Masterarbeit

Implementierung von Anwendungskomponenten zur Unterstützung des System-Engineering-Prozesses im Bereich der Daten-Modellierung und Validierung durch Verwendung geeigneter Technologien und Modelle

Beyreuther, Björn

geboren am 21. April 1982 in Zwickau

Studiengang Informatik

Westsächsische Hochschule Zwickau
Fachbereich Physikalische Technik / Informatik
Fachgruppe Informatik

Betreuer, Einrichtung: Prof. Dr. Georg Beier, WHZ Zwickau
Prof. Dr. Frank Grimm, WHZ Zwickau
Abgabetermin: 10. Juni 2014

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

Modelle sind schon seit Jahrzehnten ein Mittel, um komplexe Systeme in vereinfachter Form zu beschreiben. Heutzutage werden für die Erstellung solcher Modelle meist computergestützte Hilfsmittel verwendet. Ziel dieser Arbeit ist die Integration von Softwarekomponenten zum verbesserten Management solcher Modelle über die reinen Grenzen des Modells hinweg. Die Integration erfolgt auf Basis der von ScopeSET ([Sco14]) entwickelten Architektur, an der ich während meiner beruflichen Tätigkeit mitgewirkt habe. Ausgangspunkt ist der Daten Modell Editor, der bereits die Möglichkeit bietet, Datenmodelle zu definieren und zu verwalten. Durch die Integration von Softwarekomponenten für das Anforderungsmanagement und die Validierung wird der Prozess zur Modellbildung und Validierung zusätzlich unterstützt.

1.2 System Engineering

1.2.1 Definition

Betrachten wir zunächst die Begriffe System und Engineering im Einzelnen. Bei einem System handelt es sich um eine Menge von Elementen, die miteinander interagieren und dabei eine bestimmte Funktion erfüllen. Man kann zwischen vielen unterschiedlichen Systemen in Abhängigkeit der fachlichen Disziplin unterscheiden. So ist für den Astronomen das Sonnensystem, für den Mediziner das Kapillarsystem und für den

Softwareentwickler das Software-System von Interesse.

Bei Engineering handelt es sich primär um eine anwendungsorientierte Wissenschaft, die sich nicht nur mit der Erforschung von Systemen beschäftigt, sondern auch Techniken, Verfahren und Methoden erforscht, um neue Systeme zu realisieren. Dabei kann man verschiedene Disziplinen unterscheiden, wie zum Beispiel:

- Elektrotechnik
- Maschinenbau
- Softwaretechnik (Software-Engineering)

System-Engineering beschäftigt sich mit Techniken, Verfahren, Methoden und Realisierungen in Bezug auf eine Menge von Elementen, die in einem logischen Zusammenhang stehen.

1.2.2 Die Rolle von Modellen

Systeme werden oftmals in Form von Modellen abgebildet. Das heißt ausgehend vom realen System wird in einer abstrakteren Form eine visuelle oder textuelle Repräsentation angefertigt. Die Notation ist dabei so vielfältig wie die Anzahl der Disziplinen. Die Realisierung kann papier- (siehe Abbildung 1.1) oder computergestützt erfolgen.

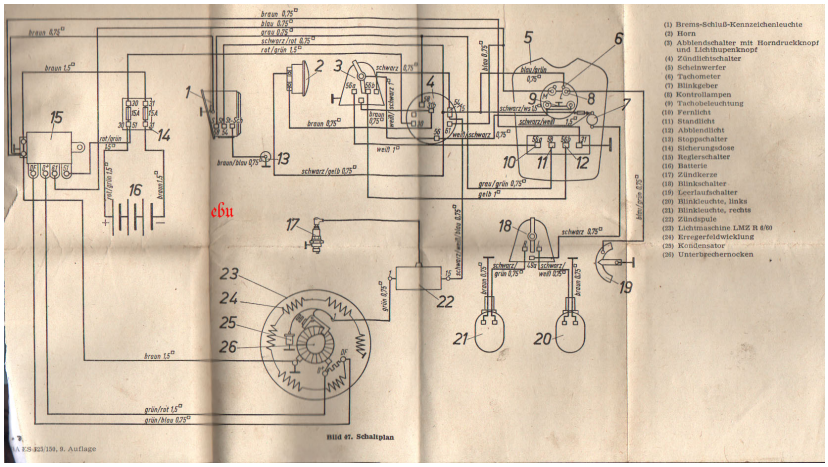

Abbildung 1.1: Schaltplan Motorrad MZ [Ost14]

Heutzutage werden meist computerbasierte Modelle verwendet, da diese gegenüber papierbasierten Modellen leichter zu handhaben sind. Besonders in Bezug auf Aktua-

lisierungen, Sichten auf Subsystem und Validität sind computergestützte Modelle im Vorteil.

Für die Beschreibung von computergestützten Modellen stehen unterschiedliche Anwendungen und Notationen zur Verfügung, die je nach Anwendungsgebiet stark variieren können. Die Abbildung 1.2 zeigt zum Beispiel ein Modell des Antriebsstrangs eines Elektrofahrzeugs in Simulink (Bestandteil von MATLAB).

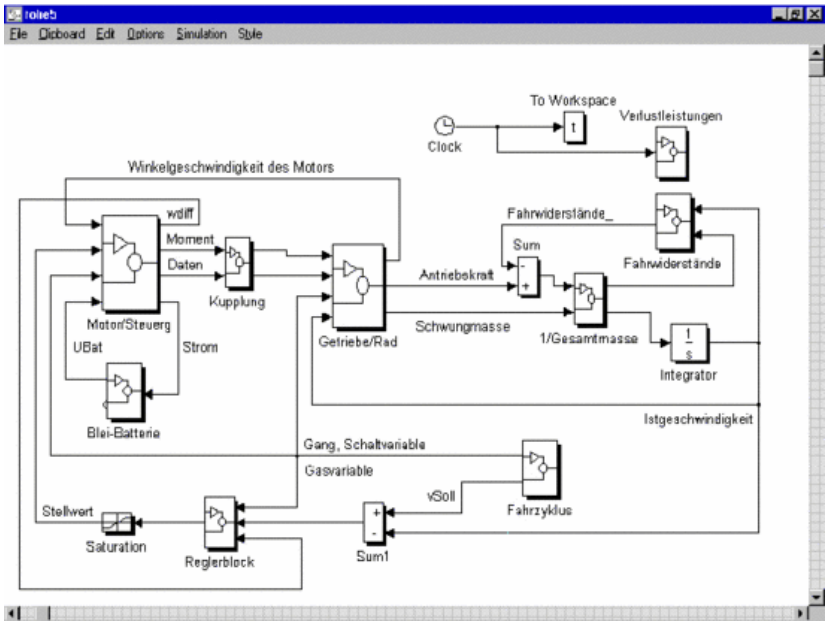

Abbildung 1.2: Modell des Antriebsstrangs eines Elektrofahrzeugs in MATLAB (Simulink) [Aac14]

Oftmals wird für die Beschreibung UML (Unified Modeling Language [OMG14]) verwendet. Dabei handelt es sich um eine Sprachdefinition, die unterschiedliche Aspekte von Systemen beschreiben kann. Unterschieden wird dabei in strukturelle Modelle und Modelle, die das Verhalten beschreiben. Zu den strukturellen Modellen zählen zum Beispiel Klassen- und Komponentenmodelle. Wohingegen das Verhalten in Form von Aktivitätsmodellen und Zustandsmodellen beschrieben wird.

1.3 Abgrenzung der Arbeit

In dieser Arbeit geht es primär um Aspekte der Luft- und Raumfahrt in Verbindung mit der EGS-CC Initiative (siehe [Age14a]). Konkret soll in Projektphase B (siehe [Age14c]) die Validierung des Datenmodells unterstützt werden. Dabei werden die An-

forderungen, das Datenmodell und die Testfälle in einer integrierten Umgebung, die hier vorgestellt wird, temporär verwaltet. Dies ermöglicht eine lückenlose Nachverfolgung zwischen den einzelnen Aspekten. In Kapitel 3 werden die grundlegenden Komponenten erläutert und kurz auf deren Implementierung eingegangen. Im Kapitel 4 wird die Erfassung und das Management von Anforderungen beschrieben sowie die Integration in die Anwendung dargestellt. Am Ende der Arbeit, in Kapitel 5, wird der Validierungsprozess und die Integration in der Anwendung erklärt.

1.4 Schreibkonventionen

Um besondere Begriffe oder implementierungsspezifische Aspekte hervorzuheben, wird *kursiver* Text verwendet.

KAPITEL 2

Eclipse-Plattform und Technologien

Dieses Kapitel befasst sich mit den für diese Arbeit wichtigen Aspekten von Eclipse und zusätzlichen Komponenten. Eclipse dient nicht nur als Entwicklungsumgebung, sondern auch als Plattform für die Applikationen, die im Rahmen dieser Arbeit entstanden sind. Für die Entwicklung von Anwendungen stellt Eclipse das Konzept der Rich Client Plattform (RCP) bereit, welches eine minimale Sammlung von Komponenten beinhaltet, die für eine Anwendung notwendig sind.

Ähnlich wie NetBeans oder IntelliJ handelt es sich bei Eclipse um eine integrierte Entwicklungsumgebung. Je nach Konfiguration kann sie zur Entwicklung von Anwendungen in unterschiedlichen Programmiersprachen verwendet werden (Java, C++). Seit der Version 3.0 basiert Eclipse auf dem Equinox Framework, welches im nächsten Kapitel kurz erläutert wird.

2.1 Komponentensytem - Equinox Framework

Die Verwaltung einzelner Komponenten in Eclipse erfolgt mit Hilfe des Equinox–Frameworks, welches eine Referenzimplementierung des OSGi (Release 4 [All14]) Standards darstellt. Die Abbildung 2.1 zeigt eine typische OSGi-Architektur.

Die Komponenten werden in der Abbildung als Bundle bezeichnet und durch das OSGi-Framework verwaltet. Dabei spielen das Laden, Aktualisieren und Beenden von Komponenten eine entscheidende Rolle.

Im Eclipse-Umfeld werden Komponenten meist Plug-ins genannt. Eine Sammlung von Plug-ins bezeichnet man als Feature. Dabei kapseln Plug-ins oder Feature bestimmte

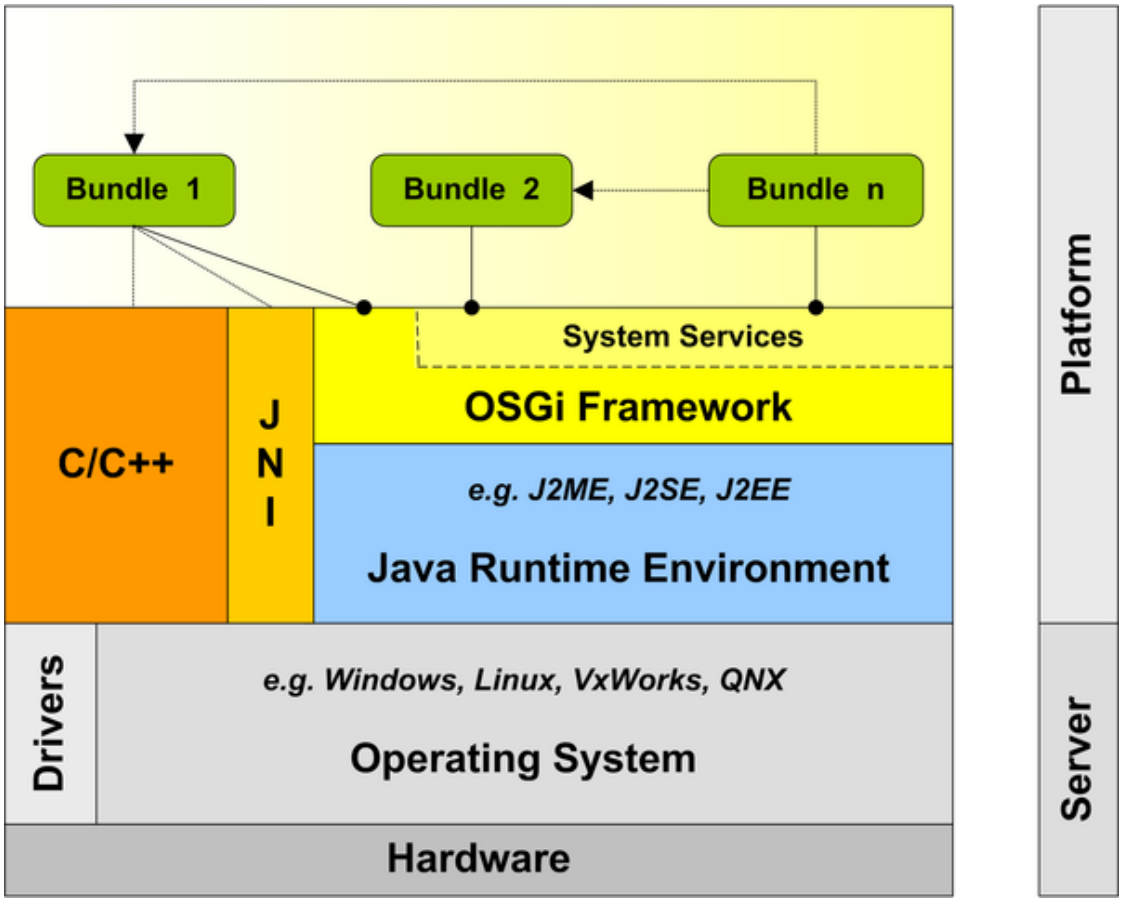

Abbildung 2.1: OSGi-Architektur [Wik14c]

Funktionalitäten, die in der Anwendung verfügbar sein sollen. Der Zugriff oder die Bereitstellung von Funktionalität erfolgt durch *Extension Points*. Ein *Extension Point* dient als eindeutige Schnittstelle, die von anderen Plug-ins genutzt werden kann.

Für ein Plug-in sind folgende Dateien von entscheidender Bedeutung:

- MANIFEST.MF – definiert Abhängigkeiten und Laufzeitinformationen für das Plug-in
- plugin.xml – enthält Angaben zu definierten und genutzten *Extension Points*

Neben diesen beiden XML basierten Konfigurationsdateien ist eine Klasse, die das Interface *BundleActivator* implementiert notwendig. Diese Klasse wird vom Equinox-Framework zum Anmelden (*start()* Methode) und zum Abmelden (*stop()* Methode) des Plug-ins verwendet.

2.2 Graphische Oberfläche

2.2.1 Standard Widget Toolkit (SWT) und JFace

Das Standard Widget Toolkit (SWT) ist eine auf Java basierende Bibliothek, welche Funktionalität bereitstellt, um graphische Oberflächen zu erstellen. SWT ist Bestandteil der Eclipse-Plattform und nutzt die nativen GUI-Bibliotheken (GUI - Graphical User Interface) des Betriebssystems (Windows, Linux, Mac OS). Vergleichbare Bibliotheken sind zum Beispiel Abstract Window Toolkit (AWT), Swing oder JavaFX. Ein entscheidender Unterschied zwischen den Graphik-Bibliotheken ist, ob die Komponenten durch das Betriebssystem oder in Java selbst gerendert werden. Bei SWT spricht man von schwergewichtigen Komponenten, da sie direkt auf Betriebssystemkomponenten aufsetzen und auch das Rendering durch das Betriebssystem erfolgt. In Swing spricht man von leichtgewichtigen Komponenten, da diese eigenständig in Java gerendert werden.

SWT bietet eine Vielzahl sogenannter Widgets. Zum Beispiel Buttons, Textfelder, Tabellen und die Möglichkeit neue Widgets zu definieren. JFace, welches ebenfalls Bestandteil der Eclipse-Plattform ist, erweitert die SWT-Basiskomponenten durch weitere komplexere GUI-Elemente. Außerdem führt JFace eine zusätzliche Abstraktionsschicht für dedizierte SWT-Komponenten ein. GUI-Elemente, die durch JFace definiert werden sind zum Beispiel: *EditorPart*, *ViewPart*, *Wizard*, *Dialog*. Die zusätzliche Abstraktionsschicht wird im Bereich der Tabellen deutlich. JFace stellt hier das Konzept eines *Viewers* bereit, durch den es möglich wird, eine Implementierung auf Basis des Modell View Controller (MVC) Musters zu realisieren.

2.2.2 Graphical Modeling Framework (GMF) und Zest

Das Graphical Modeling Framework (GMF) ist Bestandteil der Eclipse-Plattform und dient zum Entwickeln von graphischen Editoren. Es lässt sich in zwei Komponenten unterteilen. Die erste Komponente dient zum Generieren eines graphischen Editors auf Basis des Eclipse Modeling Framework (siehe Kapitel 2.3). Die zweite Komponente stellt die Laufzeitumgebung zur Verfügung, die notwendig ist für den graphischen Editor. Die Abbildung 2.2 zeigt ein Beispiel für einen graphischen Editor auf Basis von GMF:

Neben EMF basiert GMF auf dem Graphical Modeling Framework (GEF), welches

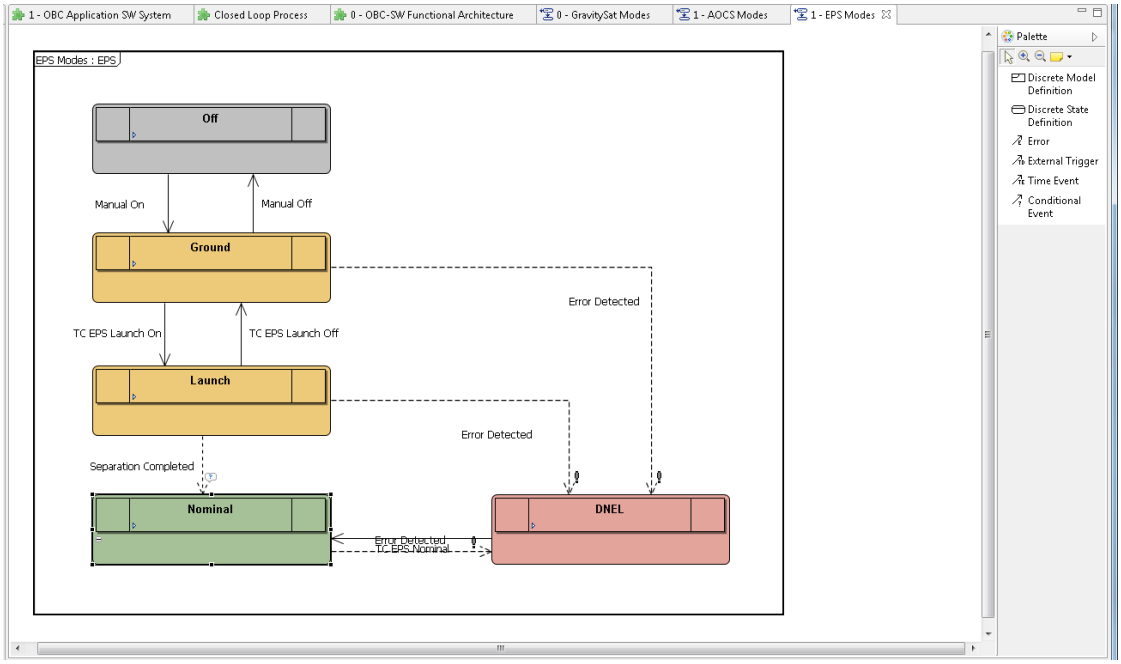

Abbildung 2.2: GMF Editor

wiederum auf Draw2d basiert. Draw2d setzt auf SWT auf und bietet die Möglichkeit graphische Elemente in einem SWT-Canvas zu zeichnen. GMF und GEF abstrahieren von den primitiven Zeichenmethoden von Draw2d und bieten die Möglichkeit Daten eines EMF-Modells zu visualisieren und zu editieren. Dabei verwenden GEF und auch GMF das Modell View Controller (MVC) Muster, um eine Entkopplung zwischen Daten, Visualisierung und Geschäftslogik zu erreichen.

Neben GMF wird für die in dieser Arbeit relevanten Anwendungen auch Zest verwendet. Dabei handelt es sich um ein Framework, welches direkt auf Draw2D basiert und im Gegensatz zu GMF sehr leichtgewichtig ist, da es primär auf die Visualisierung ausgelegt ist.

2.3 EMF - Eclipse Modeling Framework

Das Eclipse Modeling Framework ist eine Sammlung von verschiedenen Funktionen, welche zur Erstellung von Modellen und zur Entwicklung von Anwendungen für diese Modelle dient. EMF basiert auf der Meta Object Facility (MOF) Spezifikation, hat sich jedoch in den letzten Jahren weiterentwickelt ([Ecl14a]). Für die Definition der Modelle wird in EMF das Metamodell Ecore verwendet, welches Ähnlichkeiten zu den

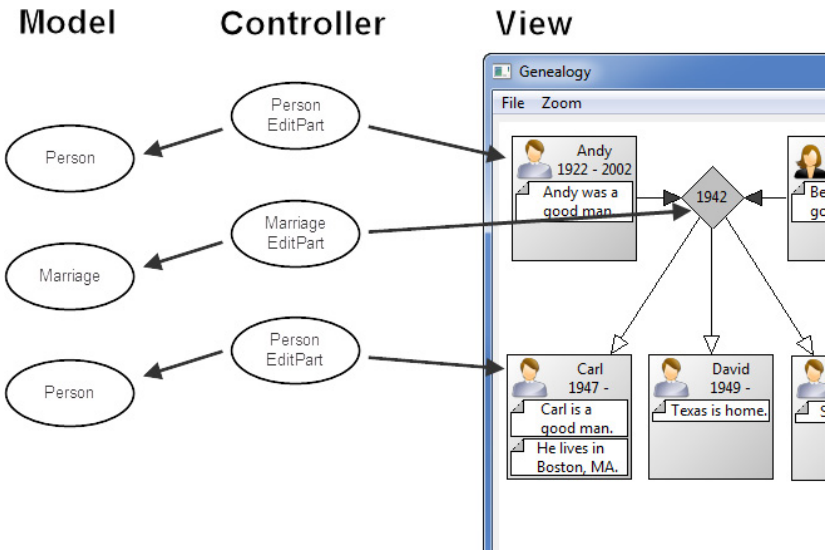

Abbildung 2.3: MVC-Muster des GMF-Editors [Cla11]

Elementen des UML (Unified Modeling Language) Klassenmodells aufweist.

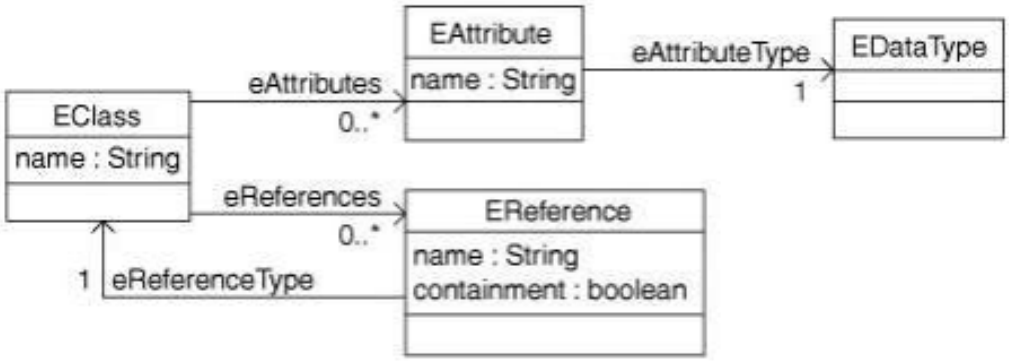

Abbildung 2.4: Vereinfachtes Ecore-Metamodell ([Mer08] Abbildung 2.3)

Um ein Ecore basiertes Modell (EMF-Modell) zu definieren, bietet EMF vier Möglichkeiten:

- Importieren einer XML-Schema-Definition (XSD)
- Import eines UML-Modells
- Import auf Basis von Java-Interfaces
- Definition innerhalb der EMF-Editoren für Ecore

Das EMF-Modell kann für die Entwicklung einer Anwendung genutzt werden. Dazu bietet EMF Codegeneratoren an, die auf Java Emitter Templates (JET) basieren. Bei der Codegenerierung kann man zwischen drei Ebenen, die sich am MVC-Muster orientieren, unterscheiden.

