1. Umsetzung der Model-View-Controller Architektur

1.1 Einleitung in das MVC Paradigma

Ein großer Teil der Funktionalität von objektorientierten Systemen spielt sich in der Interaktion zischen System und Anwender ab. Ein weitverbreiteter Ansatz zur Lösung dieses Problems ist das MVC (Model-View-Controller) Paradigma. Mit diesem Modell wird ein Interaktionsmuster in der Präsentationsschicht von Software beschrieben.

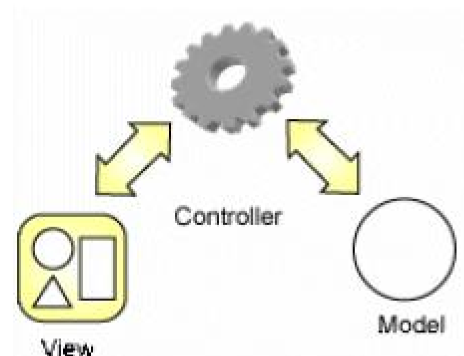

Abbildung 1: MVC - Paradigma

Im Modell selbst unterscheidet man die 3 Schichten – Modell, View und Controller. Wobei alle Schichten strikt voneinander getrennt werden.

Das Modell:

Hier werden alle persistenten Daten gespeichert. Dabei dient das Modell als „Container" für die Daten. Das Modell selbst kennt keine weiteren Teile des Programms, nur die Daten selbst. Änderungen an bestehende Daten werden über sogenannte Listener mitgeteilt.

Das View:

Im View werden die Daten der Modellschicht abgebildet. Es enthält aber keine eigene Daten sowie keine Modelllogik. Genau wie das Modell kennt das View keine weiteren Bestandteile des Programms.

Der Controller:

Die Hauptaufgabe des Controllers besteht darin, das View und Model miteinander zu verbinden. Dabei leitet er die Kommunikation vom Modell an den View weiter.

1.2 Umsetzung der MVC Architektur im Fremdobjekt

Die Umsetzung des Ecore Editors baut ebenfalls auf einer MVC – Architektur auf. Dieser wurde mit GMF entwickelt, setzt aber auf Teile des GEF auf. Insofern ist die Architektur beim Editor selbst etwas abgewandelt, basiert aber auf dem unter 1.1 beschriebenen Prinzip.
Im Modell selbst stehen persistente Daten, die über *Commands* des Views geändert werden können. Die Benutzeransicht des Editors beschreibt die View. In ihr werden *Figures* mit Hilfe der *2DAPI* gezeichnet. *EditParts* beschreiben den Controller, der *Requests* auffängt und bearbeitet.

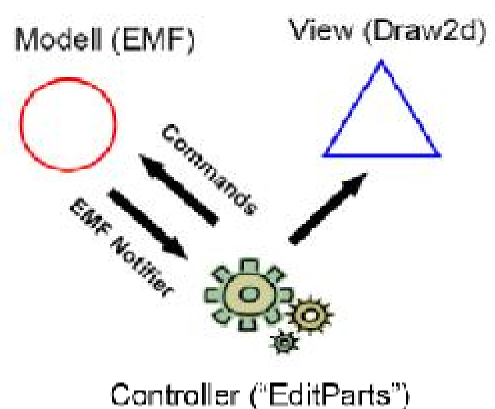

Abbildung 2: GEF MVC

Im Folgenden beschreiben wir die Elemente der Architektur. Genaueres über die genannten *Packages* können im Abschnitt 2 – Strukturierung der *Packages* – nachvollzogen werden.

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. Diese müssen sich zuvor jedoch als *Listener* anmelden. Das Modell generiert für den Controller einige *Provider*, wobei es zu jedem Objekt einen *Provider* gibt. Ein Modell kann durch die Methode *EditPart.setModel/getModell* geändert werden. Benachrichtigungen werden über *EditPart.activate()* oder über *EditPart.notify()* gesendet.

Das View:

Es werden 2 Views bereit gestellt, einen GraphicalViewer sowie einen TreeViewer. Für den Ecore Editor ist jedoch nur der GraphicalViewer relevant.
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 Drog* werden diese auf die Oberfläche gezogen und dort vom View dargestellt. Dazu werden sogenannte *EditPartViewers* benötigt, die eine einzelne Darstellung des *EditParts* darstellen.
Die grafische Darstellung wird vom Packet *org.eclipse.gmf.ecore.view.factories* übernommen. Dieses enthält Standarts zur Darstellung der verschiedenen *Figures*. Dabei zeichnen sich *Figures* selbst und rekursiv ihre Kinder. Neue Sichten auf das Diagramm können durch den *ViewService* verwirklicht werden.
Eine *Figure* wird über *EditPart.createFigure* erzeugt und über *.refresh* geändert. Die Methode *EditPart.createEditPolicies()* erzeugt *EditPolicies* zum erzeugen der *Commands* für Aktionen im View. Über *getSource()* und *getTarget()* kann man auf diese *Policies* zugreifen.

