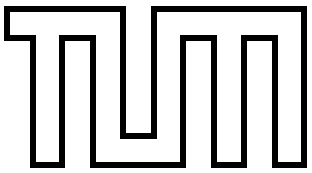


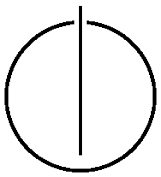
FAKULTÄT FÜR INFORMATIK

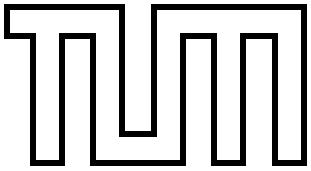
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Entwurf einer Domänenspezifischen Sprache für das Design von Prozessmodellen auf Basis des V-Modell XT Metamodells

Eugen Wachtel





FAKULTÄT FÜR INFORMATIK

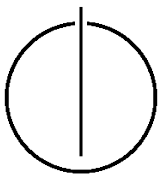
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Entwurf einer Domänenspezifischen Sprache für das Design von Prozessmodellen auf Basis des V-Modell XT Metamodells

Design of a Domain Specific Language for designing of process models based on the V-Modell XT Metamodell

Bearbeiter:	Eugen Wachtel
Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Marco Kuhrmann, Georg Kalus
Abgabedatum:	Juni 15, 2010



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 3. Juni 2010 Eugen Wachtel

Danksagung

Mein herzlicher Dank gilt zunächst meinen Betreuern Dr. Marco Kuhrmann und Georg Kalus für die hervorragende Leitung und die ergebnisreichen Gespräche. Weiterhin danke ich meiner Familie und meinem Freundeskreis für die Unterstützung während meines gesamten Studiums.

Zusammenfassung

Die Modellierung von Prozessmodellen ist eine schwere und komplexe Disziplin. Eine entsprechende Unterstützung für den Prozessingenieur ist zur Erleichterung der Modellierung selbst, aber auch für die Nachvollziehbarkeit sowie die Qualität des Modells ausschlaggebend. Insbesondere die Qualität kann über eine entsprechende Werkzeugunterstützung sichergestellt werden, indem bereits zur Designzeit eine Überprüfung der Modellkonsistenz stattfindet. Für das V-Modell XT wollen wir in dieser Arbeit eine derartige Werkzeugunterstützung mittels einer domänenspezifischen Sprache erstellen, so dass Modellinstanzen, die auf dem V-Modell XT Metamodell basieren, erstellt und bearbeitet werden können.

Abstract

The definition of process models is a challenging and complex task. Process engineers need comprehensive support for the modeling itself, but also to facilitate replicability and to ensure the quality of the model. Especially the quality can be ascertained by validating the model's consistency at design time. In this work, we aim at implementing such a tool support by using a domain specific language (DSL), so that instances of the V-Modell XT Meta-Model can be created or edited within the DSL.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Prozessmodelle sind als Vorgehensmodelle[1] der Softwareentwicklung sehr komplex. Die Konzeption beziehungsweise Verwaltung von Prozessmodellen ist eine schwere Aufgabe, so dass eine werkzeugtechnische Unterstützung dem Prozessingenieur bei der Modellierung behilflich sein muss.

Beim V-Modell XT, dem Standardvorgehensmodell der Bundesrepublik Deutschland zur Entwicklung von IT-Systemen, wird das Prozessmodell mithilfe des V-Modell XT Editors bearbeitet, der seinerseits dem Prozessingenieur auf die folgende Art und Weise Fehler vermittelt:

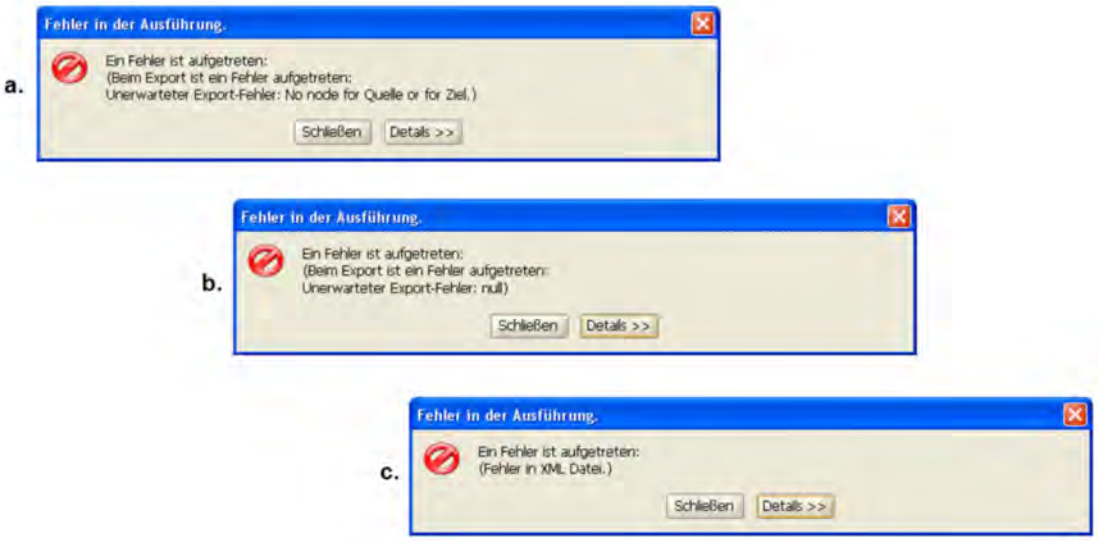


Abbildung 1.1: Fehlermeldungen des V-Modell XT Editors

Hinter den kryptischen Fehlermeldungen[2] (siehe Abbildung 1.1) verbergen sich einfache Fehler, die bei der Modellierung eines derartig komplexen Modells ohne weiteres passieren können.

Im Fall a haben wir eine Beziehung zwischen Elementen angelegt, die derartig nicht in Beziehung gestellt werden dürfen. Im Fall b haben wir vergessen eine notwendige Beziehung anzulegen, und schließlich sind wir im Fall c bereits beim Ladevorgang gescheitert, da in der Quelldatei ein Elementname falsch geschrieben wurde. Die Fehlermeldungen selbst geben uns weder Aufschluss darauf, wo konkret der Fehler zu suchen ist noch was genau als Fehlerursache zu verstehen ist, so dass zur Lösung solcher Probleme der Prozessingenieur im schlimmsten Fall alle Elemente und Beziehungen des Modells sowie die hierdurch definierten Abläufe überprüfen muss.

Fehlerfindung und Feedback. Betrachten wir nun insbesondere noch die Fälle a und b, so muss man zudem auf die verwendete Fehlererkennungsmöglichkeit eingehen, die über den so genannten *Export* gegeben ist. Der Export erzeugt die V-Modell XT Dokumentation und bricht wegen unserer Fehler mit oben gezeigten Meldungen ab. Bedenkt man an dieser Stelle, dass der Export-Vorgang mitnichten auch mehrere Minuten dauern kann und zudem auch nicht alle Fehler

1 In dieser Arbeit setzen wir voraus, dass die Bedeutung von Vorgehensmodellen in der Softwareentwicklung bekannt ist. Prominente Beispiele von Vorgehensmodellen sind das V-Modell XT [FHKS09] und der Rational Unified Process [Kru03].

2 Die hier nicht gezeigten Details der Fehlermeldungen geben dem Prozessigenieur keine weitere Hilfe. Vollständigkeitshalber haben wir sie jedoch in Anhang A mit angefügt.

findet, so ist die aktuelle Werkzeugunterstützung für das V-Modell XT in der Hinsicht der Fehlerfindung mangelhaft. Sie lässt Fehler zu und bietet auch keine sinnvolle Möglichkeit vorhandene Fehler zu finden, zeigt uns aber gleichzeitig, dass es nicht einfach ist, komplexe Prozessmodelle gebührend softwaretechnisch zu unterstützen.

Sind denn eigentlich Fehler in Prozessmodellen wirklich so schlimm und ist die Diskussion der Werkzeugunterstützung überhaupt wichtig? Dazu muss man bedenken, dass Prozessmodelle wie das V-Modell XT sehr umfangreich sind und gleichzeitig sehr komplexe innere Abläufe aufweisen:

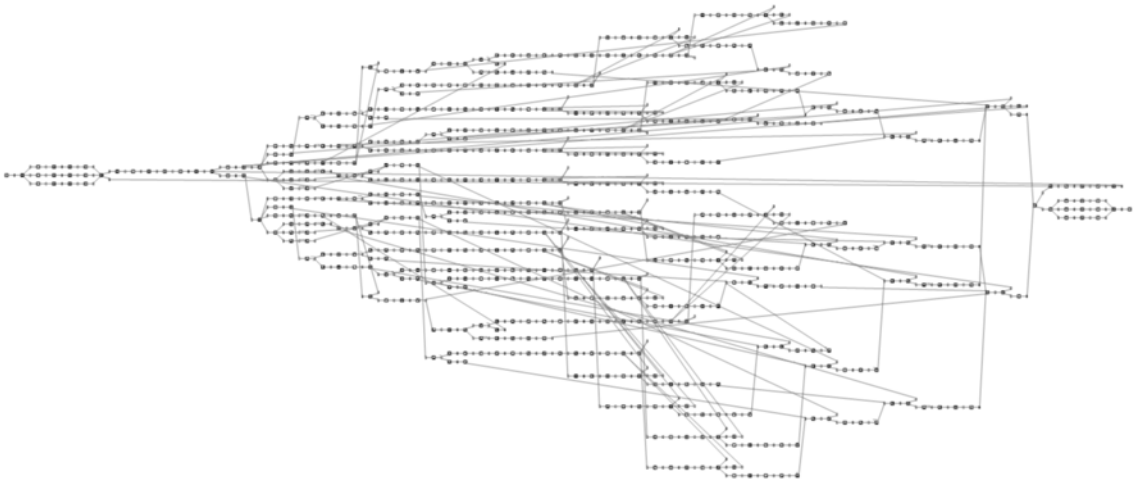


Abbildung 1.2: Unkonfigurierte Projektdurchführungsstrategie im V-Modell XT

Die Abbildung 1.2 zeigt alle möglichen Abläufe in einer unkonfigurierten Projektdurchführungsstrategie in einer Petrinetz-Darstellung [WKK09b] und verdeutlicht dabei wie schwierig die Aufgabe des Prozessingenieurs bei der Modellierung von Prozessmodellen sein kann. Zudem muss der Einsatzzweck von Vorgehensmodellen betrachtet werden. Diese stellen eine bewehrte Vorgehensweise zur IT-Entwicklung dar und werden somit für die Planung von Projektabläufen in der Softwareentwicklung eingesetzt. Findet sich nun ein kritischer Fehler bereits im Vorgehensmodell (das kann tatsächlich auch eine falsch definierte oder fehlende Beziehung sein), so kann das in der Anwendung bei einem IT-Projekt insbesondere in einem späteren Stadium gravierende Folgen haben. Nun ist aber auch klar, dass eine Werkzeugunterstützung nicht in der Lage sein kann inhaltliche Fehler in den definierten Prozessen zu identifizieren, allerdings ist es sinnvoll zumindest alle Fehler zu erkennen und zurückzumelden, die erlaubte Strukturen und Abläufe verletzen (siehe oben gezeigte Fehler). Wir beschäftigen uns in dieser Arbeit somit mit einer Werkzeugunterstützung, die sich insbesondere an die Prozessingenieure richtet.

Adäquate Werkzeugunterstützung Wie kann nun eine sinnvolle Werkzeugunterstützung aussehen und welche Funktionalitäten sind dabei wichtig? Zunächst liegt das Ziel einer jeden Werkzeugunterstützung in der Minderung der Bearbeitungskomplexität, so dass Vorgehensmodelle zumindest vom softwaretechnischen Aspekt einfach erweitert beziehungsweise angepasst werden können. Für die Qualität ist eine Werkzeugunterstützung auch mit ausschlaggebend, sie kann zumindest im Sinne der erlaubten Strukturen und Abläufe konsistenzsichernd eingesetzt werden, so dass bestehende Fehler im Modell mit Quelle und Ursache erkannt sowie neue Fehler erst gar nicht zugelassen werden. Das steigert die Qualität des Modells und erhöht zudem auch die Benutzerfreundlichkeit. Weiterhin kann eine Werkzeugunterstützung auch bei der Erfassung der Vorgänge innerhalb eines Prozessmodells helfen, indem definierte Abläufe beispielsweise auch grafisch dargestellt werden.

Bei der Definition einer Werkzeugunterstützung für ein Prozessmodell müssen wir auch bedenken, dass ein Prozessmodell eine Basis hat, die über Strukturen zur Definition von Prozessmodellen verfügt. Somit müssen wir uns auch mit den zwei zugehörigen Modellierungsbereichen beschäftigen:

- *Prozessmetamodellierung*: Das Prozessmetamodell weist Basisstrukturen auf anhand deren

Prozessmodelle instanziiert werden können.

- *Prozessmodellierung*: Das Prozessmodell ist eine Instanz des Prozessmetamodells.

In der Praxis unterliegen nicht nur die Modelle eines Prozessmodells (z.B.: V-Modell XT) sondern auch die zugrunde liegenden Metamodelle (z.B.: V-Modell XT Metamodell) einem ständigen Wandel. Im Bereich der Metamodelle sind die Änderungen sicherlich nicht so häufig wie im Modellbereich, allerdings sind diese *kritischer*, denn neben den Modellen muss auch ihre Werkzeugunterstützung (es sei denn diese ist allgemein konzipiert und automatisch bei Anpassungen kompatibel) angepasst werden, um ein in der aktuellen Version konsistentes Prozessmodell samt Modellierungswerkzeug zu erhalten.

Bei der Erstellung einer Werkzeugunterstützung ist folglich das Problem der Anpassbarkeit des Werkzeugs bezüglich der Änderungen am Metamodell von großer Bedeutung. Diese Problematik lehnt in der Regel eine direkte Softwareumsetzung des Prozessmetamodells ab (diese wäre allerdings bei rein statischen Metamodellen möglich) und fordert erweiterte Wekzeugkonzepte. Zudem lassen sich weitere Anforderungen an die Werkzeugunterstützung formulieren, die entsprechend thematisiert werden müssen. Dazu zählen:

- Validierung (Fehlerfindung, Fehlervorbeugung)
- Abhängigkeitsdarstellung
- Erweiterbarkeit (Plugin-Konzept)
- Grafische Verwaltungsmöglichkeiten
- Mehrere (auch grafische) Sichten auf ein Modell

Ein möglicher Ansatz zur Erstellung einer derartigen Werkzeugunterstützung liegt im Einsatz von domänenspezifischen Sprachen (DSL). Diese stellen formale Sprachen dar, die zur Lösung von Problemklassen einer spezifischen Domäne (Problembereich) konzipiert werden.

In dieser Arbeit wollen wir uns mit der Erstellung einer Werkzeugunterstützung auf Basis von domänenspezifischen Sprachen für das V-Modell XT beschäftigen. Dabei gehen wir sowohl auf die Unterstützung im Bereich der Prozessmetamodellierung als auch im Bereich der Prozessmodellierung ein, wobei letzteres den Kernbereich der Arbeit stellt.

1.2 Verwandte Arbeiten

In diesem Abschnitt wollen wir uns verwandten Arbeiten zuwenden. Dazu betrachten wir zunächst zwei prominente Prozessmodellierungsframeworks (siehe auch Kapitel 3), diskutieren die hierfür vorliegenden Werkzeuge und stellen schließlich einige Lösungen zum Erstellen von domänenspezifischen Sprachen vor (siehe auch Kapitel 2).

1.2.1 Metamodellierung

Für die Erstellung von Prozessmodellen liegt in der Regel ein entsprechendes Framework vor, das selbst aus einem Prozessmetamodell[3] und einem Werkzeug besteht, so dass Instanzen des Metamodells mit dem Werkzeug erstellt und bearbeitet werden können. Bevor wir einige prominente Beispiel solcher Frameworks uns anschauen, wollen wir zunächst den Begriff der Formalität eines Modells diskutieren, um den Rahmen der Betrachtung von Metamodellen in dieser Arbeit zu verdeutlichen:

Unter einem *formalen* Modell verstehen wir ein Modell, das maschinell verarbeitet werden kann und somit auch automatische Analysen bezüglich Widerspruchsfreiheit und Vollständigkeit zulässt. Wichtig für uns ist die Formalität sowohl beim Metamodell als auch bei den Instanzen des Metamodells, schließlich wollen wir eine Werkzeugunterstützung entwickeln und diese ist nur dann sinnvoll möglich, wenn die zugehörigen Modelle auch maschinenverarbeitbar sind.

3 Eine bekannte Sprache zur Erstellung von Modellen ist UML. Die Elemente von UML werden durch die *Meta Object Facility* (MOF) spezifiziert, das schichtenweise (M0 bis M3) aufgebaut ist [HKKR05]. Für die Prozessmetamodellierung ist UML selbst allerdings zu allgemein.

V-Modell XT

Das V-Modell XT stellt das Standardvorgehensmodell der Bundesrepublik Deutschland zur Entwicklung von IT-Systemen. Das V-Modell selbst ist ein ergebnisorientiertes Prozessmodell, das so genannte Projektfortschrittsstufen [Kuh08] definiert, die durch Entscheidungspunkte abgeschlossen werden. Entscheidungspunkte sind mit Meilensteinen vergleichbar, die im Rahmen des V-Modells zusätzlich qualitätssichernd eingesetzt werden, so dass zu jedem Meilenstein bestimmte Produkte vorgelegt werden müssen.

Das V-Modell XT basiert auf dem gleichnamigen V-Modell XT Metamodell [TK09], das die Strukturen zur Definition des V-Modells bereitstellt. Somit enthält und beschreibt das Metamodell alle Elemente des V-Modells. Ferner ist das Metamodell nicht nur restriktiv für das V-Modell geeignet, vielmehr können weitere Vorgehensmodelle basierend auf dem V-Modell XT Metamodell erstellt werden. Das V-Modell XT Metamodell selbst ist genauso wie seine Instanzen formal und liegt in XML-Schema vor.

Bearbeitet wird das V-Modell XT mit dem V-Modell XT Editor. Dieses Werkzeug ist nicht spezifisch für eine bestimmte Version des Metamodells geschrieben worden, kann folglich auch Modellinstanzen unterschiedlicher Metamodellversionen verarbeiten. Wir betrachten den V-Modell XT Editor eingehend weiter unten.

Eclipse Process Framework

Das Eclipse Process Framework Project (EPF) [EPF07] ist ein Projekt der Eclipse Foundation und seit Oktober 2005 verfügbar. Das Ziel von EPF ist es eine Plattform bereitzustellen, mit der Prozesse für Software-Entwicklungsprojekte definiert und standardisiert werden können und das sowohl für Einzelprojekte als auch für organisationsspezifisches Vorgehen zur Software-Entwicklung. EPF selbst bietet als Prozess das Open-UP Modell, es werden allerdings über Plug-ins weitere Softwareentwicklungsmethodiken unterstützt.

Das OpenUP-Modell ist als Softwareentwicklungsprozess an den Rational Unified Process (RUP) [LL07] angelehnt. Es verfügt über ein formales Metamodell und kann bei EPF stark angepasst werden.

Für das Authoring von OpenUP steht der EPF Composer zur Verfügung. Diesen betrachten wir eingehen weiter unten.

1.2.2 Werkzeuge

In diesem Abschnitt wollen wir uns Werkzeugen zuwenden, die für die oben erwähnten Metamodelle vorhanden sind. Dazu betrachten wir den V-Modell XT Editor, der aktuell bei der Modellierung des V-Modell XT zum Einsatz kommt, den PDS-Designer, der die Modellierung eines Teilmodells des V-Modell XT basierend auf einer domänenspezifischen Sprache bereitstellt und schließlich den EPF Composer, der das Authoring von OpenUP ermöglicht.

V-Modell XT Editor

Der V-Modell XT Editor [VMXb] dient der Modellierung des V-Modells, ist selbst allerdings kein rein für das V-Modell XT geschriebenes Werkzeug, sondern ein angepasster XML Editor [W3Cb]. Als solcher ist er nicht restriktiv für das V-Modell XT konzipiert worden, erhält allerdings auch spezifische Anpassungen, die die Modellierung des V-Modell XT erst in aller Vollständigkeit ermöglichen. Zu beachten ist hierbei, dass der V-Modell XT Editor bei Änderungen am V-Modell XT Metamodell nicht zwingend mitangepasst werden muss.

Als Basis dient dem V-Modell XT Editor ein in XML-Schema [W3Ca] vorliegendes Prozessmetamodell, dessen Strukturen folgend für die Erstellung von Prozessmodellen verwendet werden. Das XML-Schema muss allerdings auch einer vorgegebenen Struktur genügen, damit eine Verarbeitung möglich wird. Die Erstellung entsprechender XML-Schemata wird seitens des Editors nicht weiter unterstützt, so dass es an dieser Stelle leicht zu Fehlern kommen kann.

Der V-Modell XT Editor bietet dem Bearbeiter eine recht einfache Oberfläche, die sich in zwei Bereiche gliedert (siehe Abbildung 1.3). Auf der linken Seite befindet sich eine baumartige Struktur, die die Integrationsbeziehung zwischen den einzelnen Elementen aufweist und zugleich als

Navigator, mit der Möglichkeit zur Selektion der einzelnen Elemente, dient. Rechts davon ist ein Bereich, in dem sich die Eigenschaften und Beziehungen des selektierten Elements anpassen beziehungsweise anlegen lassen.

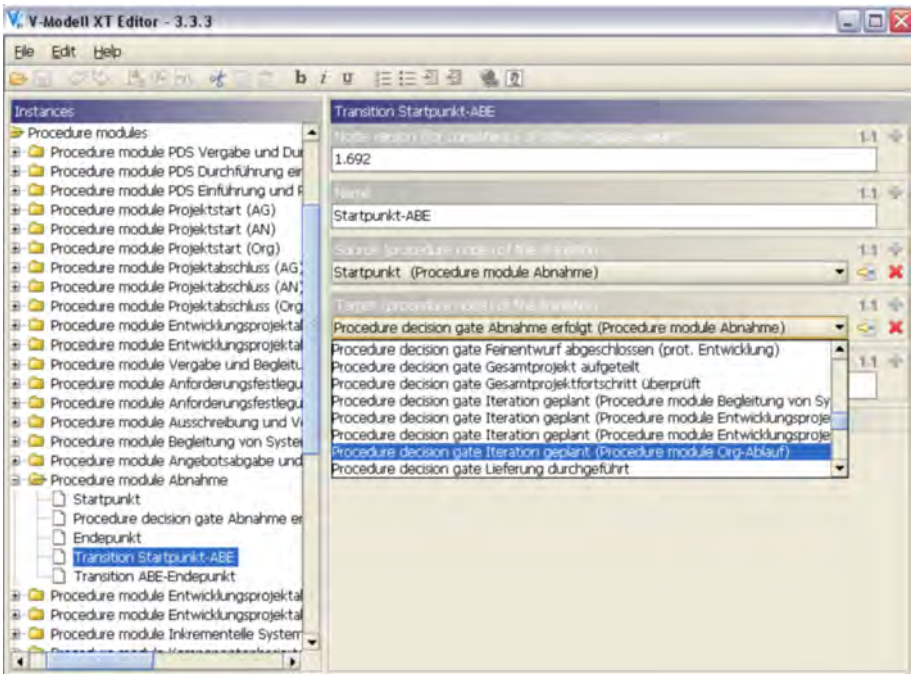


Abbildung 1.3: V-Modell XT Editor

Der V-Modell XT Editor eignet sich zum Modellieren von Prozessmodellen, setzt allerdings ein Prozessmetamodell in einer bestimmten Struktur voraus. Ihm fehlen Validierungsmöglichkeiten, so dass es selbst bei einfacheren modellbedingten Fehlern hier zu größeren Problemen kommen kann, denn der Editor quittiert Fehler mit nur bedingt aussagekräftigen Meldungen. Beispiele davon haben wir bereits in der Motivation betrachtet.

Ferner fehlen dem V-Modell XT Editor grafische Darstellungen zur erweiterten Verwaltung von Modellen. Das ist insbesondere bei komplexen Prozessmodellen wie dem V-Modell XT beispielsweise im Bereich der Projektdurchführungsstrategien problematisch, denn diese lassen sich so nur recht schwer nachvollziehen.

PDS-Designer

Der PDS-Designer [WKK09a] ist eine domänenspezifische Sprache, die der Modellierung von Projektdurchführungsstrategien [WKK09b] aus dem V-Modell XT dient. Der PDS-Designer ist explizit für diesen Einsatz entwickelt worden, so dass hier auch spezifische Verwaltungsmöglichkeiten grafischer Art vorzufinden sind (siehe Abbildung 1.5), die sich in diesem Fall nicht nur auf die Darstellung von Strategien beschränken, sondern auch die Ableitung von Projektplänen entsprechend grafisch visualisieren. Die grafische Darstellung der einzelnen Elemente wird dabei über geometrische Formen realisiert, die mittels unterschiedlicher Farben und Formen unterschiedliche Elementtypen repräsentieren. Abhängigkeiten beziehungsweise Beziehungen zwischen den einzelnen Elementen sind durch Pfeillinien gekennzeichnet, die wiederrum über Farben den jeweiligen Abhängigkeits- beziehungsweise Beziehungstyp ausmachen. Somit können sowohl Projektpläne als auch ihnen zugrunde liegende Strategien dargestellt und durch den Benutzer nachvollzogen werden.

Ferner verfügt der Designer über eine Validierung, die Fehler in den Strategien findet und sie mit Fehlerort und Fehlerursache dem Bearbeiter präsentiert (siehe Abbildung 1.4). Die Validierung wurde dabei über Quellcode spezifisch für das Untersuchen der Strategien auf Fehler angepasst, so dass diese bei der Fehlersuche auch über das entsprechende Wissen verfügt. Auf diese Art und Weise lassen sich Fehler in bestehenden Modellen leichter finden beziehungsweise neue Fehler ganz vermeiden.

Abbildung 1.4: PDS-Designer Fehlerliste

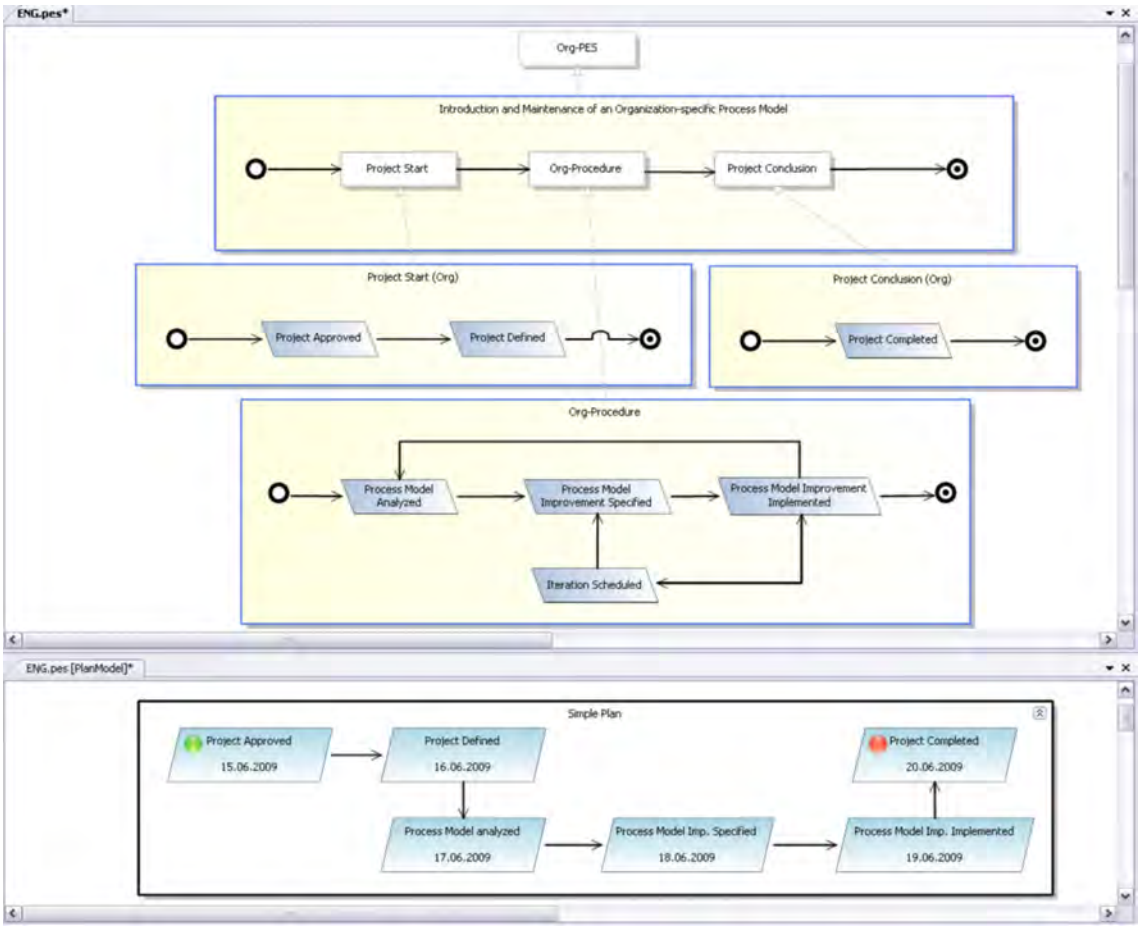


Abbildung 1.5: PDS-Designer

Der PDS-Designer selbst wurde mit den Microsoft DSL-Tools [CJKW07] entwickelt und implementiert direkt das Metamodell der Projektdurchführungsstrategien [TK09]. Somit ist dieser Editor dazu geeignet Strategien basierend auf einem vorgegeben Metamodell (V-Modell XT Metamodell 1.3 [TK09]) zu bearbeiten beziehungsweise zu erstellen. Die Implementierung des Metamodells für den PDS-Designer wurde allerdings seitens der DSL-Tools grafisch unterstützt sowie über eine eingebaute Validierung, zumindest bezüglich der erlaubten Strukturen, fehlerfrei gehalten. Somit ist aus der Sicht der Prozessmetamodellierung hier zumindest eine gewisse Unterstützung gegeben.

Der PDS-Designer kann als eine Art Vorarbeit zu dieser Arbeit gesehen werden, denn er implementiert eine domänenspezifische Sprache für ein Teilmodell des V-Modell XT. Bei dieser Vorarbeit wurden auch eigene Vor- und Nachteile einer derartigen Umsetzung festgestellt, die in dieser Arbeit mitintegriert beziehungsweise ausgebessert werden müssen. Insbesondere wurde dabei festgestellt, dass die grafische Darstellung nur mithilfe vieler Quellcode-Anpassungen realisiert werden kann, so dass beim V-Modell XT als Ganzes eine derartige Implementierung aufgrund des Aufwandes nicht sinnvoll ist. Zudem ist die grafische Darstellung auch eingeschränkt bezüglich der Flexibilität und die Gesamtlösung setzt Microsoft Visual Studio voraus (zumindest in der so genannten Shell-Version), so dass eine gleichartige Umsetzung nicht direkt thematisiert wer-

den kann. Eine Lösung obiger Probleme verlangt somit ein neues Konzept beziehungsweise eine neue Gesamtvorgehensweise.

EPF Composer

Der EPF Composer [Hau06a, Hau06b] dient der Erstellung und Anpassung von Prozessmodellen im Rahmen von EPF und bietet dem Bearbeiter hierzu ein funktionsreiches Werkzeug. Dieses verfügt dabei über eine moderne Darstellungsweise mit mehreren Fenstern, die jeweils der Navigation, Konfiguration oder dem Bearbeiten von Elementen dienen. Durch die Vielzahl an Optionen wird jedoch ein unerfahrener Benutzer gerade hierdurch überfordert, da zudem die Oberfläche überladen wirkt.

Der EPF Composer bietet im Gegensatz zum V-Modell XT Editor auch grafische Modellierungsweisen an (siehe auch Abbildung 1.6), die sich allerdings nur auf die Erstellung von Work Product Abhängigkeiten sowie Aktivitätsflußdiagrammen beschränken. Gerade letzteres kann dabei mit der Einfachheit der grafischen Definitionsweise von Abläufen beim PDS-Designer nicht mithalten.

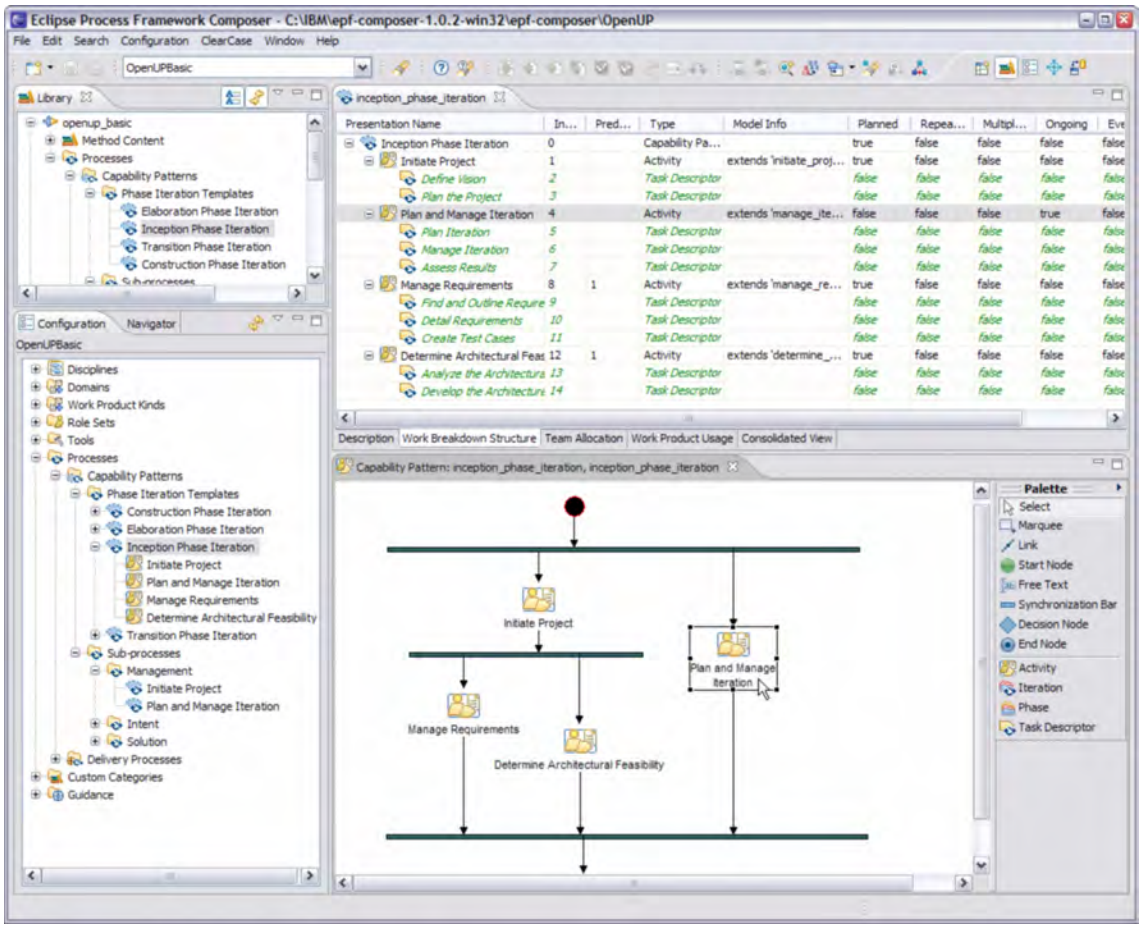


Abbildung 1.6: EPF Composer

1.2.3 DSL Modellierungsframeworks

In diesem Abschnitt widmen wir uns der Erstellung von domänenspezifischen Sprachen. Bevor wir allerdings zwei bekannte DSL Modellierungsframeworks kurz vorstellen, wollen wir einige Begrifflichkeiten für diese Arbeit festlegen, die wir folgend auch in späteren Kapiteln verwenden werden:

Domänenmetamodell Das *Domänenmetamodell* ist eine für ein Prozessmodell (oder für ein beliebiges Modell) erstellte Repräsentation in einer DSL. Es bildet Elemente und Beziehungen

eines Prozessmodells in die vorliegenden abstrahierten Strukturen der Sprache.

Domänenmodell Das *Domänenmodell* stellt eine Instanz des Domänenmetamodells in der DSL dar und liegt im Sinne der betrachteten DSL Frameworks im Quellcode vor.

Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) [SBPM08, EMF] wird von der Eclipse Foundation bereitgestellt und erlaubt die Erstellung von Applikationen oder Codefragmente anhand vorgegebener strukturierter Modelle. EMF selbst integriert sich als Plugin in die Eclipse Platform [EPT] und verfügt zur Definition des Domänenmetamodells über den so genannten ECore Editor. Das Prinzip von EMF basiert auf der Generierung von Quellcode anhand des Domänenmetamodells, der folgend benutzerspezifisch angepasst und zu einem Editor kompiliert werden kann.

Microsoft DSL Tools

Die Microsoft Domain Specific Language (DSL)-Tools [CJKW07] sind ein Framework zur Erstellung grafischer Sprachen, die als Plugins in Microsoft Visual Studio integriert werden. Das Domänenmetamodell wir bei den DSL-Tools über den so genannten DSL-Tools Designer implementiert, der selbst auch eine grafische DSL darstellt. Das Prinzip der DSL-Tools ist ähnlich zu EMF, es wird basierend auf dem Domänenmetamodell Quellcode generiert, der folgend vom Benutzer angepasst werden kann, um beispielsweise spezifische Validierungsmethoden hinzuzufügen. Die DSL-Tools erlauben somit die Festlegung einer Validierung des Modells über Quellcode, die ihrerseits die Konsistenz von Modellinstanzen sicherstellen kann.

1.3 Ziele der Arbeit

In dieser Arbeit wollen wir einerseits die komplexe Welt der Prozessmodellierung analysieren und in Verbindung mit einer Umsetzung von Prozessmodellen in domänenspezifische Sprachen stellen sowie andererseits vor allem auch eine Werkzeugunterstützung für ein anerkanntes Vorgehensmodell, das V-Modell XT, basierend auf einer DSL entwickeln. Die Ziele dieser Arbeit unterteilen sich in zwei Bereiche:

Analyse und Design: Zielkonzepte. Die Entwicklung einer domänenspezifischen Sprache für ein komplexes Prozessmodell ist keine einfache Angelegenheit, so dass das Ziel dieser Arbeit auch darin liegt, die zugrunde liegenden Konzepte der Prozessmodellierung sowie der Entwicklung von DSLs zu vermitteln. Ferner ist die Vermischung der beiden Domänen im Sinne der Umsetzung von Prozessmodellen als DSLs zu betrachten und auch insbesondere für das V-Modell XT zu evaluieren. Die Anwendung einer solchen Umsetzungsmöglichkeit soll erläutert und beispielhaft dargestellt werden. Dabei ist das Ziel auch das hierzu erstellte *Process Development Environment* detailliert vorzustellen, um dem Leser die Konzepte hinter dem Framework und die Designmöglichkeiten des Frameworks zum Erstellen von DSLs nahzubringen. Schließlich soll die konkrete Umsetzung des V-Modell XT Metamodells in eine DSL dargestellt und die so erstellte Sprache auch mitsamt ihrem Editor diskutiert werden.

Anforderungen an die V-Modell XT DSL. Als direktes Ergebnis dieser Arbeit ist der Entwurf einer domänenspezifischen Sprache für das V-Modell XT und somit die Bereitstellung einer DSL zum Design von Prozessmodellen auf Basis des V-Modell XT Metamodells zu sehen. Diese DSL soll in der Lage sein mit der aktuellen Version des V-Modell XT zu arbeiten. Dazu muss sie zum einen die aktuelle Version des V-Modell XT Metamodells implementieren und zum anderen auch über Serialisierungsmöglichkeiten verfügen, um die Instanz des V-Modells laden und speichern zu können. Die DSL selbst soll über eine grafische Modellierungsmöglichkeit verfügen, so dass ein Editor als Applikation in der Sprache integriert ist.

Die DSL muss ferner Validierungsmöglichkeiten vorsehen, um die V-Modell XT Instanzen auf Inkonsistenzen prüfen zu können und um diese auch entsprechend aufzubereiten. Diesbezüglich ist die Fehlerquelle und der Fehlergrund einer jeden Inkonsistenz festzuhalten.

Folglich soll die DSL über Verwaltungsmöglichkeiten verfügen, so dass sowohl vorhandene Prozessmodelle bearbeitet als auch neue basierend auf dem V-Modell XT Metamodell angelegt werden können. Hierzu kommt noch die Bereitstellung von erweiterten Verwaltungsmöglichkeiten, die auch grafischer Art sein können.

Die erstellte DSL soll zudem erweiterbar (im Sinne eines Pluginkonzepts) sein, so dass eine Zusammenarbeit mit weiteren Werkzeugen (im Rahmen des V-Modell XT) möglich ist.

1.4 Aufbau der Arbeit

Der strukturelle Aufbau der Arbeit ergibt sich wie folgt:

- Kapitel 2 beschäftigt sich mit domänenspezifischen Sprachen sowie mit domänenspezifischer Entwicklung und stellt verbreitete DSL-Frameworks vor.
- Kapitel 3 behandelt die Modellierung von Prozessmodellen und stellt Vor- und Nachteile bekannter Vorgehensmodelle im Sinne ihrer Modellierungsframeworks vor.
- Kapitel 4 analysiert das V-Modell XT Metamodell in Verbindung mit domänenspezifischen Sprachen. Dabei wird zunächst das V-Modell XT Metamodell vorgestellt, um folgend mögliche Umsetzungsmöglichkeiten mit ihren Vor- und Nachteilen zu diskutieren. Schließlich betrachten wir im Sinne der Entwicklung einer DSL für das V-Modell XT die hierbei relevanten Sprachanforderungen.
- Kapitel 5 stellt die domänenspezifische Sprache für das V-Modell XT vor, die auf dem *Process Development Environment* (PDE)-Ansatz aufbaut. Hier wird auch PDE selbst vorgestellt sowie spezifische Anpassungen, die für die DSL für das V-Modell XT notwendig sind, angeregt. Schließlich wird das Gesamtergebnis diskutiert.
- Kapitel 6 beinhaltet eine Zusammenfassung, eine Verallgemeinerung sowie einige offene Fragen zur Thematik der Arbeit.

2 Domänenspezifische Sprachen

In diesem Kapitel diskutieren wir den Begriff der domänenspezifische Sprache (DSL), betrachten die Idee der domänenspezifischen Entwicklung und stellen ferner das Eclipse Modeling Framework Project (EMF) sowie die Microsoft DSL-Tools vor. Dabei gehen wir insbesondere auf die DSL-Tools näher ein, da diese zu einem späteren Zeitpunkt in dieser Arbeit Verwendung finden. Schließlich geben wir noch einen Ausblick zur zukünftigen Entwicklung domänenspezifischer Sprachen.

Übersicht

2.1 Einführung

Universell einsetzbare Sprachen (General Purpose Language, GPL) sind Programmiersprachen, die für den Einsatz in vielen unterschiedlichen Domänen konzipiert werden. GPLs lassen sich nach den unterstützten Programmierparadigmen klassifizieren (Auswahl):

- funktional: Programme werden als mathematische Funktionen aufgefasst. Beispiele für funktionale Programmiersprachen sind Haskell, Lisp oder F#.
- objektorientiert: Objektorientierte Programmiersprachen verinnerlichen das Prinzip der Objektorientierung, bei dem Daten als Objekte gekapselt werden. Beispiele: C# oder Java.
- logisch: Beruht auf mathematischer Logik, so dass Programme aus einer Menge von Axiomen bestehen. Logische Programmiersprachen werden beispielsweise bei der Entwicklung Künstlicher Intelligenz oder in Expertensystemen eingesetzt. Ein bekannter Vertreter einer logischen Programmiersprache ist Prolog.
- nebenläufig: Aktionen werden parallel ausgeführt. Besonders geeignet unter anderem für spezifische mathematische Berechnungen beispielsweise bei Spielen oder im Embedded Bereich [ABC+06]. Bekannte Vertreter sind Ada oder Haskell.

Viele GPLs wie beispielsweise C# oder Python unterstützen mehrere dieser Paradigmen. Basierend auf der obigen Klassifikation lässt sich erkennen, dass GPLs auch spezifische Einsatzbereiche aufweisen können, für die sie besonders geeignet sind. So werden objektorientierte Sprachen eher beim Entwurf von Systemen eingesetzt, die über grafische Benutzeroberflächen gesteuert werden, während Bibliotheken, die zum Beispiel zur Definition einer Künstlichen Intelligenz erstellt werden, mitunter funktionale und/oder logische Programmiersprachen verwenden.

Eine *domänenspezifische Sprache* (DSL) ist eine formale Sprache zur Lösung einer Klasse von Problemen in einer bestimmten Domäne. DSLs sind im Gegensatz zu GPLs auf eine bestimmte Domäne zugeschnitten und verfügen hierdurch über eine hohe Ausdrucksmächtigkeit [MHS05]. Auch bei DSLs lassen sich Klassifikationen bezüglich des Spezialisierungsgrads beziehungsweise der Eignung vornehmen:

- Class Designer im Visual Studio: Sehr spezifische grafische DSL zum Modellieren von Klassendiagrammen.
- Html: Textbasierte Sprache zur Strukturierung von Inhalten insbesondere im World Wide Web.
- SQL: Datenbanksprache, die bei der Definition, Manipulation oder Abfrage von Daten in relationalen Datenbanken eingesetzt wird.
- VHDL: Hardwarebeschreibungssprache. Speziell zur Beschreibung komplexer digitaler Systeme geeignet.
- Microsoft DSL-Tools: Sprache zum Erstellen grafischer DSLs.

Somit stellen beispielsweise die DSL-Tools vergleichsweise zu Html oder SQL eine deutlich allgemeinere Sprache dar, schließlich können damit weitere Sprachen definiert werden, die auch mitunter unterschiedliche Spezialisierungen aufweisen können. Der aktuelle Class Designer im Visual Studio kann mit den DSL-Tools implementiert werden, die DSL-Tools selber können aber auch mit sich selbst erstellt werden.

Ein weiterer interessanter Aspekt der DSLs im Vergleich zu GPLs ist die Tatsache, dass diese weniger Programmier- und Fachkenntnisse voraussetzen und so eine größere Zielgruppe ansprechen.

Domänenspezifische Sprachen können textuell oder grafisch sein. Prominente Vertreter textueller Sprachen [MHS05] sind beispielsweise Html und LaTeX. Grafische Sprachen haben gegenüber textuellen Sprachen den Vorteil einer speziellen grafischen Darstellung für den Problembereich. So können Zusammenhänge beziehungsweise Abhängigkeiten oder Beziehungen grafisch in einer diagramartigen Art und Weise dargestellt werden, um die Erfassbarkeit und die Nachvollziehbarkeit zu erleichtern. Prominenter Vertreter grafischer DSLs sind unter anderem Microsoft Visual Studio Class Designer und die Microsoft DSL-Tools.

2.2 Domänenspezifische Entwicklung

Mittels domänenspezifischer Entwicklung [CJKW07] wird versucht eine Lösung für ein immer wieder auftretendes Problem zu finden. Die Probleme hierbei müssen gleichartig sein, so dass ei-

ne allgemeine Lösung für alle Probleme ableitbar ist. Betrachten wir hierzu als Beispiel die Sprache der regulären Ausdrücke.

