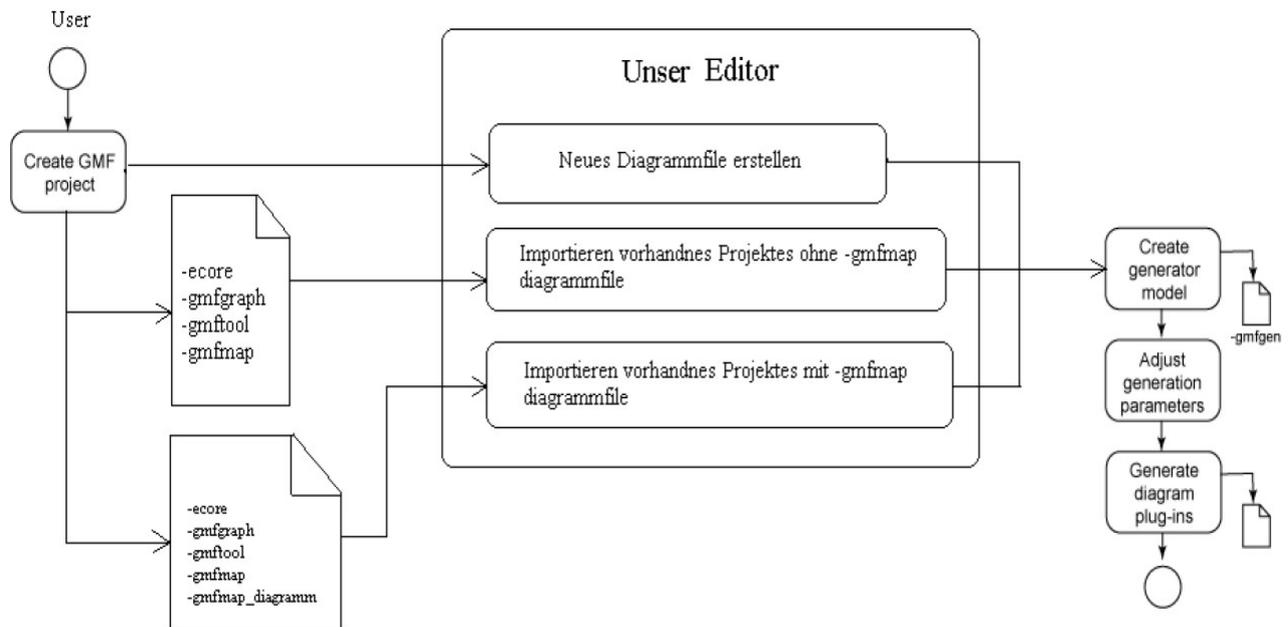


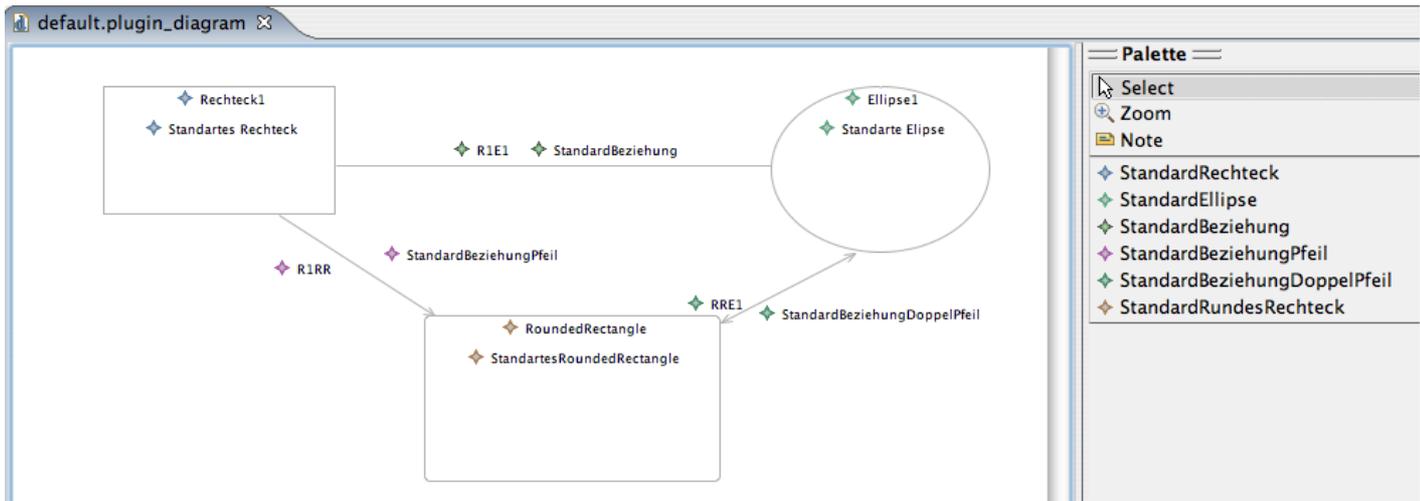
1. Allgemeines

Unser auf GMF basierender Editor soll die Möglichkeit bieten, die Arbeit mit GMF zu erleichtern, indem der Arbeitsschritt des Erstellens und Zusammenfügens von ecore, gmgraph, gmftool und gmmap mit graphischen und automatischen Mitteln vereinfacht wird. Dazu entwickeln wir auf der Basis von GMF einen Editor und ergänzen ihn um einige Funktionalitäten, sodass er sich wie folgt in den Modellierungsprozess einliedert:



2. Produktübersicht

Die Benutzeroberfläche des Editors ist typisch für einen GMF-Editor, sie besteht aus einer Zeichenfläche und einer Arbeitspalette auf welcher die einzelnen CreationTools zu Toolgroups zusammengefasst angezeigt werden. Er bietet die Möglichkeit, Objekte sowie Beziehungen dazwischen zu erzeugen und deren Eigenschaften zu manipulieren. Die meisten dieser Funktionalitäten werden bereits standardmäßig von GMF-Editoren zur Verfügung gestellt, der entscheidende Unterschied ist aber, dass Funktionen wie die Standardbeschriftung und Farbänderung vom Diagramm in die neuen Projektdateien übernommen werden und dass der Editor gleich mit mehreren Projektdateien arbeitet, die er laden oder erzeugen kann, und die er beim Speichern entsprechend zusammenfügt, indem er zum Beispiel eClasses erzeugt und die passenden Mapping-Einträge vornimmt. Aus diesen Projektdateien kann der Benutzer, wenn er weiß, wie man GMF bedient, sehr schnell einen lauffähigen Editor erzeugen.



Screenshot aus dem laufenden Editor

/F10/ Projekt erstellen und benennen:

Der Benutzer muss zuerst über New... -> GMF -> New GMF Project... ein GMF-Projekt erzeugen, dann wechselt er in den model-Ordner und erzeugt dort über New ... ein neues gmffmap_diagram-File in dem Ordner.

