**

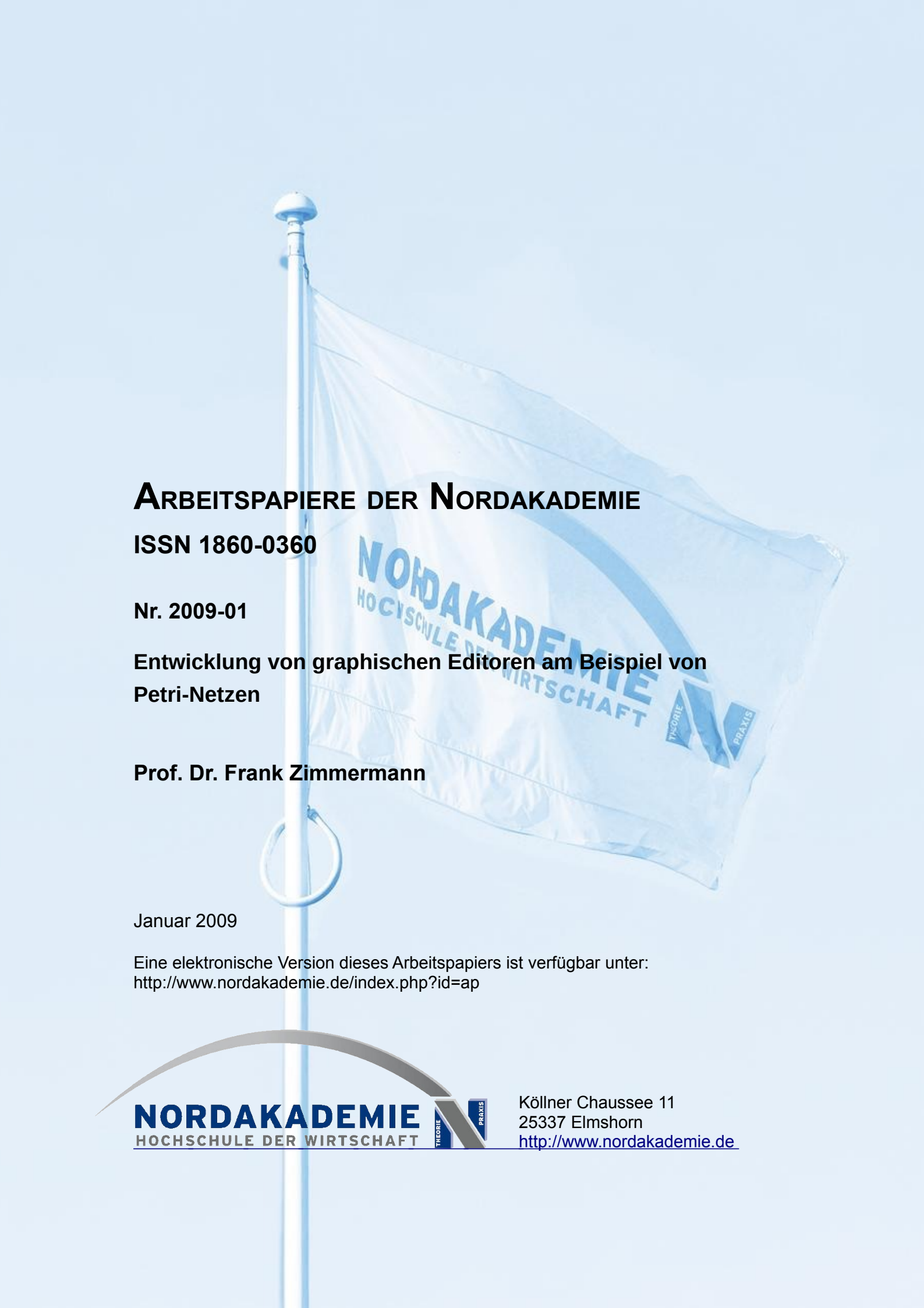
# Entwicklung von graphischen Editoren am Beispiel von Petri-Netzen

Arbeitspapiere der Nordakademie, No. 2009-01

**Provided in Cooperation with:**
Nordakademie - Hochschule der Wirtschaft, Elmshorn

*Suggested Citation:* Zimmermann, Frank (2009) : Entwicklung von graphischen Editoren am Beispiel von Petri-Netzen, Arbeitspapiere der Nordakademie, No. 2009-01, Nordakademie - Hochschule der Wirtschaft, Elmshorn

This Version is available at:
https://hdl.handle.net/10419/38594




# Arbeitspapiere der Nordakademie

**ISSN 1860-0360**

**Nr. 2009-01**

## Entwicklung von graphischen Editoren am Beispiel von Petri-Netzen

**Prof. Dr. Frank Zimmermann**

Januar 2009

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter:
http://www.nordakademie.de/index.php?id=ap

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

# Entwicklung von graphischen Editoren am Beispiel von Petri Netzen

Nordakademie
Frank Zimmermann
Januar 2009

## Zusammenfassung

In diesem Tutorial soll die Verwendung von EMF und GMF zur Erstellung eines einfachen grafischen Editors dargestellt werden. Dieses Beispiel ist eine dankbare Anwendung von EMF, weil die Modellierung von Petri Netzen mit der UML nicht möglich ist. Für dieses Beispiel gibt es Tutorials , die sich aber meistens nur auf die statischen Komponenten von GMF beziehen [PeRi08], [Kli07]. In diesem Tutorial werden die Petri Netze auch animiert, so dass auch GMF Runtime Komponenten angesprochen werden müssen. Diese werden im GMF Standard Tutorial [GMFTutorial] nicht verwendet.

Die Dokumentation von GMF befindet sich verstreut an verschiedenen Stellen, so dass es dem Anfänger sehr erschwert wird, sich in das Thema einzuarbeiten. Hierzu gehören neben vielen Sites, die über Google angeboten werden, die Eclipse GMF Dokumentation im Eclipse Wiki (http://wiki.eclipse.org/index.php/GMF_Documentation), die gmfnewsgroup (eclipse.modeling.gmf) und die GMF Q&A (http://wiki.eclipse.org/GMF_Newsgroup_Q_and_A ). Problematisch ist für den Anfänger auch die Abgrenzung zu GEF, auf dem GMF aufbaut. Abhilfe könnte das Buch [GRON09] bieten, das jedoch zur Zeit nur als Preview verfügbar ist und dessen Erscheinungsdatum seit einem Jahr immer wieder verschoben wird.

Dabei sollen die wesentlichen Schritte der Entwicklung von Editoren deutlich gemacht werden:

1. Erstellung eines Metamodells
2. Erzeugung eines EMF Projekts mit EMF Editoren
3. Erzeugung eines GMF Editors
4. Anpassung des GMF Editors (GMF Runtime)

## Installationsvoraussetzung

Die oAW Distribution (Ganymede oAW 4.3) enthält die wesentlichen Komponenten, die für modellgetriebene Entwicklung erforderlich sind.
http://oaw.itemis.de/openarchitectureware/language=de/660/downloads

## Petri Netze (Kurze Einleitung aus Wikipedia)

Ein Petri-Netz ist ein mathematisches Modell von nebenläufigen Systemen. Es ist eine formale Methode der Modellierung von Systemen.

Ein Petri-Netz ist ein bipartiter und gerichteter Graph. Er besteht aus **Stellen** (Places) und Übergängen bzw. **Transitionen** (Transitions). Stellen und Transitionen sind durch **gerichtete Kanten** verbunden. Es gibt keine direkten Verbindungen zwischen zwei Stellen oder zwei Transitionen. Stellen werden als Kreise, Transitionen als Rechtecke dargestellt. Jede Stelle hat eine Kapazität und kann entsprechend viele Token, Marken bzw. Zeichen enthalten. Ist keine Kapazität angegeben, steht das für unbegrenzte Kapazität oder für eins. Jeder Kante ist ein Gewicht zugeordnet, das die Kosten dieser Kante festlegt. Ist einer Kante kein Gewicht zugeordnet, wird der Wert eins verwendet.

Sind alle Kapazitäten der Stellen und Gewichte der Kanten eines Petri-Netzes 1, so wird es auch als Bedingungs-, Prädikat- oder Ereignis-Netz bezeichnet. Die Belegung der Stellen heißt Markierung und ist der Zustand des Petri-Netzes.

Transitionen sind aktiviert bzw. schaltbereit, falls sich in allen Eingangsstellen mindestens so viele Marken befinden, wie die Transitionen Kosten verursachen und alle Ausgangsstellen noch genug Kapazität haben, um die neuen Marken aufnehmen zu können. Schaltbereite Transitionen können zu einem beliebigen Zeitpunkt schalten. Beim Schalten einer Transition werden aus deren Eingangsstellen entsprechend den Kantengewichten Marken entnommen und bei den Ausgangsstellen entsprechend den Kantengewichten Marken hinzugefügt. Marken werden in einem Petri-Netz nicht bewegt. Sie werden entfernt und erzeugt!

Die Marken eines Petri-Netzes sind in ihrer einfachsten Form voneinander nicht unterscheidbar. Für komplexere, aussagekräftigere Petri-Netze sind Markeneinfärbungen, Aktivierungszeiten und Hierarchien definiert worden.