Wir wollen in beliebigen Zeichenketten oder auch Dateien, die als solche eingelesen werden, nach einem Ausdruck von einem bestimmten Format, beispielsweise nach E-Mail Adressen, suchen und die gefundenen Werte sammeln. Eine erste mögliche Vorgehensweise dazu wäre die Erstellung eines entsprechenden Programms, welches diese Aufgabe übernimmt. Eine zweite Möglichkeit wäre es die Sprache der regulären Ausdrücke zu benutzen, die über den regulären Ausdruck

```
(?<user>[^@]+)@(?<host>.+)
```

alle Email Adressen, die in einer Zeichenkette enthalten sind, auslesen kann. Der große Vorteil der zweiten Lösungsmöglichkeit ist die Verwendung einer formal definierten Sprache, die für diese spezielle Domäne geeignet ist. Bei der ersten Möglichkeit hat man zur Lösung derselben Aufgabe jedoch deutlich mehr Aufwand zu investieren.

Im obigen Beispiel haben wir unser Problem (suchen in Zeichenketten nach Email-Adressen) über die Repräsentation und Auswertung in der Sprache der regulären Ausdrücke gelöst. So verhält es sich auch mit der Vorgehensweise bei der domänenspezifischen Entwicklung. Hier haben wir gleichartige Probleme, die sich in gewissen Aspekten unterscheiden, und für die sich eine allgemeine Lösung ableiten lässt. Diese unterschiedlichen Aspekte können von einer speziellen Sprache repräsentiert werden, so dass das Auftreten eines jeden Problems über die Instanziierung eines Modells in der Sprache und das Auswerten desselben gelöst werden kann.

Hinweis

Die Idee der domänenspezifischen Entwicklung lässt sich bei den Microsoft DSL-Tools wiederfinden.

Die grundsätzliche Idee dabei ist es das spezifische Modell zur Lösung eines Problems in ein bereits vorhandenes, allgemeines Framework einzusetzen (siehe Abbildung 2.1).

Anwendung. Im Rahmen dieser Arbeit findet das Prinzip der domänenspezifischen Entwicklung wie folgt Anwendung: Die zu lösenden Probleme sind in der Entwicklung von Vorgehensmodellen basierend auf dem V-Modell XT Metamodell zu sehen und genau für dieses Metamodell wollen wir eine Abbildung in ein Zwischenmodell definieren, das selbst in ein vorliegendes Framework integriert werden soll, das wiederrum die Verwaltung von Instanzen des V-Modell XT Metamodells bereitstellt.

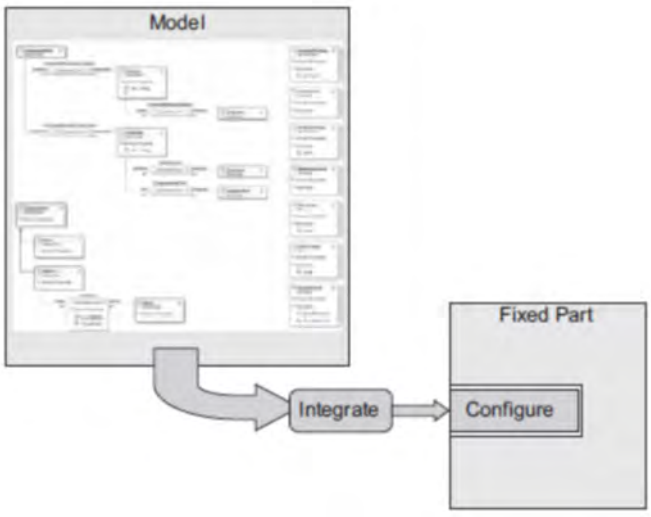


Abbildung 2.1: Idee der domänenspezifischen Entwicklung, Quelle [CJKW07]

2.3 Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) [SBPM08, EMF] ist ein Open-Source Framework, das als Modellierungs- und Quellcodegenerierungswerkzeug anhand vorgegebener strukturierter Modelle Applikationen oder Codefragmente erzeugen kann. Das EMF selbst integriert sich (als Plugin) in die Eclipse Platform [EPT], welches eine Plugin-ähnliche Plattform zur Werkzeugintegration darstellt. Sowohl Eclipse als auch EMF werden durch die Eclipse Foundation bereitgestellt und basieren auf Java, sind folglich plattformunabhängig.

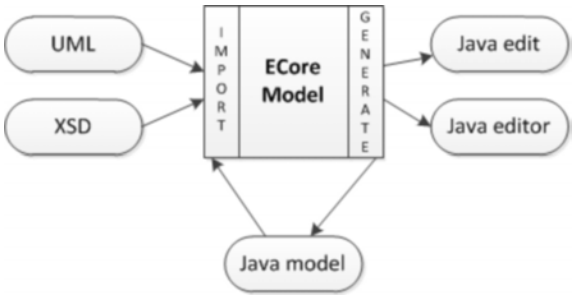


Abbildung 2.2: EMF: ECore Import und Generierung

Der Prozess der Erstellung eines Editors mit EMF läuft wie folgt ab [MS05] (siehe auch Abbildung 2.2):

1. Import or Create: EMF erlaubt die Definition des eigenen Modells über einen Import eines bereits vorhandenen strukturierten Modells, das beispielsweise in Form eines UML Klassendiagram, eines XML Schemas oder mittels Java Interfaces vorliegen kann. Das so erstellte eigene Modell ist eine Instanz des *ECore Metamodell* und nennt sich *ECore* (siehe weiter unten). Anstelle des Imports von einem bereits existierenden Modell, lässt sich genauso ein neues Modell mithilfe des EMF ECore Editors konzipieren. Der EMF ECore Editor dient folglich auch zur Bearbeitung von importierten Modellen, so dass diese spezifisch angepasst oder erweitert werden können.
2. Generate: Basierend auf dem erstellen ECore Modell wird in diesem Schritt Quellcode generiert. Die Generierung lässt sich dabei auf drei Bereiche aufteilen:
   a) Java Modell: Hierbei wird Quellcode zur Implementierung des Modells in Java erzeugt.
   b) Java Edit: Erstellt Quellcode für allgemeine Editieroperationen für Modelle sowie weitere Adapter zur Unterstützung von Editoren.
   c) Java Editor: Erzeugt einen vollständigen Editor, der als Eclipse Plugin einsetzbar ist und demzufolge für die Benutzung Eclipse voraussetzt.
3. Customize: Der generierte Quellcode kann in diesem Schritt angepasst beziehungsweise erweitert werden.

Eine generelle Problematik der Regenerierung von Quellcode ergibt sich aus der Notwendigkeit der richtigen Verwendung des *@generated* Schlüsselworts. Dieses muss entfernt beziehungsweise angepasst (zu *@generated NOT*) werden, falls modifizierter Quellcode bei der Generierung nicht überschrieben werden soll.

Folgend wollen wir noch auf das ECore Metamodell eingehen, das EMF als Basis dient.

ECore Metamodell. Die Spezifikation von strukturierten Modellen, die bei der Codegenerierung verwendet werden können, liegt in Form des *ECore Metamodell* (siehe Abbildung 2.3) vor. Das Modell, welches Modelle bei EMF repräsentiert, nennt sich dabei *ECore*. Interessanterweise ist das ECore zugleich sein eigenes Metamodell beziehungsweise das ECore Metamodell ist gleichzeitig sein eigenes Meta-Metamodell.

Zur Definition von Modellen weist das ECore Metamodell unter anderem folgende Elemente auf:

EClass Repräsentiert eine modellierte Klasse, die über ein oder mehrere Attribute sowie ein oder mehrere Referenzen verfügt.

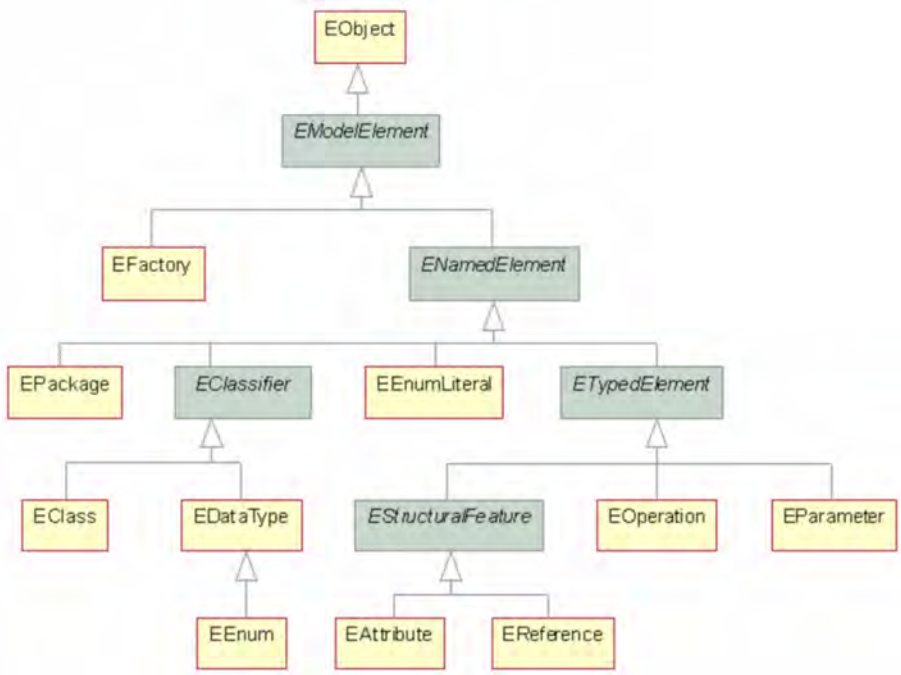


Abbildung 2.3: EMF: ECore Metamodell, Quelle [MS05]

EAttribute Beschreibt eine modellierte Eigenschaft, die über einen Namen und einen Typ definiert wird.

EReference Stellt ein Ende einer Assoziation zwischen zwei Klassen dar. Die referenzierte Klasse wird über einen festgelegten Klassentyp konkretisiert. Ferner definiert eine boolesche Eigenschaft, ob es sich bei dieser Beziehung um eine Integrationsbeziehung handelt oder nicht.

EDataType Beschreibt den Typ einer modellierten Eigenschaft. Hierbei kann es sich um einfache (z.B.: int) sowie um komplexere (z.B.: Date) Datentypen handeln.

2.4 Microsoft DSL Tools

Die Microsoft Domain Specific Language (DSL)-Tools [CJKW07] sind ein Framework zur Erstellung und Veröffentlichung von domänenspezifischen Sprachen, die der Automatisierung der Softwareentwicklungsprozesse dienen. Die DSL-Tools erlauben die Erstellung und Modifikation sowie eine grafische Visualisierung von domänenspezifischen Daten. Die Definition von domänenspezifischen Sprachen mit den DSL-Tools wird über einen in Visual Studio integrierten Designer durchgeführt. Der Designer der DSL-Tools zur Erstellung von domänenspezifischen Sprachen ist dabei selbst als DSL realisiert.

Der Prozess der Erstellung einer DSL mit den DSL-Tools läuft wie folgt ab (siehe auch Abbildung 2.4):

1. Zunächst wird mit dem *DSL-Tools Designer* das Domänenmetamodell[1] definiert, welches die wesentlichen Informationen zum strukturellen Aufbau des Modells der DSL beinhaltet. Ferner findet die Definition der Darstellung und der Serialisierung im DSL-Tools Designer statt.
2. Das mit dem DSL-Tools Designer definierte Domänenmetamodell wird mittels des Text Templating Transformation Toolkits (Kurz: T4, [T4R07, Syc07]) in Quellcode übersetzt und lässt sich nun spezifisch erweitern und anpassen.

1 Eingangs haben wir in Abschnitt 1.2 dieses Modell als das Domänenmetamodell definiert, was insbesondere für den späteren Verlauf der Arbeit wichtig ist. In der Literatur wird dieses Modell bei den DSL-Tools auch nur als das Domänenmodell bezeichnet.

3. Durch das Übersetzen der Anwendung wird der eigentliche Editor (Gesamt-DSL) erstellt, womit sich Modellinstanzen der Sprache bearbeiten lassen.

Der erstellte Editor selbst kann als Plugin im Visual Studio eingesetzt werden.

Abbildung 2.4: DSL-Tools Prozess der Erstellung von DSLs

Das Domänenmetamodell. Das Domänenmetamodell der DSL-Tools besteht aus dem Element *DomainClass* und den dazwischen definierten Beziehungen:

Embedding Relationship Eine DomainClass kann mehrere andere Klassen integrieren. Damit lassen sich hierarchische Zusammenhänge modellieren.

Reference Relationship Eine DomainClass kann andere Klassen referenzieren. Eine Referenz in diesem Sinne stellt eine Beziehung zwischen zwei Klassen her. Weiterhin lassen sich bei der Referenz die Kardinalitäten der eingehenden Beziehungspartnerinstanzen festlegen.

Inheritance Relationship Eine DomainClass wird als die vererbende Klasse festgelegt, von der mehrere andere Klassen abgeleitet werden können.

Die Elemente und Beziehungen des Domänenmetamodells werden benutzt, um hieraus Quellcodeklassen zu erstellen, die als partielle Klassen definiert werden. Eine partielle Klasse bezeichnet eine Klasse, die über mehrere Quellcodedateien aufgeteilt ist oder an mehreren verschiedenen Orten einer Datei deklariert ist. Auf diese Art und Weise lassen sich Klassen in den DSL-Tools erweitern beziehungsweise ihr Verhalten anpassen.

Visualisierung des Domänenmetamodells. Neben den Modellierungsmöglichkeiten des Domänenmetamodells in den DSL-Tools gibt es einen weiteren, wesentlichen Baustein: Die Definition der Visualisierung der Elemente des Domänenmetamodells. Dazu betrachten wir allerdings zunächst die Bereiche (Abbildung 2.5), die bei der Darstellung der domänenspezifischen Sprache im in Visual Studio integrierten Designer eine Rolle spielen:

1. *Designer Surface*: Jede Element-Instanz, die über ein zugewiesenes Darstellungselement (siehe unten) verfügt wird automatisch im Designer mit ihrer grafischen Repräsentation dargestellt.
2. *Property Window*: Hier werden die Eigenschaften des selektierten Elements dargestellt und können, falls nicht bei der Definition der Sprache als *readonly* festgelegt, auch modifiziert werden. Beziehungen werden im Property Window nur angezeigt, falls sie über Kardinalitäten der Form 0..1 oder 1..1 verfügen. Beziehungen mit höherwertigen Kardinalitäten können im Property Window standardmäßig nicht modifiziert werden.
3. *Model Explorer*: Der Model Explorer stellt das Gesamtmodell der domänenspezifischen Sprache in einer Baumstruktur dar. Er erlaubt das Erstellen, das Modifizieren oder das Löschen von Elementen über ein Kontextmenü.

Die Zuweisung von Elementen zu einer grafischen Repräsentation geschieht im DSL-Tools Designer. Dazu wird zunächst aus einer Reihe von vordefinierten Darstellungen (*Shapes*) eine ausgewählt, zum Designer hinzugefügt und ferner eine Zuweisungen zwischen dem neuen Repräsentationselement und dem DomainClass-Element vorgenommen. Dabei kann aus einer Reihe von Shapes gewählt werden (die unten genannten Shapes sind bis auf Connector alle für DomainClass definiert, für DomainRelationship ist der Connector entsprechend zu verwenden):

Geometry Shape Ein Geometry Shape definiert eine grafische Visualisierung, die einem geometrischen Objekt entspricht. Vordefiniert zur Auswahl stehen die Formen: (abgerundetes) Rechteck, Ellipse und Kreis.

Image Shape Ein Image Shape weist als Visualisierung anstatt einer geometrischen Form ein zugewiesenes Bild auf.

Compartment Shape Ein Compartment Shape ist ähnlich zu einem Geometry Shape, erlaubt jedoch zusätzlich die Definition von zusätzlichen Elementen, die im Compartment Shape

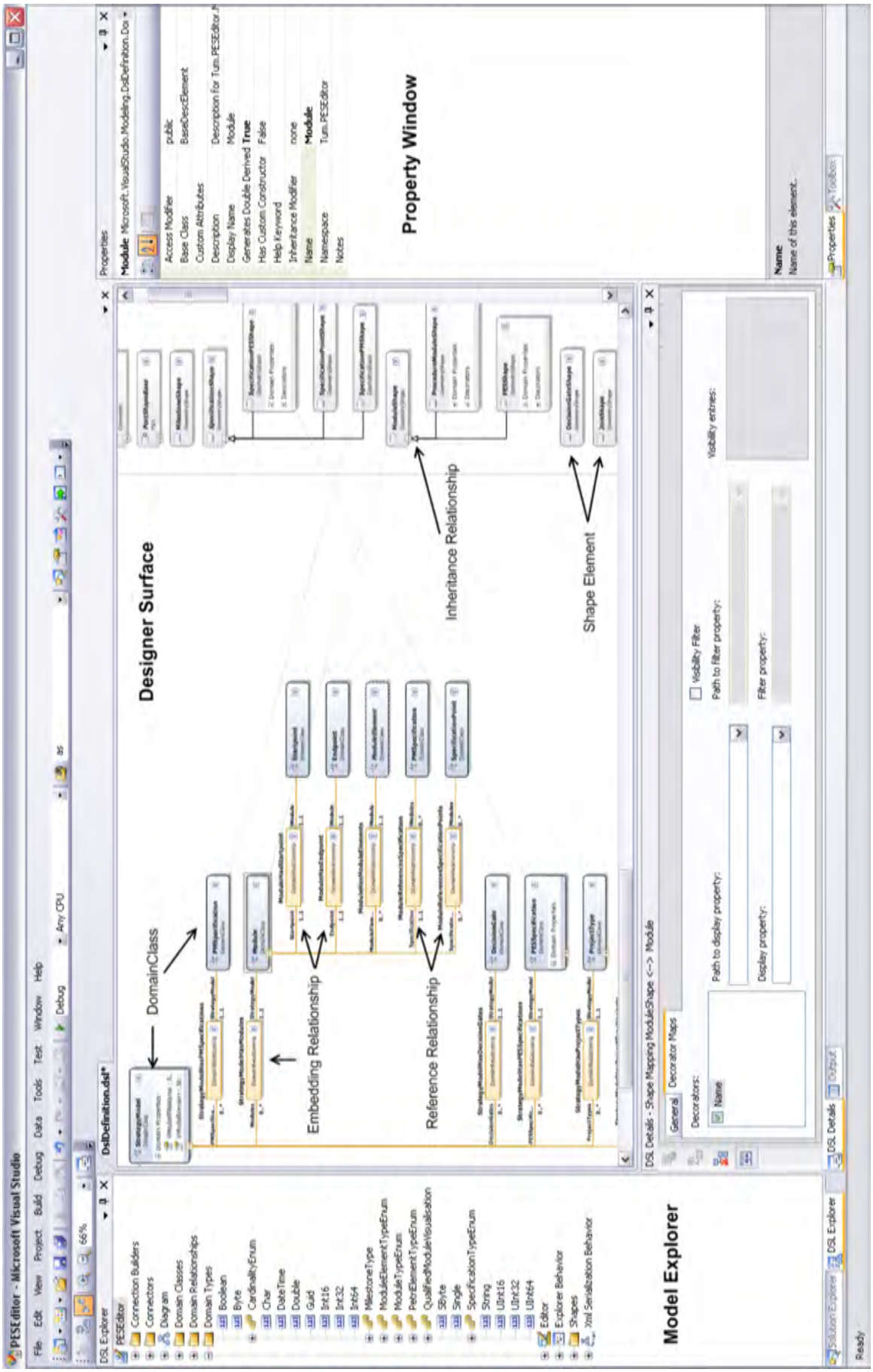


Abbildung 2.5: Visual Studio IDE zum DSL Design (DSL Tools), Quelle: [WKK09b]

selbst dargestellt werden. Eine beispielhafte Visualisierung lässt sich der Abbildung 2.5 entnehmen. Dort sind alle grafischen Darstellungen der Klasse DomainElement über eine Compartment Shape - artige Darstellung realisiert.

Port Shape Ports sind spezielle Shapes, die am Rand eines anderen Shapes dargestellt werden und auch nur entlang dieses Randes bewegt werden können.

Connector Ein Connector ist für die Repräsentation von Beziehungen zuständig. Diese werden über Pfeildarstellungen visualisiert.

Transformation, Validierung und Serialisierung. Nach der Definition des Domänenmetamodells sowie der grafischen Repräsentation, wird diese Information automatisch in Quellcode übersetzt. Hierzu setzen die DSL-Tools, wie bereits oben erwähnt, T4 ein. Bei der Übersetzung werden die oben genannten Eigenschaften berücksichtigt, so dass der generierte Quellcode auch nur über den grafischen Designer der DSL-Tools verändert werden kann (jegliche Änderungen direkt am Quellcode gehen bei der nächsten automatischen Generierung verloren). Um dennoch Eigenschaften der generierten Klassen anzupassen, bieten die DSL-Tools den Mechanismus der partiellen Klassen sowie das „Double Derived"-Feature.

Die Validierung ist ein wesentlicher Aspekt der DSL-Tools. Hierdurch wird die Überprüfung der vorliegenden Strukturen sowie die Einhaltung vorgegebener Eigenschaften möglich. Die Validierung erlaubt es zudem Feedback über Inkonsistenzen zum Designzeitpunkt dem Entwickler vorzulegen und diesen zudem exakt auf die problematischen Bereiche hinzuweisen. Die DSL-Tools definieren bei der Validierung die so genannten *Soft-* und *Hard Constraints*. Die ersten sind Auflagen die zeitweise gebrochen werden können beziehungsweise sogar müssen um eine bestimmte Struktur zu modellieren. Somit müssen Soft Constraints spätestens zum Zeitpunkt der Auslieferung des Modells oder beispielsweise zum Serialisierungszeitpunkt erfüllt werden. Die DSL-Tools erlauben das Festlegen der Soft Constraints über Quellcode, indem in partiellen Klassen spezielle Methoden definiert und bei der Validierung aufgerufen werden. Die Hard Constraints sind hingegen Auflagen, die zu keiner Zeit gebrochen werden können. Diese werden bei den DSL-Tools über automatisch generierten Quellcode eingehalten und können zusätzlich durch das Überschreiben von Framework-Methoden hinzugefügt werden. Es kann festgehalten werden, dass für Soft Constraints eigene Methoden zu erstellen sind während für Hard Constraints bereits vorhandene überschrieben und angepasst werden müssen.

Die Serialisierung bei den DSL-Tools wird mit Hilfe von automatisch generierten Serialisierungsklassen realisiert. Dabei wird für jedes DomainClass im Domänenmetamodell eine eigene Serialisierungsklasse definiert, die für das Speichern und Laden der jeweiligen Eigenschaften zuständig ist. Bei der Serialisierung muss beachtet werden, dass zwei Teilmodelle zu serialisieren sind. Zum einen das Domänenmetamodell selbst und zum anderen die Informationen über die grafische Darstellung (z.B.: Position und Größe der Elemente). Beides wird wie oben erwähnt über eigene Serialisierungsklassen erledigt, die selbst wiederum partielle Klassen sind und somit auch spezifisch angepasst werden können. Generell ist so eine Anpassung aber nicht immer möglich. Deshalb gibt es die Möglichkeit, die Serialisierung komplett durch eigenen Quellcode umzusetzen. Dabei müssen nicht zwangsläufig beide Teilmodelle eigens serialisiert werden, so kann beispielsweise das Speichern und Laden der grafischen Information nach wie vor durch den automatisch generierten Quellcode übernommen werden. Mehr Informationen zu den DSL-Tools kann der interessierte Leser [CJKW07] entnehmen.

2.5 Ausblick

Die Entwicklung einer domänenspezifischen Sprache ist eine komplexe Aufgabe, daraus ergeben sich jedoch auch erhebliche Vorteile beispielsweise bei der Erlernbarkeit oder Nutzbarkeit. Interessant ist die Entwicklung hinsichtlich der Nutzung von DSLs, diese existieren nämlich schon seit einer langer Zeit (erste sog. DSL: „Automatically Programmed Tools" von 1959 [MHS05, Ros81]), kommen allerdings erst in letzter Zeit zu einer immer größer werdenden Popularität. Nachfolgend wollen wir einen Ausblick [ASS+09] auf die Entwicklung im Bereich der DSLs geben, indem wir ein interessantes Forschungsthema ansprechen, bei dem der Einsatz von Sprachen sehr innovativ erscheint. Ein derartiges Thema stellt ein Forschungsprojekt dar, das die Vision der Unterstützung paralleler Applikationen mithilfe von DSLs verinnerlicht. Die Idee der Stanford Pervasive Parallelism Lab [ADF+08] lässt sich wie folgt einfach zusammenfassen: Programmierer

entwickeln mittels einfacher DSLs, die funktionale und objektorientierte Programmierung unterstützen, kümmern sich aber nicht um Parallelität. Die erstellte Applikation wird hingegen parallel auf einem Multicore-Rechner ausgeführt. Dies würde die nebenläufige Programmierung erheblich vereinfachen.

3 Modellierung von Prozessmodellen

In diesem Kapitel wollen wir uns mit der Modellierung von Prozessmodellen beschäftigen und im Zuge dessen feststellen, wie dieses basierend auf vorhandenen Frameworks möglich ist, welche Voraussetzungen dazu an die Frameworks gestellt werden müssen und welche Probleme dabei auftreten.

Übersicht

3.1 Einführung

Prozessmodelle erleichtern beziehungsweise unterstützen als Vorgehensmodelle in der Softwareentwicklung die Erstellung von Softwareprojekten, indem sie die Komplexität handhabbar gestalten sowie die Übersicht und Nachvollziehbarkeit während der Entwicklung steigern. Vorgehensmodelle dokumentieren eine bewährte Vorgehensweise, die einen strukturellen, zielorientierten sowie geplanten Projektablauf ermöglicht. Aktuell existiert eine Reihe von Vorgehensmodellen, die jeweils unterschiedliche Vor- und Nachteile aufweisen. Wir interessieren uns in diesem Kapitel insbesondere für die Entwicklung von Prozessmodellen, so dass wir auch die zugehörigen Metamodellen sowie das Gesamtentwicklungskonzept betrachten wollen, das aus dem Metamodell, der Werkzeugunterstützung sowie dem Prozessmodell selbst als Ergebnis besteht.

Hinweis

Die Entwicklung von Vorgehensmodellen ist auf der Prozessingenieur-Ebene eine sehr schwere und komplexe Aufgabe. Der Prozessingenieur muss neben der Konzeption und Kenntniss sinnvoller Projektabläufe, der Best Practices sowie der praktischen Einflüsse und Gegebenheiten einer Anwendung des Vorgehensmodells, auch in der Lage sein, einige Schritte gegenüber dem Anwender eines Vorgehensmodells voraus zu denken. Die definierten Prozesse können schließlich auf unterschiedliche Art und Weise angewendet werden, somit ist eine klare und sinnvolle Zusammenstellung der möglichen Abläufe essenziell, denn selbst kleinere Fehler hier können zu schwerwiegenden Problemen in der Projektentwicklung führen.

Anpassbarkeit von Vorgehensmodellen. Die Betrachtung von Prozessmodellen und ihren Modellierungsmöglichkeiten bezieht sich insbesondere auf die Voraussetzungen in den entsprechenden Metamodellen. Somit ist eine erfolgreiche Entwicklung eines Prozessmodells in einem speziellen Framework nur möglich, falls das Framework auch alle Anforderungen des zu modellierenden Prozessmodells unterstützt, sprich über sein Metamodell die entsprechenden Modellierungsmöglichkeiten auch bereitstellt. Das ist allerdings nicht immer notwendigerweise gegeben, wobei sich die Möglichkeiten zwischen den einzelnen Frameworks teilweise enorm unterscheiden.

Um die Betrachtung von Metamodellen der Prozessmodellierungsframeworks nachhaltig zu motivieren, wollen wir noch ein Beispiel vorstellen, das diese Notwendigkeit insbesondere aufzeigt. Wir betrachtet die Anpassbarkeit von Vorgehensmodellen [Kuh08], die auf zwei unterschiedlichen Ebenen vorgenommen werden kann:

- Inhaltliche Ebene
- Strukturelle Ebene

Die inhaltliche Ebene handelt projekt- sowie modellspezifische Anpassungen ab. Das kann beispielsweise in der projektspezifischen Interpretation eines Vorgehensmodells oder im Anpassen oder Ersetzen von Beschreibungstexten dargestellt werden. Die strukturelle Ebene teilt sich hingegen auf zwei Unterebenen auf:

- Modellebene
- Metamodellebene

Bei der Modellebene handelt es sich beispielsweise um das Anlegen von Elementen basierend auf existierenden Strukturen (z.B.: Hinzufügen einer neuen Beziehung zwischen zwei Produkten). Diese Art von Erweiterbarkeit beziehungsweise Anpassbarkeit bei einem existierenden Prozessmodell muss entsprechend im Metamodell, auf dem ein Prozessmodell basiert, vorgesehen werden. Somit ist das eine Aufgabe des Metamodells und nicht des Modells.

Auf der Metamodellebene geht es z.B. um das Hinzufügen von neuen Daten-, Klassen oder Beziehungstypen. Diese Erweiterungsart erlaubt es das Metamodell anzupassen, so dass es für spezifische Prozessmodelle geeignet ist. Hierzu muss das Metamodell sowie das gesamte Framework um das Metamodell eine entsprechende Unterstützung anbieten.

Anhand dieses Beispiels haben wir festgestellt, dass für die Modellierung von Prozessmodellen auch entsprechend geeignete Frameworks in der Hinsicht der Eignung ihrer Metamodelle gewählt werden müssen. Im Folgenden wollen wir einige solcher Frameworks hinsichtlich ihrer Bearbeitungsmöglichkeiten vorstellen sowie ihre Metamodelle auf erweiterte Konzepte wie die thematisierte Anpassbarkeit untersuchen.

3.2 V-Modell XT

Das V-Modell XT [FHKS09] beschreibt, wie viele andere Vorgehensmodelle auch, die Abläufe im Verlauf eines Entwicklungsprojekts über *Produkte*, *Rollen* und *Aktivitäten*. Eine besondere Einheit des V-Modells liegt in den so genannten *Vorgehensbausteinen*, die der Modularisierung von Abläufen dienen und somit flexible Zusammenstellungen erlauben.

Produkte stellen im V-Modell XT eine Information dar, die entweder im Verlauf des Projekts erarbeitet werden soll oder bereits dem Projekt zur Verfügung steht. Rollen legen Verantwortlichkeiten fest und Aktivitäten beschreiben wie einzelne Produkte zu erstellen sind. Vorgehensbausteine repräsentieren Teilprozesse in einem Projekt und bündeln somit Rollen, Produkte und Aktivitäten, sie beschreiben folglich welche Ergebnisse zu erstellen sind und wer dafür verantwortlich ist. *Projektdurchführungsstrategien* dienen weiterhin der Festlegung der Reihenfolge von *Entscheidungspunkten* und legen somit die Ergebnisse fest, die zu jedem bestimmten Ablaufszeitpunkt im Projekt vorzulegen sind. Entscheidungspunkte können hierbei als Meilensteine verstanden werden, sie haben allerdings im V-Modell XT eine erweiterte Bedeutung (dazu mehr in [FHKS09]). Projektdurchführungsstrategien eignen sich durch die möglichen Durchlaufreihenfolgen von Entscheidungspunkten für bestimmte Projekttypen. Hier kann folglich beim Design von solchen Strategien auch zielgerichtet bezüglich eines *Projekttyps* entwickelt werden. Allerdings bedürfen die so definierten Strategien noch einer Konfiguration, um beispielsweise ungewollte Abläufe auszuschließen.

Das V-Modell XT verfügt über ein formales Metamodell [TK09] und folglich ist das V-Modell XT als Instanz auch selbst wieder formal[1]. Somit ist eine maschinelle Verarbeitung sowie automatische Analysen möglich. Prozessmodelle können basierend auf dem V-Modell XT Metamodell erstellt werden, das allerdings sehr umfangreich und somit nur bedingt einsteigerfreundlich ist. Basierend auf der Dokumentation des Metamodells können dennoch auch die komplexeren Sachverhalte nachvollzogen werden, was allerdings aufgrund der Größe der Dokumentationen einen gewissen Aufwand voraussetzt.

Das V-Modell XT Metamodell wird über den V-Modell XT Editor zur Erstellung von Prozessmodellen benutzt. Den V-Modell XT Editor mit seinen Vor- und Nachteilen haben wir bereits in Abschnitt 1.2.2 betrachtet. Zusammenfassend muss man hier festhalten, dass die Werkzeugunterstützung für die Erstellung von Prozessmodellen basierend auf dem V-Modell XT Metamodell bei weitem nicht optimal ist. Es fehlen unter anderem grafische Darstellungsmöglichkeiten, um beispielsweise die genannten Projektdurchführungsstrategien besser nachvollziehen zu können, sowie sinnvolle Fehlerfindungs- beziehungsweise Fehlervorbeugungsmaßnahmen. Genauer genommen lassen sich fehlerhafte Abläufe mit dem V-Modell XT Editor erstellen und bleiben dem Prozessingenieur ohne weiteres verborgen. Erst beim Export des V-Modells kommen sie entweder über einen Fehler in der Dokumentation oder durch eine wenig aussagekräftige Fehlermeldung zum Vorschein. Allerdings können Prozessmodelle mit dem V-Modell XT Editor derzeit auch vollständig erstellt werden.

Das V-Modell XT Metamodell unterstützt modell- und projektspezifische Anpassungen. Erstere werden über die so genannten Änderungsoperationen unterstützt und erlauben beispielsweise das Anpassen von Beschreibungstexten oder Beziehungen. Für projektspezifische Anpassungen ist das so genannte Tailoring vorgesehen.

Das V-Modell XT Metamodell liegt in einer XML-Schema [W3Ca] Datei vor und kann als solche problemlos bearbeitet werden. Es steht dazu zwar kein spezifisches Werkzeug zur Verfügung, allerdings können strukturelle Anpassungen (z.B.: Hinzufügen von Datentypen oder Beziehungstypen) ohne weiteres vorgenommen werden. Solche Anpassungen werden vom V-Modell XT Editor auch direkt akzeptiert, so dass es bei der Instanziierung von Modellen zu keinen Problemen kommt. Einzige Minimalkritik ist das Fehlen eines Werkzeugs das solche Anpassungen erleichtert, schließlich können sich auch hier leicht Fehler einschleichen, die dann von Hand zu finden sind.

Die Dokumentation für ein Prozessmodell basierend auf dem V-Modell XT Metamodell wird anhand der Strukturen in einem solchen Prozessmodell automatisch generiert. Für das aktuelle V-Modell XT umfasst diese beispielsweise um die 800 Seiten. Anwendung findet das Prozessmodell mittels des Projektassistenten, der die projektspezifische Anpassung des V-Modells werkzeugtechnisch unterstützt sowie für die Generierung initialer Projektpläne sowie Produktbibliotheken

1 Instanzen des V-Modell XT Metamodells sind maschinenverarbeitbar.

genutzt werden kann, die ihrerseits weiterverarbeitenden Werkzeugen als Eingaben dienen können.

3.3 Eclipse Process Framework

Das Eclipse Process Framework Project (EPF) [EPF07] besteht aus zwei Komponenten:

EPF Composer ist eine auf Eclipse basierende Open-Source Anwendung zur Erstellung von Vorgehensmodellen.

OpenUP-Modell ist ein Softwareentwicklungsprozess, der an den Rational Unified Process (RUP) [LL07] angelehnt ist und von der Eclipse Foundation entwickelt wurde. Es beinhaltet die grundlegenden Eigenschaften des RUP, wozu

- iterative Softwareentwicklung
- use cases
- Szenarien
- Risikomanagement
- architekturzentriertes Vorgehen

zählen.

OpenUP ist ein Vertreter der SPEM-basierten [OMG05] Vorgehensmodelle und als leichtgewichtige Version des RUP zu verstehen. OpenUP wird bei EPF als ein möglicher Prozess angeboten, über Plugins können Scrum [Sch04] sowie eXtreme Programming (XP) [Bec03] als weitere Softwareentwicklungsmethodiken unterstützt werden. Das OpenUP-Modell lässt sich mittels zwei Bereichen thematisieren, den *Method* und *Process Content*.

Der *Method Content* definiert Strukturelemente wie Rollen, Tasks, Artefakte sowie sonstigen Inhalt. Der Method Content weist folglich Anleitungen auf, die schrittweise erklären wie ein spezifisches Entwicklungsziel erreicht werden kann, ohne allerdings eine Platzierung im Projektlebenszyklus vorzunehmen. Somit ist dieses Konzept mit den Vorgehensbausteinen aus dem V-Modell XT vergleichbar, schließlich machen beide keine Aussagen darüber, wann und wie oft Elemente verwendet oder erzeugt werden. Dafür ist bei OpenUP der Process Content verantwortlich.

Der *Process Content* stellt im Wesentlichen die Elemente des Method Content in eine Ordnung mittels so genannter *Capability Patterns*, indem es beschreibt wann die Elemente zu erstellen (Artefakte) und auszuführen sind (Tasks). Dabei fassen die Capability Patterns die Elemente des Method Content zu einer Einheit zusammen, können aber auch selbst zu größeren Einheiten komponiert werden. So können Patterns spezielle Projektbereiche wie beispielsweise Iterationsmanagement oder Architekturdefinition fokussieren (sie dienen den erweiterten Patterns dann als Basisblöcke) oder aber sie selbst stellen einen Prozess dar, der definierte Patterns als Basisblöcke mitverwendet.

Das OpenUP-Modell verfügt über ein formales Metamodell, eine Diskussion der zugehörigen Teilmodelle (Produkt-, Rollen- oder Aktivitätsmodell) können [Kuh08] entnommen werden. OpenUP selbst, wie auch andere Softwareentwicklungsmethodiken können bei EPF stark angepasst und erweitert werden. Hier ist EPF auf einer Stufe mit dem V-Modell XT anzusiedeln, wobei die Möglichkeit eines parametrisierbaren projektspezifischen Tailorings in EPF fehlt.

Das Authoring von OpenUP wird mittels des EPF-Composers durchgeführt, der hierzu die notwendigen Modellierungsmöglichkeiten bereitstellt. Der EPF-Composer selbst ist von der angebotenen Funktionalität dem V-Modell XT Editor überlegen, wird hierdurch allerdings für einen unerfahrenen Benutzer schwer erlernbar, da es anfangs überladen wirkt. Der EPF-Composer verfügt, im Gegensatz zum V-Modell XT Editor, über grafische Modellierungsmöglichkeiten um beispielsweise Aktivitätsdiagramme definieren zu können, diese können allerdings mit der einfachen Definitionsweise von Prozessen im PDS Designer nicht mithalten, schließlich erlaubt dieser neben der Erstellung von Teilprozessen auch die Darstellung kompletter Prozesse.

Für ein OpenUP basierendes Prozessmodell wird eine zugehörige Dokumentation automatisch generiert und liegt in einem Web-basierten Format vor. Zusätzlich werden Projektvorlagen erstellt. Ein weiteres Werkzeug, wie der V-Modell XT Projektassistent, fehlt EPF[2], so dass hier keine zum V-Modell XT vergleichbare weitreichende Werkzeugkette vorliegt.

2 Diese Aussage schränken wir nur auf EPF mit OpenUP ein, für kommerzielle Ausprägungen wie dem RUP stehen umfassende Werkzeuge zur Verfügung.

3.4 Zusammenfassung

In den obigen Abschnitten haben wir zwei Möglichkeiten zur Modellierung von Vorgehensmodellen kennengelernt. Beide haben spezifische Vor- und Nachteile vor allem in der Werkzeugunterstützung, die bei beiden Prozessmodellen nicht optimal ist. Insbesondere fehlen einfache und leicht zu benutzende grafische Modellierungsmöglichkeiten, die nicht nur Prozessdefinitionen erlauben, sondern mehrere Teilmodelle eines Vorgehensmodells gebührend abbilden sowie eine spezifische Validierung von Instanzen der Modelle, die dem Prozessingenieur bei der Fehlererkennung behilflich ist. Genau diese beiden Punkte sind auch bezüglich der Werkzeugunterstützung mittels einer DSL für das V-Modell XT Gegenstand dieser Arbeit.

Die beiden in diesem Kapitel betrachteten Prozessmodelle basieren auf einem formalen Metamodell. Die Formalität ist für eine automatische Verarbeitbarkeit, Erweiterungs- und Anpassungsmöglichkeiten sowie computergestützte Analysen notwendig, sie stellt zudem auch eine Voraussetzung für die Umsetzung eines Vorgehensmodells als domänenspezifische Sprache dar. Das wird bereits mittels des Prinzips der domänenspezifischen Entwicklung angeregt und aufgrund der Komplexität von Prozessmodellen an sich auch so definiert. Schließlich ist die Ableitung eines allgemeinen Modells basierend auf einer reinen Prozessbeschreibung nicht nur eine schwere Aufgabe, sie ist zudem mit der Problematik verbunden, dass bereits bestehende Modelle unterstützt werden sollen (zumindest bezüglich der selben Prozessmodellversion). Die Formalität ist folglich für eine Umsetzung von Vorgehensmodellen zu DSLs wichtig, sofern hierbei bereits existierende Instanzen unterstützt werden sollen. Müssen hingegen vorhandene Modelle nicht unterstützt werden, so stellt sich bei einem Prozessmodell, das über ein nicht formales Metamodell verfügt und anhand von beispielsweise Prozessbeschreibungen umgesetzt werden soll, die Frage, ob hierdurch nicht zwangsläufig ein neues Prozessmodell entwickelt wird, was in gewisser Weise auf dem zugehörigen Basisprozessmodell aufbaut.

In dieser Arbeit verwenden wir für die Entwicklung einer domänenspezifischen Sprache ein formales Metamodell, so dass sich das obige Problem nicht stellt. Durch die Formalität können (und sollen) folglich bereits erstellte Instanzen des V-Modell XT Metamodells zumindest bezüglich der gleichen Metamodellversion auch in der Sprache unterstützt werden.

4 Analyse: V-Modell XT und Domänenspezifische Sprachen

In diesem Kapitel wollen wir uns mit dem V-Modell XT, genauer mit den Strukturen zur Definition des V-Modells, sprich mit seinem Metamodell, beschäftigen, um hierfür die Umsetzung in eine domänenspezifische Sprache anzuregen. Eine generelle Umsetzung lässt sich auf unterschiedliche Arten thematisieren, die alle spezifische Vor- und Nachteile aufweisen. Folglich will dieses Kapitel genau diese Möglichkeiten gegenüberstellen und im Besonderen bezüglich der Anwendung auf das V-Modell XT Metamodell untersuchen. Schließlich betrachtet dieses Kapitel zusätzlich die Anforderungen an eine Sprache für das V-Modell, die insbesondere im Bereich der Visualisierung diskutiert werden.

Übersicht

4.1 V-Modell XT Metamodell

Das V-Modell XT ist ein anerkanntes Vorgehensmodell der Softwareentwicklung und stellt das Standardvorgehensmodell der Bundesrepublik Deutschland zur Entwicklung von IT-Systemen dar. Das V-Modell XT basiert auf einem formalen Metamodell [TK09], das im Rahmen dieses Abschnitts vorgestellt wird. Das V-Modell XT selbst ist ein recht umfangreiches und kompliziertes Prozessmodell, so dass das Metamodell hierfür auch dementsprechend komplex ist. Im Folgenden wollen wir zunächst einen Überblick über das Gesamtmetamodell geben und schließlich als Beispiel die Projektdurchführungsstrategien näher vorstellen.

Das V-Modell XT Metamodell lässt sich in fünf Paketbereiche strukturieren (siehe Abbildung 4.1), die das Metamodell zum leichteren Verständnis logisch gliedern (in der physikalischen Struktur ist diese Gliederung nicht notwendigerweise vorzufinden).

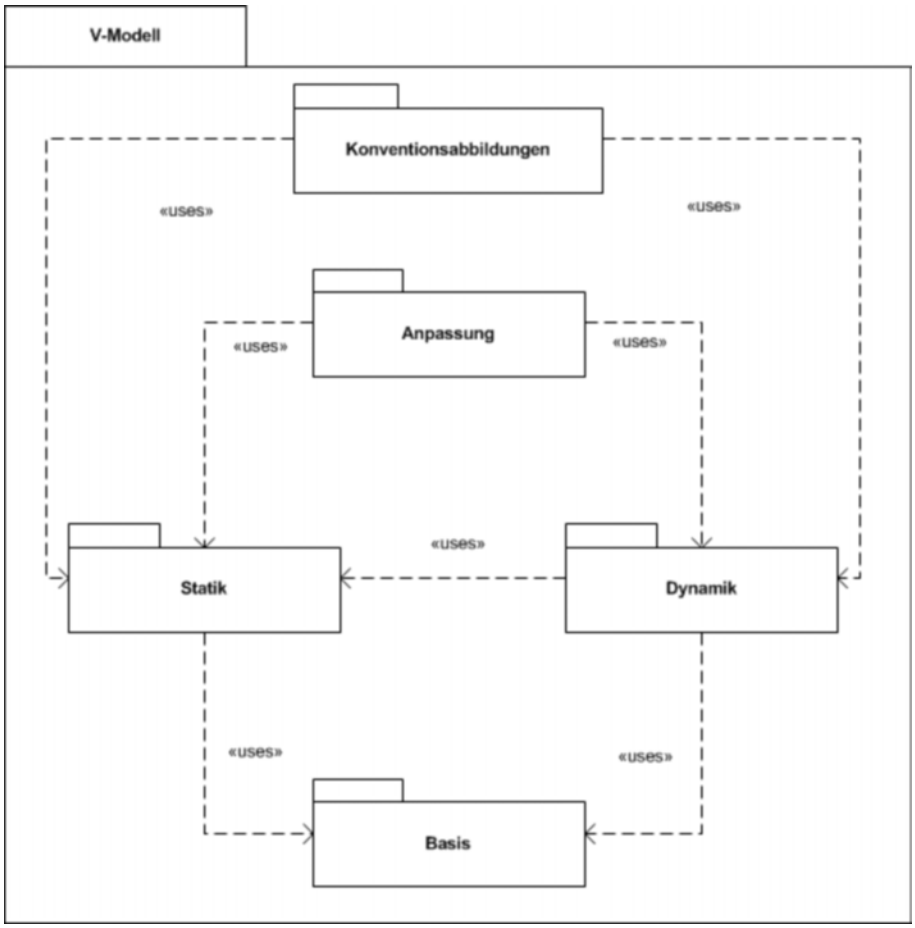


Abbildung 4.1: Überblick über das V-Modell XT Metamodell, Quelle: [TK09]

Paket Basis beschreibt, wie der Name schon andeutet, die Basisstrukturen des Metamodells sowie den prinzipiellen Aufbau der Dokumentation, die auch im Modell verankert ist. Zudem enthält dieses Paket Elemente, die seitens anderer Paketelemente referenziert werden (z.B.: Quelle oder Abkürzungen).

Paket Statik beinhaltet die Strukturen zur Definition von Vorgehensbausteinen, die die wesentlichen Strukturen zur Beschreibung des V-Modells darstellen. Vorgehensbausteine selbst sind modulare Einheiten, die weitere Beschreibungselemente wie Produkte, Aktivitäten, Disziplinen oder Beziehungen kapseln.

Paket Dynamik beschreibt die Elemente und Beziehungen zur Konstruktion von Projektdurchführungsstrategien, auf die wir später noch näher eingehen werden.

Paket Anpassung fasst alle Elemente zusammen, die der Anpassung des V-Modells, insbesondere projektspezifischer Art, dienen. Hierdurch können Anwender das V-Modell auf ihre Bedürfnisse reduzieren beziehungsweise spezialisieren.

Paket Konventionsabbildungen beschreibt die Abbildung anderer Begriffe auf die Konventionen des V-Modells.