- Java Repräsentation des Modells (*Model-Plug-in*)
- Hilfsklassen zum Editieren und Visualisieren der instanziierten Modell-Elemente (*Edit-Plug-in*)
- Beispielimplementierung eines Editors zum Editieren und Instanziieren der Modell–Elemente (*Editor-Plug-in*)

In der Regel werden dazu drei Plug-ins erzeugt, welche die generierten Java Klassen beinhalten. Befinden sich in den Plug-ins bereits generierte Klassen, werden diese beim erneuten Generieren mit Hilfe von *JMerge* aktualisiert. Dies ermöglicht es, die generierten Klassen für bestimmte Anforderungen anzupassen.

Diese drei Plug-ins stellen die minimale Funktionalität für eine Anwendung bereit, die es ermöglicht ein EMF-Modell zu instanziieren. In der Regel wird jedoch die Basisimplementierung des Editors durch einen oder mehrere Editoren ersetzt, da der generierte Editor für einen produktiven Einsatz ungeeignet ist.

Weitere wichtige Funktionen von EMF sind:

- Meta-Information über das EMF-Modell zur Laufzeit (in der erstellten Anwendung) ([Mer08] Kapitel 14)
- Speichern und Laden von Modellen ([Mer08] Kapitel 15)
- Validierung von Modellen ([Mer08] Kapitel 18)

Gerade die Meta-Informationen spielen eine entscheidende Rolle bei der Entwicklung von wiederverwendbaren Komponenten, wie sie in Kapitel 3 erläutert werden. Zusätzlich ist auch das Laden und Speichern von entscheidender Bedeutung, da EMF die Möglichkeit bietet, vom einfachen XML-Parser zu abstrahieren. So ist es nicht notwendig einen eigenen XML-Parser zu definieren, sondern man kann die von EMF vordefinierten Hilfsklassen verwenden, die auf Basis von Namensraumdefinitionen XML basierte Dateien laden und speichern können. Beim Laden einer solchen XML Datei wird der

Namensraum für die entsprechenden XML Tags mit Hilfe der *EMF EPackage Registry* aufgelöst, um die Modelldefinition zu ermitteln. Nachdem diese gefunden wurde, wird das Instanz-Modell entsprechend seiner Definition und dem Inhalt des XML-Tags befüllt.

2.4 OCL - Object Constraint Language

Bei der Object Constraint Language (OCL) handelt es sich um eine deklarative textuelle Sprache zur Definition von Bedingungen und Fakten. OCL wird in Verbindung mit anderen Sprachen wie zum Beispiel UML, Ecore oder QVT (Query View Transformation) verwendet, um Vor- und Nachbedingungen, Integritätsbedingungen oder Werte für Eigenschaften festzulegen.

OCL-Ausdrücke selbst haben keine Seiteneffekte, da beim Auswerten keine Änderungen am Modell stattfinden können. Durch das Auswerten des OCL-Ausdruckes wird immer ein Wert zurück geliefert. Dabei handelt es sich um ein einzelnes typisiertes Objekt oder eine Menge von typisierten Objekten. Bei den Typen kann es sich um OCL-Standard-Typen handeln oder um Typen aus dem jeweiligen Modell, auf welches sich der OCL-Ausdruck bezieht.

Eclipse bietet für OCL zwei Implementierungen an, die es ermöglichen OCL-Ausdrücke für UML- oder Ecore-Modelle zu definieren ([Ecl14c]). Die erste Implementierung basiert auf einen LPG-Parser mit manuell definierter Grammatik und Syntaxhilfe. Die zweite Implementierung basiert auf Xtext ([Ecl14d]) und einer zugrundeliegenden ANTLR-Grammatikdefinition.

Für das Parsen und Auswerten eines OCL-Ausdrucks stellt Eclipse eine Reihe von Hilfsklassen zur Verfügung, die in der Eclipse-Dokumentation ([Ecl14c]) erläutert sind.

KAPITEL 3

System Architektur

3.1 Kontext

Für die Arbeit sind zwei RCP-Anwendungen von Bedeutung. Die erste Anwendung ist der Daten Modell Editor (DME) und die zweite Anwendung ist der Instanz Modell Editor (IME). Der Daten Modell Editor wird dazu verwendet die Anforderungen, das Datenmodell (EMF-Model) und die Testfälle zu verwalten. Der Instanz Modell Editor bietet die Möglichkeit, dass im Daten Modell Editor definierte Datenmodell zu instanziieren und die im Daten Modell Editor definierten Testfälle (siehe Kapitel 5) durchzuführen.

Die Abbildung 3.1 zeigt die primären Eclipse-Plattform-Komponenten, die in den Anwendungen zum Einsatz kommen (siehe Kapitel 2).

Neben diesen existieren weitere Komponenten, die im Rahmen dieser Arbeit von Bedeutung sind und an denen ich während meiner beruflichen Tätigkeit bei der Firma ScopeSET mitgewirkt habe. In der Abbildung 3.2 sind vier Hauptbestandteile der Anwendung zu sehen, die in den folgenden Kapiteln erläutert werden.

Weitere Funktionen werden nur kurz bei der Erläuterung der Modelle in Kapitel 3.2 erwähnt.

3.1.1 Resource Manager

Der Resource Manager und die zugehörigen Komponenten dienen der Verwaltung von EMF basierten Modellen und zusätzlicher Dateien. Diese werden in Form ei-

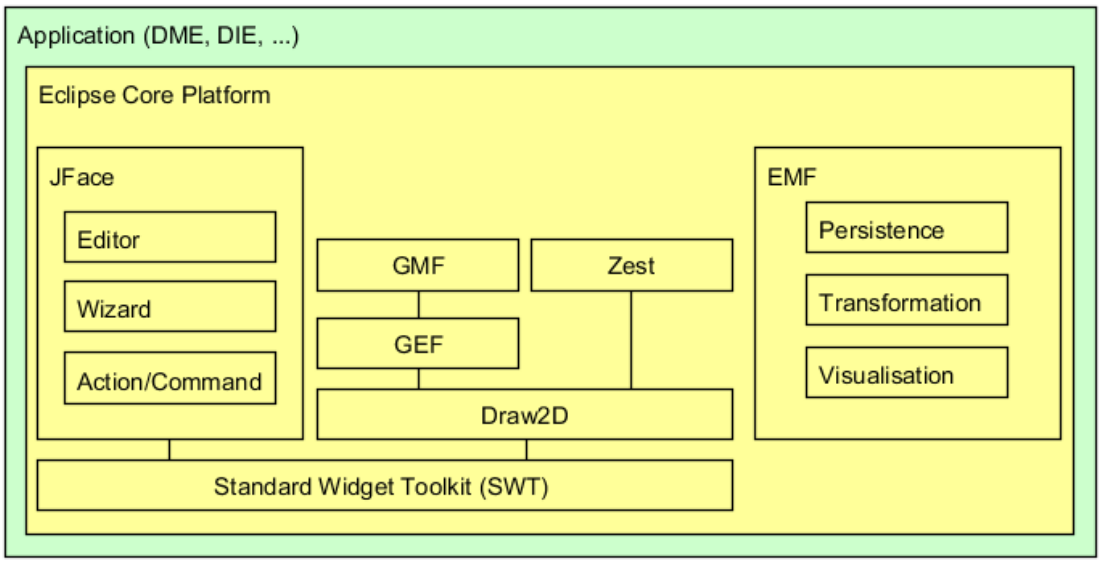

Abbildung 3.1: Eclipse basierte Hauptkomponenten der RCP-Anwendung

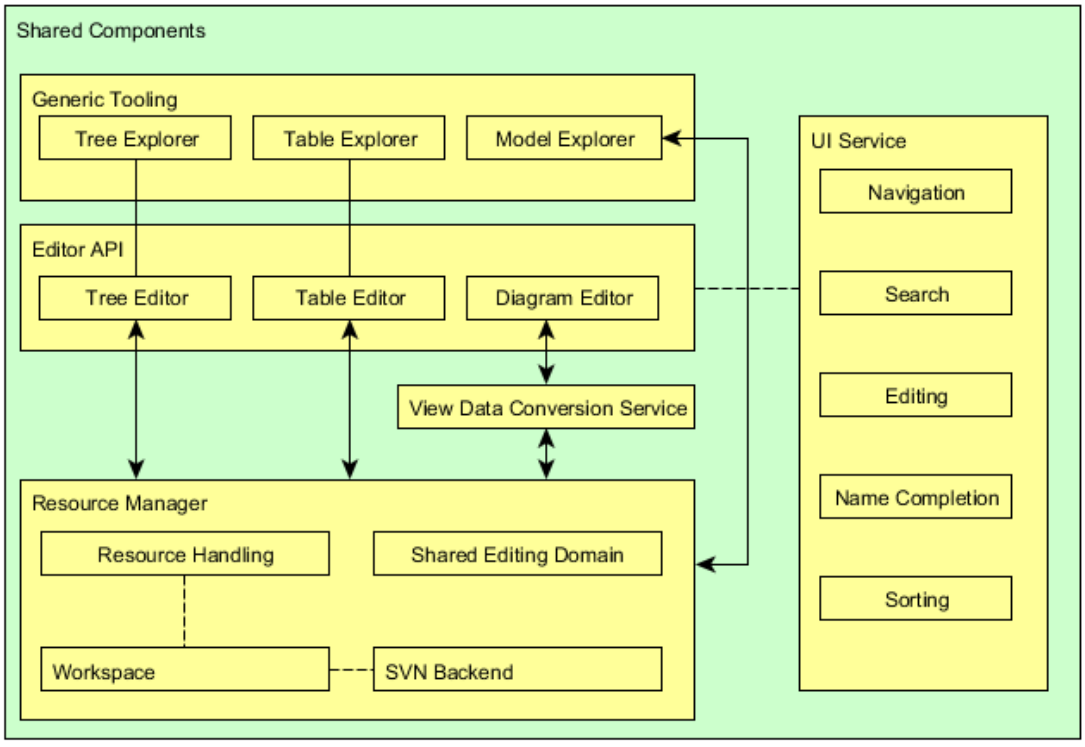

Abbildung 3.2: API-Komponenten (entwickelt von ScopeSET)

nes Eclipse-Projekts (*IProject*) gruppiert und befinden sich innerhalb des Eclipse-Workspaces (*IWorkspace*). Der Inhalt des Workspaces wird im Daten Modell Editor

und im Instanz Modell Editor mit einer erweiterten Version des Navigators dargestellt, welcher auch in der Standard Eclipse IDE verfügbar ist. Sowohl der Daten Modell Editor als auch der Instanz Modell Editor bieten die Möglichkeit dedizierte Projekte innerhalb des Workspaces anzulegen. Der Unterschied zwischen den Projekten ist das jeweilige EMF basierte Modell. Das Laden und Speichern sowie das Änderungsmanagement dieser Modelle wird vom Resource Manager ermöglicht. Alle Änderungen, die am Modell vorgenommen werden, müssen über eine sogenannte *EditingDomain* (siehe [Mer08] Kapitel 3.3) erfolgen. Ähnlich wie bei Datenbanktransaktionen können EMF basierte Modelle mit Hilfe von *EMF Commands* geändert werden. Dies stellt sicher das Änderungen immer zu konsistenten Modellen führen, da sie nur ganz oder gar nicht ausgeführt werden. Zusätzlich wird die *EditingDomain* zwischen allen Beteiligten, die das Modell ändern möchten, geteilt. Das ermöglicht es, den Zugriff auf das Modell zu synchronisieren und notwendige Updates für GUI-Komponenten zu veranlassen. Dies ist zum Beispiel dann wichtig, wenn ein Modell in unterschiedlichen Editoren editiert wird.

Zum Laden und Speichern der EMF basierten Modelle wird das von EMF bereitgestellte Konzept der *Resource* verwendet. Eine *Resource* ist eine Abstraktion von der physischen Datenebene zum Beispiel einer XML(Extensible Markup Language)-Datei. Ressourcen werden in einem sogenannten *ResourceSet* zusammengefasst, welches auch genutzt werden kann, um eine *Resource* zu erzeugen. Die aktuelle Implementierung verwendet XMI (XML Metadata Interchange) basierte Dateien unter Gebrauch dedizierter Dateiendungen. EMF bietet die Möglichkeit sogenannte *ResourceFactories* für bestimmte Dateiendungen zu registrieren. Wird eine *Resource* auf Basis einer Datei erzeugt, kann anhand der registrierten Factories festgestellt werden, welche konkrete Implementierung der *Resource* verwendet werden soll, um die Datei zu deserialisieren. Wichtig ist dabei, dass die Modelldefinitionen verfügbar sind, das heißt das der Parser der die XMI-Datei lädt alle Namensraumdefinitionen auflösen kann (siehe Kapitel 2.3).

3.1.2 UI Services und Editor API

Für eine Vielzahl von Problemen stellen diese Komponenten bereits Implementierungen zur Verfügung. Dabei ist es wichtig zu erwähnen, dass die Implementierungen stark modellgetrieben sind und teilweise die Meta-Informationen, die in EMF-Modellen bereit gestellt werden, nutzen. Es existieren drei Basisimplementierungen für Eclipse-Editoren: ein Tree Editor, ein Table Editor und ein Diagram Editor. Diese beinhalten erstens Implementierungen für die Synchronisation bei Modelländerung unter Verwen-

dung des Resource Managers, und zweitens eine adaptive Implementierung, die den Inhalt der Editoren beschreibt. Für die Entwicklung eines spezifischen Editors ist es notwendig, einen abgeleiteten Editor zu erstellen. Der Inhalt des Editors, zum Beispiel die Spalten und die anzuzeigenden Elemente werden nicht direkt im Editor beschrieben, sondern in einem eigenständigen Modell. Dieses Modell wird als Notation-Modell bezeichnet. Es dient neben anderen Artefakten als Input für die Codegenerierung während des Entwicklungsprozesses (siehe Kapitel 3.5). Die Abbildung 3.3 zeigt einen Ausschnitt aus diesem Modell.

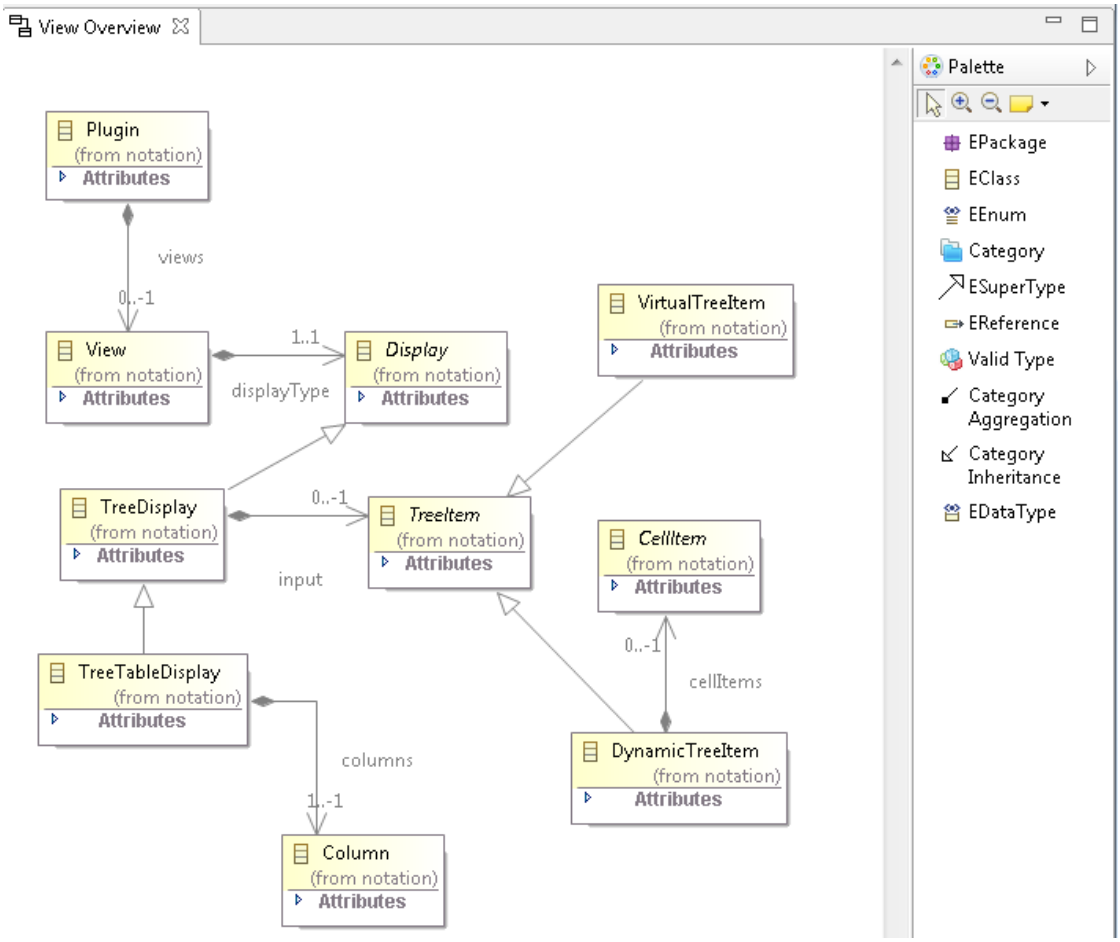

Abbildung 3.3: Notation-Modell

Basierend auf der Instanziierung dieses Modells wird Java-Code generiert, welcher von den Editoren verarbeitet wird und dadurch das Aussehen und den Inhalt beeinflusst.

Neben den Editoren stehen GUI-Elemente und Funktionen zur Verfügung, die für die Entwicklung hilfreich sind. Diese reichen von Wizards zum Selektieren von instanzierten Modellelementen für eine entsprechende Beziehung (*EReference*), über Filter und

Suchmasken bis hin zu umfangreichen Implementierungen im Bereich von OCL und der Navigation zwischen Editoren. So ist es zum Beispiel möglich für ein ausgewähltes Element, zwischen Editoren zu navigieren. Vorausgesetzt diese Editoren sind bei einem entsprechenden *Extension Point* registriert.

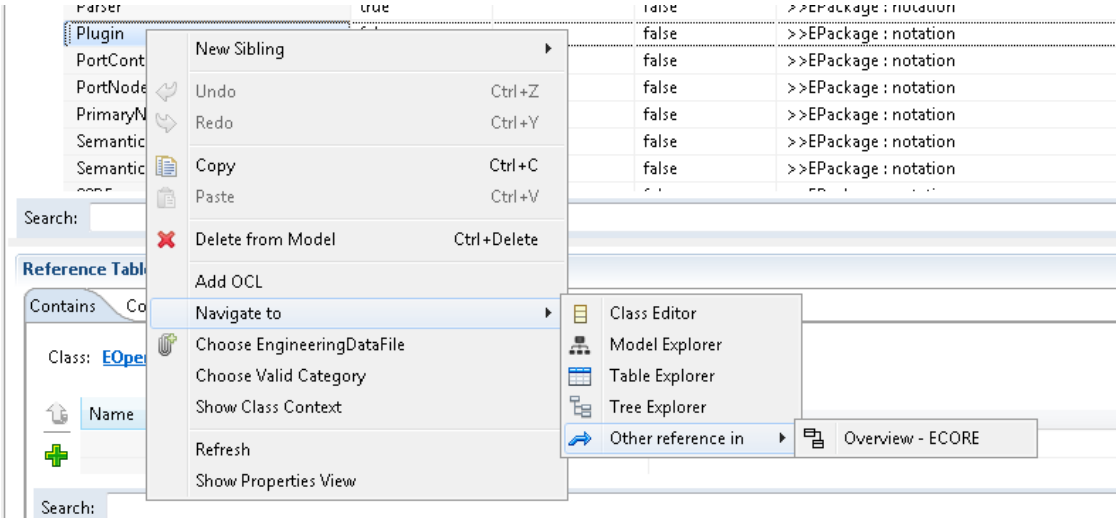

Abbildung 3.4: Navigations-Menü mit möglichen Zieleditoren

Ziel bei der Implementierung dieser Komponenten war ein hoher Grad der Wiederverwendbarkeit. Deshalb greifen sie auf Meta-Informationen aus den EMF-Modell zurück, welche über die jeweilig instanzierten Elemente (*EObject*) verfügbar sind. Des Weiteren existieren Schnittstellen, die von den jeweiligen Klassen im EMF-Modell implementiert werden müssen. Diese dienen zum einen, um gewisse Kernfunktionalitäten zu sichern und zum anderen, um Funktionalität bei Bedarf verfügbar zu machen. Die Schnittstellen und deren Bedeutung werden in Kapitel 3.2 näher erläutert.

3.1.3 Modellunabhängige Editoren

Neben den bereits erwähnten Funktionen stehen im Daten Modell Editor und auch im Instanz Modell Editor drei voll funktionsfähige Editoren zur Verfügung. Diese Editoren sind komplett unabhängig vom zugrundeliegenden Instanz-Modell beziehungsweise dessen Definition (EMF-Modell). Alle drei Editoren bieten zur Laufzeit generische Funktionalität für das entsprechende Instanz-Modell an, wie zum Beispiel das Erzeugen und Löschen von Objekten. Der erste der drei Editoren ist der sogenannte *Table Explorer*, der eine tabellenorientierte Sicht auf die instanziierten Elemente bietet. Die folgende Abbildung zeigt den *Table Explorer* im Instanz Modell Editor:

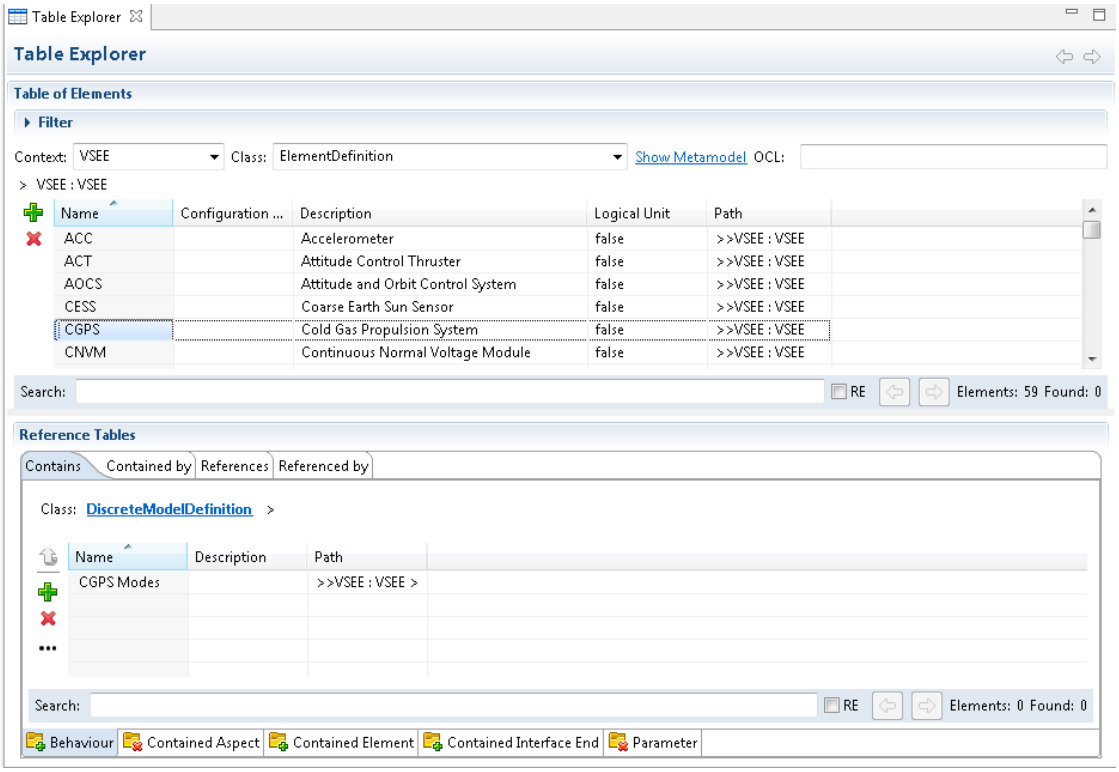

Abbildung 3.5: *Table Explorer* im Kontext des Instanz Modell Editors

Wie man in Abbildung 3.5 erkennen kann, besteht der Editor aus zwei größeren Bereichen. Im oberen Bereich beziehungsweise in der oberen Tabelle werden alle Instanzen einer bestimmten Klasse angezeigt. Die Klasse kann über ein Dropdown-Liste oberhalb der Tabelle gewählt werden. Welche Klassen ausgewählt werden können, hängt dabei vom jeweiligen EMF-Modell ab, das als Grundlage für die instanziierten Elemente verwendet wird. Im konkreten Fall handelt es sich dabei um das EGS-CC Datenmodell, welches in den folgenden Kapiteln noch eine Rolle spielen wird. Der untere Bereich des Editors ist komplett davon abhängig, welche Klasse und welche Instanz im oberen Bereich gewählt wird. Anhand der Klasse werden ausgehende und eingehende Beziehungen ermittelt und in den vier Tabs 'Contained', 'Reference', 'Contained By' und 'Referenced By' gruppiert. Die Tabs 'Contained By' und 'Referenced By' stellen dabei die Inverse der direkten Beziehungen dar. Bei 'Contained' handelt es sich um eine spezielle Beziehung ähnlich der UML-Komposition, bei der die Existenz der referenzierten Elemente abhängig vom referenzierenden Element ist ([Mer08] Kapitel 5.2). Der Inhalt in den einzelnen untergeordneten Tabs, die abhängig von der jeweiligen Beziehung (*EReference*) sind, basiert auf dem selektierten Element in der oberen Tabelle und zeigt die konkreten referenzierten Elemente. Der *Table Explorer* bietet neben der

reinen Visualisierung auch die Möglichkeit neue Elemente anzulegen, zu editieren und Beziehungen zu bearbeiten.