In diesem Tutorial sollen Ereignis-Netze erstellt und simuliert werden. Es soll dabei nicht um die Auswahl einer „praxisnahen“ Variante von Petrinetzen gehen, sondern sich der Einfachheit halber auf Ereignis-Petrinetze beschränken.

## Erstellung des Metamodells

Zunächst muss das Metamodell für Petri Netze erstellt werden.

> ***Definition 1: Meta-Modell***
>
> *Modelle beschreiben den Ausschnitt der Realität, der für einen gegebenen Zweck relevant ist. Meta- Modelle beschreiben, wie Modelle aufgebaut sind. Sie enthalten die Arten von Modellelementen, die in Modellen verwendet werden, ihre Beziehungen untereinander und die Regeln, die die Zulässigkeit von Modellen beschreiben.*

Das Metamodell von Petri Netze ist relativ einfach zu erstellen. Es besteht aus Transitionen und Stellen, die über gerichtete Kanten miteinander verbunden sind. Während die Transitionen und Stellen eindeutig als Klassenkandidaten zu identifizieren sind, gibt es bei den Kanten zwei Realisierungsmöglichkeiten.

1. Modellierung der Kanten als Klasse, hier ist zwischen eingehenden und ausgehenden Kanten zu unterscheiden, da ansonsten die Eigenschaft des Graphen, bipartit zu sein, über zusätzliche Constrains sichergestellt werden muss.
2. Modellierung der Kanten als Assoziationen zwischen Transitionen und Stellen und zwischen Stellen und Transitionen.

Beide Varianten sind umsetzbar, da die Kanten jedoch in der einfachen Aufgabenstellung weder Attribute noch Assoziationen noch Verhalten haben, wird die Variante 2 realisiert.

EMF [Bud09] und vor allen Dingen GMF erfordert aufgrund seines XML basierten Aufbaus, dass alle Modellelemente direkt oder indirekt in einer Containment Beziehung zu einem „root“ Element stehen. Containment bezieht sich in XML auf die Schachtelung von Tags. Das root Tag enthält die Tags Modells. Deshalb wird als Root Element die Klasse PetriNetz gewählt. Damit ergeben sich folgende Modellelemente

| Element | Typ | | |
|---|---|---|---|
| PetriNetz | Klasse | | |
| Stelle | Klasse | | |
| Transition | Klasse | | |
| name | Attribut | | EString |
| bestehtAusStellen | Containment | PetriNetz>Stelle | 0..* |
| bestehtAusTransitionen | Containment | PetriNetz>Transition | 0..* |
| aktiviert | Assoziation | Stelle>Transition<br>opposite: wirdAktiviertVon | 0..* |
| wirdAktiviertVon | Assoziation | Transition>Stelle<br>opposite: aktiviert | 0..* |
| setztMarke | Assoziation | Transition>Stelle | 0..* |
| istAktivierbar | Operation | Transition | boolean |
| aktivieren | Operation | Transition | void |
| aktivierbareTransitionen | Operation | PetriNetz | EList<Transition> |
| aktiviereEineTransition | Operation | PetriNetz | void |

***Definition 2: EMF/ECore***

*Meta Modelle in EMF werden im selbstbeschreibenden ecore Format abgelegt. Für die Verarbeitung in GMF ist es erforderlich, eine Containment-Hierarchie anzugeben, die die Modellobjekte in der Baumstruktur anordnet.*

Eine denkbare Alternative wäre eine abstrakte Klasse Element einzuführen, die als Oberklasse von Stelle und Transition die Containment Beziehung aufnehmen könnte.

Bei der Realisierung der Methoden „istAktivierbar“ und „aktiviere“ stellt sich heraus, dass die Assoziation „aktiviert“ in der Navigationsrichtung Transition>Stelle benötigt wird. Deshalb wird eine zweite Assoziation „wirdAktiviertVon“ definiert, die als opposite von „aktiviert“ definiert werden muss. Werden zwei Assoziationen, die in genau die entgegengesetzte Richtung laufen, als opposite definiert, so betrachtet EMF diese Assoziation wie eine einzige bidirektionale Assoziation.

Zur Umsetzung gilt es einiges zu beachten. Zunächst muss ein neues leeres EMF Projekt angelegt werden. In diesem befindet sich ein Ordner „model“. Dort legt man ein neues Ecore Diagram an. Achtung: Es gibt zwei davon: eins aus den Ecore Tools und eins aus dem Topcased Projekt. Das in Abbildung 1 abgebildete Modell wurde mit dem Ecore Tools Werkzeug erstellt.

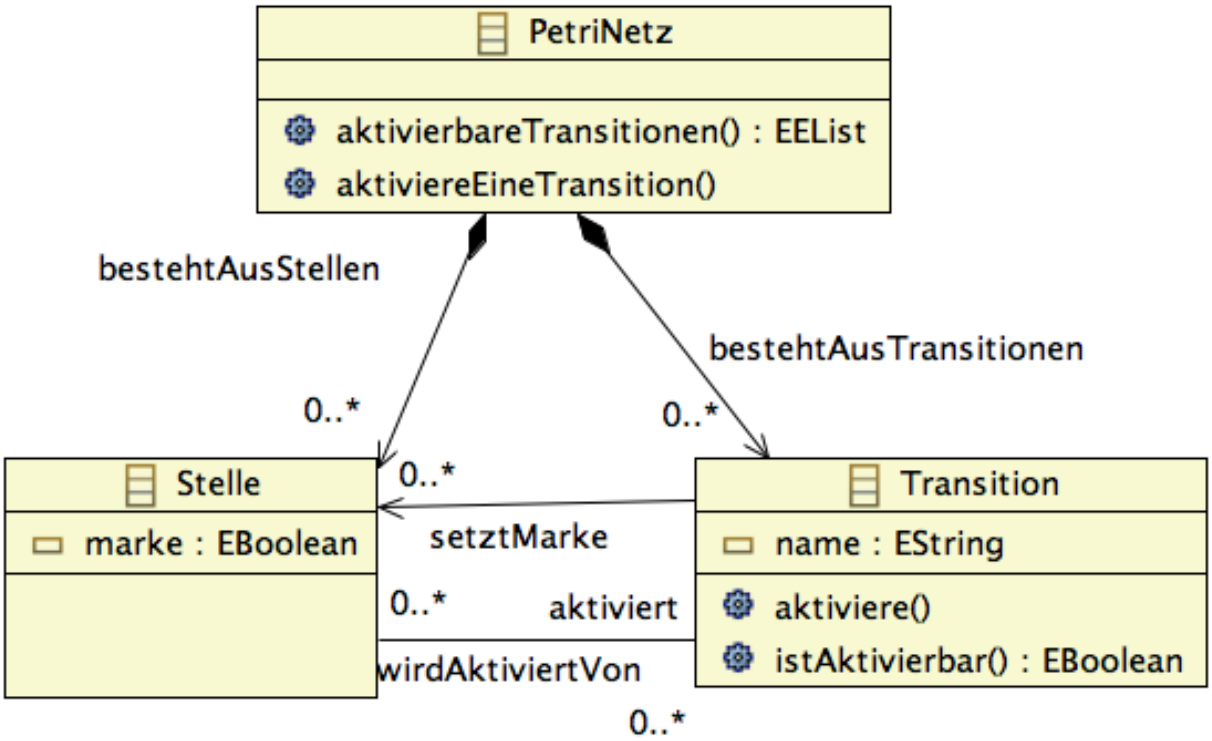



*Abbildung 1: Metamodell Ereignis-Bedingungs Petri Netz*

Hinweise zum Toolhandling:

1. Geben Sie im Wizard zum Anlegen eines Ecore Diagrams im ersten Dialogschritt als Dateinamen „Petri.ecorediag“ und im zweiten „Petri.ecore“ an.
2. Um die Multiplizität * einzugeben, muss man als upper bound eine -1 eingeben. Das wird dann als * angezeigt.
3. Containments und Assoziationen werden als EReference erfasst.
4. Um die bidirektionale aktiviert-wirdAktiviertVon Assoziation anzulegen, legt man zuerst zwei unidirektionale in entgegengesetzter Richtung an und definiert eine dann als EOpposite der anderen.
5. Einige der Eigenschaften (Containment, EOpposite) lassen sich nur über den Properties View eingeben. Dieser kann über das Kontextmenü (rechte Maustaste) „Show Properties View“ erreicht werden.

6. Beim Erfassen der Methode „aktivierbareTransitionen“ muss als return Typ ein generischer Typ angegeben werden. Dazu muss zunächst die Datei gesichert und durch Doppelklick auf die Petri.ecore Datei der Baumeditor geöffnet werden. Dann kann über den Menüpunkt Sample Ecore Editor der Menüleiste die Option „Show Generics“ eingeschaltet werden. Dann kann der generische Typ als Kindobjekt hinzugefügt werden.

Nach Bearbeitung liegen in dem Model Verzeichnis zwei Dateien. Die ecorediag Datei enthält die Informationen, die für die graphische Darstellung des eigentlichen ecore Metamodells erforderlich sind.