Das V-Modell XT Metamodell ist sehr umfangreich, so dass eine genaue Betrachtung aller Elemente und Beziehungen den Rahmen dieser Arbeit sprengen würde. Deshalb wollen wir uns folgend die Projektdurchführungsstategien (Paket Dynamik) exemplarisch herausgreifen. Für weitere Informationen, insbesondere zu den weiteren Paketen, sei der interessierte Leser auf [TK09] verwiesen.

4.1.1 Projektdurchführungsstrategien – Paket: Dynamik

Zur Definition von Projektdurchführungsstrategien [WKK09b] stellt das Metamodell des V-Modells als Hauptelement einen *Ablaufbaustein* (Abbildung 4.2) zur Verfügung, der einerseits für eine Projektdurchführungsstrategie oder für einen *Unterablauf* stehen kann. Jeder Ablaufbaustein referenziert genau eine *Ablaufbausteinspezifikation*, die die Anforderungen an einen Ablaufbaustein hinsichtlich enthaltener Entscheidungspunkte sowie ihrer Reihenfolge festlegt. Zur weiteren Modellierung der Ablaufbausteine werden Ablaufpunkte und Übergänge verwendet, die den Prozess innerhalb eines Ablaufbausteins definieren. Das sind je genau ein *Startpunkt* und ein *Endepunkt*, die die Ein- und Austrittsspunkte repräsentieren und weiterhin *Ablaufbausteinpunkte*, *Ablaufentscheidungspunkte* sowie *Splits* und *Joins*.

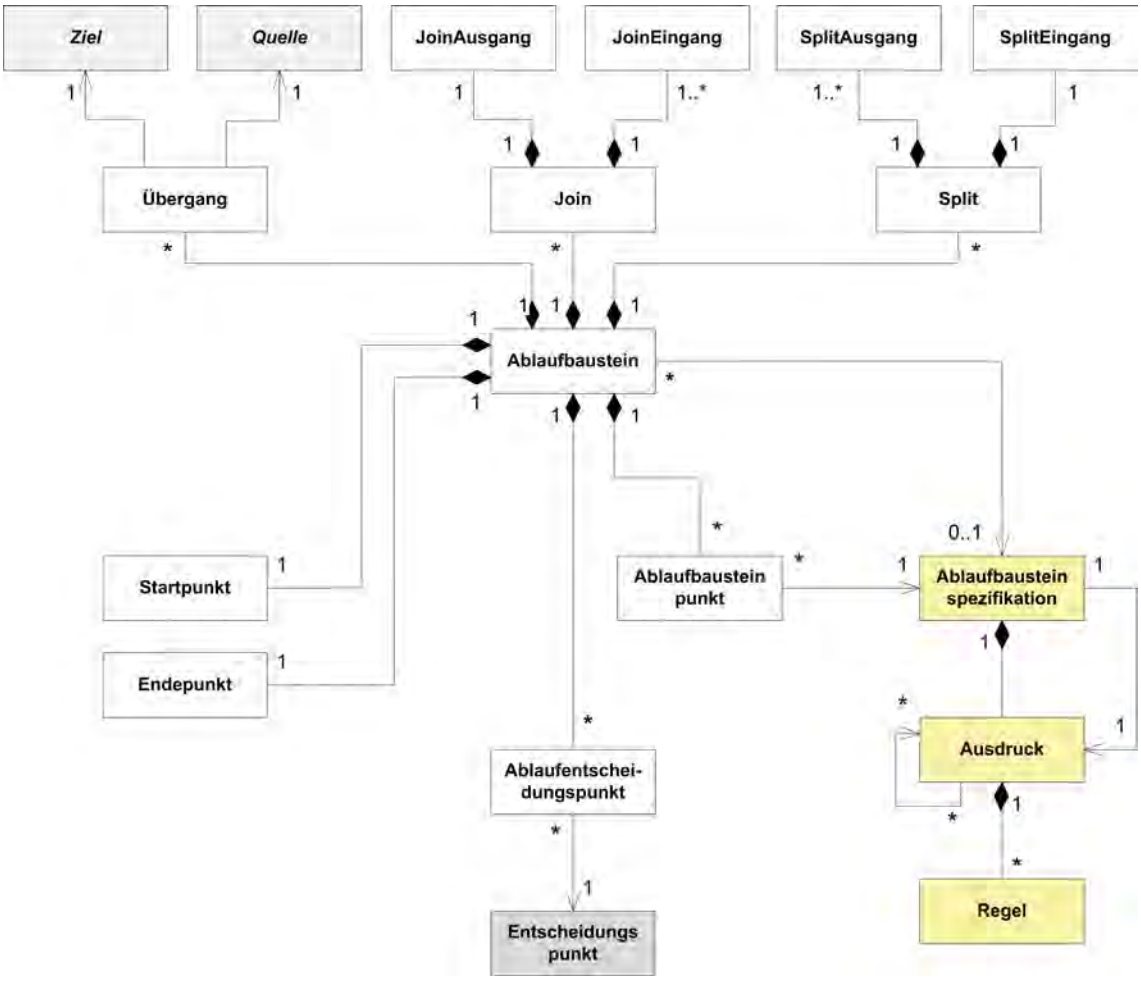


Abbildung 4.2: Modell zur Definition von Projektdurchführungsstrategien

Ablaufbausteinpunkt Ein Ablaufbausteinpunkt stellt eine Station im Ablauf des jeweiligen Ablaufbausteins dar. Er verweist auf genau eine Ablaufbausteinspezifikation, so dass an dieser Stelle weitere Ablaufbausteine integriert werden können, vorausgesetzt, sie erfüllen die referenzierte Ablaufbausteinspezifikation. Die Verwendung von Ablaufbausteinpunkten er-

laubt eine modulare Modellierung von Projektdurchführungsstrategien. Die tatsächlich relevanten Ablaufbausteine werden dabei erst zum Zeitpunkt des Tailorings festgelegt. Zum Designzeitpunkt sind nur die „Platzhalter" (Ablaufbausteinspezifikationen) und die möglichen „Kandidaten" (die der Spezifikation entsprechenden Ablaufbausteine) bekannt.

Ablaufentscheidungspunkt Ein Ablaufentscheidungspunkt ist eine Station im Ablauf des zugeordneten Ablaufbausteins und stellt eine Verbindung zu genau einem Entscheidungspunkt her, so dass so die Entscheidungspunkte in eine Projektdurchführungsstrategie integriert werden. Im Gegensatz zu Ablaufbausteinpunkten, stellen Ablaufentscheidungspunkte *konkrete* Stationen im Ablauf einer Projektdurchführungsstrategie dar und können nicht weiter verfeinert werden.

Split Ein Split ist eine Struktur, die die Modellierung von parallelen Abläufen erlaubt. Dazu enthält jeder Split genau einen Spliteingang und mindestens einen Splitausgang. Der Splitausgang in diesem Sinne dient als Startpunkt der Parallelität, so dass alle Übergänge ausgehend von einem Splitausgang im Projekt parallel geplant und ausgeführt werden können.

Join Ein Join stellt eine Struktur dar, die es erlaubt, parallele Abläufe wieder zusammenzuführen. Dazu stellt jeder Join mindestens einen Eingang und genau einen Ausgang bereit.

Um den Ablauf innerhalb eines Ablaufbausteins zu modellieren werden Übergänge eingesetzt, die jeweils gerichtete Verbindungen zwischen zwei Ablaufpunkten darstellen.

4.1.2 Projektdurchführungsstrategien – Paket: Anpassung

Die Definition von Ablaufbausteinen allein für eine Projektdurchführungsstrategie genügt nicht. Diese müssen entsprechend im Rahmen des Tailorings *konfiguriert* werden (Abbildung 4.3), um unzulässige beziehungsweise ungewollte Abläufe auszuschließen. Dies wird über Paket Anpassung erledigt [TK09, WKK09b].

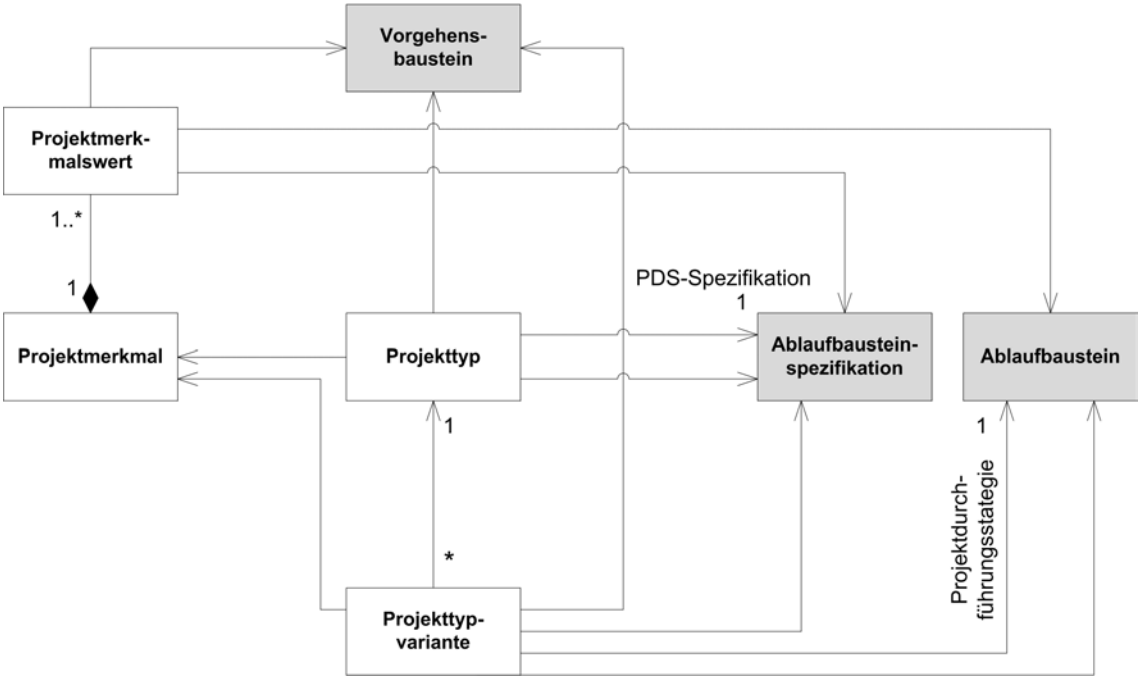


Abbildung 4.3: Metamodell – Sicht: Anpassung (Tailoring)

Dazu dienen zwei Elemente, der *Projekttyp* und die *Projekttypvariante*, mit deren Hilfe sich die Projektdurchführungsstategien bezüglich einer Klassifizierung von Projekten in der Auswahl der zulässigen und gewollten Ablaufbausteine und folglich der möglichen Abläufe einschränken lassen.

Projekttyp Ein Projekttyp stellt eine Klassifizierung der gleichartigen Projekte dar. Er legt die zu verwendenden Ablaufbausteinspezifikationen sowie die Spezifikation, die von der verwendeten Strategie zu erfüllen ist, fest.

Projekttypvariante Die Projekttypvariante verfeinert einen Projekttyp. Jede Projekttypvariante ist daher genau einem Projekttyp zugewiesen. Sie stellt den Rahmen für die Projektdurchführungsstrategie und legt zusätzliche Ablaufbausteinspezifikationen sowie die konkreten Ablaufbausteine der Strategie fest.

Durch die Konfiguration werden zuvor modellierte Projektdurchführungsstrategien sowie ihre jeweils zulässigen Abläufe konkretisiert.

Abbildung 4.4 zeigt beispielhaft die Projektdurchführungsstrategie des Auftragnehmers für den Anwendungsbereich *Entwicklung*. In der Abbildung wird gezeigt, welche möglichen Vorgehensweisen aufgrund des Tailorings für ein Projekt verfügbar sind. Konkret betrifft dies die Entwicklungsabläufe:

- Komponentenbasierte Systementwicklung und
- Inkrementelle Systementwicklung,

die standardmäßig wegen der gewählten Projekttypvariante zur Verfügung stehen. Hinzu kommen die *Prototypische Systementwicklung* und der *Unterauftrag*, die jeweils durch Projektmerkmale während des Tailorings verfügbar gemacht wurden.

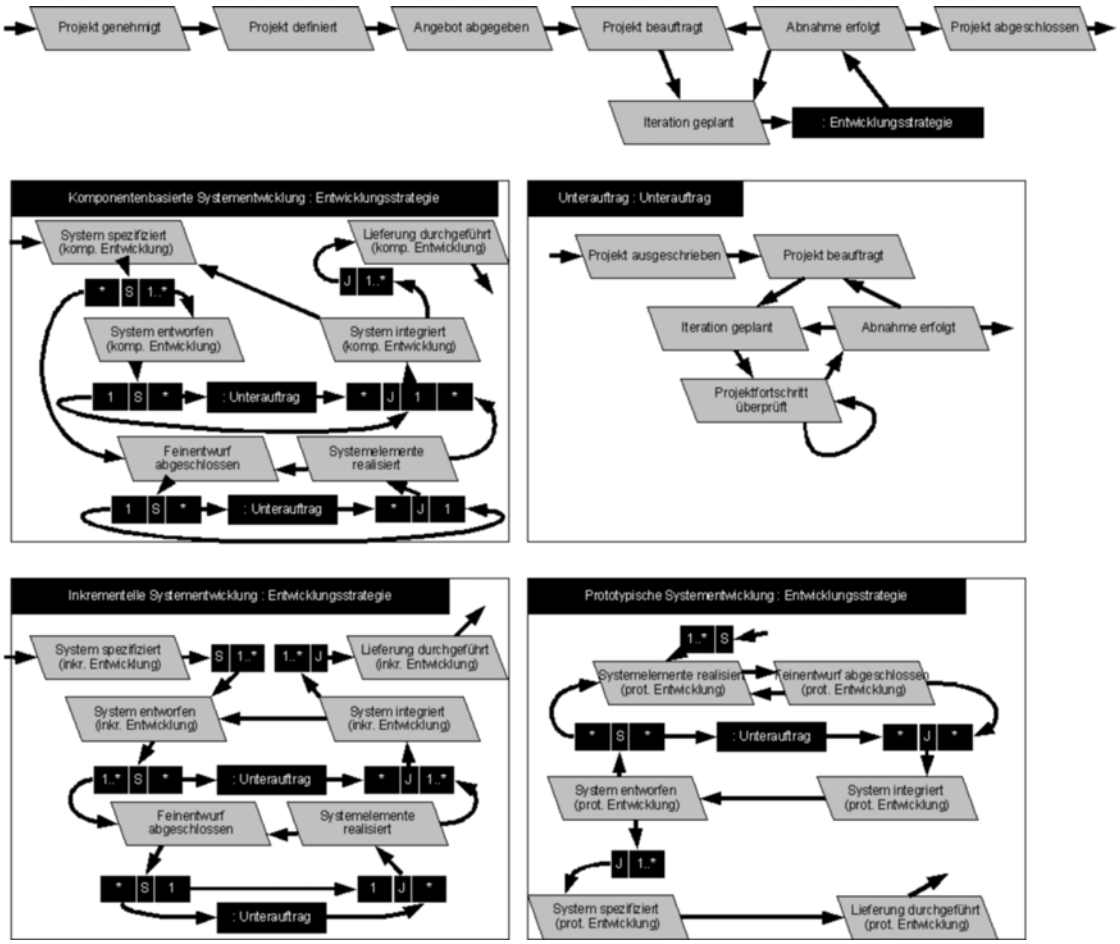


Abbildung 4.4: Projektdurchführungsstrategie des Auftragnehmers (Entwicklung)

Die „Andockpunkte" (durch Ablaufbausteinpunkte in den Ablaufbausteinen angegeben, siehe letzter Abschnitt), sind in der Abbildung hervorgehoben. Jeder der Ablaufbausteine sagt dabei aus, welcher Ablaufbausteinspezifikation er genügt. Der konkrete Ablaufbaustein *Unterauftrag* genügt beispielsweise der Ablaufbausteinspezifikation *Unterauftrag*.

4.2 Umsetzungsmöglichkeiten und Probleme

Für die Implementierung des V-Modell XT Metamodells als domänenspezifische Sprache gibt es mehrere Möglichkeiten. In diesem Kapitel wollen wir einige davon vorstellen, wir behandeln sie

allerdings im speziellen für die Microsoft DSL-Tools, da wir diese im späteren Verlauf der Arbeit für die konkrete Implementierung der Lösungsarchitektur mitverwenden werden (Das bedeutet allerdings nicht, dass die besprochenen Ansätze explizit nur in Verbindung mit den DSL-Tools gelten, die Evaluierung der Ansätze 1 und 2 ist allerdings basierend auf den DSL Tools abgeleitet worden).

4.2.1 Ansatz 1: Direkte Umsetzung

Bei einer direkten Umsetzung wird das vorliegende Metamodell direkt im Domänenmetamodell der domänenspezifische Sprache implementiert (schematisch dargestellt in Abbildung 4.5). Dabei werden Elemente und Beziehungen des Metamodells in ihre jeweiligen Repräsentanten im Domänenmetamodell abgebildet. Beispielsweise wird dann aus dem *Ablaufbaustein* des V-Modell XT bei den DSL-Tools eine DomainClass mit demselben Namen hinterlegt. Darstellungsdefinitionen sowie alle weiteren notwendigen Eigenschaften werden weiterhin, soweit möglich, im Domänenmetamodell getroffen.

Hinweis

Bei der Implementierung des PDS-Designers (siehe Abschnitt 1.2.2) wurde der hier vorgestellte direkte Ansatz gewählt.

Abbildung 4.5: Schematische Darstellung von Umsetzungsansatz 1

Das Metamodell des V-Modell XT kann auf diese Art und Weise mittels der DSL-Tools implementiert werden (siehe auch [WKK09b]), erfordert allerdings zusätzliche Anpassungen:

- Serialisierung: Um bestehende V-Modell XT Instanzen laden und speichern sowie neue anlegen zu können, muss die Serialisierung für das spezifische Speicherformat des V-Modells angepasst werden, schließlich sollen V-Modell Instanzen, die mittels der neuen DSL erstellt oder bearbeitet werden auch zu den bereits vorhanden Werkzeugen kompatibel sein (Eine mögliche Abhandlung der Kompabilität über einen Import und Export bedarf nach wie vor einer entsprechenden Serialisierung).
- Darstellung: Der mit den DSL-Tools erstellte Editor stellt alle Elemente, die instanziiert sind und über eine Definition der entsprechenden grafischen Darstellung verfügen, automatisch auf der Designoberfläche dar. Das ist für das V-Modell nicht sinnvoll, so dass hier umfassende Anpassungen zu treffen sind, um eine selektive grafische Darstellung zu erreichen. Zudem sind die DSL-Tools nicht besonders flexibel bei der Art und Weise der Darstellung, verfügen aber über die Möglichkeit zum Verzicht auf die klassischer Designeroberfläche und den Umstieg auf eine *Windows Presentation Foundation* (WPF)-seitige Visualisierung. Der Nachteil hiervon liegt in der derzeit fehlenden Unterstützung, die sich darin ausdrückt, dass bei einer WPF-seitigen Darstellungswahl die entsprechenden Darstellungsstrukturen vollständig manuell zu implementieren sind (genauer genommen ist die gesamte Diagrammoberfläche komplett zu programmieren).
- Anpassungen für die Modellierung: Hier sind Anpassungen bezüglich der Eigenschaftsbearbeitung und der Beziehungsdefinition zu treffen, die bei bestimmten Elementen eingeschränkt oder ganz verboten werden müssen.
- Erweiterbarkeit: Für das V-Modell XT existieren bereits einer Reihe von verschiedenen Werkzeugen. Im Rahmen der neuen DSL für das V-Modell ist auch eine Möglichkeit zum Zusammenspiel über eine Plugin-Schnittstelle notwendig. Bei der direkten Implementierung mit den DSL-Tools liegt der erstellte Editor aber selbst als Plugin für das Visual Studio vor, so dass es hier problematisch beziehungsweise aufwendig ist eine entsprechende Schnittstelle bereitzustellen.

Insgesamt ist Ansatz 1 zwar möglich und bei einer einmaligen Implementierung möglicherweise auch vertretbar, allerding für ein sich im Metamodell-Bereich ständig weiterentwickelndes Modell weder optimal noch tragbar. Jede Änderung am Metamodell würde großen Aufwand bei der Anpassung der DSL nach sich ziehen, so dass die Vorteile der Nutzung einer DSL in dieser Hinsicht als große Nachteile gesehen werden können. (Nochmals sei darauf hingewiesen, dass die Beurteilung dieses direkten Ansatzes insbesondere auf den Erkenntnissen bei der Implementierung des PDS-Designers [WKK09b] basiert).

4.2.2 Ansatz 2: Komponentenbasierte Umsetzung

Eine komponentenbasierte Entwicklung domänenspezifischer Sprachen [KRV08] erlaubt es - ähnlich zu modularer Softwareentwicklung - neue Sprachen aus bereits existierenden zusammenzustellen. Das führt zum einen zur Komplexitätsreduktion zumindest bei den einzelnen Komponenten und zum anderen ergibt sich hieraus der Vorteil der Wiederverwendung, der bei Neuentwicklung bereits implementierte Sprachen miteinbezieht.

Bei der Anwendung der komponentenbasierten Entwicklung auf domänenspezifische Sprachen lassen sich mehrere Möglichkeiten thematisieren (siehe auch Abbildung 4.6):

1. Zusammenstellung der DSL aus vorhandenen Sprachen: Die Idee der Komposition einer DSL aus bereits existierenden Sprachen basiert darauf, dass sich Teile des Metamodells auch tatsächlich als Teilsprachen implementieren lassen. Die Gesamt-DSL wählt nach dem Kompositionsprinzip die notwendigen Sprachen aus und fügt sie zu einer Gesamtsprache zusammen.
2. Erstellung der DSL über die Erweiterung einer vorhandenen Sprache: Eine DSL wird als Erweiterung einer bereits vorhandenen Sprache implementiert, indem es beispielsweise die Sprache um weitere Elemente, Beziehungen oder Eigenschaften ergänzt.
3. Verbindung beider oben genannter Möglichkeiten: Bei der Verbindung der beiden Ansätze können Sprachen, die bei der Komposition miteinbezogen werden sollen, auch zunächst ergänzt werden, genauso kann die zusammengestellte Gesamtsprache auch für eine spezifisch notwendige Ausprägung um Ergänzungen bereichert werden.

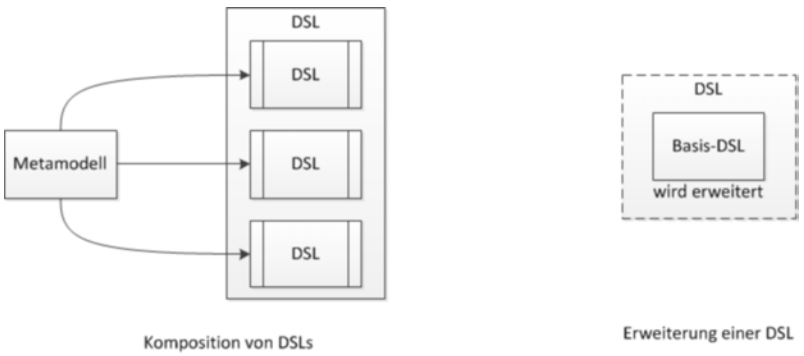


Abbildung 4.6: Schematische Darstellung von Umsetzungsansatz 2

Für die Anwendung auf das V-Modell XT müssen wir nur die Möglichkeit 1 diskutieren, denn Möglichkeit 2 ist hier nicht anwendbar, eine bereits vorhandene DSL, die das V-Modell XT Metamodell als Ganzes implementiert, ist nicht vorhanden und die Erweiterung des PDS-Designers zur Unterstützung des gesamten V-Modells ist aus Gründen, die bei Ansatz 1 (siehe 4.2.1) diskutiert wurden, nicht sinnvoll. Somit muss auch Möglichkeit 3 im Rahmen des V-Modells nicht berücksichtig werden.

Angewendet auf das V-Modell XT, wären bei Möglichkeit 1 fünf separate Sprachen zu entwickeln, je Paket (siehe auch Kapitel 4.1) eine eigene Sprache. Die so entwickelten Sprachen wären allerdings, zumindest teilweise, voneinander abhängig (z.B.: Sprache für Paket Anpassung von der Sprache für Paket Dynamik). Somit stellt sich hier die Frage, ob eine derartige Anwendung wirklich sinnvoll ist. Andererseits, ließe sich das V-Modell XT Metamodell auch beispielsweise mittels drei Sprachen strukturieren, so dass zumindest die Pakete Basis und Statik als eigene DSLs zur

Verfügung gestellt werden können, denn bei diesen kann erwartungsgemäß von sehr wenigen Änderungen ausgegangen werden.

Unabhängig von einer Implementierungswahl bei Möglichkeit 1 bleibt die Problematik der notwendigen Anpassungen bei der Implementierung mit den DSL-Tools bestehen. Diese verlagert sich nur auf die einzelnen Sprachen, ist aber nach wie vor problematisch wie beim bereits diskutierten Ansatz 1 (siehe 4.2.1). Beachtenswert ist allerdings, dass in diesem Fall zumindest einige Sprachen nach der einmaligen Implementierung relativ statisch blieben, so dass sich hieraus ein gewisser Vorteil gegenüber dem direkten Ansatz ergibt. Somit bietet dieser Ansatz eine gewisse, wenn auch nur minimale, Verbesserung gegenüber dem direkt Ansatz für die Umsetzung des V-Modells in eine domänenspezifische Sprache.

4.2.3 Ansatz 3: DSL zur Erzeugung von DSLs

Bei den zuvor betrachteten Ansätzen setzten wir den Einsatz eines bereits existierenden DSL Frameworks voraus (siehe auch Kapitel 2), um damit die von uns gewünschte DSL zu erstellen. Das hat einerseits den Vorteil, dass ein bereits ausgereiftes und vorhandenes Framework zum Einsatz kommt, andererseits muss aber die Ziel-DSL mit den festgelegten Basisstrukturen implementiert werden, so dass an unterschiedlichen Stellen umfangreiche Anpassungen im Fall des V-Modells (z.B.: bei der Serialisierung) notwendig sind.

Ein weiterer möglicher Ansatz ist die Erstellung einer domänenspezifischen Sprache, die dem Erstellen weiterer DSLs dient (schematisch dargestellt in Abbildung 4.7). Diese Basis-DSL kann darauf ausgerichtet sein bestimmte Implementierungsaspekte der Ziel-DSL zu erleichtern. So lassen sich Serialisierungsstrukturen derartig konzipieren, dass sich damit zwar beliebige Modelle speichern und laden lassen, sie allerdings im Falle des V-Modells eine Serialisierung in das aktuelle Format erlauben. Ferner müssen bei der Modellierung einer solchen Basis-DSL die folgenden Aspekte für die Ziel-DSL betrachtet werden:

- Modellierungsstrukturen: Elemente und Beziehungen, die dem Modellieren des Domänenmetamodells der Ziel-DSL dienen sollen.
- Visualisierungsstrukturen: Möglichst flexible Darstellungsmöglichkeiten in der Ziel-DSL.
- Serialisierung: Strukturen zum Festlegen der Serialisierungsklassen, die automatisch erstellt das Speichern und Laden in der Ziel-DSL erlauben sollen.
- Validierung: Unterstützung der Validierung in der Ziel-DSL.

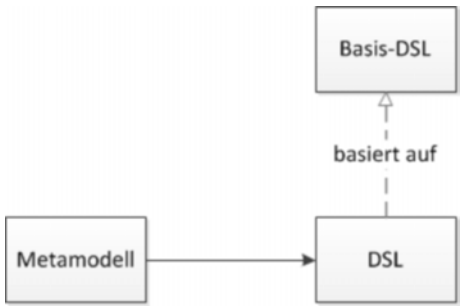


Abbildung 4.7: Schematische Darstellung von Umsetzungsansatz 3

Angewendet auf das V-Modell XT liefert dieser Ansatz eine deutliche Verbesserung gegenüber den Ansätzen 1 und 2. Der große Vorteil liegt in der möglichen Spezialisierung der Basis-DSL insbesondere auf das V-Modell XT, ohne allerdings damit notwendigerweise andere Anwendungsmöglichkeiten auszuschließen. Die Spezialisierung erlaubt es das große Problem der Wartbarkeit handhabbar zu gestalten, so dass auch bei Änderungen am Metamodell die Aktualisierung der zugehörigen DSL mit einem akzeptablen Aufwand verbunden ist. Zudem ist die Spezialisierung dem Vermeiden von Fehlern dienlich, wie sie beispielsweise bei einer eigens geschriebenen Serialisierung des V-Modells (aufgrund der Größe des Metamodells) durchaus auftreten könnten.

Der große Nachteil bei diesem Ansatz liegt in der notwendigen Implementierung einer solchen Basis-DSL. Das ist zum einen mit einem erheblichen Aufwand verbunden, zum anderen ist eine

derartige Implementierung eine durchaus schwere Aufgabe. Generell ist die Erstellung von domänenspezifischen Sprachen nicht einfach, die Modellierung einer DSL zum Erstellen von weiteren DSLs ist allerdings mit einer beträchtlich höheren Komplexität verbunden. Zudem muss aufgrund der Spezialisierung die Basis-DSL zwar so allgemein wie möglich, aber zugleich auch so spezifisch für das V-Modell XT wie notwendig konzipiert werden.

4.2.4 Ansatz 4: DSL zur Erzeugung von DSLs und komponentenbasierte Umsetzung

Der letzte der hier betrachteten vier Ansätze stellt lediglich die Anwendung der komponentenbasierten Umsetzung auf Ansatz 3 dar (schematisch dargestellt in Abbildung 4.8). So basieren hier die einzelnen Teilsprachen auf der betrachteten Basis-DSL und werden von der Ziel-DSL entsprechend der Notwendigkeit kombiniert. Die bei diesem Ansatz möglichen Anwendungsmöglichkeiten wurden bereits bei Ansatz 2 (siehe auch Abschnitt 4.2.2) thematisiert und gelten hier genauso. Einzig die Vor- und Nachteile der diskutierten ersten Möglichkeit, bei der die Ziel-DSL aus bereits existierenden Sprachen zusammengesetzt wird, wollen wir hier nochmals untersuchen.

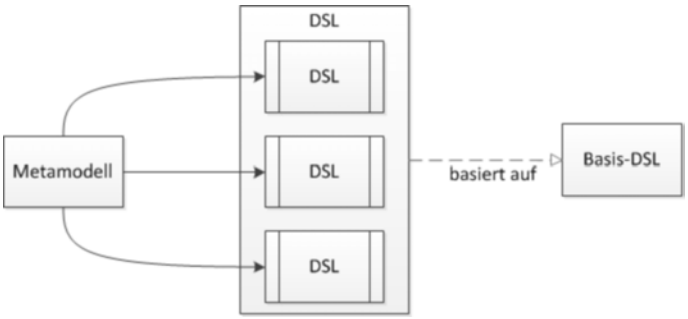


Abbildung 4.8: Schematische Darstellung von Umsetzungsansatz 4

Hinweis

Eine weitere mögliche Strukturierung des V-Modells wäre die Aufteilung des Metamodells auf drei Sprachen für die Pakete Basis, Statik und das restliche Modell. Diese Ansatzmöglichkeit hat allerdings dieselbe Problematik wie der diskutierte Ansatz, reduziert jedoch die vorhandenen Abhängigkeiten zwischen den Sprachen erheblich.

Angewendet auf das V-Modell XT bietet dieser Ansatz an, die Implementierung der Ziel-DSL mittels fünf Sprachen (gemäß den logischen Paketen, siehe Abschnitt 4.1) zu strukturieren:

1. Paket Basis DSL: Hier sind die Elemente des Pakets Basis enthalten. Diese weisen keine Beziehungen zu den übrigen Paketen auf und sind zudem größtenteils statisch bezüglich möglicher Veränderungen im Metamodell.
2. Paket Statik DSL: Hier sind Elemente und Beziehungen des Pakets Statik enthalten. Diese DSL steht in Abhängigkeit zur Paket Basis DSL.
3. Paket Dynamik DSL: Hier werden Elemente und Beziehung aus dem Paket Dynamik modelliert. Diese DSL ist abhängig von der Paket Statik DSL, es werden beispielsweise die Entscheidungspunkte aus demselben Paket referenziert.
4. Paket Anpassung DSL: Diese DSL fasst Strukturen zum Anpassen des V-Modells zusammen. Sie ist abhängig von der Paket Dynamik DSL als auch von der Paket Statik DSL.
5. Paket Konventionsabbildung DSL: Diese DSL beinhaltet die Strukturen des Pakets Konventionsabbildungen. Sie ist abhängen von allen oben genannten DSLs.

Die Ziel-DSL für das V-Modell XT würde sich demnach aus allen oben genannten DSLs zusammensetzen. Einerseits hätte eine derartig kombinierte DSL den Vorteil einer tatsächlich physikalischen Trennung der einzelnen Pakete. So wären die einzelnen Pakete erfassbarer und könnten, bis auf die Beziehungen, unabhängig voneinander implementiert und später verwaltet werden. Die

Beziehungen sind es aber, die eine tatsächliche Implementierung einer solchen Gesamt-DSL erschweren. Diese erfordern eine explizite Einbindung der jeweiligen referenzierten DSLs, so dass der Verwaltungsaufwand beim Gesamtsystem womöglich recht groß ist. Als Beispiel für die Problematik lassen sich einfache Änderungsoperationen wie z.B. das Löschen von Elementen aus der Paket Statik DSL betrachten. Beim Löschen von Elementen, die von bestimmten anderen DSLs referenziert werden, lässt sich in der Paket Statik DSL nicht feststellen, ob die Elemente beziehungsweise Beziehungen tatsächlich referenziert und somit in den anderen DSLs benötigt werden. Löscht man nun diese Elemente, so kann es hier zu schwerwiegenden Fehlern kommen, da insbesondere eine derartige Löschung später in den übrigen DSLs nicht unbedingt einfach nachvollzogen werden kann.

Somit kann es bei einer derartige Implementierung insbesondere bei Änderungsoperationen an den DSLs, die von anderen referenziert werden, zu Problemen kommen. Im Fall des V-Modells gibt es recht viele Abhängigkeiten zwischen den vorgeschlagenen DSLs. Allerdings sind die einzelnen Abhängigkeiten größtenteils nur bei der Löschung von Elementen problematisch, so dass bei einer Implementierung und späteren Verwaltung insbesondere darauf geachtet werden müsste. Trotz der oben genannten Schwierigkeiten wäre eine Implementierung der Gesamt-DSL, wie sie dieser Ansatz vorstellt, somit durchaus denkbar.

4.2.5 Zusammenfassung und Bewertung

Die oben vorgestellten Umsetzungsmöglichkeiten wollen wir in diesem Abschnitt gegenüberstellen und zusammenfassend bewerten. Dazu legen wir zunächst Bewertungskriterien fest, anhand deren wir die vier Möglichkeiten evaluieren können. Dabei trennen wir zwischen Kriterien bei der Entwicklung und der Nutzung der so erstellten Sprachen. Zur Betrachtung der Entwicklung gehören die Kriterien

Komplexität Die Entwicklung einer DSL für das V-Modell ist sicher keine einfache Aufgabe, allerdings ist hier auch die Komplexität der Gesamtlösung, die auch das zugrunde liegende DSL Framework miteinbezieht, zu bewerten.

Aufwand Der Aufwand der Entwicklung einer Sprache berücksichtigt auch spezifische Anpassungen wie Validierung oder erweiterte Darstellung sowie das gegebenenfalls zu entwickelnde DSL Framework.

Während bezüglich der Nutzung folgende Kriterien interessant sind:

Sprachfunktionalität Mit der Sprachfunktionalität wollen wir einen Richtwert angeben, der aussagt, ob ein Ansatz die Prozessmodellierung auch gebührend unterstützen kann. Dabei sind insbesondere Modellierungsmöglichkeiten, die mit den jeweiligen Ansätzen erreicht werden können, ausschlaggebend. So verstehen wir unter einer hohen Sprachfunktionalität vor allem auch grafische Editoren sowie auch spezifische Erweiterungen für die Modellierung von Teilmodellen eines Prozessmodells. Im Sinne der Umsetzungsmöglichkeiten muss hierbei betrachtet werden, ob diese in einer so erstellten Sprache zugelassen werden.

Änderungsaufwand Der Änderungsaufwand bewertet wie umständlich Anpassungen am Metamodell, das der DSL zugrunde liegt, auch in der Sprache abgebildet werden können. Dieser ist natürlich immer spezifisch bezüglich der tatsächlichen Änderungen, allerdings kann hierbei sehr wohl geschätzt werden, wie im Schnitt der Änderungsaufwand bei den vorgeschlagenen Ansätzen aussieht.

Wichtig ist, dass wir in der Bewertung die einzelnen Umsetzungsmöglichkeiten gegenüberstellen, so dass beispielsweise ein geringer Implementierungsaufwand bei einer Möglichkeit auch immer in Kontext zu den restlichen Möglichkeiten zu stellen ist und nicht bedeutet, dass die Sprache innerhalb weniger Tage entwickelt werden kann. Zudem beachten und bewerten wir auch nur die technische Realisierung, so dass die fachliche Abbildung eines Modells in eine Sprache hierbei nicht einbezogen wird. Für die Erstellung der zusammenfassenden Evaluierung müssen wir die letzten beiden Kriterien bezüglich der Ansätze diskutieren, die ersten beiden haben wir bereits bei der Vorstellung der Umsetzungsmöglichkeiten betrachtet.

Sprachfunktionalität. Die Sprachfunktionalität bei Ansätzen 1 und 2 wäre bei der Nutzung der DSL Tools relativ starr. Die Erweiterung eines solchen Sprache um spezifische Darstellungen (beispielsweise listenartig oder tabellarisch) ist sehr schwer bis hin zu unmöglich, so dass die Modellierungsmöglichkeiten hier eingeschränkt sind. Deshalb vergeben wir hier die Bewertung mittel.

Bei Ansätzen 3 und 4 hingegen können wir fordern, dass das zu erstellende DSL Modellierungsframework in der Lage ist, beliebige Darstellungen und somit auch Modellierungsmöglichkeiten für die Daten zu unterstützen. Somit ist die Sprachfunktionalität, die bei diesen beiden Ansätzen erreicht werden kann, mit sehr gut zu bewerten.

Änderungsaufwand. Den Änderungsaufwand für Ansätze 1 und 2 haben wir bereits bei der Vorstellung derselben als hoch beziehungsweise mittels bis hoch festgestellt. Bezüglich Ansatz 3 und 4 kann dieser allerdings als gering bis hoch gewertet werden, schließlich erstellen wir hier ein eigenes DSL Modellierungsframework, das insbesondere auf das V-Modell XT abgestimmt sein soll und demzufolge auch die Erreichung eines geringeren Änderungsaufwands als Ziel vorweisen kann. Hohe Aufwände lassen sich hierbei allerdings auch nicht ausschließen.

Die zusammengefasste Evaluierung der Umsetzungsmöglichkeiten kann der Tabelle 4.1 entnommen werden.

	Entwicklung		**Nutzung**	
	Komplexität	*Aufwand*	*Sprachfunktionalität*	*Änderungsaufwand*
Ansatz 1	gering	mittel	mittel	hoch
Ansatz 2	mittel	mittel	mittel	mittel bis hoch
Ansatz 3	sehr hoch	sehr hoch	sehr gut	gering bis hoch
Ansatz 4	sehr hoch	sehr hoch	sehr gut	gering bis hoch

Tabelle 4.1: Zusammenfassung der vier Umsetzungsmöglichkeiten

4.3 Sprachanforderungen

Bei der Implementierung einer DSL für das V-Modell muss insbesondere die Verbesserung der derzeitigen Modellierungsmöglichkeiten (V-Modell XT Editor) im Vordergrund stehen. Insofern wollen wir uns in diesem Abschnitt mit den Anforderungen an die zu entwickelnde DSL befassen und dabei Ziele für dieselbe definieren. Generell werden dabei die Bereiche der Validierung, Visualisierung und Modellierung des Modells diskutiert und mit dem V-Modell XT Editor bezüglich Verbesserungen verglichen.

4.3.1 Validierung

Bei der Validierung des V-Modells geht es um die Überprüfung des Modells auf Inkonsistenzen. Diese können beispielsweise in Form von nicht vorhandenen Elementen, verletzten Beziehungskardinalitäten oder schlicht mittels fehlender Eigenschaftszuweisungen auftreten. Das Ziel der Validierung ist es nicht, diese Fehler selbstständig zu bereinigen. es geht vielmehr um die Fehlerfindung und -aufbereitung, so dass der Benutzer in der Lage ist, diese Inkonsistenzen entsprechend zu lösen. Konkret legen wir für die Validierung folgende Anforderung fest:

Anforderung 4.1 (Validierung der Modellinstanz):
Bei der Validierung des V-Modells sind Inkonsistenzen in der Modellinstanz in Form von

- fehlenden Elementen
- fehlerhaften oder fehlenden Beziehungen
- verletzten oder nicht erfüllten Beziehungskardinalitäten
- fehlerhaften oder fehlenden Eigenschaftszuweisungen

zu finden und für den Benutzer mit Ursache und Quelle aufzubereiten.

Wir fordern bei Anforderung 4.1 allerdings nur dort eine vollständige Umsetzung, wo es tatsächlich sinnvoll ist und der Aufwand die Vorteile der Erkennung nicht übersteigt. Ein Beispiel bei den Projektdurchführungsstrategien (siehe Abschnitt 4.1.1) zeigt die Notwendigkeit dieser Einschränkung auf:

Strategien weisen zur Definition von Abläufen neben normalen Ablaufpunkten auch Splits und Joins auf, so dass parallele Abläufe definiert werden können. Problematisch bei der Validierung

sind an dieser Stelle die Kardinalitäten der Split-Ausgänge sowie der Join-Eingänge. Einerseits ist es nicht immer möglich automatisch festzustellen, ob die Kardinalitäten richtig gesetzt sind, sprich ob die Anzahl der Abläufe definiert durch einen Split-Ausgang auch der Anzahl der von einem Join-Eingang vereinigten Abläufe entspricht. Andererseits erfordert die Überprüfung, ob das zumindest größtenteils (beim aktuellen V-Modell ist das nur an einer Stelle nicht entscheidbar) passt, einen relativ hohen Implementierungsaufwand [WKK09b] (eine Idee eines möglichen Konzepts lässt sich [BF09] entnehmen). Insofern wollen wir derartige Validierungsmethodiken nicht zwingend bei der Anforderung 4.1 voraussetzen.

Zur Validierung des V-Modells zählt auch die Überprüfung der Modellstrukturen (serialisierte Form) beim Laden von Modellinstanzen. Dieses dient zum einen dem Auffinden von Fehlern, die eine Verwaltung des Modells unmöglich machen, zum anderen zeigt es grundsätzliche Probleme im serialisierten Modell auf, so dass diese behoben werden können, um Problemen nach dem Laden auf der Sprachinstanz-Ebene vorzubeugen. Konkret definieren wir:

Anforderung 4.2 (Validierung des serialisierten Modells):
Beim Laden des V-Modells sind Inkonsistenzen in der serialisierten Form des Modells zu finden. Das sind

- doppelte IDs
- unbekannte Elemente, Beziehungen, Eigenschaften
- ungültige Beziehungen, deren Teilnehmer im Modell nicht vorhanden sind
- fehlende Eigenschaften, die notwendigerweise gesetzt sein müssen (z.B.: ID)

Vergleichen wir nun die definierten Anforderungen an die Validierung mit dem Tatsachenbestand beim aktuellen V-Modell XT Editor, so lässt sich feststellen, dass beide Anforderungen nicht erfüllt werden. Genauer genommen ist es sogar so, dass grobe Fehler im serialisierten Modell beim Laden mit einer minder aussagekräftigen Fehlermeldung quittiert werden (siehe Abbildung 4.9), wobei der Ladevorgang abgebrochen wird. Die Validierung des geladenen Modells ist in der von uns geforderten Form nicht vorhanden.

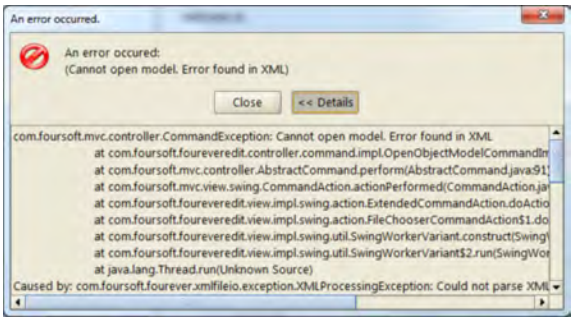


Abbildung 4.9: Fehler beim Laden eines fehlerhaften serialisierten Modells im V-Modell XT Editor

4.3.2 Visualisierung und Modellierung

Bei der Visualisierung und der Modellierung geht es in erster Linie um die Definition von Visualisierungsbereichen für eine Sprache für das V-Modell. Ferner müssen spezifische Modellierungsmöglichkeiten betrachtet werden, die zum einen zwingend notwendig sind für die Verwaltung des Modells, zum anderen zur Benutzbarkeit und Erlernbarkeit beitragen. Grundsätzlich orientieren wir uns an der derzeitigen Modellierungslösung für das V-Modell, dem V-Modell XT Editor, diskutieren allerdings Verbesserungen sowie Erweiterungen, die komplett neue Darstellungen oder Modellierungsweisen einführen.

Darstellungsbereiche. Die Diskussion von Darstellungsbereichen lässt sich zunächst auf einfache, zwingend notwendige, Bereiche reduzieren. Diese dienen der Navigation, der einfachen Editierung von Eigenschaften sowie Beziehungen und der Darstellung der Validierungsergebnisse. Die ersten beiden können dabei sehr ähnlich zum V-Modell XT Editor (siehe Abschnitt 1.2.2)

gehalten werden, für die Darstellung der Validierungsergebnisse sehen wir eine ähnliche Darstellung zu derselben im PDS-Designer (siehe Abschnitt 1.2.2) vor. Zusammenfassend halten wir fest:

Anforderung 4.3 (Einfache Darstellungsbereiche):
Die DSL für das V-Modell soll drei einfache Darstellungsbereiche aufweisen, die über entsprechende Funktionalitäten verfügen:

- Navigationsbereich: Das V-Modell lässt sich navigieren sowie editieren in der Hinsicht, dass neue Elemente angelegt und gelöscht werden können.
- Eigenschaftenmodellierungsbereich: Eigenschaften und Beziehungen des ausgewählten Elements lassen sich in diesem Bereich bearbeiten.
- Fehleraufbereitungsbereich: Fehler, die während der Validierung gefunden werden, sind in diesem Bereich mit Quelle und Ursache aufzubereiten.

Für den Fehleraufbereitungsbereich setzen wir zudem eine Filterfunktion voraus, die spezifische Fehler ausblenden kann.

Die oben erwähnte Filterfunktion ist notwendig, da bei der Validierung auch Fehler erkannt werden können, die tatsächlich keine sind. Als Beispiel können wir dabei das Pflichtfeld *Beschreibung* betrachten, das bei jedem beliebigen Element zu einem Validierungsfehler führt, falls es nicht ausgefüllt ist. Allerdings ist ein leeres Beschreibungsfeld bei Elementen, die eine Referenz auf einen *Textbaustein* erhalten, tatsächlich kein Fehler. Das ist aber im Metamodell nicht verankert, kann insofern auch nicht automatisch erkannt werden. Während an dieser Stelle noch die Forderung nach einer erweiterten Validierungsdefinition diskutiert werden könnte, die diesen Fehler entsprechend vermeidet, kann diese beispielsweise bei einer Variante des V-Modells, dem V-Modell XT Bund nicht mehr geführt werden. Dort sind Beschreibungsfelder bei speziellen Elementen, wie Split oder Join aus den Projektdurchführungsstrategien (siehe Abschnitt 4.1.1), nicht mehr Pflicht. Somit ist eine solche Filterfunktionalität auch zwingend notwendig und muss modellspezifisch gehalten werden.

