Statisches Modell

Grundlage des Editors bildet das Ecore Modell aus dem EMF. Diese Basis ermöglicht uns die einfache Erstellung eines grafischen Editors, der auf einer einfachen Paketstruktur (Abbildung 1) aufsetzt. Die Struktur und Beziehungen zwischen den Paketen bleiben während der Entwicklung unverändert, da sie bereits durch GMF generiert wurden und so dessen Standard unterliegen. Wir nehmen uns aber vor, einzelne Klassen im Zuge der Implementierung auf unser Produkt anzupassen und gegebenenfalls Klassen zu ergänzen, um die vorgesehene Funktionalität zu Gewährleisten.

Die Programmlogik baut auf selbst erstellten ecore-, und GMF-Modellen auf, wobei speziell das ecore-Modell in Anlehnung an die Ecore.ecore erstellt wurde, um alle wesentlich Features abdecken zu können. Aus den einzelnen Elementen der Modelle und ihren Beziehungen untereinander sollen die resultierenden ecore, gmfgraph/tool/map-Dateien abgeleitet werden. Der Vorteil dieser Vorgehensweise liegt darin, dass mit den selbst erstellten Modellen, eine höhere Flexibilität bei der Implementierung gewährleistet werden kann, sowie die Funktionalität des Editors besser skalierbar ist. Damit wird es möglich sein, auch in einer fortgeschrittenen Implementierungsphase noch relativ problemlos Änderungen an den zu Grunde liegenden Modellen durchführen zu können.