Der zweite Editor ist der sogenannte *Tree Explorer*, der in der folgenden Abbildung dargestellt wird:

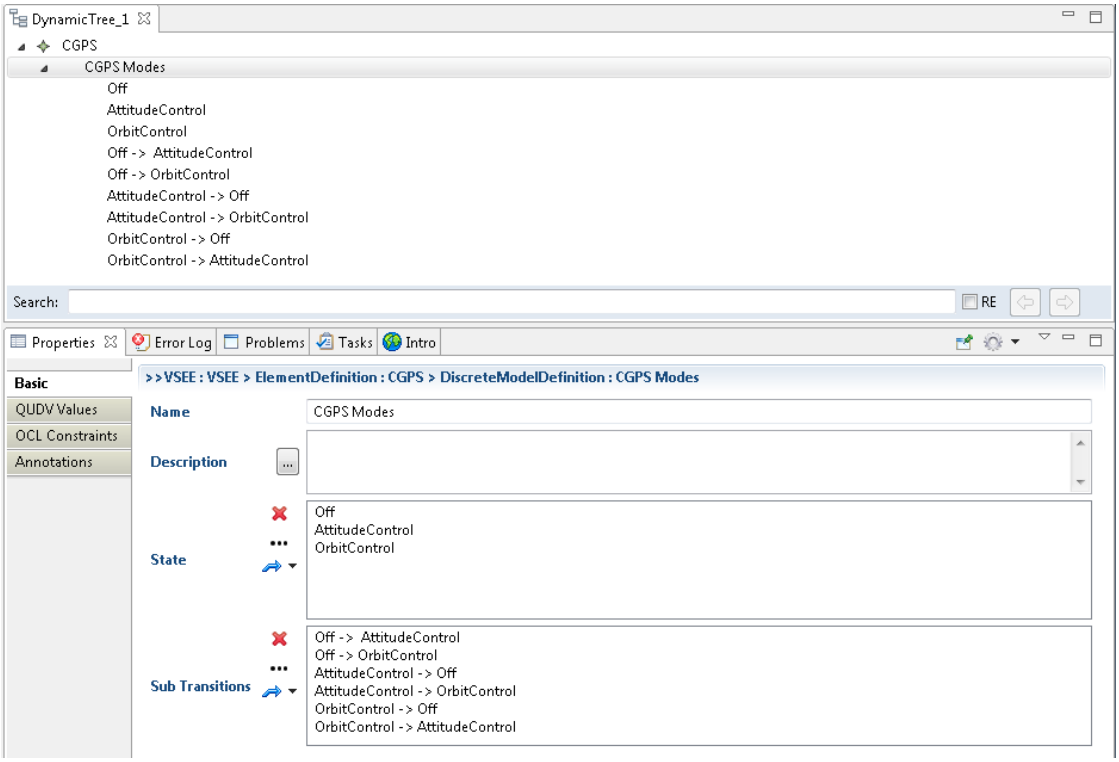

Abbildung 3.6: Tree Explorer mit Properties View

Der *Tree Explorer* kann auf Basis jedes Instanz-Elementes initialisiert werden, was wiederum als Wurzelknoten für den Baum dient. Ähnlich wie beim *Table Explorer* bietet der *Tree Explorer* die Möglichkeit Beziehungen darzustellen. Die Darstellung erfolgt dabei in Form eines Baums ausgehend vom Wurzelknoten-Element. Im Gegensatz zum *Table Explorer* ist dabei die Tiefe der Beziehungsebenen nicht auf eine Ebene begrenzt (Element -> referenziertes Element), sondern kann beliebig fortgesetzt werden. Diese Darstellungsform ist dann geeignet, wenn man die Beziehungen zwischen Elementen über mehrere Ebenen hinweg darstellen will, zum Beispiel einen 'Containment'-Baum aufbaut. Interessant ist, dass die Sicht auf die Daten persistierbar ist und zu einem späteren Zeitpunkt einfach geladen werden kann. Die persistierte Form einer solchen Sicht wird als *Dynamic Tree* bezeichnet.

Neben den zwei tabellen- und baumorientierten Editoren existiert noch ein graphischer Editor auf Basis des Zest-Frameworks, der im folgenden als *Modell Explorer* bezeichnet

wird. Dieser Editor besitzt keine konkrete Notation, wie man sie zum Beispiel von graphischen UML-Werkzeugen kennt. Der *Modell Explorer* unterscheidet nur zwischen Knoten und Kanten. Eine Kante stellt dabei immer eine Verbindung zwischen zwei Knoten dar. Die folgende Abbildung zeigt den *Modell Explorer* im Instanz Modell Editor.

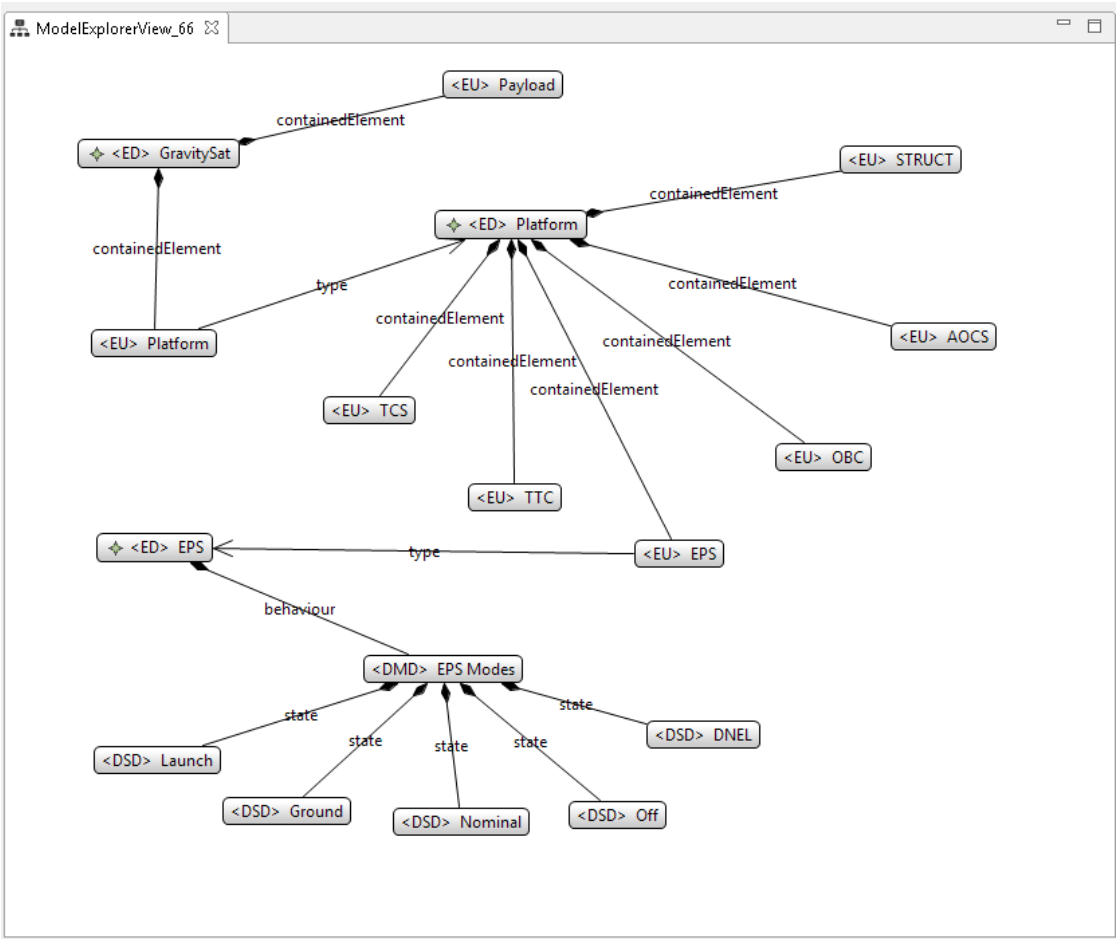

Abbildung 3.7: *Modell Explorer* im Kontext des Instanz Modell Editors

Gleich den vorhergehenden Editoren dient der *Modell Explorer* zur Visualisierung bestimmter Zusammenhänge im Modell und nutzt dabei die Meta-Informationen, die von EMF-Modellen bereitgestellt werden. Der *Modell Explorer* bietet sich, durch die graphische Repräsentation und die Möglichkeit des Exports von Bildern, als Basis für die Kommunikation mit unterschiedlichen Parteien an. Die entsprechenden Sichten im *Modell Explorer* können wie beim *Tree Explorer* persistiert werden, was es ermöglicht die Sichten zu jedem Zeitpunkt wiederherzustellen.

3.2 EMF Modelle und deren Funktion

Für die Basisfunktionalität und zum Vereinfachen der Integration neuer Funktionalität setzen der Daten Modell Editor und der Instanz Modell Editor intensiv Bestandteile von EMF ein. Es ist also nicht verwunderlich, dass eine Reihe von EMF-Modellen zum Einsatz kommen, die für diese Funktionen genutzt werden. In diesem Kapitel werden die eingesetzten EMF-Modelle und ihre Funktion kurz erläutert. Gruppiert werden diese in einem sogenannten Core-Modell, welches die dedizierten Sub-Modelle in Form von *EPackages* beinhaltet.

Neben den hier vorgestellten Modellen entstanden im Rahmen dieser Arbeit zwei weitere Modelle. Das erste ist das Anforderungs-Modell, welches im Kapitel 4 erläutert wird und das zweite ist das Validierungs-Modell, welches im Kapitel 5 vorgestellt wird.

Die Instanziierung der hier vorgestellten Modelle wird als Bestandteil der *Resource* im Daten Modell Editor und Instanz Modell Editor verwaltet und stellt somit in Verbindung mit den für den Editor spezifischen Daten eine Einheit dar. Im Daten Modell Editor handelt es sich bei den spezifischen Daten um die Pakete, Klassen, Attribute und Referenzen basierend auf der Modellierungssprache EcoreExt (siehe Kapitel 3.3.1). Im Instanz Modell Editor hingegen hängen die Daten vom jeweiligen Datenmodell ab.

3.2.1 Core-Modell

Wie erwähnt dient das Core-Modell zum Gruppieren einzelner Sub-Modelle, die in den folgenden Kapiteln erläutert werden. Konkret handelt es sich dabei um ein *EPackage*, welches die weiteren Modelle als Sub-*EPackages* beinhaltet. Das Core-Modell an sich definiert nur eine einzige *EClass* namens *ICommonElement*. Diese *EClass* dient als primäres Interface für alle Modelle, die im Kontext des Instanz Modell Editors und Daten Modell Editors verwendet werden. Konkret bedeutet das, dass alle Klassen innerhalb dieser Modelle von *ICommonElement* erben müssen. Dies geschieht entweder explizit beim Definieren der Modelle, wie es bei den Modellen die in den folgenden Kapiteln erläutert werden der Fall ist oder durch eine automatisierte Transformation. Die Transformation wird eingesetzt um Datenmodelle, die im Daten Modell Editor definiert wurden, für die Entwicklung des Instanz Modell Editors zu exportieren (siehe auch Kapitel 3.6).

Das Interface *ICommonElement* definiert folgende Eigenschaften:

Namens-Attribut (*nameValue*) - Dies stellt sicher, dass alle Elemente einen Namen besitzen und in der Benutzeroberfläche sinnvoll dargestellt werden können. So wird in einer Baum-Struktur oder innerhalb eines graphischen Elements eines GMF-Editors immer dieses Attribut genutzt, um ein Element zu beschreiben. Wie der Name definiert wird, ist dabei nicht von Bedeutung und kann sowohl automatisiert als auch vom Benutzer selbst vergeben sein.

UUID-(Universally Unique Identifier) Attribute (*unique_identifier*) - Der UUID wird in der Benutzeroberfläche nicht angezeigt und dient als technisches Hilfsmittel, um Elemente eindeutig zu identifizieren. Dies ermöglicht es, Objekte unabhängig von ihrem Namen und ihres Kontextes zu identifizieren, was zum Beispiel bei einem Vergleich von zwei Modellen von Bedeutung ist. So kann ermittelt werden, ob sich nur der Name oder der Kontext eines Elements verändert hat, oder ob dieses Element nicht mehr existiert beziehungsweise durch ein neues Element ersetzt wurde.

Delete-Attribute (*delete*) - Dieses Attribut ermöglicht es, Elemente als gelöscht zu markieren. So ist es möglich beim Löschen von Elementen zu definieren, dass sie vorläufig als gelöscht markiert werden. Das gilt ebenfalls für die 'containt' Elemente. Diese Elemente werden in der Benutzeroberfläche entsprechend markiert und können weiterhin als Basis für Diskussionen innerhalb des Modell-Kontextes verwendet werden, bevor sie endgültig gelöscht werden oder je nach Diskussionsausgang Bestandteil des Modells bleiben. Wie das UUID-Attribut ist dieses Attribut nicht direkt sichtbar, sondern kann nur über bestimmte Funktionen innerhalb der Benutzeroberfläche beeinflusst werden (Löschen, Wiederherstellen).

Locked-Attribut (*locked*) - Wie bereits bei dem UUID-Attribut und dem Delete-Attribut ist dieses Attribut nicht direkt in der Benutzeroberfläche sichtbar. Das Attribut wird dazu verwendet, ein Element oder eine Menge von Elementen gegen Modifikationen über die Benutzeroberfläche zu schützen. Diese Elemente werden innerhalb der Benutzeroberfläche gesondert markiert und können nicht mehr verändert werden. Dies betrifft einfache Attribute und auch Referenzen der entsprechenden Elemente. Interessant ist diese Funktionalität zum Beispiel beim Import von externen Daten, die nur temporär im Daten Modell Editor oder Instanz Modell Editor verwaltet werden. Das betrifft zum Beispiel die in Kapitel 4 angesprochenen Anforderungen, die im Daten Modell Editor und Instanz Modell Editor verfügbar sind.

Die Abbildung 3.8 zeigt einige Sub-Modelle, die Bestandteil des Core-Modells sind, im graphischen Editor des Daten Modell Editors.

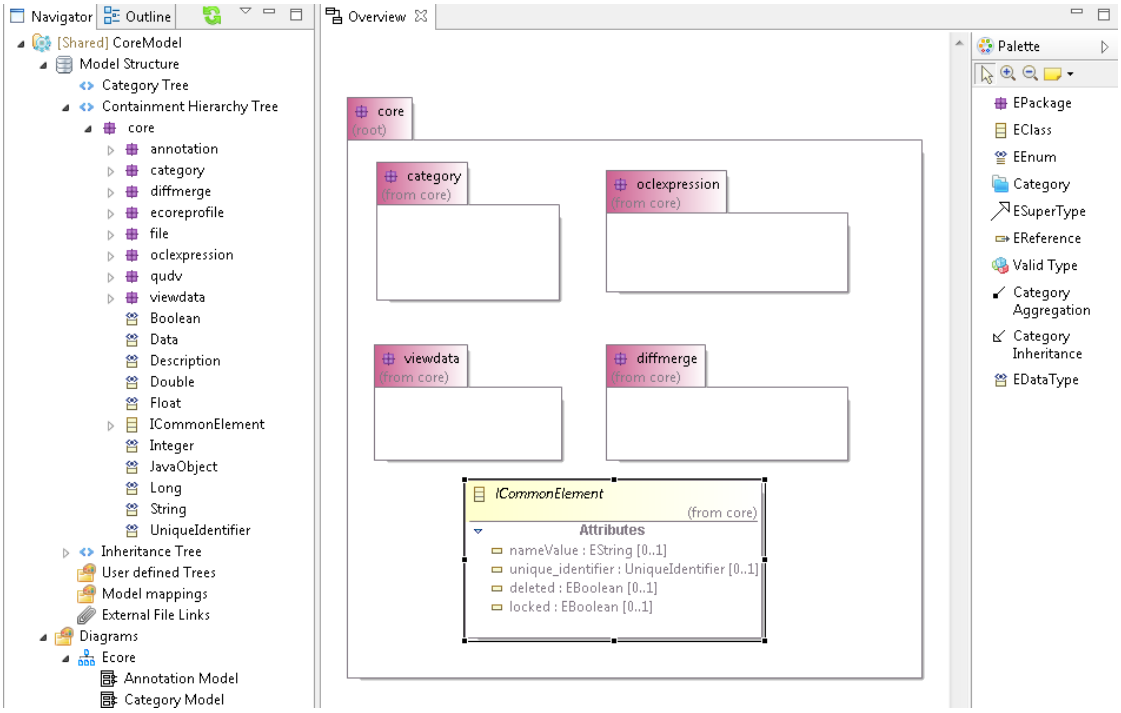

Abbildung 3.8: Core- und Sub-Modelle im Diagramm Editor des DME

3.2.2 Category-Modell und QUDV-Modell

Das Category-Modell sowie das QUDV-(Quantities, Units, Dimensions, Values) Modell bieten die Möglichkeit spezielle Engineering-Aspekte abzubilden. Die Modelle ermöglichen es, das benötigte Wissen einer spezifischen Domain (Engineering-Parameter) zum Beispiel im Bereich der Elektrotechnik in Form von Kategorien zu kapseln. Das Wissen wird dabei unter anderem in Form von *ValueProperties* definiert, die Bestandteil des QUDV-Modells sind. Das QUDV-Modell orientiert sich stark an dem konzeptionellen Datenmodell, welches von einer SysML-Arbeitsgruppe vorgestellt wurde (siehe auch [For14]).

Die Abbildung 3.9 zeigt den Zusammenhang zwischen *Category* und *ValueProperty*. Dabei ist anzumerken, dass in beiden Modellen noch weitere Klassen existieren. So definiert das QUDV-Modell zum Beispiel auch die Klasse *Unit*, die als Basis für die Definition von Maßeinheit dient.

Der Daten Modell Editor und der Instanz Modell Editor bieten spezielle Funktionalität

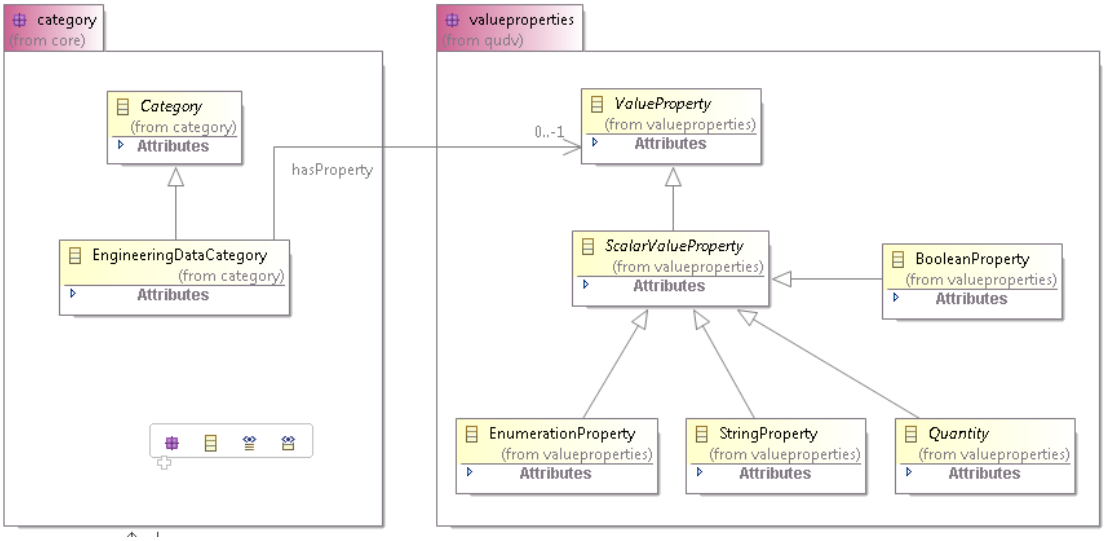

Abbildung 3.9: Category- und QUDV-Modell

basierend auf diesen Modellen. So ist es möglich im Daten Modell Editor Kategorien (*Category*) und Engineering-Parameter (*ValueProperty*) zu definieren, die im Instanz Modell Editor verwendet werden, um für Instanzen von Klassen aus dem Datenmodell weitere Informationen abzulegen. Zusätzlich kann der Gültigkeitsbereich für eine Kategorie im Daten Modell Editor eingeschränkt werden, indem man definiert, welche Klassen aus dem Datenmodell für diese Kategorie valide sind. Diese Information wird später vom Instanz Modell Editor genutzt, um zu überprüfen, ob einer Instanz eine spezifische Kategorie zugewiesen werden kann. Nach dem Zuweisen einer Kategorie an ein Element im Instanz Modell Editor ist es möglich, konkrete Werte für die im Daten Modell Editor definierten Engineering-Parameter zu vergeben.

Die Abbildung 3.10 zeigt einen Ausschnitt aus dem Instanz Modell Editor, indem einer Instanz eine Kategorie zugewiesen wurde. Nach dem Zuweisen ist es möglich, in einem sogenannten *QUDV Values View* konkrete Werte für die Eigenschaften zu definieren.

Wichtig ist dabei, dass sowohl die Kategorien als auch die Engineering-Parameter im Daten Modell Editor geändert werden können und zu jedem Zeitpunkt im Instanz Modell Editor wieder importierbar sind. Im Gegensatz dazu erfordern Änderungen am Datenmodell einen kompletten Entwicklungszyklus, wie er in Kapitel 3.6 erläutert wird. Außerdem ist es möglich nur ein Subset von Kategorien und Engineering-Parameter zu importieren, die für den jeweiligen Engineering-Bereich von Relevanz sind.