## Generierung der Java Komponenten und des Editors

Auf Grundlage des Meta Modells kann EMF eine Java Implementation generieren. Um den Generierungsprozess wiederholbar zu machen, werden die erforderlichen Einstellungen in einem weiterem Modell abgelegt, dem Petri.genmodel. Dieses legt man am einfachsten über den EMF Model Wizard an. In diesem Wizard muss man das eben erstellte Ecore Model importieren. Die meisten vorgeschlagenen Einstellungen für das Petri.genmodel werden automatisch generiert und sind brauchbar und deshalb nicht weiter von Interesse. Die einzige Änderung, die gemacht werden muss, ist die Angabe eines Basepackages de.nordakademie für das petri package. Siehe dazu Abbildung 2.

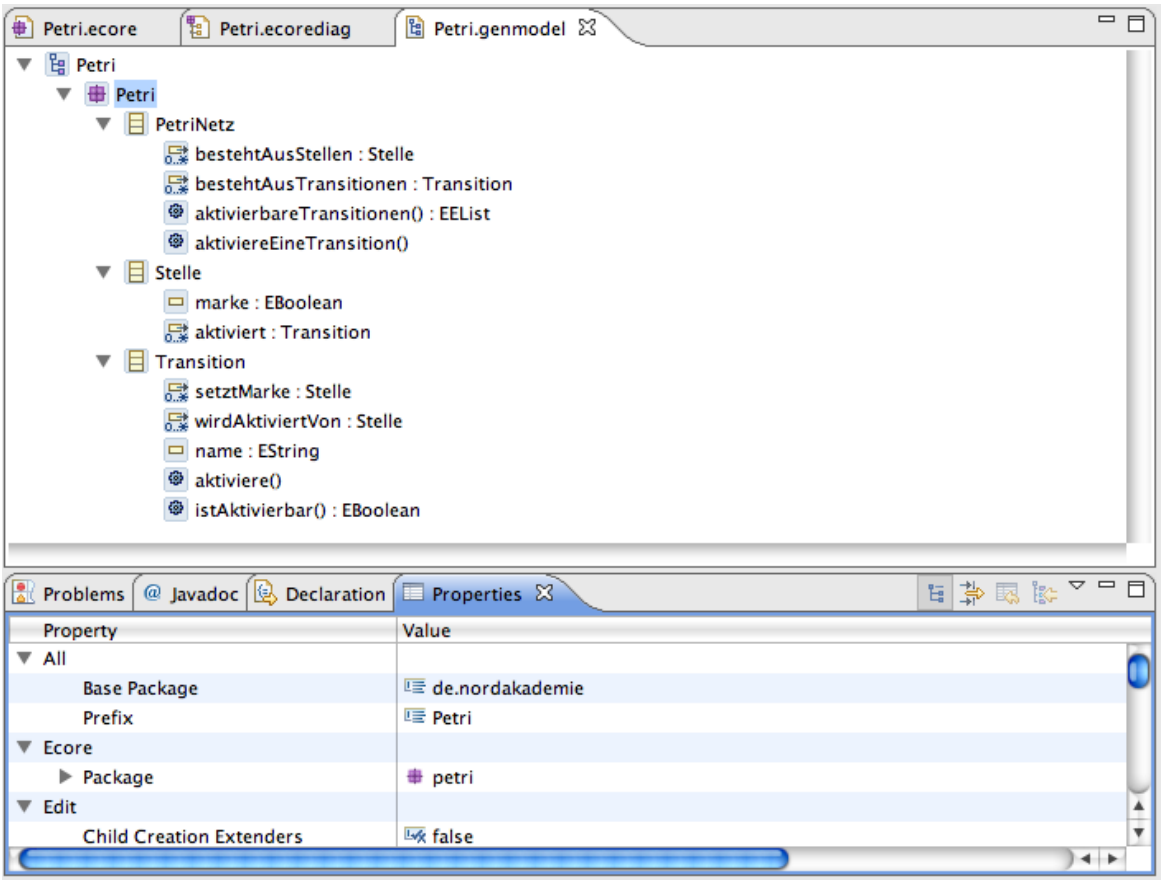



*Abbildung 2: Petri.genmodel*

Wenn das genmodel geöffnet ist, befindet sich in der Menüleiste ein Menüpunkt Generator, über den „alle“ Projekte und Klassen generiert werden können.

## *Anpassung der generierten Methoden*

Aus dem Meta Model wurde eine Implementation in Java generiert. EMF generiert Interfaces für alle Meta-Klassen und Implementationen für die Interfaces. Um Instanzen für die Implementationen zu erzeugen, steht eine Klasse PetriFactoryImpl zur Verfügung. Hier gibt es für alle Metamodell Klassen eine create-Methode. In den generierten Klassen fehlen erwartungsgemäß die Rümpfe der Operationen. In den generierten Impl-Klassen sind nur Stümpfe vorhanden. Diese sind anzupassen. Damit bei einer erneuten Generierung der Klassen nicht die manuell geschriebenen Teile übergeneriert werden, muss in dem der Methode zugeordneten Kommentar @generated NOT eingetragen werden. JMerge wird dann bei einer Neugenerierung den manuell geschriebenen Teil mit den neu generierten Teilen abgleichen.

Im dem src Ordner befindet sich im Package de.nordakademie.petri.impl die Klasse TransitionImpl. Dort ist die Methode „istAktivierbar“ zu implementieren. Dies wird in Snippet 1 dargestellt.

```
/**
 * @generated NOT
 */
public boolean istAktivierbar() {
    Iterator<Stelle> iter = getWirdAktiviertVon().iterator();
    while ( iter.hasNext() ){
      if (!iter.next().isMarke()) return false;
    }
    iter = setztMarke.iterator();
    while ( iter.hasNext() ){
      if (iter.next().isMarke()) return false;
    }
    return true;
}
```

*Snippet 1: Transition istAktivierbar*

Ebenso muss das Aktivieren einer Transition programmiert werden. Siehe dazu Snippet 2.

```
public void aktiviere() {
      if (!istAktivierbar()) return;
      Iterator<Stelle> iter = wirdAktiviertVon.iterator();
      while ( iter.hasNext() ){
            iter.next().setMarke(false);
      }
      iter = setztMarke.iterator();
      while ( iter.hasNext() ){
            iter.next().setMarke(true);
      }
}
```

*Snippet 2: Transition aktiviere*

Die Simulation eines Petri Netzes ist Aufgabe der Klasse PetriNetz. Dort werden die Methoden „aktivierbareTransitionen“ und „aktiviereEineTransition benötigt“. Sie greifen auf die soeben erstellten Methoden zurück. „aktiviereEineTransition“ wählt dabei zufällig eine aktivierbare Transition aus, um das Verhalten von nebenläufigen System möglichst praxisnah zu simulieren. Die Realisierung wird in Snippet 3 gezeigt.

```
      public EList<Transition> aktivierbareTransitionen() {
          EList<Transition> aktivierbare = new BasicEList<Transition>();
          Iterator<Transition> iter = bestehtAusTransitionen.iterator();
          while ( iter.hasNext() ){
            Transition n = iter.next();
            if(n.istAktivierbar()) aktivierbare.add((Transition)n);
          }
          return aktivierbare;
      }
      public void aktiviereEineTransition() {
            EList<Transition> list =aktivierbareTransitionen();
            Transition tr = list.get((int)(Math.random()*list.size()));
            tr.aktiviere();
      }
```

*Snippet 3: PetriNetz Methoden*

Wenn die generierten Programme angepasst werden, dürfen selbstverständlich Tests nicht fehlen. Die Klasse PetriNetzFactory stellt Methoden zum Erzeugen von Instanzen her. Snippet 4 zeigt den Einsatz dieser Factory Klasse zum Erzeugen einer Testfixture und zum Überprüfen der Aufrufergebnisse.

```
protected void setUp() throws Exception {
	Stelle stelle1= PetriFactory.eINSTANCE.createStelle();
	Stelle stelle2= PetriFactory.eINSTANCE.createStelle();
	Transition tr1 = PetriFactory.eINSTANCE.createTransition();
	tr1.getWirdAktiviertVon().add(stelle1);
	tr1.getSetztMarke().add(stelle2);
	setFixture(tr1);
}

public void testAktiviere() {
	Transition fixture = getFixture();
	fixture.aktiviere();
	assertFalse(fixture.getSetztMarke().get(0).isMarke());
	fixture.getWirdAktiviertVon().get(0).setMarke(true);
	fixture.aktiviere();
	assertTrue(fixture.getSetztMarke().get(0).isMarke());
	assertFalse(fixture.getWirdAktiviertVon().get(0).isMarke());
}

public void testIstAktivierbar() {
	Transition fixture = getFixture();
	assertFalse(fixture.istAktivierbar());
	fixture.getWirdAktiviertVon().get(0).setMarke(true);
	assertTrue(fixture.istAktivierbar());
	fixture.getSetztMarke().get(0).setMarke(true);
	assertFalse(fixture.istAktivierbar());
}
```

*Snippet 4: Testmethoden*

Obwohl die Projekte PetriNetz.edit und PetriNetz.editor hier nicht weiter besprochen werden, werden sie für den weiteren Verlauf benötigt und sollten nicht gelöscht werden.