Der Controller:

Der Controller wird durch die *EditParts* umgesetzt. Dazu existieren 3 wichtige Methoden:

- createFigure()
  - Erstellen der *Figure* zu dieser *EditPart*
  - Verbindet Controller und View
- refreshVisuals()
  - Aktualisieren der Daten in der View mit den Daten des Modells
- getModelChildren()
  - erstellt eine Liste von Modellklassen, die logisch von dem zum *EditPart* korrespondierenden Modellelement gehören

Deren *Requests* werden durch *EditPolicies* behandelt. Im Controller werden *EditParts* 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*.

Der Controller sorgt also dafür, dass das ausgewählte Element aus der Toolbar im Modell instanziiert wird und gleichzeitig auf dem View, sprich der grafischen Oberfläche abgebildet wird.

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.

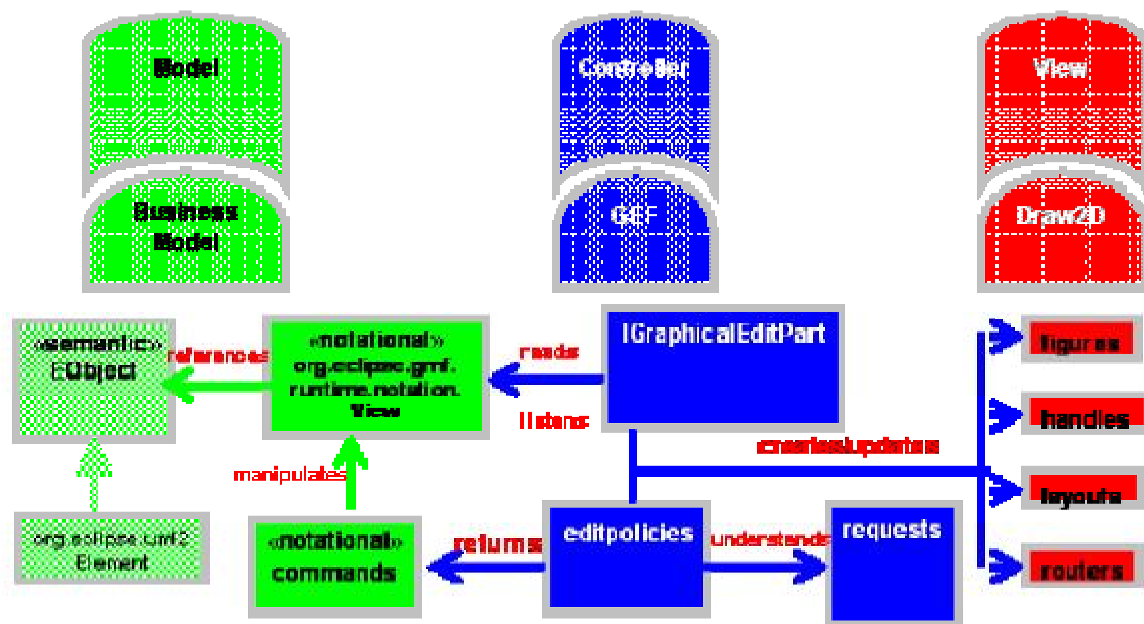

Abbildung 3: MVC Übersicht

2. Strukturierung der Packages

Das Example *org.eclipse.gmf.ecore.editor* besteht aus 8 Java Packages, einer *Manifest* Datei mit den Metainformationen, einer *plugin.properties*, sowie einer *plugin.xml*. Dazu werden die JRE System Library und PlugIn Dependencies benötigt.

Der Editor besteht aus folgenden Paketen

- *org.eclipse.gmf.ecore.*edit.commands
- *org.eclipse.gmf.ecore.*edit.helpers
- *org.eclipse.gmf.ecore.* edit.parts
- *org.eclipse.gmf.ecore.*edit.policies
- *org.eclipse.gmf.ecore.*expressions
- *org.eclipse.gmf.ecore.*part
- *org.eclipse.gmf.ecore.* providers
- *org.eclipse.gmf.ecore.* view.factories

welche wieder Klassen enthalten.

Im Folgenden erläutern wir kurz die Inhalte und Aufgaben der Pakete.