Neben dem Konzept Zusatzinformationen dynamisch zur Laufzeit verfügbar zu machen, dient das Category-Modell auch dazu Informationen, die in PUS (Packet Utilisation

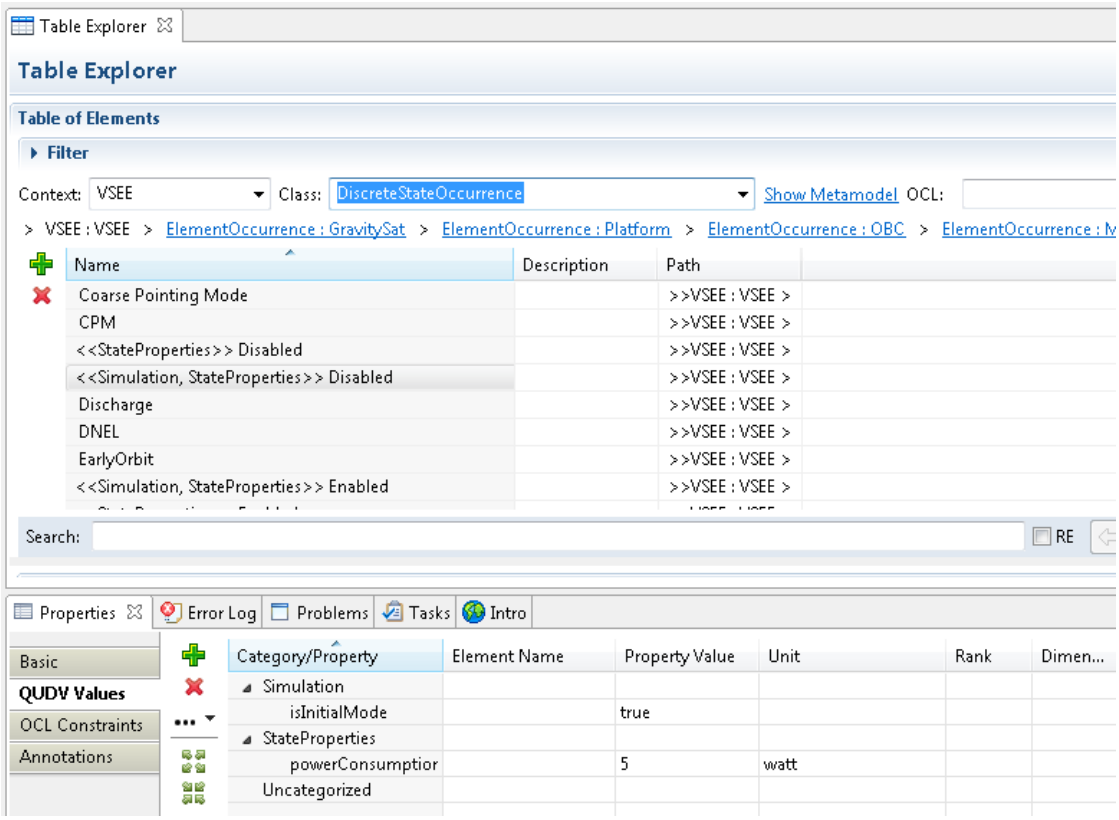

Abbildung 3.10: Setzen von Engineering-Parametern im Instanz Modell Editor

Standard) definiert sind, abzubilden und den Benutzer des Instanz Modell Editor bei der Definition der Packets (Datenpakete zum Informationsaustausch) zu unterstützen (siehe auch [Age14b]).

Klassen, deren Instanzen dynamisch zur Laufzeit Kategorien zugewiesen bekommen sollen, müssen das Interface *ICategorizable* implementieren, welches ebenfalls Bestandteil dieses Modells ist.

3.2.3 OCL-Modell

Wie bereits in Kapitel 2.4 erwähnt, kann OCL für unterschiedliche Aspekte eingesetzt werden. Der Daten Modell Editor bietet die Möglichkeit OCL-Ausdrücke zu definieren, die zum Ableiten von Werten für Attribute und zum Überprüfen von semantischen Zusammenhängen genutzt werden können. Der Instanz Modell Editor bietet ergänzend dazu die Möglichkeit diese OCL-Ausdrücke auszuwerten. Der Datenaustausch zwischen Daten Modell Editor und Instanz Modell Editor sowie die Speicherung der

OCL-Ausdrücke erfolgt dabei wieder durch ein Modell. Dieses OCL-Modell beinhaltet den OCL-Ausdruck selbst sowie den Kontext des Ausdrucks. Der Kontext kann eine Klasse oder ein Attribut sein. Neben diesen Informationen werden der Typ des Ausdrucks und zusätzliche Parameter gespeichert. Die Abbildung 3.11 zeigt das OCL-Modell.

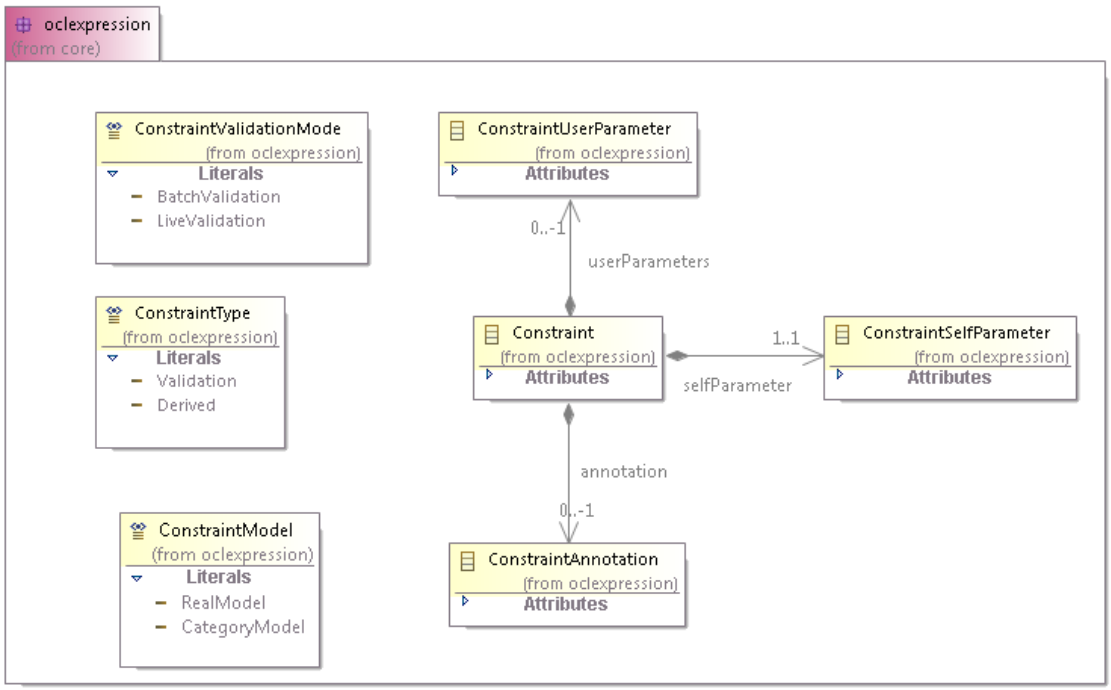

Abbildung 3.11: OCL-Modell

Neben der bereits erwähnten Funktionalität bieten der Daten Modell Editor und Instanz Modell Editor auch eine spezielle Unterstützung für Engineering-Parameter, die in Kapitel 3.2.2 erläutert werden. Wie in dem betreffenden Kapitel bereits erwähnt wird, handelt es sich dabei um eine dynamische Erweiterung zur Laufzeit, um zusätzliche Informationen für Instanzen im Instanz Modell Editor zu hinterlegen. Um eine Validierung solcher Eigenschaften mit OCL zu gewährleisten, ist es möglich spezielle OCL-Ausdrücke zu definieren. Diese basieren nicht direkt auf dem Datenmodell im Daten Modell Editor, sondern verwenden ein transientes Modell während der Definition und Evaluation des OCL-Ausdrucks. Dazu wird mit Hilfe von *Dynamic EMF* (siehe [IBM14a]) zur Laufzeit ein Modell erstellt, welches die Engineering-Parameter (*ValueProeprty*) als Attribute einer Klasse behandelt und entsprechend typisiert. Im Instanz Modell Editor wird durch dieselbe Funktionalität dieses Modell auch erstellt und entsprechend der Werte im Instanz Modell populiert. Durch dieses Verfahren ist es möglich, OCL-Ausdrücke im Daten Modell Editor zu definieren und im Instanz Modell

Editor entsprechend zu evaluieren.

Im Daten Modell Editor werden zusätzlich alle OCL-Ausdrücke überwacht, das heißt bei Änderungen am Datenmodell werden die OCL-Ausdrücke angepasst, falls dies möglich ist. Zum Beispiel kann eine Namensänderung eines Attributes Auswirkungen auf einen OCL Ausdruck haben, wenn dieses Attribut im Ausdruck verwendet wird. Ermittelt wird diese Abhängigkeit mit Hilfe des *Abstract Syntax Tree* (AST) des OCL-Ausdrucks, der mittels einer *Visitor* Implementierung traversiert und analysiert werden kann. Änderungen, die nicht automatisiert angepasst werden können, werden als *ConstraintAnnotation* für den OCL-Ausdruck hinterlegt und die Fehler müssen durch den Anwender selbst behoben werden.

3.2.4 Weitere Modelle

Neben den bereits erwähnten Modellen existieren weitere Modelle mit unterschiedlichen Funktionen. Eines dieser Modelle ist das sogenannte View-Modell, es dient dazu Informationen der graphischen Editoren (GMF/Zest) zu speichern. GMF-Editoren verwenden bereits ein ähnliches Modell, um die Position und Größe von Knoten, Ankerpunkte von Kanten und andere Informationen zu speichern. Ziel des View-Modells ist es, implementationsabhängig zu sein. Dadurch wird es möglich, die gespeicherten Informationen auch in anderen Technologien wie zum Beispiel Graphiti ([Ecl14b]) wiederzuverwenden.

Ein weiteres Modell ist das Diffmerge-Modell, welches Informationen über den 'Merge'-Prozess zweier Datenmodelle speichert. So beinhaltet es unter anderem Informationen über Konflikte, die während des Zusammenführens der beiden Modelle bestanden. Dadurch wird es möglich, nachträglich diese Konflikte im resultierenden Modell zu behandeln.

Neben diesen zwei Modellen existiert ein weiteres sogenanntes File-Modell, dessen Aufgabe es ist Informationen über externe Daten zu speichern. So ist es möglich, Objekten im Daten Modell Editor oder Instanz Modell Editor externe Dateien zuzuweisen. Die Dateien selbst werden in einem dedizierten Ordner innerhalb des jeweiligen Projektes ablegt. Die Verbindung zwischen physischer Datei und dem jeweiligen Objekt werden durch Instanzen des File-Modells repräsentiert. Die Klassen, deren Instanzen eine externe Datei zugewiesen bekommen soll, müssen das Interface *IFileTarget* implementieren, welches ebenfalls Bestandteil des File Modells ist.

3.3 Daten Modell Editor (DME)

Die Abbildung 3.12 zeigt einige der DME-Komponenten, die zusätzlich zu den bereits erwähnten Funktionen im Daten Modell Editor verfügbar sind.

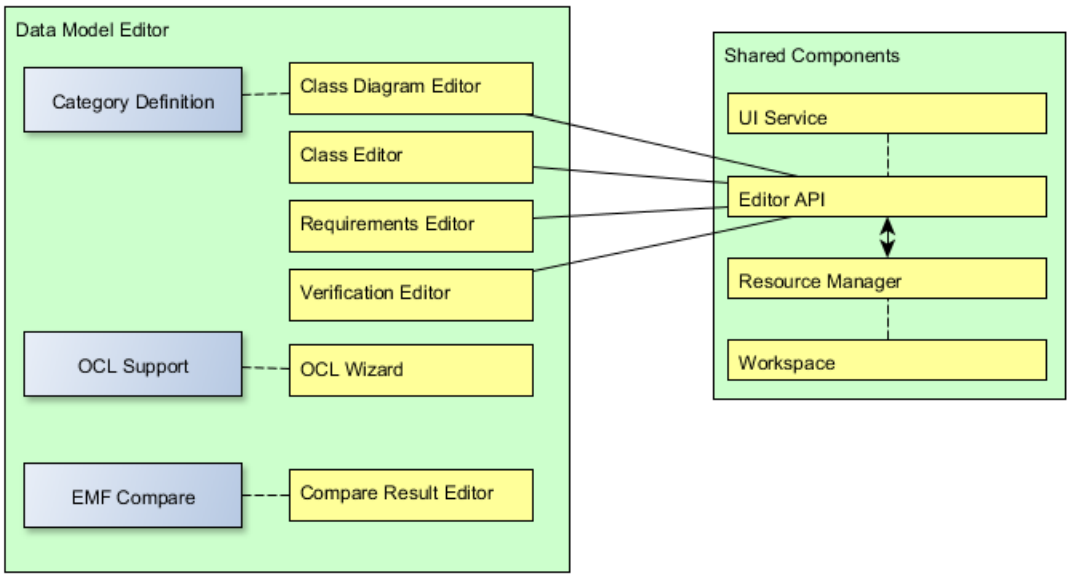

Abbildung 3.12: Übersicht über einige DME-Komponenten

Für die Definition und das Management des Datenmodells ist dabei der graphische Editor (*Class Diagram Editor*) von besonderer Bedeutung. Dieser bietet die Möglichkeit Klassen und Referenzen graphisch zu definieren, wie es ebenfalls in Klassendiagrammen von UML-Werkzeugen üblich ist. Zusätzlich kann der graphische Editor Kategorien definieren. Ein weiterer interessanter Editor ist der *Class Editor*, der zum Editieren einer Klasse genutzt werden kann. Dieser Editor bietet einen Überblick über alle für Klassen wichtigen Aspekte wie zum Beispiel Attribute, ein- und ausgehende Referenzen sowie OCL-Ausdrücke.

Neben diesen beiden Editoren sind der *Anforderungs-Editor* (Requirements Editor) und *Validierungs-Editor* (Verification Editor) Bestandteil des Daten Modell Editors. Erläutert werden diese beiden Editoren in Kapitel 4.3.3 und 5.4.

3.3.1 Modellierungssprache - EcoreExt

Die Modellierungssprache, die im Daten Modell Editor zur Definition des Datenmodells genutzt wird, basiert auf Ecore. Im Gegensatz zu UML ist der Sprachumfang von Ecore sehr begrenzt. Jedoch reichen die Sprachmittel für die Definition des Datenmodells zum jetzigen Zeitpunkt aus beziehungsweise werden von UML auch nicht direkt unterstützt. So werden zum Beispiel für das Datenmodell selbst keine Zustandsautomaten benötigt wie sie UML bereitstellt. Besondere Anforderungen wie der Umgang mit Engineering-Parameter wird hingegen von beiden Modellierungssprachen nicht unterstützt. Aus diesem Grund wird im Daten Modell Editor eine spezielle Modelldefinition (Category-Modell) genutzt, um diese Informationen zu modellieren.

Für den Daten Modell Editor selbst wird basierend auf dem Ecore-Metamodell ein sogenanntes *EcoreExt*-Modell erzeugt. Dabei wird das Ecore-Metamodell durch Interfaces des Core-Modells erweitert. Dieser Prozess gestattet es, die Sprachmittel von Ecore selbst zu erweitern, jedoch kann das *EcoreExt*-Modell keine von Ecore definierten Aspekte ändern, da alle Typen des *EcoreExt*-Modells von den Ecore-Metatypen erben. Das bedeutet aber auch, dass eine Reduktion des Datenmodells auf Basis von *EcoreExt* konform zum Ecore-Standard ist. Die Abbildung 3.13 zeigt den Zusammenhang zwischen Ecore, EcoreExt, Core-Modell und dem Datenmodell (CDM) im Daten Modell Editor.

Wie in der Abbildung 3.13 zu sehen ist, besteht das Datenmodell im Daten Modell Editor aus Instanzen des *EcoreExt*-Modells und Instanzen des Core-Modells. Zu beachten ist dabei, dass die Instanzen des *EcoreExt*-Modells im Instanz Modell Editor wieder instanziiert werden, wohingegen die Instanzen des Core-Modells direkt wiederverwendet werden. Dadurch ergibt sich die in Tabelle 3.1 dargestellte Klassifikation nach MOF (Meta Object Facility):

	Modelle
M3-Ebene	Ecore
M2-Ebene	Ecore / EcoreExt
M1-Ebene	Klassenmodell (Instanzen von EcoreExt) / Core Modell
M0-Ebene	Instanzen des Klassenmodells / Instanzen des Core Modells

Tabelle 3.1: Klassifikation nach Meta Object Facility (MOF)

Anhand der Tabelle 3.1 ist zu erkennen, dass der Daten Modell Editor sowohl Modelle der Ebene 1 als auch der Ebene 2 instanziiert. Der Instanz Modell Editor hingegen

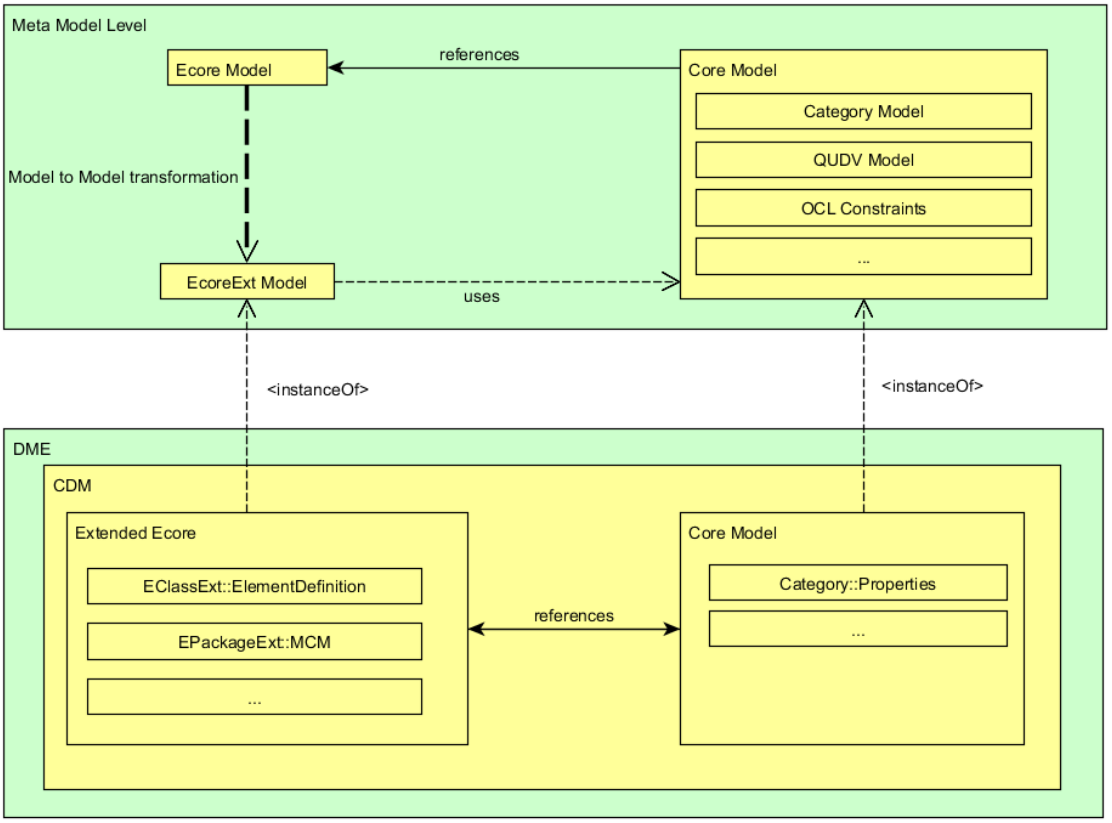

Abbildung 3.13: Die Modellierungssprache *EcoreExt* und das Datenmodell im DME

instanziiert nur Modelle der Ebene 1. Jedoch tauscht der Instanz Modell Editor Informationen auf Basis des Core-Modells mit dem Daten Modell Editor aus. Das bedeutet das Teile des instantiierten Core-Modells zwischen Daten Modell Editor und Instanz Modell Editor geteilt werden. So werden zum Beispiel Instanzen von *Categories*, *Value-Properties*, *Requirements* und *TestCases* zwischen den beiden Editoren auf Basis der gemeinsamen Modelldefinition geteilt.

3.3.2 Benutzeroberfläche des Daten Modell Editors

In der Abbildung 3.14 sind die Hauptbereiche des Daten Modell Editors hervorgehoben. Auf der linken Seite befindet sich der Navigator, der die Projekte verwaltet und als Einstiegspunkt für die Modelle dienen kann. Um das zu realisieren, werden neben den physischen Dateien auch modellspezifische Informationen angezeigt. Diese umfassen zum Beispiel Diagramme oder auch *Dynamic Trees* (Tree Explorer), die als Einstieg verwendet werden können.

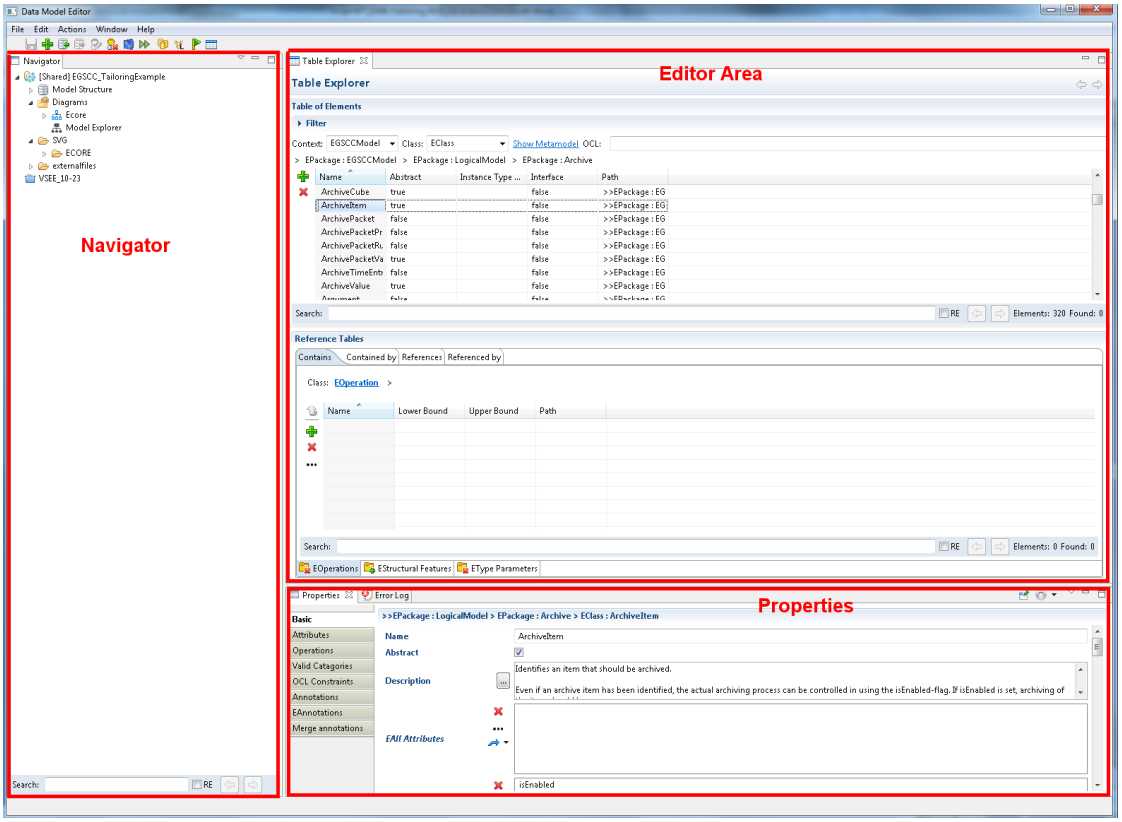

Abbildung 3.14: DME Layout mit aktiven Table Explorer

Im oberen rechten Bereich werden die jeweiligen Editoren angezeigt, wobei die Abbildung 3.14 den *Table Explorer* zeigt. Andere Editoren werden als separate Tabs innerhalb dieses Bereiches nebeneinander gelistet.

Der untere Bereich wird für den *Properties View* verwendet. Dieser zeigt in unterschiedlichen Tabs verschiedene Aspekte des im Editor selektierten Elements. Die Informationen reichen dabei von einfachen Attributen über Referenzen bis hin zu, in Kapitel 3.2.2 erwähnten, Engineering-Parametern.

3.4 Instanz Modell Editor (IME)

Ähnlich zum Daten Modell Editor beinhaltet der Instanz Modell Editor spezifische Funktionen, die in Abbildung 3.15 dargestellt sind.