## Erzeugung eines graphischen Werkzeugs für Petri Netze

Graphische Editoren müssen in der Lage sein, Modelle zu einem vorgegebenen Meta-Modell in Form von graphischen Objekten, GMF nennt sie „Nodes“, „Connections“, „Label“ und „Compartments“, darzustellen. Dabei müssen sie neben den Informationen des Modells auch Informationen der graphischen Darstellung, wie zum Beispiel die Positionen und die Farben der Figuren speichern. Aus Sicht eines Frameworks gibt es also zwei Modelle: das fachliche „Domain Model“, das eine Instanz des Meta Modells darstellt, und das „Graphical Model“.

***Definition 3: Domain Model***

*Domain Model ist ein anderer Name für Meta-Modell, der den fachlichen Charakter betont. Instanzen des Domain Models enthalten die fachlichen Informationen. Die Objekte der Instanzen eines Domain Models sind in einer Baumstruktur unterhalb eines Root Elements angeordnet.*

***Definition 4: Graphical Definiton Model***

*Das Graphical Model enthält die Knoten, Kanten, Textfelder und Abteilungen, die zur Darstellung von Domain Modell Instanzen erforderlich sind. Instanzen enthalten Informationen über die Position und das Aussehen graphischer Objekte. Die Objekte der Instanzen des Graphical Definition Models sind in einer Baumstruktur unterhalb eines Root Elements angeordnet.*

Um eine graphische Oberfläche mit GMF zu erzeugen, benötigt man ein neues GMF Projekt, das mit einem speziell auf GMF Projekte ausgelegten Wizard erstellt werden kann. Es ist guter Stil in der Entwicklung von Eclipse Plugins, kleine übersichtliche Projekte mit abgegrenzter Funktionalität zu haben. Dem Anfänger erscheint dieses Vorgehen unübersichtlich, aber Modularisierung von Software ist eine Grundvoraussetzung für Wartbarkeit.

Die Arbeit mit GMF fällt dem Einsteiger schwer, weil das Ergebnis nicht sofort angezeigt werden kann. Eine graphische Darstellungsmöglichkeit wird erst in GMF Version 2.2 angeboten (verfügbar mit Eclipse 3.5 Galileo voraussichtlich ab Juli 2009). Als Hilfe für Neueinsteiger existiert ein Dashboard, mit dem der Arbeitsablauf und der Arbeitsfortschritt transparent gemacht wird. Beim Anlegen eines GMF Projekts muss deshalb unbedingt darauf geachtet werden, dass die Checkbox „Show Dashboard View“ markiert ist.

Grundsätzlich müssen basierend auf dem Petri Metamodell drei verschiedene Spezifikationen erzeugt werden:

1. Eine Spezifikation des Graphischen Modells. Also im Kern der Figuren, die den Meta Modell Klassen entsprechen. Für die Knoten eines Graphen werden Rechtecke, Kreise oder beliebige andere zweidimensionale Dekorationen von Graphenknoten verwendet. Die Kanten werden durch Verbindungen zwischen den Knoten mit Quell- und Zieldekorationen (Pfeile, Kreise, etc) dargestellt. Sollen Attribute der Klassen im Graphischen Editor dargestellt werden, so müssen entsprechende Label definiert werden. Diese graphischen Objekte können kombiniert und zu komplexeren Objekten zusammengesetzt werden. Um das Aussehen solcher komplexer Objekte zu steuern, stehen die Layout Manager aus der Draw2D Bibliothek zur Verfügung.
2. Eine Spezifikation der Werkzeuge, mit denen neue graphische Objekte durch den Endanwender erzeugt werden können. Im Prinzip muss für jede Metamodellklasse und jeden Ver-

bindungstyp ein Erzeugungswerkzeug erstellt werden. Diese Werkzeuge können in einer Toolpalette zu Gruppen zusammengefasst werden.

3. Eine Mapping Spezifikation, die die Metamodellklassen mit den graphischen Objekten und Werkzeugen in Verbindung bringt. In einer hierarchischen Struktur beginnend mit dem Diagramm muss jeder MM Klasse ein Grafisches Objekt und ein Werkzeug zugeordnet werden.

Aus diesen drei Spezifikationen werden dann eine Generatordatei *.gmfgen (analog zum *.genmodel) und daraus wiederum der Javacode für den graphischen Editor generiert werden.

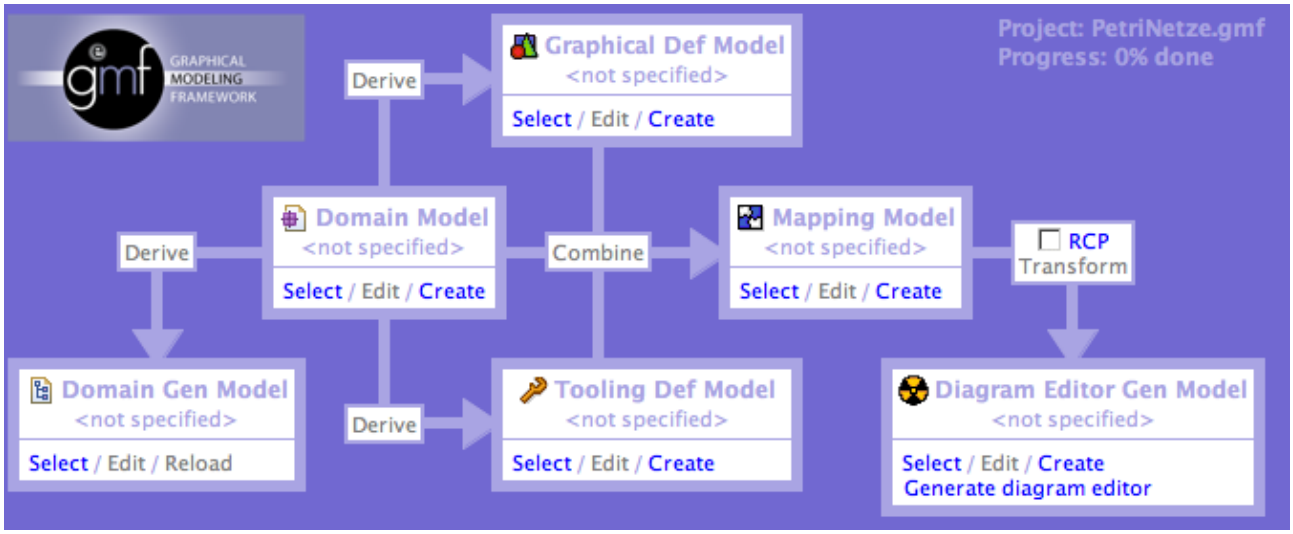



*Abbildung 3: GMF Workflow*

Grundsätzlich sind GMF Anwendungen nach dem bekannten Model View Controler Muster aufgebaut(vgl. [Fow03] Seite 330). Die Besonderheit der graphischen Objekte und des dahinter liegenden Meta Modells macht es jedoch erforderlich, dieses Muster zu erweitern. Einer der wesentlichen Aspekte ist die Tatsache, dass neben dem fachlichen Modell auch ein graphisches Modell existiert. Das fachliche Modell ist eine Instanz des Domain Models. Das graphische Modell enthält Informationen zu den Knoten und Kanten. GMF speichert diese Modelle in zwei verschiedenen Dateien. Neben diesen Models gibt es auch noch die View Objekte. Sie werden durch Figures (aus geometrischen Objekten zusammengesetzte graphische Objekte) dargestellt. Im Gegensatz zu Figures werden Nodes persitiert. Figures leben nur so lange, wie die Anwendung offen ist. Die Komponenten, die in den unterschiedlichen Modellen beschrieben werden, sind gemäß MVC den Applikationsbestandteilen zuzuordnen:

1. Domain Model: fachliches Modell ohne graphische Darstellung
2. Graphical Definition Model: Nodes und Figures aus dem graphical Model und dem View.
3. Tooling Definition Model: Teil des Views
4. Mapping Model: Modell des Controllers

Der Workflow zum Erstellen einer GMF Anwendung ist in dem Dashboard View in Abbildung 3 dargestellt. Man führt die folgenden Schritte durch:

1. Da das Domain und das genmodel schon existieren, muss dieses nur für die weitere Verarbeitung angegeben werden. Der Hyperlink bei Select im Domain Model (eigentlich nichts als mal ein anderes Wort für Meta Modell) auf dem Dashboard bringt den Anwender direkt zu dem entsprechenden Wizard.
2. Wenn das Domain Model ausgewählt wurde, wird der Link Derive für das Tooling Def Model und das Graphical Def Model aktiv. Beide Modelle können unabhängig voneinander abgeleitet werden. Für Graphical Def Model soll PetriNetz als root Element und die Einstellungen aus Abbildung 4 verwendet werden.

