TUM

INSTITUT FÜR INFORMATIK

Eine domänenspezifische Sprache zur Modellierung und Prüfung von Projektabläufen

Eugen Wachtel, Marco Kuhrmann, Georg Kalus

TUM-I0928
November 09

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-11-I0928-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Eine domänenspezifische Sprache zur Modellierung und Prüfung von Projektabläufen

Eugen Wachtel, Marco Kuhrmann, Georg Kalus

Technische Universität München
Institut für Informatik, Software & Systems Engineering
Boltzmannstr. 3
85748 Garching, Germany
wachtel@in.tum.de, kuhrmann@in.tum.de, kalus@in.tum.de

Zusammenfassung

Dieser Bericht beschreibt die Struktur und Eigenschaften einer domänenspezifischen Sprache sowie des dazugehörigen Prototyps zur Modellierung und Prüfung von Projektdurchführungsstrategien sowie von Projektplänen basierend auf dem Metamodell des V-Modell XT. Es werden die wesentlichen Elemente sowie Erweiterungen und Einschränkungen erklärt, die im Vergleich zum V-Modell XT notwendig sind, um eine valide Modellierung zu gewährleisten.

Keywords

Sofware Engineering, Software Development Process, V-Modell XT, Domänenspezifische Sprachen, Microsoft DSL-Tools

CR-Classification: D.2

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Entwicklung von Softwareprojekten wird zunehmend komplexer und muss folglich gründlich und sinnvoll geplant werden um erfolgreich zu sein [BEJ06, Int04]. Ausgangsbasis für die Erstellung von Projektplänen sind im deutschen V-Modell XT [FHKS08] die sogenannten *Projektdurchführungsstrategien*. Diese legen fest, in welcher Reihenfolge *Entscheidungspunkte* durchlaufen werden können und definieren hierdurch erlaubte und sinnvolle Abläufe. Während das V-Modell bereits einige vordefinierte Projektdurchführungsstrategien beinhaltet, ist das Design einer neuen Projektdurchführungsstrategie komplex und aufwendig, da hierzu verschiedenste Mechanismen [TK09] berücksichtigt werden müssen.

1.1 Design von Projektdurchführungsstrategien

Projektdurchführungsstrategien im V-Modell legen über die Reihenfolge von Entscheidungspunkten die Projektfortschrittsstufen eines Projekts fest, siehe Abbildung 1.1. Mit der Version 1.3 des V-Modells hat sich die zugrunde liegende Struktur radikal geändert. Neben der bislang be-

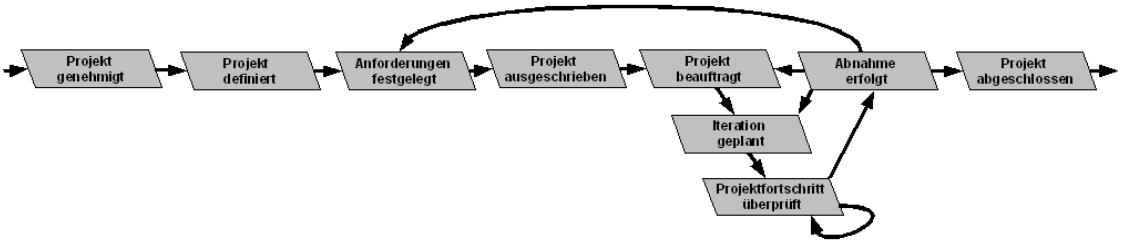

Abbildung 1.1: Projektdurchführungsstrategie eines AG-Projekts

reits vorhandenen Fähigkeit, auf Grundlage von Projektdurchführungsstrategien Projektpläne herzuleiten, werden diese nun auch für die Generierung der Abbildungen in der Prozessdokumentation verwendet. Weiterhin werden Projektdurchführungsstrategien nun auch im *Tailoring* des V-Modells verwendet. Ihre grundlegende Struktur ist nun analog zu *Vorgehensbausteinen* in sog. *Ablaufbausteinen* organisiert, die während des Tailorings zu einer integrierten Projektdurchführungsstrategie komponiert werden.

1.1.1 Bestandteile einer Projektdurchführungsstrategie

Um den Anforderungen hinsichtlich Modularität gerecht zu werden, wurden Ablaufbausteine als Modulkonzept für Projektdurchführungsstrategien grundlegend neu entwickelt. Ein Ablaufbaustein besteht in der Version 1.3 des V-Modell im wesentlichen aus folgenden Elementen:

- Start- und Endepunkte
- Ablaufbaustein- und Ablaufentscheidungspunkte
- Split- und Join-Knoten
- Übergängen

Hierbei gilt, dass jeder Ablaufbaustein genau einen Start- und genau einen Endepunkt hat. Zwischen diesen Punkten können beliebig viele Ablaufentscheidungs- oder Ablaufbausteinpunkte untergebracht werden. Ein Ablaufentscheidungspunkt referenziert genau einen Entscheidungspunkt des V-Modells. Ein Ablaufbausteinpunkt ist ein getypter Platzhalter, der einen anderen Ablaufbaustein über seine *Ablaufbausteinspezifikation* einbinden kann. Somit kann ein Ablaufbaustein hierarchisch aus anderen Ablaufbausteinen aufgebaut werden. Zwischen den einzelnen Punkten im Ablaufbausteinen sind Übergänge definiert, die die Reihenfolge (mögliche Pfade) vorgeben.

Das bislang im V-Modell existierende Konzept der *Parallelabläufe* wurde in der Version 1.3 aufgegeben. Parallele Ablaufstrukturen werden nun mithilfe von Splits und Joins realisiert. Ein Ablaufbaustein entspricht in seinem Aufbau einem bipartiten Graphen, d.h. zwischen zwei Punkten gibt es immer einen Übergang und zwei Übergänge sind immer durch einen Knoten (Punkt) getrennt. Kapitel 3 geht auf die Metamodellkonzepte im Zusammenhang mit Projektdurchführungsstrategien im Detail ein.

1.1.2 Entwurfsmethoden und Module

Der Entwurf einer Projektdurchführungsstrategie auf Basis von Ablaufbausteinen ist ein komplexer Vorgang. In Anlehnung an [KTF09] geben wir an dieser Stelle einen kurzen Einblick in die Entwurfsmethode für das Design einer Projektdurchführungsstrategie.

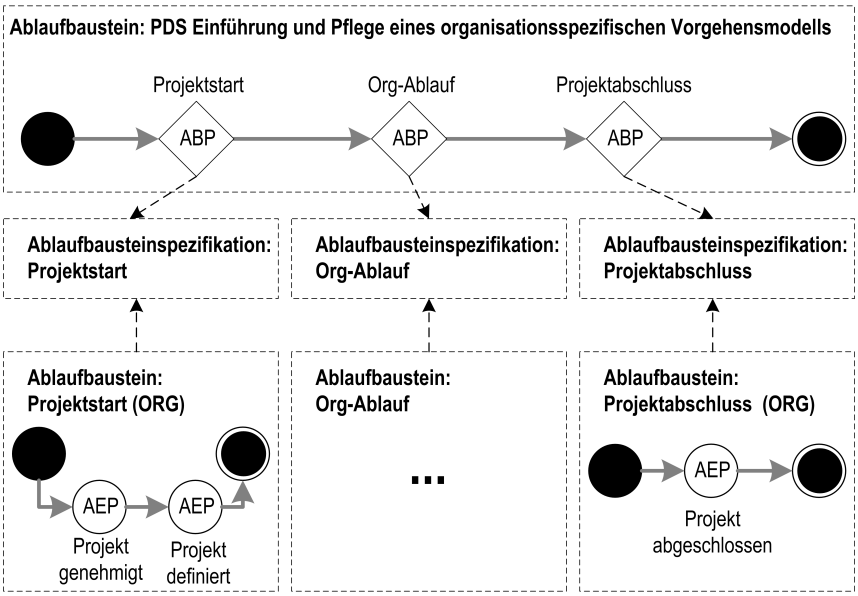

Abbildung 1.2: Grob granularer Entwurf einer Projektdurchführungsstrategie

Abbildung 1.2 zeigt den globalen Rahmen einer Projektdurchführungsstrategie. Oberstes Element ist ein „Wurzelablaufbaustein". Dieser definiert den Rahmenablauf – hier bestehend aus drei Ablaufbausteinpunkten *Projektstart*, *Org-Ablauf* und *Projektabschluss*. Diese Ablaufbausteinpunkte referenzieren je eine Ablaufbausteinspezifikation.

Zeitpunkt der Auflösung

Die Zuordnung von konkreten Ablaufbausteinen zu einem Ablaufbausteinpunkt erfolgt erst in der Projektplanung. Im Tailoring wird zunächst nur festgestellt, welche Ablaufbausteine gem. der geforderten Spezifikationen in Frage kommen. Mindestens einer *muss* vorhanden sein. In der Planung werden dann entsprechend Alternativen angeboten. Der Prozessingenieur kann dies nur bedingt über die Konfiguration des Tailorings steuern, muss jedoch eine Konsistenz der möglichen Abläufe sicher stellen.

Eine Projektdurchführungsstrategie *ist* also selbst bereits ein *hierarchischer Ablaufbaustein*, der mehrere „flache" (atomare) oder weitere hierarchische Ablaufbausteine einbinden kann.

Flache Ablaufbausteine. In Abbildung 1.2 wird die Modulstruktur (hier für die Projektdurchführungsstrategie *Einführung und Pflege eines organisationsspezifischen Vorgehensmodells*) gezeigt. Die einzelnen Module können nun schrittweise entworfen werden. Abbildung 1.3 zeigt den prinzipiellen Aufbau eines sog. „flachen" Ablaufbausteins. Ein flacher bzw. atomarer Ablaufbaustein referenziert ausschließlich Entscheidungspunkte (über Ablaufentscheidungspunkte). Die Abbildung zeigt gleichzeitig die Entsprechungen der Modellelemente zu den grafischen Elementen der Bilder aus der V-Modell Dokumentation. Es ist klar zu sehen, dass das zugrunde liegende Modell komplexer ist als die Darstellung – eine Schwierigkeit, die bei der Modellierung von Abläufen berücksichtigt werden muss.

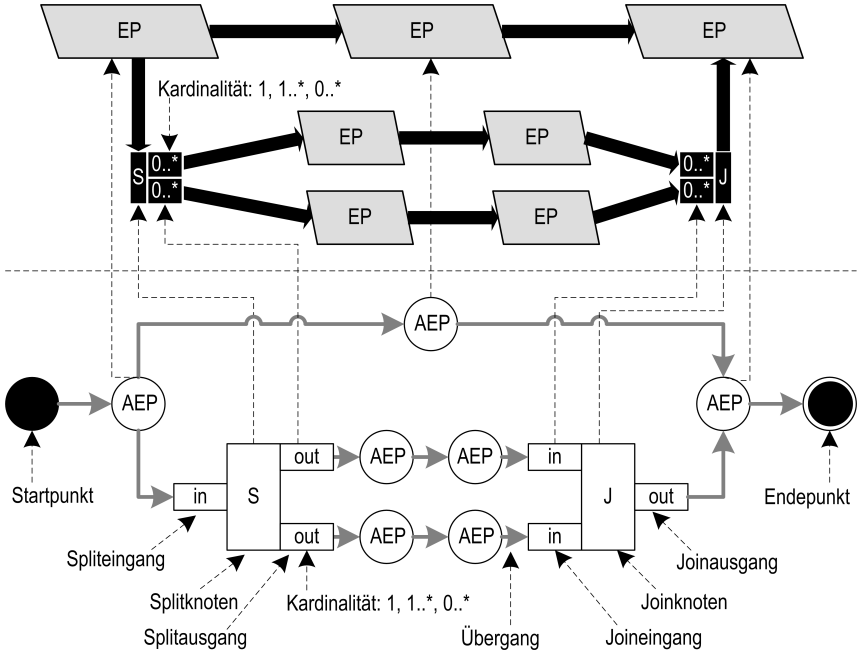

Abbildung 1.3: Aufbau eines „flachen" Ablaufbausteins

Hierarchische Ablaufbausteine. Ablaufbausteine können nicht nur direkt Entscheidungspunkte referenzieren, sondern weitere Ablaufentscheidungspunkte. Damit entsteht ein hierarchischer Ablaufbaustein, siehe Abbildung 1.4, möglich. Hier ist zu sehen, dass ein Teilablauf durch einen anderen Ablaufbaustein erbracht wird.