Wie auch der Daten Modell Editor enthält der Instanz Modell Editor die zwei Editoren zum Verwalten der Anforderungen und Daten der Validierung. Neben diesen beiden

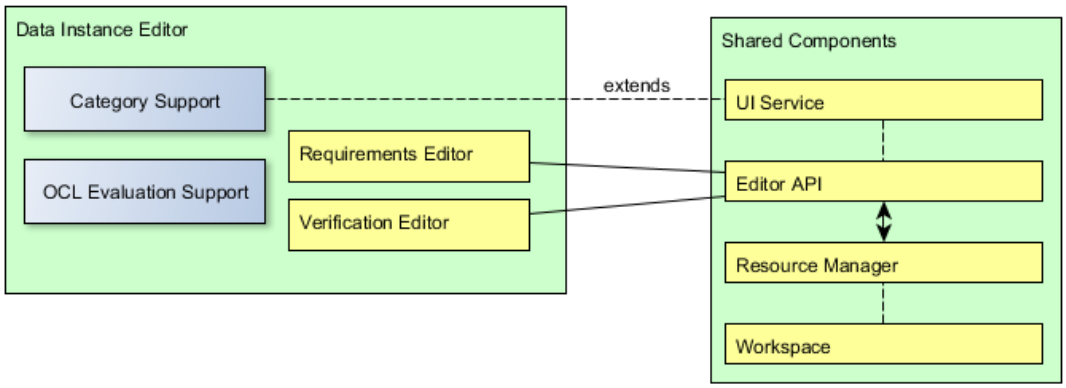

Abbildung 3.15: IME components overview

Editoren bietet der Instanz Modell Editor keine zusätzlichen Editoren bis auf die drei modellunabhängigen Editoren, die auch im Daten Modell Editor verfügbar sind.

Die Benutzeroberfläche selbst ist dabei genauso strukturiert wie im Daten Modell Editor und folgt dem in Abbildung 3.14 dargestellten Layout.

3.5 Code Generation Framework - CGF

Das Code Generation Framework (CGF) besteht aus einer Reihe von Plug-ins die von der Firma ScopeSET entwickelt wurden. Die Plug-ins werden innerhalb der Eclipse-Entwicklungsumgebung genutzt, um diverse Prozesse bei der Entwicklung von RCP-Anwendungen zu automatisieren. Die zu automatisierenden Prozessschritte, auch Tasks genannt, werden in einem speziellen Modell innerhalb des Workspaces in der Entwicklungsumgebung definiert. Die Abbildung 3.16 zeigt einen Ausschnitt aus diesem Modell, welches im folgenden CGF-Modell genannt wird:

Für die Entwicklung des Daten Modell Editor und Instanz Modell Editor sind dabei zwei Tasks von besonderer Bedeutung. Der erste Task generiert die *Model-* und *Edit-Plug-ins* der im CGF-Modell definierten EMF-Modelle. Der zweite Task generiert auf Basis des Notation-Modells unter Verwendung der Templatesprache Acceleo Code, der von den Editoren, die in diesem Kapitel erläutert werden, verarbeitet werden kann. Die Abbildung 3.17 zeigt schematisch den Entwicklungsprozess.

Zu Beginn werden alle relevanten Plug-ins, zum Beispiel aus einem Versionsverwaltungssystem in den Workspace importiert. Danach werden die CGF-Tasks ausgeführt,

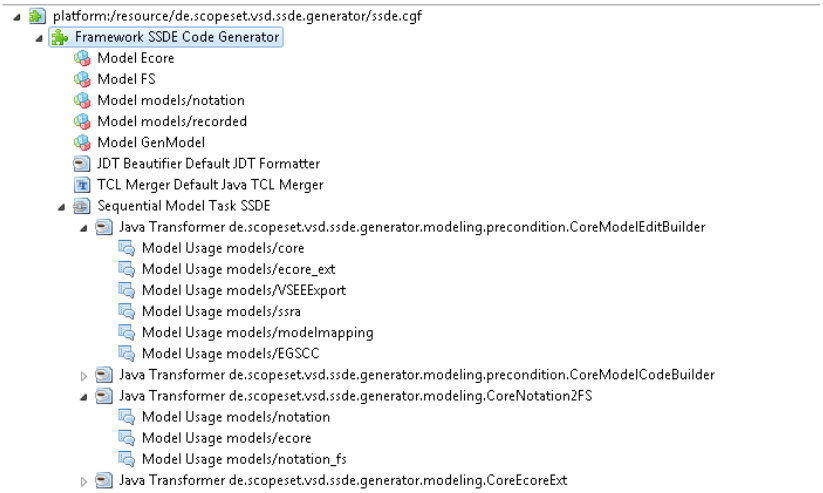

Abbildung 3.16: CGF-Model

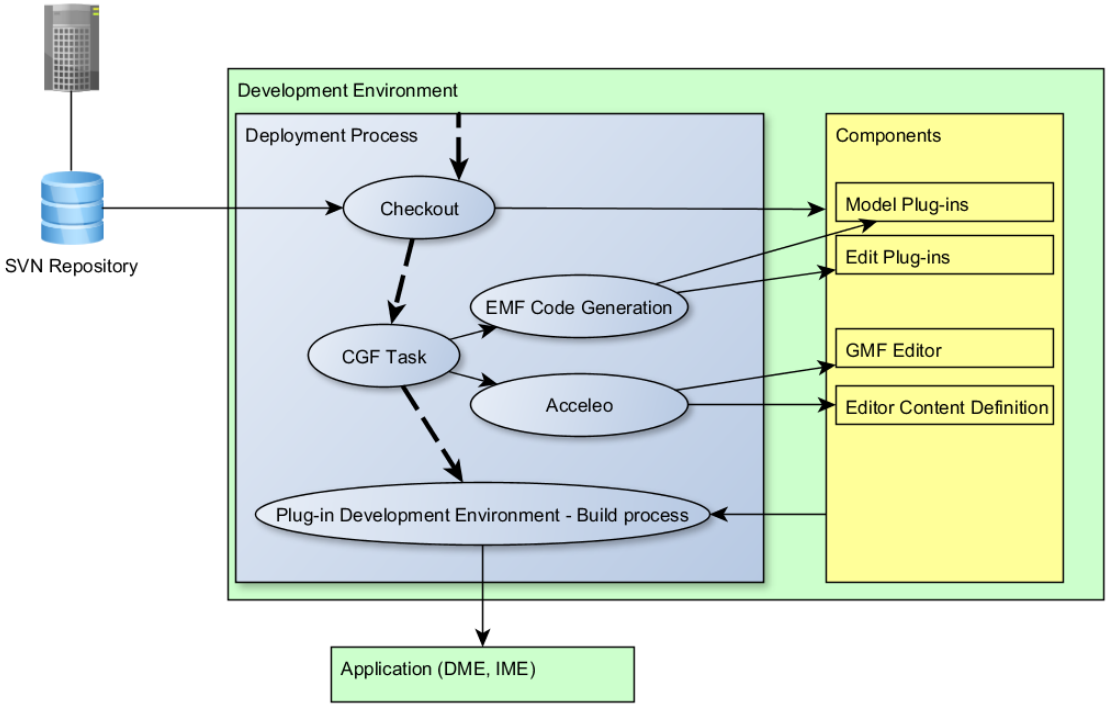

Abbildung 3.17: Automatisierter Build-Prozess mit CGF

das heißt die Acceleo- und EMF-Codegenerierung wird vorgenommen. Am Ende wird die RCP-Anwendung unter Verwendung der von Eclipse bereitgestellten Funktionen exportiert.

3.6 Deployment Prozess - Instanz Modell Editor

Für den Validierungsprozess (siehe Kapitel 5) des konzeptionellen Datenmodells wird ein Prototyp verwendet. Dieser Prototyp wird im Kontext dieser Arbeit als Instanz Modell Editor (IME) bezeichnet und basiert direkt auf dem Datenmodell. Da die Validierung iterativ erfolgt und dadurch auch kontinuierlich Änderungen am Datenmodell erfolgen, ist es notwendig einen einfachen Prozess für die Entwicklung des Instanz Modell Editors zu haben. Die Entwicklungszeit eines neuen Instanz Modell Editors soll den Validierungsprozess nicht unnötig lang verzögern. Das erreicht man durch die Basisarchitektur, welche am Anfang des Kapitels 3 erläutert wurde. Durch die folgenden drei Schritte kann ein Instanz Modell Editor basierend auf dieser Architektur definiert werden.

1. Die Definition einer RCP-Anwendung in Form eines Eclipse Plug-ins. Die Abhängigkeiten sind dabei alle Plug-ins, welche zur Basisarchitektur gehören und Bestandteil des Instanz Modell Editor sein sollen. Zusätzlich werden die Plug-ins benötigt, die durch EMF auf Basis des Datenmodells generiert werden. Dabei handelt es sich um das *Model-* und *Edit-Plug-in*. Neben diesen können auch weitere Plug-ins integriert werden, wie es zum Beispiel für den Anforderungs-Editor (siehe Kapitel 4.3.3) der Fall ist.

2. Das Erstellen eines Eclipse-Wizards zum Erzeugen eines Projekts mit der entsprechenden *Resource*, die Instanzen des Datenmodells beinhalten soll.

3. Das Aufbereiten des Datenmodells und Generierung des *Model-* und *Edit-Plug-ins*.

Der iterative Anteil, der für jede neue Version des Instanz Modell Editors durchzuführen ist, beschränkt sich dabei auf Punkt 3. Die Abbildung 3.18 zeigt eine Iteration, die notwendig ist, um eine neue Version des Instanz Modell Editors zu erstellen.

Wie in der Abbildung 3.18 zu sehen ist, wird durch einen speziellen Export das konzeptionelle Datenmodell aus dem Daten Modell Editor in ein EMF-Modell exportiert. Während dieser Transformation werden die Interfaces aus dem Core-Modell zu den Basisklassen im konzeptionellen Datenmodell hinzugefügt. Das resultierende EMF-Modell dient dann als Basis für den Entwicklungsprozess des Instanz Modell Editors. So werden aus dem EMF-Modell das sogenannte *Model-Plug-in* und *Edit-Plug-in* generiert.

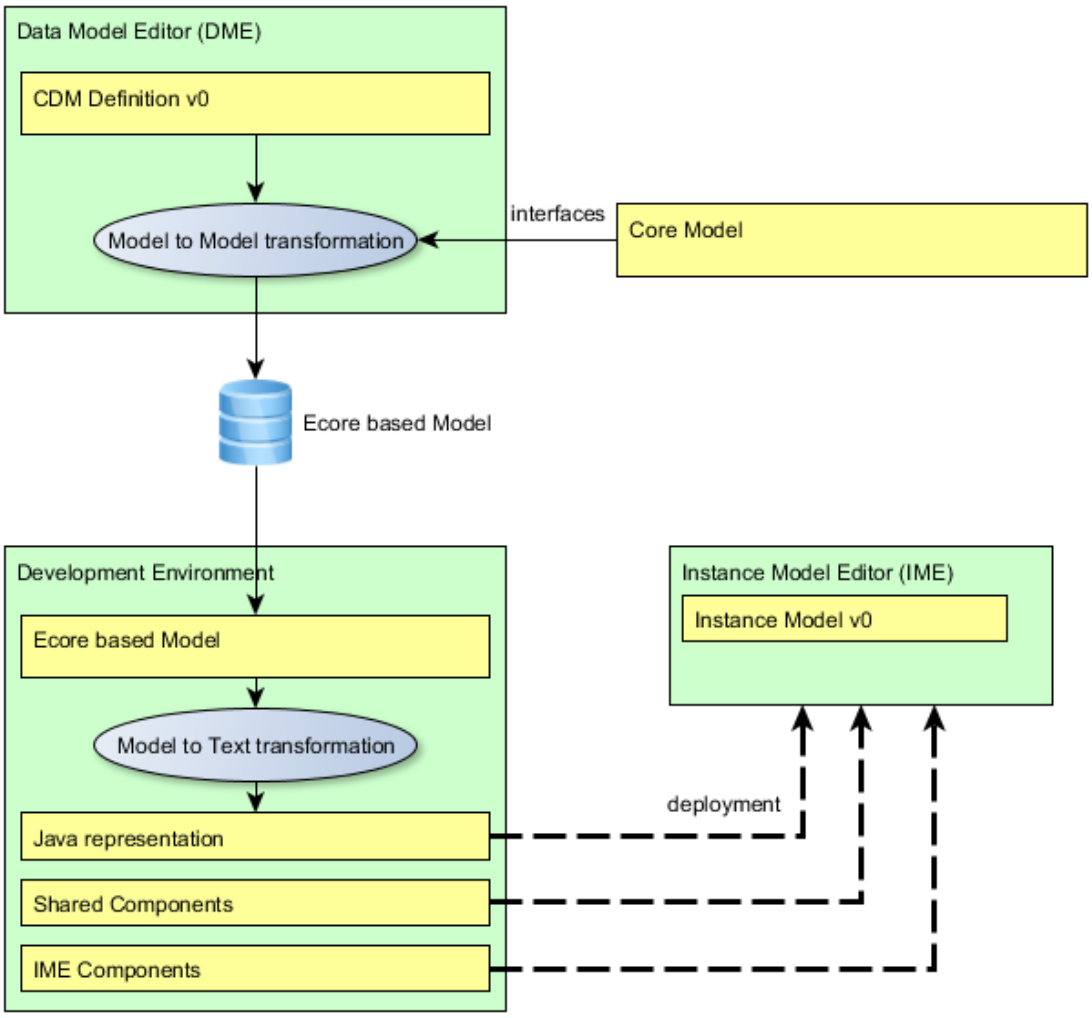

Abbildung 3.18: Deployment Prozess des Instanz Modell Editors

Mithilfe der RCP-Export Werkzeuge, die von Eclipse bereitgestellt werden, werden danach alle relevanten Plug-ins gebündelt und in eine RCP-Anwendung exportiert. Die dadurch erstellte Anwendung kann im Anschluss dazu genutzt werden, Instanzen des konzeptionellen Datenmodells zu erstellen.

3.7 Zusammenfassung

Die hier vorgestellte Architektur und der Daten Modell Editor bieten eine sehr gute Basis für die Erweiterung durch zusätzliche Aspekte. So ist nicht nur der OCL-Support und die Unterstützung für Engineering-Parameter ein Indiz dafür, dass die Komponenten sehr gut für ingenieurtechnische Aspekte erweitert werden können, sondern auch

die Masterarbeit von Bryan Beck (siehe [Bec14]). In dieser Arbeit erweitert er den Daten Modell Editor, um Aspekte der Prozessmodellierung zu unterstützen. Konkret bedeutet dies, die Einführung eines Prozessmodells und eines graphischen Editors auf Basis von Graphiti, um es zu ermöglichen Prozesse zu modellieren.

In dieser Arbeit wird ebenso die bestehende Architektur durch zwei Komponenten erweitert, die es ermöglichen, Anforderungen und Testfälle für die Validierung zu verwalten. Dabei werden die einzelnen Artefakte nicht nur separat betrachtet, sondern auch in ihrem Kontext zueinander.

KAPITEL 4

Anforderungsmanagement

4.1 Definition und Qualitätskriterien

Der Prozess der Anforderungsanalyse und das Management der Anforderungen wird oft auch als Requirements-Engineering bezeichnet. Wie wir aus Kapitel 1.2.1 ableiten können, handelt es sich um einen Prozess der bestimmten Regeln folgt, um ein System bestehend aus Anforderungen zu erstellen und zu verwalten. Wie auch bei anderen Engineering Disziplinen gibt es auch für diesen Regeln, Methoden, Qualitätsmerkmale und Modelle. Wobei die Modelle häufig in einer textuellen nicht formalen Sprache definiert sind (zum Beispiel ein Worddokument).

Betrachten wir zunächst die Anforderung an sich und deren Bedeutung. Nach IEEE ([Wik14a]) und IREB ([Wik14b]) beschreibt eine Anforderung eine Funktion oder Bedingung in einem System, welche zum Erreichen dedizierter Ziele eines Benutzers erforderlich sind. Außerdem spezifizieren Anforderungen welche Normen, Standards und Regeln ein System erfüllen muss. Man unterscheidet dabei zwischen funktionalen und nicht funktionalen Anforderungen. Funktionale Anforderungen beschreiben die Interaktion mit dem Benutzer oder auch Abläufe im System selbst, wohingegen nicht funktionale Anforderungen zum Beispiel Qualitätskriterien und die zu verwendenden Technologien beschreiben (siehe [Rup09] Seite 18).

Nicht zu unterschätzen bei den Anforderungen ist die Verwendung von Verben, welche die Verbindlichkeit einer Anforderung beschreiben. Dabei kann man prinzipiell zwischen drei Arten unterscheiden (siehe [Rup09] S.18-19 und [PR10] S.64-65):

- Pflicht - muss umgesetzt werden (Schlüsselwort: 'muss')

- Wunsch - sollte umgesetzt werden (Schlüsselwort: 'sollte')
- Absicht - wird umgesetzt werden, sobald eine andere Bedingung erfüllt ist (Schlüsselwort: 'wird')

Der Ingenieursverband IEEE definiert für die Anforderungsspezifikation Qualitätskriterien, die im Buch 'Requirements-Engineering und -Management' erweitert und Anwendung auf die einzelnen Anforderungen finden. Die wichtigsten Kriterien, die in [PR10] (S.53ff.) definiert werden, sind dabei:

- Konsistent - Die Anforderungen dürfen sich gegenseitig und auch inhaltlich nicht widersprechen.
- Verfolgbar - Jede Anforderung muss eindeutig identifiziert werden können (Identifikationsnummer), damit andere Anforderungen oder andere Elemente diese referenzieren können.
- Aktuell und gültig - Es existiert nur eine Version der Anforderung, die zum aktuellen Zeitpunkt volle Gültigkeit für das System hat.
- Realisierbar und prüfbar - Es muss möglich sein, die Anforderung mit den gegebenen Rahmenbedingungen (Geld, personelle Ressourcen, Grenzen des Systems) umzusetzen. Außerdem muss es möglich sein, die Realisierung zu testen.

4.2 Vorgehensmodelle

Vorgehensmodelle dienen der Organisation von Abläufen und definieren für einzelne Prozessabschnitte Aktivitäten, die für die Entwicklung des Systems durchgeführt werden sollten. Prinzipiell kann man zwischen wasserfall-orientierten und agilen Vorgehensmodellen unterscheiden. Wasserfall-orientierte Prozesse bestehen aus einzelnen Tätigkeiten, die jeweils abgeschlossen werden müssen, bevor mit der nächsten Tätigkeit begonnen wird. Die generierten Artefakte einer solchen Tätigkeit dienen dann als Input für die darauf folgende Aktion. Ein bekannter Vertreter dieses Modells ist das V-Modell, welches zum Beispiel in der Softwareentwicklung eingesetzt wird. Der Ablauf der einzelnen Phasen erfolgt dabei wie folgt (siehe [Wik14d]):

1. Voruntersuchung

2. Ist-Aufnahme
3. Ist-Kritik
4. Sollkonzeption – Lösungsgenerierung
5. Sollkonzeption – Lösungsbewertung und -auswahl
6. Einführung/Umsetzung
7. Evaluierung und Weiterentwicklung

Das Ziel dieser statisch strukturierten Modelle ist es, Kosten durch spätere Änderungen zu vermeiden, indem das System bereits in frühen Phasen vollständig formal beschrieben ist. Es hat sich aber in den letzten Jahren gezeigt, das diese Herangehensweise nicht zwingend zielführend ist. Das bedeutet es kann passieren, dass ein System noch während der Entwicklung überflüssig wird beziehungsweise nicht mehr den Anforderungen des Auftraggebers und des freien Marktes gerecht wird. Aus diesem Grund haben sich agile Modelle in den letzten Jahren immer mehr durchgesetzt. Diese Entwicklung zeigt sich zum Beispiel auch bei der Weiterentwicklung des V-Modells in das *V-Modell XT*. Dabei beschreibt das Modell primäre einzelne Bausteine, die das Vorgehen definieren und schränkt aber deren zeitliche Abfolge nicht ein. Ein weiteres Beispiel dafür ist die Veröffentlichung von Thomas Mathoi ([Mat14]), die zeigt das agile Prozesse auch in eher statisch anmutenden Bereichen, wie dem Bauingenieurwesen, an Bedeutung gewinnen.

Gerade in der Softwareentwicklung setzen sich immer mehr agile Methoden wie zum Beispiel Scrum (siehe [Ste10] Kapitel 3.3), Crystal Methoden (siehe [Ste10] Kapitel 3.4) oder Extreme Programming durch. Die Idee dahinter ist sehr treffend im 'Manifest für Agile Softwareentwicklung' zusammengefasst:

Individuen und Interaktionen mehr als Prozesse und Werkzeuge
Funktionierende Software mehr als umfassende Dokumentation
Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung
Reagieren auf Veränderung mehr als das Befolgen eines Plans

Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden,
schätzen wir die Werte auf der linken Seite höher ein. ([KB])

Das Ziel ist durch agile Prozesse besser auf Kundenwünsche und Marktentwicklungen eingehen zu können und damit den Erfolg sowie den Gewinn zu maximieren.

Die Wahl der Methode hat wiederum auch Auswirkung auf das Anforderungsmanagement. So werden bei wasserfall-orientierten Prozessen die Anforderungen in einer frühen Phase vollständig definiert. Änderungen an diesen erfolgen nur selten oder mit hohen bürokratischen Aufwand. Bei agilen Prozessen ist das Anforderungsmanagement prozessbegleitend angesiedelt. Das bedeutet die Anforderungen werden neuen Situationen angepasst und für Teile des Systems verfeinert. Ein Beispiel dafür ist das *Backlog*, welches in der *Scrum*-Methode zum Einsatz kommt und zwischen dem *Product Backlog* und *Sprint Backlog* unterscheidet. Das *Product Backlog* ist dabei eine grobgranulare Sammlung von Anforderungen an das System. Wohingegen das *Sprint Backlog* eine Verfeinerung der Anforderungen aus dem *Product Backlog* darstellt und die Aufgaben für eine Iteration der Entwicklung des Systems beinhaltet. Prinzipiell kann man sagen, dass zu Beginn des Systementwicklungsprozesses Anforderungen mit einer hohen bis mittleren Granularität definiert werden, um eine Kosten- und Zeitabschätzung zu gewährleisten. Diese werden während des Entwicklungsprozesses verfeinert, um einzelne Aktionen planen und durchführen zu können.

4.3 Anwendungs-Integration

Um Anforderungen im Daten Modell Editor und Instanz Modell Editor zu verwalten, war es wichtig eine gemeinsame Datenbasis zu schaffen. Dies erfolgte durch die Definition eines geeigneten Datenmodells, welches in Kapitel 4.3.1 erläutert wird. Neben dem Datenmodell spielt der Datenaustausch zwischen Daten Modell Editor und Instanz Modell Editor sowie externen Anwendungen eine zweite wichtige Rolle. Der dritte Aspekt, der betrachtet wird, ist die graphische Benutzeroberfläche, die für die Anforderungen im Daten Modell Editor und Instanz Modell Editor bereitgestellt wird (siehe Kapitel 4.3.3).

4.3.1 Anforderungs-Modell

Bei der Definition des Anforderungs-Modells (Requirements-Modell) werden mehrere Aspekte berücksichtigt. So liegt der Fokus nicht allein auf der Anforderung selbst, sondern ebenso auf deren Kontext im Bereich der Datenmodellierung. Es ist zu berücksichtigen, dass Anforderungen zum Teil in Form von Klassen, Referenzen oder Desi-

gnentscheidungen abgebildet werden. Betrachten wir aber zunächst das Anforderungs-Modell ohne Kontextbezug zum Datenmodell. Die folgende Abbildung zeigt die Klassen des Anforderungs-Modells, welches aus dem VSD-Projekt (Virtual Spacecraft Design, siehe [Age12a]) abgeleitet wurde.