Modellierungsmöglichkeiten. Beim Erstellen von Beziehungen über das Eigenschaftsfenster des V-Modell XT Editors tritt mitunter das Problem auf, dass die Auswahl der Beziehungspartner nicht spezifisch genug gehalten werden kann. Als Beispiel betrachten wir das Anlegen von Übergängen bei den Projektdurchführungsstrategien. Übergänge können nur zwischen Elementen desselben Ablaufbausteins angelegt werden, so dass diese modular gehalten werden. Beim V-Modell XT Editor (siehe Abbildung 4.10) ist diese Einschränkung so nicht umgesetzt, da sie ja auch nicht im Metamodell verankert ist. Natürlich könnten wir diese Problematik über die Validierung abfangen, genauer genommen tun wir das auch. Allerdings wollen wir die Modellierung fehlerhafter oder ungültiger Elemente sowie Beziehungen soweit wie möglich vermeiden, so dass wir an dieser Stelle die folgende Anforderung an die DSL für das V-Modell formulieren:

Anforderung 4.4 (Einschränkbarer Eigenschaftenmodellierungsbereich):
Die DSL für das V-Modell soll beim Eigenschaftenmodellierungsbereich Einschränkungen bezüglich der teilnehmenden Beziehungspartner erlauben, so dass mögliche Beziehungspartner bezüglich jeder Editierungsinstanz der Beziehung spezifiziert werden können.

Weiterhin legen wir für den Eigenschaftenmodellierungsbereich fest, dass dieser auch spezifisch erweiterbar sein muss. Das ist notwendig, da das V-Modell intern für die Dokumentation eine Untermenge von HTML [W3Cc] benutzt, welches entsprechend im Editor auch zu bearbeitet ist. Somit sehen wir in der DSL für das V-Modell eine solche Erweiterungsmöglichkeit vor. Wir halten fest:

Anforderung 4.5 (Erweiterbarer Eigenschaftenmodellierungsbereich):
Die DSL für das V-Modell soll beim Eigenschaftenmodellierungsbereich erweiterbar sein, so dass spezifische Modellierungsweisen, wie das Erstellen von HTML mithilfe eines Editors, implementiert werden können.

Erweiterte Darstellungsbereiche und Modellierungsmöglichkeiten. Soweit haben wir uns nur mit einfachen beziehungsweise bereits aktuell zumindest größtenteils vorhanden Darstellungs- und Modellierungsmöglichkeiten befasst. Folgend wollen wir einige erweiterte Möglichkeiten festlegen, die insbesondere der Benutzbarkeit, Erlernbarkeit sowie der Nachvollziehbarkeit dienen sollen.

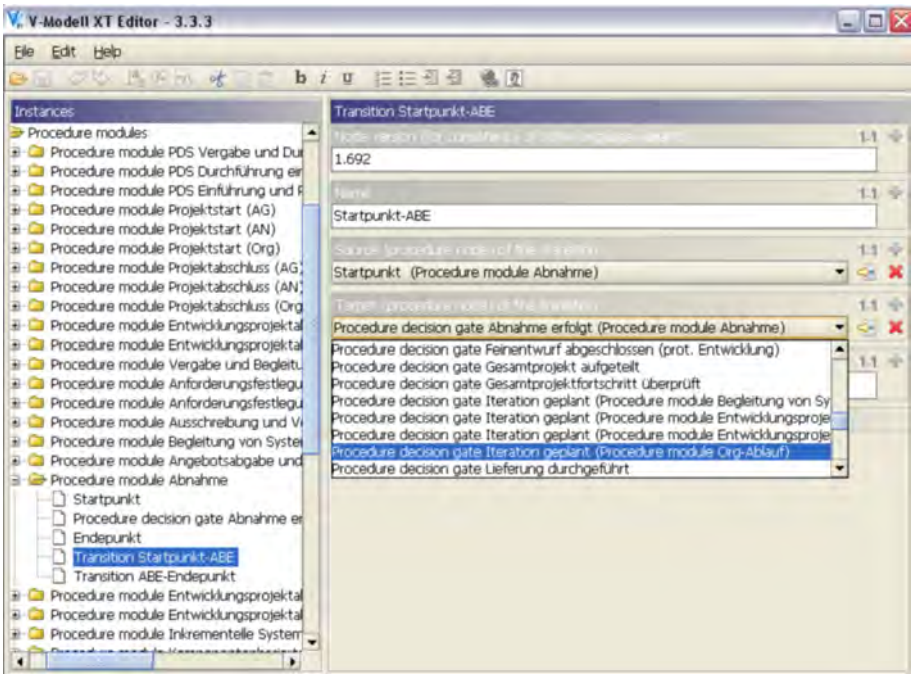


Abbildung 4.10: Modellierung von Übergängen (Quelle und Ziel) im V-Modell XT Editor

Abhängigkeiten Das V-Modell XT verfügt als Prozessmodell über viele Beziehungen, die zwischen den einzelnen Elementen festgelegt werden können. Folglich müssen diese auch bei Bedarf entsprechend aufgelistet werden, so dass alle Beziehungen, an denen ein spezifisches Element teilnimmt, übersichtlich bereitgestellt werden. Im V-Modell XT Editor ist das über eine listenartige Darstellung (siehe Abbildung 4.11) realisiert, die allerdings Mängel bezüglich Informationsgehalt und Übersichtlichkeit aufweist. Dort werden nämlich die Pfade von Elementen dargestellt, die in Beziehung zum ausgewählten Element stehen. Problematisch ist die Länge dieser Pfade, die öfters abgekürzt wird, allerdings lassen sich die Elemente zumindest noch ansteuern. Für die DSL für das V-Modell wollen wir diese Abhängigkeitsdarstellung verbessern, indem wir die folgende Anforderung definieren:

Anforderung 4.6 (Abhängigkeitsdarstellung):
Die Darstellung von Abhängigkeiten [Wac08] in der DSL für das V-Modell soll listenartig mit Quelle, Ziel, Beziehungstyp sowie der Kategorie (referenziert oder wird referenziert) erfolgen. Zudem soll es möglich sein zur Quelle beziehungsweise zum Ziel zu navigieren.

Löschen von Elementen Beim Löschen von Elementen werden auch alle Beziehungen, an denen diese Elemente entsprechend teilnehmen, mitgelöscht. Interessant beziehungsweise wichtig bei so einem Löschvorgang ist genau die Liste der Beziehungen, die automatisch mitgelöscht werden. Diese ist nämlich selbst für einen sehr erfahrenen Benutzer nur bedingt direkt ableitbar, schließlich müssten dem Benutzer alle möglichen Beziehungstypen sowie die zugehörigen Beziehungsinstanzen bekannt sein, so dass hier eine visuelle Unterstützung erforderlich wird. Beim V-Modell XT Editor ist diese nicht gegeben, hier können Elemente gelöscht werden, ohne dass der Benutzer überhaupt erst nachgefragt wird, ob er denn tatsächlich die Elemente mitsamt allen zugehörigen Beziehungen aus dem Modell entfernen möchte. Es ist zudem gar nicht erst möglich aufzuzeigen, genau welche Elemente (als Kinder des zu löschenden Elements) und Beziehungen bei einem Löschvorgang betroffen sind. Das wollen wir bei der DSL für das V-Modell verbessern und definieren dazu die folgende Anforderung:

Anforderung 4.7 (Löschung von Modellelementen):
Das Löschen von Elementen in der DSL für das V-Modell soll vor dem tatsächlichen Entfernen von Elementen über die automatisch mitgelöschten Beziehungen informieren und somit dem Benutzer die Wahl offen lassen, ob er den initiierten Löschvorgang durchführen möchte. Die Darstellung der Beziehungen soll dabei wie bei Anforderung 4.6 erfolgen.

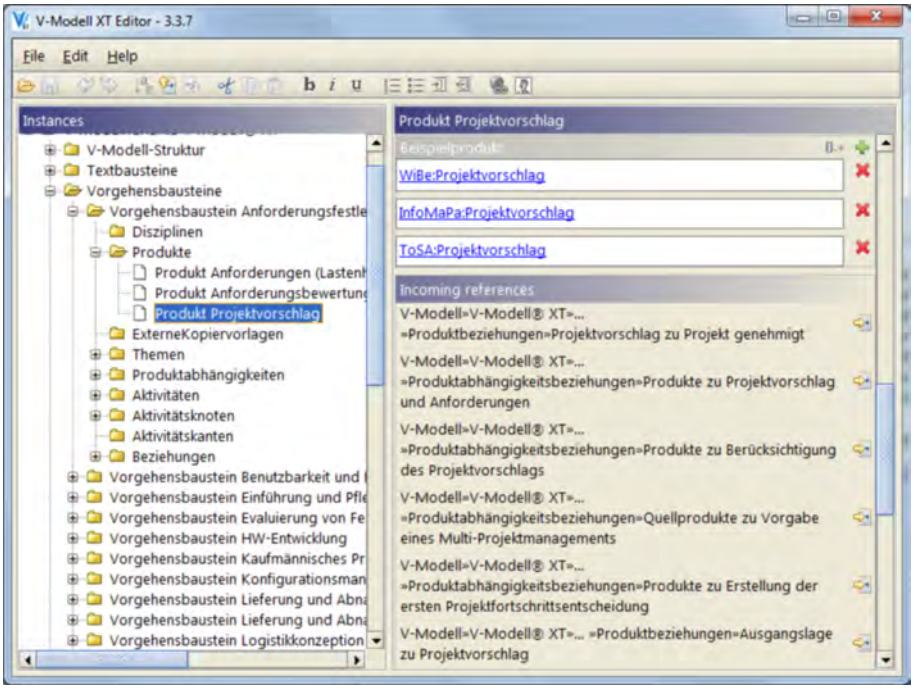


Abbildung 4.11: Abhängigkeitsdarstellung im V-Modell XT Editor

Grafische Darstellungen. Grafische Darstellungen sind ein wesentliches Ziel unserer DSL für das V-Modell. Derzeit bietet der V-Modell XT Editor noch gar keine grafischen Modellierungsmöglichkeiten, so dass gerade an dieser Stelle der größte Bedarf besteht. Wie eine grafische Lösung zumindest für die Definition von Projektdurchführungsstrategien (siehe Abschnitt 4.1.1) aussehen kann haben wir beim PDS-Designer (siehe Abschnitt 1.2.2) gesehen. Eine ähnliche Modellierungslösung wollen wir auch für die Strategien in unserer DSL haben. Grundsätzlich ist aber unser Ziel bei der DSL eine grafische Darstellung auch für die Vorgehensbausteine [Wac08] oder für die Konfiguration von Strategien sowie für weitere Bereiche des V-Modells bereitzustellen. Im Rahmen dieser Arbeit beschränken wir uns bei der konkreten Anforderung nur auf die Definition von Strategien, weitere Darstellungen sind an dieser Stelle erwünscht, aber nicht explizit gefordert.

Wichtig ist, dass unsere DSL für das V-Modell die Möglichkeit bietet beliebige Darstellungen, sei es diagramartig oder listenartig, zu definieren und als Darstellungsbereiche in der Sprache zu integrieren. Auf diese Art und Weise können erweiterte Visualisierungen der Daten definiert werden, die dem Benutzer die Erlernbarkeit beziehungsweise die Nachvollziehbarkeit der Prozesse im Modell erleichtern.

Anforderung 4.8 (Grafische Modellierung von Projektdurchführungsstrategien):
Die DSL für das V-Modell soll über grafische Darstellungen für die Definition von Projektdurchführungsstrategien (siehe Abschnitt 4.1.1) verfügen sowie beliebige Darstellungen (als Sichten[1] auf die Daten, sprich auf das Modell) als Darstellungsbereiche integrieren können.

Pluginsystem. An der Technischen Universität München werden für das V-Modell zahlreiche Werkzeuge entwickelt [VMXa]. Ziel der Sprache für das V-Modell ist es auch eine Integrationsplattform für diese Werkzeuge bereitzustellen. Um das zu erreichen, bedarf die Sprache eines Plugin-Konzepts. Dieses soll allerdings nicht nur der Integration vorhandener Werkzeuge dienen, es soll zudem die Erweiterung der Sprache mittels einfacher Plugins ermöglichen. Das Hauptziel des Pluginsystems ist die Integration der Process Enactment Tool Framework (PET) Plattform [KKT10] über ein einfaches Plugin, so dass dessen Funktionalität aus der DSL genutzt werden kann.

1 Den Begriff einer Sicht greifen wir später beim Design der DSL für das V-Modell XT wieder auf.

Anforderung 4.9 (Erweiterbarkeit):
Für die DSL für das V-Modell soll ein Pluginsystem entworfen werden, das die Integration vorhandener Werkzeuge sowie die Erweiterung der Sprache ermöglicht.

Sonstiges. Für die Erleichterung der Modellierung des V-Modells sehen wir noch Standardfeatures wie „Undo, Redo“ sowie „Cut, Copy, Paste“ vor. Zudem wollen wir die Navigation durch das Modell ermöglichen, die zum zuvor selektierten Element navigieren lässt. Zusammengefasst stellen wir die folgende Anforderung auf:

Anforderung 4.10 (Benutzbarkeit):
Die Sprache für das V-Modell soll über die folgenden Features verfügen, die die Arbeit mit dem Modell erleichtern sollen:

- Cut, Copy und Paste
- Undo und Redo
- Navigation Forward und Back

4.3.3 Zusammenfassung

In Tabelle 4.2 sind die diskutierten Anforderungen bezüglich ihrer konkreten Zielsetzung zusammengefasst. Erwähnenswert bleibt, dass die gestellten Anforderungen keineswegs vollständig sind und in dieser Form nur eine erste Zielsetzung für die DSL für das V-Modell definieren.

Nr.	Name	Ziel
4.1	Validierung der Modellinstanz	Validierung des V-Modells auf Inkonsistenzen
4.2	Validierung des serialisierten Modells	Validierung des serialisierten V-Modells auf Inkonsistenzen beim Laden
4.3	Einfache Darstellungsbereiche	3 einfache Darstellungsbereiche zur Navigation, Eigenschaftenbearbeitung und Fehlerauflistung
4.4	Einschränkbarer Eigenschaftenmodellierungsbereich	Eigenschaftenbearbeitung soll möglichst flexibel konfigurierbar sein
4.5	Erweiterbarer Eigenschaftenmodellierungsbereich	Eigenschaftenbearbeitung soll spezifische Erweiterungen erlauben
4.6	Abhängigkeitsdarstellung	Listenartige Abhängigkeitsdarstellung für Modellelemente
4.7	Löschung von Modellelementen	Löschvorgang informiert über automatisch mitgelöschten Beziehungen
4.8	Grafische Modellierung von PDSen	Grafische Darstellung und Modellierungsweise für die Projektdurchführungsstrategien; Beliebige grafische Darstellungen.
4.9	Erweiterbarkeit	Pluginsystem
4.10	Benutzbarkeit	Standardfeatures: Cut, Copy und Paste, Undo und Redo sowie Navigation Forward und Back

Tabelle 4.2: Anforderungen an die DSL für das V-Modell mit ihren Zielsetzungen

5 Design: Domänenspezifische Sprache für das V-Modell XT

In diesem Kapitel beschäftigen wir uns mit dem Design einer domänenspezifischen Sprache für das V-Modell XT. Hierzu betrachten wir zunächst das *Process Development Environment* (PDE) und stellen dessen Konzepte anhand eines einfachen Beispiels vor, um folgend die Implementierung des V-Modell XT Metamodells im PDE mitsamt den zusätzlich notwendigen spezifischen Anpassungen zu beschreiben. Als Ergebnis dieses Kapitels wird ein Editor für das V-Modell XT basierend auf PDE präsentiert.

Übersicht

5.1 Process Development Environment

Das Process Development Environment (PDE) wurde im Rahmen dieser Arbeit an der Technischen Universität München entwickelt. PDE verfolgt das Ziel, Prozessmodelle mittels domänenspezifischer Sprachen zu implementieren, und wird insbesondere zur Unterstützung des an der TU München mitentwickelten V-Modell XT Vorgehensmodells konzipiert. Der Einsatz von PDE lässt sich allerdings nicht nur auf das V-Modell oder Prozessmodelle im Speziellen reduzieren, es kann vielmehr für beliebige Modelle, die als domänenspezifische Sprachen implementiert werden können, eingesetzt werden. In diesem Sinne lässt sich PDE mit dem Eclipse Modeling Framework (siehe Abschnitt 2.3) oder den Microsoft DSL-Tools (siehe Abschnitt 2.4) vergleichen. Genauer genommen orientiert sich PDE weitestgehend an den DSL-Tools und verwendet diese bei der eigentlichen Implementierung mit. PDE weist allerdings einige Erweiterungen beziehungsweise Neuheiten auf, die bei den DSL-Tools nicht vorhanden sind. Dazu zählt unter anderem die größere Flexibilität bei der Anpassung, eine WPF-basierende grafische Darstellung, die beliebige Sichten auf dieselben Daten erlaubt sowie die Unabhängigkeit vom Microsoft Visual Studio beim Zieleditor der Sprache (dieser ist nämlich eine selbstständige Applikation). Somit ist PDE ein nach dem Prinzip der domänenspezifischen Entwicklung (siehe Abschnitt 2.2) funktionierendes Framework, das zur Erstellung grafischer DSLs konzipiert wurde, die jedoch ihrerseits nicht als Plugins sondern als eigenständige Applikationen erstellt werden.

Hinweis

In Kapitel 4 haben wir mögliche Konzeptionsweisen (siehe Abschnitt 4.2.3 und 4.2.4) für eine domänenspezifische Sprache für das V-Modell XT vorgestellt, die jeweils die Erstellung eines Basisframeworks zur Definition von DSLs voraussetzen. PDE basiert hierbei auf dem ersteren Ansatz, ein komponentenweises Konzept würde den Rahmen dieser Arbeit sprengen zumal die tatsächlichen Vorteile hieraus für das V-Modell eher gering beziehungsweise nur bedingt ausschlaggebend sind. Allerdings ist die Erweiterung von PDE zur Unterstützung von komponentenweiser DSL-Entwicklung durchaus zu einem späteren Zeitpunkt durchführbar und wird hier somit keinesfalls ausgeschlossen.

PDE besteht selbst aus zwei domänenspezifischen Sprachen (siehe auch Abbildung 5.1), die weiter unten eingehend beschrieben werden:

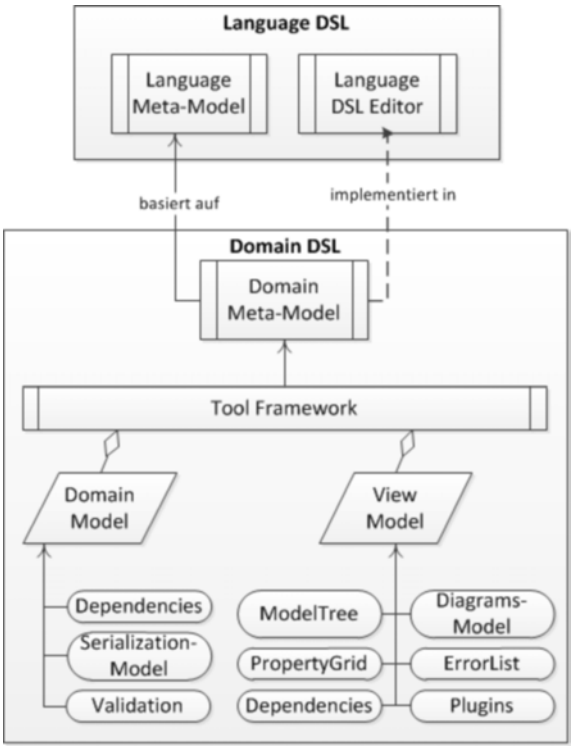


Abbildung 5.1: Überblick über den Aufbau von PDE

Language DSL ist die Basis-Sprache zur Implementierung von DSLs. Als solche weist sie entsprechende Strukturen auf, so dass Prozessmetamodelle implementiert werden können. Zudem stellt sie weitere Grundstrukturen bereit, die unter anderem für die Validierung, Serialisierung oder Darstellung der zu erstellenden DSL notwendig sind. Die Language DSL besitzt ferner einen grafischen Editor, der beim Konzipieren von Sprachen behilflich ist und der die Implementierung des Domänenmetamodells erlaubt, welches über das Metamodell der Sprache sowie dessen eigentliche Konfiguration auch bezüglich der Serialisierung sowie der grafischen Darstellung verfügt. Die Language DSL stellt weiterhin eine Validierung in ihrem Editor bereit. Somit weist sie eine Unterstützung bei der Modellierung einer Sprache auf, die zumindest bezüglich erlaubter Strukturen, Namensgebungen sowie fehlender oder fehlerhafter Information dem Entwickler Aufschluss gibt.

Domain DSL ist die Ziel-Sprache, die für ein Prozessmetamodell oder ein anderes Metamodell implementiert wird. Sie basiert auf der Language DSL und wird im Editor der Language DSL entworfen. Die Domain DSL selbst weist ein Domänenmodell auf, das Instanzen des implementierten Metamodells serialisieren und verwalten kann. Ferner verfügt die Domain DSL über Darstellungen, die mithilfe von View Modellen festgelegt und benutzt werden. Das Domänenmetamodell der Domain DSL dient der automatischen Generierung von Quellcode. So liegt nach dieser Generierung unter anderem das gesamte Metamodell der Sprache in Quellcode vor, so dass auch Instanzen dieses Metamodells in Quellcode instanziiert werden können.

Hinweis

Im Rahmen der Language und Domain DSL verwenden wir oft Begrifflichkeiten, um zugehörende Modelle auszuweisen. Wir nennen sie wie folgt:

Domänenmetamodell bezeichnet das mittels des Language DSL Designers implementierte Metamodell der Sprache, das für einen bestimmten Sachverhalten definiert und konfiguriert wurde.

Domänenmodell weist die Instanz des Domänenmetamodells aus und wird in Quellcode generiert. Das Domänenmodell beinhaltet alle definierten Typen (Domänenklassen sowie -beziehungen) des Domänenmetamodells als Klassen in Quellcode.

Im Folgenden wollen wir sowohl die Language DSL als auch die Domain DSL mit ihren jeweiligen Strukturen und Teilmodellen genauer betrachten, um schließlich die Vorgehensweise bei der Implementierung von DSLs mittels PDE zu beschreiben.

5.1.1 Language DSL

Die Language DSL stellt, wie bereits erwähnt, die Basis-Sprache zur Erstellung von DSLs dar und wurde selbst mit den Microsoft DSL-Tools (siehe auch Abschnitt 2.4) implementiert. Somit kann sie auch als eine Metasprache zur Erstellung von Sprachen gesehen werden. Insofern, verfügt die Language DSL über die notwendigen Elemente, die die strukturelle Abbildung eines Modells (beispielsweise Prozessmetamodell) in dieselben erlauben. Damit weist die Language DSL Strukturen zur Implementierung des Domänenmetamodells der Ziel-DSL auf. Wir gehen folgend auf die wesentlichen Elemente ein, insbesondere werden die Elemente zum Erstellen von Metamodellen der Zielsprache eingehend beschrieben. Verzichten wollen wir hier hingegen auf Modellierungsstrukturen, die der internen Konzeption des Editors der Language DSL dienen, da sie für das Verständnis von PDE eher unwesentlich sind.

Definition des (Meta-)Modells. Zur Definition des Domänenmetamodells der Domain DSL im Language DSL Designer stellt die Language DSL die folgenden Strukturen bereit (diese orientieren sich an den Microsoft DSL Tools, siehe Abschnitt 2.4):

Domain Type beschreibt einen Typ im Domänenmetamodell der Domain DSL. Ein Typ kann einerseits externer Art (*ExternalType*) sein, so dass er einen existierenden Typ wie beispielsweise *System.String* spezifiziert. Externe Typen können allerdings auch zur Einbindung von neuen speziellen Datentypen benutzt werden, die für die Ziel-DSL notwendig sind. Somit können neue Datentypen, wie beispielsweise ein *HTML*-Datentyp erstellt und in die Ziel-DSL eingebunden werden. Andererseits kann es sich bei einem Typ auch um eine Enumeration (*DomainEnumeration*) handeln, die mittels der automatischen Quellcodegenerierung

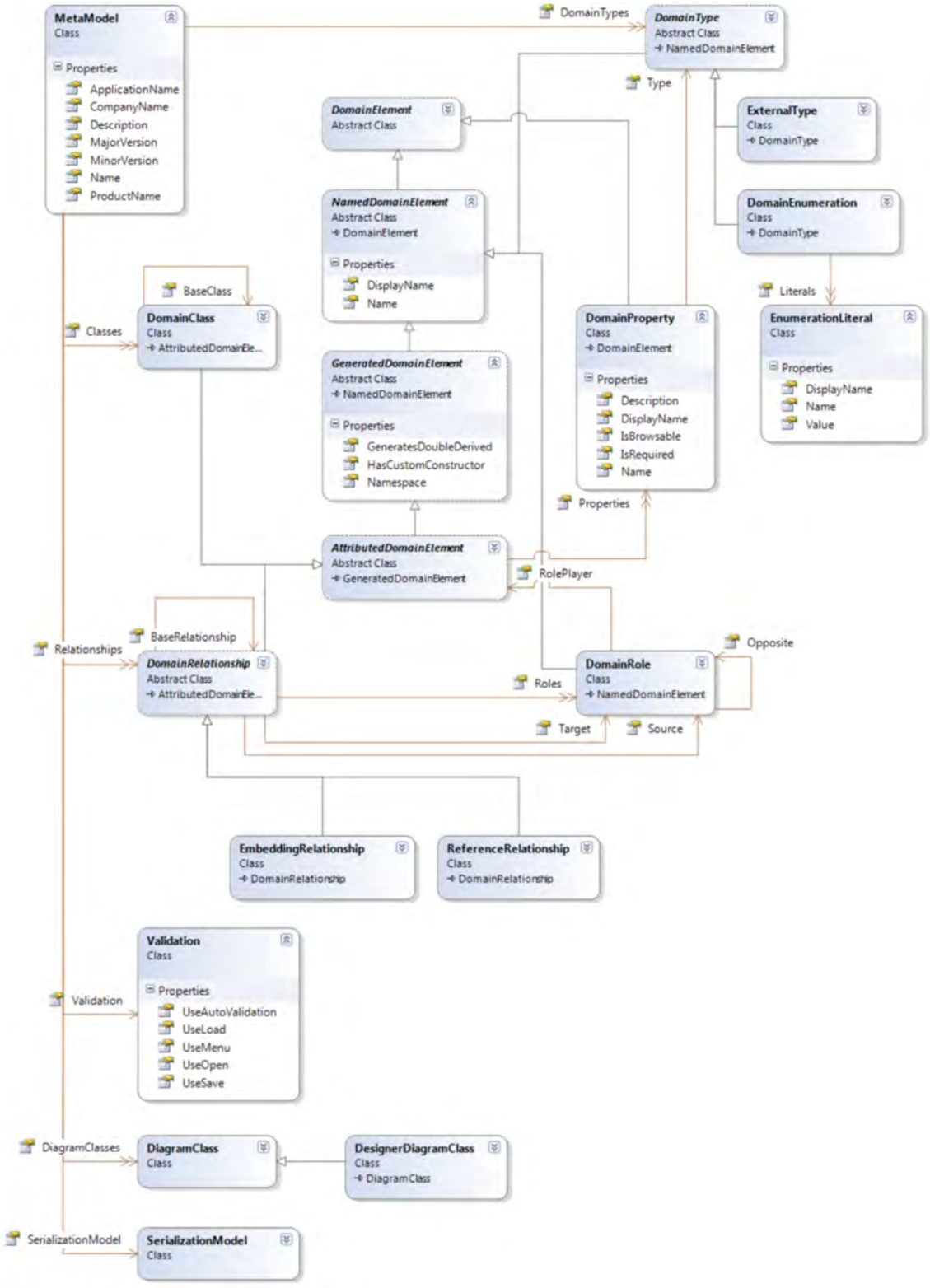


Abbildung 5.2: Language DSL: Strukturen zur Definition des Domänenmetamodell der Domain DSL

basierend auf dem Domänenmetamodell mitgeneriert wird. Jede solche Enumeration verfügt hierbei über eine Liste möglicher Elemente, die über einen Namen und einen Wert gekennzeichnet werden. Hierzu kommt, dass ein solcher Element zudem noch über einen Anzeigenamen verfügt, der die genaue Bedeutung des Elements kennzeichnet.

Jeder Domänentyp weist ferner einen Namen (*Name*), eine Beschreibung (*Description*) sowie einen Anzeigenamen (*DisplayName*) auf. Dieser Anzeigename steht dabei für einen Namen, der dem Benutzer in der Ziel-DSL für die Identifikation des Typs angezeigt werden soll. So ist es beispielsweise sinnvoll für die Enumeration *GeneratedAP* eine Bezeichnung wie *Generated after Product* anzuzeigen, damit der Anwender die Bedeutung des Datentyps besser nachvollziehen kann.

Domain Property legt eine Eigenschaft fest, die bei Elementen (Klassen und Beziehungen) definiert werden kann. Jede Eigenschaft ist typisiert über einen Verweis auf einen Typ (*DomainType*) und beinhaltet selbst unter anderem die Eigenschaften *IsRequired* und *IsBrowsable*. Diese sagen aus, ob ein Eigenschaftswert verpflichtend anzugeben ist und ob dieser über UI-Mechanismen verändert werden darf. Weiterhin werden bei einer Domäneneigenschaft der Name (*Name*) und der Anzeigenamen (*DisplayName*) vorausgesetzt sowie eine Beschreibung (*Description*) zur näheren Erklärung der Eigenschaft ermöglicht.

Eine Domäneneigenschaft weist eine Eigenschaft aus, die zu einem bestimmten Modell gehört und somit nur im Kontext dieses Modells verändert werden darf. Diese Thematik, die sich auf jegliche Veränderungen am Modell bezieht, werden wir später wieder aufgreifen und detailliert betrachten.

Domain Class definiert ein Element im Domänenmetamodell der Domain DSL. Eine Domänenklasse ist dabei als ein abstraktes Element zu verstehen, das einen Elementtyp ausweist. So kann beispielsweise ein *Produkt* als Domänenklasse implementiert werden, so dass der Ziel-DSL dann eine Klasse beziehungsweise ein Typ *Produkt* zur Verfügung steht. Das heißt, Domänenklassen definieren keine konkreten Elementinstanzen, sondern bestimmen Elementtypen für die Zielsprache.

Jede Domänenklasse verfügt weiterhin über eine Reihe von Eigenschaften. So können neben den verpflichtend zu definierenden Namen (*Name*) und Anzeigenamen (*DisplayName*) noch die Eigenschaften *GeneratesDoubleDerived* und *HasCustomConstructor* angepasst werden. Erstere sagt dabei aus, dass eine Klasse doppelt abgeleitet generiert wird (jeweils partiell), so dass virtuelle Methoden und auch Eigenschaften überschrieben werden können. Zweitere erlaubt die Definition eines Konstruktors über eigenen Quellcode (d.h.: im automatisch generierten Quellcode fehlt der entsprechende Konstruktor und muss in der partiellen Klasse vom Benutzer vorgelegt werden). Weiterhin können Domänenklassen über mehrere Domäneneigenschaften (*DomainProperty*) sowie über eine Vererbungsbeziehung verfügen.

Im Domänenmetamodell der Domain DSL wird neben einer beliebigen Anzahl von Klassen auch eine spezifische Domänenklasse hinterlegt und über die Eigenschaft *IsDomainModel=true* entsprechend gekennzeichnet. Diese Eigenschaftszuweisung sagt aus, dass diese Domänenklasse das oberste Element im Hierarchiebaum darstellt. Somit kann und muss es pro Modell auch genau eine solche Klasse geben, was allerdings die Sprache selbst sicherstellt.

Hinweis

Eine *partielle Klasse* [Par07] bezeichnet eine Klasse, die über mehrere Quellcodedateien aufgeteilt ist oder an mehreren verschiedenen Orten einer Datei deklariert ist. Partielle Klassen werden erst bei der Kompilierung aus den vorliegenden Teildefinitionen kombiniert. Auf diese Art und Weise lassen sich Klassen in der Domain DSL erweitern beziehungsweise ihr Verhalten anpassen.

Domain Relationship dient der Festlegung von Beziehungen zwischen Elementen. Somit legt eine Domänenbeziehung eine Assoziation zwischen zwei Beziehungsteilnehmern fest, die über die so genannten Domänenrollen (*DomainRole*) definiert werden. Diese legen neben den Eigenschaftsnamen der Beziehungspartner auch noch ihre Kardinalitäten fest. Somit verfügen Domänenbeziehungen über genau zwei solche Domänenrollen, jeweils eins für die Quelle und das Ziel der Assoziation. Weiterhin unterscheidet sich die Art der Beziehung zwischen

- Reference Relationship legt eine Referenzbeziehung zwischen zwei Elementen fest, und
- Embedding Relationship legt eine Integrationsbeziehung zwischen zwei Elementen fest.

Hierbei werden als Ausgangsinstanz jeweils Domänenklassen vorausgesetzt, während bei der Zielinstanz zumindest bei der Referenzbeziehung auch wieder eine Referenzbeziehung möglich ist. Zu beachten ist hierbei wieder, dass Referenz- oder Integrationsbeziehungen keine konkreten Elementinstanzen festlegen, sondern Beziehungstypen für die Domain DSL definieren.

Ähnlich zu einer Domänenklasse, weisen auch Domänenbeziehungen eine Reihe von vordefinierten Eigenschaften auf, die der Anpassung der automatischen Generierung dienen. So sind neben dem Namen (*Name*) und dem Anzeigenamen (*DisplayName*) auch wieder die Eigenschaften *GeneratesDoubleDerived* und *HasCustomConstructor*, die hier bei Beziehungen die gleiche Bedeutung wie bei den Domänenklassen aufweisen, vorhanden. Weiterhin können Domänenbeziehungen über eine Reihe von Eigenschaften (*DomainProperty*) sowie über eine Vererbungsbeziehung verfügen.

Validation legt fest, zu welchen Zeitpunkten beziehungsweise auf welche Art und mittels welcher Mittel eine Validierung des Gesamtmodells durchgeführt werden soll. Dabei wird unterschieden zwischen

- UseLoad: Die Validierung des Gesamtmodells wird nach dem Laden des Modells, noch vor dem Laden der Darstellungsinformationen gestartet. Diesbezüglich werden Fehler im Modell auf jeden Fall dem Benutzer aufgelistet, auch wenn der Ladevorgang bezüglich der Deserialisierung der Visualisierungsdaten fehlschlägt.
- UseOpen: Die Validierung des Gesamtmodells wird nach dem Laden aller Teilmodelle gestartet. Sind sowohl *UseLoad* als auch *UseOpen* ausgewählt, so wird bei einem erfolgreichen Ladevorgang die zum Validierungszeitpunkt von *UseLoad* gesammelte Information durch eine erneute Validierungsausführung zum Zeitpunkt von *UseOpen* überschrieben. In der Regel sollte somit das Auswählen beider Validierungsarten vermieden werden.
- UseMenu: Die Validierung wird über Menueinträge gestartet. Dabei kann sowohl das Gesamtmodell als auch ein selektiertes Element validiert werden. Bei einer Nichtauswahl von *UseMenu* sind die entsprechenden Menueinträge nicht vorhanden.
- UseSave: Die Validierung wird vor dem Speichern des Modells durchgeführt. Werden dabei Fehler festgestellt, so entscheidet der Benutzer darüber, ob das fehlerbehaftete Modell tatsächlich gespeichert werden soll. Diese Einstellung dient vor allem als Vorbeugungsmaßnahme, so dass vorhandene Modellfehler nicht übersehen werden können.

Zudem lässt sich über *UseAutoValidation* definieren, ob Klassen für eine automatische Validierung, die zumindest die Einhaltung der festgelegten Beziehungskardinalitäten sowie die Definition von benötigten Eigenschaftswerten überprüfen können, generiert werden sollen.

Diagram Class dient der Definition der Darstellung, genauer genommen einer bestimmten Sicht auf die Modelldaten. Standardmäßig wird dabei eine diagrammartige Visualisierung der Elemente unterstützt, eine *DiagramClass* ist aber bei weitem nicht nur darauf beschränkt. Weiter unten werden wir vor allem die diagrammartige Darstellung näher diskutieren.

Hinweis

Der Begriff der *Sicht auf ein Modell* dient uns der genauen Unterscheidung einer generellen Visualisierung von Daten (beispielsweise über eine Liste) gegenüber der hier angeboteten freien Darstellungswahl, die auch dieselben Daten auf unterscheidliche Weisen darstellen kann. Der Begriff der Sicht ist in der Informatik vielfältig und entstammt in unserem Fall dem verwendeten Konzept bei der Implementierung in der Domain DSL. Dort werden nämlich so genannte *View Models* in Verbindung mit dem *Model View ViewModel* Design Pattern (MVVM) [Smi09] verwendet, dazu später mehr.

Serialization Model beschreibt Elemente und Beziehungen, die notwendig sind um eine automatische Serialisierung für die Domain DSL erzeugen zu können. Diese werden weiter unten eingehend beschrieben.

Definition des Diagram-Modells. Das Diagram-Modell (siehe Abbildung 5.3) dient der Definition von grafischen Darstellungen für Elemente und Beziehungen des Domänenmetamodells. Jede dieser Darstellungen kann entweder mittels des Language DSL Editors definiert und generiert werden oder manuell vom Benutzer in Quellcode implementiert werden. Diese Darstellungen sind als Darstellungsbereiche komplett unabhängig voneinander, so dass auch gleiche Daten unterschiedlich visualisiert werden können. Wir sprechen dabei von Sichten auf das Modell.

Das Diagram-Modell weist zur Definition von Sichten die folgenden Strukturen auf:

Diagram Class definiert eine Sicht auf das Modell. Standardmäßig sind solche Sichten diagramartig, können aber auch benutzerspezifisch erstellt werden. Jedes Domänenmetamodell der Domain DSL verfügt dabei über eine vordefiniert Instanz dieser Klasse, die sich *DiagramDesignerClass* nennt. Diese ist als die Hauptsicht auf das Modell zu verstehen.

Eine Diagram-Klasse verfügt als Sichtdefinition über vordefinierte Eigenschaften:

- *Name*: Eindeutiger Name einer Sicht.
- *Title*: Titel einer Sicht.
- *IsCustom*: Weist eine Sicht als benutzerspezifisch aus, so dass beliebige Darstellungen definiert werden können. Das bedeutet, dass die automatisch generierte diagramartige Darstellung nicht mehr verwendet wird.

Folgend kann eine Diagram-Klasse über sogenannte Shapes verfügen, die die Darstellungen für Elemente und Beziehungen repräsentieren.

Shape Class repräsentiert die grafische Darstellungsinformation für eine *DomainClass* des Domänenmetamodell. Ein Shape Class kann als Kind der Diagram Class festgelegt werden, es kann aber weiterhin ein Shape Class genauso als Kind einer anderen Shape Class definiert werden, so dass zwischen Shape Classes eine Hierarchie spezifiziert werden kann.

Relationship Shape Class erlaubt es eine Referenzbeziehung (*Reference Relationship*) über eine entsprechende pfeilartige Darstellung zwischen den teilnehmenden Partnern zu visualisieren. Dazu müssen allerdings auch die teilnehmenden Partner selbst über eine Darstellungsdefinition mittels Shape-Klassen verfügen, denn die Beziehung selbst wird über eine Verbindungslinie zwischen den Beziehungsteilnehmern visualisiert. Somit ist eine solche Darstellung nur für Referenzbeziehungen wählbar, die zwischen Domänenklassen existieren (Referenzbeziehungen können nämlich als Ziel auch wieder Referenzbeziehungen aufweisen).

Mapping Relationship Shape Class definiert die Darstellung einer *DomainClass* über eine pfeilartige Beziehungsvisualisierung. Dazu müssen folgende Voraussetzungen erfüllt werden:

- Die Domänenklasse verfügt über zwei Referenzbeziehungen, die sie als das Quellelement aufweisen. Sie werden jeweils als *Source* beziehungsweise *Target* bei der *Mapping Relationship Shape Class* definiert.
- Die Zielelemente dieser Referenzbeziehungen sind wieder Domänenklassen, die über eine Darstellungsdefinition mittels Shape-Klassen verfügen.

Damit sind die Zielelemente der beiden Referenzbeziehungen dann die tatsächlichen Beziehungspartner, zwischen denen eine pfeilartige Visualisierung definiert wird.

Die Strukturen des Diagram-Modells in der Language DSL dienen der Definition der Darstellung, legen allerding keine konkrete grafische Visualisierungsinformation fest. Sie spezifizieren hingegen nur, dass ein Element oder eine Beziehung über ein spezifisches Shape dargestellt wird, so dass dieses später als Basis für die Darstellung und das Layouting genutzt werden kann. Hintergrund einer solchen Definition ist die Nutzung eines weiteren Frameworks (Diagrams-Framework), so dass diese Definition in die dieses Framework generiert werden kann. So erhalten wir auch Undo, Redo sowie Cut, Copy und Paste bei grafischen Darstellungen und können das Diagrams-Framework fürs Layouting nutzen. Das Diagrams-Framework ist wieder Teil von PDE, genauer genommen Teil der Domain DSL. Deshalb gehen wir auf dieses Gesamtkonzept noch konkreter bei der Domain DSL ein.

Definition des Serialisierungsmodells. Das Serialisierungsmodell (siehe Abbildung 5.4) stellt Strukturen zur Definition eines auf XML basierenden Serialisierungsformats bereit. Das Ziel hierbei ist es, eine möglichst flexible Serialisierungslösung bereitzustellen, die bereits auf der Designer-Ebene eingehend beeinflusst werden kann. Wichtig ist, dass das Serialisierungsmodell selbst eine hierarchische Struktur definiert, die basierend auf dem Hierarchiebaum, der durch die

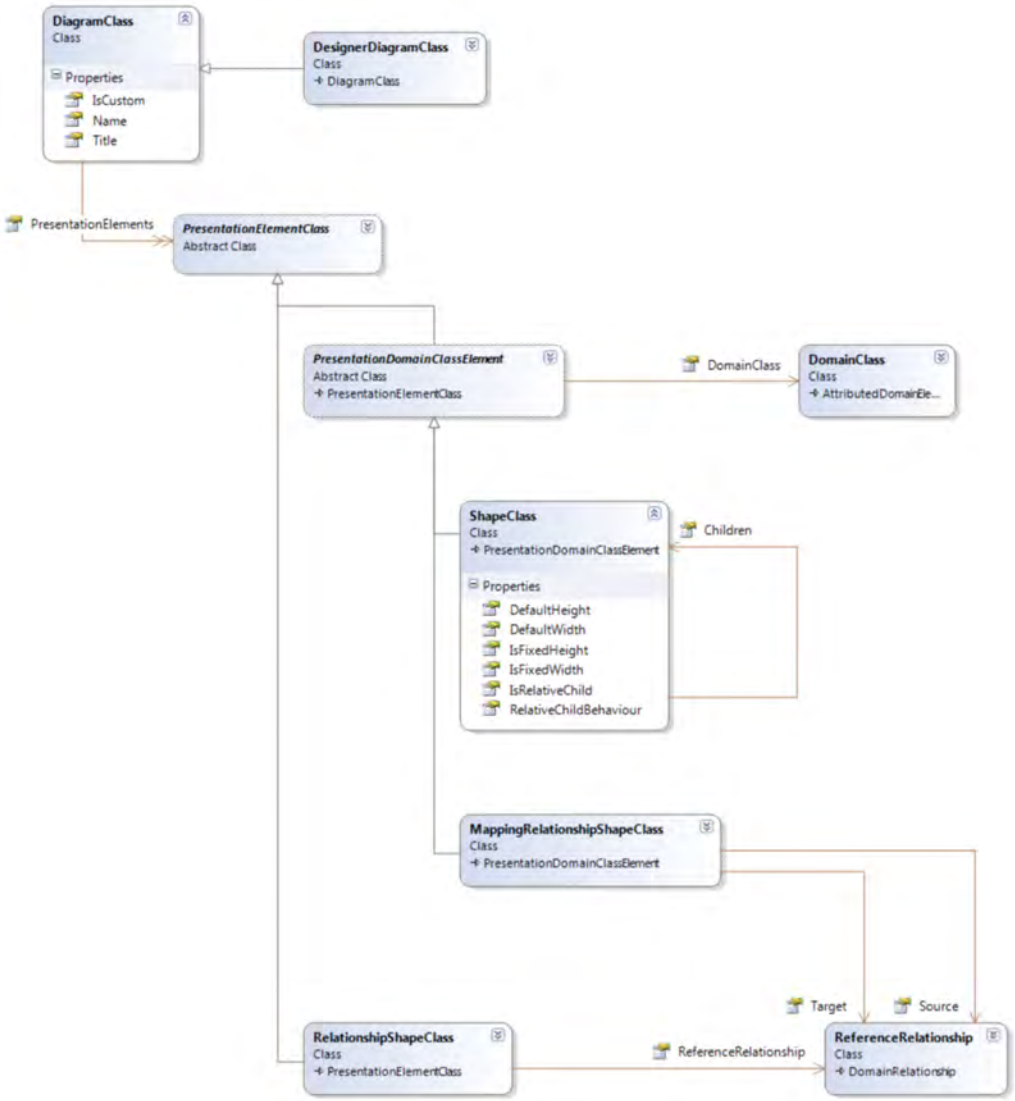


Abbildung 5.3: Language DSL: Strukturen des Diagram Modells

Domänenklassen sowie die Integrationsbeziehungen aufgespannt wird, aufbaut. Somit bezieht das Serialisierungsmodell auch nur Elemente in seinen Aufbau mit ein, die von der Domänenklasse mit *IsDomainModel=true* erreichbar sind.

Auf der Designer-Ebene sind dabei unter anderen folgende Anpassungsmöglichkeiten vorgesehen:

- Namenswahl der zu serialisierenden Klassen, Beziehungen oder Eigenschaften.
- Reihenfolge der Kindelemente (dazu zählen alle definierten Beziehungsziele).
- Reihenfolge der Attributelemente (das sind alle Eigenschaften, die über Attribute serialisiert werden können).
- Auswahl der Serialisierungsart bei Eigenschaften. So können diese als Kind- oder Attributelemente gespeichert werden.
- Ausschluss aus dem Serialisierungsmodell für Eigenschaften oder Beziehungen. Damit können beispielsweise bestimmte Hilfselemente oder -eigenschaften von der Speicherung ausgeschlossen werden.
- Vollständiges oder kurzes Serialisierungsformat bei Beziehungen, was insbesondere bei Referenzbeziehungen interessant ist. Das kurze Format sieht dabei vor, dass ein Verweis auf das Ziel der Beziehung direkt bei der Quelle als Kindelement mitgespeichert wird. Das vollständige Format speichert eine Beziehung als Kindelement der Quelle und weist somit z.B. auch eine *Id*-Eigenschaft auf.