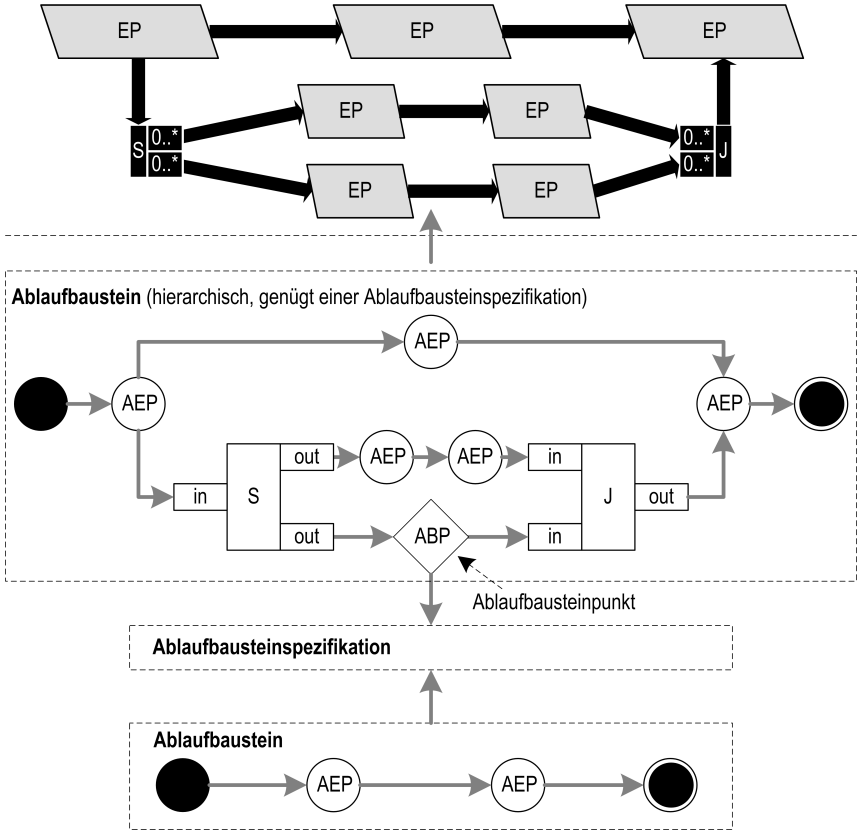

Abbildung 1.4: Ablauf eines hierarchischen Ablaufbausteins

Anstelle die Entscheidungspunkte des Teilablaufs direkt im Ablaufbaustein zu referenzieren,

wird hier ein Ablaufbausteinpunkt eingebunden. Dieser referenziert eine Ablaufbausteinspezifikation, die durch einen weiteren Ablaufbaustein erfüllt wird. Beim Entwurf ergibt sich eine gewisse Komplexität aus der Tatsache, dass die Darstellung *und* das Verhalten in der Projektplanung identisch sind. Die Modellstrukturen verbergen durch die Werkzeuge des V-Modells die Komplexität zwar vor dem Anwender. Der Prozessingenieur allerdings muss diese Komplexität beim Entwurf einer Projektdurchführungsstrategie sehr wohl beherrschen. Hinzu kommt, dass dieser vorausdenken muss, um die Modellierung so zu gestalten, dass die Anwender das gewünschte Resultat erhalten.

Vorgehen beim Entwurf. Beim Entwurf von Ablaufbausteinen schlagen [KTF09] vor, in Ermangelung einer entsprechenden Werkzeugunterstützung zunächst „auf Papier" die gewünschten Ablaufstrukturen zu skizzieren. Dabei sollte zunächst ein *Top-Down*-Ansatz verwendet werden, der zuerst den prinzipiellen Ablauf als Wurzelablaufbaustein erfasst. Dieser soll, z. B. phasenorientiert, zerlegt werden. Jede Zerlegung wird selbst wieder als Ablaufbaustein verstanden und modelliert. Dabei ist dann bereits zu entscheiden, ob

1. der aktuelle Ablaufbaustein schon atomar ist, oder ob
2. der aktuelle Ablaufbaustein hierarchisch ist und eine weitere Zerlegung erforderlich ist.

Dies ist solange zu wiederholen, bis alle Ablaufbausteine als atomare Ablaufbausteine vorliegen. Ist dies geschehen, sind für alle Ablaufbausteine entsprechende Ablaufbausteinspezifikationen zu fertigen, damit atomare Ablaufbausteine in hierarchischen auch referenziert werden können. Danach können die Ablaufbausteine *Bottom-Up* im V-Modell implementiert werden. Während bei der Konzeption durchaus Darstellungen wie z. B. in den oberen Teilen der Abbildungen 1.3 und 1.4 verwendet werden können, ist für die Implementierung der Entwurf zu verfeinern (siehe untere Teile der Abbildungen). Diese sind schrittweise im V-Modell zu implementieren. Liegen die atomaren Ablaufbausteine und alle Spezifikationen vor, wird dieser Umsetzungsschritt für die Ablaufbausteine der nächsten Hierarchiestufe wiederholt, solange bis der Wurzelablaufbaustein im Modell umgesetzt ist.

1.1.3 Schwierigkeiten beim Entwurf

Problematisch beim Design einer Projektdurchführungsstrategie ist, dass es zwar eine Methode [KTF09] und eine Zielstruktur gibt, die technische Unterstützung jedoch lediglich die Zielstruktur abbildet. Der V-Modell-Editor (Abbildung 1.5) stellt lediglich die XML-Struktur dar. Diese ist flach und behandelt alle Modellelemente (Ablaufentscheidungspunkte, Splits etc.) gleichrangig. Allein aus dieser Struktur ist nicht sofort ersichtlich, ob alle Anforderungen des Entwurfs auch im Modell eingegangen sind. Wird beispielsweise ein Ablaufbaustein gem. dem gerade skizzierte Vorgehen erstellt und liegt ein ausmodelliertes Konzept (z. B. Abbildung 1.3) vor, kann ein Prozessingenieur mit dem Editor lediglich *manuell* prüfen, ob:

- alle Kanten als Übergänge modelliert wurden
- alle Kanten die korrekten Ziele referenzieren
- alle Kardinalitäten an Splits und Joins stimmen
- alle Kardinaltiäten an Splits über den Gesamtablauf hinweg konsistent sind
- …

Dazu kommt, dass die Projektdurchführungsstrategie zwar über den Wurzelablaufbaustein als Rahmen festgelegt wird, die einzelnen konkreten Ablaufbausteine jedoch erst durch das Tailoring bestimmt werden. Der Prozessingenieur muss also neben der korrekten Modellierung der einzelnen Module auch sicher stellen, dass auch *alle* Module (inkl. der hierarisch eingebundenen) im Tailoring entsprechend konfiguriert sind.

*Hierfür bietet das V-Modell XT **keine** Werkzeugunterstützung.*

Einzig der Export des V-Modells dient als Prüfung. Dadurch, dass die Ablaufbausteine auch für die Generierung der Bilder in der Prozessdokumentation verwendet werden, analysiert der Export das Modell und erstellt bei einer (syntaktisch) korrekt modellierten Ablaufbausteinstruktur eine Abbildung (siehe Abbildung 1.6).

Achtung: Der Export überprüft jedoch nur, ob eine syntaktisch korrekte Modellierung vorliegt. Er beachtet dabei keinerlei Konsistenzbedingungen oder Ähnliches. Diese können vom Prozessingenieur nur manuell durchgeführt werden. Eine weitere Prüfinstanz ist auch der V-Modell-Projektassistent – genauer sein Planungsmodul. Da dieses die Erstellung eines konkreten Projekt-

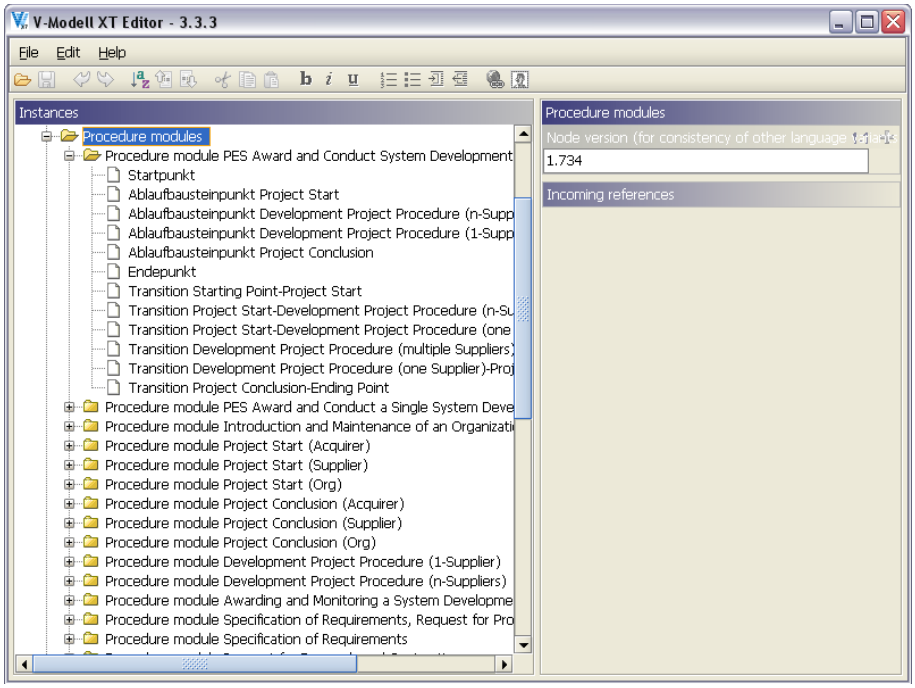

Abbildung 1.5: V-Modell XT Editor

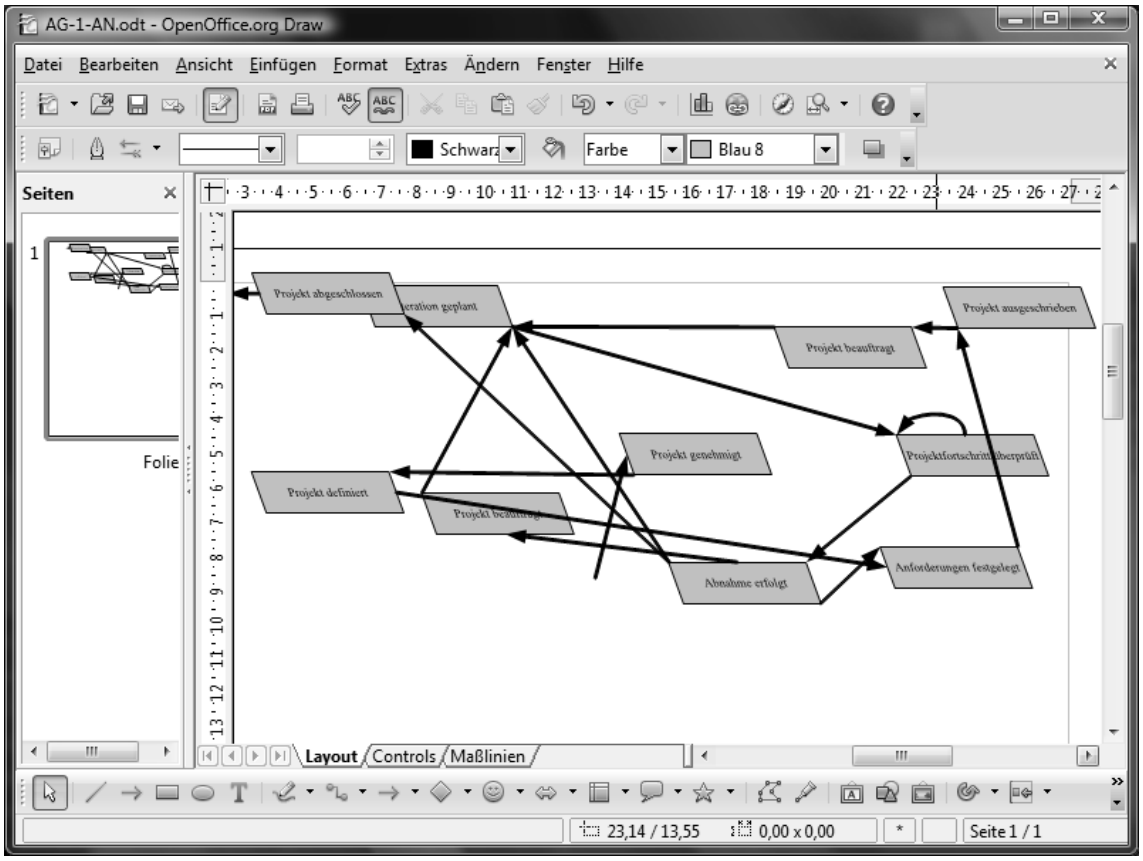

Abbildung 1.6: Automatisch generiertes Bild für eine Projektdurchführungsstrategie

plans nur entlang der Vorgaben der zugrunde liegenden Projektdurchführungsstrategie erlaubt, kann es genutzt werden, um zu püfen, ob die Modellierung den Vorstellungen entspricht.

Gerade bei komplexeren Abläufen ist es aufgrund der oben beschriebenen nicht optimalen Unterstützung durch den V-Modell-Editor jedoch nicht auszuschließen, dass Fehler enthalten sind. An dieser Stelle wird der Prozessingenieur allein gelassen, da die Fehlermeldungen, die der Export produziert, keine Rückschlüsse auf die Ursache liefern. Abbildung 1.7 zeigt eine typische Fehlermeldung. Es ist hier nicht möglich zu bestimmen, was die Ursache ist. Mögliche Probleme sind

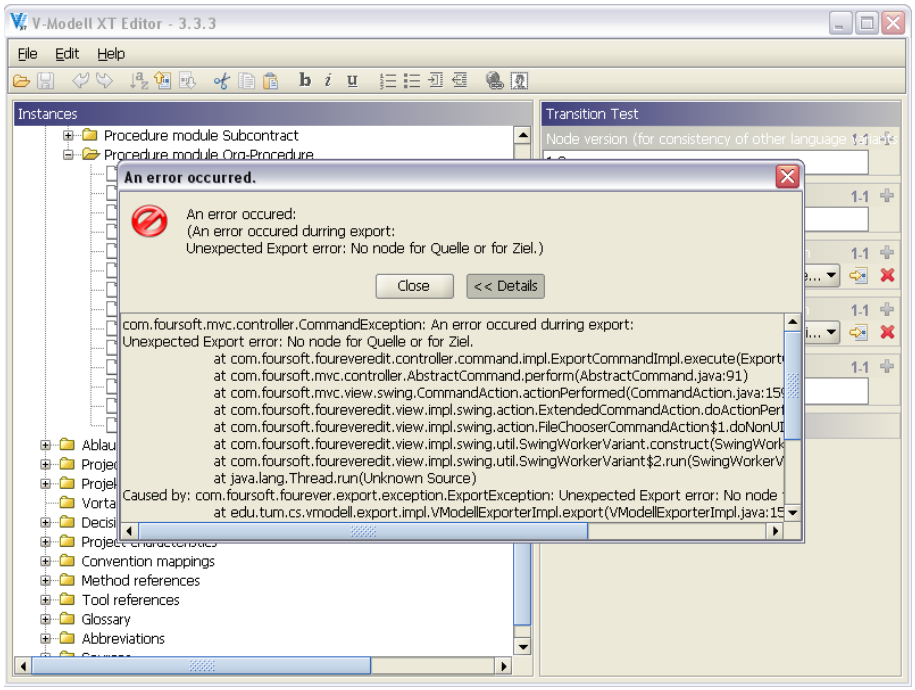

Abbildung 1.7: V-Modell XT Editor - Feststellung eines Fehlers beim Export

fehlende Übergänge, falsche oder fehlende Referenzen oder falsch konfigurierte Tailoringmodelle. Die Ursachenforschung ist an dieser Stelle sehr aufwändig, da der Export einfach abbricht und somit auch keine Informationen darüber gibt, an welcher Stelle (i.S. welcher Ablaufbaustein) des Modells genau der Fehler aufgetreten ist. Die aktuell einzige Möglichkeit ist mithilfe der detaillierten Entwürfe der Abläufe und im Ausschlussverfahren den Fehler einzugrenzen.