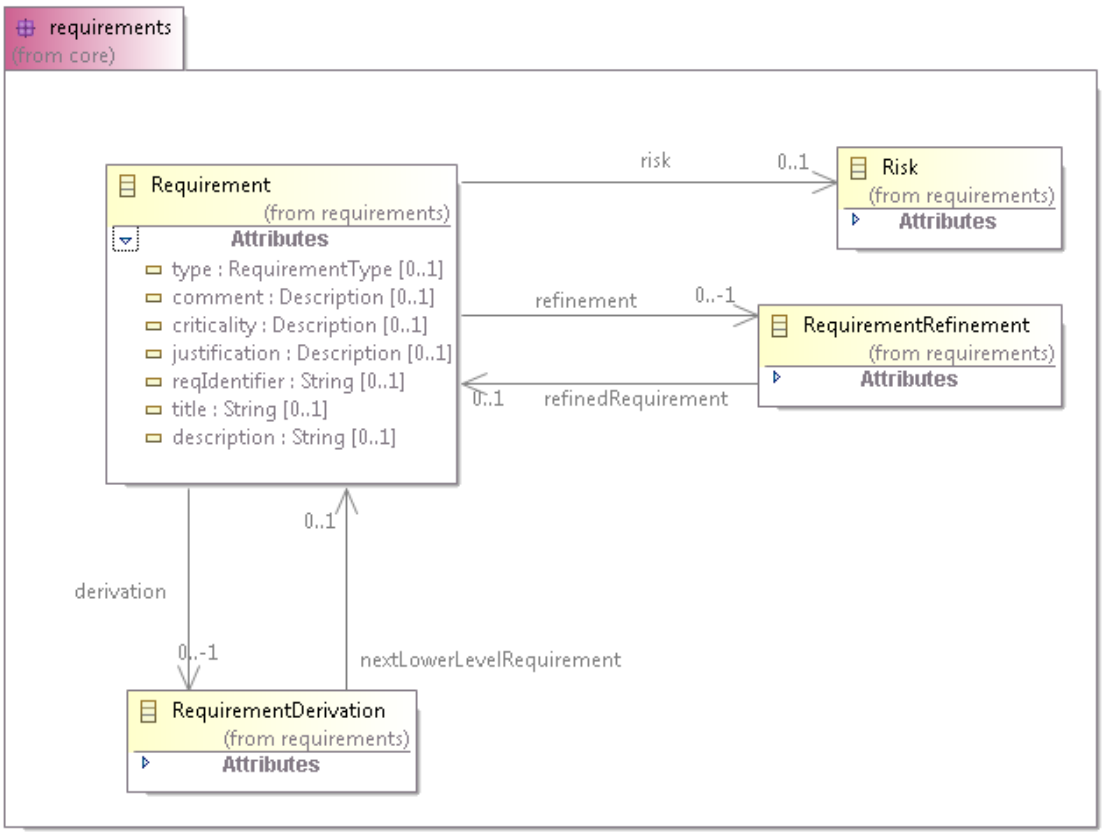

Abbildung 4.1: Anforderungs-Modell

RequirementRepository - Dient der Organisation von *Requirements* (Anforderungen), um sie voneinander abzugrenzen oder hierarchische Strukturen aufzubauen.

Requirement - Die Anforderung selbst mit der textuellen Definition sowie Indikatoren über Art und Risikobewertung, wenn die Anforderung nicht erfüllt wird.

RequirementRefinement - Dient dazu Anforderungen untereinander zu strukturieren und den Detaillierungsgrad für eine Anforderung in einer oder mehrere Anforderung zu erhöhen.

RequirementDerivation - Ist ähnlich dem *RequirementRefinement* und dient ebenfalls zur Strukturierung der Anforderungen untereinander. Dabei wird definiert, dass

sich Anforderungen, zum Beispiel an ein Subsystem, von Anforderungen aus einem übergeordneten System ableiten.

Bei *RequirementRefinement* und *RequirementDerivation* handelt es sich um zwei spezielle Klassen, die einem Modellierungsmuster folgen, welches bereits im VSD-Projekt zum Einsatz kam. Dabei handelt es sich um sogenannte *ConceptLinks*, die in der Oberfläche des Instanz Modell Editors ähnlich wie Beziehungen (*EReference*) behandelt werden. Das Konzept ähnelt dabei der Assoziationsklasse in UML.

Betrachten wir nun den Bezug zum Datenmodell und die Verbindung zum Validierungs-Modell, das in Kapitel 5.3 erläutert wird.

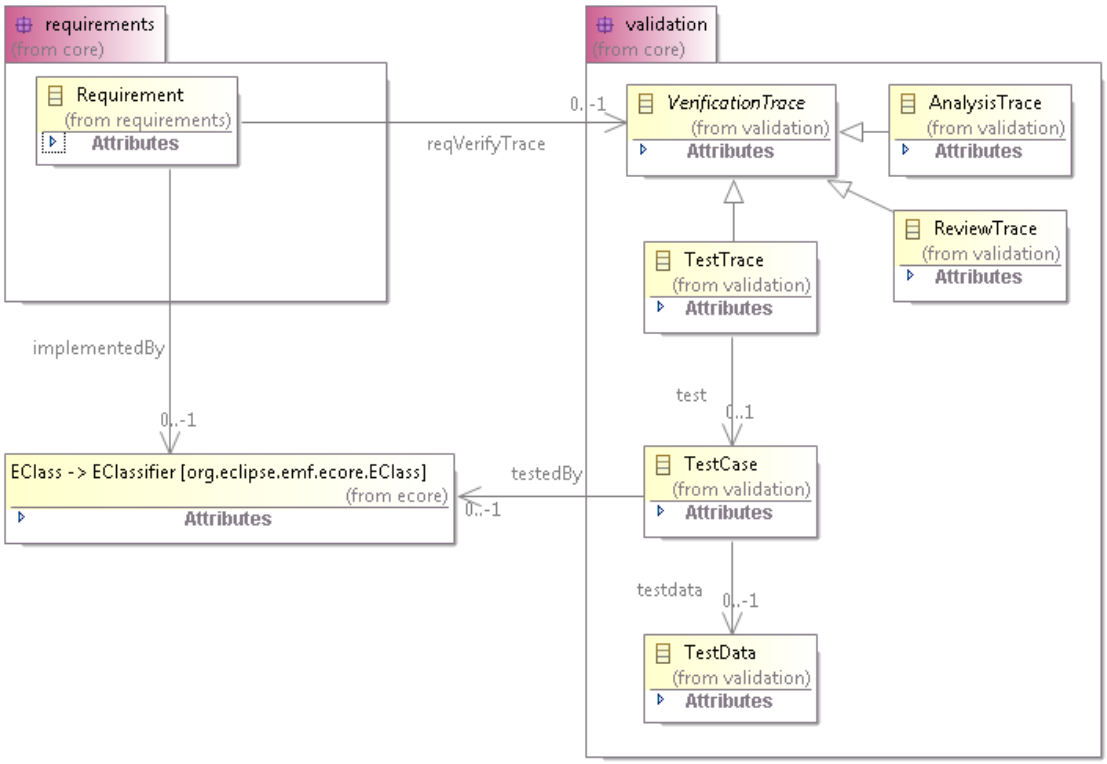

Abbildung 4.2: Anforderungs-Modell im Kontext

Wie in der Abbildung 4.2 zu sehen ist, definiert ein *Requirement* eine Beziehung *implementedBy* zum Ecore Type *EClass*. Diese ermöglicht es, Klassen im Datenmodell mit Anforderungen zu verknüpfen. Zusätzlich wurde die Klasse *TestTrace* eingeführt. Diese verbindet ein *Requirement* mit der Klasse *TestCase* aus dem Validierungs-Modell. Es kann also definiert werden, auf Basis welcher Anforderung eine Klasse implementiert wurde, was wiederum als Begründung für die Existenz dieser Klasse dient. Außerdem gibt es die Möglichkeit Anforderungen mit Testfällen (*TestCase*) zu verknüpfen, die

genutzt werden um zu überprüfen, ob die Anforderung richtig umgesetzt ist (siehe Kapitel 5) beziehungsweise ob das Datenmodell der im Testfall definierten Aspekte gerecht wird.

Außerdem erben alle Klassen vom Interface *ICommonElement*, welches in Kapitel 3.2.1 definiert wurde.

4.3.2 Import von Anforderungen

Da sowohl der Daten Modell Editor als auch der Instanz Modell Editor dasselbe Anforderungs-Modell verwenden, stellt der Austausch von Information zwischen den beiden Anwendungen kein größeres Problem dar. Zu beachten ist aber, dass die Beziehungen zu den Klassen im Datenmodell im Daten Modell Editor anders definiert sind als im Instanz Modell Editor. Im Daten Modell Editor kann ein *Requirement* die Klasse (*EClass*) direkt referenzieren, da sie ein Bestandteil der Resource ist. Im Instanz Modell Editor hingegen steht die Instanz nicht mehr direkt zur Verfügung. Die referenzierte Klasse muss beim Import der Anforderungen in den Instanz Modell Editor auf eine Klasse der Laufzeitumgebung gemappt werden. Dies wird dadurch möglich, weil der Instanz Modell Editor auf Basis des Datenmodells aus dem Daten Modell Editor entwickelt wird und somit alle Klassen Bestandteil der Laufzeitumgebung des Instanz Modell Editors sind. Über eine sogenannte *EMF Registry* ([Mer08] Kapitel 14.1.2) kann zur Laufzeit das entsprechende Datenmodell und die Klasse ermittelt werden.

Der Import von Anforderungen aus anderen Quellen erweist sich jedoch viel komplexer und hängt stark von der für die Anforderungen genutzten Anwendung ab. So bietet zum Beispiel DOORS ([IBM14b]) ein XML basiertes Format zum Austausch von Anforderungen an (RIF/ReqIF - Requirements Interchange Format), wohingegen eine Anwendung wie Word keine spezielle Unterstützung bereitstellt.

Die für diese Arbeit und für die Testfälle relevanten Anforderungen liegen in Form eines Worddokumentes vor. Dabei wird schnell deutlich das der Formalisierungsgrad für den Menschen ausreichend erscheint, jedoch eine technische Weiterverarbeitung stark erschwert ist. Zwar lassen sich die Anforderungen extrahieren, da sie in Tabellenform definiert sind und somit mittels eines dedizierten Parsers gelesen werden können, jedoch fehlen Zusatzinformationen, wie zum Beispiel der semantische Zusammenhang zwischen Anforderungen. Außerdem gehen Informationen verloren, die außerhalb der Tabellen definiert sind und den Kontext für bestimmte Anforderungen beschreiben. In 'Basiswissen Requirements Engineering' ([PR10] Kapitel 9.3.2) werden Ursachen ge-

nannt, warum Büroanwendungen so weit verbreitet sind in Projekten. Dies liegt zum einen an der hohen Verfügbarkeit und zum anderen an der einfachen Handhabung. Jedoch wird auch erläutert das “solche Werkzeuge die grundlegenden Funktionen des Requirements Management nur in geringem Umfang” [PR10] unterstützen.

4.3.3 Implementierung der Benutzeroberfläche

Um die Anforderungen im Daten Modell Editor und Instanz Modell Editor zu visualisieren und zu verwalten, wird auf Basis der in Kapitel 3 vorgestellten Funktionalität ein Editor entwickelt. Dazu wird ein neues Plug-in angelegt, welches sowohl in den Daten Modell Editor als auch in den Instanz Modell Editor integriert werden kann. Das Plug-in enthält die konkrete Implementierung für den Anforderungs-Editor, wobei die folgenden Klassen von Bedeutung sind:

- *RDEEditor* - Der Haupteditor, der es ermöglicht mehrere Editoren als Subtabs darzustellen.
- *RDEContainmentEditorPart* - Zeigt die *Requirements* und *RequirementRepositories* in einer Baum-Struktur, wobei ’contained’ Elemente als Kindelemente dargestellt werden.
- *RDEDerivationEditorPart* - Zeigt die *Requirements* in einer Baum-Struktur, wobei abgeleitete *Requirements* als Kindelemente dargestellt werden.
- *RDERefinementEditorPart* - Zeigt die *Requirements* in einer Baum-Struktur, wobei präzisierte *Requirements* als Kindelemente dargestellt werden.
- *CoreRDEContainmentContent* - Beinhaltet die Beschreibung über Aussehen und Inhalt der Tabelle für den jeweiligen Editor. Diese Klasse wird auf Basis des Notation-Modells generiert.
- *CoreRDEDerivationContent* - Beinhaltet die Beschreibung über Aussehen und Inhalt der Tabelle für den jeweiligen Editor. Diese Klasse wird auf Basis des Notation-Modells generiert.
- *CoreRDERefinementContent* - Beinhaltet die Beschreibung über Aussehen und Inhalt der Tabelle für den jeweiligen Editor. Diese Klasse wird auf Basis des Notation-Modells generiert.

Die Abbildung 4.3 zeigt die Definition des Inhalts des *RDEContainmentEditorParts* im Notation-Modell.

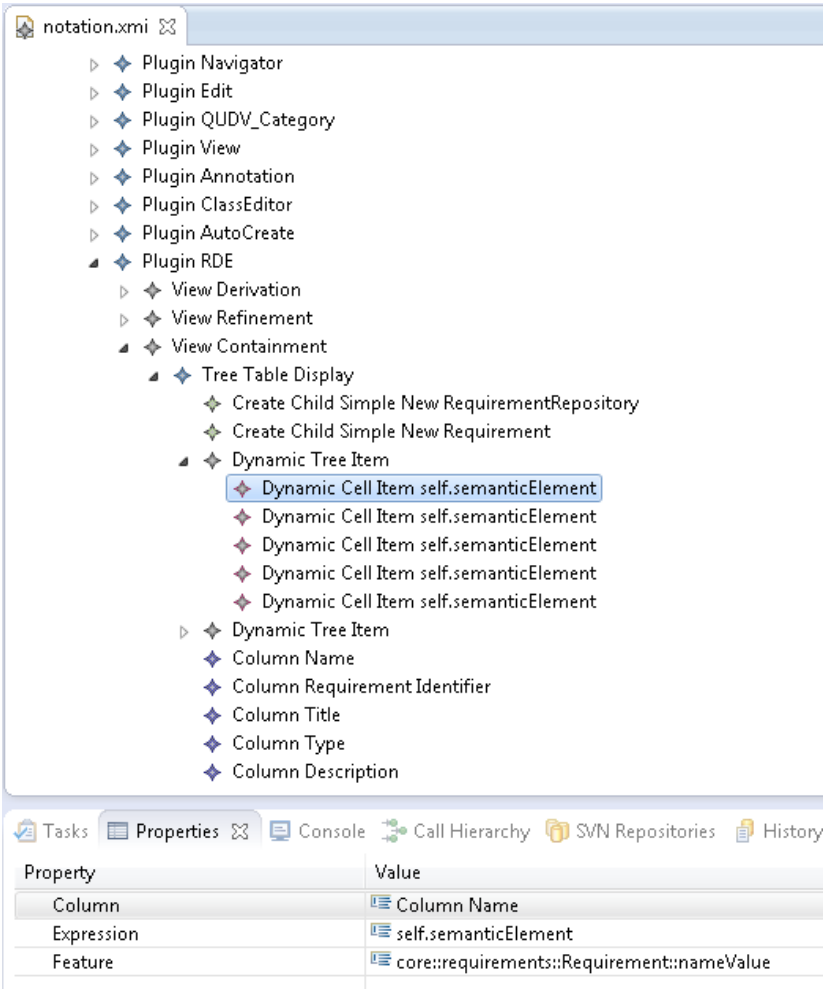

Abbildung 4.3: Beschreibung des *RDEContainmentEditorParts* im Notation-Modell

Wie in der Abbildung zu sehen ist, wird ein Plug-in namens 'RDE' definiert, welches drei View-Elemente beinhaltet, die den drei oben genannten Editoren entsprechen. Innerhalb des Containment-View-Elementes wird die Darstellungsform definiert, welche in diesem Fall eine Tree-Table-Darstellung ist. Für die Darstellungsform werden Spalten (Column) definiert, die den Tabellenspalten im Editor entsprechen. Der konkrete Inhalt der darzustellen ist, also die *Requirements* oder die *RequirementRepositories*, wird durch *Dynamic Tree Items* definiert. Diese *Dynamic Tree Items* beinhalten einen OCL-Ausdruck, der wiederum beschreibt wie die Elemente für die Zeilen in der Tabelle zu bestimmen sind. Unterhalb dieses Elements wird definiert, wie der Inhalt der einzelnen Zellen für die entsprechende Objektmenge zu ermitteln ist, beziehungsweise was in den entsprechenden Spalten angezeigt werden soll. Zusätzlich zur Beschreibung des

Aussehens und des Inhalts bietet das Notation-Modell die Möglichkeit zu definieren, welche Elemente innerhalb des Editors neu angelegt werden können. Diese Definition erfolgt mittels des Elements *Create Child Simple*, welches exakt definiert wie ein Element anzulegen ist. Wie in Kapitel 3.5 bereits erklärt, wird aus diesen Informationen Java-Code generiert, der vom Editor verarbeitet werden kann.

Zusätzlich zu der Definition im Notation-Modell und dem Anlegen der Java-Klassen wird der Editor mittels *Extension Point* in der Toolbar der Anwendung zur Verfügung gestellt. In der Abbildung 4.4 ist der Anforderungs-Editor mit einigen Anforderungen an das EGS-CC Datenmodell zu sehen.

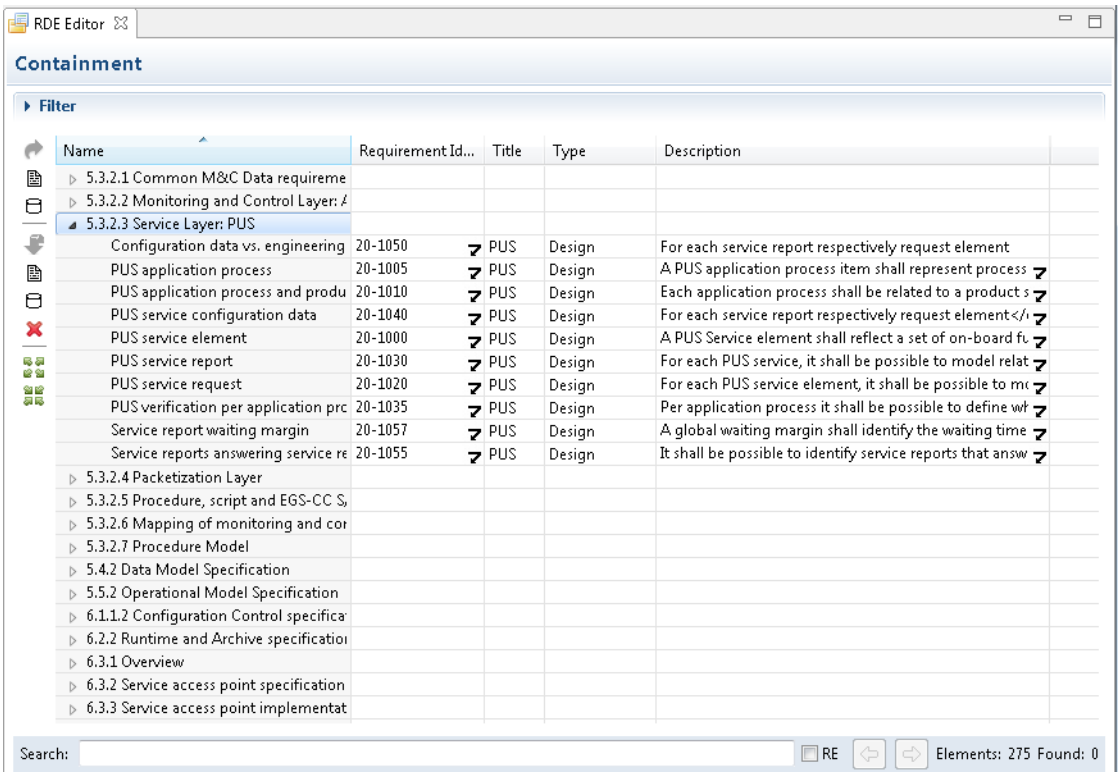

Abbildung 4.4: Anforderungs-Editor im Daten Modell Editor

4.4 Zusammenfassung

Durch die Integration des Anforderungs-Editor in den Daten Modell Editor ist es möglich Anforderungen zu verwalten und diese mit dem Datenmodell in Verbindung zu setzen. Es ist weiterhin möglich, die Anforderungen zu strukturieren, zu verfeinern sowie eindeutig über ein sogenanntes *reqIdentifier* Attribut zu identifizieren. Dies er-

möglicht es zum Beispiel, eine Anforderung in externen Dokumenten zu referenzieren. Durch die in Kapitel 3 vorgestellte Funktion des File-Modells ist es außerdem möglich externe Daten, wie zum Beispiel Word Dokumente, mit Anforderungen zu verknüpfen.

Neben diesen Funktionen bietet es sich an, das DME-Projekt mit dem Datenmodell und den Anforderungen in einem Versionsverwaltungssystem zentralisiert zu speichern, um die Aktualität und die übergreifende Verfügbarkeit der Anforderungen zu gewährleisten. Diese Funktion wird ebenfalls bereits durch den Daten Modell Editor unterstützt.

Im Instanz Modell Editor bietet der Anforderungs-Editor die Möglichkeit für den Domainexperten, Hintergrundinformationen für seine Testfälle zu erhalten. Um die Anforderungen im Instanz Modell Editor gegen Modifikation zu schützen, wird das in Kapitel 3.2.1 erläuterte *locked* Attribut verwendet.

KAPITEL 5

Verifikation und Validierung

5.1 Definition

Oftmals wird Validierung und Verifikation synonymhaft verwendet, um zu beschreiben, dass ein System einer bestimmten Menge an Kriterien genügt. Betrachtet man aber die Definitionen in CMMI ([Tea02]) und IEEE ([Wik14a]) näher:

> Verification confirms that work products properly reflect the requirements specified for them. In other words, verification ensures that "you built it right." ([Tea02])
>
> Validation confirms that the product, as provided, will fulfill its intended use. In other words, validation ensur es that "you built the right thing." ([Tea02])

so wird deutlich das es Unterschiede zwischen den zwei Methoden gibt, das Ziel aber dasselbe ist. Bei der Verifikation wird sichergestellt, dass das System den definierten Anforderungen genügt. Das System berücksichtigt alle in der Spezifikation definierten Aspekte, wie zum Beispiel bereitzustellende Funktionen, zu verwendende Technologien und rechtliche Rahmenbedingungen. Bei der Validierung hingegen wird überprüft, ob das System überhaupt das ist, was der Kunde oder Anwender will. Dabei ist aber zu berücksichtigen, dass die Anforderungen, also die Spezifikation des Systems, bereits auf den Bedürfnissen des Kunden basieren und somit ein kausaler Zusammenhang zwischen Validierung und Verifikation besteht. Das bedeutet, wenn die Anforderungen valide für

den Kunden sind, das bei erfolgreicher Verifikation des Systems, das System auch valide ist. Ist das nicht der Fall, deutet dies auf falsche oder fehlende Anforderungen hin.

Anstelle die Unterschiede zwischen Verifikation und Validierung zu analysieren, sollte man sie mehr als Sammlung von Methoden betrachten, die abhängig von Teilnehmern und der Input- und Output-Artefakte eingesetzt werden können [Ste02].

Der im nächsten Kapitel vorgestellte Prozess ist primär eine Validierung des Datenmodells. Es wird geprüft, ob das Datenmodell den Wünschen und Bedürfnissen des Domainexperten genügt. Dazu werden Testfälle für die entsprechende Domain erstellt und diese wiederum mit Anforderungen verknüpft. Die Testfälle werden danach im Instanz Modell Editor bearbeitet und durch teilweise Instanziierung des Datenmodells überprüft.

Mit der Verknüpfung und dem daraus resultierenden Zusammenhang zwischen Benutzertests und Anforderung wird außerdem eine partielle Verifikation des Datenmodells erreicht. Das ist jedoch nur dann gewährleistet, wenn die Benutzertests die in den Anforderungen definierten Aspekte abdecken.