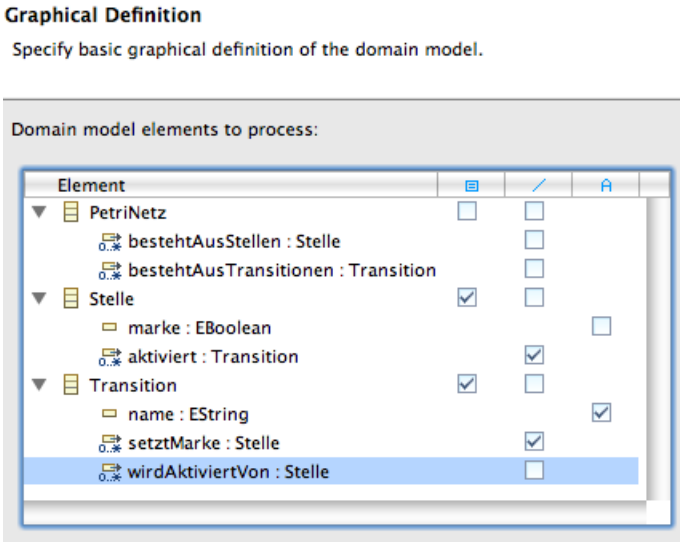



*Abbildung 4: Graphical Definition Model*

Dabei ist insbesondere darauf zu achten, dass das marke Attribut nicht in einem Label angezeigt werden soll, denn hierfür gibt es ja eine „Visualisierung“ in Form des Schwarzen Punktes. Eine Anpassung dieses Modells wird in der folgenden Beschreibung weiter unten

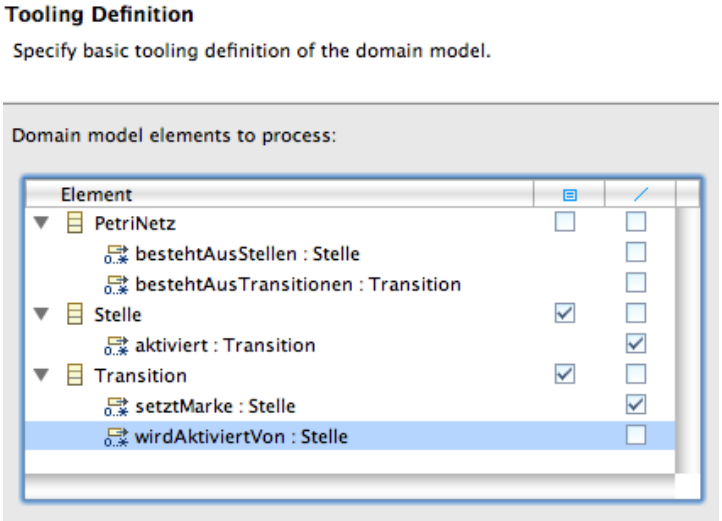



*Abbildung 5: Tooling Definition Model*

vorgenommen. Zweitens soll nur eine der beiden Asssoziationen aktiviert und wirdAktiviertVon ein Werkzeug angelegt werden, weil die andere Assoziation als opposite Assotiation ja automatisch mit gepflegt wird.

3. Ebenso ist bei der Ableitung der Tooling Definition darauf zu achten, dass wie in Abbildung 5 dargestellt „wirdAktiviertVon“ nicht verwendet wird.

## *Anpassung des Graphical Definition Model*

Der durch den Wizard generierte Graph ist nur in Ausnahmefällen zu gebrauchen. Meist müssen Anpassungen gemacht werden, die voraussetzen, dass man sich mit den Elementen des Graphischen Modells auseinandergesetzt hat. Viele davon sind selbsterklärend, jedoch sollte man ein paar grundsätzliche Dinge wissen.

Das Graphical Definition Model besteht

1 aus einer „Figure Gallery“, die die Zusammensetzung der graphischen Objekte und ihr Layout enthält. Dabei wird für jeden Knoten und jede Verbindung ein FigureDescriptor angelegt.

2 aus den Peer Objekten, die im Mapping benutzt werden können. Hinter jedem dieser Peer Objekte steckt ein Figure Descriptor. Das sind

2.1 Node Objekte, die auf Domain Objekte gebunden werden können

2.2 Connection Objekte, die den Verbindungen zwischen den Nodes entsprechen

2.3 Label Objekte, die zur Anzeige von Attributwerten verwendet werden

2.4 Compartment Objekte, die eine variable Anzahl von Kindobjekten aufnehmen können, in diesem Tutorial aber nicht verwendet werden.

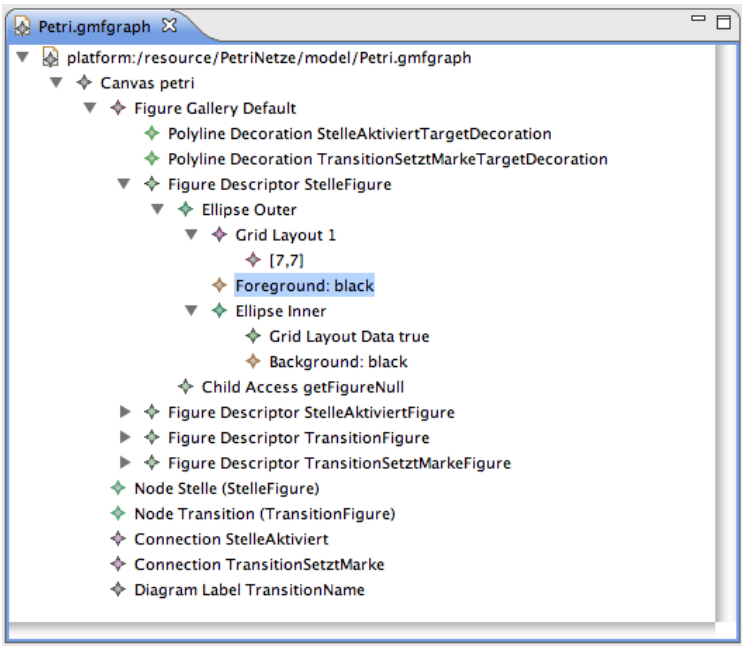



*Abbildung 6: Angepasstes Graphical Definition Model*

Der vom Wizard generierte Figure Descriptor zeigt für alle Knoten ein Rechteck. Diese Einstellung ist nicht wünschenswert, weil die Stellen durch Kreise mit einer Marke dargestellt werden müssen. Deshalb muss der Figure Descriptor wie in Abbildung 6 angepasst werden

Dabei sind einige der Attribute im Properties View (erreichbar über das Kontext Menü des Objekts) anzupassen.

| Objekt | Attribute | Werte |
|---|---|---|
| Ellipse | Name; Fill | Outer; false |
| MarginDimensions | Dx; Dy | 7 ; 7 |
| Ellipse | Name; Fill | Inner; true |
| GridLayoutData | HorzAlign; VertAlign;HorzExc;VertExc | Fill; Fill; true;true |
| ChildAccess | Figure | Elipse Inner |

Aus dem Figure Descriptor wird eine (innere) Klasse generiert werden, und damit die innere Ellipse nach aussen zugreifbar ist, muss ein ChildAccess angelegt werden. Dieser ChildAccess kann später benutzt werden, um einen schwarzen Kreis für die Marke sichtbar/unsichtbar zu machen.

## *Definition des Mapping Models*

Nun können das Graphical Defintion Model, das Tool Definition Model und das Domain Model miteinander kombiniert werden. Im Kern ist das eine 1:1:1 Beziehung, jedoch gibt es leider immer wieder Ausnahmen von dieser Regel. Abbildung 7 zeigt den wesentlichen Dialog des combine Wizards.