Hier besteht ein Handlungsbedarf, da das Verfahren zum Design einer Projektdurchführungsstrategie komplex und fehleranfällig ist. Die existierende Methode zum Design muss durch ein Werkzeug unterstützt werden, um dem Prozessingenieur ein strukturiertes Vorgehen anzubieten. Ziel ist es, eine domänenspezifische Sprache zu erstellen, die in der Lage ist, Projektdurchführungsstrategien über grafisch visualisierte Strukturen zu modellieren bzw. zu verwalten. Diese soll über eine Validierungsmöglichkeit verfügen, die bereits zur Designzeit Fehler findet und diese dem Prozessingenieur rückmeldet. Zusätzlich sehen wir die Möglichkeit zur Definition von Projektplänen in unserer domänenspezifischen Sprache vor, obwohl hier nicht der Fokus des vorliegenden Berichts liegt.

1.2 Zielgruppen

Zielgruppen dieses Berichts sind die Autoren des V-Modells selbst sowie Prozessingenieure, die Projektabläufe beschreiben und erstellen. Weiterhin ist dieser Bericht für Werkzeughersteller für das V-Modell als auch für weitere Vorgehensmodelle von Interesse und spricht zudem auch Entwickler im Bereich der domänenspezifischen Modellierung an, die sich insbesondere für die Microsoft DSL-Tools interessieren.

1.3 Aufbau des Berichts

Der Bericht ist wie folgt aufgebaut: Kapitel 2 befasst sich mit den Microsoft DSL-Tools und stellt die Möglichkeiten des Frameworks vor. Im Wesentlichen werden hier die Modellierungs- und Darstellungselemente des Frameworks angesprochen. Kapitel 3 stellt die wesentlichen Strukturen des Metamodells des V-Modell XT vor und beschreibt die Abbildung dieser Strukturen in das Domänenmodell der DSL-Tools. Weiterhin werden die Erweiterungen und Einschränkungen im Vergleich zum V-Modell XT erläutert, die notwendig sind um eine valide Modellierung zu gewährleisten sowie Prüfungen von bereits erstellten Strategien zu erlauben. Kapitel 4 beschreibt den zur domänenspezifischen Sprache gehörenden Prototyp und die wesentlichen Implementierungsdetails sowie die Vor- und Nachteile der hier verwendeten Microsoft DSL-Tools. Kapitel 5 gibt eine Zusammenfassung sowie einen Ausblick zur behandelten Thematik. Im Anhang dieses Berichts sind weitere technische Details der Umsetzung der DSL zu finden.

2 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 Erstellung 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.

Dieses Kapitel stellt die DSL-Tools im Detail vor. Insbesondere die Möglichkeiten der Modellierung sowie die dafür relevanten Elemente werden vorgestellt.

Übersicht

2.1 Das Domänenmodell

Das Domänenmodell der DSL-Tools besteht aus dem Element *DomainClass* und den dazwischen definierten Beziehungen (siehe auch Abbildung 2.1):

Embedding Relationship Eine DomainClass kann als Container für mehrere andere Klassen dienen. Damit lassen sich hierarchische Zusammenhänge modellieren.

Reference Relationship Eine DomainClass kann andere Klassen referenzieren. Eine Referenz in diesem Sinne stellt eine Verbindung zwischen zwei Klassen her, die sowohl ein- als auch beidseitig sein kann. 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 ersten beiden genannten Beziehungen sind selbst von der Klasse *DomainRelationship*, die von DomainClass abgeleitet ist. Damit werden die folgenden Eigenschaftsdefinitionen von DomainClass auch auf DomainRelationship vererbt.

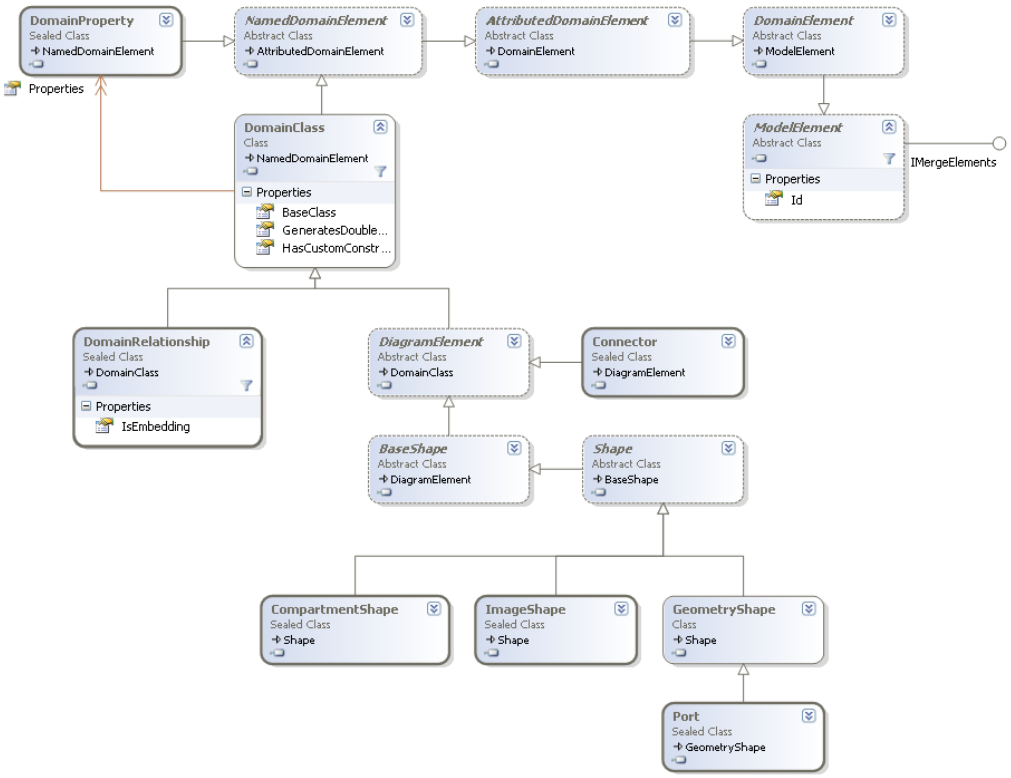

Abbildung 2.1: DSL-Tools Designer Modell (Ausschnitt)

Jede DomainClass-Instanz in den DSL-Tools weist eine Reihe vordefinierter Eigenschaften auf. Sie dienen der Festlegung von Kriterien, die bei der automatischen Quellcodegenerierung (siehe Abschnitt 2.3) berücksichtigt werden. Folgend finden sich die wichtigsten Eigenschaften wieder:

Id Jedes instanziierte Element hat eine eindeutige Id, die bei der Erstellung des Elements festgelegt wird und ab dann unveränderbar bleibt.

Has Custom Constructor Hierdurch wird festgelegt, dass der Constructor für die DomainClass-Instanz im Quellcode selbstständig zu implementieren ist. Das kann beispielsweise sinnvoll sein, wenn bei der Erstellung der jeweiligen Instanz gewissen Eigenschaften im Vorfeld festgelegt werden müssen, die nach der Erstellung nicht mehr modifizierbar sind (z.B.: *Id*).

Generates Double Derived Diese Eigenschaft legt fest, ob eine Klasse so generiert wird, dass alle überschreibbaren Eigenschaften in der partiell gebildeten Klasse auch tatsächlich überschreibbar sind. Das ist bei Eigenschaften und Methoden wichtig, die bereits in automatisch generierten Klassen überschrieben werden und somit sonst in der partiellen gebildeten Klasse nicht überschreibbar wären.

Hinweis

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 bzw. ihr Verhalten anpassen.

Weiterhin lassen sich jeder DomainClass-Instanz eigene Eigenschaften hinzufügen (*DomainProperty*). Diese können von einem Standard-Typ wie *String* bzw. *Bool* oder aber von einem eigens definierten DomainTyp sein. Auch externe Typen sind hier möglich.

Mithilfe der oben genannten Elemente lässt sich das Domänenmodell einer domänenspezifischen Sprache im Designer der DSL-Tools definieren. Die oben vorgestellten Elemente lassen sich dargestellt im DSL-Tools Designer der Abbildung 2.2 entnehmen.

2.2 Visualisierung des Domänenmodells

Neben den Modellierungsmöglichkeiten des Domänenmodells in den DSL-Tools gibt es einen weiteren, wesentlichen Baustein: Die Definition der Visualisierung der Elemente des Domänenmodells. Dazu betrachten wir allerdings zunächst die Bereiche (Abbildung 2.2), 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 Elemente 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 selbst dargestellt werden. Eine beispielhafte Visualisierung lässt sich der Abbildung 2.2 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.

Die vorgestellten Shapes sind alle (nach der automatischen Quellcodetransformation) von der Klasse *ShapeElement* abgeleitet, die wiederum von der Klasse *ModelElement* erbt.

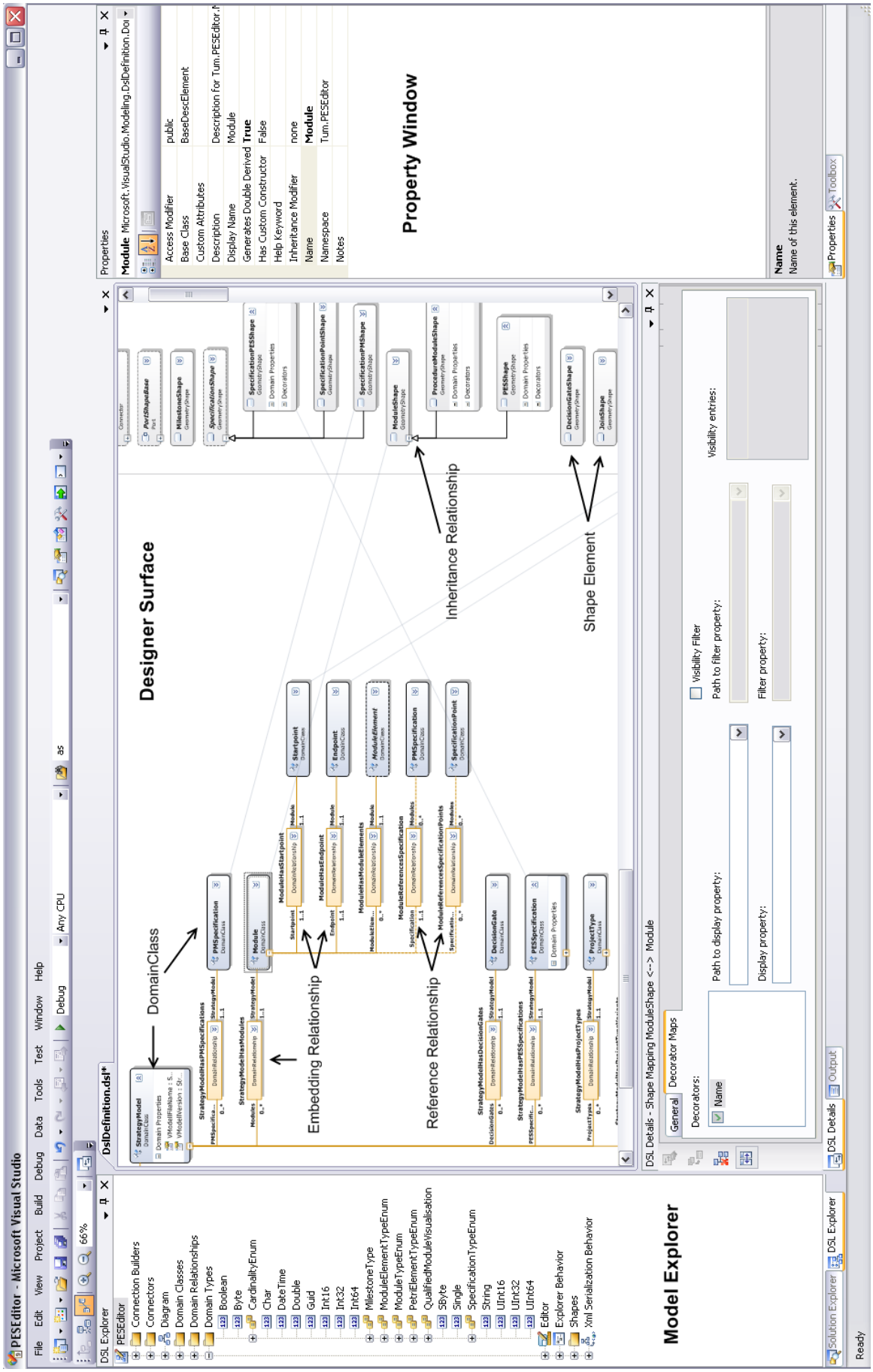

Abbildung 2.2: Visual Studio IDE zum DSL Design

2.3 Transformation, Validierung und Serialisierung

Nach der Definition des Domänenmodells sowie der grafischen Repräsentation, wird diese Information automatisch in Quellcode übersetzt. Hierzu setzen die DSL-Tools das *Text Templating Transformation Toolkit* (Kurz: T4, [Syc07]) ein. Bei der Übersetzung werden die oben genannten Eigenschaften berücksichtigt, sodass 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.