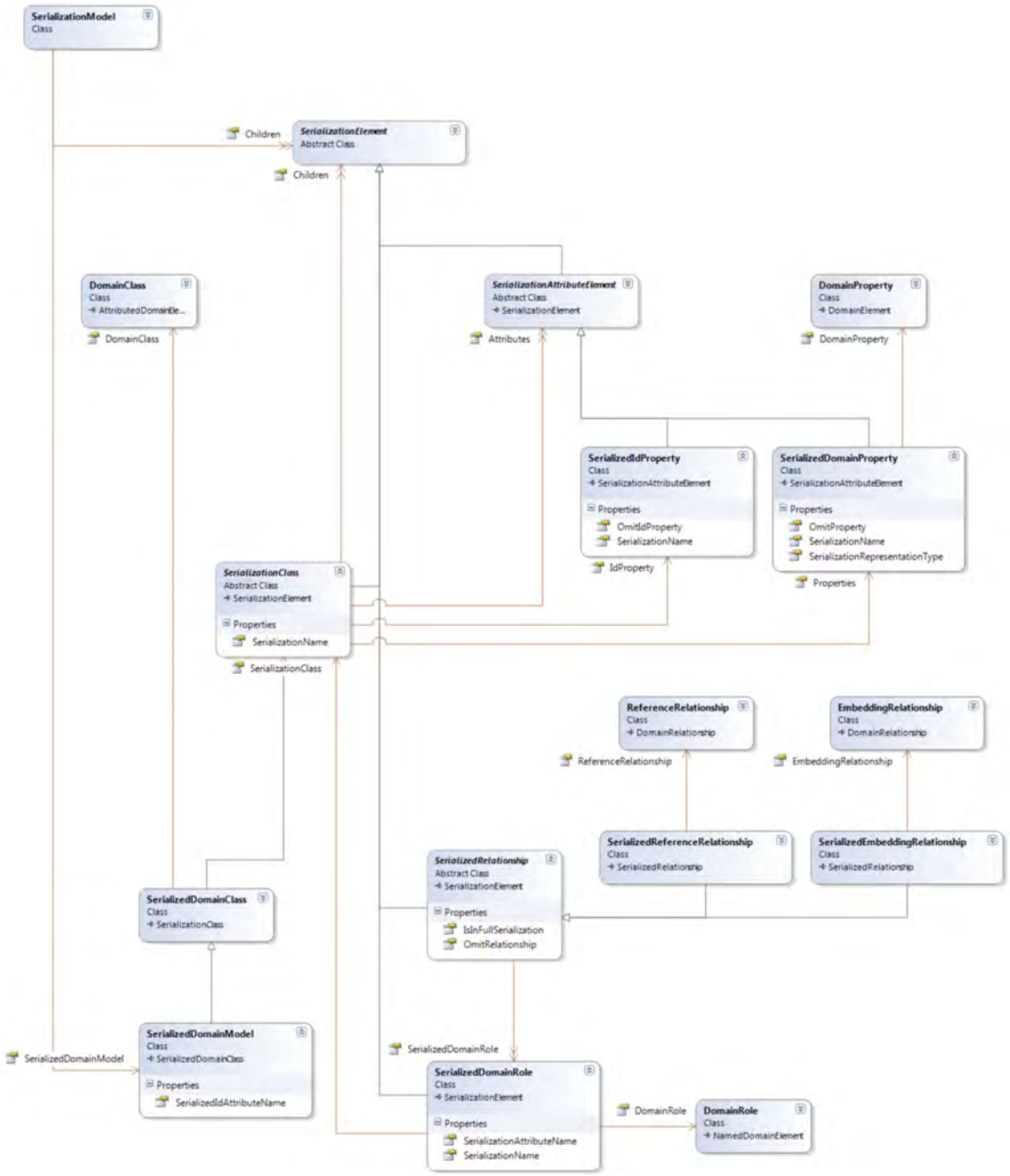


Abbildung 5.4: Language DSL: Strukturen des Serialisierungsmodells

Die wesentlichen Elemente des Serialisierungsmodells sind:

SerializedIdProperty repräsentiert die Serialisierungsinformation für eine ID-Eigenschaft, die ausweist unter welchem Namen (*SerializationName*) die ID-Eigenschaft gespeichert werden soll. Der Name wird allerdings global beim *SerializedDomainModel*-Element festgelegt. Weiterhin kann auch über *OmitIdProperty* für die ID-Eigenschaft spezifiziert werden, ob diese überhaupt zu serialisieren ist oder nicht.

SerializedDomainProperty definiert, wie eine Eigenschaft (*DomainProperty*) zu serialisieren ist. Dabei kann der Serialisierungsname (*SerializationName*) festgelegt werden, so dass sich dieser vom Namen der Eigenschaft auch unterscheiden kann. Ferner lässt sich über *OmitProperty* spezifizieren, ob die Eigenschaft einer Serialisierung bedarf. Schließlich erlaubt *SerializationRepresentationType* die Festlegung ob die Eigenschaft als Kind beziehungsweise Attribut oder aber als Inhalt des Elements, zu dem es gehört, gespeichert werden soll.

SerializedDomainClass beschreibt wie eine *DomainClass* serialisiert wird. Wichtig ist, dass jede einzelne *DomainClass* im Domänenmetamodell diese Information aufweist. Festlegen lässt sich wieder der Name (*SerializationName*), unter dem die Klasse gespeichert werden soll, sowie weiterhin die Reihenfolge der zu speichernden Attribute beziehungsweise Kindelemente.

SerializedDomainModel unterscheidet sich von *SerializedDomainClass* nur indem, dass es eine spezielle *DomainClass* repräsentiert, nämlich die *DomainClass*, die als das oberste Element mittels *IsDomainModel=true* ausgewählt wurde. Somit ist *SerializedDomainModel* auch das oberste Element in der Hierarchie des Serialisierungsmodells und es existiert zu jeder Zeit nur ein einziges solches Element.

SerializedEmbeddingRelationship repräsentiert die Serialisierungsinformation für eine Integrationsbeziehung (*EmbeddngRelationship*). Es lässt sich über *OmitRelationship* festlegen, ob die Beziehung und damit insbesondere ihr Ziel serialisiert werden sollen oder nicht. Weiterhin spezifiziert die Eigenschaft *IsInFullSerialization* die Serialisierungsform der Beziehung. So wird bei kurzer Form das Beziehungsziel, genauer genommen das *SerializedDomainClass* des Ziels, direkt als Kind beim *SerializedDomainClass* der Quellklasse der Integrationsbeziehung gespeichert. In vollständiger Form hingegen, wird die Beziehung selbst unter dem anpassbaren Serialisierungsnamen (*SerializationName*) als Kind beim *SerializedDomainClass* der Quellklasse gespeichert, während das Ziel der Beziehung nun entsprechend dem *SerializedEmbeddingRelationship* selbst hinterlegt wird.

Wichtig ist, dass die vollständige Serialisierungsform nur dann notwendig ist, falls die Integrationsbeziehung über Eigenschaften verfügt oder aber falls die ID-Eigenschaft unbedingt mitgespeichert werden soll. Somit kann auch nur in vollständiger Form die Serialisierungsreihenfolge der Eigenschaften der Beziehung festgelegt werden.

SerializedDomainRole beschreibt die Serialisierungsinformation für eine Domänenrolle (*DomainRole*). Diese erlaubt die Definition eines Serialisierungsnamens (*SerializationName*) sowie des Attributnamens (*SerializationAttributeName*), der den Verweis auf das Element enthalten soll, das die Domänenrolle referenziert. Folgend enthält die *SerializedDomainRole* einen Verweis auf die Serialisierungsinformation des Zielelements, das entweder eine *SerializedDomainClass* oder eine *SerializedReferenceRelationship* darstellt.

SerializedReferenceRelationship stellt die Serialisierungsinformation für eine *ReferenceRelationship* bereit. Jede *SerializedReferenceRelationship* verfügt über genau zwei *SerializedDomainRole*, die die Quell- und die Ziel-Domänenrolle mit den jeweiligen Serialisierungsdefinitionen darstellen.

Bei der *SerializedReferenceRelationship* kann über *OmitRelationship* festgelegt werden ob die Referenzbeziehung zu speichern ist oder nicht. Ferner spielt auch hier die Eigenschaft *IsInFullSerialization* eine entscheidende Rolle bezüglich der Serialisierungsform. So wird in kurzer Form nur ein Verweis auf das Ziel der Beziehung als Kind beim *SerializedDomainClass* der Quellklasse hinterlegt. Dieser setzt sich zusammen aus dem festgelegten Attributnamen der Ziel-Domänenrolle und dem ID-Wert des Zielelements.

In vollständiger Form hingegen wird die Serialisierungsinformation der Beziehung beim *SerializedDomainClass* der Quellklasse hinterlegt. Hier, wie auch schon beim *SerializedEmbeddingRelationship*, kann nun die Reihenfolge der Eigenschaften festgelegt werden, falls die Referenzbeziehung über diese verfügt.

Bei der Erstellung des Serialisierungsmodells spielen Vererbungsbeziehungen eine wichtige Rolle. Wird nämlich eine Vererbungsbeziehung zwischen zwei Domänenklassen x und y definiert, so bedeutet das, dass x von y abgeleitet ist. Damit sind aber alle Eigenschaften von y auch bei x vorhanden. Für das Serialisierungsmodell bedeutet das, dass bei der Erstellung einer Vererbungsbeziehung auch die Serialisierungsinformation der Eigenschaften und Kindelemente (dazu gehören auch referenzierte Elemente) kopiert und der Serialisierungsinformation der abgeleiteten Klasse hinterlegt werden müssen. Wird eine Vererbungsbeziehung entfernt, so muss die Serialisierungsinformation der Eigenschaften sowie Kindelemente entsprechend entfernt werden. Notwendig sind diese Kopier- und Entfernungsvorgänge, um die Reihenfolge der Kind- und Attributelemente spezifisch festlegen zu können, denn diese müssen bei einer abgeleiteten Klasse x nicht zwingend gleich sein wie bei der Basisklasse y.

Das Serialisierungsmodell wird in der Domain DSL zur Generierung von Klassen benutzt, die das Laden und Speichern von Modellen ermöglichen. Diese Klassen werden als partielle Klassen

generiert und können bei Notwendigkeit angepasst oder erweitert werden, indem Funktionalität überschrieben wird. Somit kann das Ergebnis des Serialisierungsmodells einerseits anhand der Anpassung der Strukturen des Serialisierungsmodells und andererseits mittels der Modifikation des generierten Quellcodes beeinflusst werden. Folglich ergibt sich für den Benutzer die Wahl beziehungsweise die Möglichkeit bei seiner Sprache die automatisch generierte Standardserialisierung für das Laden und Speichern zu verwenden, oder aber eine benutzerdefinierte Quellcodelösung vorzugeben, die vollkommen oder teilweise die generierte ersetzt.

5.1.2 Language DSL Designer

Für das Definieren der oben vorgestellten Modelle weist die Language DSL einen entsprechend dafür vorgesehenen grafischen Designer auf. Dieser integriert sich in Microsoft Visual Studio, setzt folglich zum Einsatz auch Visual Studio voraus[1].

Die Language DSL wurde, wie bereits erwähnt, mit den Microsoft DSL-Tools (siehe auch Abschnitt 2.4) erstellt. Somit weist sie auch dieselben grundsätzlichen Darstellungsbereiche auf, die wir hier bezüglich der jeweiligen Einsatzziele beschreiben wollen:

1. *Model Explorer*: Verwaltung von Domänentypen. Hier können externe Typen sowie Enumerationen angelegt, bearbeitet oder gelöscht werden. Ansonsten stehen weitere Löschmöglichkeiten beispielsweise für Klassen oder Beziehungen zur Verfügung. Anlegen lassen sich diese mittels des Model Explorers allerdings nicht.
2. *Property Window*: Dient der Bearbeitung von Eigenschaften.
3. *Designer Surface*: Weicht von der Standardimplementierung in den DSL-Tools ab, indem es drei verschiedene Bereiche zur Verwaltung des Domänenmetamodells anbietet. Genauer genommen sind diese Darstellungsbereiche als Sichten auf die Daten zu verstehen und wurden nach dem *Model View ViewModel* Design Pattern (MVVM) [Smi09] in Verbindung mit *Windows Presentation Foundation* (WPF) implementiert. Wir werden diese Sichten noch weiter unten näher betrachten, die Spezialität und Notwendigkeit sowie die Vorteile des MVVM Design Patterns sehen wir dann später bei den Darstellungsdefinitionen in der Domain DSL.

Die drei Darstellungsbereiche in der Designoberfläche wollen wir nun eingehend anhand eines sehr einfachen Beispiels betrachten, das uns ferner auch zur Diskussion der automatischen Quellcodegenerierung, sprich zur Vorstellung der Ergebnisse derselben, dienen wird.

Beispiel: *Familienstammbaum*: Ein Familienstammbaum ist eine Darstellungsform zur Festhaltung der Nachfahren einer Person. Die Person soll dabei über einen Namen, eine Geschlechtszuweisung sowie die Auflistung von Hobbies verfügen. Jede Person hat weiterhin eine Auflistung von Kindern und weist ferner noch einen Beziehungspartner aus, mit dem sie verheiratet ist.

Das Beispiel wäre an dieser Stelle schon ausreichend, allerdings definieren wir noch die Zuweisung von Haustieren zu den einzelnen Personen, um einige Besonderheiten des Language DSL Designers zeigen zu können (Für das Beispiel selbst ist das natürlich weniger relevant).

Um eine domänenspezifische Sprache für das oben definierte Beispiel erstellen zu können, müssen wir zunächst ein Modell des Beispiels als Domänenmetamodell implementieren (siehe Abbildung 5.5). Dafür legen wir zunächst eine Domänenklasse *Person* an, die uns als Basisklasse dienen soll. Eine Person, die im Stammbaummodell hinterlegt werden kann, nennen wir *FamilyTreePerson*[2]. Damit eine Person auch am Stammbaummodell teilnehmen kann, definieren wir eine Integrationsbeziehung zwischen *FamilyTreeModel*[3] und *FamilyTreePerson*, die *FamilyTreeModelHasFamilyTreePerson* genannt wird.

Jede Person kann nun eine beliebige Anzahl von Kindern haben. Das bedeutet, dass eine Person sich selber referenzieren muss, denn es dient ja als Elternteil für weitere Personen. Deshalb definieren wir eine Referenzbeziehung zwischen *FamilyTreePerson* und *FamilyTreePerson*, die sich

1 Grundsätzlich ist die Notwendigkeit von Visual Studio auch für die automatische Quellcodegenerierung zwingend, denn hier setzen wir auf das Text Templating Transformation Toolkit (Kurz: T4, [T4R07, Syc07]), das in Visual Studio integriert ist.

2 Die Unterscheidung zwischen *Person* und *FamilyTreePerson* führen wir nur ein, um die Darstellung der Vererbungsbeziehung zeigen zu können.

3 Das ist das oberste Element des Domänenmetamodells, sprich die *DomainClass*, die über *IsDomainModel=true* markiert ist.

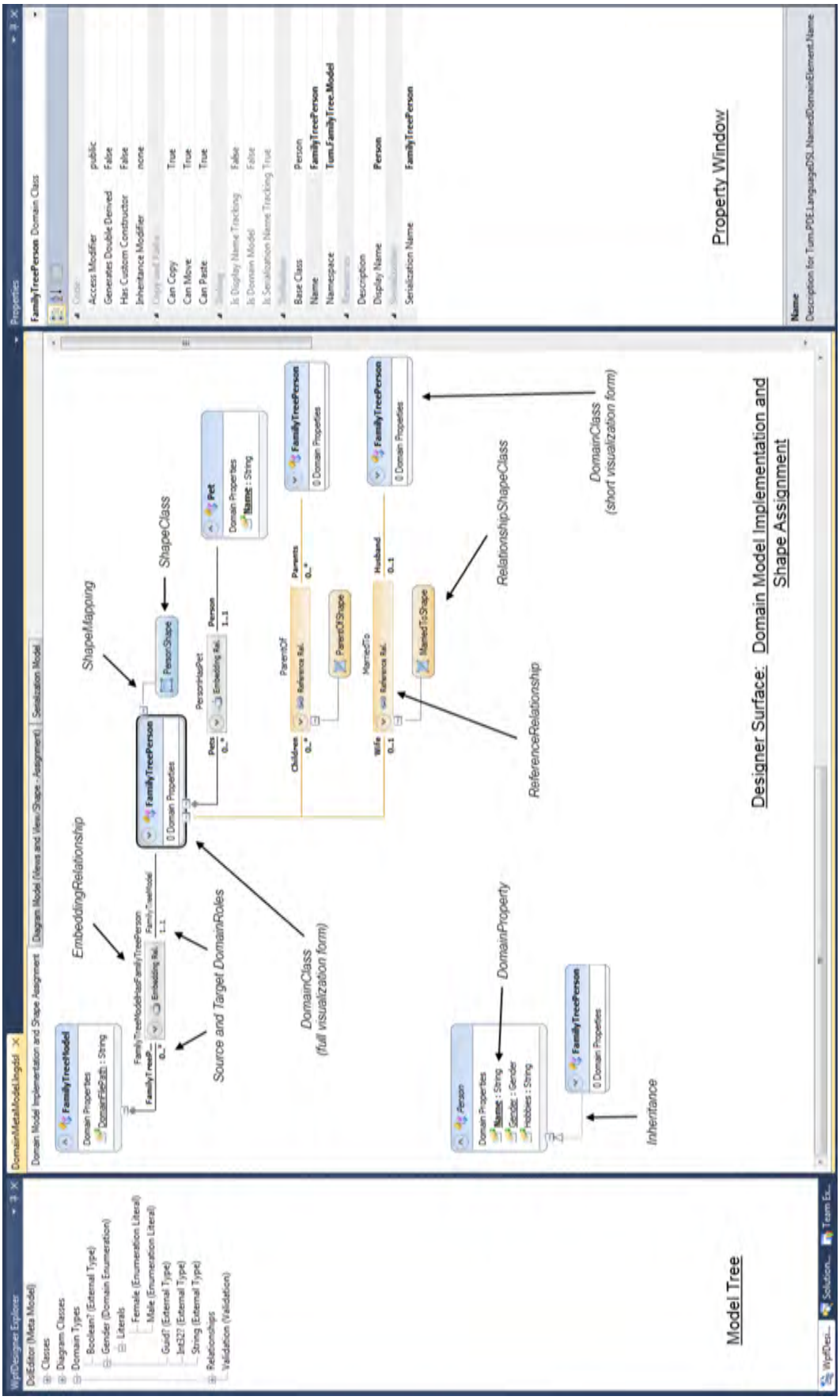


Abbildung 5.5: Language DSL Designer: Darstellung des Domänenmetamodell der Sprache

ParentOf nennt. An dieser Stelle im Modell erlauben wir auch, dass ein Kind beliebig viele Elternteile haben kann, und kümmern uns später bei der Validierung beziehungsweise bei der Erstellung von solchen Beziehungen um die Erhaltung der Konformität der Modelle.

Personen können ferner einen Beziehungspartner ausweisen, mit dem sie verheiratet sind. Dazu erstellen wir eine Referenzbeziehung *MarriedTo* zwischen *FamilyTreePerson* und *FamilyTreePerson*. Dort legen wir die Multiplizitäten so fest, dass jede Person nur an einer einzigen solchen Beziehung teilnehmen kann. Damit schließen wir allerdings nicht aus, dass so eine Beziehung zu sich selbst definiert werden kann. Das muss wieder über die Validierung beziehungsweise beim Vorgang der Erstellung solcher Beziehungen ausgeschlossen werden.

Schließlich erlauben wir in unserm Beispiel die Zuweisung von Haustieren zu einer Person, was wir über die Domänenklasse *Pet* sowie die Integrationsbeziehung *PersonHasPet* zwischen *FamilyTreePerson* und *Pet* berücksichtigen[4].

Darstellung des Domänenmetamodell der Sprache. Die Darstellung des Domänenmetamodell einer Sprache lässt sich Abbildung 5.5 entnehmen. Konzeptuell bauen wir bei dieser Darstellung einen Hierarchiebaum auf, der sich ausgehend von der speziellen *DomainClass*, die mit *IsDomainModel=true* markiert ist, aufspannt. In unserem Beispiel ist das die Domänenklasse *FamilyTreeModel*. Grundsätzlich hätten wir damit aber, ohne ein weiteres beziehungsweise ohne ein erweitertes Konzept, sehr breite Baumdarstellungen, die insbesondere bei größeren Metamodellen kaum mehr zu überblicken geschweige denn zu modifizieren wären. Insofern sehen wir für *DomainClass*-Darstellungen das Konzept des so genannten *Hosts* vor.

Ein *Host* definiert eine Darstellung einer *DomainClass*, die zugleich auch die Darstellung aller weiteren Teilbäume „trägt“. Das bedeutet, dass die entsprechend über Beziehungen aufgespannten Teilbäume beim *Host* dargestellt werden. Dazu zählen

- Integrationsbaum, aufgespannt durch die Integrationsbeziehungen.
- Referenzbaum, definiert durch die Referenzbeziehungen.
- Vererbungsbaum, spezifiziert über die abgeleiteten Klassen.
- Visualisierungsbaum, definiert mittels des *ShapeMappings*[5].

In unserem Beispiel können wir die Domänenklasse *FamilyTreePerson* betrachten. Diese wird auf zwei unterschiedliche Arten dargestellt, davon einmal mittels der vollen Darstellung (das ist die *Host*-Darstellung). Jede Domänenklasse verfügt zu jeder Zeit über genau eine einzige *Host*-Darstellung. Somit können Domänenklassen, die im tatsächlichen Hierarchiebaum auf tieferen Stufen stehen zumindest auf der Oberfläche direkt auf der obersten Stufe angezeigt werden.

Weiterhin lassen sich gewisse Eigenschaften der jeweiligen Elemente direkt anhand der Darstellung ablesen. So lassen sich Elemente, die als abstrakte Klassen generiert werden sollen, anhand eines kursiv dargestellten Namens erkennen, während Eigenschaften, die verpflichtend zuzuweisen sind, über einem unterstrichenen Namen markiert werden. Im Beispiel lässt sich beides der Domänenklasse *Person* entnehmen. Generell haben in der Darstellung des Metamodells unterschiedliche Elemente auch unterschiedliche grafische Darstellungen, die sich insbesondere farblich unterscheiden.

Darstellung des Diagrammodels. Die Darstellung des Diagrammodells dient der Definition von Sichten für die Domain DSL. Dabei können, je nach Notwendigkeit, beliebig viele Sichten definiert werden. Der Language DSL Designer verfolgt diesbezüglich das Ziel, die Sichtdefinition zumindest bei der Verwendung einer diagramartigen Darstellung zu erleichtern. Somit erlaubt er die Definition von *ShapeClasses*, *RelationshipShapeClasses* und *MappingRelationshipShapeClasses* zur Designzeit der Sprache.

Das Konzept hierbei sieht vor, dass *ShapeClasses* auch Kindelemente haben könnten, die in der Darstellung entweder innerhalb oder relativ zu einer *ShapeClass* bewegt werden können. Somit wird auch bei der Darstellung des Diagrammodells ein Hierarchiebaum aufgebaut, allerdings sind dabei nur *ShapeClasses* auf tieferen Stufen vorzufinden, die restlichen Darstellungsmöglichkeiten werden immer auf der obersten Stufe hinterlegt.

4 Bei der hier gewählten Modellierungsweise sind Haustiere immer eindeutig zu einer Person zugeordnet, so dass zwei unterschiedliche Personen nicht die gleichen Haustiere aufweisen können.

5 Als ShapeMapping verstehen wir die Zuweisung einer Darstellungsinformation zu einem Element des Domänenmetamodells.

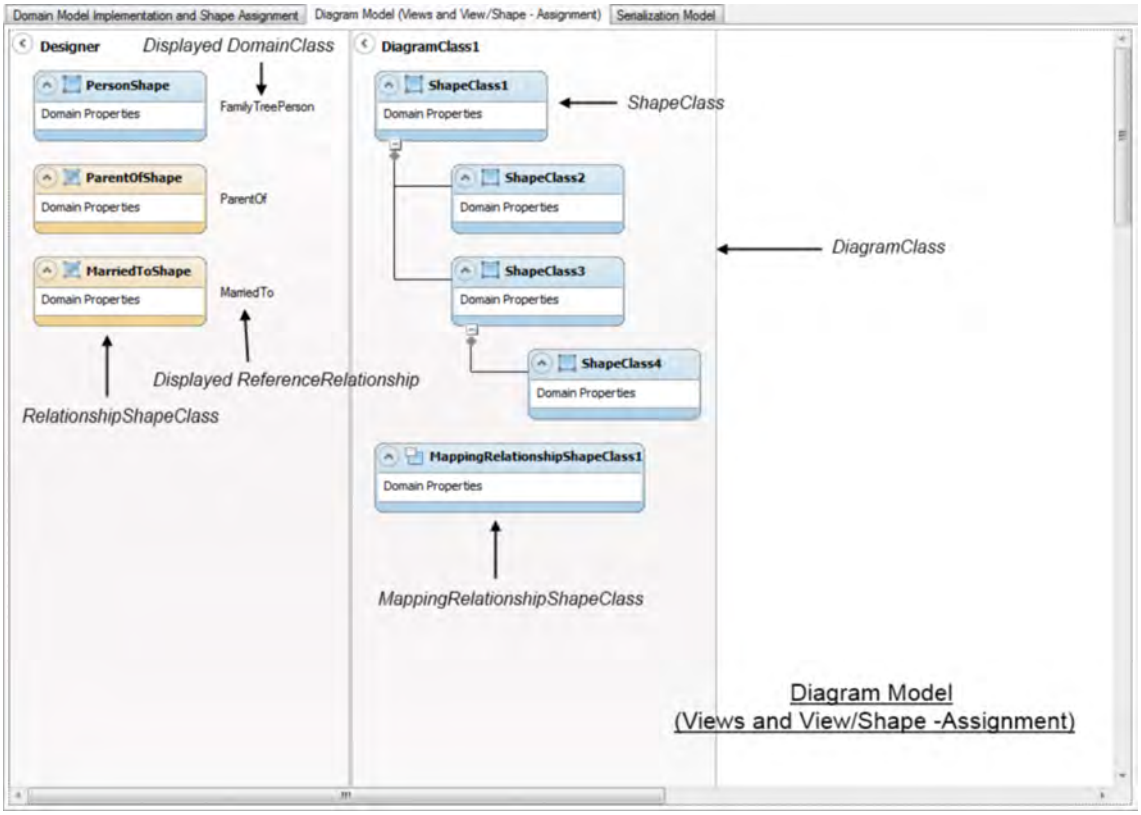


Abbildung 5.6: Language DSL Designer: Darstellung des Diagrammodells

Die genaue Darstellung des Diagrammodells lässt sich Abbildung 5.6 entnehmen. Dort werden auch die jeweils referenzierten, genauer genommen die über eine Darstellungsinformation visualisierten Elemente des Metamodells der Sprache mitdargestellt. Eine Darstellungsinformation muss nicht notwendigerweise ein solches Element im Metamodell referenzieren, wird allerdings erst mit einer solchen Referenz relevant für die Darstellung im Editor der Domain DSL.

In unserem Beispiel erkennt man die Darstellung einer Person über die Darstellungsinformation *PersonShape*, während die Beziehungen über *ParentOfShape* respektive *MarriedToShape* auf dem Standard-Diagramm dargestellt werden. Die Idee hierbei ist es, das *PersonShape* später über eine geometrische Form oder ein Bild darzustellen, während die Beziehungen als Pfeile dazwischen visualisiert werden.

Das zweite Diagramm *DiagramClass1* wurde hier nur zwecks der Möglichkeiten der Diagrammmodelldarstellung abgebildet.

Darstellung des Serialisierungsmodells. Die Darstellung des Serialisierungsmodells lässt sich Abbildung 5.7 entnehmen. Sie kann, ähnlich zu einer XML Schema Definition, als Vorgabe für die Serialisierung von Instanzen des Metamodells der Sprache gesehen werden. Somit kann anhand der Darstellung des Serialisierungsmodells bereits die genaue Struktur der serialisierten Daten abgelesen werden. Insbesondere kann aber die Darstellung genutzt werden, um die Serialisierung anwenderspezifisch zu beeinflussen. Das geht hier im Designer zumindest soweit, wie es das Serialisierungsmodell zulässt. Erweiterte Anpassungen müssen auf Quellcodeebene vorgenommen werden.

Folgend wollen wir nun die Darstellungen der einzelnen Elemente anhand unseres Beispiels erklären.

DomainClass mit *IsDomainModel=true* ist das oberste Element des Serialisierungsmodells. Folglich ist es auch das oberste Element in der Darstellung des Serialisierungsmodells. Von links nach rechts stellen wir dabei erst den Serialisierungsnamen, dann den tatsächlichen Elementnamen dar. In unserem Beispiel sieht man das beim ersten Element in der Baumdarstellung. Dort wird zunächst *FamilyTreeModel* als Serialisierungsname der Domänenklasse

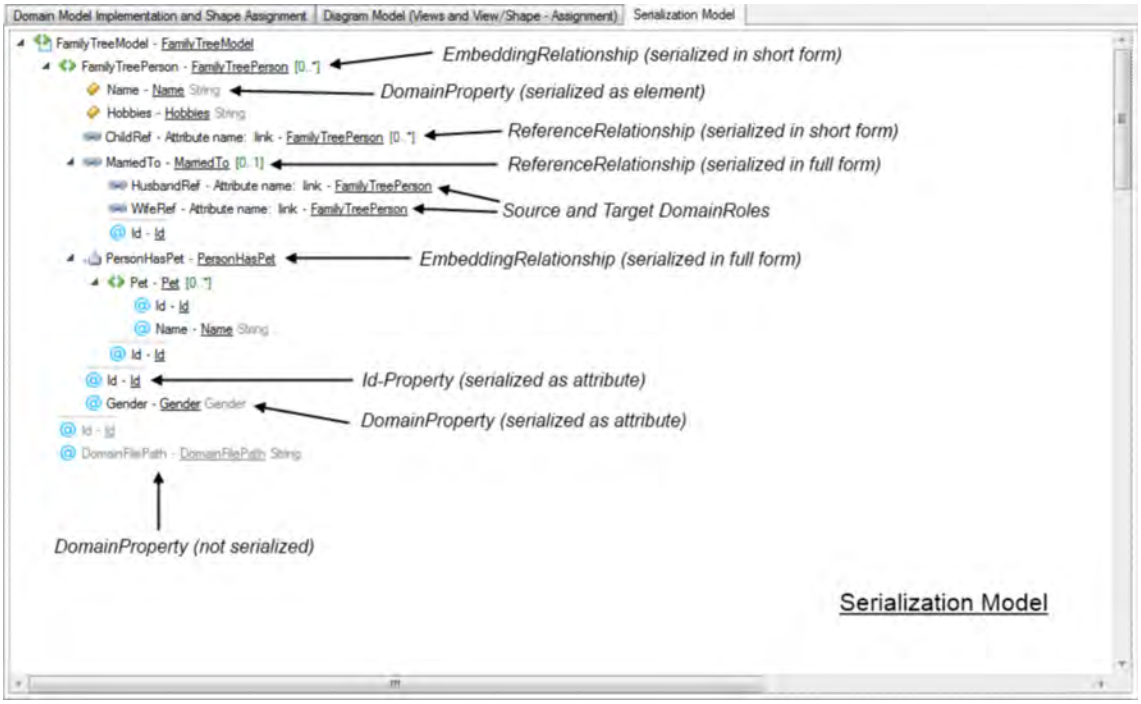


Abbildung 5.7: Language DSL Designer: Darstellung des Serialisierungsmodells

FamilyTreeModel abgebildet.

EmbeddingRelationship Bei der Serialisierung der Integrationsbeziehung unterscheiden wir zwei mögliche Darstellungen:

1. kurze Form: In der kurzen Form wird das Ziel einer Integrationsbeziehung direkt bei der Quelle derselben als Kind gespeichert. Das sehen wir in unserem Beispiel bei den Domänenklassen *FamilyTreeModel* und *FamilyTreePerson*. *FamilyTreePerson* wird dabei als Kindelement mit dem gleichen Serialisierungsnamen direkt bei der Domänenklasse *FamilyTreeModel* serialisiert. Zusätzlich geben wir hier auch die Anzahl der derartig serialisierbaren Kindelemente an, die in diesem Fall mit *[0:n]* nicht beschränkt ist.
2. vollständige Form: In der vollständigen Form wird die Beziehung selbst als Kindelement bei der Quelle gespeichert. Im Beispiel ist das bei der Domänenklasse *FamilyTreePerson* und der Integrationsbeziehung *PersonHasPet* zu sehen. Das Ziel der Beziehung, die Domänenklasse *Pet*, wird folglich als Kindelement der Beziehung selbst abgespeichert und weist auch die mögliche Instanzanzahl der so serialisierbaren Elemente auf, die in diesem Fall mit *[0:n]* nicht beschränkt ist. Folgend wird die Id-Eigenschaft bei der Beziehung als Attribut serialisiert (In der kurzen Form werden weder Id- noch weitere gegebenenfalls vorhandene Eigenschaften mitgespeichert).

ReferenceRelationship unterscheidet bei der Serialisierung zwei mögliche Darstellungsarten, die für die zu erreichende Serialisierungsstruktur ausschlaggebend sind:

1. kurze Form: In der kurzen Form wird ein Verweis auf das Ziel einer Referenzbeziehung direkt bei der Quelle derselben als Kind gespeichert. Der Verweis selbst baut sich aus den Informationen der Serialisierungsinformation der zugehörigen Domänenrolle auf. Im Beispiel sehen wir das bei der Domänenklasse *FamilyTreePerson* und der Referenzbeziehung *ParentOf*. So werden von links nach rechts zunächst der Serialisierungsname, dann der Attributname, dem der Wert der Id-Eigenschaft des Zielelements zugeordnet werden soll, sowie zuletzt der Name des Zielelements, angezeigt. Schließlich wird auch die Anzahl der derartig serialisierbaren Verweise auf Zielelemente im Beispiel mit *[0:1]* dargestellt.
2. vollständige Form: In der vollständigen Form wird die Beziehung selbst als Kindelement bei der Quelle gespeichert. Im Beispiel ist das bei der Domänenklasse *FamilyTreePerson* und der Referenzbeziehung *MarriedTo* zu sehen. Die Verweise auf die Quelle

und auf das Ziel werden hierbei jeweils als Kindelemente der Beziehung gespeichert und weisen neben dem Serialisierungsnamen, dem Attributname, dem der Wert der Id-Eigenschaft des Zielelements zugeordnet werden soll, noch den Namen des Quellelements respektive des Zielelements. Die mögliche Instanzanzahl der so serialisierbaren Elemente wird bei der Beziehung angezeigt. Folgend wird die Id-Eigenschaft bei der Beziehung als Attribut serialisiert.

DomainProperty können entweder als Kinder oder als Attribute eines Elements serialisiert werden. Ersteres sehen wir in unserem Beispiel bei der Domänenkasse *FamilyTreePerson* und der Domäneneigenschaft *Hobbies*, letzeres bei der Domänenkasse *Pet* und der Domäneneigenschaft *Name*. Dargestellt werden Domäneneigenshaften in der Serialisierungsdarstellung mit einem Serialisierungsnamen, dem tatsächlichen Namen und schließlich mit dem Typ der Eigenschaft.

5.1.3 Domain DSL

Die Domain DSL entspricht, wie bereits erwähnt, der zu entwickelnden Zielsprache. Das Domänenmetamodell, sprich das Metamodell der Sprache, wird im Language DSL Editor implementiert und bezüglich spezifischer Eigenschaften angepasst. Das implementierte Metamodell selbst basiert dabei auf den Strukturen der Language DSL.

In diesem Abschnitt wollen wir betrachten was mit dem so erstellen Domänenmetamodell passiert, was genau daraus automatisch generiert wird und welche Teile des Frameworks bereits vor der Generierung vorliegen (schematisch dargestellt in Abbildung 5.8). Dabei wollen wir das oben angesprochene Beispiel (siehe Abschnitt 5.1.2) bezüglich der Ergebnisse der Generierung betrachten und so exemplarisch einen ersten Überblick über die Zusammensetzung einer dafür entwickelnden Zielsprache aufzeigen. Dazu zählen neben dem Domänenmodell, Serialisierungsmodell und Validierungsmodell auch noch die Darstellungsmodelle, die wir bereits als Sichten vorgestellt haben. Hierbei wollen wir bei den Sichten betrachten, wie für die Familienstammbaum-Sprache eine einfache diagramartige Darstellung definiert werden kann. Die Einordnung der Domain DSL in PDE ist wie folgt zu verstehen:

1. Das Domänenmetamodell liegt implementiert vor.
2. Dieses Modell wird genutzt, um das so genannte transformierte Gesamtmodell zu generieren.
3. Das transformierte Gesamtmodell wird in das *ToolFramework* integriert[6].
4. Die Domain DSL beinhaltet als Sprache das nun integrierte *ToolFramework*.

Somit verfügt die Domain DSL neben einem Domänenmodell auch noch über eine Applikation, die für die Modifikationen von Instanzen des Domänenmodells genutzt werden kann. Die Applikation selbst basiert auf dem MVVM Design Pattern und erlaubt somit viele voneinander unabhängige Sichten auf die Modelldaten. Zudem verfügt es noch über ein Erweiterungskonzept mittels Plugins, das auf dem *Managed Extensibility Framework* (MEF) [MEF10] aufbaut.

Automatische Quellcodegenerierung. Zur Betrachtung der automatischen Quellcodegenerierung wollen wir wieder das oben eingeführte Beispiel aufgreifen, um die Ergebnisse derselben aufzuzeigen. Vorweg, zur Quellcodegenerierung benutzen wir das Text Templating Transformation Toolkit (Kurz: T4, [T4R07, Syc07]), das in Visual Studio integriert ist. Somit muss nach jeder Änderung des Domänenmetamodells der Sprache eine Generierung, genauer genommen eine Transformation der Daten des Domänenmetamodells in Quellcode[7], durchgeführt werden. Bei der Transformation werden die folgenden Teilmodelle generiert

- Domänenmodell
- Serialisierungsmodell
- Validierungsmodell (automatisch durchführbarer Teil)
- View Modelle

6 Genau genommen finden Schritte 2 und 3 gleichzeitig statt. Das generierte Gesamtmodell liegt nicht als separates Modell vor, allerdings trennen wir hier diese Schritte übersichtshalber.

7 Wir generieren in PDE Quellcode in der Programmiersprache C#, im Rahmen von T4 wären sehr wohl auch weitere Programmiersprachen und Zielformate definierbar.

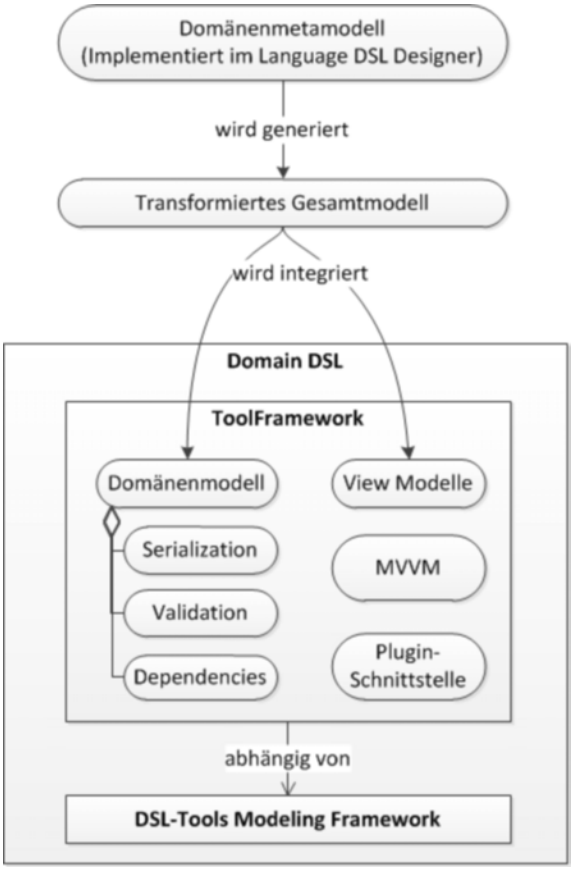


Abbildung 5.8: Einordnung der Domain DSL in PDE

und in das bereits vorliegenden *Tool Framework* integriert. Das *Tool Framework* stellt ein Basisframework dar, welches auf dem Modellierungsframework der Microsoft DSL-Tools aufbaut und dieses folglich erfordert. Die Integration der Teilmodelle in das *Tool Framework* ist grundsätzlich nach der folgenden Vorgehensweise konzipiert:

Es werden Klassen erstellt, die direkt die Elemente aus dem Domänenmetamodell repräsentieren oder Methoden für dieselben darstellen. Diese Klassen sind entweder von einer Basisklasse abgeleitet oder sie implementieren eine bestimmte Schnittstelle aus dem *Tool Framework*.

Im Folgenden wollen wir die einzelnen Teilmodelle betrachten und einige interessante Eigenschaften kurz diskutieren.

Domänenmodell. Das Domänenmodell stellt eine Instanz des Domänenmetamodells dar und beinhaltet demzufolge die Elemente und Beziehungen aus dem Domänenmetamodell. Dabei werden diese Elemente und Beziehungen als Strukturen im Quellcode generiert, so dass das Domänenmodell als Instanz des Domänenmetamodells vereinfach als die Instanz der obersten Klasse (Domänenklasse mit *IsDomainModel=true*) hieraus gesehen werden kann. Das Domänenmodell beinhaltet ferner noch das Serialisierungs- und das Validierungsmodell, auf die wir weiter unten eingehen werden. Folgend wollen wir nun die Generierung von Domänenklassen und -beziehungen exemplarisch und vereinfacht aufzeigen.

Domänenklassengenerierung Die Generierung einer Domänenklasse wollen wir am Beispiel von *FamilyTreePerson* betrachten. Dazu müssen wir allerdings zuerst ihre Basisdomänenklasse *Person* samt den zugehörigen Eigenschaften näher anschauen (siehe Listing 5.1). Dort ist zunächst zu sehen, dass die Domänenklasse *Person* mittels der gleichnamigen Klasse abgebildet ist, die selbst als *abstract partial* markiert ist. Folgend ist sie abgeleitet von *ModelElement*, das selbst als Basisklasse für alle Modelelemente, sprich Domänenklassen und -beziehungen dient und dem DSL-Tools Modellierungsframework entstammt. Die Notwendigkeit der Verwendung von *ModelElement* und die damit verbundene Abhängigkeit vom besagten Framework erklären wir weiter unten.

Person verfügt ferner über die so genannte *DomainClassId*. Das ist die Identifikation des Typs der Domänenklasse in der Domain DSL und gleichzeitig der Wert der ID-Eigenschaft der besagen Domänenklasse in der Language DSL. Die *DomainClassId* findet folglich Verwendung beispielsweise bei der Suche nach Elementen eines spezifischen Typs. Ähnlich zur *DomainClassId* ist die *NameDomainPropertyId* die Identifikation der Eigenschaft *Name* bei der Domänenklasse *Person*. Die Eigenschaft *Name* wird über die gleichnamige Eigenschaft ausgelesen sowie zugewiesen[8].

Schließlich wollen wir noch den Konstruktor bei *Person* ansprechen. Interessant hierbei ist die Notwendigkeit einer so genannten *Partition*. Diese entstammt wieder dem DSL-Tools Modellierungsframework und erlaubt es uns, Instanzen der Klasse *Person* einem so genannten *In-Memory Store* (kurz Store, [CJKW07]) zuzuweisen. Dieser stellt unter anderem bereit:

- Erstellung, Löschung und Modifikation von Elementen auf der Instanzebene
- Transaktionen
- Rules
- Undo und Redo
- Zugriff auf das Domänenmetamodell

Insofern stellt der *Store* wichtige Basisfunktionalitäten bereit, die wir über die Verwendung des DSL-Tools Modellierungsframework als Basis von PDE nutzen können. Folglich entfällt an dieser Stelle der recht große Aufwand der Bereitstellung obiger Funktionalität.

Listing 5.1: Generierung einer Domänenklasse am Beispiel der abstrakten Klasse *Person*

```
namespace Tum.FamilyTree.Model
{
    public abstract partial class Person : ModelElement
    {
        public static readonly new Guid DomainClassId = new Guid("82667313-9a82-485
            d-9aa8-8f322865b14e");

        /// <summary>
        /// Constructor.
        /// </summary>
        /// <param name="partition">Partition where new element is to be created.</
            param>
        /// <param name="propertyAssignments">List of domain property id/value
            pairs to set once the element is created.</param>
        protected Person(Partition partition, PropertyAssignment[]
            propertyAssignments)
            : base(partition, propertyAssignments)
        {
        }

        public static readonly Guid NameDomainPropertyId = new Guid("fce32e4a-94e7
            -495a-b620-73b283a2a72f");
        private global::System.String namePropertyStorage = null;

        public global::System.String Name
        {
            get { return namePropertyStorage; }
            set { NamePropertyHandler.Instance.SetValue(this, value); }
        }

        // ...
    }
}
```

Person ist wie bereits erwähnt die Basisdomänenklasse von *FamilyTreePerson*. Folglich lässt sich die Beziehung auch auf Klassenebene dem Listing 5.2 entnehmen. Damit verfügt natürlich die Klasse *FamilyTreePerson* auch über die Eigenschaften der Klasse *Person*, hat allerdings eine andere *DomainClassId*, spezifiziert es schließlich den Typ einer anderen Domänenklasse.

Interessant sind nun die Eigenschaften *Children* und *Parent*. Diese sind stellvertretend für mögliche Kinder sowie Eltern zu einer Instanz der Klasse *FamilyTreePerson*. Diese entstammen der Beziehung *ParentOf* und werden hier generiert, um Beziehungsinstanzen dieses

8 Die Hilfsklasse *NamePropertyHandler* übernimmt die Zuweisung von neuen Werten für die entsprechende Eigenschaft.

Typs anlegen oder auch löschen zu können. Die Funktion *GetRoleCollection* bei *Children* greift hierbei auf die Funktionalität der Klasse *ModelElement* zur Ermittlung vorhandener Instanzen der Beziehung *ParentOf* mit der aktuellen Klasse als Beziehungsquelle und liefert die Ziele solcher Beziehungen zurück.

Listing 5.2: Generierung einer Domänenklasse am Beispiel von *FamilyTreePerson*

```
namespace Tum.FamilyTree.Model
{
    public partial class FamilyTreePerson : Person
    {
        // ...

        public static readonly new Guid DomainClassId = new Guid("e8bb96de-cc1e-492
            a-bf1f-a8f9de3d3f8f");

        public virtual LinkedElementCollection<FamilyTreePerson> Children
        {
            get { return GetRoleCollection<LinkedElementCollection<FamilyTreePerson
                >, FamilyTreePerson>(ParentOf.ParentDomainRoleId); }
        }

        public virtual LinkedElementCollection<FamilyTreePerson> Parents
        {
            get { return GetRoleCollection<DslModeling::LinkedElementCollection<
                FamilyTreePerson>, FamilyTreePerson>(ParentOf.ChildDomainRoleId); }
        }

        // ...
    }
}
```

Domänenbeziehungsgenerierung Domänenbeziehungen, zu denen die Referenz- und die Integrationsbeziehungen zählen, werden nach dem gleichen Muster generiert, so dass wir uns hier exemplarisch auf die Referenzbeziehung *ParentOf* (siehe Listing 5.3) beschränken. Zunächst wird hierzu eine gleichnamige Klasse generiert, die abgeleitet ist von der Klasse *ElementLink*, die Teil des DSL-Tools Modellierungsframeworks ist, und für jegliche Beziehungen als Basisklasse dient. Weiterhin weist die generierte Klasse *ParentOf* auch wieder eine *DomainClassId* auf, so dass auf Quellcodeebene ein Verweis auf den Beziehungstyp im Domänenmetamodell erhalten bleibt. Ferner ist ein einfacher Konstruktor zu sehen, der als Quelle und Ziel jeweils eine Instanz der Klasse *FamilyTreePerson* erwartet.