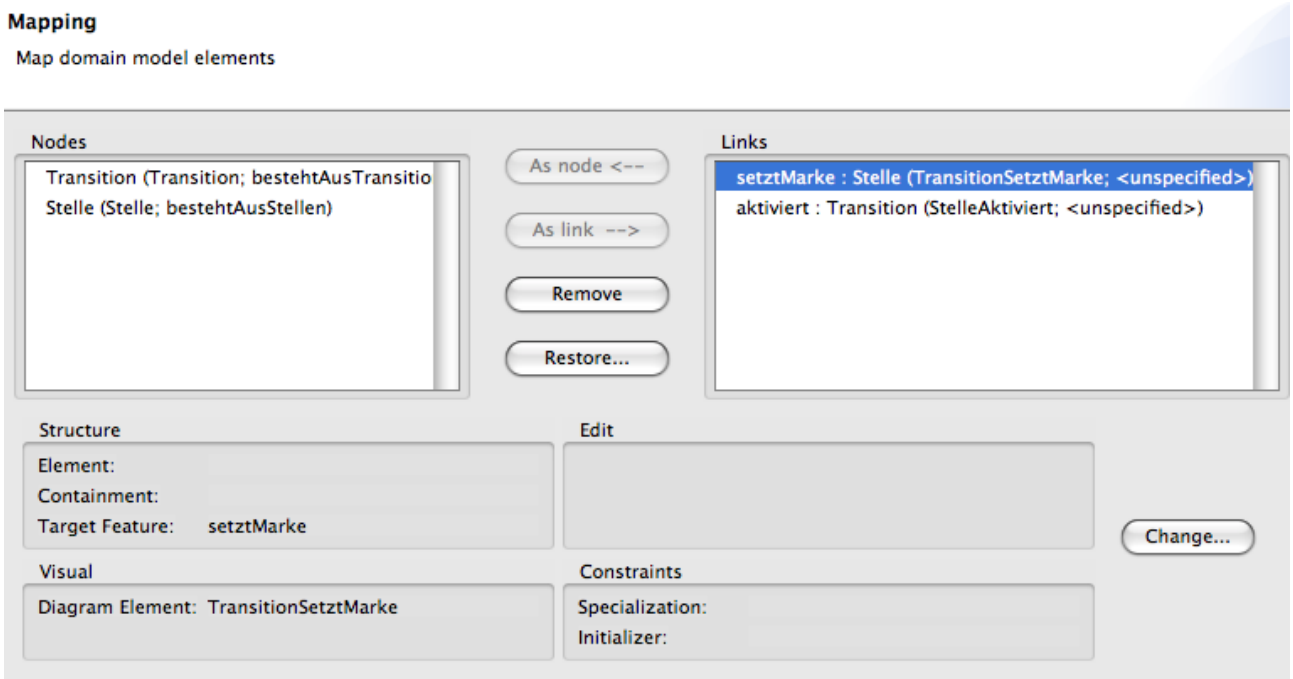



*Abbildung 7: Mapping Wizard*

Dieser Wizard erzeugt gerne einmal unerwünschte Zuordnungen, was leider nur durch besondere Sorgfalt in der Benutzung vermieden werden kann. Fehler hinterher zu finden und zu korrigieren ist unangenehm und zeitaufwendig, aber vermutlich unvermeidlich. Glücklicherweise ist die Unterstützung von Wiederholungen der Generierungen inzwischen sehr zuverlässig, vorausgesetzt, man macht alle Änderungen an dem Graphical Definition Model und dem Tooling Definition Model über den Editor des Mapping Definition Models. Nur dadurch kann sichergestellt werden, dass die Referenzen, die im Mapping Modell auf das Graphical Model und auf das Tooling Model bestehen auch nach Einfügungen/Löschungen noch gültig sind. Beachtet man dies, so kann man also bei der Erstellung der Modelle inkrementell vorgehen.

Zunächst ist sicherzustellen, dass das Link Mapping für den Domain Link wirdAktiviertVon gelöscht wird.

Insbesondere bei den Creation Tools setzt der Wizard gerne unerwartete Werte ein. Deshalb muss unbedingt kontrolliert werden, dass die Links „setztMarke“ und „aktiviert“ auch durch die richtigen Tools erzeugt werden. Das kann leicht über den Change Dialog erfolgen.

Leider wird das Label Mapping für den Namen einer Transition nicht vollständig durch den Generator erzeugt. Es fehlt die Angabe des Diagram Labels, und die muss hinterher manuell eingetragen werden. Abbildung 8 zeigt das Ergebnis.

Auf jeden Fall lohnt sich auch eine Überprüfung der Creation Tools, die in den Node Mappings angegeben werden. Sie müssen auch sinnvoll angepasst werden.

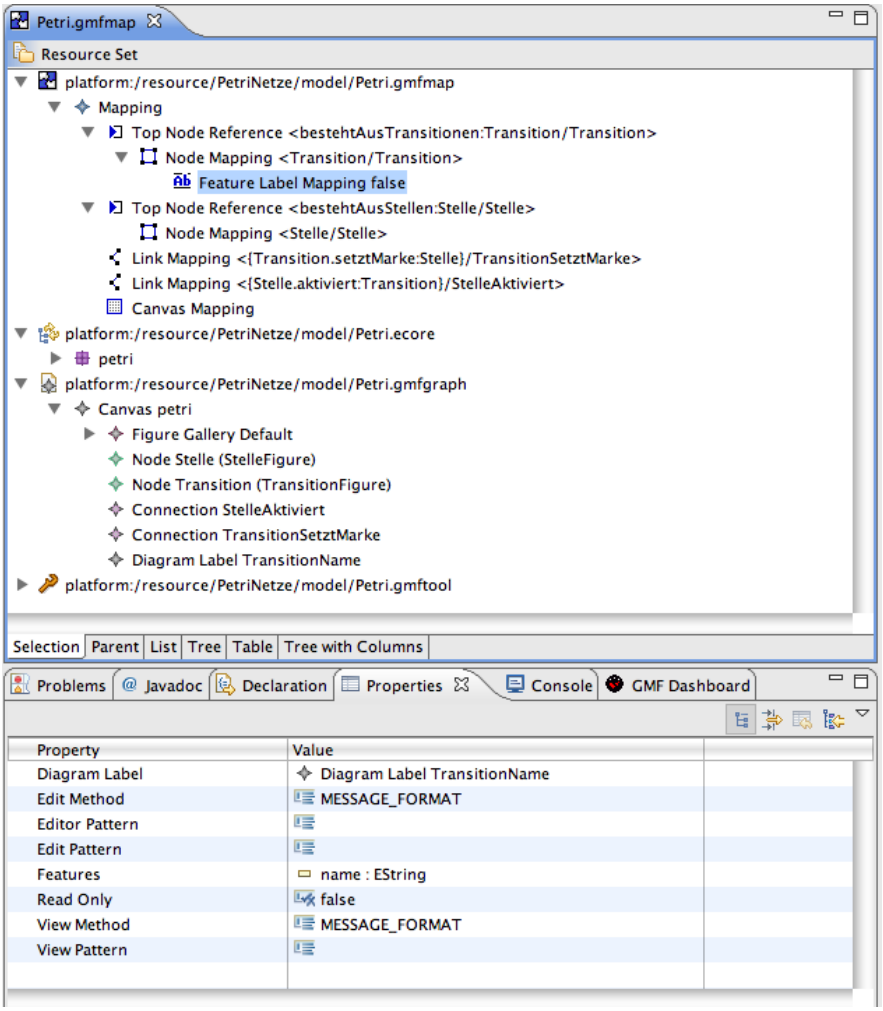



*Abbildung 8: Fertiges Mapping Definition Model*

Es ist sinnvoll, das Mapping zu validieren. Den erforderlichen Menüpunkt findet man im Kontextmenü.

## Generieren des Java Codes und Testen des Diagramms

Wenn die Validierung erfolgreich ist, kann man über das Kontextmenü der gmfmap datei das Generator Model gmfgen erzeugen. In dem einfachen Beispiel ist keine Anpassung in diesem Modell durchzuführen und deshalb kann sofort in dessen Kontextmenü der Diagram Code erzeugt werden. Wie gesagt, dieses Vorgehen kann gerne iterativ durchgeführt werden. Änderungen an den generierten Artefakten bleiben erhalten.

Zum Testen der Anwendung kann nun in den Startkonfigurationen eine Eclipse Application konfiguriert werden. Leider benötigt die verwendete Distribution mehr Speicherplatz als normal und müssen in den VM Arguments der Startkonfiguration die Parameter `-Xms64m -Xmx1024m` angegeben werden.

Nochmals die Anweisungen zur Korrektur von Fehlern:

1. Änderungen immer nur im gmfmap Editor vornehmen. Ansonsten geraten die drei Modelle aus der Synchronisation und müssen komplett neu erstellt werden.
2. Nach Änderung der gmfmap muss die gmfgen Generierung und die Codegenerierung durchgeführt werden.
3. Geänderte Methoden müssen mit @generated NOT annotiert werden.

# Setzen und Zurücksetzen der Marke

Um das Setzen und Zurücksetzen der Marke im graphischen Editor vornehmen zu können, müssen zwei Dinge getan werden:

1. Der Anwender muss durch Doppelklick auf eine Stelle eine Änderung des Attributs marke der Stelle auslösen können.
2. Wenn sich das marke Attribut ändert, muss der View angepasst werden.

## *Verändern des View bei einer Attributänderung eines Domain Objects*

Der zweite Punkt ist einfach zu realisieren, die Darstellung folgt im wesentlichen [GMFTips]: GMF Anwendungen sind nach dem Model View Controller Prinzip aufgebaut. Änderungen an dem Modell werden vom Controller erkannt und behandelt, indem die Views an den neuen Stand angepasst werden. Dabei ist natürlich sicherzustellen, dass die Modell-Objekte die Controller Objekte nicht kennen dürfen. Hier kommt das Observer-Observable Entwurfsmuster zum Einsatz. Der Controller registriert sich beim Modell, um über Zustandsänderungen informiert zu werden. Tritt eine solche Zustandsänderung des Modells auf, werden die Controller Objekte informiert. Diese holen sich den neuen Zustand beim Modell ab und leiten ihn an den View weiter.

In GMF kennt jeder Controller sein Modell und hat sich bei ihm als Observer registriert. Dabei heißen die Controllerklassen in GMF EditPart. Jeder Modellklasse entspricht dabei ein EditPart. Zu Stelle gehört StelleEditPart, zu Transition gehört TransitionEditPart etc.

Die von EMF generierten Klassen TransitionImpl, StelleImpl und PetriNetzImpl erzeugen nach dem Observer Observable Entwurfsmuster Änderungsereignisse (Notifications), die von den Controllerkomponenten abgefragt werden. Um beim Ändern des Zustandes einer Stelle im zugehörigen Controller reagieren zu können, muss die handleNotification Methode überschrieben werden.