Hinweis

Ein weiterer interessanter Aspekt der automatischen Generierung ist der, dass sich eigene T4-Templates schreiben lassen, die wie die vordefinierten DSL-Tools-Templates Quellcode direkt aus dem DSL-Tools Domänenmodell erstellen.

Ferner lassen sich bereits vorhandene DSL-Tools T4-Templates, die dem automatischen Erzeugen von Quellcode aus dem Domänenmodell dienen, auch anpassen, um somit grundlegende Veränderungen zu erreichen und gewünschte Vorgänge zu realisieren.

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 s.g. *Soft-* und *Hard Constraints*. Die ersten sind Auflagen die zeitweise gebrochen werden können bzw. 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 Domänenmodell eine eigene Serialisierungsklasse definiert, die für das Speichern und Laden der jeweiligen Eigenschaften zuständig ist.

Hinweis

Eigene Serialisierungsklassen werden sowohl für die DomainClass als auch für DomainRelationship und die Shape-Elemente generiert.

Bei der Serialisierung muss beachtet werden, dass zwei Teilmodelle zu serialisieren sind. Zum einen das Domänenmodell 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 automatischen generierten Quellcode übernommen werden.

Mehr Informationen zu den DSL-Tools kann der interessierte Leser [CJKW07] entnehmen.

3 Modellierung und Prüfung von Projektabläufen

In diesem Kapitel befassen wir uns mit der domänenspezifischen Sprache zur Modellierung und Prüfung von Projektabläufen. Hierzu werden wir die V-Modell XT Strukturen betrachten, die zur Definition und Konfiguration von Projektdurchführungsstrategien notwendig sind und eine Möglichkeit vorstellen, wie diese Strukturen im Domänenmodell der DSL-Tools umgesetzt werden können. Ferner stellen wir eine Modellierungsmöglichkeit vor, um mithilfe der hier entwickelten DSL Projektpläne abzuleiten.

Übersicht

3.1 Das Domänenmodell

Das Domänenmodell orientiert sich in großen Teilen am Metamodell des V-Modells [TK09]. Jedoch sind an einigen Stellen Änderungen bzw. Ergänzungen erforderlich, um alle an die DSL gestellten Anforderungen zu erfüllen. Insbesondere da die Metamodellkonstrukte des V-Modells im Wesentlichen nur Strukturen definieren, jedoch für Fähigkeiten wie eine Validierung weitere Ausdrucksmittel erforderlich sind. In diesem Kapitel führen wir daher detailliert in das Domänenmodell (Abbildung 3.1) ein und stellen durchgehend den Bezug zum V-Modell XT Metamodell her.

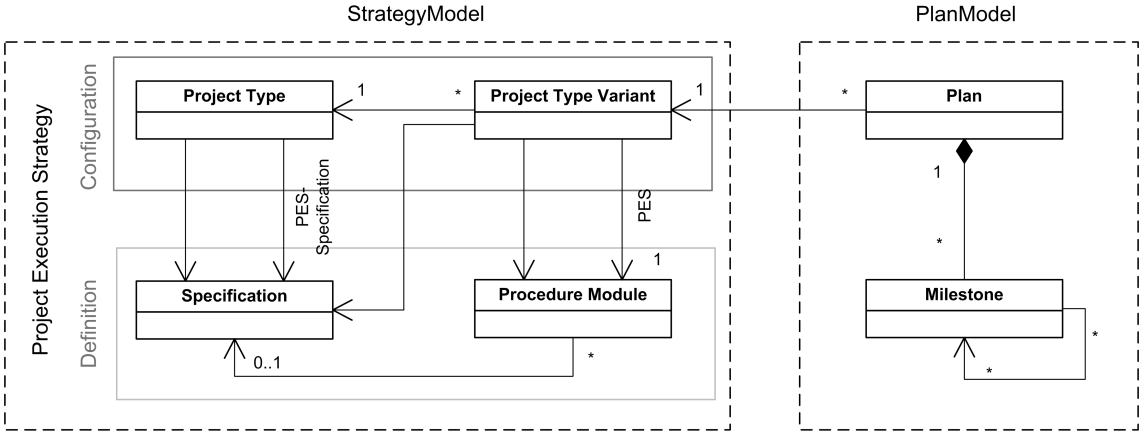

Abbildung 3.1: Domänenmodell der DSL

Weiterhin stellen wir in diesem Kapitel auch die Verknüpfung des Domänenmodells mit dem Planmodell vor. Diese Verknüpfung gewährleistet, dass sich aus einer modellierten Projektdurchführungsstrategie (unabhängig davon, ob diese aus dem „original" V-Modell stammt oder mithilfe der DSL definiert wurde) ein Projektplan ableiten lässt. Der Fokus im Rahmen der Modellierung liegt dabei auf den Definitionselementen (siehe Abbildung 3.1, links unten) sowie der Verknüpfung der Teilmodelle. Die Konfiguration einzelner Ablaufbausteine etc. im Rahmen von Projekttypvarianten betrachten wir indes nicht. Dies ist den weiterführenden Arbeiten (Kapitel 5) vorbehalten.

Die Darstellung der entsprechend relevanten Metamodellelemente ist aus [TK09] entnommen und stellt die technische Realisierung des Metamodells als XML-Schema dar. Analog entwerfen wir auch das Domänenmodell zunächst technisch orientiert. Der XML-Schemastruktur stehen jeweils die Modellelemente des Visual Studio DSL-Designers gegenüber.

3.1.1 Teilmodelle des Domänenmodells

Bei der Konzeption des Domänenmodells stellt sich zum einen die Frage nach der Umsetzung der Strukturen der Projektdurchführungsstrategien und zum anderen die Frage nach der Einordnung der Modellierungsstrukturen für Projektpläne in die entwickelte DSL.

Grundlegend orientieren sich dabei die Strategien am V-Modell, bedürfen allerdings einiger Anpassungen bzw. Erweiterungen um unsere Anforderungen erfüllen zu können. Die Definition von Projektplänen ist im V-Modell Metamodell hingegen gar nicht vorgesehen, sodass hier in der Gesamtheit unser Domänenmodell inkompatibel zum V-Modell ist. Hierdurch wird eine Aufteilung des Gesamtmodells auf zwei Teilmodelle notwendig, um einerseits die Modellierung von Strategien immer noch in Bezug zum V-Modell Metamodell stellen zu können und um andererseits die Erstellung von Projektplänen frei und sinnvoll zu gestalten. Folglich besteht das Domänenmodell unserer DSL aus zwei Teilmodellen:

StrategyModel Das *StrategyModel* dient der Definition und Konfiguration der Projektdurchführungsstrategien.

PlanModel Das *PlanModel* weist die notwendigen Strukturen zur Definition von Projektplänen auf.

Die Einbettung der beiden Teilmodelle ins Gesamtmodell lässt sich der Abbildung 3.2 entnehmen (*PESEditor* steht hier für den Namen des Prototyps).

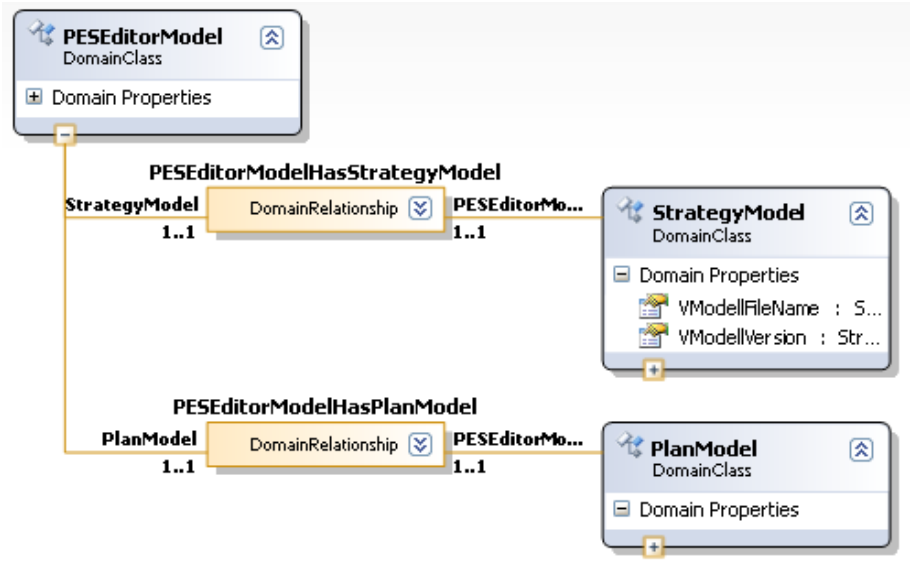

Abbildung 3.2: DSL Domänenmodell – Überblick

Das *StrategyModel* stellt die zentrale Schnittstelle zum V-Modell dar. Es stellt den Container dar, in dem die für die Modellierung von Projektdurchführungsstrategien (i.S. Ablaufbausteinen, Projekttypvarianten etc.) relevanten DSL-Elemente eingeordnet sind.

3.2 Projektdurchführungsstrategien

Das *StrategyModel* basiert auf den Metamodellstrukturen des V-Modell XT [TK09] für die Modellierung und Konfiguration der Projektdurchführungsstrategien. Im Folgenden wollen wir zunächst diese Strukturen vorstellen, um daraufhin die Implementierung derselben im DSL-Tools Editor zu präsentieren.

3.2.1 Das V-Modell XT Metamodell – Paket: Dynamik

Zur Definition von Projektdurchführungsstrategien stellt das Metamodell des V-Modell als Hauptelement einen *Ablaufbaustein* (Abbildung 3.3) 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 (siehe auch Kapitel 1.1). 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*:

Ablaufbausteinpunkt Ein Ablaufbausteinpunkt stellt eine Station im Ablauf des jeweiligen Ablaufbausteins dar. Er verweist auf genau eine Ablaufbausteinspezifikation, sodass an dieser Stelle weitere Ablaufbausteine integriert werden können, vorausgesetzt, sie erfüllen die referenzierte Ablaufbausteinspezifikation. Die Verwendung von Ablaufbausteinpunkten erlaubt 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, sodass so die Entscheidungspunkte in eine Projektdurchführungsstategie integriert werden. Im Gegensatz zu Ablaufbausteinpunkten, stellen Ablaufentscheidungspunkte *konkrete* Stationen im Ablauf einer Projektdurchführungsstategie 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, sodass 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.

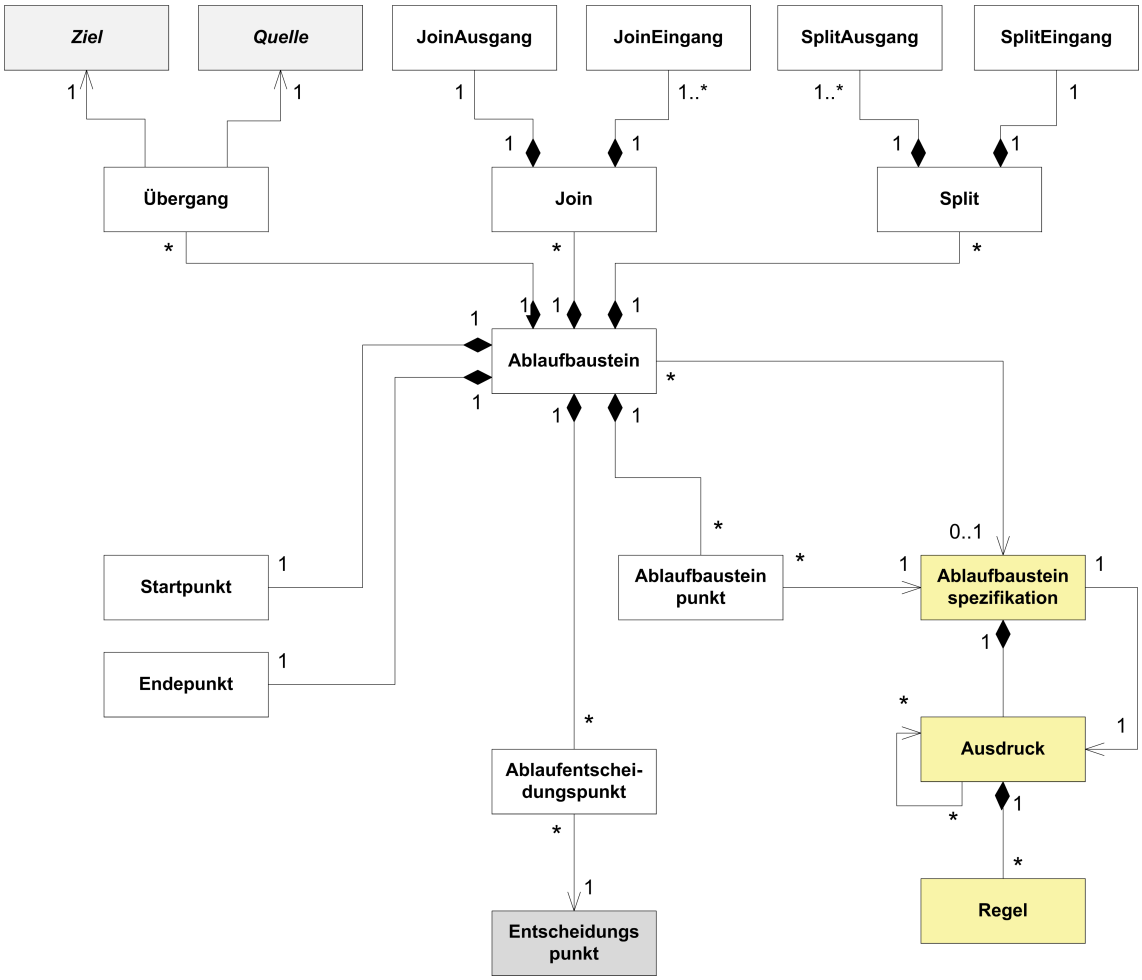

Abbildung 3.3: Modell zur Definition von Projektdurchführungsstrategien

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

3.2.2 Das V-Modell XT Metamodell – Paket: Anpassung

Wie in Kapitel 1.1 dargestellt, genügt es nicht, nur Ablaufbausteine für eine Projektdurchführungsstrategie zu definieren. Diese müssen entsprechend im Rahmen des Tailorings *konfiguriert* werden (Abbildung 3.4), um unzulässige bzw. ungewollte Abläufe auszuschließen. 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