Interessant wird es nun bei der Abbildung der Domänenrollen im Quellcode. Diese sind bei den Eigenschaften *Parent* und *Child* zu sehen, die gleichzeitig über *ParentDomainRoleId* und *ChildDomainRoleId* eindeutig die zugehörigen Domänenrollen im Domänenmetamodell identifizieren. Die Eigenschaftszuweisung sowie -ablesung erfolgt bei den Eigenschaften *Parent* und *Child* mittels statischer Methoden *GetRolePlayer* und *SetRolePlayer* der Klasse *DomainRoleInfo*, die selbst wieder Teil des DSL-Tools Modellierungsframeworks ist.

Listing 5.3: Generierung einer Domänenbeziehung am Beispiel von *ParentOf*

```
namespace Tum.FamilyTree.Model
{
    public partial class ParentOf : ElementLink
    {
        public static readonly new Guid DomainClassId = new Guid("161d9598-2ab0-4
            c0d-884a-213db8aa88d2");

        /// <summary>
        /// Creates a ParentOf link in the same Partition as the given
            FamilyTreePerson
        /// </summary>
        public ParentOf(FamilyTreePerson source, FamilyTreePerson target)
                : base( // ...)
        {
        }

        public static readonly Guid ParentDomainRoleId = new Guid("e363dad0-097b
            -4380-be4d-068aa19c9e8a");
        public virtual FamilyTreePerson Parent
        {
```

```
                get { return (FamilyTreePerson)DomainRoleInfo.GetRolePlayer(this,
                    ParentDomainRoleId); }
                set { DomainRoleInfo.SetRolePlayer(this, ParentDomainRoleId, value); }
            }

            public static readonly Guid ChildDomainRoleId = new Guid("cb01d2b0-c4b2
                -4291-be68-a1618e6fa4e5");
            public virtual FamilyTreePerson Child
            {
                get { return (FamilyTreePerson)DomainRoleInfo.GetRolePlayer(this,
                    ChildDomainRoleId); }
                set { DomainRoleInfo.SetRolePlayer(this, ChildDomainRoleId, value); }
            }

            // ...
        }
    }
```

Die oben exemplarisch gezeigten Generierungsergebnisse sind sehr einfach gehalten und sollen einsteigend zum Verständnis der automatischen Quellcodegenerierung beitragen. Tatsächlich wird allerdings deutlich mehr generiert als aufgezeigt.

Validierungsmodell. Das Validierungsmodell dient der automatischen Überprüfung des Domänenmodells auf Inkonsistenzen. Dabei basiert diese automatische Validierung insbesondere auf den bereitgestellten Informationen aus dem Domänenmetamodell, die sich allerdings nur auf eine rein strukturelle Ebene beschränkt. So ist es möglich

- Domänenbeziehungen bezüglich der festgelegten Multiplizitäten zu überprüfen. Es wird dabei nach fehlenden oder zu oft vorhandenen Ziel- oder Quellelementen gesucht.
- Domäneneigenschaften, die mit *IsRequired=true* markiert sind, auf eine Wertzuweisung zu überprüfen.

Somit ist eine automatische Validierung recht einfach gehalten und muss in der Regel spezifisch erweitert beziehungsweise angepasst werden. Hierzu stellen wir auch eine solche Erweiterungsmöglichkeit bereit[9]. Exemplarisch betrachten wollen wir das bei unserem Beispiel anhand der Klasse *FamilyTreePerson*. Diese Klasse erlaubt nämlich die Definition beliebig vieler Eltern-Elemente und verfügt somit über keine Einschränkung, die diese auf maximal zwei beschränkt. Somit kann es hier leicht zu Inkonsistenzen kommen. In Listing 5.4 ist dargestellt, wie eine Validierungsmethode definiert werden kann, um derartige Probleme herauszufinden.

Die Validierung für die Klasse *FamilyTreePerson* wird über die Partialität der Klasse definiert. Zunächst muss festgelegt werden, dass für diese Klasse die Validierung aktiviert ist. Das wird über *[ModelValidationState(ModelValidationState.Enabled)]* spezifiziert. Ferner wird in derselben Klasse eine Methode definiert, die bei der Validierung aufgerufen werden soll. Bei dieser Methode wird über das Attribute *ModelValidationMethod* spezifiziert, zu welchen Zeitpunkten die Validierung diese Methode aufrufen darf.

Listing 5.4: Validierung am Beispiel von *FamilyTreePerson*

```
namespace Tum.FamilyTree.Model
{
    /// <summary>
    /// Partial class used to validate FamilyTreePerson.
    /// </summary>
    [ModelValidationState(ModelValidationState.Enabled)]
    public partial class FamilyTreePerson
    {
        /// <summary>
        /// Automatically validates FamilyTreePerson.
        /// </summary>
        [ModelValidationMethod(ModelValidationCategories.Open |
            ModelValidationCategories.Save | ModelValidationCategories.Menu)]
        public virtual void Validate(ModelValidationContext context)
        {
            // validate parents count
            if (this.Parents.Count > 2)
```

9 Anpassungsmöglichkeiten sind schlichtweg über das Überschreiben von Methoden der generierten Klasse der automatischen Validierung möglich.

```
            {
                context.AddMessage(new ModelValidationMessage(
                    FamilyTreeEditorValidationMessageIds.ParentsCountMessageId,
                    ModelValidationViolationType.Error, 'A child can only have up to two
                        parents.', this));
            }
        }
    }
}
```

Wir halten fest: Validierung kann über die Definition benutzerdefinierter Methoden in partiellen Klassen der Modellelemente des Domänenmodells erweitert werden. Dazu muss allerdings zunächst die Validierung für die entsprechenden Klassen aktiviert werden. Eine Validierung mittels Methoden außerhalb von Modellelementklassen ist direkt nicht möglich, allerdings spricht nichts dagegen, aus den Methoden der Modellelementklassen andere Klassen aufzurufen, die beispielsweise auch in einer eigenen Bibliothek liegen können.

Serialisierungsmodell. Das Serialisierungsmodell dient dem Laden und Speichern des Domänenmodells in XML-Format. Hier wird für jedes Element des Domänenmetamodells eine eigene Klasse generiert, die das entsprechend übernehmen soll. Jede dieser Klassen erlaubt es die Art und Weise der Speicherung zu beeinflussen. So können beispielsweise Methoden zur Speicherung von Eigenschaften als Attribute überschrieben und benutzerspezifisch angepasst werden. Genauso lässt sich die Gesamtserialisierung auch in Quellcode schreiben, so dass die generierte Serialisierung umgangen wird.

Wir wollen allerdings in dieser Arbeit nicht weiter darauf eingehen, da die hierbei generierten Klassen eher unwesentlich für das Verständnis des Gesamtkonzepts sind.

View Modelle. View Modelle dienen der Darstellung und der Modifikation des Domänenmodells. Die gesamte Darstellung folgt hierbei dem MVVM Design Pattern, und erlaubt folglich, dass viele unabhängige Sichten für die Modelldaten definiert werden können. Im Folgenden wollen wir neben dem Gesamtüberblick über die View Modelle noch einige Besonderheiten aufzeigen, insbesondere sind die View Modelle für die diagrammartige Visualisierung, basierend auf der Definition im Diagrammodell, interessant. Wir betrachten die View Modelle folgend anhand unseres Beispiels (siehe Abbildung 5.9).

Wie bereits erwähnt, werden beziehungsweise können viele Sichten für eine DSL definiert werden, so dass auch eine Hauptsicht vorliegen muss, die diese vereinigt. Diese Hauptsicht dient als eine Art Container, in dem alle anderen Sichten platziert werden können. Bei unserer Familienstammbaum DSL ist diese mit der Klasse *FamilyTreeEditorMainViewModel* gegeben, die ihrerseits von der Basisklasse *MainViewModel* abgeleitet ist. Man beachte, dass die erste Klasse automatisch generiert wird, während die zweite Klasse als Basisklasse bereits im *ToolFramework* vorliegt. *MainViewModel* ist also die Hauptsicht und stellt die folgenden Standardsichten bereit

ModelTree dient der Navigation sowie der Verwaltung des Hierarchiebaums innerhalb des Domänenmodells. Insofern können hier neue Elemente basierend auf Integrationsbeziehungen aus dem Domänenmetamodell erstellt werden. Für die Beispiel DSL wurde hierbei die Klasse *FamilyTreeEditorModelTreeViewModel* generiert, die selbst von der Basisklasse *ModelTreeViewModel* aus dem *ToolFramework* abgeleitet ist. Die Idee einer solchen Generierung besteht darin, dass hierdurch Eigenschaften und Methoden, die in den jeweiligen Basisklassen vorhanden sind, in den automatisch generierten gegebenenfalls durch den Benutzer überschrieben und angepasst werden können.

PropertyWindow erlaubt die Modifikation von Eigenschaften des selektierten Elements. Hierzu zählen auch diejenigen Domänenrollen aus den Referenzbeziehungen, an denen das selektierte Element teilnimmt[10].

Das Eigenschaftsfenster wird als Sicht in unserem Beispiel in Form der Klasse *FamilyTreeEditorPropertyGridViewModel* dargestellt, die selbst von der Basisklasse *MainPropertyGridViewModel* abgeleitet ist. Die automatische Generierung erlaubt hierbei auch eine flexible Anpassung. So kann (über den Language DSL Designer) ein View Modell für einen bestimmten

10 Jedes Element wird bei einer Beziehung über eine Domänenrolle repräsentiert. Im Eigenschaftsfenster werden diejenigen Domänenrollen dargestellt, die den Gegensatz (*Opposite DomainRole*) zur Domänenrolle bilden, die das selektierte Element repräsentiert.

Typ oder aber auch spezifisch für eine beliebige Domäneneigenschaft sowie -rolle angegeben werden, so dass dieses bei der Darstellung im Eigenschaftsfenster auch Verwendung findet. Damit können insbesondere für benutzerdefinierte Typen auch passende Editoren erstellt werden[11].

ErrorList stellt Fehler, Warnungen oder sonstige relevante Informationen für den Benutzer dar. In unserem Beispiel wird diese Sicht über die generierte Klasse *FamilyTreeEditorErrorListViewModel* abgebildet, die ihrerseits auf *MainErrorListViewModel* aus dem *ToolFramework* basiert.

Fehler werden in der Fehlerlistensicht mit Quelle und Ursache dargestellt, so dass sie sich auch entsprechend der Fehlerquelle ansteuern und beheben lassen. Zudem lassen sich Fehler in dieser Sicht auch filtern, falls sie für das aktuelle Modell nicht relevant sind. Die so gefilterten Fehler werden ausgeblendet, können jedoch jederzeit vom Benutzer wieder angezeigt werden.

DependeciesWindow ist für die Darstellung von Abhängigkeiten, die bezüglich des selektierten Elements existieren, verantwortlich. Abhängigkeiten sind genauer genommen Instanzen der Beziehungen, die im Domänenmetamodell definiert wurden. Somit erhalten wir vier Abhängigkeitstypen, wovon insbesondere die ersten zwei die Wesentlichen darstellen:

- *Referencing*: Das aktuell selektierte Element ist Quelle einer Referenzbeziehung.
- *Referenced*: Das aktuell selektierte Element ist Ziel einer Referenzbeziehung.
- *Embedding*: Das aktuell selektierte Element ist Quelle einer Integrationsbeziehung.
- *Embedded*: Das aktuell selektierte Element ist Ziel einer Integrationsbeziehung.

Zusätzlich bietet *MainViewModel* neben der Standarddiagramsicht (*FamilyTreeEditorDesignerDiagramSurfaceViewModel*) noch die Möglichkeit zur Definition beliebig vieler weiterer Sichten, die von der Basisklasse *BaseDiagramSurfaceViewModel* abgeleitet sein müssen. Somit ist natürlich die Standarddiagrammsicht sowie die *FamilyTreeEditorDesignerDiagramClass1SurfaceViewModel*, die für die Diagramklasse *DiagramClass1* generiert wird, von dieser Basisklasse abgeleitet.

Die Diagramsichtgenerierung müssen wir nun etwas genauer betrachten, insbesondere wollen wir aufzeigen, wie die im Beispiel definierten Darstellungsinformationen in Form von *PersonShape*, *ParentOfShape* und *MarriedToShape* mittels der automatischen Quellcodegenerierung abgebildet werden.

Diagrammodell im ToolFramework Das *ToolFramework* weist ein eigenes Diagrammodell auf, das bei der Transformation des Domänenmetamodells als Basismodell dienen soll. Es baut wieder auf dem DSL-Tools Modellierungsframework auf und wurde selbst als eine rein modellbasierte[12] DSL implementiert. Wir nennen dieses Diagrammodell von nun an die Diagram DSL, um eine klare Trennung zum Diagrammodell der Language DSL zu erreichen.

Die Diagram DSL verfügt auf ihrer Domänenmetamodell-Ebene über Strukturen zur Definition von Diagrammen sowie von Shapes[13]. Insofern wird zunächst für jedes Shape aus unserem Beispiel auch eine Repräsentation in der Diagram DSL generiert und ferner in einem generierten View Modell weiterverwendet. Das View Modell für *PersonShape* nennt sich zum Beispiel *PersonShapeDiagramItemViewModel*.

Hinweis

Die Diagram DSL ist im Grunde ein Zwischenmodell, das dem Zweck der Strukturdefinition, des Layoutings beziehungsweise des Routings sowie der Serialisierung der Diagraminformation dient.

Das für *PersonShape* generierte View Modell verfügt soweit über keine direkte Visualisierungsinformation. Wir haben gegebenenfalls lediglich eine Standardgröße gegeben, wissen aber nicht wie wir diese Shape tatsächlich darstellen sollen. Dazu muss der Benutzer mittels eines in Xaml [MSD10b] definierten *DataTemplate* [MSD10a] eine Darstellung angeben. Eine Möglichkeit hiervon ist in Listing 5.5 gegeben.

11 Diese Thematik greifen wir später bei der Definition eines HTML-Editors für das V-Modell wieder auf.

12 Als modellbasiert verstehen wir hier eine DSL, die über keinen Designer zur Bearbeitung verfügt. Die Modifikation von ihrem Domänenmodell erfolgt nur auf der Quellcode-Ebene.

13 An dieser Stelle wollen wir nicht weiter darauf eingehen, da das sonst den Rahmen dieser Arbeit sprengen würde.

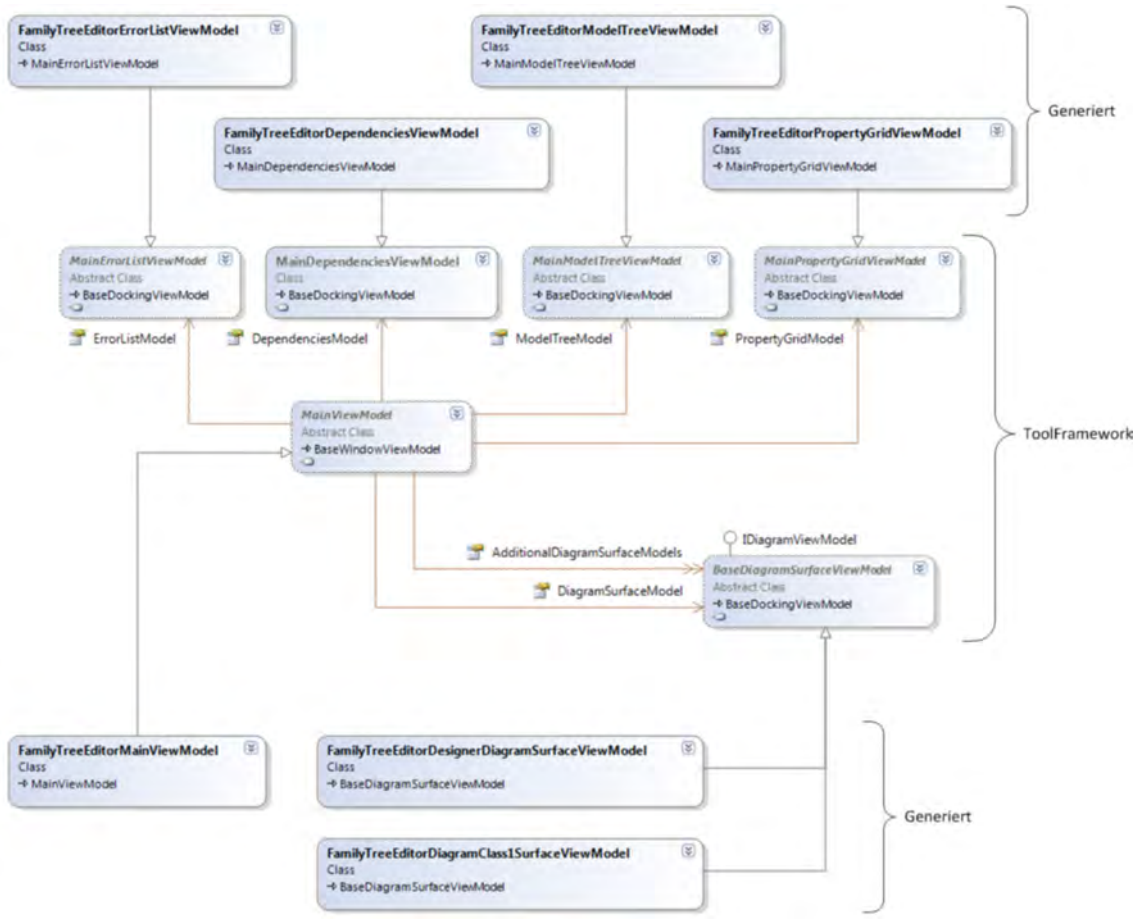


Abbildung 5.9: Views in der Domain DSL in PDE

Wir verwenden somit für die Angabe der Darstellungsinformation Xaml-DataTemplates, so dass eine Flexibilität gegeben ist, die neben einfachen auch sehr komplexe Darstellungen problemlos zulässt. Das ist selbstverständlich nicht die trivialste Art und Weise Darstellungen zu definieren, es erlaubt uns allerdings spezifische UI-Darstellungen anzugeben, die bei umfangreicheren Sprachen mit komplexen inneren Abläufen und Verflechtungen sehr sinnvoll eingesetzt werden können, um die Komplexität zu mindern. Wirklich schwierig sind die Darstellungsdefinitionen ohnehin nur bei komplexen UI-Darstellungen, so dass in den einfachen Fällen (siehe Listing 5.5) dies nur unschwer als Nachteil gelten kann. Eine weitere Entwicklung an dieser Stelle könnte die Darstellungsdefinitionen zumindest bezüglich der Visualisierung über geometrische Formen erleichtern, indem diese direkt im Language DSL Designer unterstützt werden, und über eine Auswahl eine bestimmte geometrische Form angegeben und folgend automatisch generiert werden kann.

Listing 5.5: Definition von Xaml-Templates für die Familienstrammbaum DSL

```
<!-- PersonShape template -->
<DataTemplate DataType='{x:Type c:PersonShapeDiagramItemViewModel}'>
    <StackPanel Orientation='Vertical' IsHitTestVisible='False'>
        <Image x:Name='GenderImage' Source='/Resources/Images/Avatars/MaleAvatar.png'
            HorizontalAlignment='Center' Width='77' Height='77'/>
        <TextBlock Margin='3' Grid.Row='0' HorizontalAlignment='Center' Text='{Binding
            Path=DomainElementName}' TextTrimming='CharacterEllipsis'/>
    </StackPanel>
    <DataTemplate.Triggers>
        <DataTrigger Binding='{Binding Path=Element_Gender, Mode=OneWay}' Value='Female
            '>
            <Setter TargetName='GenderImage' Property='Source' Value='/Resources/Images/
                Avatars/FemaleAvatar.png'/>
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>
```

```
<!-- MarriedToShape template -->
<Style x:Key='MarriedToShapeConnectionPathStyle' TargetType='Path'>
    <Setter Property='StrokeThickness' Value='1'/>
    <Setter Property='Stroke' Value='Gray'/>
</Style>
<DataTemplate DataType='{x:Type c:MarriedToShapeDiagramItemLinkViewModel}'>
    <diag:DiagramDesignerItemLink StartAnchorStyle='Diamond' EndAnchorStyle='Diamond'
        PathStyle='{StaticResource MarriedToShapeConnectionPathStyle}'/>
</DataTemplate>

<!-- ParentOfShape template -->
<DataTemplate DataType='{x:Type c:ParentOfShapeDiagramItemLinkViewModel}'>
    <diag:DiagramDesignerItemLink EndAnchorStyle='Arrow' />
</DataTemplate>
```

Dar Ergebnis der grafischen Darstellung für eine Person in der Familienstammbaum DSL ist der Abbildung 5.10 zu entnehmen. Auf der linken Seite der Abbildung ist die selektierte Person *Anataloj* zu sehen, die über vier Dreiecke an allen Rändern verfügt. Diese werden genutzt, um Beziehungen anzulegen, indem ausgehend von diesen Dreiecken eine Ziehbewegung zum Zielelement der anzulegenden Beziehung gezogen wird. Das ist in Abbildung 5.10 die Person *Maria*. Bei mehreren möglichen Beziehungen zwischen Quell- und Zielelement, muss der Benutzer mittels eines Auswahlfensters entscheiden, welche Beziehung er anlegen möchte. Sind hingegen für ein Element keine Beziehungen möglich, an denen er als Quelle teilnimmt, so werden die Dreiecke entsprechend nicht angezeigt.

Abbildung 5.10: Grafische Darstellung in der Familienstammbaum DSL

Kommunikation zwischen View Modellen View Modelle sind in der Domain DSL komplett unabhängig voneinander. Dennoch kann es notwendig beziehungsweise sinnvoll sein, zwischen den Sichten zu kommunizieren. Dabei geht es um spezifische Nachrichten, die von gewissen Sichten gesendet sowie von anderen entsprechend empfangen werden. Dazu muss allerdings ein spezielles System beziehungsweise Kommunikationsverfahren definiert werden, um einen derartigen Nachrichtenaustausch zu ermöglichen. Das wird im *ToolFramework* mittels des so genannten *EventAggregator* [Eve08] übernommen, der Teil vom MVVM Framework *Prism* [Pri09] ist. Kurz zusammengefasst funktioniert die Kommunikation wie folgt (siehe auch Abbildung 5.11):

1. Sichten übergeben dem *EventAggregator* eine spezielle Methode, die bezüglich eines spezifischen Nachrichtentyps (abgeleitet von *CompositeWpfEvent*) aufgerufen werden soll und werden hierdurch beim *EventAggregator* als so genannte *Subscriber* markiert.
2. Sichten publizieren über den *EventAggregator* spezifische Nachrichten und stellen in diesem Sinne den so genannten *Publisher* dar.
3. *EventAggregator* ruft Methoden bezüglich der publizierten Nachrichten bei jedem Subscriber auf, soweit die Nachrichtentypen (sowie weitere, definierbare Parameter) übereinstimmen.

Im Rahmen unseres Beispiels wollen wir das Kommunikationsverfahren mittels des *EventAggregators* zwischen den Sichten *ModelTree* und der *ErrorList* verdeutlichen, in dem bei jeder Selektion eines Elements eine Informationsnachricht in der Fehlerliste angezeigt wird. Dazu muss zunächst ein Nachrichtentyp definiert werden, der allerdings bereits im Rahmen des *ToolFrameworks* vorliegt (siehe Listing 5.6). Dabei ist die Klasse *BaseErrorListItemViewModel* Teil des *ToolFrameworks*

und muss bei der Definition eines View Models für einen Fehler als Basisklasse benutzt werden[14].

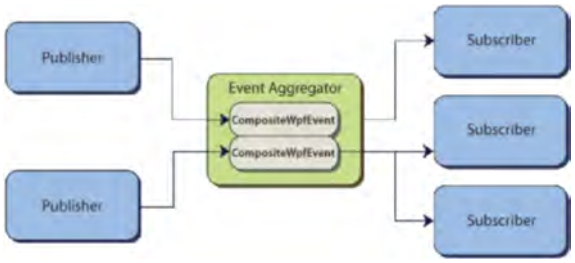


Abbildung 5.11: Event Aggregator

Listing 5.6: Definition eines Nachrichtentyps zum Anzeigen von Fehlern in der Fehlerliste

```
/// <summary>
/// Event to notify that error list items need to be added to the error list.
/// </summary>
public class ErrorListAddItems : ViewModelEvent<List<BaseErrorListItemViewModel>>
{
}
```

Nach der Definition des Nachrichtentyps, müssen wir im View Model der Fehlerlistensicht entsprechend auf das Senden von derartigen Nachrichten reagieren (siehe Listing 5.7). Hierzu registrieren wir eine spezielle Methode namens *AddItems*, die seitens des *EventAggregators* aufgerufen werden soll, sobald eine Nachricht des Typs *ErrorListAddItems* publiziert wird.

Listing 5.7: Reaktion auf das Versenden des Nachrichtentyps zum Anzeigen von Fehlern in der Fehlerliste

```
// subscribe to events of type ErrorListAddItems beeing published
this.EventManager.GetEvent<ErrorListAddItems>().Subscribe(new Action<List<
    BaseErrorListItemViewModel>>(AddItems));

/// <summary>
/// Add multiple error items to the error list.
/// </summary>
/// <param name="items">Error items to add.</param>
protected virtual void AddItems(List<BaseErrorListItemViewModel> items)
{
    // ...
}
```

Schließlich müssen wir nur noch bei der Selektion eines Elements in der *ModelTree*-Sicht eine solche Nachricht publizieren (siehe Listing 5.8).

Listing 5.8: Senden einer Nachricht zum Anzeigen von Fehlern in der Fehlerliste

```
public partial class FamilyTreeEditorModelTreeViewModel
{
    /// <summary>
    /// Callback from SelectionChangedEvent.
    /// </summary>
    /// <param name="eventArgs">SelectionChangedEventArgs.</param>
    protected override void ReactOnViewSelection(SelectionChangedEventArgs eventArgs)
    {
        base.ReactOnViewSelection(eventArgs);

        List<BaseErrorListItemViewModel> items = new List<BaseErrorListItemViewModel>();
        items.Add(new StringErrorListItemViewModel(this.ViewModelStore, "TestErrorId",
            ErrorListItemCategory.Message, "Hello"));

        // publish error items
        this.EventManager.GetEvent<ErrorListAddItems>().Publish(items);
    }
}
```

14 Im *ToolFramework* existieren bereits Definitionen von View Modellen für Fehler, die zum Beispiel über eine Quelle, Ursache sowie eine Textnachricht verfügen.

Transaktionen bei Modifikationen. Für eine Modifikation des Domänenmodells (beispielsweise Anlegen eines neuen Elements) werden so genannte Transaktionen benötigt. Diese entstammen dem DSL-Tools Modellierungsframework und dienen insbesondere zur Bereitstellung der *Rules*[15] sowie der *Undo/Redo* Systeme. Wie eine Transaktion definiert und eingesetzt werden kann, ist in Listing 5.9 dargestellt. Dort wird ein neues Element *FamilyTreePerson* angelegt, mit dem Namen *Eugen* versehen und dem Domänenmodell hinzugefügt.

Listing 5.9: Transaktionen bei Modifikationen

```
using (Transaction transaction = this.ModelData.Store.TransactionManager.
    BeginTransaction('Add new person'))
{
    // add new FamilyTreePerson
    FamilyTreeModel domainModel = this.ModelData.RootElement as FamilyTreeModel;
    FamilyTreePerson person = new FamilyTreePerson(this.ModelData.Store);
    person.Name = "Eugen";
    domainModel.FamilyTreePersons.Add(person);

    transaction.Commit();
}
```

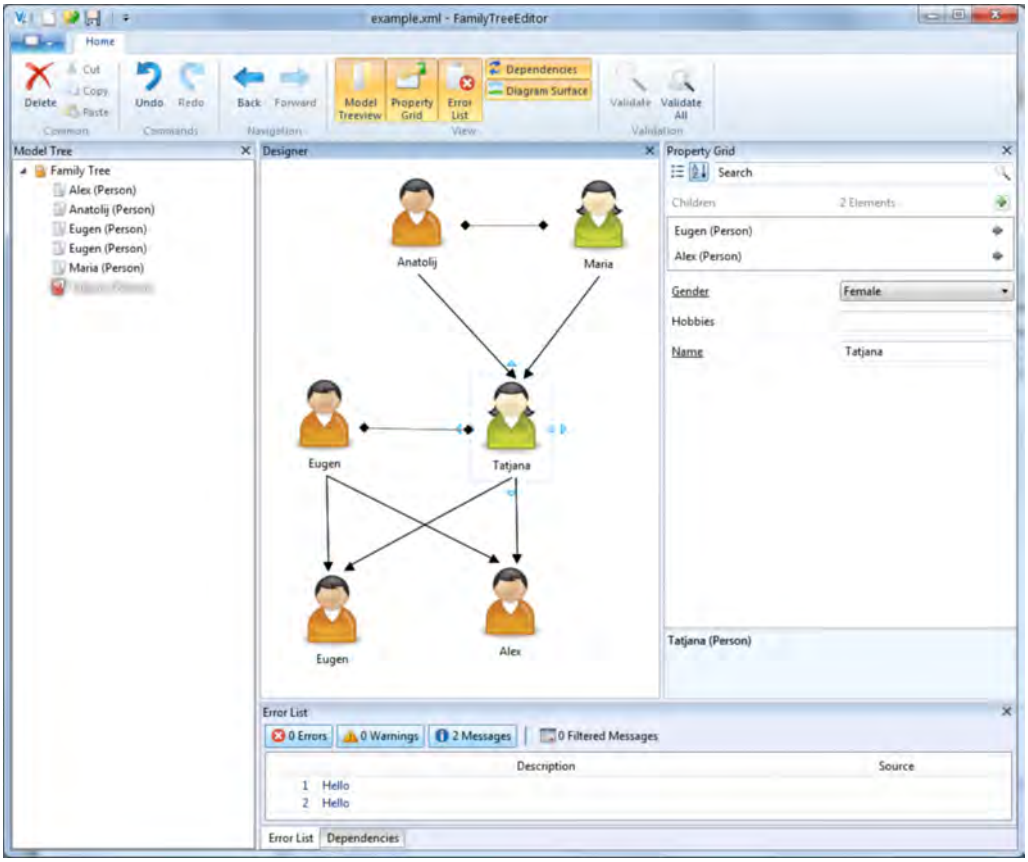


Abbildung 5.12: Editor der Familienstrammbaum DSL

5.1.4 Domain DSL Designer

In diesem Abschnitt wollen wir den Editor der Domain DSL betrachten, um insbesondere auch die Abbildung der oben angesprochenen Sichten vorzustellen. Wir machen das wieder anhand des Familienstammbaum-Beispiels, das wir in Abschnitt 5.1.2 eingeführt haben. Für dieses Beispiel haben wär nämlich eine einfache Sprache erstellt, die standardmäßig auch über einen Designer verfügt, der selbst als eigenständige Applikation vorliegt (siehe Abbildung 5.12[16]).

15 Nähere Informationen hierzu entnehme der interessierte Leser [CJKW07].
16 Die Avatar-Bilder auf der Designoberfläsche des Editors entstammen der Microsoft Clipart Webseite.

Sichten werden im Editor innerhalb so genannter *Docking Windows* dargestellt. Die Anordnung dieser *Docking Windows* im Editor ist nicht zwingend vorgegeben und kann demzufolge auch flexibel angepasst werden. Damit kann der Editor der Sprache auch bezüglich des Layouts seiner Sichten benutzerspezifisch konfiguriert werden.

Hinweis

Im Editor der Domain DSL benutzen wir beim Hauptmenu das so genannte *Ribbon Menu*, das Microsoft in seinen Office Produkten in 2007 eingeführt hat. Dieses moderne Menu erlaubt es auch kontextsensitive Menus zu erstellen, die beispielsweise erst bei bestimmten Editoren des Eigenschaftsbereichs oder aber bei der Selektion eines Elements auf der Designeroberfläche sichtbar werden.

Im Folgenden wollen wir die Darstellungen der Standardsichten (*ModelTree*, *PropertyWindow*, *ErrorList* sowie *DependenciesWindow*) im Editor betrachten. Dabei liegen diese Sichten jedem Spracheditor standardmäßig vor, während die Designoberflächensicht für die Beispielssprache spezifisch implementiert wurde.

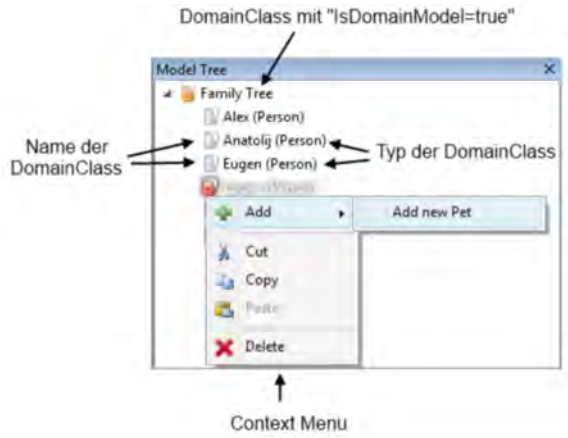


Abbildung 5.13: Darstellung der ModelTree-Sicht in der Domain DSL

ModelTree. Das *ModelTree* dient der Navigation, dem Anlegen von neuen sowie dem Löschen von vorhandenen Elementen. Der strukturelle Aufbau des *ModelTree* ergibt sich basierend auf Integrationsbeziehungen aus dem Domänenmetamodell, wobei das oberste Elemente durch die Domänenklasse, die über „IsDomainModel=true“ markiert wurde, dargestellt wird (siehe auch Abbildung 5.13). Zum Anlegen von neuen Elementen steht ein Kontextmenu zur Verfügung, das selbst automatisch aufgebaut wird und nur diejenigen Optionen anbietet, die tatsächlich für das Domänenmodell gültig sind. Das heißt, dass eine bereits bestehende Integrationsbeziehung nicht nochmals über das Kontextmenu erstellt werden kann, falls die Multiplizitäten so gesetzt sind, dass sie nur einmal vorhanden sein darf und sie gleichzeitig bereits existiert.

Die Darstellung von Elementen im *ModelTree* erfolgt in einer Baumstruktur, so dass unterschiedliche Level jeweils Kind- und Elternelemente ausweisen zwischen denen entsprechende Integrationsbeziehungen vorliegen. Verfügt ein Element über eine Namenseigenschaft bei der zugehörigen Domänenklasse, so wird es mit seinem Namen sowie seinem Typ[17] dargestellt, liegt andererseits keine Namenseigenschaft vor, so wird nur sein Typ angezeigt.

PropertyWindow. Das *PropertyWindow* dient der Bearbeitung von Eigenschaften und Referenzbeziehungen[18] des selektierten Elements. Das Eigenschaftsfenster selbst verfügt über zwei mögliche Darstellungsformen (siehe Abbildung 5.14):

- *Alphabetical Mode*: Die Eigenschaften werden alphabetisch bezüglich des *DisplayName* der Domäneneigenschaft beziehungsweise der -rolle aufgelistet.

17 Das ist der *DisplayName* einer Domänenklasse.
18 Hierbei sind die Domänenrollen gemeint, die den Gegensatz zur Rolle des ausgewählen Elements darstellen.

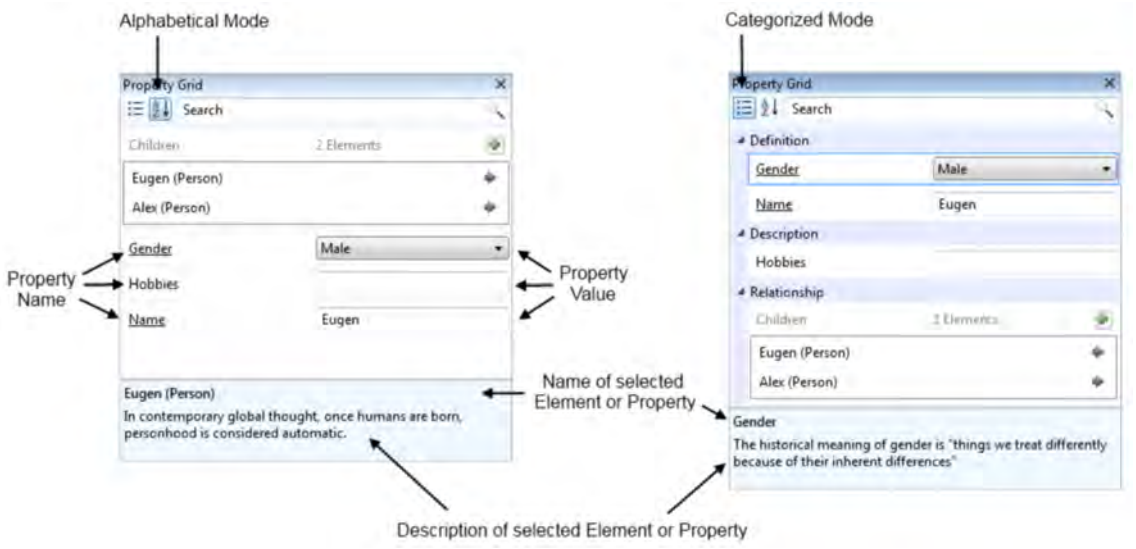


Abbildung 5.14: Darstellung der PropertyWindow-Sicht in der Domain DSL

- *Categorized Mode*: Die Eigenschaften werden in Kategorien dargestellt, die bei der Domäneneigenschaft beziehungsweise der -rolle über *Category*[19] festgelegt wird. Innerhalb der Kategorien wird wieder bezüglich des *DisplayName* alphabetisch sortiert.

Die Darstellung der eigentlichen Editoren für Eigenschaften und Rollen wird durch die Editoren selbst bestimmt, sie folgt allerdings bei den vorliegenden Standardeditoren (*String*, *Boolean*, *Enumeration*, etc.) einem festen Muster. So sind die Namen der zu bearbeitenden Eigenschaft beziehungsweise Rolle immer links oben zu finden, während die Bearbeitung von Werten rechts neben dem Namen sowie unter dem Namen erfolgen kann.

Die Darstellung des Eigenschaftsfensters verfügt schließlich über eine Zusatzinformation im unteren Bereich. Diese wird bezüglich des Selektionskontextes zusammengesetzt, so dass bei einem selektierten Editor der Name der Eigenschaft beziehungsweise Rolle sowie die zugehörige Beschreibung angezeigt wird, während bei keinem selektierten Editor dieselbe Information für das selektierte Element bereitgestellt wird.

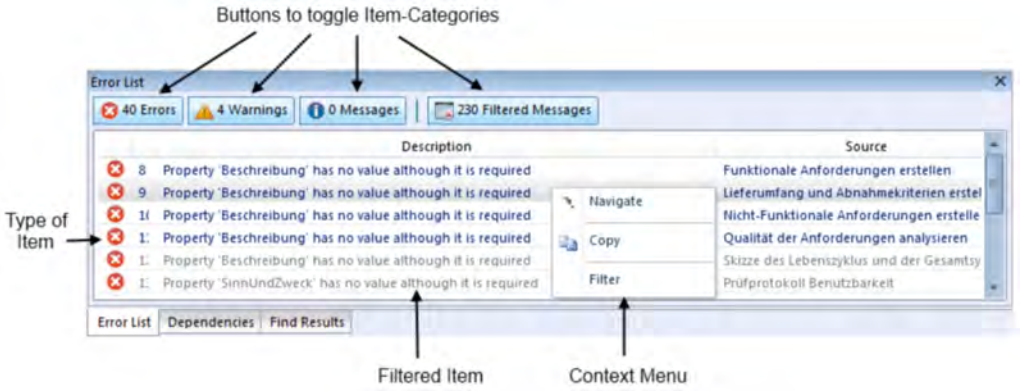


Abbildung 5.15: Darstellung der ErrorList-Sicht in der Domain DSL

ErrorList Die *ErrorList*-Sicht stellt Fehler- sowie Warnungsmeldungen listenartig mit Quelle und Ursache dar (siehe Abbildung 5.15). Sie kann die Darstellung der einzelnen Elemente bezüglich ihrer Fehlerkategorie (Fehler, Warnungen, sonstige Informationen) sowie bezüglich der gefilterten Elemente einschränken und weist zudem ein Kontextmenu auf, das zum einen die

19 Ist bei dieser Eigenschaft kein Wert festgelegt, so wird standardmäßig der Wer „Misc“ verwendet.

Navigation zur Fehlerquelle bereitstellt und zum anderen Elemente als *filtered* und wieder *normal* markieren kann.

Die Darstellung der einzelnen Einträge in der Fehlerliste verfügt neben der Fehlerursache und -quelle noch über die Ausweisung des Typs über ein entsprechendes Bild (ganz links).

DependenciesWindow. Das *DependenciesWindow* stellt Abhängigkeiten dar, die für das selektierte Element vorliegen. Dabei kann die Darstellung bezüglich der Abhängigkeitskategorien eingeschränkt werden, so dass nur diejenigen angezeigt werden, die für den Benutzer aktuell relevant sind. Die einzelnen Einträge im *DependenciesWindow* weisen neben der Kategorie (die zugleich auch als Typ verstanden werden kann) noch das Quell- und Zielelement sowie den Typ der Abhängigkeit auf, wobei ein Kontextmenu die Navigation zum Quell- oder Zielelement ermöglicht.

Abbildung 5.16: Darstellung der DependenciesWindow-Sicht in der Domain DSL

5.1.5 Vorgehensweise bei der Implementierung einer DSL mit PDE

Die Vorgehensweise zur Implementierung einer domänenspezifischen Sprache mit PDE lässt sich wie folgt strukturieren (siehe auch 5.17):

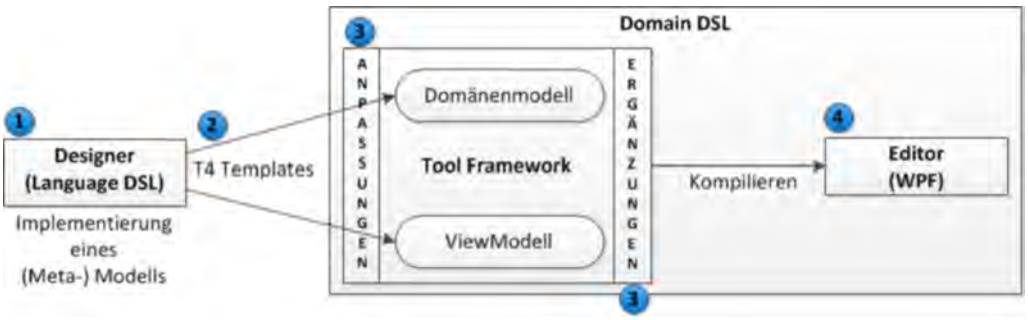


Abbildung 5.17: Vorgehensweise bei der Implementierung einer DSL mit PDE

1. Erstellung des Domänenmetamodells der Domain DSL im Editor der Language DSL. Dabei werden
   - das Metamodell der Sprache basierend auf den Strukturen der Language DSL
   - das Serialisierungsmodell, das Strukturen zum Speichern und Laden des Metamodells der Sprache zu Verfügung stellt
   - das Diagram-Modell, das für die Darstellung der Strukturen des Metamodells der Sprache genutzt wird

   implementiert sowie konfiguriert.

   Das Ergebnis dieses ersten Schrittes, sprich das Domänenmetamodell, liegt schließlich selbst in XML-Form vor.

2. Generierung des Quellcodes basierend auf dem Domänenmetamodell der Domain DSL. Dabei wird generiert:
   - Domänenmodell: Abbildung des Domänenmetamodells in Quellcode.
   - Serialisierungsmodell: Klassen zur Serialisierung des Domänenmodells.
   - Validierungsmodell: Klassen zur Validierung des Domänenmodells.
   - View Modelle: Standard- sowie benutzerspezifische Sichten auf das Domänenmodell.

   Die so generierten Teilmodelle werden automatisch in das *ToolFramework* integriert, was als Basis der Domain DSL vorliegt.
3. Die generierten Klassen sind partiell und können angepasst werden, indem entweder bestehende Funktionalität überschrieben wird oder um weitere Methoden und Eigenschaften erweitert wird. Für das Überschreiben müssen die Klassen vor der Generierung gegebenenfalls mit *GeneratesDoubleDerived=true* gekennzeichnet werden.

   Die Domain DSL kann nun um spezifische Validierungsmethoden erweitert werden, die für die entwickelte Sprache notwendig sind.

   Ferner werden in diesem Schritt auch Xaml-Templates für die konkrete grafische Darstellung vom Anwender implementiert, so dass sie für die Darstellung von View Modellen verwendet werden können.
4. Das Ergebnis liegt nun vor: Eine domänenspezifische Sprache mit einem WPF Editor zur Verwaltung von Instanzen ihrer Modelle.
5. Die fertige Sprache kann über Plugins um zusätzliche Funktionalitäten erweitert werden.

5.1.6 Spezifika für das V-Modell XT

Die neueste Version der Microsoft DSL-Tools erlaubt die Erweiterung des DSL-Tools Designers (beispielsweise um neue Domäneneigenschaften). Diese Funktionalität war zu Beginn dieser Arbeit so noch nicht vorhanden, allerdings hätten wir uns ohnehin für die Entwicklung der Language DSL entscheiden müssen, da diese bereits Spezifika für die Entwicklung einer Sprache für das V-Modell enthält. Diese spezifischen Eigenschaften der Language DSL wollen wir in diesem Abschnitt ausschnittsweise kurz vorstellen, es sei aber angemerkt, dass sich die Language DSL sowie das Gesamtkonzept von PDE hierdurch nicht nur auf eine Sprache für das V-Modell beziehungsweise ähnliche Sprachen beschränkt, vielmehr erlauben wir hiermit für die V-Modell Sprache eine einfachere Implementierung beziehungsweise machen eine solche erst sinnvoll möglich.

Serialisierungsmodell. Eines der wesentlichen Probleme des PDS Designers (siehe Abschnitt 1.2.2) war die Serialisierung in das bereits vorhandene Format des V-Modells (XML-Format). Hierdurch wurde eine in Quellcode geschriebene Serialisierungslösung notwendig, die zum einen aufwendig und zum anderen unflexibel gegenüber Änderungen war. Zusätzlich ist eine derartige Lösung immer fehleranfällig. Für das V-Modell XT musste insofern eine bessere Lösung konzipiert werden, denn wenn eine direkte Quellcodelösung bereits problematisch für die Projektdurchführungsstrategien war, so ist sie nicht akzeptabel für das deutlich umfangreichere Gesamtmodell. Deshalb haben wir das Serialisierungsmodell in der Language DSL so definiert, dass das Laden und Speichern von Instanzen des Domänenmodells in der Domain DSL in das derzeit eingesetzte V-Modell XT Format möglich ist. Hierzu müssen nur die jeweiligen Serialisierungsnamen sowie die Reihenfolge der Elemente im Editor der Language DSL spezifiziert werden.

Eigenschaftendefinition. Beim Domänenmetamodell wurden wichtige Eigenschaften wie *IsRequired* festgelegt, die notwendig sind für eine Implementierung des V-Modells. Bei den DSL-Tools ist beispielsweise die genannte Eigenschaft nicht vorhanden, so dass die Validierung diesbezüglich auch manuell geschrieben werden muss, was recht viel Aufwand darstellt.

Grafisches Modell. Das Diagrammmodell der Language DSL und der Domain DSL wurden so entwickelt, dass beliebige erweiterte Darstellungen möglich sind, so dass die komplexen Vorgänge des V-Modells einerseits für den Prozessingenieur und andererseits später in einer erweiterten

Form für den Anwender dargestellt werden können. Hierdurch soll das Verständnis auf den jeweils notwendigen Ebenen ermöglicht und der Umgang mit dem V-Modell erleichtert werden.