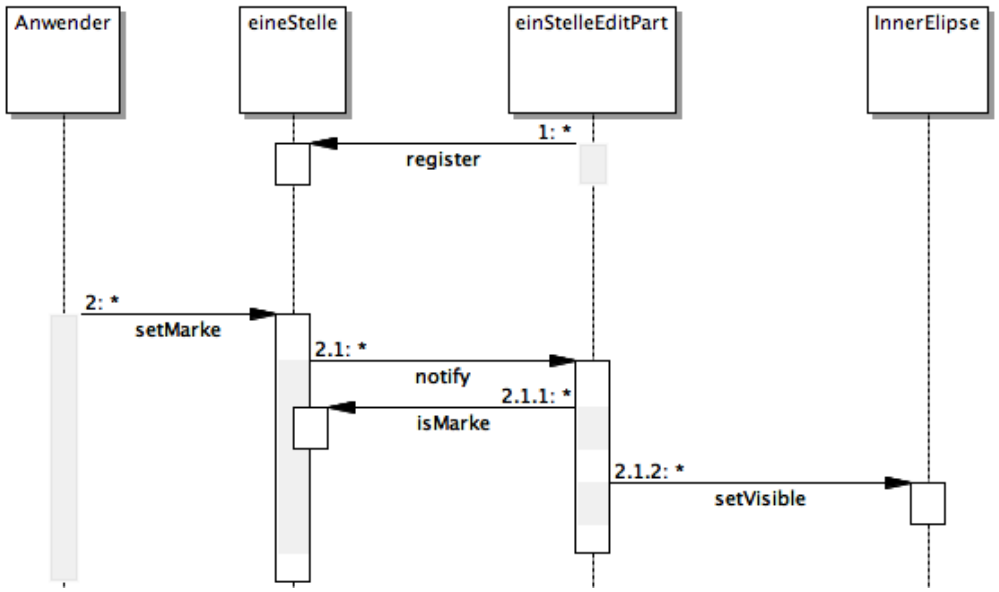



*Abbildung 9: vereinfachtes Interaktionsdiagramm MVC*

Der zugehörige Controller liegt im Package de.nordakademie.petri.diagram.edit.parts und heißt Stelle EditPart.

```
protected void handleNotificationEvent(Notification notification) {
	Object feature = notification.getFeature();
	if (PetriPackage.eINSTANCE.getStelle_Marke().equals(feature)) {
		setInnerFigureVisibility();
		refreshVisuals();
	}
	super.handleNotificationEvent(notification);
}

private void setInnerFigureVisibility() {
	IFigure fFigure = getPrimaryShape().getFigureInner();
	Node model = (Node)getModel();
	fFigure.setVisible(((Stelle)model.getElement()).isMarke());
}
```

*Snippet 5: Behandeln von Modelländerungen*

Der primaryShape ist die der Stelle zugeordnete Figur. Damit die Änderungen an der Figur auch tatsächlich angezeigt werden, wird nach der Änderung der Visibility mit der Methode refreshVisuals ein Neuzeichnen angestoßen. Diese Änderung kann sofort getestet werden. Man kann im Properties View die marke Eigenschaft von true auf false setzen, und damit sollte die Marke unsichtbar werden.
Es stellt sich heraus, dass bei einer neuen Stelle marke false ist, der schwarze Punkt aber dennoch angezeigt wird. Das liegt daran, dass beim Anlegen der Figur setInnerFigureVisibility nicht aufgerufen wird. Deshalb muss die generierte Methode getNodeShape modifiziert werden.

```
/**
 * @generated NOT
 */
protected IFigure createNodeShape() {
	StelleFigure figure = new StelleFigure();
	primaryShape = figure;
	setInnerFigureVisibility();
	return primaryShape;
}
```

*Snippet 6: Korrektes Erzeugen der Figur*

## Interaktion mit dem Anwender

Nun muss die Reaktion auf den Doppelklick programmiert werden. Auch hier kommt das MVC Prinzip zum Tragen. Das Sequenzdiagramm aus zeigt den Austausch von Nachrichten, der durch einen Doppelklick des Anwenders ausgelöst wird. Die prinzipelle Funktionsweise ist in [GEFGuide] näher erläutert.

Events, die durch den Anwender ausgelöst werden, werden von GMF (GEF) an die zugehörigen Controller weitergeleitet. Die delegieren ihrerseits an sog. EditPolicies. Diese beschreiben die Reaktion auf ein durch den Anwender ausgelöstes Event. Ein EditPart kann viele verschiedene Reaktionen auslösen, deshalb können sich viele EditPolicies beim EditPart registrieren. Eine Priorität regelt die Ausführungsreihenfolge. Darüber hinaus kann der Editpart auch unterschiedliche Anwenderak-

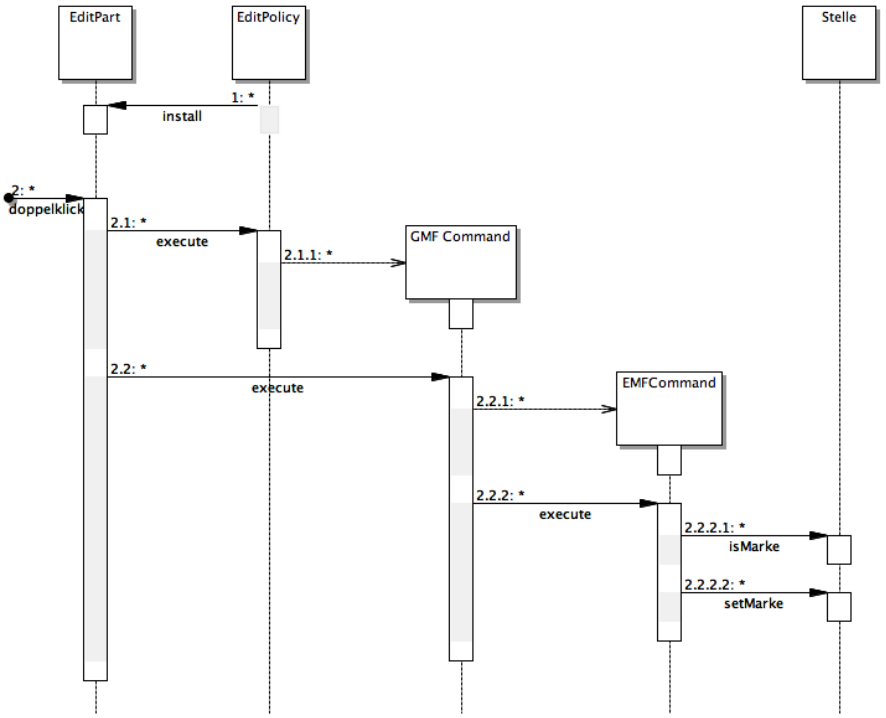



*Abbildung 10: Nachrichtenaustausch bei Doppelklick*

tionen verarbeiten. Zum Beispiel Rechtsklick oder Doppelklick oder einfacher Klick. Welche EditPolicy bei welcher Interaktion verwendet wird, hängt von der Rolle ab, für die sie installiert wurde. Die OPEN_ROLE ist die Rolle, die einem Doppelklick entspricht. Abbildung 10 zeigt den Nachrichtenaustausch, der durch einen Doppelklick ausgelöst wird. Die EditPolicy wird nach einem OpenCommand gefragt, das von der EditPolicy neu erzeugt wird. Dieses Command wird dann ausgeführt. Das GMF Command erzeugt ein EMF Command, das Transaktionssicherheit gewährleistet. In dem EMF Command wird das marke Attribut der Stelle modifiziert. Eine Darstellung der grundlegenden Zusammenhänge findet man in [GEFGuide].
Das Installieren der EditPolicy könnte als Modifikation im EditPart vorgenommen werden. Viel eleganter ist es jedoch, einen Extension Point von GMF zu verwenden. Dadurch braucht nichts am generierten Code verändert zu werden.
GMF stellt einen Extension Point zur Verfügung, bei dem EditPolcyProvider registriert werden können. Beim Erzeugen der EditParts werden diese Provider befragt, ob für den neu erzeugten EditPart eine EditPolicy erzeugt werden muss. Ist das der Fall, so wird die EditPolicy beim EditPart installiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
       <extension  point="org.eclipse.gmf.runtime.diagram.ui.editpolicyProviders">
        <editpolicyProvider class="petrinetze.extensions.providers.ToggleMarkeEditPolicyProvider">
             <Priority name="High"/>
        </editpolicyProvider>
    </extension>
</plugin>
```

*Snippet 7: ExtensionPoint editPolicyProvider*

Dazu ist es erforderlich, eine neue Extension anzulegen. Damit das generierte Projekt so wenig wie möglich angepackt werden muss, wird ein neues Plugin Projekt erstellt, das die Petri extensions aufnehmen kann.
Damit dieses neue Plugin Projekt die richtigen Plugin Dependencies, die ja dem Classpath entsprechen, hat, ist der einfachste Weg die Required-Bundels aus der MANIFEST.MF des diagram Projekts in das neue Projekt zu kopieren.
Danach wird in dem neuen Plugin Projekt eine plugin.xml angelegt. Siehe dazu Snippet 7.