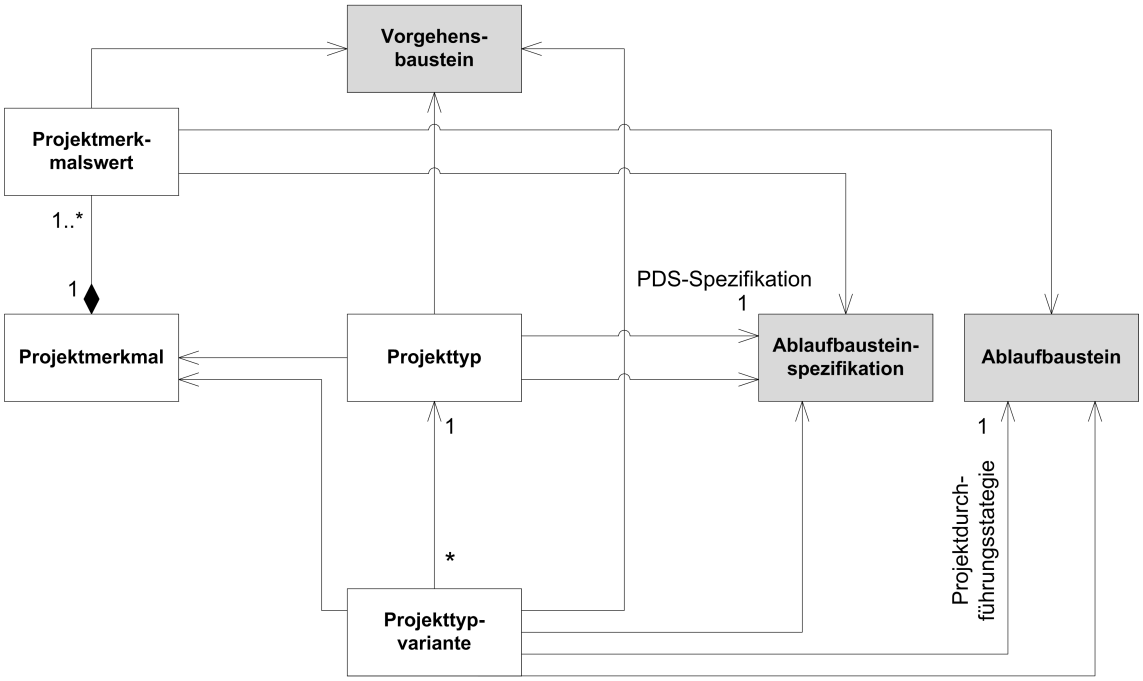

Abbildung 3.4: Metamodell – Sicht: Anpassung (Tailoring)

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 Ablaubausteinspezifikationen sowie die konkreten Ablaufbausteine der Strategie fest.

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

Abbildung 3.5 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.

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*.

3.2.3 Validierung

Das oben vorgestellte Modell erlaubt die Definition und Konfiguration von Projektdurchführungsstrategien, muss allerdings um Erweiterungen sowie Einschränkungen ergänzt werden, mit deren Hilfe die Validität des Modells sichergestellt werden kann. Diese werden später durch die unterschiedlichen Constraint-Arten bei der Validierung in den DSL-Tools erfüllt, sodass wir an dieser Stelle auch angeben mit welcher Constraint-Art (siehe Kapitel 2.3) sich das jeweils implementieren lässt:

Hard Constraint 3.1:
Ein Übergang ausgehend von einem Startpunkt kann nie einen JoinEingang als Ziel haben, da

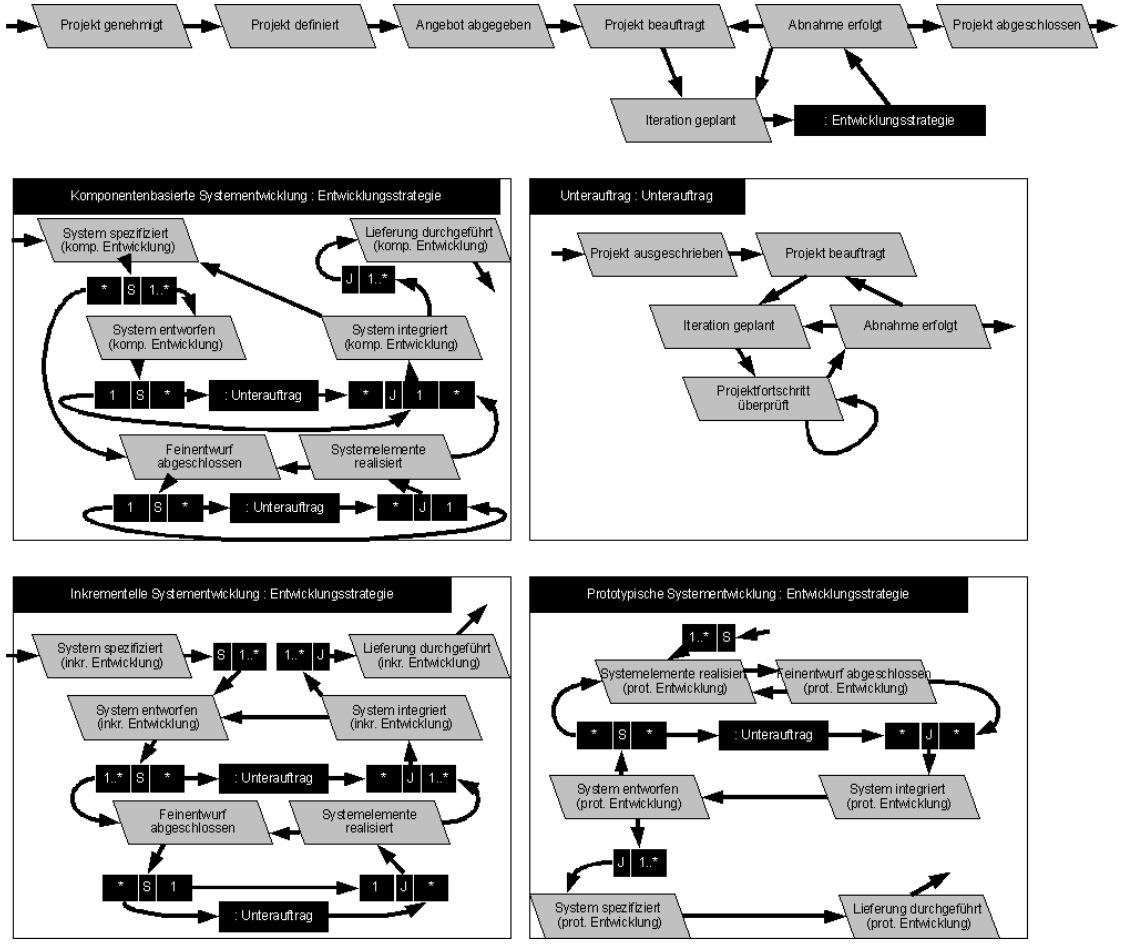

Abbildung 3.5: Projektdurchführungsstrategie des Auftragnehmers (Entwicklung)

noch keine parallelen Abläufe festgelegt wurden (oder: vor einem JoinEingang muss ein Split-Ausgang stehen).

Hard Constraint 3.2:
Ein Übergang eingehend zu einem Endpunkt kann nie einen SplitAusgang als Quelle haben, da auf diese Art und Weise parallele Abläufe festgelegt werden würden, die nie vereinigt werden könnten.

Hard Constraint 3.3:
Übergänge zwischen JoinAusgängen und JoinEingängen sowie SplitAusgängen und SplitEingängen derselben Join- und Split-Elemente können nicht erlaubt werden, da hierdurch unendliche Abläufe ohne definiertes Projektende festgelegt werden würden.

Hard Constraint 3.4:
Übergänge zwischen Ablaufpunkten unterschiedlicher Ablaufbausteine dürfen nicht zugelassen werden.

Soft Constraint 3.1:
Ein SplitAusgang kann nie ohne den dazugehörigen JoinEingang existieren und umgekehrt.

Soft Constraint 3.2:
Die Kardinalitäten paralleler Abläufe müssen konsistent zueinander sein.

Soft Constraint 3.3:
Ein Ablaufbausteinpunkt kann nie dieselbe Ablaufbausteinspezifkation referenzieren wie der Ablaufbaustein indem er enthalten ist, denn das würde eine unendliche Verschachtelungstiefe definieren.

Soft Constraint 3.4:
Jeder Pfad durch einen Ablaufbaustein muss möglich sein, d.h. er muss mit dem Startpunkt beginnen und beim Endpunkt ankommen.

Die oben genannten Punkte lassen sich über das Validierungssystem der DSL-Tools implementieren. Weitere Informationen hierzu finden sich in Kapitel 4.4.

3.2.4 Strukturabbildung

Wir haben nun die Strukturen des *StategyModels* basierend auf dem V-Modell vorgestellt. Im Folgenden nehmen wir die tatsächliche Abbildung der wesentlichen XML-Strukuren des V-Modell Metamodells in das DSL-Tools Domänenmodell vor. Dazu betrachten wir zunächst die Abbildung der Attribute (siehe Abbildung 3.6), die jedem Element des Metamodells zugrunde liegen. Diese sind:

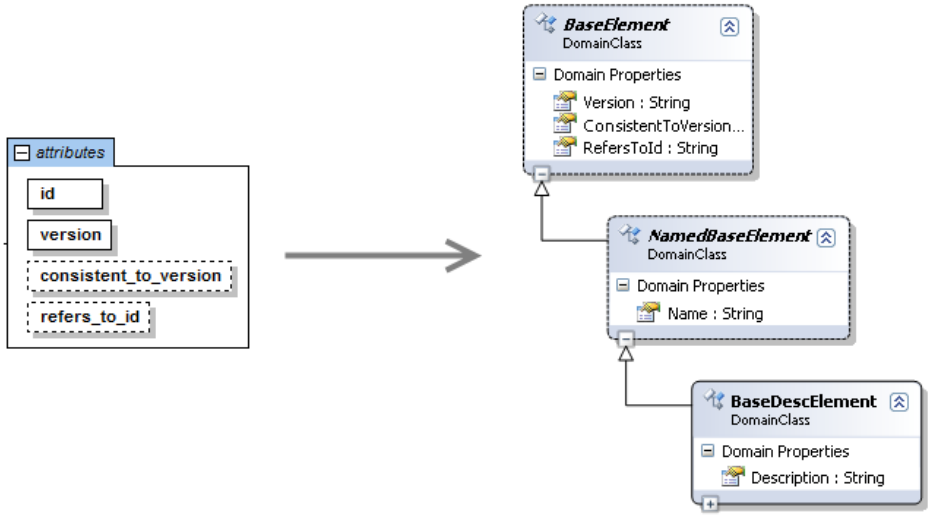

Abbildung 3.6: Abbildung der Attribute auf Klassen

- *id*
- *version*
- *consistent_to_version*
- *refers_to_id*

Die letzten drei werden über editierbare Eigenschaften der Klasse *BaseElement* eingebunden, die jedem anderen Element, das aus dem V-Modell Metamodell stammt, von nun an als Basisklasse dient. Die *id*-Eigenschaft wird über ein bereits existierendes Identifikationsfeld des DSL-Tools Elements belegt. Weiterhin sind in der Abbildung 3.6 noch die Klassen *NamedBaseElement* und *BaseDescElement* zu sehen. Diese erweitern die Basisklassen um Eigenschaften zur Definition des Namens und der Beschreibung und werden von weiteren Elementenklassen verwendet.

Hinweis: Durch die Verwendung mehrerer Basisklassen, die jeweils Erweiterungen über unterschiedliche Eigenschaften mitbringen, wird hier eine Art Wiederverwendung modelliert, die es beispielsweise erlaubt, benutzerdefinierte Editoren für spezifische Eigenschaften nur an einer einzelnen Stelle zu definieren und im Modell zu verwenden.

Zur Definition und Konfiguration von Projektdurchführungsstrategien müssen im Metamodell im Wesentlichen die folgenden Elemente mit ihren jeweiligen Eigenschaften und Unterelementen berücksichtigt werden:

All diese Elemente sind dem Hauptknoten im Domänenmodell untergeordnet, wie in Abbildung 3.7 darstellt ist.

Ablaufbaustein. Das Hauptelement zur Modellierung von Projektdurchführungsstrategien ist durch einen Ablaufbaustein gegeben, dessen Implementierung basierend auf der Struktur des Metamodells der Abbildung 3.8 entnommen werden kann.