/F20/ Projekt öffnen:

Ein vorhandenes Projekt ohne gmffmap_diagram wird geöffnet, indem auf die map-Datei rechtsgeklickt und dann „Open Gmffmap diagram“ ausgewählt wird.

Ein vorhandenes Projekt mit gmffmap_diagram wird mittels dieses gmffmap_diagram geöffnet.

/F40 – F110/ Elemente erzeugen und deren Eigenschaften verändern

Die neuen Elemente lassen sich wie gewohnt erzeugen, indem man in dem Editorfenster das jeweilige CreationTool auswählt und auf die Zeichenfläche zieht. Das Löschen erfolgt nach Auswahl über die Löschen-Taste.

Überblick über Elementeneigenschaften:

- ID und Beschriftung:

Man sieht, dass jedes Element zwei Label hat. Das obere Label ist das ID-Label und dient, um dem Element die Namen für alle Repräsentationen in den Projektdateien zuzuweisen: eClass, Mappingeintrag, Figure usw. Diese ID ist beim Erzeugen des Elements „ID“ und muss geändert werden, es darf sie aber nur je einmal geben, sonst gibt es beim Speichern eine exception.

Im fertigen Endprodukt werden die ID-Labels nur noch auf der Beschriftung des Creation Tool angezeigt. Jedes Element bekommt außerdem ein Label, auf das die Standardbeschriftung übertragen wird. So lässt es sich machen, dass auch mehrere Elemente die gleiche Beschriftung tragen, dabei aber in Form, Farbe und Klasse unterschiedlich sind. Beispiel in unserem Endprodukt: Die Beziehung, die einen Zustandsübergang repräsentiert, wird mit der dafür eingelesenen Variable beschriftet, ist aber durch ihre interne ID dennoch eindeutig identifizierbar.

Die Beschriftung kann auch leergelassen werden, dann wird ihr Wert aus der ID ermittelt.