```
package petrinetze.extensions.providers;

import ...;
public class ToggleMarkeEditPolicyProvider extends AbstractProvider
                                        implements IEditPolicyProvider {
     public void createEditPolicies(EditPart editPart) {
          if (editPart instanceof StelleEditPart) {
               editPart.installEditPolicy(EditPolicyRoles.OPEN_ROLE,
                         new ToggleMarkeEditPolicy());
          }
     }

     public boolean provides(IOperation operation) {
          if (operation instanceof CreateEditPoliciesOperation) {
               CreateEditPoliciesOperation op =
                              (CreateEditPoliciesOperation) operation;
               if (op.getEditPart() instanceof StelleEditPart) {
                    return true;
               }
          }
          return false;
     }
}
```

*Snippet 8: Edit Policy Provider*

Der ToggleMarkeEditPolicyProvider erzeugt eine ToggleMarkeEditPolicy für StelleEditParts. Alle anderen EditParts bekommen keine EditPolicy.

## Animation der Petri Netze

Für die Animation muss ein Menüpunkt „Animation“ ergänzt werden. Eclipse ermöglicht es, Menüpunkte an verschiedenen Stellen unterzubringen. Ein nahe liegender Ort ist das Kontextmenü des PetriNetz Objekts.

Damit die Verarbeitung, die hinter dem Menüpunkt steht, wiederverwendet werden kann, verwendet man in GMF Commands, die dann durch unterschiedliche Auslöser aufgerufen werden können. Dieses Vorgehen hat den Vorteil, dass die Commands auch auf einem Command Stack gehalten werden können, der für ein Undo genutzt werden kann [GHVJ96]. Das AnimateCommand ist Snippet 10 dargestellt.

Das Command muss den aktuellen EditPart kennen. Über den EditPart können ja sowohl der View als auch das Modell bearbeitet werden. Der EditPart wird dem Command über Dependency Injection mitgegeben. Im Fall des AnimateCommands ist der EditPart ein PetriNetzEditPart. Als Modell Objekt ist diesem EditPart ein PetriNetz Objekt zugeordnet. Dieses kann mit Hilfe der Methode „aktiviere eine Transition“ animiert werden.

```
package petrinetze.extensions.commands;
import ...;

public class AnimateCommand extends Command {

   private IGraphicalEditPart part;

   public AnimateCommand() {super("Run App");}
   public boolean canExecute() { return true;}
   public void redo() {redoModel();}
   public void execute() {  redo();}
   public void undo() {undoModel();}
   protected void undoModel() {}

   protected void redoModel() {
      if (part != null) {
         new Job("Animate Application") {
            public IStatus run(IProgressMonitor monitor) {
               try {
                  monitor.beginTask("Animating", 100);
                  Diagram node = (Diagram)(part.getModel());
                  final PetriNetz p = (PetriNetz)node.getElement();
                  while (!monitor.isCanceled()) {
                     part.getEditingDomain().getCommandStack().execute(
                     new AbstractCommand(){
                           public void execute() {
                              p.aktiviereEineTransition();
                           }
                           protected boolean prepare() {return true;}
                           public void redo() {}
                     });
                     Thread.sleep(1000);
                  }
               } catch (Exception e) {}
               return new Status(IStatus.OK,
                       "petrinetz",
                       "Animation ended", null);
            }
         }.schedule();
      }
   }

   public Command setCurrentPart(IGraphicalEditPart part) {
      this.part = part;
      return null;
   }
}
```

*Snippet 10: AnimateCommand*

Da die Animation nicht die gesamte Eclipse Anwendung blockieren soll, muss die Animation im Hintergrund laufen. In Eclipse steht für solche Hintergrundprozesse die Klasse Job (vgl. [Anj05]

```
package petrinetze.extensions.actions;

import ...;

public class PetriNetzAnimateAction implements IEditorActionDelegate {

   private IGraphicalEditPart editPart;

   public void setActiveEditor(IAction action, IEditorPart targetEditor){
   }

   public void setActivePart(IAction action, IWorkbenchPart targetPart){
   }

   public void run(IAction action) {
      if (this.editPart != null) {
         AnimateCommand cmd = new AnimateCommand();
         cmd.setCurrentPart(this.editPart);
         cmd.execute();
      }
   }

   public void selectionChanged(IAction action, ISelection selection) {

      if (selection instanceof IStructuredSelection) {
         EditPart first = (EditPart) ((IStructuredSelection) selection)
               .getFirstElement();
         if (first != null && (first instanceof PetriNetzEditPart)) {
             editPart=(IGraphicalEditPart)first;
             return;
         }
      }
      editPart=null;
   }
}
```

*Snippet 11: AnimateAction*

Seite 701 ff) zur Verfügung. Jobs verfügen über einen Progress Monitor, der den Anwender über den Fortschritt des Jobs informiert und die Möglichkeit zu Verfügung stellt, den Job abzubrechen.

```
package petrinetze.extensions.editPolicies;
import ...;
public class ToggleMarkeEditPolicy extends OpenEditPolicy {
protected Command getOpenCommand(Request request) {
  return new Command("ToggleMarkeCommand") {
      public void execute() {
        try {
            StelleEditPart stelleEditPart = (StelleEditPart) getHost();
            EditingDomain domain = stelleEditPart.getEditingDomain();
            Node node = (Node) stelleEditPart.getModel();
            Stelle stelle = (Stelle) node.getElement();
            domain.getCommandStack().execute(
                  SetCommand.create(domain, stelle,
                              PetriPackage.eINSTANCE.getStelle_Marke(),
                        !stelle.isMarke()));
      } catch (Exception c) {}
 };}}
```

*Snippet 12: ToggleMarkeEditPolicy*

Das Command ist aber nicht direkt mit dem Menüeintrag verbunden. In einem Kontext Menü ist es ja möglich, mehrere Objekte zu selektieren und diese gemeinschaftlich zu bearbeiten. Das Command bearbeitet aber immer nur ein Objekt. Deshalb gibt es eine weitere Klasse, die zwischen dem Command und der Selektion vermittelt. Diese Klasse wird Action genannt, in unserem Fall also die AnimateAction. Diese Action übernimmt die Verarbeitung der Selektion und erzeugt ein Command für jedes zu verarbeitende Objekt. Darüber hinaus kann über die Action der Menüpunkt disabled werden, wenn die Selektion nicht zu dem zu erzeugenden Kommando passt. Snippet 11 auf Seite 23 zeigt den Source Code der AnimateAction. In der Methode selectionChanged wird der erste Eintrag der Selektion herausgesucht. Ist es ein PetriNetzEditPart, wird dieser für den Aufruf von run gemerkt. In der Methode run wird der aktuell ausgewählte EditPart genommen und ein AnimateCommand erzeugt.

Schließlich ist die AnimateAction noch bekannt zu machen. Für Actions, die in einem Kontext Menü liegen, steht ein ExtensionPoint popupMenus zur Verfügung. Die nötige Auszeichnung steht in Snippet 13 auf Seite 24.

```
<extension point="org.eclipse.ui.popupMenus">

   <viewerContribution id="de.nordakademie.petrinetz.animation"
        targetID="org.eclipse.gmf.runtime.diagram.ui.DiagramEditorContextMenu">
      <action class="petrinetze.extensions.actions.PetriNetzAnimateAction"
           enablesFor="1"
           id="petrinetze.extensions.actions.PetriNetzAnimateAction"
           label="Animate"
           tooltip="Connect and wait for animation messages"
           menubarPath="additions">
         <enablement>
            <objectClass name="de.nordakademie.petri.diagram.edit.parts.PetriNetzEditPart"/>
         </enablement>
      </action>
   </viewerContribution>
</extension>
```

*Snippet 13: ExtensionPoint popUpMenus*

# Literaturverzeichnis

Anj05: Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellermann, Pat McCarthy; The Java Developers Guide to Eclipse;Addison Wesley;2005
Bud09: David Steinberg, Frank Budinsky, Marcelo Paternostro, und Ed Merks; EMF: Eclipse Modeling Framework (2nd Edition);Addison-Wesley Longman;2009
Fow03: Martin Fowler; Patterns of Enterprise Application Architecture;Addison Wesley;2003
GEFGuide: Eclipse GEF Project, GEF Programmers Giude, 1.2009, http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.gef.doc.isv/guide/guide.html
GHVJ96: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Entwurfsmuster;Addison Wesley;1996
GMFTips: GMF Project, GMF Tips, 01.2009, http://wiki.eclipse.org/GMF_Tips
GMFTutorial: Eclipse GMF Project, GMF Tutorial, 1.2009, http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.gef.doc.isv/guide/guide.html
GRON09: Richard Gronback; Eclipse Modeling Project;Addison Wesley;2009
Kli07: Helge Klimek; Building Graphical Editors using GMF Technology; Workshop on Model Driven Software Development Today; Software Engineering 2007 ; 3.2007
PeRi08: Luis Pedro; Matteo Risoldi;Metamodeling with Eclipse;Université Genève;Techn. Dokumentation;5.2008;URL:http://smv.unige.ch