Die Abhängigkeitssicht wurde speziell im Hinblick auf die V-Modell DSL entwickelt, da sie dort auch notwendig ist.

Als Zieleditor der Sprache liegt eine Applikation vor, die von weiteren Programmen unabhängig ist. Das ist, verglichen mit den DSL-Tools, eine deutliche Verbesserung für die V-Modell Sprache, schließlich wird hier Visual Studio weder vorausgesetzt noch erwünscht. Das Visual Studio ist für die Prozessingenieure schlicht zu umfangreich und zu komplex, bietet zudem auch keine erweiterte Funktionalität, die im Rahmen einer V-Modell Sprache sinnvoll wäre. Somit ist eine eigenständige Applikation wesentlich vorteilhafter.

Plugins. PDE verfügt über ein Plugin-System, so dass die Sprache für das V-Modell auch als Integrationsplattform für die Werkzeuge des V-Modells dienen kann.

5.2 Anwendung auf das V-Modell XT

In Kapitel 4 haben wir das V-Modell XT Metamodell vorgestellt und im Detail die Projektdurchführungsstrategien (PDS) betrachtet. Zur Thematisierung der Erstellung einer V-Modell XT DSL basierend auf PDE, wollen wir die Umsetzung der PDSen in das Domänenmetamodell der Domain DSL zeigen. Das soll ausschnittsweise verdeutlichen, wie die DSL für das V-Modell implementiert wurde. Eine derartige Abbildung haben wir bereits schon beim PDS-Designer [WKK09b] thematisiert, so dass wir uns darauf stützen können. Allerdings haben wir in dieser Arbeit einige Verbesserungen eingeführt, die sich aus den Erkenntnissen beim PDS-Designer ergeben haben.

Für die tatsächliche Abbildung der XML-Strukturen des V-Modell XT Metamodells in das Domänenmetamodell der Domain DSL betrachten wir zunächst die Abbildung der Basis-Attribute (siehe Abbildung 5.18), die jeder Domänenklasse fortan zugrunde liegen. Dazu legen wir zunächst die abstrakte Domänenklasse *BaseElement* an und berücksichtigen die Attribute wie folgt:

- *id* liegt bei jeder Klasse und Beziehung des Domänenmetamodells automatisch vor.
- *version* wird als *Intern_Version* berücksichtigt.
- *consistent_to_version* ist mittels *Intern_ConsistentToVersion* abgebildet.
- *refers_to_id* liegt als *Intern_RefersToId* vor.

Abbildung 5.18: Abbildung der Basis-Attribute auf Domänenklassen

Der Definition von Projektdurchführungsstrategien dienen im Metamodell des V-Modells die so genannten Ablaufbausteine. Bevor wir diese im Detail betrachten, müssen wir die Integration derselben im Domänenmetamodell thematisieren (siehe Abbildung 5.19). Zunächst liegt uns hier der Hauptknoten, die Domänenklasse *V-Modell*, vor. Diese enthält über eine Integrationsbeziehung die Klasse *VModellvariante*, die ihrerseits die Klasse *Ablaufbausteine* integriert. Genau bei der letzten Klasse werden nun die einzelnen Ablaufbausteine, abgebildet über die gleichnamige Domänenklasse, mittels einer Integrationsbeziehung hinterlegt.

Ablaufbaustein. Das Hauptelement zur Definition von PDSen ist durch einen Ablaufbaustein gegeben, dessen Implementierung basierend auf den Strukturen des Metamodells der Abbildung 5.20 entnommen werden kann.

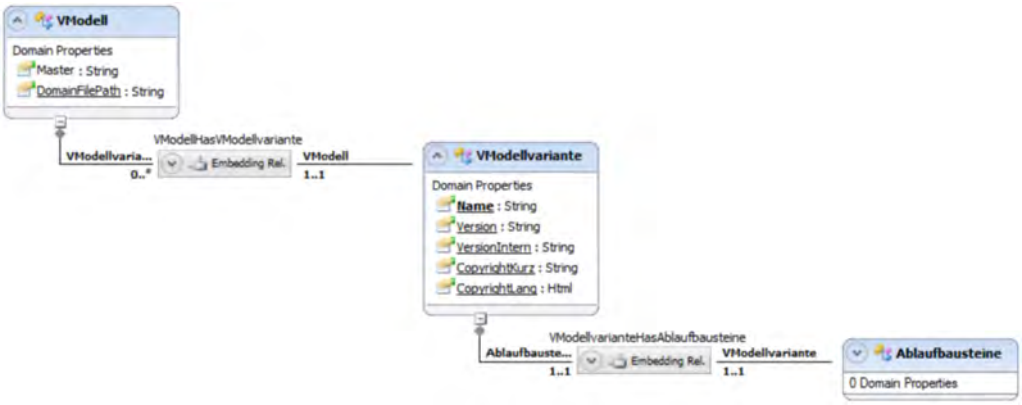


Abbildung 5.19: Integration der Ablaufbausteine im Domänenmetamodell

Jeder Ablaufbaustein im Metamodell hat die Eigenschaften Name und Beschreibung sowie eine Reihe von Elementen, die es integriert. Sie werden alle im Domänenmetamodell über gleichnamige Domäneneigenschaften und -klassen realisiert. Die Referenz von genau einer Ablaufbausteinspezifikation ist mittels der Referenzbeziehung *AblaufbausteinReferencesAblaufbausteinspezifikation* hinterlegt. Insofern wurde hier zur Abbildung des Ablaufbausteins eine zum Metamodell des V-Modell XT beinahe identische Struktur gewählt.

Ein Ablaufbaustein verfügt noch über eine Darstellungsdefinition mittels *AblaufbausteinShape*. Interessant ist hier, dass diese Shape als die Hauptdarstellung für die modularen Ablaufbausteine dienen soll und folglich alle Kinder (genannt *Ablaufpunkte*) fortan innerhalb seiner Darstellung zusammenfasst. Das bedeutet, dass die Shapes der Ablaufpunkte selbst Kindelemente des *AblaufbausteinShapes* sind.

Als nächstes wollen wir Übergänge zwischen den einzelnen Ablaufpunkten betrachten, schließlich werden erst durch diese Elemente die eigentlichen Prozesse innerhalb von Ablaufbausteinen definiert. Vorweg sei noch erwähnt, dass die Ablaufpunkte selbst zwar über jeweils gleichnamige Klassen im Domänenmetamodell abgebildet werden, sie sind allerdings von einer weiteren Basisklasse abgeleitet[20], die sich *Ablaufpunkt* nennt[21]. Diese stellt eine Abstraktion eines Ablaufpunkts dar und erlaubt folglich die Definition von Übergängen.

Übergänge (siehe Abbildung 5.21) werden als Domänenklassen beim zugehörigen Ablaufbaustein integriert. Sie selbst weisen genau zwei Beziehungen auf, um einen gerichteten Übergang zwischen zwei Ablaufpunkten zu definieren:

- *UebergangReferencesAblaufpunktQuelle* weist die Quelle eines Übergangs aus.
- *UebergangReferencesAblaufpunktZiel* definiert das Ziel eines Übergangs.

Insofern muss ein Übergang im Domänenmodell immer über zwei Beziehungen verfügen, die die Quelle und das Ziel definieren.

Ablaufbaustein- und Ablaufentscheidungspunkte. Ablaufbaustein- und Ablaufentscheidungspunkte stellen Ablaufpunkte mit teilweise gleichartigen Strukturen dar. Insofern sind diese von einer weiteren Basisklasse *AblaufDokumentationpunkt* abgeleitet, die ihrerseits *Ablaufpunkt* als Basisklasse aufweist, so dass diese Elemente auch bei Übergängen mitberücksichtig werden können (siehe Abbildung 5.22).

Diese Ablaufpunkte implementieren ferner das Dokumentationskonzept (für nähere Informationen siehe [TK09]), welches im Domänenmetamodell über die abstrakte klasse *AblaufDokumentationpunkt* verwirklicht wird. Dabei werden die Referenzen *DokFolgtNachAblaufpunktRef* aus dem Metamodell durch eine entsprechende Referenzbeziehung zwischen zwei *AblaufDokumentationpunkten* dargestellt.

Die Referenzen zwischen einem Ablaufbausteinpunkt und einer Ablaufbausteinspezifikation sowie zwischen einem Ablaufentscheidungspunkt und einem Entscheidungspunkt werden mithilfe

20 Ausgenommen hier sind die *Split* und die *Join* Domänenklassen.
21 *Ablaufpunkt* ist selbst wieder von *BaseElement* abgeleitet.

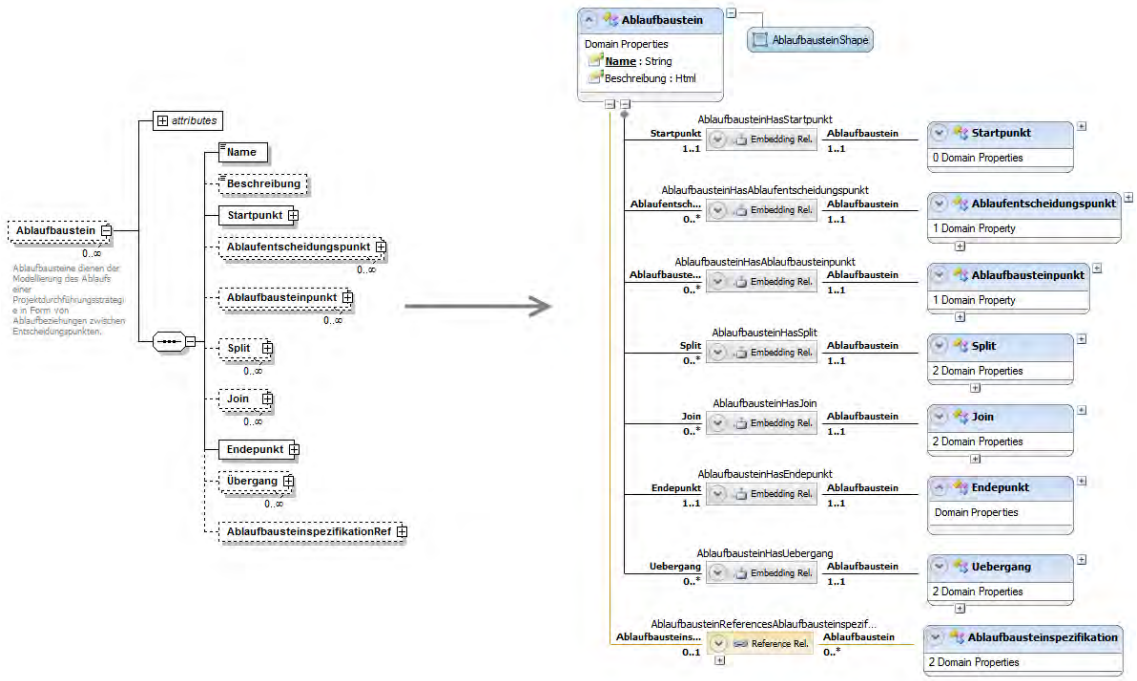


Abbildung 5.20: Implementierung des Ablaufbausteins im Domänenmetamodell

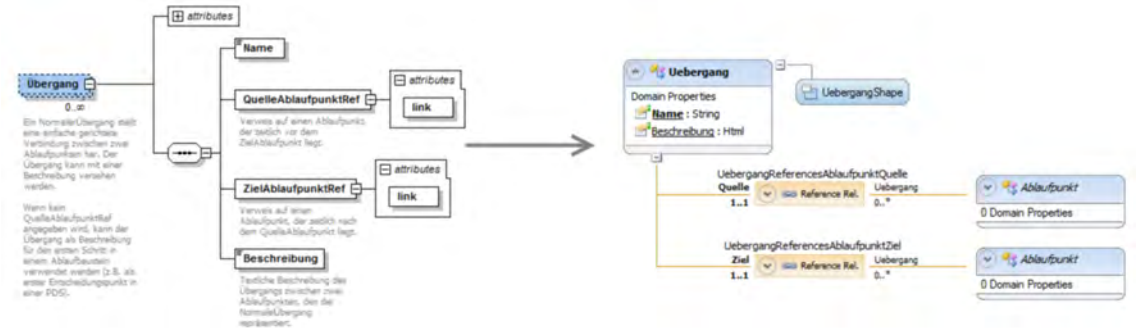


Abbildung 5.21: Implementierung von Übergängen im Domänenmetamodell

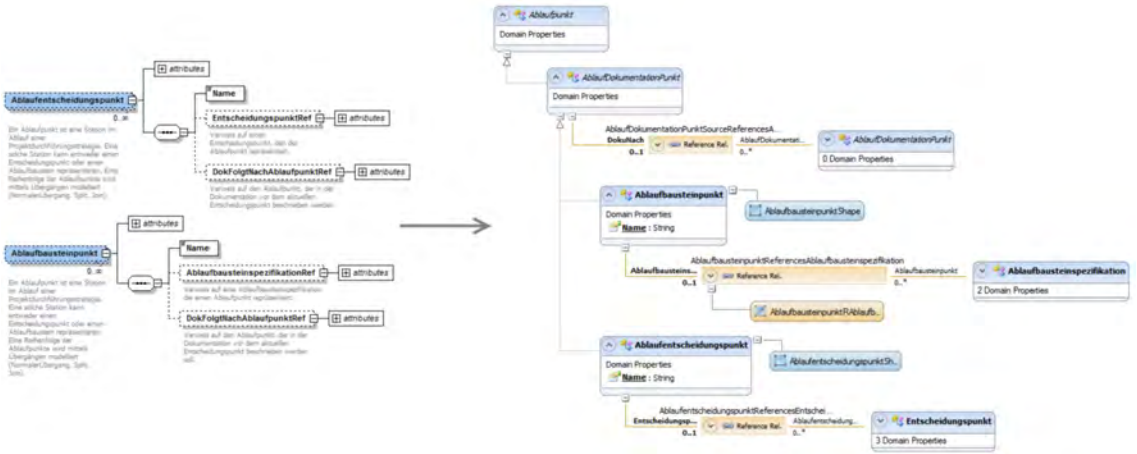


Abbildung 5.22: Implementierung der Ablaufbaustein- und Ablaufentscheidungspunkte im Domänenmetamodell

der Referenzbeziehungen *AblaufbausteinPunktReferencesAblaufbausteinspezifikation* beziehungsweise *AblaufentscheidungspunktReferencesEntscheidungspunkt* implementiert.

Splits. Splits werden über die Domänenklasse Split implementiert (siehe Abbildung 5.23). Der SplitEingang und die SplitAusgänge werden über gleichnamige Domänenklassen abgebildet und sind ferner von der Basisklasse *Ablaufpunkt* abgeleitet, so dass sie bei Übergängen mitberücksichtigt werden können.

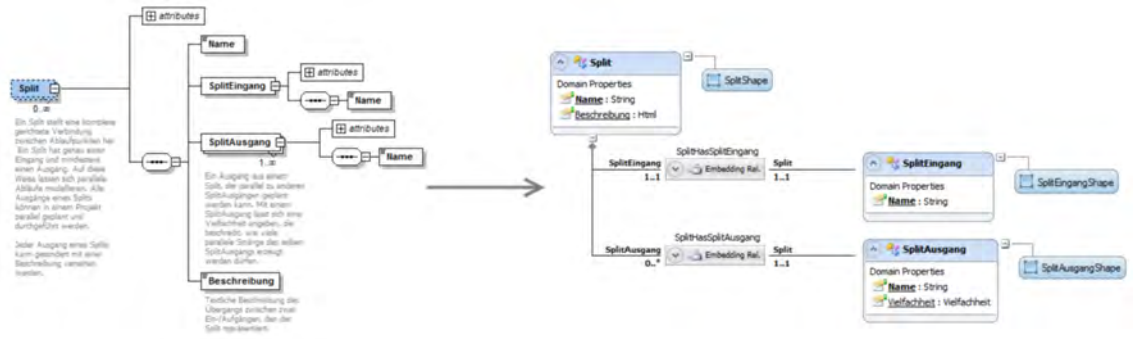


Abbildung 5.23: Implementierung des Splits im Domänenmetamodell

Beim *SplitAusgang* lässt sich zudem noch die Kardinalität festlegen, die für die Anzahl der möglichen startenden parallelen Abläufe steht. Möglich hier sind 0..1, 1..1, 0..*, 1..* und 2..*. Die Idee des *SplitEingangs* beziehungsweise der *SplitAusgänge* an dieser Stelle ist, dieselben später über Shapes darzustellen, die als Kindelemente des *SplitShapes*, der einen Split darstellt, nur auf dessen Rand bewegt werden können und somit eine Art *Port* zum und aus dem Split darstellen. Damit lassen sich insbesondere die Kardinalitäten der *SplitAusgänge* grafisch anzeigen.

Joins werden auf eine ähnliche Art und Weise wie Splits abgebildet, so dass wir diese hier nicht weiter detailliert vorstellen wollen.

Gesamtabbildung des Ablaufbausteins. Zusammenfassend ist in Abbildung 5.24 die Gesamtumsetzung eines Ablaufbausteins aus dem Metamodell des V-Modell XT in das Domänenmetamodell der zugehörigen DSL gegeben. Für die V-Modell Sprache wurde selbstverständlich das ganze Metamodell auf diese Art und Weise implementiert, allerdings würde die Erklärung dieser Gesamtabbildung den Rahmen der Arbeit sprengen, zumal diese in einem derartigen Detail auch nicht weiter interessant ist. Wichtig ist nur, das Konzept dieser Abbildung zu verstehen, so dass wir nicht weiter auf die Umsetzung der Strukturelemente des Metamodells eingehen wollen.

5.2.1 Spezifische Anpassungen für das V-Modell XT

Soweit haben wir uns mit der Implementierung des Domänenmetamodells beschäftigt. Dieses liegt uns nun vor und nach der automatischen Quellcodegenerierung wollen wir spezifische Anpassungen sowie Erweiterungen vornehmen, die für die Sprache des V-Modell XT notwendig sind. Wir beschränken uns hierbei ausschnittsweise auf vier Beispiele:

- Erweiterung: HTML Editor
- Einschränkungen Eigenschaftsfenster bezüglich der Beziehungsmodellierung
- Einschränkungen bei der grafischen Modellierung von Beziehungen
- Einfaches Plugin zur Anzeige des serialisierten Modells

HTML Editor. Das V-Modell integriert die zugehörige Dokumentation im Modell (die Dokumentation wird genauer genommen basierend auf der V-Modell Instanz exportiert). Zur Definition der Dokumentation müssen unter anderem Beschreibungen bei den entsprechenden Elementen vorgelegt werden, die auf HTML basieren. Das verwendete HTML selbst ist allerdings auf einige gewisse Elemente beschränkt. Somit versteht das V-Modell diesbezüglich auch nur diese

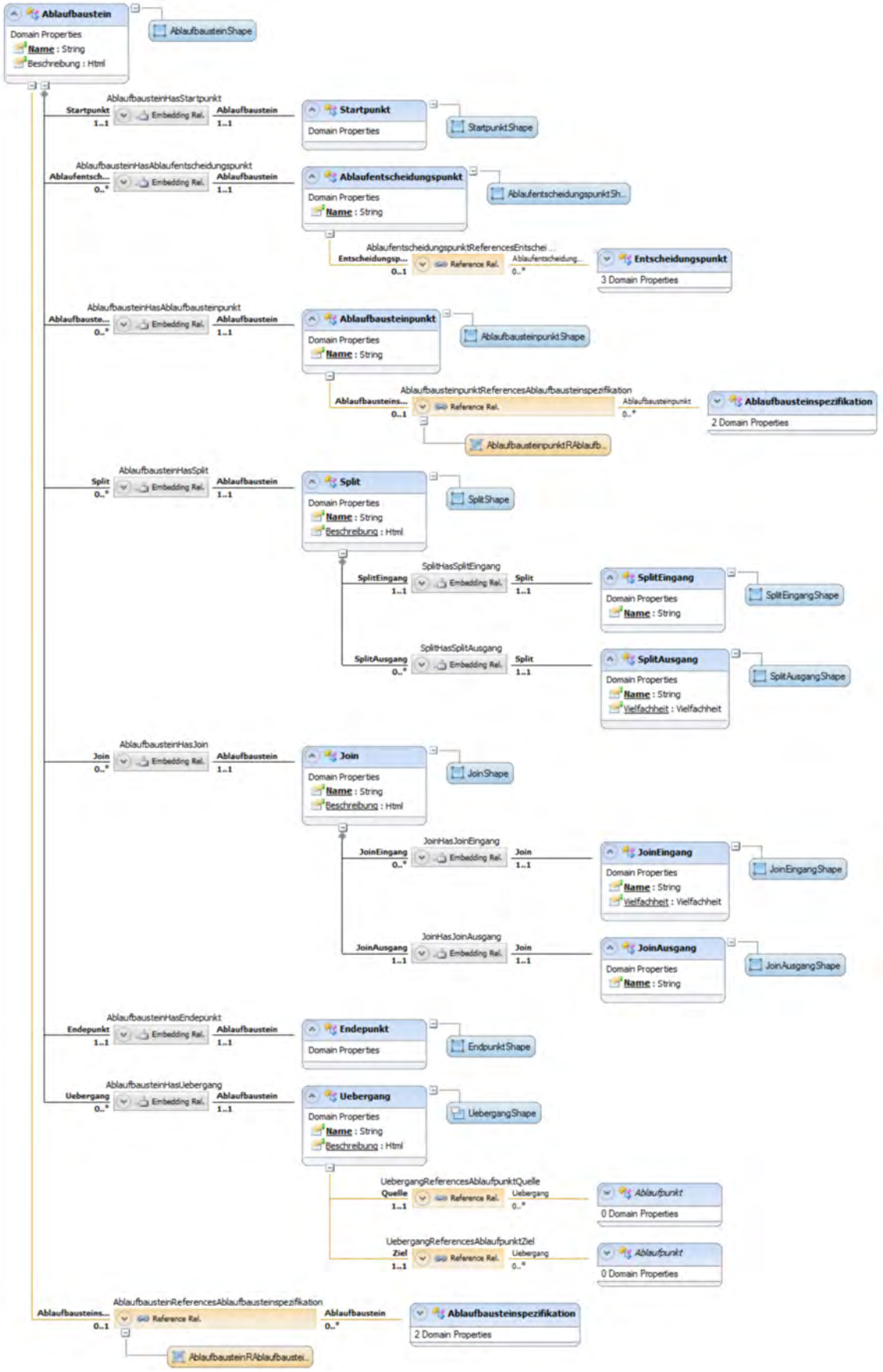


Abbildung 5.24: Gesamtabbildung des Ablaufbausteins im Domänenmetamodell

gewissen Elemente. Um die Definition von HTML im V-Modell zu unterstützen, muss für die zugehörige DSL ein entsprechender Editor entwickelt werden, der selbst die folgenden Anforderungen erfüllt:

- HTML Editor verfügt über zwei Bearbeitungsmodi:
  - Design-Modus: HTML wird *geparsed* dargestellt, so dass eine Vorschau auf die Darstellung in der Dokumentation gegeben ist.
  - Editor-Modus: HTML wird als Quellcode dargestellt und kann diesbezüglich auch angepasst werden.
- Anlegen und Modifizieren von bekannten HTML-Elementen ist im Design-Modus mittels einer Menu-Auswahl möglich.
- Validierung des festgelegten HTMLs wird direkt nach einer Bearbeitung gestartet und liefert Fehler bezüglich unbekannter Elemente.

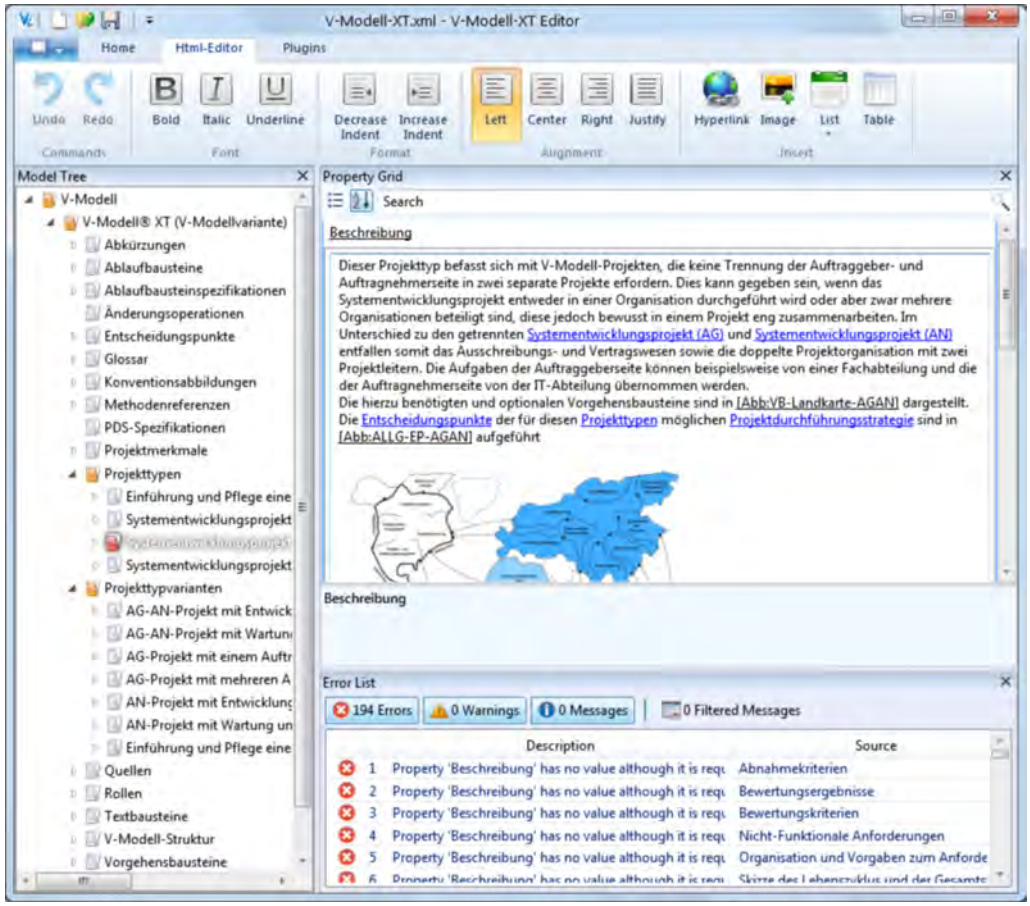


Abbildung 5.25: HTML Editor in der DSL für das V-Modell XT

Vor der eigentlichen Implementierung des HTML Editors müssen wir uns mit dem Teil der HTML-Elemente beschäftigen, die für den Editor relevant sind (siehe Tabelle 5.1). Diese sind folglich im Editor zu implementieren und für die Darstellung im Design-Modus zu einem so genannten *FlowDocument* [MSD] zu konvertieren, so dass wir diesen *FlowDocument* für die Design-Darstellung nutzen können. Zur Erhaltung des HTML-Codes ist eine Rückkonvertierung von *FlowDocument* zu HTML notwendig.

Das Ergebnis des HTML Editors lässt sich nun Abbildung 5.25 entnehmen. Dort ist auch zu sehen, dass der implementierte HTML Editor in das Eigenschaftenfester integriert wurde und zudem über ein Bearbeitungsmenu verfügt, das selbst kontextsensitiv angezeigt wird (sobald der HTML Editor aktiv ist).

Einschränkungen beim Eigenschaftsfenster. Das Eigenschaftsfenster wurde bei PDE so konzipiert, dass jeder Editor bezüglich seiner Eigenschaften angepasst werden kann. Das wollen wir

Elementname	Unterstütze Attribute	Beschreibung
p	align, style=margin	Darstellung von Paragraphen
a	href	Abbildung von Hyperlinks
b		Fett formatierter Text
u		Unterstrichener Text
i		Kursiver Text
img	align, src, alt, id	Darstellung von Bildren
ul, ol, li		Definition von Listen
table, tr, td	border für table	Einfache Tabellen

Tabelle 5.1: Liste unterstützter HTML-Elemente

anhand der Editoren für das Modellelement *Übergang* (Projektdurchführungsstrategien) aufzeigen, indem wir die Auswahl der Quelle und des Ziels, die jeweils zur Verfügung stehen, einschränken. Diese Einschränkung ist übrigens nicht nur sinnvoll, sondern sogar notwendig, denn Übergänge können in den modularen Ablaufbausteinen nur innerhalb derselben definiert werden, so dass die Auswahl eines Ablaufelements aus einem anderen Ablaufbaustein im Grunde genommen ein schwerwiegender Fehler ist. Im aktuellen Editor des V-Modells (siehe Abschnitt 1.2.2) ist diese Einschränkung übrigens nicht vorhanden, so dass dort genau dieser Fehler leicht gemacht werden kann. Unabhängig von der fehlerhaften Modellierung von Übergängen wird die Notwendigkeit dieser Einschränkung aufgrund der Benutzerfreundlichkeit deutlich, schließlich ist das Heraussuchen der richtigen Ablaufpunkte bei einer größeren Anzahl derselben sehr zeitaufwändig.

Zur Definition der genannten Einschränkung müssen wir Anpassungen beim View Modell treffen, das für das Bearbeiten von Eigenschaften des Modellelements *Übergang* generiert wird. Dieses nennt sich *PropertyGridUebergangViewModel* und enthält die automatisch generierten Methoden

- *CreateEditorViewModelForQuelleRole*
- *CreateEditorViewModelForZielRole*

für die Erstellung der Editoren zur Auswahl des Quell- beziehungsweise des Zielelements. Diese beiden Methoden müssen überschrieben werden, so dass die von ihnen gelieferten Editoren unseren Anforderungen entsprechen. In Listing 5.10 ist das für die erste Methode dargestellt.

Listing 5.10: Anpassung der Editorerstellung für die Auswahl des Quellelements eines Übergangs

```
public partial class PropertyGridUebergangViewModel
{
    /// <summary>
    /// Create property grid editor view model for Uebergang.Quelle
    /// </summary>
    protected override void CreateEditorViewModelForQuelleRole(List<
        PropertyGridEditorViewModel> collection)
    {
        UnaryRoleEditorViewModel editor = new UnaryRoleEditorViewModel(this.
            ViewModelStore, this.Store.DomainDataDirectory.GetDomainRelationship(
            UebergangReferencesAblaufpunktQuelle.DomainClassId),
            UebergangReferencesAblaufpunktQuelle.UebergangDomainRoleId);

        // editor properties copied from genereated code ...
        // ...

        // change: supply a specific element provider
        editor.DefaultValuesProvider = new LookupListEditorDefaultValuesProvider<object
            >(GetDefaultElements);

        // add to editors collection
        collection.Add(editor);
    }

    /// <summary>
    /// Provides specific elements that can be chosen through a given lookup view model.
    /// </summary>
    private List<object> GetDefaultElements(LookupListEditorViewModel viewModel)
    {
```

```
        UnaryRoleEditorViewModel editorVM = viewModel as UnaryRoleEditorViewModel;

        bool bSource = false;
        if (editorVM.SourceRoleId == global::Tum.VModellXT.
            UebergangReferencesAblaufpunktQuelle.UebergangDomainRoleId)
            bSource = true;

        List<object> elements = new List<object>();
        Uebergang uebergang = this.Element as Uebergang;

        elements.AddRange(uebergang.Ablaufbaustein.Ablaufbausteinpunkt);
        elements.AddRange(uebergang.Ablaufbaustein.Ablaufentscheidungspunkt);

        if (bSource)
        {
            if (uebergang.Ablaufbaustein.Startpunkt != null)
                elements.Add(uebergang.Ablaufbaustein.Startpunkt);
        }
        else
        {
            if (uebergang.Ablaufbaustein.Endepunkt != null)
                elements.Add(uebergang.Ablaufbaustein.Endepunkt);
        }

        foreach (Split split in uebergang.Ablaufbaustein.Split)
        {
            if (bSource)
                elements.AddRange(split.SplitAusgang);
            else
                if (split.SplitEingang != null)
                    elements.Add(split.SplitEingang);
        }

        foreach (Join join in uebergang.Ablaufbaustein.Join)
        {
            if (bSource && join.JoinAusgang != null)
                elements.Add(join.JoinAusgang);
            else
                elements.AddRange(join.JoinEingang);
        }
        return elements;
    }
}
```

Die Methode *GetDefaultElements* ist dabei dafür verantwortlich, dass nur diejenigen Elemente zur Auswahl gestellt werden, die tatsächlich auch ausgewählt werden dürfen.

Einschränkungen bei der grafischen Modellierung. Die grafische Modellierungsweise im Rahmen der diagramartigen Darstellung erlaubt es Beziehungen zwischen Elementen auf eine einfache Art und Weise definieren zu können. Allerdings kann es auch hier sinnvoll sein, gewisse Elemente dabei auszuschließen und Beziehungen zwischen diesen Elementen nicht zu erlauben. Im V-Modell XT ist das beispielsweise wieder bei den Übergängen der Fall, wo dieselben nur zwischen Ablaufelementen eines Ablaufbaustein, nie zwischen unterschiedlichen Elementen unterschiedlicher Ablaufbausteine festgelegt werden dürfen. Hierzu müssen Anpassungen in der grafischen Modellierungsweise getroffen werden, so dass der Anwender keine derartig fehlerhaften Übergänge anlegen kann.

Zur Definition der genannten Einschränkung müssen wir die generierte Methode *CanCreateMappingRelationshipShapeUebergangShape* in der Klasse *VModellXTDesignerDiagramSurfaceViewModel* überschreiben und entsprechend Listing 5.11 anpassen.

Listing 5.11: Anpassung der Diagramoberfläche für die Einschränkung der Beziehungserstellung hinsichtlich von Übergängen

```
/// <summary>
/// This class represents the default diagram surface view model of the V-Modell-XT
    Editor.
/// </summary>
public partial class VModellXTDesignerDiagramSurfaceViewModel
{
    /// <summary>
```

```
    /// Verifies if a mapping relationship of type UebergangShape can be created based
        on the given data.
    /// </summary>
    /// <param name="info">Info specifying the relationship to create.</param>
    /// <returns>True if a relationship can be created. False otherwise.</returns>
    protected override bool CanCreateMappingRelationshipShapeUebergangShape(
        ViewModelRelationshipCreationInfo info)
    {
        Ablaufpunkt source = info.Source.Element as Ablaufpunkt;
        Ablaufpunkt target = info.Target.Element as Ablaufpunkt;

        if (source == null || target == null)
            return false;

        if (target is Startpunkt || target is SplitAusgang || target is JoinAusgang)
            return false;

        if (source is Endepunkt || source is SplitEingang || source is JoinEingang)
            return false;

        if (VModellXTElementParentProvider.Instance.GetEmbeddingParent(source,
            Ablaufbaustein.DomainClassId) != VModellXTElementParentProvider.Instance.
            GetEmbeddingParent(target, Ablaufbaustein.DomainClassId))
            return false;

        return base.CanCreateMappingRelationshipShapeUebergangShape(info);
    }
}
```

Plugin zur Anzeige des serialisierten Modells. Zur Vorstellung der Erweiterungsmöglichkeit der Domain DSL mittels Plugins wollen wir ein sehr einfaches Plugin beschreiben, das nur das aktuell geladen Domänenmodell in seiner serialisierten Form in einem separaten Fenster anzeigt. Die Definition und Einbindung von Plugins für die Domain DSL baut auf dem *Managed Extensibility Framework* (MEF) [MEF10] auf, so dass jedes Plugin als Bibliothek vorliegen kann und zur Berücksichtigung nur die *IPlugin*-Schnittstelle aus dem *ToolFramework* sowie das *Export*-Attribut von MEF implementieren muss.

Die *IPlugin*-Schnittstelle weist, wie in Listing 5.12 dargestellt, die Methode *Execute* auf, die von der Domain DSL aufgerufen wird, sobald die Funktionalität des Plugins aus dem Hauptmenu angefordert wird. Der Methode *Execute* wird dabei die so genannte *ModelData*-Klasse übergeben, die neben dem Zugriff auf das Domänenmodell zusätzlich Funktionalitäten bereitstellt, um das Domänenmodell auch beispielsweise in serialisierten Form als Zeichenkette zurückzugeben. In diesem einfachen Plugin wollen wir nur das aktuell geladen Modell in seiner serialisierten Form anzeigen, was mittels der Darstellung der Zeichenkette *modelData.SerializedModel* erreicht wird (siehe Abbildung 5.26).

Listing 5.12: Einfaches Plugin für die Domain DSL

```
[Export(typeof(IPlugin))]
public class ExamplePlugin : IPlugin
{
    /// <summary>
    /// Executes the plugins function.
    /// </summary>
    /// <param name="modelData">
    /// Class representing the currently loaded domain model instance.
    /// modelData.RootElement is the root element of the domain model (in this case its
        the VModell class).
    /// </param>
    public void Execute(ModelData modelData)
    {
        if (modelData.RootElement != null)
        {
            // display modelData.SerializedModel
            ExampleWindow wnd = new ExampleWindow();
            wnd.Init(modelData);
            wnd.ShowDialog();
        }
    }
}
```

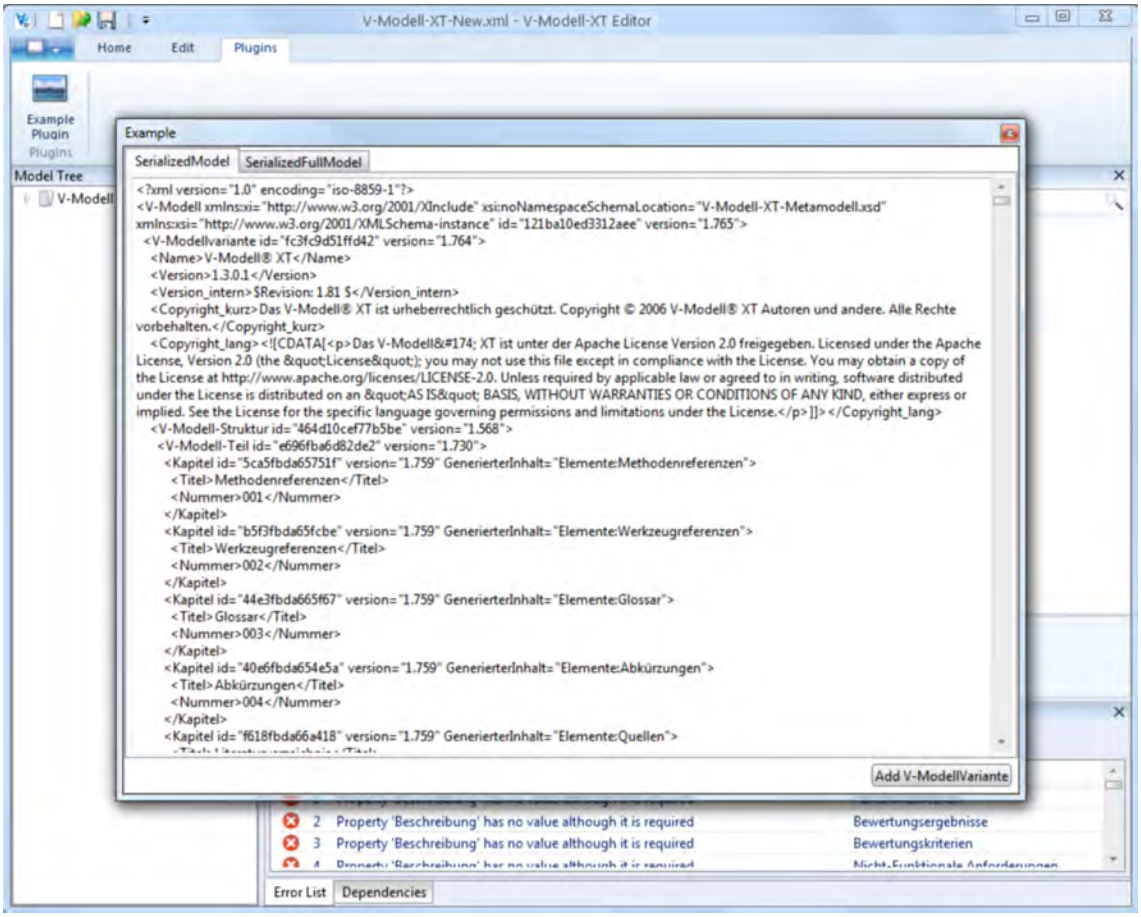


Abbildung 5.26: Beispielplugin in der V-Modell XT DSL

5.2.2 Validierung

Validierung stellt eine der zentralen Funktionalitäten in der Domain DSL dar und ist insbesondere für das V-Modell wichtig, schließlich wollen wir mit der Sprache eine Verbesserung der Bearbeitungsqualität hinsichtlich Fehlervorbeugung aber auch hinsichtlich Fehlerfindung für den Prozessingenieur erreichen. Gerade für den zweiten Punkt ist die Validierung ausschlaggebend. In diesem Abschnitt wollen wir zusammenfassen, was genau automatisch generiert wird und was mittels Quellcode benutzerspezifisch vorgelegt werden muss. Wir machen das wieder beispielshaft anhand der Projektdurchführungsstrategien (siehe Abschnitt 4.1.1).

Automatische Validierung. Die automatische Validierung überprüft bei den Projektdurchführungsstrategien ob

- beim *Ablaufbaustein* der Name gesetzt ist.
- beim *Ablaufbaustein* ein *Startpunkt* und ein *Endepunkt* hinterlegt ist.
- ein *Übergang* über einen Namen, eine Beschreibung sowie genau ein Quell- und Zielelement verfügt.
- allen Ablaufpunkten ein Namen zugewiesen wurde.
- beim *Split* ein *SplitAusgang* hinterlegt wurde.
- beim *Join* ein *JoinEingang* vorhanden ist.
- ob jedem *SplitAusgang* sowie jedem *JoinEingang* eine Vielfachheit zugewiesen wurde.

Diese automatische Validierung ist selbstverständlich für die Projektdurchführungsstrategien des V-Modells nicht ausreichend, so dass wir diese entsprechend erweitern müssen.

Benutzerspezifische Validierungserweiterung. Bei der Erweiterung der Validierung wollen wir zunächst überprüfen, ob die einzelnen Ablaufbausteine als Module konsistent sind. Dazu muss überprüft werden, ob

- jeder *Ablaufbaustein* eine *Ablaufbausteinspezifikation* referenziert.
- jeder *Ablaufbausteinpunkt* über eine Referenz zu einer *Ablaufbausteinspezifikation* verfügt.
- bei jedem *Join* zumindest ein *JoinEingang* und bei jedem *Split* zumindest ein *SplitAusgang* hinterlegt wurde.
- ob jeder Ablaufpunkt gemäß seiner Bedeutung über einen Übergang verfügt und folglich ob jeder Pfad durch einen Ablaufbaustein beim *Startpunkt* startet und beim *Endepunkt* endet.
- die *DokuNach*-Eigenschaft richtig gesetzt ist.
- ob jeder *Übergang* zwischen Elementen desselben *Ablaufbausteins* definiert ist, zudem auch der *Übergang* selbst gehört.

Ferner erweitern wir die Validierung der Ablaufbausteine um die Überprüfung der fehlerfreien Definition der Abläufe hinsichtlich der Inklusion von Ablaufbausteinen. So wird beispielsweise festgestellt, ob Ablaufbausteinpunkte eines Ablaufbausteins wieder das Ablaufbaustein selbst integrieren oder ob gegenseitige Integrationen vorhanden sind.

5.3 Ergebnis: V-Modell XT DSL

Das Ergebnis dieser Arbeit ist eine domänenspezifische Sprache für das V-Modell XT. In diesem Abschnitt wollen wir diese Sprache bezüglich der soweit implementierten Funktionalität vorstellen und insbesondere auf den zugehörigen Editor eingehen, der in Lage ist Instanzen des V-Modells zu bearbeiten beziehungsweise zu erstellen. Dabei wollen wir auch die in Abschnitt 4.3 festgelegten Sprachanforderungen bezüglich der Erfüllung in der V-Modell DSL vergleichen.

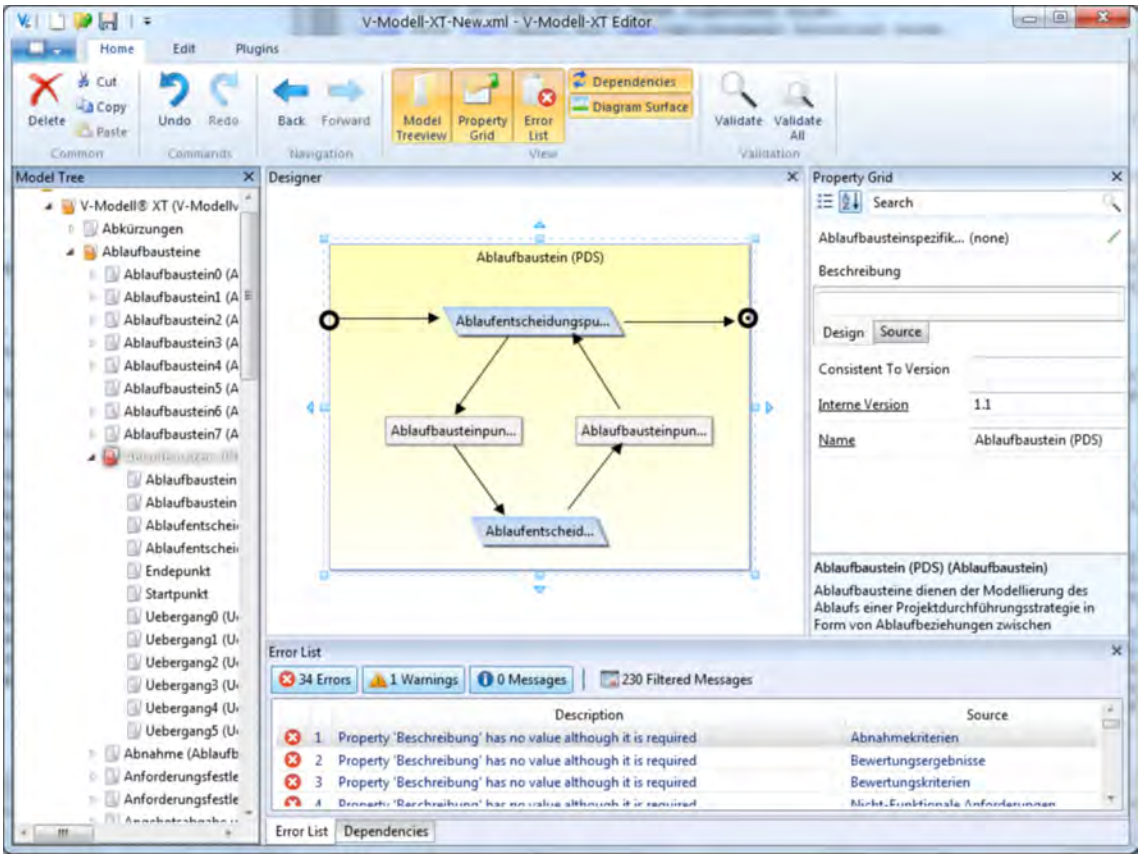


Abbildung 5.27: Editor der V-Modell XT DSL

Sprachanforderungen. Die Sprachanforderungen 4.1 und 4.2 fordern eine Validierung von Modellinstanzen sowie des serialisierten Modells beim Laden. Erstere wird insbesondere mittels der Erweiterung der generierten Validierung um benutzerspezifische Methoden erreicht, während letzteres automatisch im Serialisierungsmodell berücksichtigt wird. Insofern sind beide Anforderungen erfüllt.