- Farbe:

Die Farbe eines Elements ändert man über sein Kontextmenü: Format -> Fill Color/Line Color, diese Änderungen werden abgefangen und im Gmffgraph des Endprodukts gespeichert.

- minimale und maximale Anzahl:

Jedes Element bekommt noch im ecore des Editors ein Attribut namens minimalAnzahl und maximalAnzahl, mithilfe derer man dann im PropertiesView die zulässigen Anzahlen im Endprodukt einstellen kann. Diese Daten werden ab dem vierten Release in den jeweiligen Containment-Referenzen im ecore gespeichert.

- Regeln für Beziehungen:

Die verbundenen Klassen können im PropertiesView bei src und trg aus denen im ecore des Projekts

ausgewählt werden, aber der normale Weg ist das Ziehen von Klasse zu Klasse bei der Erstellung der Beziehung.

/F160/ Projekt speichern

Durch Auswahl von File -> Save wird das Projekt wie gewohnt gespeichert, dabei werden die Projektdateien im selben Verzeichnis wie die Diagrammdatei angelegt.

/F170/ Projekt schließen

Geschieht durch Schließen des Editorfensters, die erzeugten Dateien werden nicht gelöscht.

3. Aufbau und Funktionalität des Programms

Unser Editor ist ein mit GMF erzeugtes eclipse-Plugin, er hat also vier grundlegende Modelldateien: ecore, gmfgraph, gmftool, gmfmap. Aus diesen wurden mehrere Pakete Javacode erzeugt, von denen das .diagram-Paket am wichtigsten ist. Der von GMF erzeugte Teil ist lauffähig, die zusätzlichen Funktionen werden durch Modifikationen im Klassensystem realisiert.

3.1 Umsetzung der MVC-Architektur

3.1.1. Model:

Der ecore sowie die graph-, tool- und map-Modelle sind von uns erstellte Vormodelle unseres Editors. Aus ihnen werden umfangreiche Javapakete erzeugt, die dann das statische Datenmodell darstellen. In diesen Klassen enthalten sind genauso Modelldaten enthalten, wie auch Viewdaten und Controllerdaten.

Die Modelldaten sind (abgesehen von Hilfs- und Providerklassen) im Paket Prototyp/src/Plugin.impl enthalten.

Das Vormodell wird in 3.2 dargestellt.

3.1.2. View:

Das View ist die Benutzeroberfläche des Editors und wird von den zugrundeliegenden Klassen gesteuert, die aus den Vormodellen erzeugt werden. Die meisten Klassen werden von eclipse und GMF zur Verfügung gestellt, als Schnittstelle während der Laufzeit und zum Erzeugen des Editors fungieren Requests, die von GMF an den Editor geschickt werden und im Paket edit.policies aufgefangen und verarbeitet werden.

Diese edit.policies werden durch edit.part-Klassen im View installiert. Diese werden unter edit.part von der Klasse PluginEditPartFactory eingerichtet, die somit ein Modell für das View darstellt.

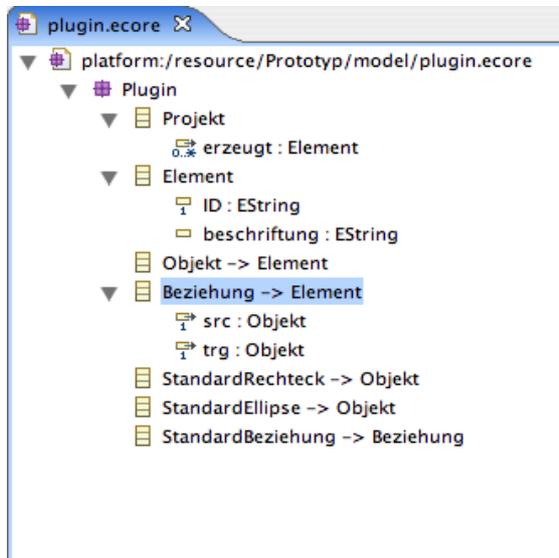
3.1.3. Controller:

Das Package edit.policies repräsentiert bei unserem Editor die Controller der einzelnen Elemente (s. **3.3.1 edit.policies**)