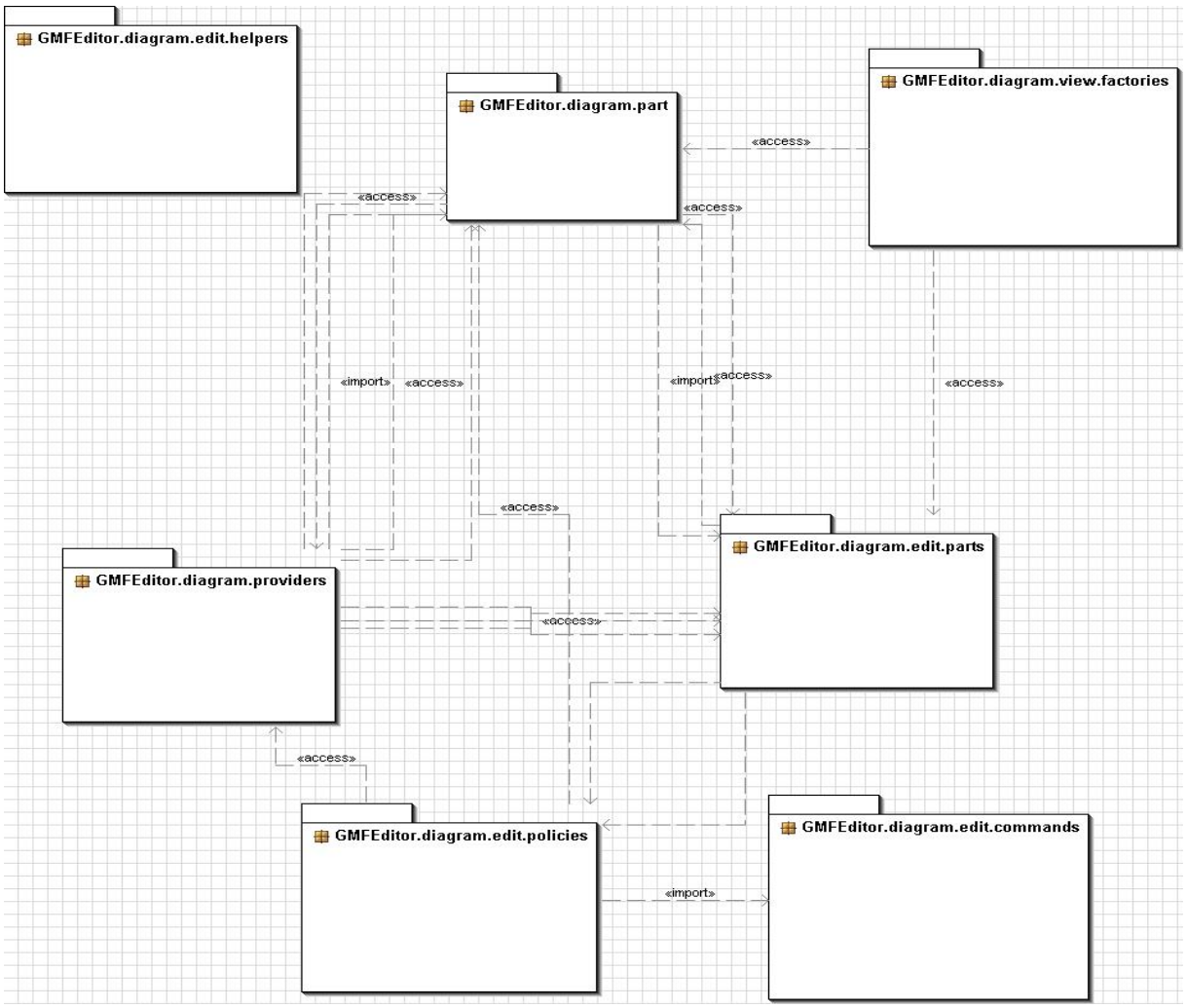

Abbildung 1: Paketdiagramm

Im Folgenden werden kurz Inhalte und Aufgaben der Pakete dargestellt:

***org.eclipse.gmf.ecore.*edit.commands**

Dieses Paket kann das EMF Modell verändern und bietet dafür sogenannte *IFiles* an, die verändert werden, wenn eine Aktion getätigt wurde. Dafür arbeitet die Klasse mit einem *Adapter*.

***org.eclipse.gmf.ecore.*edit.helpers**

Das Paket stellt für Objekte *Helper* bereit. Das Paket dient nun zum Auffangen von User *Requests*, die als selbstdefinierte *Commands* zurückgesandt werden.

***org.eclipse.gmf.ecore.* edit.parts**

Hier findet sich der Code für alle im Diagram verfügbaren Elemente, also alle Elemente, mit denen eine .ecore beschrieben werden kann. Diese enthalten Code für die definierte *Figur*, das Erzeugen des entsprechenden Elements und für die *Labels*.

***org.eclipse.gmf.ecore.e*dit.policies**

Dieses Paket enthält wie *edit.parts* den Code für jedes Diagramelement. Das Paket dient zur Behandlung der *Requests*, welche von *edit.parts* aufgerufen werden. Dabei kann eine *policie* von mehreren *EditParts* verwendet werden.

***org.eclipse.gmf.ecore.*expressions**

Dieses Paket stellt Methoden bereit, um eine einheitliche Sprache der XML zu definieren. Weiterhin beschreibt das Paket *Constraints*, also Beschreibungen für *Bedingungen*. Diese sind *Invariationen*, sowie *Vor-* und *Nachbedingungen* von *Operationen*. Dabei wird die *OCL* genutzt.

***org.eclipse.gmf.ecore.*part**

Dient zur Erstellung der Objekte für den Editor und erstellt daraus selbst das Erscheinungsbild des Editors.

***org.eclipse.gmf.ecore.*providers**

Dieses Paket stellt *Provider* zur Verfügung. Dafür gibt es *itemProvider*, die eine Zugriffsschicht bilden, welche unabhängig von der Oberfläche ist. Ein *itemProvider* implementiert alle für den Editor benötigten Schnittstellen. Der *Provider* ist also beim Modell angemeldet und sendet bei Veränderungen Nachrichten. Diese sind wiederum aus dem Modell generiert. Für jedes Objekt existiert ein solcher *Provider*.

***org.eclipse.gmf.ecore.*view.factories**

Dieses Paket stellt Standards zur Darstellung der Komponenten auf der grafischen Oberfläche bereit.

Dabei besitzt jedes Element, das auch im Editor gezeichnet werden kann, eine *Factory* Klasse.

ÄNDERUNGEN IM QUELLCODE:

1) Geänderte Pakete

Design.digram.editPolicies

Um die Anzahl der ausgehenden PolylineSource und PolylineTarget Referenzen zu kontrollieren, haben wir in der PolylineEditSemanticEditPolicy die Methode createRelationShipCommand um eine Abfrage ergänzt, die prüft ob diese Referenzen schon existieren. Wenn sie schon existieren werden die nachfolgenden Methoden blockiert und so kann keine neue Referenz mehr erstellt werden.