Jeder Ablaufbaustein im Metamodell hat die Eigenschaften *Name* und *Beschreibung*, sodass wir *Module* von *BaseDescElement* ableiten, um diese Eigenschaften bereitzustellen. Weiterhin wird die

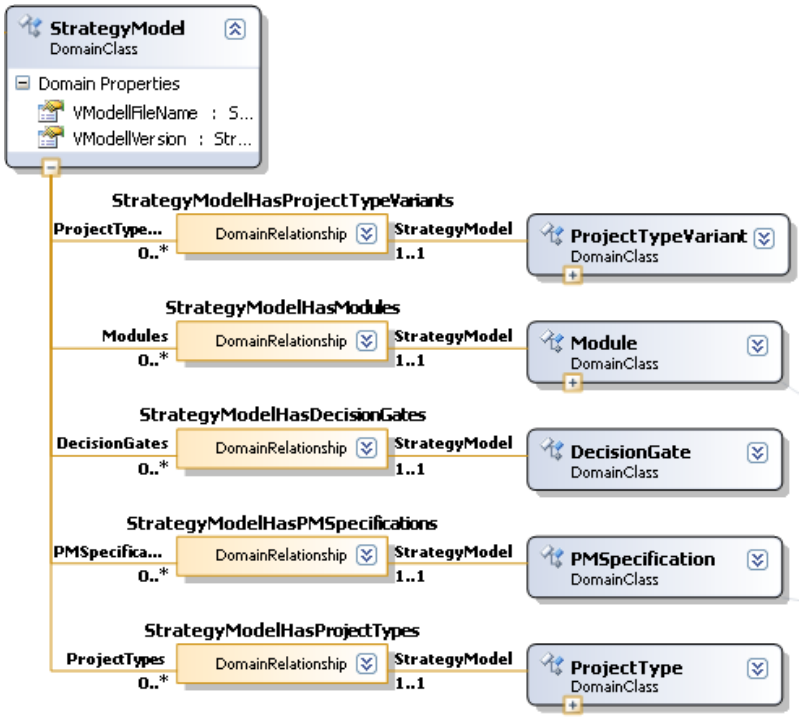

Abbildung 3.7: Einordnung der Hauptelemente im StrategyModel

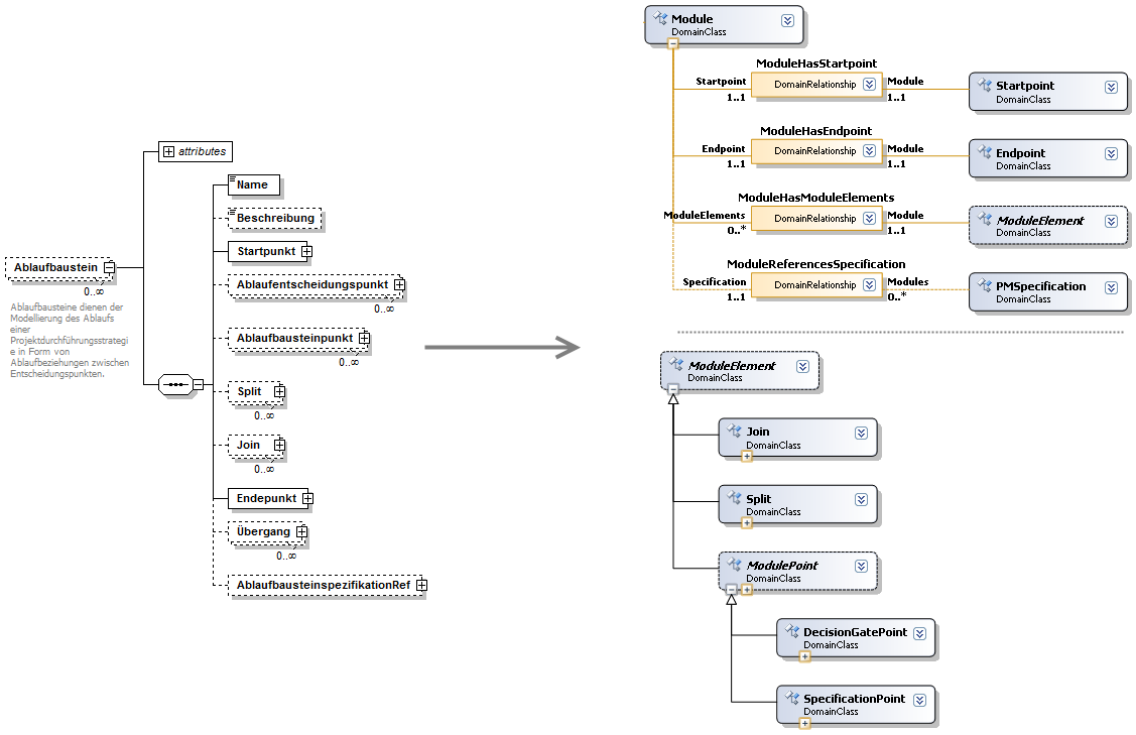

Abbildung 3.8: Implementierung der Ablaufbaustein-Struktur im StrategyModel

Name im V-Modell XT Metamodell	Zugehörige Name im Domänenmodell
Ablaufbaustein	Module
Ablaufbausteinspezifikation	PMSpecification
Entscheidungspunkt	DecisionGate
Projekttyp	ProjectType
Projekttypvariante	ProjectTypeVariant

Tabelle 3.1: Zuordnung der Elementennamen bei der Strukturabbildung

Referenz von genau einer Ablaufbausteinspezifikation über die Beziehung *ModuleReferencesSpecification* implementiert. Die Abbildung der einzelnen Elemente aus dem Metamodell ist wie folgt gegeben (siehe Tabelle 3.2):

- Der Startpunkt bzw. der Endpunkt werden durch *Startpoint* bzw. *Endpoint* implementiert und durch die $1:1$-Beziehungen *ModuleHasStartpoint* sowie *ModuleHasEndpoint* beim *Module* hinterlegt, die gleichzeitig festlegen, dass jede Instanz der Klasse *Module* immer genau einen *Startpoint* und genau einen *Endpoint* haben muss.
- Die Ablaufpunkte Join, Split, Ablaufentscheidungspunkt (*DecisionGatePoint*) und Ablaufbausteinpunkt (*SpecificationPoint*) sind jeweils von der abstrakten Klasse *ModuleElement*, die ihrerseits von *BaseLinkElement* (siehe unten) abgeleitet ist, und werden über die Beziehung *ModuleHasModuleElements* einem *Module* zugeordnet.

Eigenschaft/Unterelement im V-Modell XT	Zugehörige Abbildung im Domänenmodell
Name und Beschreibung	über die Basisklasse *BaseDescElement*
Startpunkt	Startpoint
Endepunkt	Endpoint
Ablaufentscheidungspunkt	DecisionGatePoint
Ablaufbausteinpunkt	SpecificationPoint
Split	Split
Join	Join
Übergang	BaseConnection (siehe unten)
AblaufbausteinspezifikationRef	ModuleReferencesSpecification

Tabelle 3.2: Zuordnung der Elemente und Eigenschaften des Ablaufbausteins bei der Strukturabbildung

Als nächstes sind die Übergänge zwischen den einzelnen Ablaufpunkten umzusetzen. Hierzu wird das Basismodell um eine neue Basisklasse *BaseLinkElement* erweitert, die jeweils als Quelle und Ziel für die Übergänge dienent. Zwischen *BaseLinkElements* wird eine neue Referenzbeziehung eingeführt und mit den entsprechenden Eigenschaften versehen, die bei einem Übergang berücksichtigt werden müssen (siehe Abbildung 3.9).

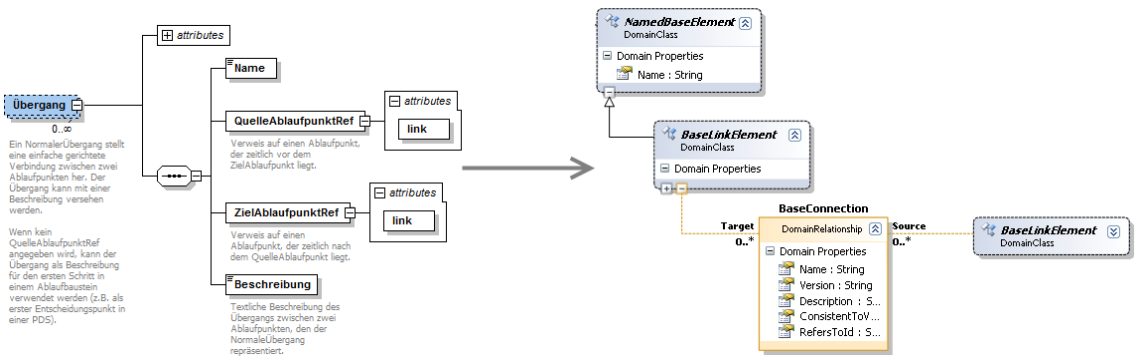

Abbildung 3.9: Implementierung von Übergängen im StrategyModel

Ablaufbaustein- und Ablaufentscheidungspunkte. Diese stellen Ablaufpunkte mit teilweise gleichartigen Elementen dar (siehe Abbildung 3.10). Sie sind von der Klasse *ModulePoint* abgeleitet, die ihrerseits *BaseLinkElement* als Basisklasse hat, so dass beide Ablaufpunkte bei Übergängen berücksichtigt werden müssen.

Eigenschaft/Unterelement im V-Modell XT	Zugehörige Abbildung im Domänenmodell
Name	über die Basisklasse *BaseLinkElement*
AblaufbaustreinspezifikationRef	SpecificationPointReferencesSpecification
EntscheidungspunktRef	DecisionGatePointReferencesDecisionGate
DokFolgtNachAblaufpunktRef	über die Eigenschaften von *ModulePoint*

Tabelle 3.3: Zuordnung der Elemente und Eigenschaften der Ablaufbaustein- und Ablaufentscheidungspunkte bei der Strukturabbildung

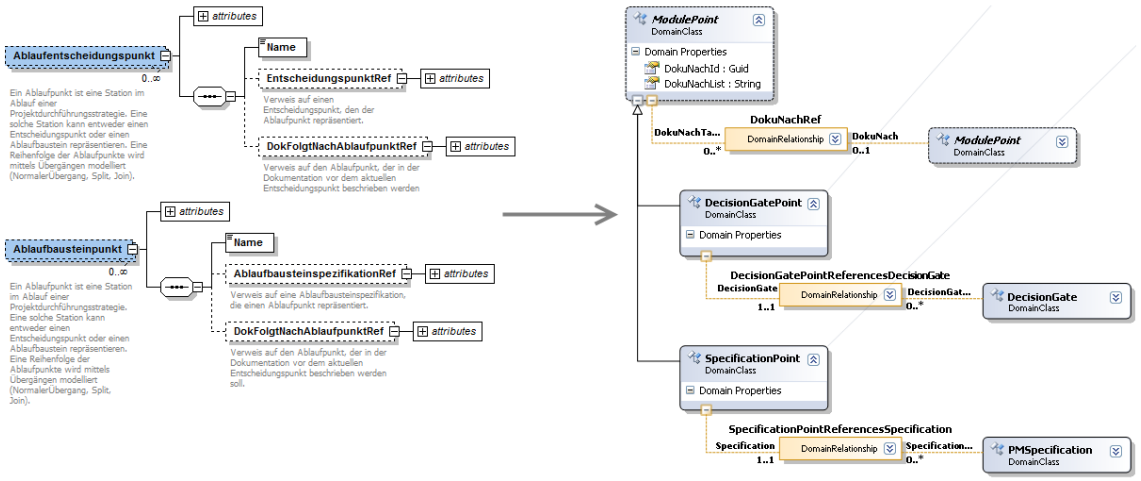

Abbildung 3.10: Implementierung der Ablaufbaustein- und Ablaufentscheidungspunkte im StrategyModel

Die Ablaufpunkte implementieren ferner das Dokumentationskonzept (für nähere Informationen siehe [TK09]), welches im Domänenmodell über die abstrakte klasse *ModulePoint* verwirklicht wird. Dabei werden die Referenzen *DokFolgtNachAblaufpunktRef* aus dem Metamodell durch die Referenzbeziehung *DokuNachRef* dargestellt. Die zusätzlichen Eigenschaften der Klasse *ModulePoint* dienen der Darstellung der *DokuNach*-Eigenschaft im Eigenschaftenfenster des Editors.

Die Referenzen zwischen einem Ablaufbausteinpunkt und einer Ablaufbausteinspezifikation sowie zwischen einem Ablaufentscheidungspunkt und einem Entscheidungspunkt werden mithilfe der Referenzbeziehungen *SpecificationPointReferencesSpecification* bzw. *DecisionGatePointReferencesDecisionGate* implementiert (siehe auch Tabelle 3.3)

Splits. Splits werden über die gleichnamige Klasse *Split* implementiert. Der SplitEingang und die SplitAusgänge werden über eigenen Klassen *SplitInPort* bzw. *SplitOutPort* mit den jeweiligen Eigenschaften, die durch die Ableitung von der Basisklasse *BaseLinkElement* zugeordnet werden, im Domänenmodell dargestellt (siehe Abbildung 3.11). Sie werden dem zugehörigen *Split* über die Beziehungen *SplitHasSplitInPort* und *SplitHasSplitOutPorts* zugeteilt (siehe auch Tabelle 3.4).

Eigenschaft/Unterelement im V-Modell XT	Zugehörige Abbildung im Domänenmodell
Name	über die Basisklasse *BaseLinkElement*
Beschreibung	Description
SplitEingang	SplitInPort
SplitAusgang	SplitOutPort

Tabelle 3.4: Zuordnung der Elemente und Eigenschaften des Splits bei der Strukturabbildung

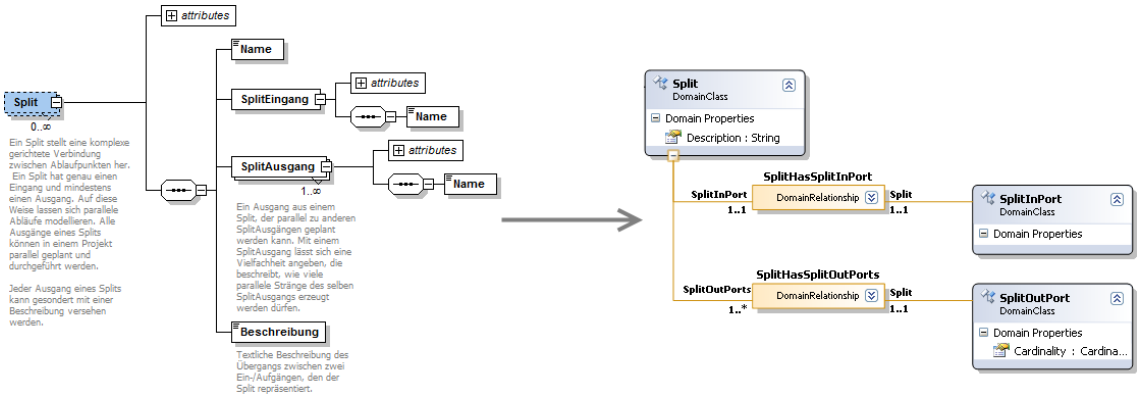

Abbildung 3.11: Implementierung der Splits im StrategyModel

Beim *SplitOutPort* 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 *SplitInPorts* bzw. der *SplitOutPorts* an dieser Stelle ist, dieselben später über die *Port Shapes* darzustellen (siehe Kapitel 2.2), so dass sich die Übergänge bequem über die Ports visualisieren lassen und die Kardinalität der parallelen Abläufe beim *SplitOutPort* sich auch grafisch anzeigen lässt.