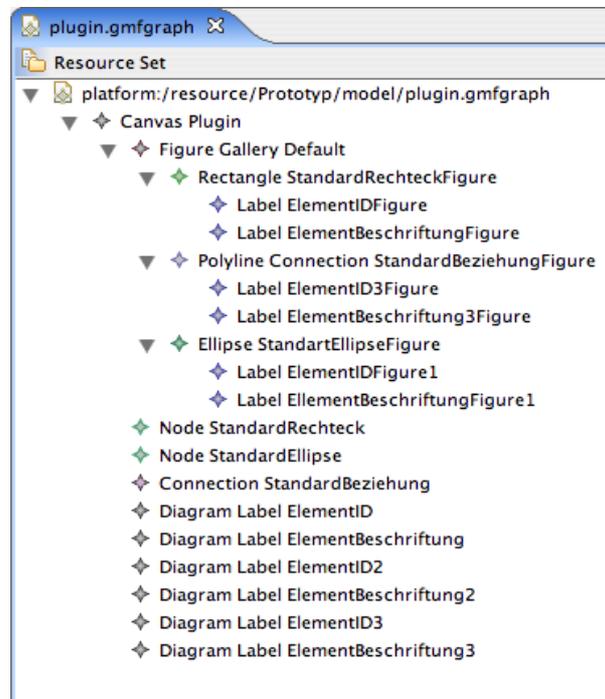
Zum Controller zählen allerdings auch die Klassen in part, wo Speicher- und Öffnungsvorgänge sowie die Initialisierung des Editors gesteuert werden. (s. **3.3.2 part**)

3.2 Vormodell: Projektdateien ecore und gmfigraph

ecore:



gmfigraph:



Hier sieht man den Aufbau unseres ecores: Das Diagrammelement ist Projekt und alle Standardelemente erben von Objekt oder Beziehung. Es gibt bis jetzt drei Standardelemente: Standardrechteck, -ellipse und -beziehung. Diese haben je eine Form und zwei Label, die in gmfigraph aufgeschlüsselt sind.

3.3 statisches Modell: Paketbeschreibungen der von GMF erzeugten Pakete

3.3.1. edit.policies

Hat vor allem Controllercharakter: Empfängt Requests aus den View-Teilen (edit.parts) und wandelt diese in Commands um, die von GMF ausgeführt werden. Diese Package dient zur Festlegung der Regeln bei bestimmten Interaktionen des Benutzers.

Alle Nodes haben eine GraphicalNodeEditPolicy, die von PluginGraphicalNodeEditPolicy erbt und keine zusätzlichen Eigenschaften hat. Diese ist fürs Node zuständig. Alle Nodes haben mindestens eine CanonicalEditPolicy: Eine die von CanonicalEditPolicy erbt und wenn sie Referenzen haben eine die von Canonical-ConnectionEditPolicy erbt. Hier wird eine Liste der Kinder des semantischen Objekts ausgegeben. Auch Connections haben so eine CanonicalEditPolicy, mit der semantische Verbindungen erstellt und zB. source und target ausgegeben werden: protected EObject getSourceElement(EObject relationship). Die Methode refreshSemantic() dient zur Aktualisierung des Views.

Alle Elemente haben eine ItemSemanticEditPolicy, die von ECoreBaseItemSemanticEditPolicy erbt. Wichtige Methoden: getSemanticElement, getSemanticCommand, getMoveCommand, getCreateCommand, getDuplicateCommand, getDestroyCommand ! Verbindung vom View zum Modell.

3.3.2. part

CONTROLLER-Paket - startet den Ablauf, initialisiert und öffnet Diagramme, Editor, Toolbar, Wizards und dergleichen. Hängt den Klassen mithilfe der VIDRegistry ihre edit.parts aus dem Paket edit.parts an. Enthält die meisten Schnittstellen des Plugins zu eclipse und gmfigraph.

Hier wird auch die Speicherung vorgenommen, deswegen haben wir unsere Speicherklasse PluginEcoreNeu hier

eingefügt. Näheres dazu steht im 2. Datenflussdiagramm.