Um die Kompositionen und Containments zu kontrollieren änderten wird den Code in KnotenItemSemanticEditPolicy und fügten eine zu oben analoge Abfrage hinzu.

Um das Löschen von Objekten abzufangen haben wir die Methode getDestroyElementCommand in Compartment-, *Containment*- und *KnotenItemSemanticEditPolicy* geändert. Dort rufen wir einen Nutzer Dialog auf, der über weiteres vorgehen informiert. Entsprechend werden die *Requests* weitergegeben oder blockiert.

Allgemein werden für das Erstellen und Löschen von Elementen die Klassen CreateElementCommand bzw. DestroyElementCommand verwendet.

Für das setzen von Form-Werten für Knoten, Polylines und Compartments wird auf die Kasse CreateFormCommand und DestroyFormCommand zugegriffen.

Design.diagram.part

Dieses Package enthält wesentliche Klassen für das Anlegen und Öffnen einer Diagrammdatei (*.design_diagram*), sowie Klassen für das Erzeugen der zur Diagrammdatei gehörigen Modelldatei *(.design*). Aus diesem Grund, wurde dieses Package gewählt um die Erzeugung der *Dateien* zu realisieren. Wesentlich ist hier die Klasse MyDiagraMEditorUtil. Dort werden, in der Methode createNewDiagramFile, die Commands zur Erzeugung der Modelle initiiert. Zur Speicherung von Änderungen in den Dateien wurde zusätzlich noch die Klasse *DocumentFileProvider.java* verändert. Diese greift auf die Klasse *MyFileProvider.java* im neue angelegten Package *Design.diagram.fileprovider* zu.

2) Geschriebene Pakete

Design.diagram.geditCommands

Um die *GMF* Modelle zu erzeugen wurde dieses Paket erzeugt. Es enthält vier *Command* Klassen die von *AbstractTransactionalCommand* erben und so zur Veränderungen der Modelle dienen. Dabei existiert für jedes Modell eine Klasse, die über die entsprechende *Resource* auf das Modell zugreift und ein Wurzel Element in das Modell speichert. Alle vier *Commands* werden in *MyDiagramEditorUtil* initialisiert.

Design.diagram.dialog

Dieses Paket enthält nur die Klasse *UserDialog*. Deren Funktion ist es mit dem Nutzer in Verbindung zu treten. So werden Löschabfragen eines *Knotens* kontrolliert und ein Begrüßungsdialog angezeigt.

Design.diagram.connectionRegistries

In diesem Paket finden sich Klassen, die die Anzahl der erstellen *Kompositionen*, *Containments* und *PolylineSource/TargetReferencen* zu kontrollieren. Dabei werden die erstellten Diagramobjekte, die eine solche Referenz enthalten in den jeweiligen *RegistryEntry's* gespeichert. Diese werden wiederum von der drüber liegenden *Registry* Klasse verwaltet. So lassen sich Objekte, an denen eine Referenz anliegt oder abgeht *registrieren* und über *unregister* Methoden aus der *Registry* löschen. Weiter existieren Methoden, die über ein Objekt feststellen, ob dieses über Referenzen verfügt. Mit diesen Klassen beschränken wir die ausgehenden *Source* und *Target* Referenzen einer *Polyline*, sowie *Kompositionen* eines *Compartments* und *Containments* eines *Knoten*s, auf jeweils eine Instanz. So wird auch die Anzahl der *Containments* auf ein *Compartment* auf eine Instanz reguliert. Die find Methoden der *Registries* werden in den jeweiligen *EditPolicies* aufgerufen. Beim erstellen eines Objektes wird die *register* Methode der *Registry* in *MyDataType* und entsprechend beim Löschen die *unregister* Methode in *MyElementTypeFactory* aufgerufen.