Joins. Joins werden ähnlich zu den Splits in das Domänenmodell übertragen (siehe Abbildung 3.12). Hierzu werden auch eigene Klassen für die JoinEingänge (*JoinInPort*) und den JoinAusgang (*JoinOutPort*) angelegt, die jeweils von der Basisklasse *BaseLinkElement* abgeleitet sind. Hierbei wird die Kardinalität der möglichen parallelen Prozesse, die vereinigt werden können, beim *JoinInPort* hinterlegt (siehe auch Tabelle 3.5).

Eigenschaft/Unterelement im V-Modell XT	Zugehörige Abbildung im Domänenmodell
Name	über die Basisklasse *BaseLinkElement*
Beschreibung	Description
JoinEingang	JoinInPort
JoinAusgang	JoinOutPort

Tabelle 3.5: Zuordnung der Elemente und Eigenschaften des Joins bei der Strukturabbildung

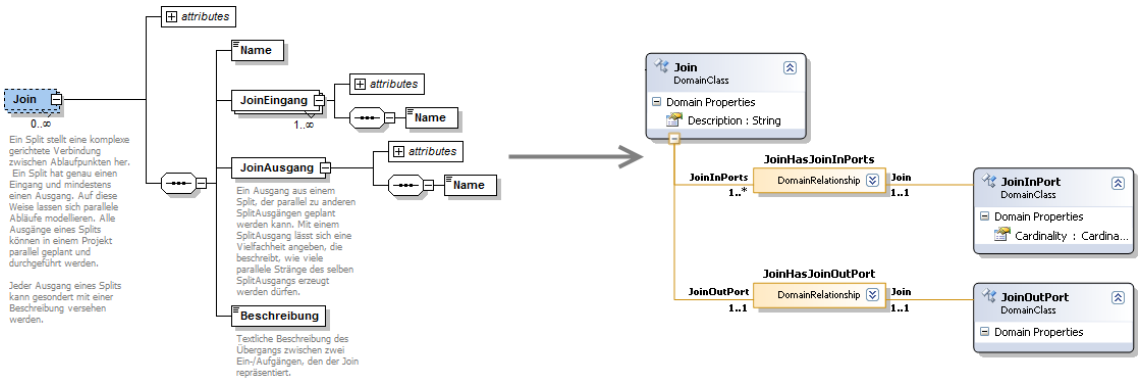

Abbildung 3.12: Implementierung der Joins im StrategyModel

Auf diese Weise werden die Strukturen eines Ablaufbausteins aus dem V-Modell Metamodell in das DSL-Tools Domänenmodell abgebildet. Die Abbildung der restlichen Elemente erfolgt analog, sodass an dieser Stelle auf die Implementierung verwiesen und nicht weiter darauf eingegangen wird.

3.3 DSL für Projektpläne

Ein Projektplan (Abbildung 3.13) besteht aus unserer Sicht aus mehreren Meilensteinen und den dazwischen definierten Übergängen. Ein Übergang, modelliert als *MilestoneFlow*, steht für eine sequentielle Anordnung der Meilensteine und legt folglich insgesamt die Reihenfolge der jeweiligen Meilensteine im Projektplan fest. Ein Meilenstein verfügt über die Eigenschaften *Name*, *Date* und *Typ*, wobei durch den Typ festgelegt wird, ob der jeweilige Meilenstein als Start-, Stopp- oder eine normale Durchgangsstation im Plan zu verstehen ist. Weiterhin beinhaltet ein Meilenstein eine Referenz zu einem Entscheidungspunkt (*MilestoneReferencesDecisionGate*), sodass der Gesamtplan auch als eine Abfolge von Entscheidungspunkten gesehen werden kann.

Um die Verbindung zwischen einem Plan und der dazugehörigen Strategie herzustellen, wurde eine Referenz zwischen einem Plan und einer Projekttypvariante hinterlegt (*PlanReferencesProjectTypeVariant*). Insofern verweist ein Plan auf eine bereits konfigurierte Strategie.

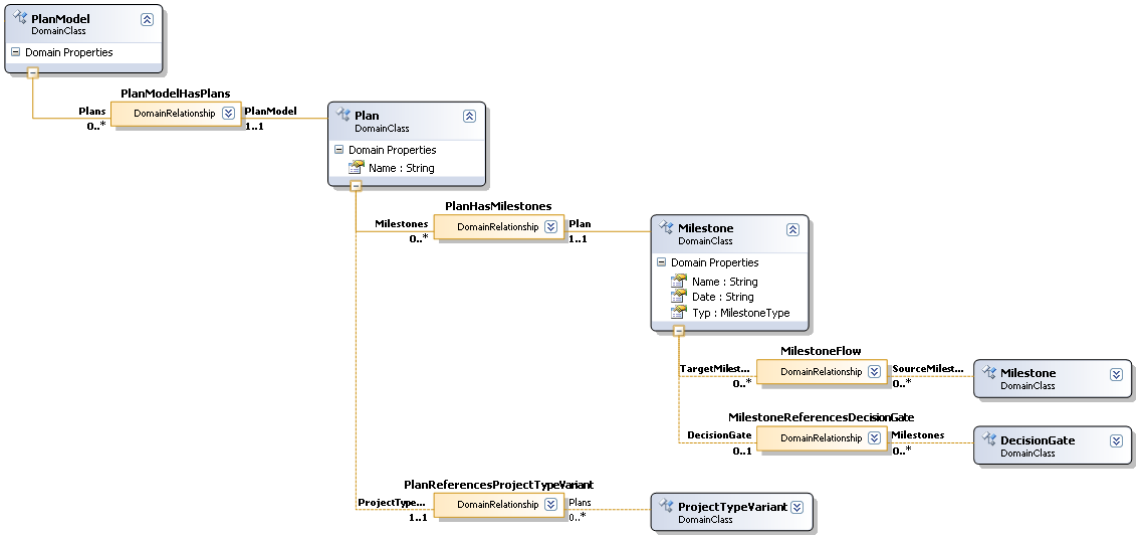

Abbildung 3.13: DSL PlanModel

Die oben vorgestellte Modellierung von Projektplänen erlaubt es, Überprüfungsalgorithmen zu definieren, die einen Plan bezüglich der zugrunde liegenden Strategie validieren und folglich feststellen, ob ein Plan aus der entsprechenden Strategie ableitbar ist. Dies wird in Kapitel 4.4 dargestellt.

4 Prototyp

In diesem Kapitel betrachten wir die Implementierung des Prototyps, der auf dem Domänenmodell (siehe Kapitel 3) aufbaut, und stellen dabei die relevanten Implementierungsaspekte vor. Dazu geben wir zunächst einen Überblick über die erreichte Funktionalität (siehe Abschnitt 4.1), um folgend die entscheidenden Aspekte (siehe Abschnitt 4.2 – Abschnitt 4.4) im Detail zu thematisieren. Insbesondere gehen wir dabei auf die Validierung ein. Abschließend gehen wir noch auf die DSL-Tools selbst ein (siehe Abschnitt 4.5) und fassen die während der Implementierung gewonnenen Erfahrungen mit diesen Werkzeugen zusammen.

Übersicht

4.1 Überblick

Der Prototyp (*PESEditor*) für die Modellierung von Projektdurchführungsstrategien sowie von Projektplänen baut auf dem zuvor vorgestellten Domänenmodell auf, welches mit dem DSL-Tools Designer implementiert wurde. Der Prototyp erweitert diese Definition vor allem über die Modifikation von Quellcode.[1] Im Quellcode wird das *Verhalten* des Modells definiert und damit die gewünschte Funktionalität umgesetzt. Überblicksweise wollen wir nun die wesentlichen Implementierungsaspekte darstellen; einige hiervon werden folgend ausführlicher besprochen.

Der Prototyp ist in der Lage, das gesamte V-Modell Datenmodell zu laden und zu speichern, was durch eine eigene Implementierung der Serialisierung erreicht wurde. Im DSL-Designer führt dies dazu, dass automatisch alle Elemente, die hierbei instanziiert werden und über ein *ShapeMapping* verfügen, auch gleichzeitig automatisch auf der Designeroberfläche grafisch dargestellt werden. Das ist aber offensichtlicherweise nicht optimal, denn hierdurch wird jegliche Übersicht einer grafischen Visualisierung durch die große Anzahl der grafischen Elemente zunichte gemacht. Deshalb musste zur Lösung dieser Problematik eine Selektion in der grafischen Darstellung erfolgen, wodurch das Datenmodell in seiner Gesamtheit nur noch im *ModelExplorer* dargestellt wird (näheres hierzu in Abschnitt 4.2).

Durch die Darstellung des Gesamtmodells im ModelExplorer musste eine Möglichkeit gegeben werden, die vorhandenen Elemente selektiv auf der Designeroberfläche darzustellen. Dazu lassen sich Drag&Drop-Funktionalitäten einführen, wie sie im Anhang C vorgestellt werden.

Die grafische Darstellung der Elemente, wie sie durch die DSL-Tools definiert werden, beschränkt sich auf einige einfache grafische Formen. Für die modellspezifische Visualisierung von Elementen, die der Erwartung des Anwenders entspricht, müssen einige grundlegende Methoden eingeführt und überschrieben werden (siehe Anhang B). Da die Darstellung der Elemente nur selektiv ist, musste weiterhin ein Layoutmanager erstellt werden, der sich die entsprechende Anordnung der Elemente innerhalb eines Ablaufbausteins (*Module*) merkt und gegebenenfalls auch wiederherstellt. (Abschnitt 4.3 vertieft diese Thematik).

Ein wesentlicher Aspekt des Prototyps ist die Validierung. Hierbei müssen, wie Eingangs bei der Vorstellung der DSL-Tools erwähnt, *Soft* und *Hard Constrains* berücksichtigt werden. Diese werden anhand der Definition aus Kapitel 3.2.3 implementiert (siehe Abschnitt 4.4).

Die Darstellung von Projektdurchführungsstrategien sowie von Projektplänen bedarf einer gesonderten Visualisierung derselben, die zumindest bereichsweise erfolgen muss. Hierzu führen wir unterschiedliche Diagramme ein, sodass sich pro Darstellungsbereich jeweils ein Diagramm findet. Diese Vorgehensweise basiert auf [RG09] und wird dort auch eingehend vorgestellt.

Schließlich wurde eine einfache Form des Copy&Paste eingeführt, die es erlaubt bereits vorhandene Ablaufbausteine oder auch einzelne Ablaufelemente zu kopieren (hierbei werden alle Eigenschaften kopiert und mit neuen IDs versehen). Hierdurch wird eine einfache Möglichkeit gegeben, einzelne Ablaufbausteine basierend auf bereits vorhanden zu erweitern, ohne dass die entsprechenden Ablaufbausteine, die als Basis genutzt werden, selbst verändert werden müssen.

4.2 Einbindung der Daten aus dem V-Modell XT

Bei der Verwendung des Prototyps müssen die Modelldaten sowie die grafischen Informationen geladen werden. Die Zuordnung zwischen Modelldatei und dem jeweiligen Modellaspekt geschieht dabei über Dateiendungen:

- *Model Xml File (.pes)*: Diese Datei steht für die Identifikation der Zugehörigkeit zum Prototyp. Sie enthält eine Referenz zu der zugehörigen V-Modell Datendatei sowie die serialisierten Datensätze vom *PlanModel*.
- *Diagram Xml File (.diagram)*: In dieser Datei werden die grafischen Informationen gespeichert, die zur Darstellung der Elemente benötigt werden, die sich *aktuell* auf der Designeroberfläche befinden.

Nachfolgend wollen wir die Aufteilung der Serialisierungsinformation auf diese beiden Dateien näher vorstellen.

1 Wie schon eingangs in Kapitel 2.3 erwähnt wird, dient die definierte Information im DSL-Tools Designer als Vorlage für die anschließende Transformation in den Quellcode.

Hinweis: Die DSL-Tools verwenden standardmäßig zur Serialisierung genau zwei Dateien, das *Model Xml File* und das *Diagram Xml File*. Die erste Datei ist für die Modelldaten sowie die Identifikation der Zugehörigkeit zum entsprechenden Editor vorgesehen, während die zweite die Daten der grafischen Repräsentation enthält. Die zum Laden und Speichern notwendigen Klassen werden hierbei für die einzelnen Elemente des Domänenmodells automatisch generiert.

Arbeiten auf bereits existierenden Daten. Bei der Implementierung des Prototyps hat sich die Frage gestellt, wie bereits bestehende Modelldaten geladen, verwaltet und wieder gespeichert werden können. Dabei gäbe es zunächst auch die Möglichkeit, das Serialisierungsmodell der DSL-Tools, welches jeweils eine Datei für die Daten und die grafische Repräsentation vorsieht, dahingehend zu erweitern, dass bestehende Daten importiert werden können. Das allerdings erfordert auch eine Exportfunktionalität, bei der Überprüfungsmaßnahmen hinsichtlich bereits definierter Elemente und deren Äquivalenz notwendig wären.