3.3.3. edit.parts

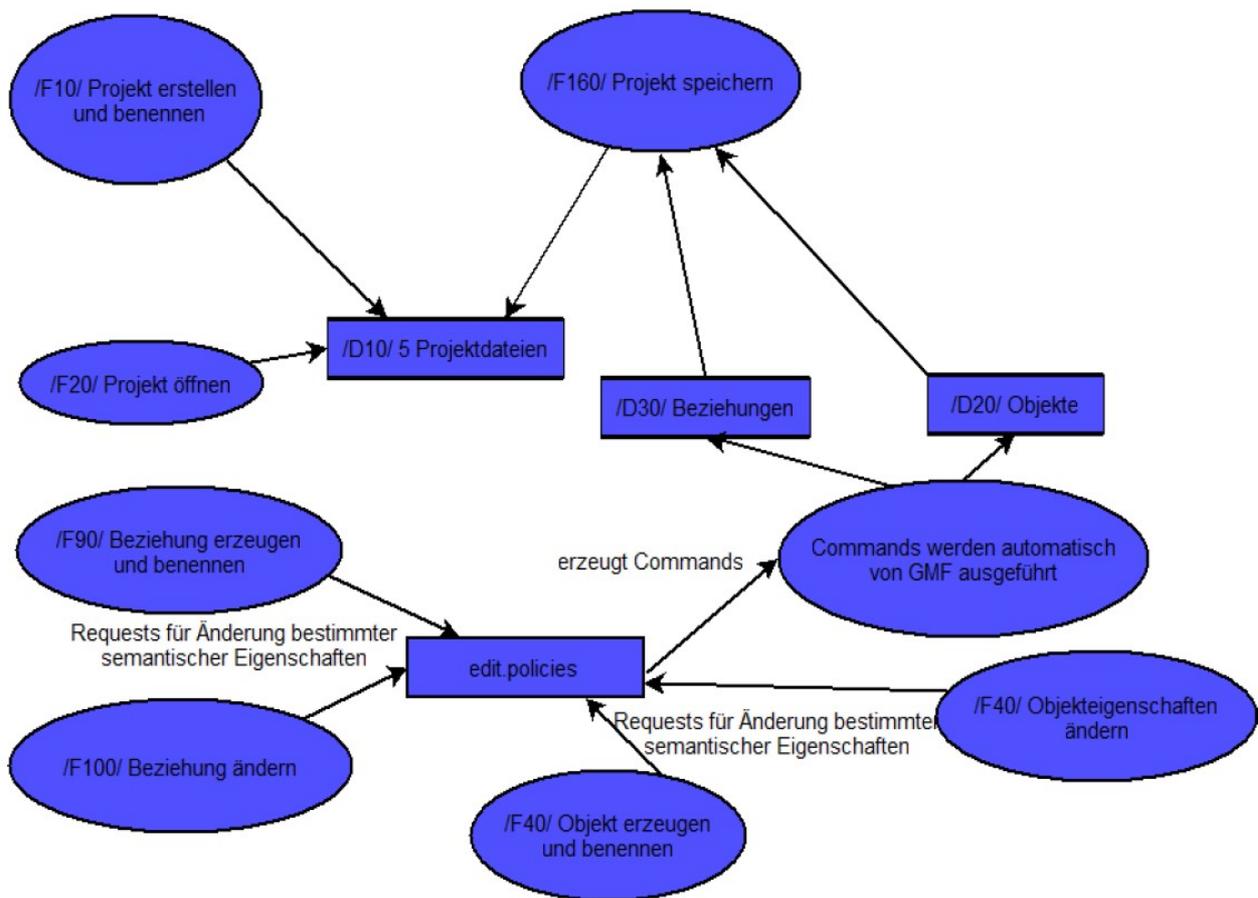
Hier werden die Elemente erzeugt: Die edit.policies werden initialisiert und die zum grafischen Element gehörigen grafischen Formen werden erzeugt.

3.4. Abläufe im Programm

3.4.1. Datenflussdiagramm für die Speicherung von Elementen

Im ersten Datenflussdiagramm unten wird prinzipiell veranschaulicht, wie die Speicherung von Objekten und Beziehungen über die Schnittstellenklassen im Paket edit.policies vonstatten geht. Daraus wird auch ersichtlich, dass Befehle für die Aktualisierung des semantischen Modells aufgrund der Änderungen, die im Diagramm vorgenommen werden, am besten im Paket edit.policies aufgehoben sind.