Design.diagram.geditMapping

Die wesentlichste Klasse dieses Paketes ist die *MappingEntry*. Diese wird beim Erstellen oder Löschen eines Objektes aufgerufen und sorgt dafür, dass ein entsprechender Eintrag in die *GMFMap* gesetzt wird. Das Paket dient also zur Generierung der Instanzen für die *GMFMap*. Ein *MappingEntry* bekommt eine *MappingConstant* und eine Instanz von *MyDataType* übergeben. Dabei enthält die Klasse *MappingConstants* diese Konstanten, welche die ausgeführte Aktion beschreiben. Die Instanz von *MyDataTyp* enthält die Informationen zum erstellten Objekt. Der *Konstruktor* der *MappingEntry* fragt nun diese *MappingConstants* ab und ruft daraufhin eine Methode auf, um den Eintrag zu setzen und zu verarbeiten. Diese Methoden erstellen Instanzen des *GMF Mappings* und setzen es über den *GMFMapObjectCreator* in die *GMFMap*. Die erstellten Instanzen und ihr *Diagramobjekt* werden wieder in einer *Registry* verwaltet und gespeichert. Dafür existiert für jede Art eines *Mapping* eine *Registry*, welche in der *MappigRegistry* gespeichert wird. Diese enthält die *register* und *unregister* Methoden, sowie Operationen, um nach den Instanzen zu suchen.

Design.diagram.fileprovider

MyFileProvider.java übernimmt Änderungen in den *Resources* der Dateien.

Außerdem wird eine Instanz der Klasse der MyDataType.java erzeugt, welche die *diagram*-, *ecore*-, *gmfgraph*- und *gmftool*-Struktur beinhaltet und verwaltet. Mittels der zusätzlich bereitgestellten Klassen in diesem Package, ist es *MyDataType.java* möglich, diese Strukturen zu verändern. Damit ist diese Klasse der Kern des gesamten Editors.

DESIGNENTSCHEIDUNGEN:

Um ein gutes Maß an Funktionalität bieten zu können, aber auch eine hohe Benutzerfreundlichkeit zu gewährleisten, wird die Auswahl, der in der *ecore*-Datei zu erstellenden Objekte vom System übernommen. Das heißt, dass zu jedem *diagram-Objekt* im *Editor*, das dazugehörige *ecore-Objekt* fest vorgegeben ist. Damit ist die Integrität der logischen Struktur gesichert, sodass es zu keinen Fehlern kommen kann. Dagegen ist es möglich, dass die grafische Repräsentation der *diagram-Objekte* vom Benutzer selbst gewählt werden kann. So können zum Beispiel *Knoten*, mit *Rechtecken* und weiteren *Formen* versehen werden. Diese *Formen* sind direkt im *diagram-Objekt* sichtbar, um eine gute Übersichtlichkeit gewährleisten zu können. Dabei sind für jedes *diagram-Objekt* defaults vorgeschrieben, sodass keine ungültige *gmfgraph*-Datei entstehen kann. Außerdem werden für *child-Objekte*, wie etwa *Attribute*, die *Formen* fest vorgegeben, da Änderungen wenig Sinn ergeben und das Arbeiten mit dem *Editor* unnötig komplizieren würden. Damit hat der Benutzer eine Vielzahl an Optionen, seinen *Editor* zu modellieren, ohne dabei mit komplizierten Problemen konfrontiert zu werden.

MVC-PRINZIP:

Der Ecore-Editor baut auf der MVC-Architektur auf. Die Trennung des Models von dem View und dem Controller wird dabei zum Teil bereits durch das verwendete GEF vorgegeben:

Das Modell:

Das Modell bilden die Objekte aus dem EMF – Modell. Dabei werden die EMF Objekte durch *Observer* genutzt, die auch gleichzeitig andere Objekte beobachten und damit auch zum Controller gezählt werden können.

Das Modell generiert für den Controller einige *Provider*, wobei es zu jedem Objekt einen *Provider* gibt.

Das View:

Das View wird durch die Ansicht des Editors dargestellt. Auf ihr findet man die grafische Oberfläche sowie eine Toolbar in der verschiedenen *Figures* ausgewählt werden können.

Durch *Drag and Drop* werden diese auf die Oberfläche gezogen und dort vom View dargestellt.

Der Controller:

Der Controller wird durch die *EditParts* umgesetzt. Sie werden für alle im Diagramm definierten Modellelemente erzeugt. Diese enthalten Code für die definierte *Figur*, das Erzeugen des entsprechenden Elements und für die *Labels*. Dazu gehören *Erzeugen*, *refresh*, *direktes Editieren*, *löschen*, *Listener* und *Notifications*.