5.2 Validierungsprozess

Wie bereits eingangs erwähnt, dient der hier vorgestellte Prozess zur (Pre-)Validierung des Datenmodells. Konkret soll für die Projektphase B der EGS-CC Initiative die Validität des Datenmodells überprüft werden. Dabei soll sichergestellt werden, dass für die Folgephasen keine oder nur sehr geringe Änderungen am Datenmodell nötig sind, da diese mit hohen Kosten verbunden wären. Neben der reinen Validierung ist dies auch eine Verifikation für die Anforderungen, die in diesem Prozess involviert sind. Der Prozess selbst wird in mehreren Iterationen erfolgen, wobei eine Iteration immer mit der Änderung des Datenmodells und einer neuen Version des Instanz Modell Editors endet. Die Änderungen am Datenmodell ergeben sich aus den Testfällen, die nicht erfolgreich waren. Die Abbildung 5.1 zeigt einen ersten Überblick über die involvierten Daten und Komponenten.

Ausgehend von den Anforderungen werden Testfälle definiert und verlinkt, die im sogenannten Validierungs-Editor (siehe Kapitel 5.4) verwaltet werden. Der Daten Modell Editor ist somit die initiale Plattform für die Definition beziehungsweise die Bereitstellung der zu testenden Anforderungen und Testfälle. Die Testfälle beinhalten dabei eine verbale Beschreibung der durchzuführenden Aktionen. Die wiederum betreffen Teile

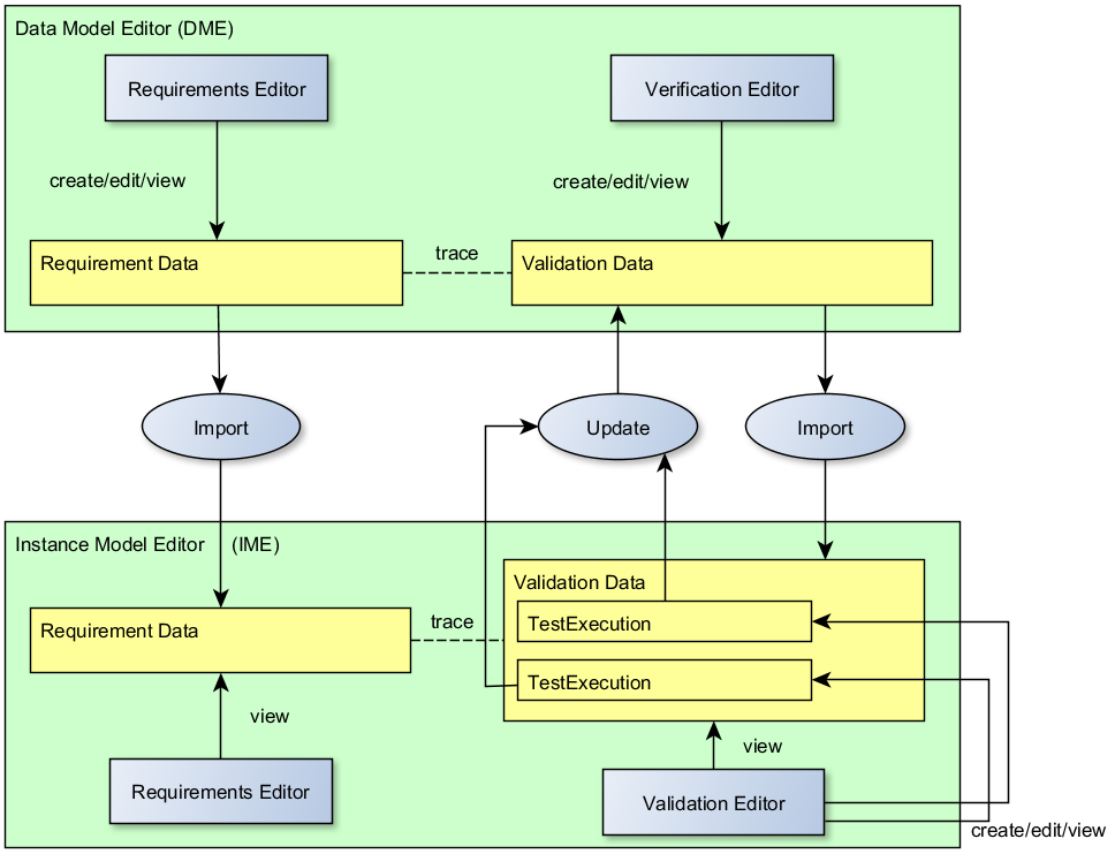

Abbildung 5.1: Überblick über die Daten und Komponenten für Validierung

des Datenmodells und sollen die Validität dieser Teile sicherstellen. Diese Konfiguration wird danach in den Instanz Modell Editor importiert. Sie dient dem Domainexperten als Basis für die Testdurchführung. Dazu bietet der Validierungs-Editor die Möglichkeit der Testdurchführung, welche wiederum eine formalisierte Beschreibung der Aktion des Anwenders ist. Die Durchführung eines Tests kann entweder positiv oder negativ enden, was zusätzlich im Validierungs-Editor definiert werden kann. Die negativ verlaufenen Tests haben Auswirkungen auf das Datenmodell, da sie mit hoher Wahrscheinlichkeit Fehler aufgedeckt haben. Nach Beendigung aller Tests beziehungsweise nach einer vordefinierten Zeitspanne werden alle Resultate zurück in den Daten Modell Editor importiert und können da analysiert werden. Je nach Ausgang der Tests können Änderungen am Datenmodell im Daten Modell Editor erfolgen. Nachdem alle Änderungen am Datenmodell erfolgt sind, wird ein neuer Instanz Modell Editor zur Verfügung gestellt (siehe Kapitel 3.6), der als Basis für die nächste Iteration der Validierung dient.

In der Abbildung 5.2 ist die erste Iteration des Validierungsprozesses dargestellt.

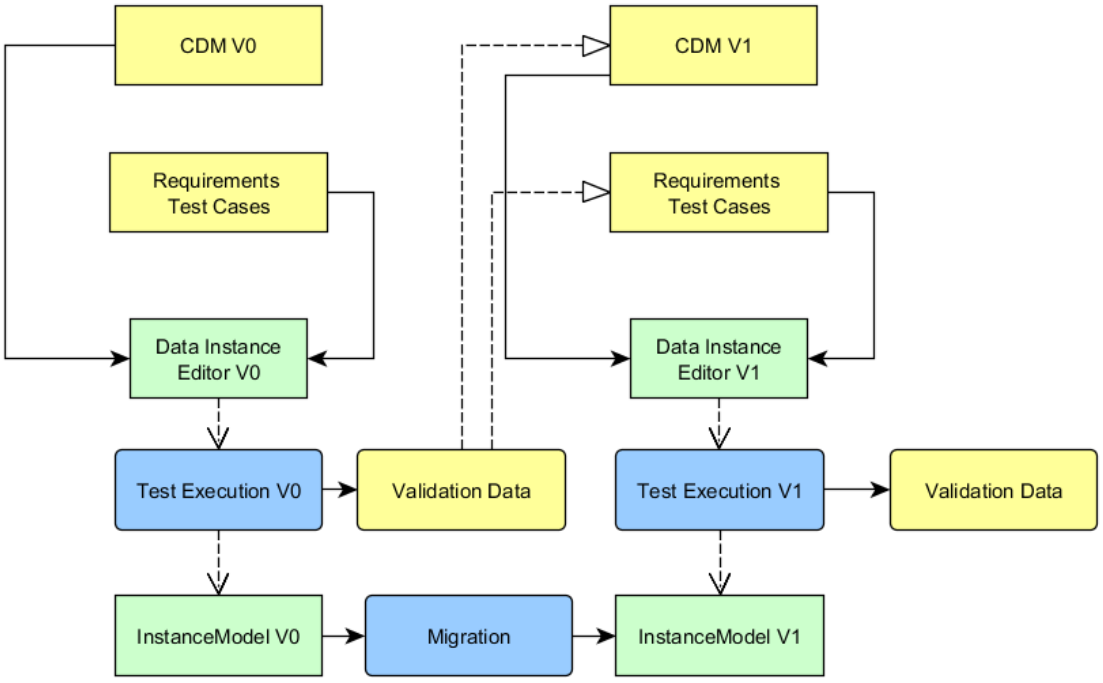

Abbildung 5.2: Erste Iteration der Validierung

Basierend auf einem Datenmodell 'V0' (CDM V0) wird ein Instanz Modell Editor 'V0' entwickelt. In den Instanz Modell Editor 'V0' werden dann die Anforderungen und Testfälle importiert. Bei der Durchführung der Tests entstehen im Instanz Modell Editor zum einen Instanz-Modelle und zum anderen Informationen über die Ergebnisse der Validierung (Validation Data). Die Instanz-Modelle enthalten dabei Daten, die für die Tests erzeugt wurden, um zu ermitteln, ob bestimmte Aspekte mit dem Datenmodell abgebildet (instanziiert) werden können. Da die Instanz-Modelle abhängig vom jeweiligen Datenmodell sind, müssen sie für eine erneute Verwendung in der nächsten Iteration der Validierung migriert werden.

Die Abbildung 5.3 zeigt drei Testfälle im Bereich PUS (Packet Utilisation Standard) während einer beliebigen Iteration. Wie in der Abbildung zu sehen ist, wurde der erste Testfall bereits getestet und das Ergebnis war negativ. Aus diesem Grund wird in einer erneuten Iteration der Testfall nochmals getestet, diesmal mit positiven Resultat. Neben diesem werden die zwei weiteren Testfälle bearbeitet, die in der vorhergehenden Iteration noch nicht getestet wurden. Das Ergebnis sind drei Testfälle mit positiven Resultat. Wobei der erste Testfall zweimal durchgeführt wurde. Damit nachverfolgt werden kann, wann ein bestimmter Test fehlgeschlagen ist, wird zu jedem Testdurchlauf die Version des Datenmodells hinterlegt. Dies ermöglicht es zu einem späteren

Zeitpunkt zu überprüfen, warum der Test mit einer bestimmten Datenmodell-Version fehlgeschlagen ist.

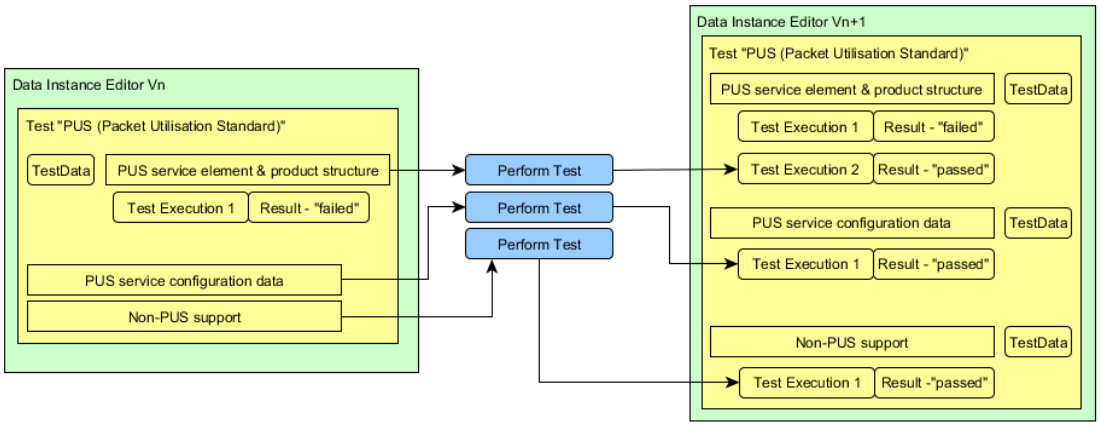

Abbildung 5.3: Bearbeiten von Testfällen

Es ist davon auszugehen, dass die Validierung von mehreren Personen (Domainexperten) durchgeführt wird, was einen weiteren wichtigen Aspekt, nämlich die Datenverteilung, innerhalb diese Prozesses zeigt. Wie bereits erwähnt, werden die Daten für die Validierung im Instanz Modell Editor importiert. Es stellt kein Problem dar, diesen Prozess für mehrere Instanz Modell Editoren zu wiederholen. Das Ergebnis sind eine Reihe von Instanz Modell Editoren mit denselben Daten für die Validierung. Ausgehend davon bearbeitet jeder Anwender aber eigene Testfälle und erzeugt dafür Ergebnisdaten. Die Abbildung 5.4 zeigt eine mögliche Verteilung dieser Informationen, die am Ende einer Iteration in den Daten Modell Editor importiert werden. Dabei wird deutlich, dass die Aggregation für den zweiten und dritten Testfall kein Problem darstellt, da die Anwender exklusive an einen der beiden Testfälle gearbeitet haben. Beim ersten Testfall wird jedoch deutlich, dass es auch zu Problemen kommen kann, wenn beide Benutzer am selben Testfall arbeiten. So sind in diesem Beispiel sogar die Ergebnisse der Tests unterschiedlich. In diesem Fall muss der Anwender, welcher für die Aggregation im Daten Modell Editor zuständig ist, handeln und gegebenenfalls muss der Test erneut durchgeführt werden.

5.3 Validierungs-Modell

Analog zum Anforderungs-Modell (siehe Kapitel 4.3.1) wird für die Daten zur Validierung ebenfalls ein EMF-Modell als Beschreibung für die Informationen verwendet.

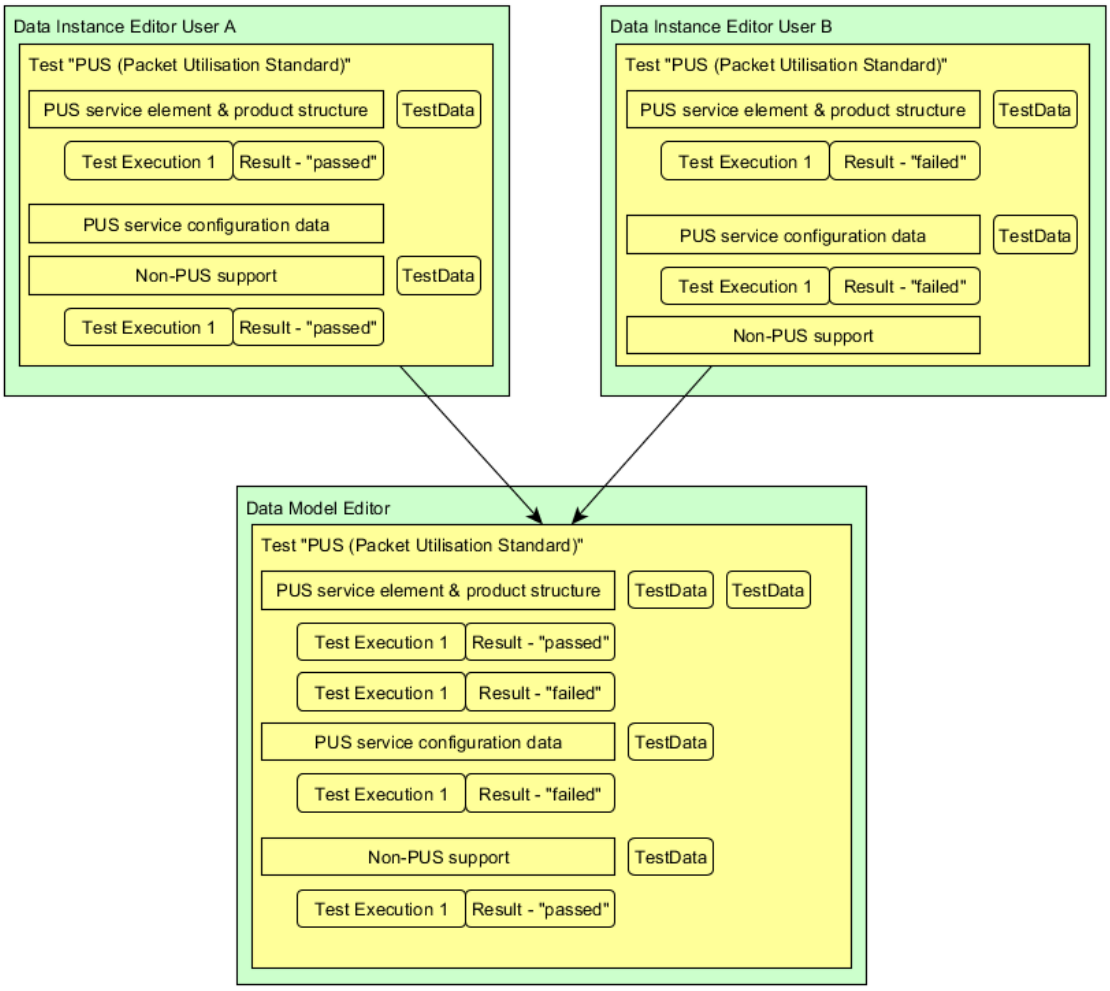

Abbildung 5.4: Zusammenführen von Testergebnissen

Dies ermöglicht einen formalisierten Austausch der Informationen zwischen Daten Modell Editor und Instanz Modell Editor. Abgeleitet ist das Validierungs-Modell aus dem *ECSS ETM 10-23* Modell ([Age12b]). Die Abbildung 5.3 zeigt einen Ausschnitt aus diesem Modell, wobei folgende Klassen von besonderer Bedeutung sind:

Test - Dient zum Gruppieren mehrerer Testfälle für unterschiedliche Domains wie zum Beispiel: Produktstruktur, PUS (Packet Utilisation Standard, MC (Monitoring and Control).

TestCase - Repräsentiert den Testfall und beinhaltet neben der Beschreibung wie der Test durchzuführen ist auch die Person/Abteilung, die den Test durchführen soll sowie weitere Informationen.

TestExecution - Repräsentiert den Vorgang beim Durchführen des Tests und bein-

haltet Informationen über den Zeitpunkt, verwendete Datenmodellversion, das Ergebnis und Erklärungen für das Ergebnis.

TestVerdict - Repräsentiert das Resultat eines durchgeführten Testfalls und kann entweder erfolgreich oder nicht erfolgreich sein.

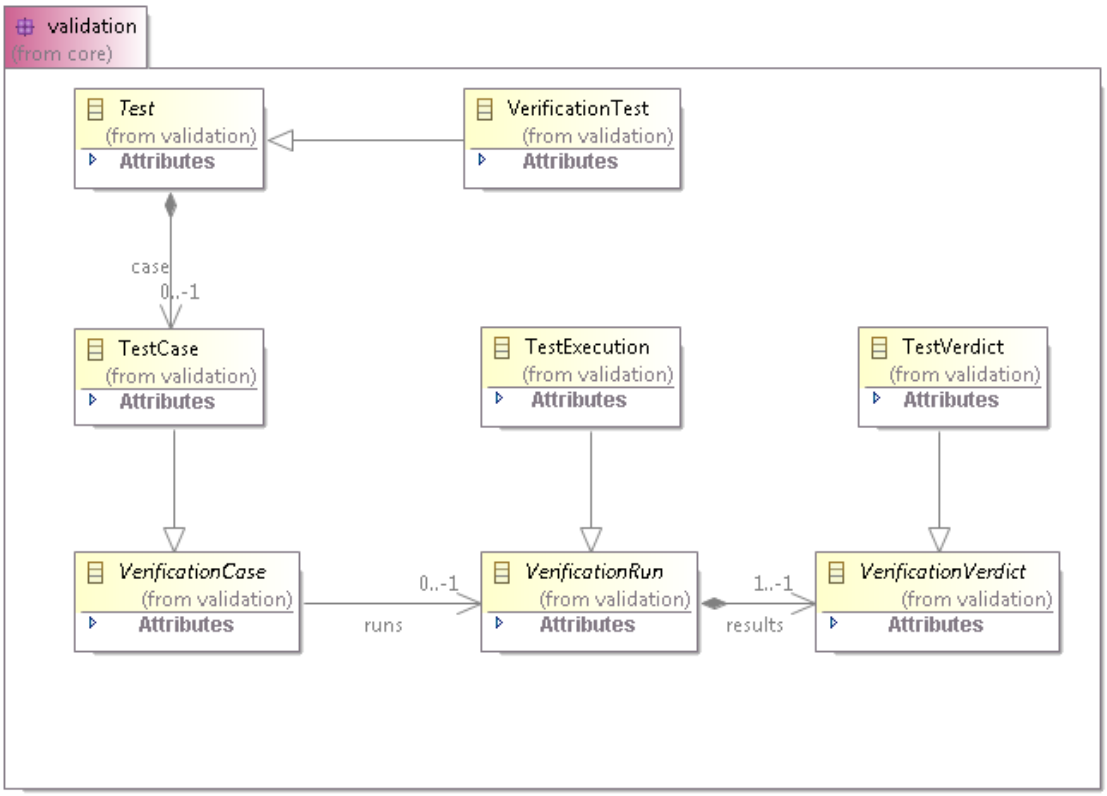

Abbildung 5.5: Validierungs-Modell

Wie bereits in Kapitel 4.3.1 erwähnt existiert eine Verbindung zwischen Anforderungen und Testfällen. Das heißt Testfälle lassen sich bestimmten Anforderungen zuordnen und dienen somit auch als Verifikation der jeweiligen Anforderung. Darüber hinaus definiert jede Anforderung Beziehungen zum Datenmodell in Form von Referenzen (*implementedBy*) auf die Klassen, die für die Umsetzung der Anforderung relevant sind. Die Abbildung 4.2 in Kapitel 4.3.1 zeigt außerdem das ein Testfall auch eine Verbindung (*testedBy*) zum Datenmodell aufweist. Die Menge der Klassen, die durch die *testedBy* Referenz spezifiziert wird, dient dazu explizit zu definieren, welche Klassen für den Test relevant sind und überprüft beziehungsweise instanziiert werden müssen. Die Menge der zu testenden Klassen ist dabei durch die Klassen, die durch die Anforderung definiert werden und mit denen der Testfall verknüpft ist, beschränkt. Das heißt die Klassen, die

durch *testedBy* definiert werden können, sind eine Submenge aller Klassen, die durch referenzierte Anforderungen per *implementedBy* vordefiniert sind. Anders ausgedrückt kann man keine Klassen testen, die nicht durch eine referenzierte Anforderung implementiert wurden. Diese Informationen können verwendet werden, um unterschiedliche Analysen durchzuführen, wie sie in Kapitel 5.5 beschrieben werden.

5.4 Benutzeroberfläche und Datentransfer

Analog zur Entwicklung des Anforderungs-Editors ist der Validierungs-Editor auf Basis der in Kapitel 3 vorgestellten Funktionalität entwickelt. Dazu wird ein zusätzliches Plug-in definiert, welches den konkreten Editor enthält. Der Inhalt sowie die Struktur des Editors wird dabei wiederum durch das Notation-Modell beschrieben. Die Abbildung 5.6 zeigt den Validierungs-Editor mit einigen Testfällen in Form einer Baumstruktur. Dabei ist zu erkennen, dass die obersten Elemente (Verifcation Test) wie MC (Monitoring and Control), FEA (Functional Electrical Architectures) und Packets (PUS - Packet Utilisation Standard) zum Gruppieren der Testfälle (*TestCase*) dienen. In der Abbildung ist außerdem zu sehen, dass für den Testfall 'CDM-PKT-100' die verknüpften Anforderungen sowie die durchgeführten Ausführungen des Tests angezeigt werden.

Wie der Anforderungs-Editor steht auch der Validierungs-Editor im Daten Modell Editor und im Instanz Modell Editor zur Verfügung. Die Testfälle werden dabei im Daten Modell Editor definiert und ein entsprechende Verlinkung zwischen Datenmodell und Anforderungen hergestellt. Nachdem das Datenmodell einen stabilen Zustand erreicht hat und alle Testfälle für die erste Iteration der Validierung definiert sind, kann der Instanz Modell Editor, wie in Kapitel 3.6 beschrieben, entwickelt und bereitgestellt werden. Danach werden alle Testfälle mittels Import in die jeweiligen Instanz Modell Editoren gebracht und können da von den jeweiligen Domainexperten bearbeitet werden. Die Datenhaltung erfolgt zu diesem Zeitpunkt dezentral. Änderungen an den Testfällen sowie Anforderungen sollten zu diesem Zeitpunkt nicht erfolgen, da Änderungen zu inkonsistenten Informationen führen.