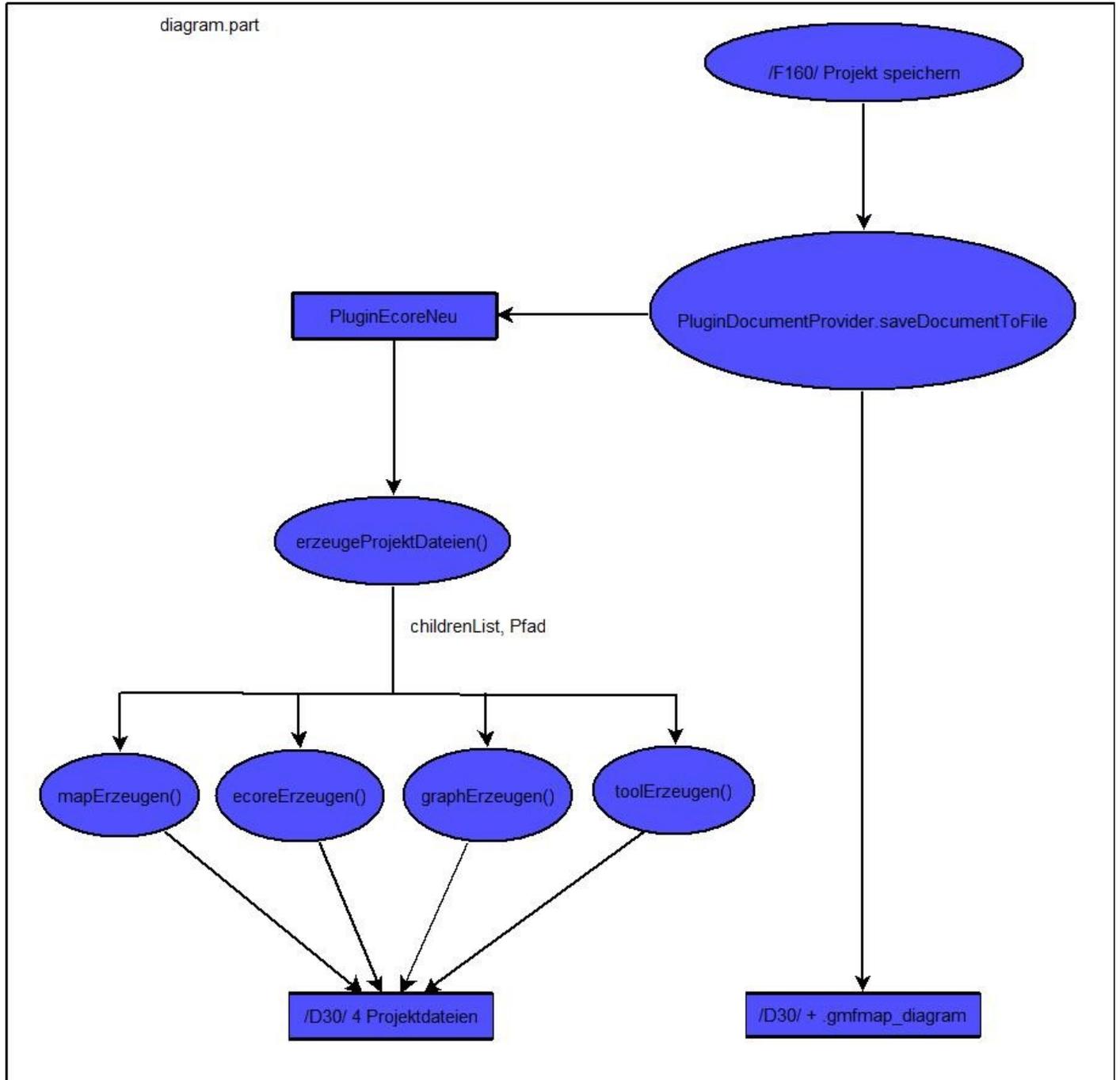
1. Datenflussdiagramm



3.4.2. Datenflussdiagramm für die Speicherung der Projektdateien

Beim Speichern des Projekts wird die Methode saveDocumentToFile in der Klasse PluginDocumentProvider aufgerufen. In dieser Methode erfolgt ein Aufruf der neu erstellten Klasse namens PluginEcoreNeu.java. Die oben genannte Klasse besteht aus fünf Methoden, eine zentrale "erzeugeProjektDateien()" und vier weitere die die .ecore, .gmfgraph, .gmftool, .gmfmap Dateien erzeugen.

"erzeugeProjektDateien()" ruft hintereinander die Methoden "ecoreErzeugen()", "graphErzeugen()", "toolErzeugen()", "mapErzeugen()" auf und übergibt ihnen die childrenList mit allen Nodes und Edges, die aus dem View des Editors extrahiert wurden.



3.4.3. Überblick über die erzeugten Dateien
 Hier sieht man die erzeugten Dateien zum Beispiel aus dem Screenshot.