Weiter gibt es sogenannte *Provider*. Diese sind beim Modell angemeldet und gehören jedem *Objekt* an. *Provider* senden wie *Listener* Nachrichten bei Veränderungen, zum Beispiel an die *Factory*. Verbindungen zwischen Modell und Controller werden über *EditPartFactories* gesteuert. Dazu wird in der Factory das dazugehörige *EditPart* Element gesucht und die Verbindung verknüpft.

Dynamisches Modell:

Das dynamische Modell spiegelt besondere Anwendungsfälle, den Informationsfluss, sowie die Interaktion zwischen Nutzer und System wieder.

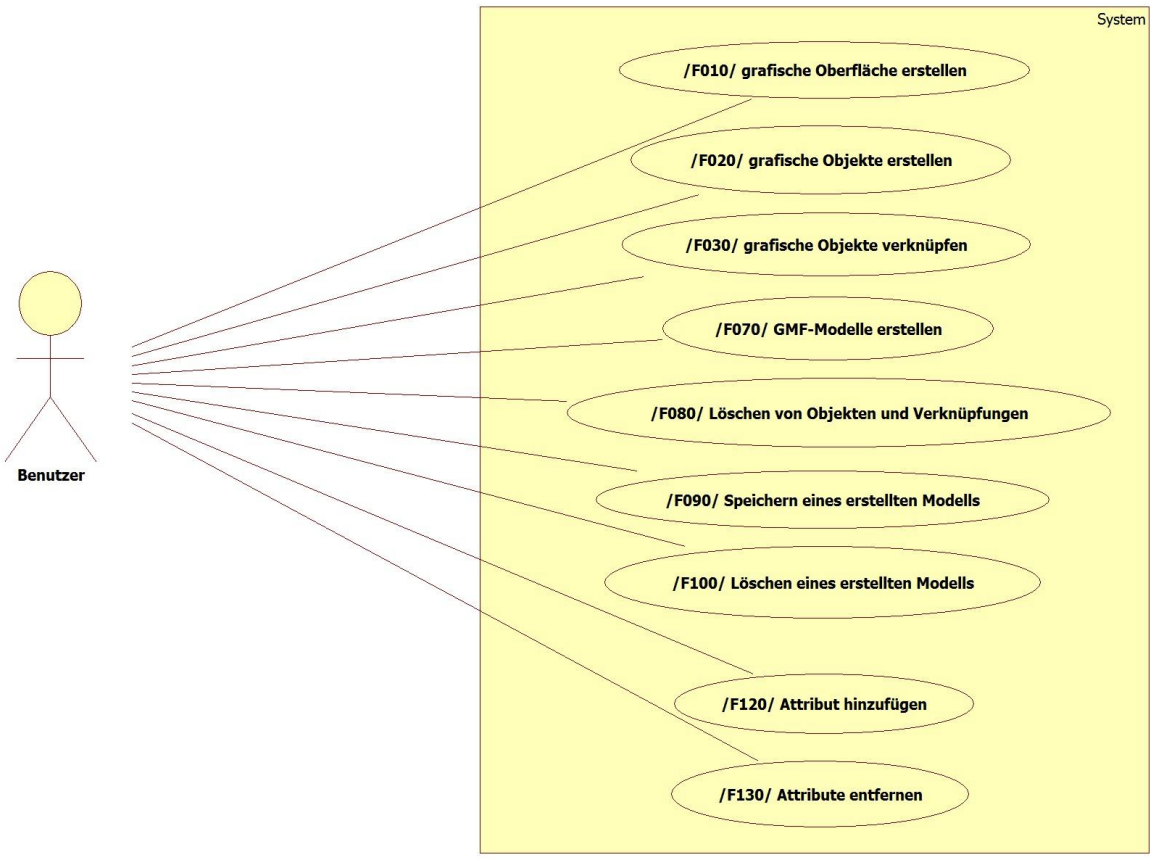

Abbildung 2: Use-Case-Diagramm

Anwendungsfälle sind im nachfolgenden Diagramm zu sehen. Dieses zeigt die wesentlichen Produktfunktionen und die Entwicklungsmöglichkeiten beim Entwerfen eigener Modelle.

Im Folgenden beziehen wir uns auf ausgewählte Aktivitäten, die die Funktionsweise des Editors genauer beschreiben und einen ersten Eindruck der Funktionalität des späteren Produktes wiedergeben.