Die Anforderung 4.3 fordert 3 einfache Darstellungsbereiche zur Navigation, Eigenschaftenbearbeitung und Fehlerauflistung, die im Editor der Sprache (im *ToolFramework*) festgelegt werden und folgend die geforderten Basisfunktionalitäten bereitstellen. Abbildung 5.27 zeigt diese 3 Darstellungsbereiche im Editor der V-Modell DSL. Die folgenden Anforderungen 4.4 sowie 4.5 legen eine Erweiterbarkeit und Einschränkbarkeit für den Eigenschaftsmodellierungsbereich. Beide Möglichkeiten haben wir bereits einführend in Abschnitt 4.5 vorgestellt. Somit gelten sie auch als erfüllt.

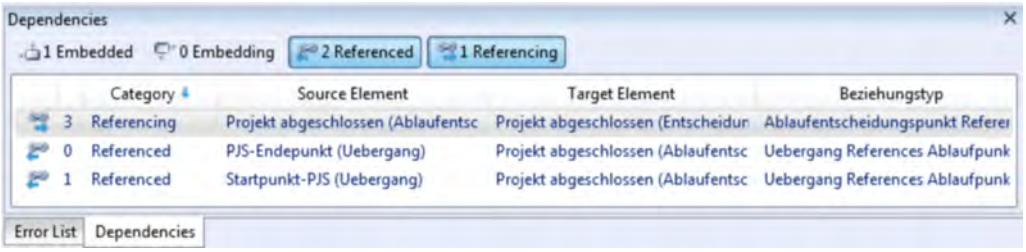


Abbildung 5.28: Abhängigkeitsdarstellung im Editor der V-Modell XT DSL

Die Anforderung 4.6 fordert eine listenartige Darstellung von Abhängigkeiten mit Quelle, Ziel und Typ. Diese haben wir genauso im Rahmen des *ToolFrameworks* vorliegen, wie in Abbildung 5.28 zu sehen ist. Die Abhängigkeitssicht erlaubt es zudem, nach dem entsprechenden Typ zu filtern und somit beispielsweise nur die referenzierenden Abhängigkeiten darzustellen. Ferner ist auch eine Navigation zum Quell- und Zielelement (entsprechend dem Typ der Abhängigkeit) möglich.

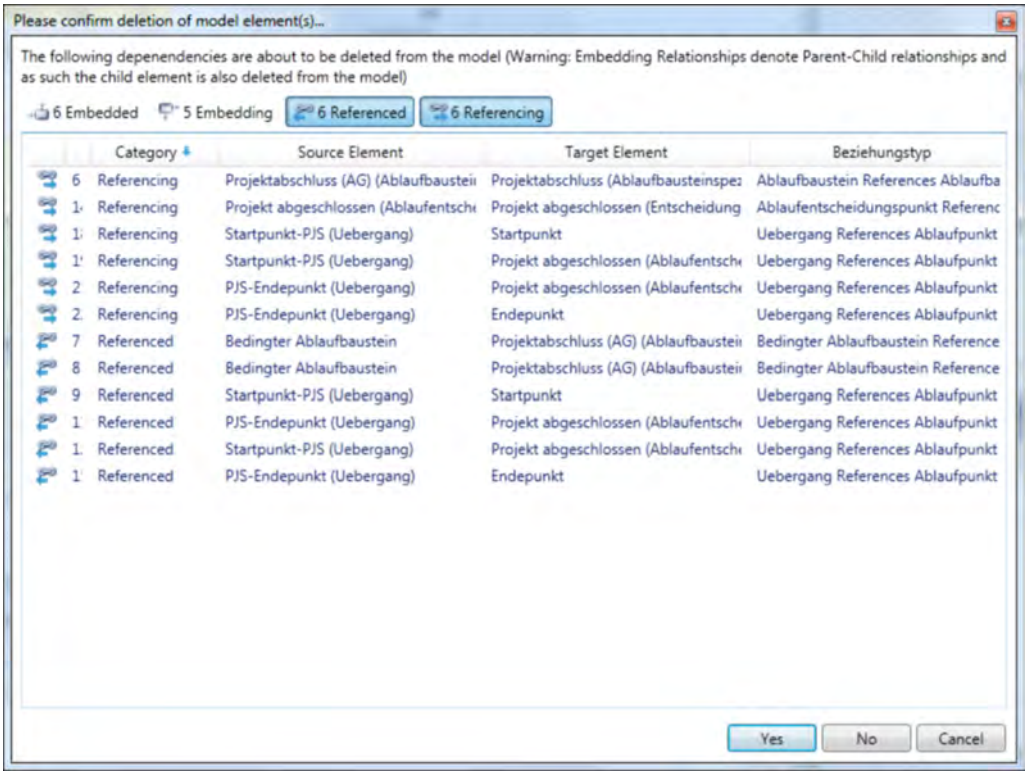


Abbildung 5.29: Bestätigungsaufforderung beim Löschvorgang im Editor der V-Modell XT DSL

Mittels der Anforderung 4.7 wird festgelegt, dass die Sprache über eine Bestätigungsaufforderung beim Löschen verfügen muss, die zugleich aufzeigen soll, was genau (insbesondere bezüglich der Beziehungen) automatisch mitgelöscht wird. Diese Anforderung wird wieder mittels einer entsprechenden Implementierung im Rahmen des *ToolFrameworks* erfüllt, wie in Abbildung 5.29 zu sehen ist.

In Abbildung 5.27 ist neben den Darstellungsbereichen auch die Diagramoberfläche zu sehen, die eine grafische Modellierungsweise für die Projektdurchführungsstrategien, wie in Anforderung 4.8 gefordert, anbietet. Diese Modellierungsweise ist noch in einem frühen Implementierungsstadium und wird später noch erweitert und verbessert. Sie erlaubt allerdings jetzt schon, Ablaufbausteine anzulegen und Prozesse innerhalb der Ablaufbausteine grafisch zu definieren.

Die Sprache für das V-Modell soll, wie bereits mehrfach erwähnt, als Integrationsplattform für vorhandene Tools des V-Modells dienen. Insofern musste, wie in Anforderung 4.9 festgelegt, ein entsprechendes Plugin-System implementiert werden. Die zugehörigen Integrationen über dieses Plugin-System stehen noch aus, werden allerdings demnächst aktuell.

Für die Sprache des V-Modells war eine benutzerfreundliche Modellierungsweise wichtig, so dass mittels der Anforderung 4.10 Standardfeatures wie Copy und Paste gefordert wurden, die auch in der V-Modell XT DSL enthalten sind.

Sprachfunktionalität. Zusammenfassend wollen wir die Funktionalitäten der V-Modell XT Sprache gegenüber dem V-Modell XT Editor stellen und folgend auch festhalten, was unserer Sprache noch fehlt, welche Funktionalitäten eine äquivalente Implementierung erhalten haben und an welchen Stellen bei der DSL gegenüber dem V-Modell XT Editor tatsächlich Vorteile erreicht wurden:

- Modellierungsmöglichkeiten: Zur Verwaltung des V-Modells bietet der V-Modell XT Editor genau zwei Darstellungsbereiche, die bezüglich Funktionalität dem *Model Tree* und dem *Property Window* gleichen. So erlaubt der *Model Tree* das Navigieren, Erstellen oder Löschen von Elementen basierend auf Integrationsbeziehungen, während das *Property Window* der Modifikation von Eigenschaften und Referenzbeziehungen beim selektierten Element dient.

  Verglichen mit dem V-Modell XT Editor ist insbesondere das *Property Window* bezüglich Funktionalität hervorzugeben, erlauben wir hier schließlich die Definition von neuen Editoren für spezifische Typen sowie die Anpassung von vorhandenen Editoren, um beispielsweise bestimmte Sprachvorgaben nicht zu verletzen. Zusätzlich erlaubt das Eigenschaftsfenster die gleichzeitige Modifikation von gleichartigen Eigenschaften unterschiedlicher Elemente.

  In unserer DSL für das V-Modell haben wir noch weitere Modellierungsmöglichkeiten in Form von Sichten. So ist die Standard-Diagrammoberfläche eine solche Sicht, die das Anlegen und Löschen von Elementen und Beziehungen bei den Projektdurchführungsstrategien ermöglicht.
- V-Modell XT Varianten Unterstützung: Das V-Modell XT weist ein Referenzmodell auf, das der Integration vorhandener V-Modelle dient. Man spricht hierbei von V-Modell Varianten. Die V-Modell XT Sprache wurde erweitert um diese Funktionalität bereitzustellen, die auch insbesondere erfordert, dass Elemente aus diesen Varianten als Beziehungspartner auswählbar sind. Der V-Modell XT Editor unterstützt dieses Konzept ebenfalls.
- HTML Unterstützung: Das V-Modell XT benutzt zur Dokumentation von Elementen einen bestimmten Teil von HTML, so dass hierzu (wie oben beschrieben) ein eigener HTML-Editor geschrieben wurde, der sich in das Eigenschaftsfenster integriert. Beim V-Modell XT Editor ist diese Unterstützung auch vorhanden.
- Validierung (Error List): Validierung spielte bei der Definition einer Sprache für das V-Modell eine entscheidende Rolle, schließlich wollten wir hier den Prozessingenieur entsprechend unterstützen. So bieten wir nun neben der automatischen Validierung auch die Möglichkeit einer sprachspezifischen Validierungserweiterung, die wir bei der V-Modell DSL aktuell zur Validierung der Projektdurchführungsstategien nutzen, die ihrerseits die zugehörigen Ablaufbausteine sowie die hierbei definierten Prozesse sinnvoll zur Design-Zeit überprüfen kann.

  Der V-Modell XT Editor bietet kein vergleichbares Validierungskonzept.
- Abhängigkeiten: Die Abhängigkeitssicht in der V-Modell XT DSL erlaubt es unterschiedliche Abhängigkeitstypen bezüglich des selektierten Elements aufzulisten (wird bei der

Löschbestätigung mitverwendet). Zudem können die Beziehungspartner angesteuert werden. Der V-Modell XT Editor listet Abhängigkeiten des selektierten Elements im Eigenschaftsmodellierungsbereich auf, die Darstellung ist allerdings bezüglich Informationsgehalt problematisch, wie wir bereits in Abschnitt 4.3.2 festgestellt haben.

- Löschvorgang: Beim Löschen von Elementen wird der Benutzer in der V-Modell DSL über Beziehungen und Elemente informiert, die automatisch mitgelöscht werden würden und kann diesbezüglich entscheiden, ob der Löschvorgang tatsächlich durchgeführt werden soll oder nicht.

  Der V-Modell XT Editor bietet keine solche Funktionalität.
- Bearbeitungsfunktionalität: Folgende Funktionalitäten werden in der V-Modell XT Sprache zur Verfügung gestellt, die ersten beiden werden auch vom V-Modell XT Editor angeboten:
  - Cut, Copy und Paste
  - Undo und Redo
  - Navigate Forward und Back
- V-Modell zusammenführen: Der V-Modell XT kann V-Modelle zusammenführen. Die zugehörigen Algorithmen liegen in der V-Modell DSL nicht vor, so dass diese Funktionalität nicht bereitgestellt wird.
- Export: Der Export der Dokumentation für das V-Modell ist in der V-Modell XT DSL noch nicht möglich, der V-Modell XT Editor verfügt hingegen über diese Funktionalität.

Gesamtlösung. Abschließend wollen wir die in dieser Arbeit erstellte Gesamtlösung betrachten, die aus PDE und der damit entwickelten V-Modell XT DSL besteht. So haben wir eingangs erwähnt, dass eine allgemeine Werkzeugunterstützung für ein Prozessmodell sich sowohl mit der Prozessmeta- als auch mit der Prozessmodellierung befassen muss. Letzteres haben wir in Form der V-Modell Sprache gegeben und oben auch vorgestellt. Ersteres ist bereits bei PDE im Language DSL Designer verankert. Dieser stellt sicher, dass die definierten Strukturen auch hinsichtlich des Language DSL Metamodells konsistent sind, so dass zumindest diesbezüglich eine Unterstützung vorhanden ist.

6 Fazit und Verallgemeinerung

Im Rahmen dieser Masterarbeit haben wir uns zunächst mit domänenspezifischen Sprachen (siehe Kapitel 2) beschäftigt, um insbesondere die Idee der domänenspezifischen Entwicklung kennenzulernen, deren Kernpunkt in der Integration eines speziellen Modells in ein bereits vorhandenes Framework zu sehen ist. Folgend haben wir uns mit Prozessmodellen auseinandergesetzt (siehe Kapitel 3) und dabei festgestellt, dass ein formales Metamodell gegenüber informellen oder semi-formalen deutliche Vorteile bietet, die sich insbesondere bezüglich der maschinellen Bearbeitbarkeit sowie Prüfbarkeit ausweisen. Das V-Modell XT ist ein solches formales Prozessmodell, so dass wir uns dieses genauer angeschaut und bezüglich möglicher Umsetzungsmöglichkeiten in eine domänenspezifische Sprache diskutiert haben (siehe Kapitel 4). Dort haben wir genau vier solcher Möglichkeiten vorgestellt, wobei nur die letzten beiden für eine sinnvolle Implementierung in Frage kamen, so dass unser tatsächlicher Ansatz auf Umsetzungsmöglichkeit 3 basiert.

Für die Implementierung der Sprache für das V-Modell (siehe Kapitel 5) haben wir zunächst das hierzu erstellte *Process Development Environment* (PDE) vorgestellt, welches selbst auf zwei domänenspezifischen Sprachen basiert, und generell als Framework zur Implementierung domänenspezifischer Sprachen zu sehen ist. Nach der Vorstellung der beiden DSLs, wobei wir insbesondere die Domain DSL (zu entwickelnde Ziel DSL) näher vorgestellt haben, wurde die Entwicklung der V-Modell XT Sprache näher beschrieben und aufgezeigt, wie beispielsweise das V-Modell XT Metamodell in die Strukturen von PDE umgesetzt werden kann. Als DSL Modellierungsframework setzt PDE auf automatische Quellcodegenerierung, die anhand eines einfachen Beispiels näher gebracht wurde.

Als Ergebnis dieser Arbeit steht neben PDE auch die V-Modell XT DSL bereit, die ihrerseits über einen eigenständigen Editor in Form einer Applikation verfügt, das unabhängig von weiteren Applikationen ausgeführt werden kann. Diese DSL weist neben einer grafischen Modellierung für Projektdurchführungsstrategien auch noch unter anderem eine benutzerspezifische Validierung derselben auf, so dass zum einen Fehlervorbeugung direkt in den Modellierungsmöglichkeiten der Sprache und zum anderen Fehlerfindung mittels Validierung implementiert wurden.

Das *Process Development Environment* ist auch ein Ergebnis dieser Arbeit, das als Basis zur Entwicklung der Sprache des V-Modells Verwendung fand. Dieses DSL-Modellierungsframework kann zur Entwicklung weiterer Sprachen benutzt werden, insbesondere wenn sie auf einem Metamodell basieren, das dem V-Modell XT ähnlich ist. Das Konzept von PDE als auch der Entwicklung von Sprachen mit PDE wurden im Rahmen im dieser Arbeit erarbeitet und liegen nun zur Verwendung vor.

Schließlich wollen wir noch die Fehlermeldungen aus der Einleitung zu dieser Arbeit (siehe Kapitel 1) im Kontext der V-Modell XT Sprache betrachten. Dazu machen wir genau dieselben Fehler in der DSL, und erhalten (bei beiden Fehlern gleichzeitig) die in Abbildung 6.1 dargestellte Fehlerliste:

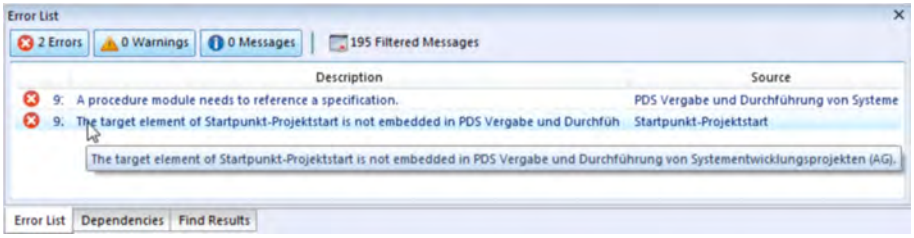


Abbildung 6.1: Fehlermeldungen in der V-Modell XT Sprache

Interessant hierbei ist vor allem auch der notwendige Aufwand, um solch einfache Fehler zu erkennen. Für die obigen Fehler haben wir den notwendigen Quellcode in Anhang B hinterlegt, generell kann der Aufwand dabei als gering festgehalten werden. Somit haben wir auch gesehen,

dass im Rahmen der V-Modell Sprache die Validierung dem Prozessingenieur bei der Modellierung behilflich ist.

Entwicklungsaufwand. Folgend wollen wir noch einige Zahlen bezüglich Quellcodelinien festhalten, die Aufschluss darüber geben sollen, wie aufwendig die Implementierung war und welchen Anteil insbesondere bei der Reduzierung des Implementierungsaufwandes die automatische Quellcodegenerierung aufweisen kann. Insbesondere wollen wir dabei auch die bereits im Rahmen von PDE vorliegenden Quellcodezeilen den spezifisch für die V-Modell Sprache implementierten Zeilen gegenüberstellen.

Die Ergebnisse sind in Tabelle 6.1 zusammengefasst[1]. Bei der Domain DSL splittet sich dabei der benutzerspezifische Quellcode auf zwei Bereiche auf:

V-Modell XT fasst alle Quellcodezeilen zusammen, die spezifisch für die V-Modell XT Sprache geschrieben wurden (HTML-Editor, Einschränkungen in den Modellierungsbereichen, ...).

T4 Templates stellen die Basis zur Generierung von Quellcode dar und wurden für die Domain DSL spezifisch geschrieben. Die Language DSL weist auch T4 Templates auf, diese liegen allerdings im Rahmen der Microsoft DSL-Tools bereits implementiert vor, so dass wir sie hier nicht mitzählen müssen.

Interessant an den erhaltenen Quellcodelinien-Ergebnissen ist die Tatsache, dass für die V-Modell Sprache nur sehr wenige Zeilen zu schreiben waren, die sich zudem größtenteils auf den relativ statischen Html-Editor sowie auf die Validierung verteilt haben. Der bereits vorliegende Quellcode im Rahmen von PDE ist dabei für die Entwicklung der Sprache selbst weniger relevant und zeigt lediglich, dass bei PDE selbst auch einiges an Aufwand erbracht werden muss. Ist dieser Aufwand allerdings erbracht, so dass ein solches DSL Modellierungsframework bereits vorliegt, so kann damit eine Sprache auch recht schnell entwickelt werden.

Fazit: Liegt ein ausreichendes beziehungsweise passendes DSL Modellierungsframework vor, dann ist die Implementierung einer Sprache basierend auf einem Modell aus der technischen Sicht relativ einfach. Hierdurch kann bei der Umsetzung insbesondere die fachliche Sicht, die aus der Abbildung eines (Meta)-Modells in das Domänenmetamodell der Sprache besteht, fokussiert werden.

Name	Typ	Anzahl	
Language DSL	benutzerspezifisch	12640	18%
	generiert	58106	82%
	gesamt	*70756*	
ToolFramework	benutzerspezifisch	14596	58%
	generiert	10578	42%
	gesamt	*25174*	
Domain DSL	V-Modell XT	4581	**2**%
	T4 Templates	9121	3%
	generiert	251751	95%
	gesamt	*265453*	
Summe	benutzerspezifisch	40938	11%
	generiert	320445	89%
	gesamt	**361383**	

Tabelle 6.1: Anzahl der Quellcodelinien in PDE (V-Modell XT DSL eingeschlossen)

Aufwand bei Metamodelländerungen. Bei der Betrachtung einer Sprache für das V-Modell spielt nicht nur der Erstentwicklungsaufwand eine Rolle, vielmehr sind die Folgeaufwände, die durch Änderungen am Metamodell initiiert und in der DSL mitabgebildet werden müssen von

1 Zur Zählung der Quellcodelinien wurde die Suche bei Visual Studio mit der Regex-Eingabe [Sch09]
`^~(:Wh@//.+)~(:Wh@\{:Wh@)~(:Wh@\}:Wh@)~(:Wh@/#).+_`
und der Einschränkung auf **.cs* und **.xaml* Dateien benutzt. Die Zählung haben wir über die Suche durchgeführt, da die Visual Studio Analyse (*Calculate Code Metrics*) den generierten vom benutzerspezifischen Quellcode nicht trennen konnte und dazu gleichzeitig zur Analyse ein kompilierbares Programm voraussetzte.

großer Wichtigkeit. Dazu wollen wir zunächst ausschnittsweise Änderungen betrachten, die häufig sind und das Metamodell erweitern, um abschließend auf kritischere Anpassungen einzugehen, die bestehende Strukturen verändern. Dabei schätzen wir auf Basis der Erfahrungen neben den Aufwänden auch Eintrittswahrscheinlichkeiten der jeweiligen Änderungsvorgänge.

Hinzufügen einer neuen Beziehung oder eines neuen Elements.

1. Kein Validierungscode notwendig:

   Eine neue Beziehung oder ein neues Element werden im Language DSL Designer für die Sprache angelegt und um notwendige Eigenschaften erweitert. Folgend muss noch die Quellcodegenerierung durchgeführt werden.

   Aufwand: sehr gering, 0 benutzerspezifische Quellcodezeilen

   Eintrittswahrscheinlichkeit: sehr hoch

2. Validierungscode notwendig:

   Wir legen wie oben ein neues Element an, führen die Quellcodegenerierung durch und implementieren die zugehörigen Validierungsmethoden.

   Aufwand: sehr gering bis mittel, abhängig von den Validierungsmethoden.

   Eintrittswahrscheinlichkeit: sehr hoch

3. Eigenschaften erfordern eine Enumeration als Typ:

   Bevor wir ein Element oder eine Beziehung anlegen, müssen wir zunächst eine neue Enumeration im Language DSL Designer anlegen, danach gehen wir wie oben bereits beschrieben vor.

   Aufwand: sehr gering bis mittel, abhängig von den Validierungsmethoden.

   Eintrittswahrscheinlichkeit: selten

4. Eigenschaften erfordern benutzerspezifische Typen:

   Bevor wir ein Element oder eine Beziehung anlegen, müssen wir zunächst einen benutzerspezifischen Typ implementieren. Dieser muss in Quellcode definiert werden und bedarf zudem eines spezifischen Editors für das Eigenschaftsfenster. Der Implementierungsaufwand ergibt sich hier direkt aus der Komplexität dieses Editors. So wäre dieser, bei ähnlichen Editoren wie dem für die V-Modell Sprache erstellten HTML Editor, mit sehr hoch zu bewerten.

   Aufwand: mittel bis sehr hoch, abhängig von den Editoren, die für die benutzerspezifischen Typen notwendig sind.

   Eintrittswahrscheinlichkeit: sehr selten

Änderungen an vorhandenen Strukturen.

1. Element oder Beziehungsänderungen bezüglich Eigenschaften

   Die Anpassungen für das Modell werden im Language DSL Designer durchgeführt. Der Aufwand der Anpassung von Validierungsmethoden oder Sichten, sollten welche vorliegen, ist hierbei in der Regel gering.

   Aufwand: gering

   Eintrittswahrscheinlichkeit: selten

2. Elementänderungen bezüglich der Positionierung im Hierarchiebaum des Modells

   Die Positionierung kann im Language DSL Designer vorgenommen werden, wobei hier gegebenenfalls neben dem Domänenmetamodell auch das Serialisierungsmodell angepasst werden muss. Der eigentliche Aufwand ergibt sich allerdings erst, falls Validierungsmethoden sowie spezifische Sichten vorliegen, die das Element miteinbeziehen.

   Aufwand: gering bis mittel, abhängig von Validierungsmethoden und Sichten

   Eintrittswahrscheinlichkeit: selten

3. Ganze Teilmodelle, die über keine Validierung oder spezifische Sichten verfügen

   Diese Änderung lässt sich auch anhand zwei separater Vorgänge betrachten. Zunächst werden nicht mehr gebrauchte Strukturen gelöscht, um folgend neue anzulegen oder bestehende anzupassen. Dabei gelten die oben betrachteten Aufwände.

Aufwand: gering bis hoch, abhängig von neuen Strukturen

Eintrittswahrscheinlichkeit: selten

4. Ganze Teilmodelle, die über Validierung und/oder spezifische Sichten verfügen

   Neben der Löschung und Anpassung bestehender Elemente, müssen auch zugehörige Validierungsmethoden sowie Sichten angepasst oder entfernt werden. Folgend kommen noch die Aufwände zum Erstellen neuer Elemente.

   Aufwand: mittel bis hoch, abhängig von neuen Strukturen

   Eintrittswahrscheinlichkeit: selten

Verallgemeinerung. Viele Prozessmodelle verfügen über spezifische Werkzeuge, einige davon sind allerdings entweder zu umfangreich und zu komplex oder erschweren die Arbeit des Prozessingenieurs zwecks mangelnder oder unzureichender Funktionalität. Gerade für diese Prozessmodelle ist eine Umsetzung mittels einer DSL interessant und, wie beim V-Modell XT in dieser Arbeit gezeigt, sinnvoll. Für das V-Modell XT wurde allerdings zunächst ein DSL-Modellierungsframework entwickelt, das insbesondere darauf abgezielt hat, Flexibilität in der Anpassung sowie handhabbaren Änderungsaufwand bei Modifikationen am V-Modell XT Metamodell bereitzustellen. Wäre dieses Modellierungsframework bereits zu Beginn dieser Arbeit zur Verfügung gestanden, so hätte der Implementierungsaufwand enorm reduziert werden können. Somit lässt sich verallgemeinert festhalten, dass domänenspezifische Sprachen sinnvoll bei Prozessmodellen eingesetzt werden können, sofern das Modellierungsframework, das der Implementierung der entsprechenden DSL dient, die Anforderungen des Prozessmodells genügend unterstützt und die so erstellte DSL später angepasst und erweitert werden kann.

Das in dieser Arbeit erstellte *Process Development Environment* erlaubt die Definition von DSLs, sofern das zugehörige Modell im Domänenmetamodell von PDE abgebildet werden kann. Somit können DSLs mithilfe von PDE erstellt werden, wobei PDE selbst zunächst nur als Basisframework für die V-Modell XT DSL gedacht war.

Die Umsetzung von Modellen zu domänenspezifischen Sprachen zwecks der Bereitstellung eines unabhängigen Editors in Form einer Applikation ist generell eine interessante Thematik. Betrachtet man beispielsweise die triviale Familienstammbaumsprache aus Abschnitt 5.1.2, so stellt man fest, dass für dieses einfache Modell eine recht ansehnliche Modellierungsmöglichkeit erstellt wurde. Der Aufwand dieser Erstellung (PDE selbst nicht berücksichtigt) beläuft sich auf wenige Stunden, eine eigene Applikation mit all den angebotenen Funktionalitäten hingegen würde deutlich größeren Aufwand verlangen. Somit verringert sich der Aufwand bei einer derartigen Umsetzung, wobei die Verwendung eines DSL Frameworks bei einem solchen Vorhaben auch hinsichtlich Fehlerfreiheit wichtig ist. Nehmen wir an, dass der generierte sowie der vorliegende Quellcode möglichst fehlerfrei sind, so müssen im Rahmen der V-Modell XT Sprache nur in etwa 4500 Quellcodezeilen auf Fehlerfreiheit getestet werden. Bei einer eigenen Applikation wie dem V-Modell XT Editor ist da deutlich mehr Aufwand notwendig.

Schließlich lässt sich festhalten, dass diese Arbeit verallgemeinert auch für andere Anwendungen interessant ist, genauer genommen genau dort, wo sich DSLs anwenden lassen und insbesondere grafische Editoren zur Modellierung von Sprachinstanzen wichtig sind.

Offene Fragen. Die Entwicklung von PDE sowie der DSL für das V-Modell XT hat einige Fragen aufgeworfen, die im Rahmen der Arbeit nicht bearbeitet werden konnten:

- Validierungssprache: Validierung wird bei PDE mittels Quellcode durchgeführt. Die Frage nach einer speziellen Validierungssprache, die bereits zur Designzeit notwendige Constraints festlegt, stellt sich unmittelbar aufgrund der Nachverfolgbarkeit bezüglich der implementierten Validierungslösung (für welche Elemente sind Validierungsmethoden vorhanden und was überprüfen diese?). So ließe sich einfacher feststellen, was, wo und wie genau validiert wird. Andererseits müsste eine solche Validierungssprache gleichzeitig offen für Anpassungen und Erweiterungen sein, denn sie würde, verglichen mit der Quellcodelösung, zwangsläufig weniger Möglichkeiten zur Überprüfung bereitstellen.
- Das Diagrammodell erfordert für die tatsächliche Darstellung ein mittels Xaml benutzerspezifisch für ein Shape implementiertes *DataTemplate*. Interessant an der Stelle ist die Frage, ob die Xaml-Templates nicht auch automatisch erzeugt werden könnten. Das setzt allerdings eine entsprechende Unterstützung in der Language DSL sowie in ihrem Designer

voraus. Möglicherweise wäre das zwar nur für geometrische Darstellungen sinnvoll, allerdings würde PDE auch davon schon profitieren.

- Layouting- und Routing-Algorithmen beim Diagrammodell fehlen PDE noch. Diese, insbesondere spezifisch auf die Möglichkeiten der diagrammartigen Darstellung ausgelegt, wären sehr sinnvoll und würden die Modellierungsweise bezüglich Benutzerfreundlichkeit deutlich verbessern.

Weitere offene Fragen beziehen sich auch auf eine Verbesserung des Serialisierungsmodells, indem es beispielsweise selbst als eine Basissprache implementiert wird und folglich noch mehr Flexibilität bezüglich des Serialisierungsformats bereitstellt. Generell ist das Löschen und Speichern von Daten aus unterschiedlichen Quellen (bei einer Serialisierungssprache könnten tatsächlich gleichzeitig mehrere unterstützt werden) eine sehr interessante Frage. Auch die generelle Skalierbarkeit von Sprachen könnte hierbei untersucht werden. Derzeit wird ja das gesamte serialisierte Modell geladen, eine Umstellung beispielsweise auf eine datenbankorientierte Lösung würde dem Sprachumsetzungsansatz neue Möglichkeiten vor allem bezüglich Sprachen, die über deutlich größere Instanzen verfügen, verleihen. Dazu müsste allerdings neben der Domain DSL womöglich auch die Language DSL und ihr Designer erweitert werden, so dass diese mittels eines komponentenbasierenden Ansatzes vorliegen, um auch noch deutlich größere Metamodelle zu unterstützen. Gerade die Language DSL wirft zudem die Frage nach einer erweiterten Spezialisierung auf, die noch vor der Auslieferung vorgenommen werden könnte. So wäre es bei PDE sinnvoll, auch schon die Language DSL mit einer spezifischen Validierung hinsichtlich des V-Modell XT Metamodells auszustatten, so dass eine noch einfachere und aufschlussreiche Modellierungsmöglichkeit vorliegt.

Ausblick. Die Sprache für das V-Modell XT sowie das PDE befindet sich weiterhin im Entwicklungsbeziehungsweise Erweiterungsstatus. Insbesondere wird demnächst die folgende Funktionalität umgesetzt:

- Suche nach bestimmten Eigenschaftswerten von Elementen und Beziehungen im Domänenmodell.
- Integration der am Lehrstuhl vorhandenen Werkzeuge sowie eine Erweiterung der Plugin-Schnittstelle zur Unterstützung von Plugins, die View Modelle zur Verfügung stellen.
- Geplant sind Erweiterungen der Diagrammoberfläche um
  - Speicherung der Darstellung als Layout
  - Löschen und Wiederherstellen von Layouts
  - Drag&Drop Funktionalitäten zum Anlegen von Beziehungen
  - Kennzeichnung von Elementen, die über nicht visualisierte Abhängigkeiten verfügen.

Das derzeit festgelegte Ziel ist es PDE als Integrationsplattform bereitzustellen. Dazu zählt auch eine Werkzeugkette, die aus der V-Modell XT DSL, dem *Process Enactment Tool Framework* (PET) [KKT10] sowie dem *IntelligentWork Item Tracking System* (WinterRain) [The10] bestehen soll.

A Fehlermeldungen des V-Modell XT Editors

In der Einleitung haben wir Fehlermeldungen des V-Modell XT Editors betrachtet. Diese, sowie ausgewählt einige nicht erkannte Fehler, wollen wir hier eingehender vorstellen.

Fehler a**.** Beim ersten Fehler haben wir beim Übergang *Startpunkt-Projektstart* des Ablaufbausteins *PDS Vergabe und Durchführung von Systementwicklungsprojekten (AG)* das Ziel auf ein Element außerhalb des Ablaufbausteins gelegt, genauer genommen auf den Ablaufbausteinpunkt *Abnahme* des Ablaufbausteins *Begleitung von Systementwicklungsprojekten (AG)*. Der Export liefert hier den folgenden Fehler:

Abbildung A.1: Meldung für den Fehler a im V-Modell XT Editor

Fehler b**.** Den zweiten Fehler haben wir beim Ablaufbaustein *PDS Vergabe und Durchführung von Systementwicklungsprojekten (AG)* gemacht, indem wir dort die Referenz der Ablaufbausteinspezifikation *AG-PDS* entfernt haben. Daraufhin haben wir beim Export die folgende Fehlermeldung erhalten:

Abbildung A.2: Meldung für den Fehler b im V-Modell XT Editor

Fehler c**.** Den letzten gezeigten Fehler haben wir über die Änderungen der V-Modell-XT.xml Datei erreicht, indem wir das Element *Vorgehensbausteine* in *VVorgehensbausteine* umbenannt haben. Das Aufzeigen dieses Fehler erscheint auf den ersten Blick in der Hinsicht der Eintrittswahrscheinlichkeit als weniger sinnvoll, allerdings kann das V-Modell auch von unterschiedlichen Werkzeugen verarbeitet werden, so dass auch solche Fehler eintreten können, die dann auch sehr schwer zu finden sind. Die Fehlermeldung des V-Modell XT Editors sieht dabei wie folgt aus:

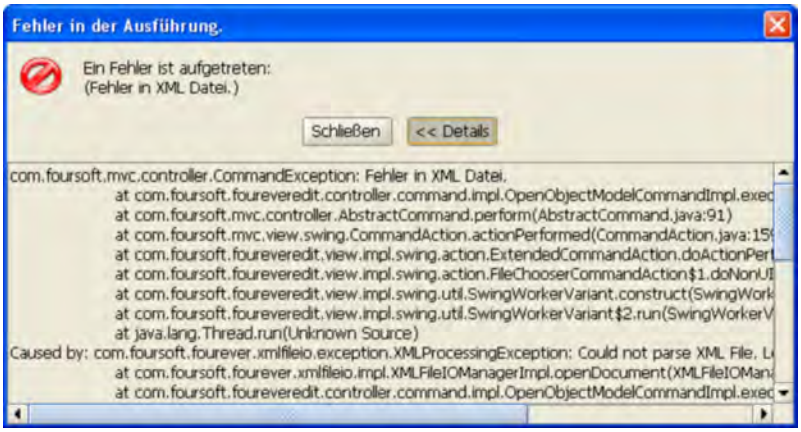


Abbildung A.3: Meldung für den Fehler c im V-Modell XT Editor

Fehler d**.** In der Einführung zu dieser Arbeit haben wir bereits erwähnt, dass der Export nicht alle vorhanden Fehler im Modell findet. Einen solchen Fehler wollen wir hier aufzeigen. Hierzu entfernen wir den Übergang *Org nach Projektabschluss* im Ablaufbaustein *PDS Einführung und Pflege eines organisationsspezifischen Vorgehensmodells*. Der anschließende Export läuft fehlerfrei ab, so dass wir eine Dokumentation für das V-Modell erhalten, die allerdings über den in Abbildung A.4 dargestellten Fehler verfügt. Die dort dargestellte Strategie kann nie abgeschlossen werden, da ein Übergang zu *Projekt abgeschlossen* fehlt.

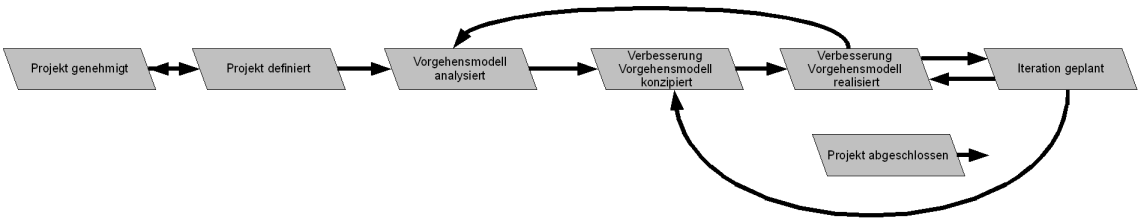


Abbildung A.4: Fehler d in der V-Modell XT Dokumentation

B Validierung für die Fehlermeldungen des V-Modell XT Editors

In der Einleitung haben wir Fehlermeldung des V-Modell XT Editors betrachtet und sie auch genauer in Anhang A aufgezeigt. Nun wollen wir vorstellen, wie in der entwickelten Sprache für das V-Modell solche Fehler validiert werden. Hierbei gehen wir nur auf Fehler a und b ein, der Fehler c wird automatisch von den generierten Serialisierungsklassen erkannt.

Fehler a. Den ersten Fehler erkennen wir, indem wir in jeder Instanz eines Ablaufbausteins die vorliegenden Übergänge validieren und dabei schauen, ob Ziel und Quelle im aktuellen Ablaufbaustein liegen. Dazu dient uns der folgende Quellcode:

Listing B.1: Validierungscode zur Erkennung von Fehler a

```
public partial class Ablaufbaustein
{
    public void Validate(ModelValidationContext context)
    {
        // ...

        foreach (Uebergang uebergang in this.Uebergang)
        {
            if (uebergang.Quelle != null)
            {
                if (!IsAblaufpunktInAblaufbaustein(this, uebergang.Quelle))
                    context.AddMessage(
                           new ModelValidationMessage(
                               VModellXTValidationMessageIds.
                                   PDSUebergangElementNotInAblaufbausteinID,
                               ModelValidationViolationType.Error,
                               'The source element of ' + uebergang.Name + ' is not
                                   embedded in ' + this.Name + '.',
                               uebergang));
            }

            if (uebergang.Ziel != null)
            {
                if (!IsAblaufpunktInAblaufbaustein(this, uebergang.Ziel))
                    context.AddMessage(
                           new ModelValidationMessage(
                               VModellXTValidationMessageIds.
                                   PDSUebergangElementNotInAblaufbausteinID,
                               ModelValidationViolationType.Error,
                               'The target element of ' + uebergang.Name + ' is not
                                   embedded in ' + this.Name + '.',
                               uebergang));
            }
        }

        // ...
    }

    public static bool IsAblaufpunktInAblaufbaustein(Ablaufbaustein ablaufbaustein,
        Ablaufpunkt ablaufpunkt)
    {
        ModelElement parent = VModellXTElementParentProvider.Instance.GetEmbeddingParent
            (ablaufpunkt, Ablaufbaustein.DomainClassId);
        if (parent == ablaufbaustein)
            return true;
        else
            return false;
    }
}
```

Fehler b**.** Der zweite Fehler ist in unserer DSL sehr einfach zu erkennen. Dazu bedarf es des folgenden Quellcodes:

Listing B.2: Validierungscode zur Erkennung von Fehler b

```
public partial class Ablaufbaustein
{
    public void Validate(ModelValidationContext context)
    {
        if (this.Ablaufbausteinspezifikation == null)
        {
            context.AddMessage(
                   new ModelValidationMessage(
                       VModellXTValidationMessageIds.PDSSpecMissingId,
                       ModelValidationViolationType.Error,
                       'A procedure module needs to reference a specification.',
                       this));
        }

        // ...
    }
}
```

Literaturverzeichnis

[ABC+06] ASANOVIC, KRSTE, RAS BODIK, BRYAN CHRISTOPHER CATANZARO, JOSEPH JAMES GEBIS, PARRY HUSBANDS, KURT KEUTZER, DAVID A. PATTERSON, WILLIAM LESTER PLISHKER, JOHN SHALF, SAMUEL WEBB WILLIAMS und KATHERINE A. YELICK: *The Landscape of Parallel Computing Research: A View from Berkeley*. Technischer Bericht, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

[ADF+08] AIKEN, A., B. DALLY, R. FEDKIW, P. HANRAHAN, J. HENNESSY, M. HOROWITZ, V. KOLTUN, C. KOZYRAKIS, K. OLUKOTUN, M. ROSENBLUM und S. THRUN: *Towards Pervasive Parallelism (Pervasive Parallelism Laboratory, Stanford University)*. Online: `http://ppl.stanford.edu/wiki/images/9/93/PPL.pdf`, 2008.

[ASS+09] AXELSSON, EMIL, MARY SHEERAN, PER STENSTRÖM, GERGELY DÉVAI, ZOLTÁN HORVÁTH und ANDRÁS VAJDA: *Domain Specific Languages: state of the art and future directions*. Online: `http://dsl4dsp.inf.elte.hu/swday2009.pdf`, 2009.

[Bec03] BECK, K.: *Extreme Programming*. Nummer ISBN-13: 978-3827321398. Addison-Wesley, München, 2003.

[BF09] BERGNER, K. und J. FRIEDRICH: *Syntax und Semantik der Projektdurchführungsstrategien im V-Modell XT 1.3*. Technischer Bericht, 2009.

[CJKW07] COOK, S., G. JONES, S. KENT und A. C. WILLS: *Domain-Specific Development with Visual Studio DSL Tools*. 2007.

[EMF] *Das Eclipse Modeling Framework (EMF) Online Portal*. Online: http://www.eclipse.org/modeling/emf/.

[EPF07] *Das Eclipse Process Framework (EPF) Online Portal*. Online, `http://www.eclipse.org/epf`, Juli 2007.

[EPT] *Eclipse Platform Technical Overview*. Online: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[Eve08] *Composite Application Guidance for WPF - Event Aggregator*. Online: `http://msdn.microsoft.com/en-us/library/cc707867.aspx`, Juni 2008.

[FHKS09] FRIEDRICH, JAN, ULRIKE HAMMERSCHALL, MARCO KUHRMANN und MARC SIHLING: *Das V-Modell XT*. Springer, 2 Auflage, 2009.

[Hau06a] HAUMER, P.: *Eclipse Process Framework Composer – Part 1: Key Concepts*. 2006. Published online: http://www.eclipse.org/epf/general/EPFComposerOverviewPart1.pdf.

[Hau06b] HAUMER, P.: *Eclipse Process Framework Composer – Part 2: Authoring method content and processes*. 2006. Published online: http://www.eclipse.org/epf/general/EPFComposerOverviewPart2.pdf.

[HKKR05] HITZ, MARTIN, GERTI KAPPEL, ELISABETH KAPSAMMER und WERNER RETSCHITZEGGER: *UML@Work*. dpunkt.verlag, 2005.

[KKT10] KUHRMANN, MARCO, GEORG KALUS und MANUEL THEN: *Flexible Process-Tool-Integration - The Process Enactment Tool Framework*. Technischer Bericht, Technische Universität München, 2010.

[Kru03] KRUCHTEN, PHILIPPE: *The Rational Unified Process: An Introduction*. Addison-Wesley Longman, 3 Auflage, 2003.

[KRV08] KRAHN, HOLGER, PROF. DR. BERNHARD RUMPE und STEVEN VÖLKEL: *Domänenspezifische Sprachen modular entwickeln*. OBJEKTspektrum, 4, 2008.

[Kuh08] KUHRMANN, MARCO: *Konstruktion modularer Vorgehensmodelle*. Doktorarbeit, Technische Universität München, 2008.

[LL07] LUDEWIG, JOCHEN und HORST LICHTER: *Software Engineering*. dpunkt.verlag, 2007.

[MEF10] *Managed Extensibility Framework*. Online: `http://mef.codeplex.com/`, Februar 2010.

[MHS05] MERNIK, MARJAN, JAN HEERING und ANTHONY M. SLOANE: *When and How to Develop Domain-Specific Languages*. ACM Computing Surveys, Vol. 37, 2005.

[MS05] MERKSAND, ED und DAVE STEINBERG: *From Models to Code with the Eclipse Modeling Framework*. Online: `http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial11final.pdf`, 2005.

[MSD] MSDN: *Flow Document Overview*. Online: `http://msdn.microsoft.com/en-us/library/aa970909.aspx`.

[MSD10a] MSDN: *DataTemplate Class*. Online: `http://msdn.microsoft.com/en-us/library/system.windows.datatemplate.aspx`, 2010.

[MSD10b] MSDN: *XAML Overview (WPF)*. Online: `http://msdn.microsoft.com/en-us/library/ms752059.aspx`, 2010.

[OMG05] OMG: *Software Process Engineering Metamodel Specification*. Technischer Bericht, Object Management Group, 2005.

[Par07] *Partielle Klassen und Methoden (C#-Programmierhandbuch)*. Online: `http://msdn.microsoft.com/de-de/library/wa80x488.aspx`, November 2007.

[Pri09] *Prism - patterns & practices: Composite WPF and Silverlight*. Online: `http://compositewpf.codeplex.com/`, Oktober 2009.

[Ros81] ROSS, DOUGLAS T.: *Origins of the APT language for automatically programmed tools*. 1981.

[SBPM08] STEINBERG, DAVE, FRANK BUDINSKY, MARCELO PATERNOSTRO und ED MERKS: *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2 Auflage, 2008.

[Sch04] SCHWABER, K.: *Agile Project Management with Scrum*. Microsoft Press, 2004.

[Sch09] SCHUAGER, GERMÁN: *Line count in Visual Studio*. Online: `http://blog.schuager.com/2009/01/line-count-in-visual-studio.html`, 2009.

[Smi09] SMITH, JOSH: *WPF Apps With The Model-View-ViewModel Design Pattern*. msdn Magazine, Online: `http://msdn.microsoft.com/en-us/magazine/dd419663.aspx`, February, 2009.

[Syc07] SYCH, O.: *T4: Text Template Transformation Toolkit*. http://www.olegsych.com/2007/12/ text-template-transformation-toolkit/, 2007.

[T4R07] *Generating Artifacts By Using Text Templates*. Resource pag: http://msdn.microsoft.com/en-us/library/bb126445.aspx, 2007.

[The10] THEN, MANUEL: *Design of an IntelligentWork Item Tracking System*. Bachelor Thesis, Februar 2010.

[TK09] TERNITÉ, T. und M. KUHRMANN: *Das V-Modell XT 1.3 Metamodell*. Technischer Bericht TUM-I0905, Technische Universität München, 2009.

[VMXa] *Das V-Modell XT an der Technischen Universität München*. http://v-modell-xt.in.tum.de/Default.aspx.

[VMXb] *V-Modell XT Editor Resource page*. http://sourceforge.net/projects/fourever.

[W3Ca] *W3C, Extensible Markup Language Schema (XSD)*. Resource page: http://www.w3.org/XML/Schema.

[W3Cb] *W3C, Extensible Markup Language (XML)*. Resource page: http://www.w3.org/XML.

[W3Cc] *W3C, HyperText Markup Language (HTML)*. Resource page: http://www.w3.org/TR/html401/.

[Wac08] WACHTEL, EUGEN: *Ein Konfigurationswerkzeug für Prozesskomponenten - Darstellung von Abhängigkeiten auf größeren Datenstrukturen*. Bachelor Thesis, Oktober 2008.

[WKK09a] WACHTEL, EUGEN, MARCO KUHRMANN und GEORG KALUS: *A Domain Specific Language for Project Execution Models*. In: *39th Annual Conference of the German Computer Society*, 2009.

[WKK09b] WACHTEL, EUGEN, MARCO KUHRMANN und GEORG KALUS: *Eine domänenspezifische Sprache zur Modellierung und Prüfung von Projektabläufen*. Technischer Bericht, Technische Universität München, 2009.