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

Das Paket enthält nur die Klasse *EcoreReorientConnectionViewCommand*, welche von *AbstractTransactionalCommand* erbt. Diese 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 jedes Objekt(also *EAttributes*, *EClasses*, *EOperations*, ...) einen *Helper* bereit. Die Klassen des *Packages* kann man in zwei Teile gliedern. Zum einen die *editHelperAdvice*, die allgemein von *AbstractEditHelper* erben. Und zum anderen die *editHelper*, welche von *EcoreBaseEditHelper* erben. Dabei stellt *ecoreBaseEditHelper* eine Spezialisierung von *abstractHelper* dar.
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*. Dazu gehören *Erzeugen*, *refresh*, *direktes Editieren*, *löschen*, *Listener* und *Notifications*.

*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

Um eine einheitliche Sprache beim definieren des *XML* Dokumente zu haben, wird das *Package .expressions* verwendet. Dieses stellt dafür Methoden bereit, um eine einheitliche Sprache zu definieren. Diese werden in den *extension parts* der *XML* verwendet werden und ist nicht an bestimmt gebunden. 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 sogenannte *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 (zum Beispiel *ITreeContentProvider*, *ITreeItemContentProvider*). 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 Standarts zur Darstellung der Komponenten auf der grafischen Oberfläche bereit. Dabei besitzt jedes Element, das auch im Editor gezeichnet werden kann, eine *Factory* Klasse. Jede Klasse besitzt eine *createStyles()* und eine *decorateView()* Methode.
CreateStyles() gibt eine Liste der *Styles* für die Komponenten zurück. *Styles* werden dabei durch eine *create* (*Bsp.:createFontStyle()*) Methode in *NotationFactory* aus dem Paket *org.eclipse.gmf.runtime.notation.Notationfactory* erzeugt. Durch *decorateView()* wird der Note „dekoriert“. Dabei werden über *getViewService() Nodes* angelegt.

3. Anbindung an die Eclipse IDE

Nachdem herunterladen des *Examples* Archives, muss dieses in den Ordner der Eclipse Installation entpackt werden. Nachdem man ein neues Projekt erstellt hat, kann man den Editor unter dem Menüpunkt *New -> Example -> ...> Ecore Diagram* laden und starten. Daraufhin befindet man sich im grafischen Editor, mit dessen Hilfe man sich ein Diagramm bauen kann, welche eine *Ecore* Datei beschreibt und generiert.

In Eclipse werden PlugIns dynamisch bei jedem Start geladen. Es können mehrere PlugIns gleichzeitig geladen werden. Dabei können Fehlkonfigurationen mit dem Befehl *–clear* behoben werden. Innerhalb der laufenden Eclipse Workbench ist das PlugIn eine Instanz einer PlugIn – Klasse. Diese stellt Konfiguration und Verwaltungsdaten bereit. Eine PlugIn - Klasse baut dabei auf dem Package *org.eclipse.core.runtime.PlugIn* auf, welche zur abstrakten Darstellung der PlugIns dient. Im Ecore Editor selbst wird diese PlugIn – Klasse im Paket *org.eclipse.gmf.ecore.part* implementiert. Diese erbt wiederum von der Superklasse im *runtime*.

Alle PlugIns werden über eine plugIn.xml bei Eclipse angemeldet. Dazu teilt sie der Eclipse IDE mit, wie das PlugIn selbst aktiviert werden soll. Eine darauf aufbauende Manifest .mf Datei zeigt die Inhalte der grafischen Darstellung und dient zur Änderung der *plugIn.xml*. Die XML Datei enthält dabei auch sogenannte *Extension Points*, welche das Erstellen der Ecore aus de Editor beschreibt. Weiterhin enthält das Gesamtpaket eine *Properties* Datei, die die Randbedingungen und Optionen enthält.

4. Beschreibung der internen Abläufe des Editors bei der Modellierung einer EClass

Zur Modellierung einer *EClass* wählt man zunächst das Objekt „*EClass*" aus der *Toolbar* des Editors aus und zeiht es per *Drag and Drop* auf die Zeichenfläche. Damit wird ein Request ausgelöst, indem das Erstellen der *EClass* beantragt wird. *EClassEditPart* verarbeitet diesen Request nun und leitet ihn als Command an *EClassSemanticItemEditPolicy* weiter. Dort wird der Command zum instanziieren einer *EClass* erzeugt. Dieser Command wird nun an das *CreationTool EClass* geleitet, welche daraufhin das *Rectangle* zeichnet. Dabei wird die Methode *createView()* aufgerufen. Das Package *.view.factories* sorgt für die einheitliche Darstellung der Grafik, sowie für die Änderung im Metamodell. Die bei der Erzeugung der *EClass* entstehenden *Commands* werden nacheinander in einen *CommandStack* geleitet, der sequentiell abgearbeitet wird. Ist der *Stack* leer, ist die *EClass*

Abbildung 4: Beispiel eines Extension Points

erfolgreich modelliert.