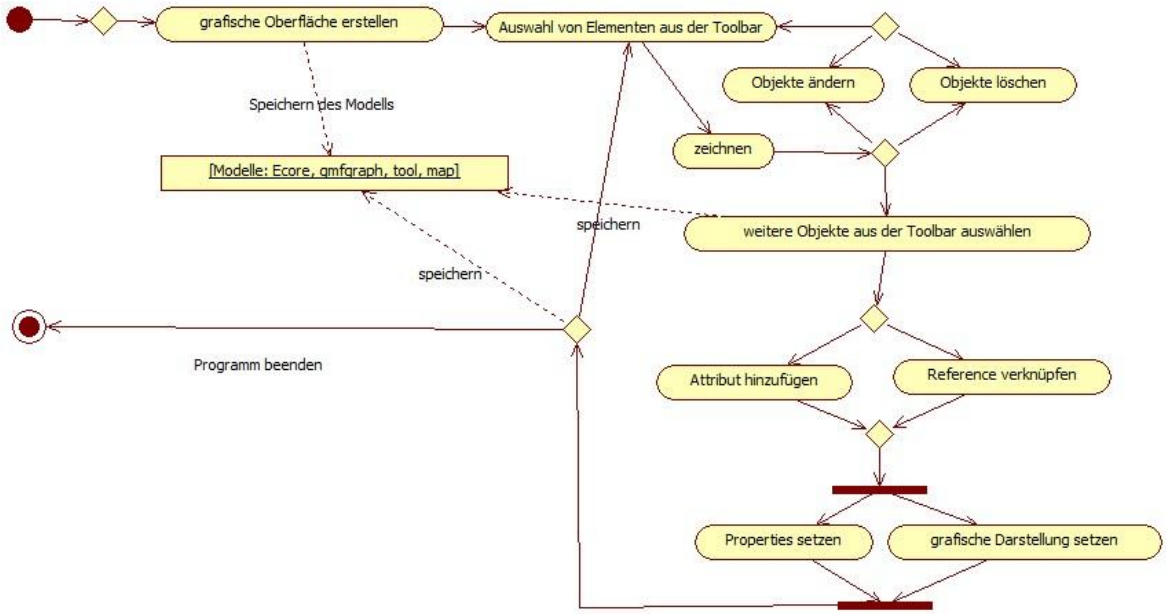

Abbildung 3: Aktivitätsdiagramm

Das Zustandsdiagramm zeigt den normalen Arbeitsfluss mit dem Editor. Ausgehend vom Startzustand (hier in rot), der als Start des Editors angesehen werden kann, bieten sich dem Benutzer zwei Möglichkeiten, er kann entweder ein bereits bestehendes Modell laden oder beginnen, auf der vorhandenen Oberfläche zu zeichnen.
Hierzu steht ihm die Toolbar zur Verfügung, die diverse Elemente zur Modellierung bereitstellt. Aus dieser Toolbar können per Drag'n'Drop die gewünschten Elemente gezogen werden, welche damit direkt auf die Zeichenfläche gezeichnet werden. Die Elemente repräsentieren je nach Wahl Objekte im ECore-Modell: EClasses, Aggregationen, usw. Im obigem Diagramm wird dies durch einen Zyklus dargestellt, der sich je nach Benutzerwunsch beliebig oft wiederholt.
Sind alle Elemente positioniert , kann der Benutzer weitere Einstellungen vornehmen – den Elementen der Zeichenfläche weitere Attribute hinzufügen. Diese können entweder direkt in das Element geschrieben werden oder lassen sich über den PropertyView festlegen. Ebenso können Elemente über References verknüpft werden.
Auch dieser letzte Schritt ist wieder an einen übergeordneten Zyklus gebunden – dem Benutzer steht frei, auch nach Festlegen genauerer Elementattribute, weitere Elemente auf der Zeichenfläche zu erstellen.
Hat der Benutzer alle gewünschten Aktionen erledigt, steht es ihm frei, nun die Modelldateien zu erzeugen oder das Programm schlussendlich zu beenden (Endzustand, hier rot und doppelt umrandet).