Im Instanz Editor werden nun durch den Domainexperten die Testfälle, für die er zuständig ist, bearbeitet. Das Ergebnis dieses Prozesses ist zumindest die Definition einer *TestExecution*, welche definiert, dass der Test durchgeführt wurde und eines *TestVerdicts*, welches beschreibt, ob der Test erfolgreich oder nicht erfolgreich war. Zusätzlich zu diesen Informationen kann der Domainexperte Testdaten, also instantiierte Modelle

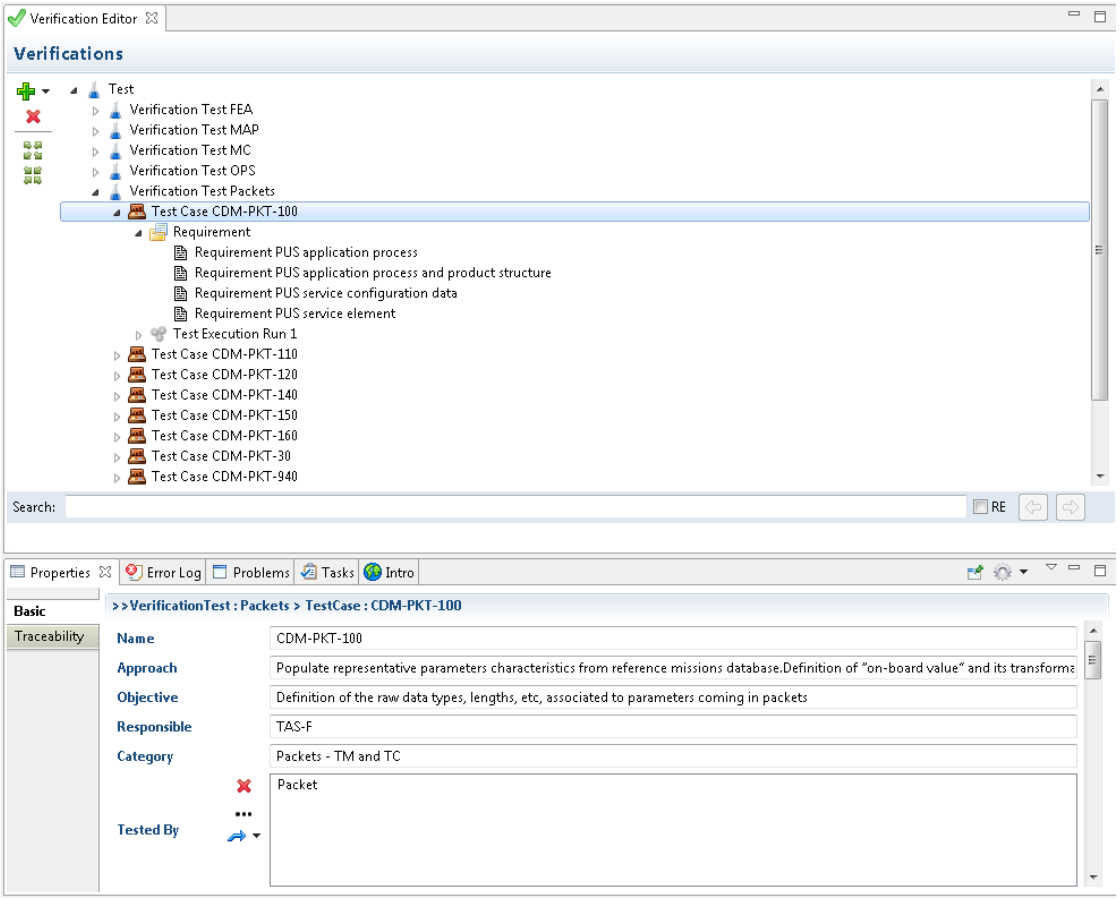

Abbildung 5.6: Validierungs-Editor

des Datenmodells mit dem Testfall verknüpfen. Diese können zum einen genutzt werden, um Fehler im Datenmodell zu reproduzieren oder sie dienen als Basis für eine erneute Durchführung dieses Testfalles.

Nachdem alle Tests beendet sind oder zu einem festgelegten Zeitpunkt werden alle Daten innerhalb des Daten Modell Editors zentralisiert. Dies geschieht durch einen Import der Daten aus den jeweiligen Projekten der Instanz Modell Editoren. Dabei werden die Testfälle im Daten Modell Editor aktualisiert und entsprechend *TestExecution* Elemente, auf Basis der Testfälle in den Instanz Modell Editoren, eingefügt. Weiterhin werden alle Testdaten auch im Kontext des Daten Modell Editors hinterlegt. Die Abbildung 5.7 zeigt den Prozess, um die Daten im Daten Modell Editor zu sammeln.

Wichtig ist dabei, dass die Testdaten nicht ohne Weiteres im Daten Modell Editor bearbeitet oder visualisiert werden können, da die Modelldefinition nicht Bestandteil der Laufzeitumgebung ist. Sie ist nur als Datenmodell im Projekt innerhalb der DME-Umgebung vorhanden. Trotzdem ist es wichtig die Testdaten zu sammeln, da sie für

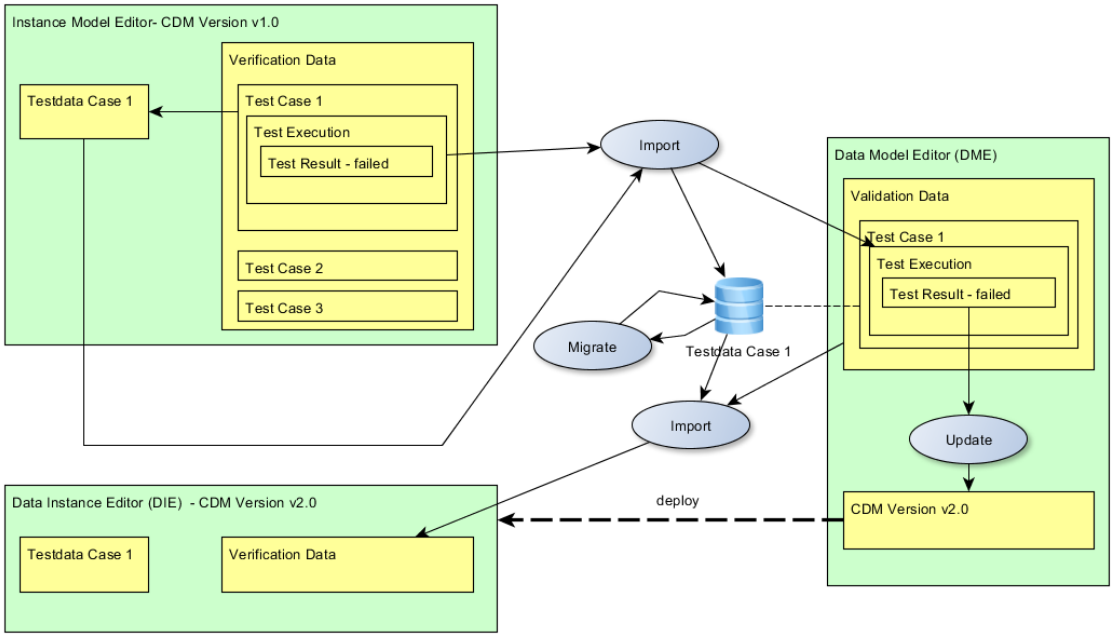

Abbildung 5.7: Zusammenführen aller Daten im Daten Modell Editor

die nächste Iteration migriert werden müssen. Die Migration wird dann nötig, falls sich auf Basis der Testergebnisse Änderungen am Datenmodell ergeben. Da die Änderungen am Datenmodell auch Einfluss auf die instanziierten Elemente in den Testdaten haben können, müssen diese angepasst werden und zwar sobald das Datenmodell für die nächste Iteration der Validierung freigegeben ist. Analog zum initialen Verteilen der Daten werden in den Folgeschritten alle Daten in den Instanz Modell Editoren importiert. Hinzu kommt dabei, dass die Testfälle bereits *TestExecution* Elemente enthalten können und die migrierten Testdaten auch importiert werden.

Dieser Prozess wiederholt sich dann solange bis alle Testfälle erfolgreich durchgeführt wurden.

5.5 Analysen

Wie bereits in Kapitel 4.3.1 und 5.3 erklärt, existieren mehrere Verknüpfungen zwischen Datenmodell, Anforderungen und Testfällen. Die Abbildung 5.8 zeigt einen Ausschnitt dieser Zusammenhänge im *Modell Explorer*, auf deren Basis unterschiedliche Analysen erfolgen können.

Betrachten wir zunächst die *implementedBy* Beziehung zwischen Anforderung und ei-

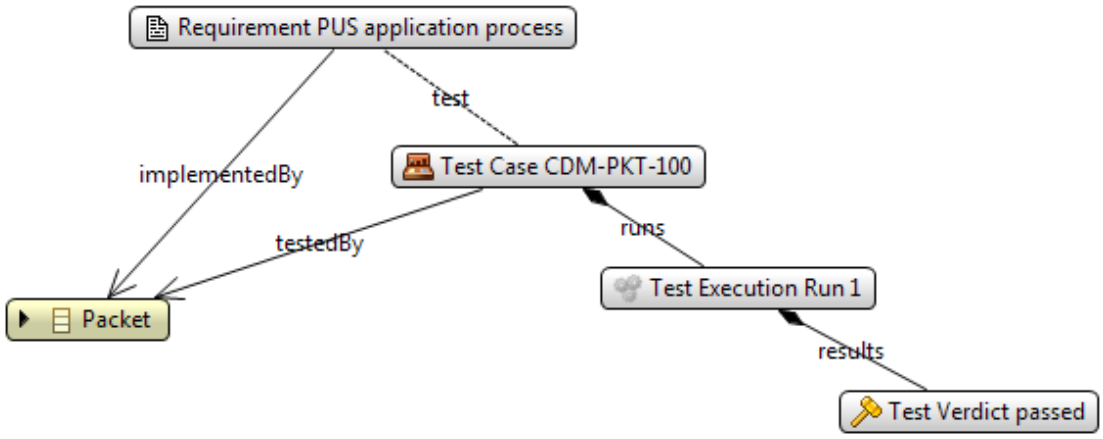

Abbildung 5.8: Zusammenhang zwischen Datenmodell, Anforderungen und Testfällen

ner Klasse im Datenmodell. Mittels dieser Beziehung kann analysiert werden, ob alle Klassen auf der Basis einer Anforderung definiert wurden. Es wird geprüft, ob mindestens eine Anforderung die betreffende Klasse referenziert. Das kann nützlich sein, um die Existenz einer Klasse zu rechtfertigen. Im Umkehrschluss bedeutet das für Klassen, die nicht auf einer Anforderung basieren, dass sie unter Umständen nicht notwendig beziehungsweise eine 'Goldrandlösung' darstellen.

Eine zweite mögliche Analyse betrifft die Testfälle in Verbindung mit den Klassen des Datenmodells. Mit der expliziten Definition der Klassen durch die *testedBy* Beziehung für einen entsprechenden Testfall lässt sich analysieren, ob im Instanz Modell Editor überhaupt die entsprechenden Klassen instanziiert wurden und Bestandteil der Testdaten sind.

Zusätzlich ist es durch die Verknüpfung von Testfällen und Anforderungen auch möglich zu überprüfen, ob alle auf der Basis der Anforderung definierten Klassen getestet wurden.

5.6 Zusammenfassung

Der hier aufgezeigte Prozess auf Basis eines Prototyps (Instanz Modell Editor) stellt eine mögliche Methode zur Überprüfung des Datenmodells dar. Dabei wird durch die Bereitstellung des Instanz Modell Editors die Möglichkeit gegeben, Domainexperten einzubeziehen und mit realen Daten zu arbeiten. Der Prozess an sich schließt aber keine anderen Methoden aus. So ist es zum Beispiel auch denkbar, die Testfälle durch Reviews

zu ersetzen beziehungsweise zusätzlich zu *TestCases* zu definieren. Diese könnten dann zur Verifikation der Anforderungen im Daten Modell Editor verwendet werden, ohne die Notwendigkeit einen Prototyp zu entwickeln.

Durch die Bereitstellung des Validierungs-Editors im Instanz Modell Editor wird außerdem erreicht, dass die Ergebnisse der Validierung formalisiert und strukturiert erfasst werden können. Zusätzlich stehen dem Anwender weitere Informationen in Form der Anforderungen im Anforderungs-Editor zur Verfügung.

KAPITEL 6

Zusammenfassung

Wie bereits am Anfang der Arbeit dargestellt, spielen Modelle eine entscheidende Rolle im Engineering-Bereich. Das Erstellen und Verwalten solcher Modelle ist also von großer Bedeutung und wird durch unterschiedliche Anwendungen unterstützt. Die in dieser Arbeit vorgestellte Anwendung (Daten Modell Editor) ist dabei primär für das Management von Datenmodellen zuständig, kann aber auch für andere Modelltypen erweitert werden. Bei der Entwicklung des Daten Modell Editors und der Bereitstellung von zusätzlichen Komponenten wird deutlich, dass auch in der Softwareentwicklung der Einsatz von Modellen sinnvoll ist. Durch die Formalisierung auf Basis von Modellen wird zum Beispiel der Austausch von Informationen zwischen Anwendungen vereinfacht.

Durch den Einsatz der Eclipse Technologien, die in Kapitel 2 erläutert wurden, ist es außerdem möglich Anwendungskomponenten zu entwickeln, die in einem hohen Maße wieder verwendet werden können. Außerdem wird durch EMF die Effizienz bei der Entwicklung von Anwendungen auf Basis von Modellen erhöht, da das Schreiben von 'Boilerplate Code' entfällt. Zusätzlich kann durch die Generierung mithilfe geeigneter Template-Sprachen ein Teil der Applikation selbst generiert werden (siehe Kapitel 3.5), indem man die Definition der Anwendung in ein Modell verlagert. Dies hat jedoch auch Nachteile, denn je nach Komplexität der Anwendungsteile steigt auch die Komplexität des Modells. Das führt dazu, dass es passieren kann, dass der Zeitaufwand zum Definieren der Anwendungsteile mittels Modell den Zeitaufwand des einfachen Implementierens um ein Vielfaches übersteigt.

In Kapitel 3 wurde die Basisarchitektur, der Daten Modell Editor und der Instanz Modell Editor näher erläutert. Sie bilden dabei die Basis für die Integration zweier

neuer Komponenten für das Anforderungsmanagement und die Validierung. In diesem Kapitel wurde außerdem auf bereits integrierte Funktionen eingegangen. Das sind zum Beispiel der Umgang mit Engineering-Parametern und OCL-Ausdrücken. Weiterhin wurde ein Überblick gegeben, welche Modelle in der bestehenden Architektur verwendet werden und welche Rolle sie in den Anwendungen spielen. Es hat sich gezeigt, dass die bestehende Architektur die Integration von neuen Komponenten vereinfacht, da bereits viele Funktionen vorhanden sind und wiederverwendet werden können.

Das Anforderungsmanagement und die Implementierung des Anforderungs–Editors wurde in Kapitel 4 erörtert. Dabei ist zu erwähnen, dass nur ein kurzer Überblick über das Management von Anforderungen und Vorgehensmodelle gegeben wurde. Der Fokus liegt in diesem Kapitel auf der Integration des Anforderungs-Editors, dem Bezug zum Datenmodell und der Validierung. Der Anforderungs-Editor selbst bietet dabei die Möglichkeit, Anforderungen zu erfassen und zu strukturieren. Zusätzlich wurde erläutert, warum Anforderungen nicht in Standard-Büroanwendungen verwaltet werden sollten und welche Probleme beim Import auftreten können. Anzumerken ist hier, dass im Moment nur Klassen mit Anforderungen verknüpft werden können. Jedoch existieren noch weitere Aspekte wie zum Beispiel Engineering-Parameter oder OCL-Ausdrücke, die für Anforderungen auch relevant sein können.

Kapitel 5 erläutert den für diese Arbeit relevanten Validierungsprozess und beschreibt die Integration des Validation Editors. Die Validierung des Datenmodells erfolgt dabei durch den Einsatz des Instanz Modell Editors. Für den Datenaustausch der Ergebnisse der Validierung wird ähnlich wie für die Anforderungen ein EMF-Modell verwendet, welches ebenfalls in diesem Kapitel beschrieben wurde. Wichtig ist hier, dass die vorgestellte Methode zur Validierung andere Methoden nicht ausschließt. Durch Erweiterung des Validierungs-Modells lassen sich auch andere Methoden abbilden. Es ist zum Beispiel denkbar, neben den Testfällen auch Reviews zu definieren, die zur Verifikation der Anforderungen genutzt werden und nur unter Verwendung des Daten Modell Editors stattfinden.

Thesen

- Eclipse bietet sich als Plattform für die Entwicklung von komplexen Anwendungen (RCP) an.
- Das Eclipse Modeling Framework (EMF) bietet sich bei der Arbeit mit Modellen und als Basis für die Codegenerierung für Anwendungen an.
- Mit Hilfe von OCL lassen sich komplexe semantische Zusammenhänge innerhalb von Modellen beschreiben und validieren.
- Die von ScopeSET entwickelte Architektur lässt sich einfach erweitern und kann als Basis für die Integration von neuen Engineering-Aspekten genutzt werden.
- Das Anforderungsmanagement ist bei agilen Vorgehensmodellen meist prozessbegleitend, wohingegen bei streng wasserfall-orientierten Modellen das Anforderungsmanagement nur in frühen Phasen durchgeführt wird.
- Das Erfassen und Verwalten von Anforderungen muss in dedizierten Anwendungen erfolgen, um eine effizientes Management der Informationen zu gewährleisten.
- Durch die Integration unterschiedlicher Aspekte, wie zum Beispiel Datenmodell, Anforderungen und Validierungsdaten in eine Anwendung lässt sich eine effiziente Verfolgung zwischen den Informationen realisieren.
- Die Verknüpfung von Anforderungen, Klassen des Datenmodells und Validierungsdaten ermöglicht es unterschiedliche Analysen durchzuführen, die der Qualitätssicherung dienen.
- Durch den Einsatz eines Prototyps kann die Validierung in Verbindung mit Domainexperten optimiert werden.
- Die Entwicklung eines Software-Prototyps ist heutzutage nicht mehr so aufwendig wie vor 10 Jahren.

Abkürzungsverzeichnis

AST	–	Abstract Syntax Tree
AWT	–	Abstract Window Toolkit
CGF	–	Code Generation Framework
DME	–	Daten Modell Editor (Data Model Editor)
EGS-CC	–	European Ground Systems - Common Core
EMF	–	Eclipse Modeling Framework
GEF	–	Graphical Modeling Framework
GMF	–	Graphical Modeling Framework
GUI	–	Graphical User Interface
IDE	–	Integrated Development Environment
IME	–	Instanz Modell Editor (Instance Model Editor)
MOF	–	Meta Object Facility
MVC	–	Modell View Controller
OCL	–	Object Constraint Language
RCP	–	Rich Client Plattform
SWT	–	Standard Widget Toolkit
XML	–	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Aac14] Aachen, IKA: *Antriebsstrangs eines Elektrofahrzeugs*. http://www.ika.rwth-aachen.de/forschung/veroeffentlichung/1996/16.-17.09/index.php. Version: 2014. – [Online; Stand 14. Januar 2014]

[Age12a] Agency, European S.: *Virtual Spacecraft Design*. http://www.vsd-project.org/. Version: 2012. – [Online; Stand 14. Januar 2014]

[Age12b] Agency, European S.: *VSEE Metamodel*. http://www.vsd-project.org/vsee-metamodel/. Version: 2012. – [Online; Stand 14. Januar 2014]

[Age14a] Agency, European S.: *About EGS-CC*. http://www.egscc.org/about.html. Version: 2014. – [Online; Stand 14. Januar 2014]

[Age14b] Agency, European S.: *ESA Packet-Utilization Standard (PUS)*. http://www.integ-europe.com/news. Version: 2014. – [Online; Stand 14. Januar 2014]

[Age14c] Agency, European S.: *The European Ground Systems Common Core (EGS-CC) Initiative*. http://www.spaceops2012.org/proceedings/documents/id1282732-Paper-001.pdf. Version: 2014. – [Online; Stand 14. Januar 2014]

[All14] Alliance, OSGi: *OSGi Specifications*. http://www.osgi.org/Specifications/HomePage. Version: 2014. – [Online; Stand 14. Januar 2014]

[Bec14] Beck, Bryan: *Erweiterung des Data Model Editors (DME) zur Unterstützung modellgetriebener Prozessentwicklung*, Westsächsische Hochschule Zwickau, Diplomarbeit, 2014

[Cla11] Clayberg, Dan Rubel; Jaime Wren; E.: *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011

[Ecl14a] ECLIPSE: *Eclipse Modeling Framework Project (EMF).* `http://www.eclipse.org/modeling/emf/`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Ecl14b] ECLIPSE: *Graphiti - a Graphical Tooling Infrastructure.* `http://www.eclipse.org/graphiti/`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Ecl14c] ECLIPSE: *The two Eclipse OCLs.* `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FCompleteOCLTutorial.html`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Ecl14d] ECLIPSE: *Xtext.* `http://www.eclipse.org/Xtext/`. Version: 2014. – [Online; Stand 14. Januar 2014]

[For14] FORCE, SysML 1.2 Revision T.: *Quantities, Units, Dimensions, Values (QUDV).* `http://www.omgwiki.org/OMGSysML/doku.php?id=sysml-qudv:quantities_units_dimensions_values_qudv`. Version: 2014. – [Online; Stand 14. Januar 2014]

[IBM14a] IBM: *Build metamodels with dynamic EMF.* `http://www.ibm.com/developerworks/library/os-eclipse-dynamicemf/`. Version: 2014. – [Online; Stand 14. Januar 2014]

[IBM14b] IBM: *DOORS.* `http://www-03.ibm.com/software/products/de/ratidoor`. Version: 2014. – [Online; Stand 14. Januar 2014]

[KB]

[Mat14] MATHOI, Thomas: *Mehr Agilität in Lehre und Forschung bitte...* `http://www.mathoi.eu/cms/2014/04/17/mehr-agilitaet-in-lehre-forschung-bitte/#more-943`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Mer08] MERKS, Dave Steinberg; Frank Budinsky; Marcelo Paternostro; E.: *Eclipse Modeling Framework (EMF), Second Edition.* Addison-Wesley Professional, 2008

[OMG14] OMG: *Unified Modeling Language (UML).* `http://www.uml.org/`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Ost14] OSTMOTORRAD: *MZ Schaltplan*. `http://www.ostmotorrad.de/mz/es/img/es_125_schaltplan.jpg`. Version: 2014. – [Online; Stand 14. Januar 2014]

[PR10] POHL, Klaus ; RUPP, Chris: *Basiswissen Requirements Engineering*. dpunkt.verlag, 2010

[Rup09] RUPP, Chris: *Requirements-Engineering und -Management*. Hanser, 2009

[Sco14] SCOPESET: *ScopeSET GmbH*. `http://www.scopeset.de/?lang=de_de`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Ste02] STEVE: *The difference between Verification and Validation*. `http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation`. Version: 2002. – [Online; Stand 14. Januar 2014]

[Ste10] STEYER, Manfred: *Agile Muster und Methoden*. Entwickler.Press, 2010

[Tea02] TEAM, CMMI P.: *CMMI for Software Engineering*. `http://www.sei.cmu.edu/reports/02tr029.pdf`. Version: 2002. – [Online; Stand 14. Januar 2014]

[Wik14a] WIKIPEDIA: *Institute of Electrical and Electronics Engineers*. `http://de.wikipedia.org/wiki/Institute_of_Electrical_and_Electronics_Engineers`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Wik14b] WIKIPEDIA: *International Requirements Engineering Board*. `http://en.wikipedia.org/wiki/International_Requirements_Engineering_Board`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Wik14c] WIKIPEDIA: *OSGi*. `http://de.wikipedia.org/wiki/OSGi`. Version: 2014. – [Online; Stand 14. Januar 2014]

[Wik14d] WIKIPEDIA: *Vorgehensmodell*. `http://de.wikipedia.org/wiki/Vorgehensmodell`. Version: 2014. – [Online; Stand 14. Januar 2014]

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlich und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Zwickau, 10. Juni 2014