Um den Arbeitsfluss für den Benutzer mit dem entworfenen Editor noch etwas genauer zu beschreiben, fassen wir ein Beispielszenario in Form eines Sequenzdiagramms auf: Die Erstellung eines Zustandsautomation mit unserem Editor.

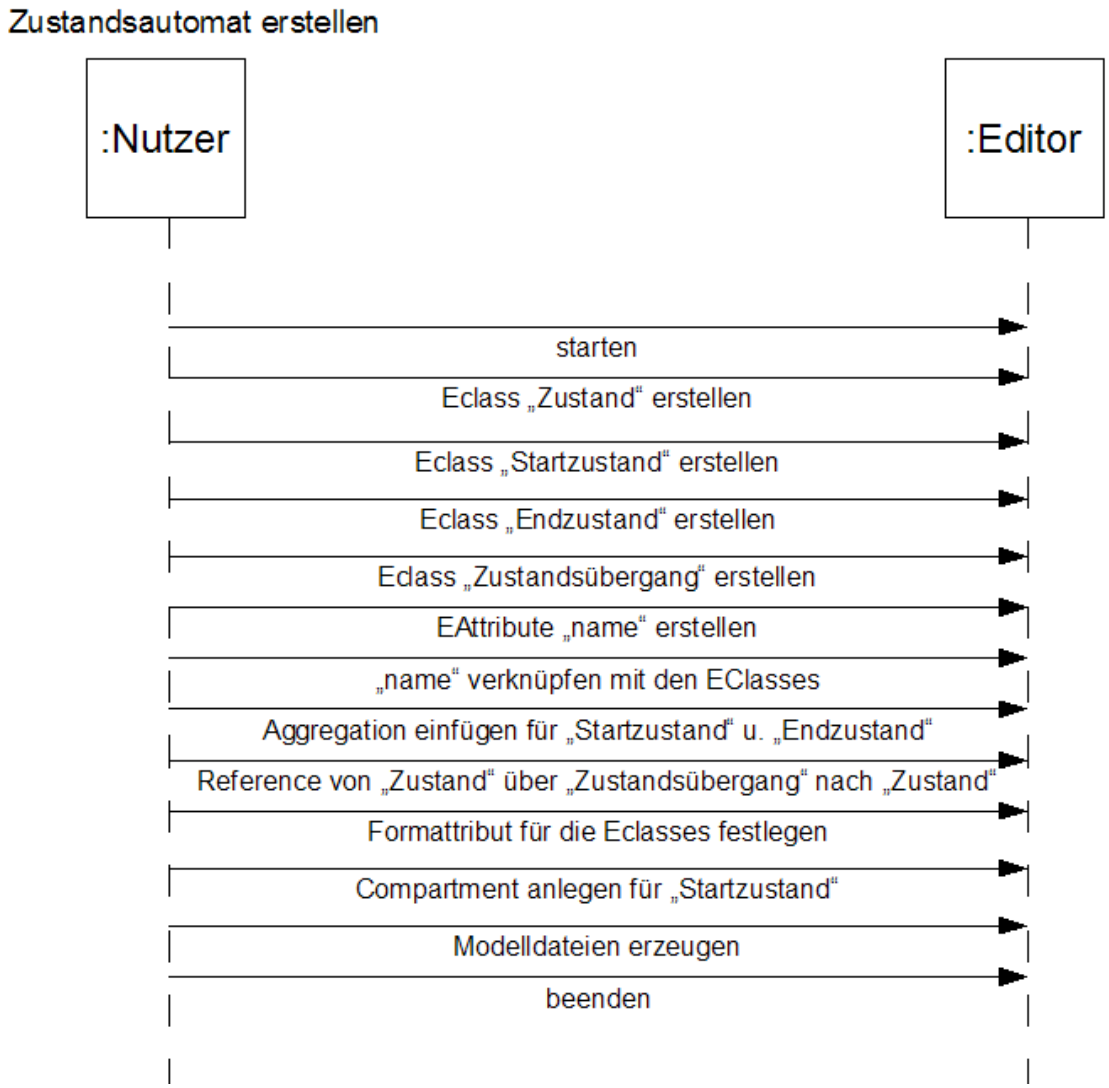

Abbildung 4: Sequenzdiagramm

Dieses Beispiel verdeutlicht die Nutzung des Editors bei der Erstellung eines einfachen Zustandsautomaten.