Ein für das V-Modell sinnvollerer Ansatz besteht in der Anpassung der Serialisierung derart, dass der erstellte Prototyp direkt auf der Instanz des Metamodells mit den vorhandenen Daten arbeiten kann. Dazu müssen entsprechende Funktionalitäten im DSL-Tool Framework überschrieben und angepasst werden (eine mögliche Vorgehensweise lässt sich dabei dem Anhang A entnehmen). Hierbei muss allerdings eine Referenz auf die V-Modell Quelldatei erhalten bleiben, sodass diese im *Model Xml File* zu setzen ist, welches nach wie vor zusätzlich als Identifikator der Zugehörigkeit zum Prototyp Verwendung findet.

Wie bereits erwähnt, ergibt sich aus dieser Funktionalität die Notwendigkeit einer selektiven grafischen Darstellung, da eine gleichzeitige Visualisierung aller vorhandenen Elemente nicht gewünscht ist und die Übersichtlichkeit stark einschränken würde (siehe 4.3).

Serialisierung von zusätzlichen, benutzerdefinierten Strukturen. Das Domänenmodell unseres Prototyps besteht nicht nur aus den Strukturen zur Modellierung von Projektdurchführungsstrategien (*StrategyModel*), deren Instanziierung in den Datensätzen des V-Modells zu finden ist, sondern zusätzlich aus den Elementen des *PlanModel*, die zur Definition von Projektplänen verwendet werden können. Diese Datensätze können einerseits natürlich nicht in der V-Modell Datei gespeichert werden und andererseits lassen sich die von den DSL-Tools automatisch generierten Serialisierungsklassen nicht mehr auf eine einfache Art und Weise verwenden, sodass an dieser Stelle eine eigene Implementierung die sinnvollste Lösung darstellt. Diese lässt sich über die Verwendung des *Model Xml File* implementieren, sodass dort die serialisierten Daten des *PlanModel* zu finden sind.

Auf diese Art und Weise lassen sich auch zusätzliche, benutzerdefinierte Erweiterungen des Domänenmodells hinsichtlich der Serialisierung einbinden. Gleichzeitig können so aber auch weitere Informationen gespeichert werden, die für spezifische Prozesse erforderlich sein könnten.

4.3 Benutzerdefinierte Darstellung der Elemente

Zur grafischen Darstellung der Elemente des Domänenmodells ist ein *ShapeMapping* notwendig, das auch mit dem DSL-Tools Designer vorgenommen werden kann. Bei unserem Prototyp mussten dabei unterschiedliche Shapes implementiert werden, die sich zunächst in zwei Kategorien einordnen lassen:

- *Hosting Shape*: Shape, das über Kindelemente verfügen kann und diese entsprechend in einem Container gruppiert.
- *Nested Shape*: Shape, welches als Kindelement einem *Hosting Shape* zugeordnet werden kann.

Grundsätzlich ist die Unterstützung der DSL-Tools für eine derartige Aufteilung nicht ausreichend, sodass an dieser Stelle einige Funktionalitäten angepasst werden mussten, um dieses Verhalten zu erreichen.

Beispiel: Ein Beispiel aus dem Prototyp für die Shapezuweisungen lässt sich der Abbildung 4.1 entnehmen. Dabei steht hier beispielsweise die grafische Darstellung des *Procedure Module* für ein *Hosting Shape*, während die Elemente innerhalb dieses *Procedure Module* über *Nested Shapes* visualisiert werden.

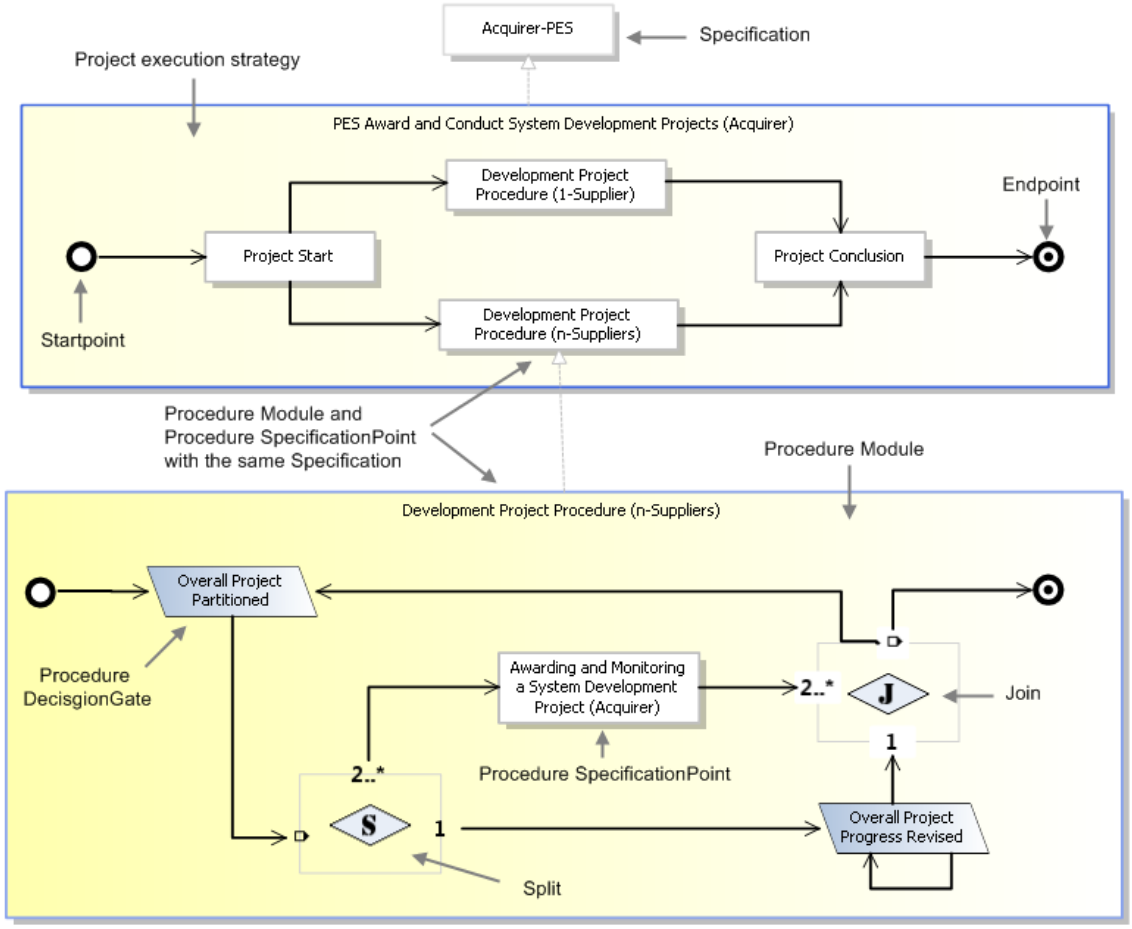

Abbildung 4.1: Darstellungselemente des Prototyps

Hinweis: Jedes instanziierte Element des Domänenmodells, welches über ein ShapeMapping verfügt, wird standardmäßig automatisch auf der Designeroberfläche grafisch dargestellt.

Selektive Visualisierung der Elemente. Das Laden aller vorhandenen Daten des V-Modells erzwingt eine Selektion in der grafischen Darstellung der Elemente auf der Designeroberfläche. Dieses folgt aus der großen Anzahl der Elemente und der Anforderung, eine bestmögliche Übersicht zu erreichen, sodass die einzelnen Strategien auch besser nachzuvollziehen sind. Deshalb wird die Instanz des Modells in seiner Gesamtheit auch nur noch im *Model Explorer* dargestellt und dem Anwender die Möglichkeit gegeben, einzelne Elemente selektiv grafisch zu visualisieren. Hierbei wurden folgende Aspekte umgesetzt:

- Das Standardverhalten der DSL-Tools bezüglich der Darstellung eines instanziierten Elements, welches über ein ShapeMapping verfügt, wurde angepasst, sodass nur noch Elemente dargestellt werden, die vom Anwender ausgewählt wurden.
- Zum Anlegen neuer Elemente wurden Drag&Drop-Funktionalitäten (siehe Anhang C) eingeführt, die das Einfügen ausgehend von der Visual Studio Toolbox abfangen und den Anlegevorgang implementieren (an dieser Stelle lässt sich der automatisch generierte Quellcode nicht mehr nutzen).
- Zur Erleichterung der Modifikation bzw. Modellierung von Strategien wurden zusätzlich Drag&Drop-Funktionalitäten (siehe Anhang C) ausgehend vom *Model Explorer* eingeführt. Diese erlauben bereits vorhandene *Procedure Module* grafisch auf der Designeroberfläche darzustellen.

Layout Manager für vorhandene Procedure Modules. Durch die Einführung der Selektion in der grafischen Darstellung hat sich die Frage gestellt, wie eine gute Lösung für das *Layouting* der

Procedure Modules gefunden werden kann. Grundsätzlich gibt es zwei Möglichkeiten:

Automatisches Layouting: Hierbei wird allerdings eine eigens hierfür geschriebene Engine benötigt, die zudem auch keine perfekten Ergebnisse liefern kann.

Manuelles Layouting: Die Veränderungen an der Anordnung innerhalb eines *Procedure Module* werden manuell vorgenommen und automatisch in einer separaten Datei abgespeichert, so dass sie folgend beim Darstellen der einzelnen *Procedure Modules* zur Verwendung kommen können.

Beim Prototyp haben wir uns für eine Variante entschieden, die beide genannten Layouting-Arten berücksichtigt. Zunächst benutzen wir automatisches Layouting (implementiert mit den DSL-Tools Methoden), so lange es keine benutzerdefinierte Darstellungsinformation für das entsprechende *Procedure Module* gibt. Diese wird in diesem Schritt auch automatisch erstellt und folglich zur Visualisierung durch den *Layout Manager* verwendet. Gibt es bereits benutzerdefinierte Darstellungsinformationen, so werden Methoden des *Layout Managers* aufgerufen, um diese bei dem Shape-Element des entsprechenden *Procedure Modules* anzuwenden.

Hinweis: Der *Layout Manager* zum Abspeichern und Anwenden der Darstellungsanordnungen für die *Procedure Modules* wird automatisch über eine *T4 Template* generiert. Dieses sammelt Informationen über die einzelnen Elemente sowie Beziehungen, die für die grafische Darstellung bei einem *Procedure Module* relevant sind und erstellt daraufhin automatisch Funktionen zum Anwenden, Aktualisieren und Speichern dieser Informationen.

Benutzerdefinierte Shape-Darstellung. Eine benutzerdefinierte Darstellung von Elementen ermöglicht es, die einzelnen Elemente schnell und zuverlässig auszumachen und somit die dabei dargestellten Strukturen besser zu verstehen. Im vorliegenden Modell war das Ziel, die Darstellung der Elemente an die grafische Notation des V-Modells anzugleichen, um die Einstiegshürde zu verringern. Spezifische Darstellungen sind mit den „Bordmitteln" des DSL-Tools Designers nicht mehr möglich. Dieser verfügt schließlich nur über die Modifikation bereits definierter Darstellungseigenschaften. Um dennoch eine benutzerdefinierte grafische Visualisierung zu erreichen, wie sie beim Prototyp beispielsweise beim Shape vom *Decision Gate Point* gegeben ist, müssen Funktionalitäten im DSL-Tools Framework überschrieben und angepasst werden. Anhang B zeigt eine Möglichkeit, wie dies realisiert werden kann.

4.4 Validierung des Modells

Bei der Validierung mit den DSL-Tools unterscheidet man grundsätzlich die sogenannten *Soft-* und *Hard Constraints* (siehe Kapitel 2.3). In unserem Fall werden diese wie folgt verwendet:

- *Soft Constraints* werden unter Anderem eingesetzt
  - zur Überprüfung der einstelligen Referenzbeziehungen, wie sie beispielsweise zwischen *Module* und *Specification* sowie *DecisionGatePoint* und *DecisionGate* vorliegen.
  - zur Validierung aller möglichen Pfade durch einen *Module*. Dabei muss jeder einzelne Pfad beim *Startpoint* starten und beim *Endpoint* enden.
  - zum Aufzeigen von fehlerhaften einstelligen Referenzbeziehungen.
- *Hard Constraints* werden zur Verhinderung von invaliden Ablaufbeziehungen angewandt. Somit lassen sich fehlerhafte Beziehungen zwischen Ablaufelementen nicht mehr durch den Anwender erstellen.

Validierung bei der Erstellung von Referenzen. Das Domänenmodell beinhaltet eine einzelne Referenzbeziehung für alle Elemente, die *BaseLinkElement* als Basisklasse aufweisen, und modelliert hierdurch Abläufe innerhalb von *Modules*. Allerdings dürfen hierbei nicht alle möglichen Beziehungen erstellt werden, sodass an dieser Stelle Erweiterungen zu treffen sind, die eine Selektion bezüglich der erlaubten Elemente durchführen. Diese können in Form der *Soft Constraints* erfolgen, wodurch aber fehlerhafte Beziehungen angelegt werden können und erst durch die Fehlerliste aufgezeigt wird, welche der entsprechenden Beziehungen tatsächlich falsch sind.

Eine deutlich bessere Möglichkeit an dieser Stelle bieten die *Hard Constraints*. Diese verhindern das Anlegen invalider Beziehungen. Sie unterstützen die Vorgehensweise beim Modellieren von Abläufen für den Anwender indem nicht sinnvolle Assoziationen gar nicht erst möglich sind. Zur Implementierung der entsprechenden *Hard Constraints* müssen allerdings bestehende

Framework-Methoden der DSL-Tools überschrieben und angepasst werden, wobei an dieser Stelle auch explizite Abfragen bezüglich der tatsächlich erlauben und nicht erlaubten Elemententypen notwendig sind.

In unserem Prototyp haben wir uns für die zweite Möglichkeit entschieden und verbieten somit im Vorfeld bereits das Anlegen invalider Ablaufbeziehungen. Hierdurch wird zum einen eine saubere Modellierungsmöglichkeit für das Anlegen von Abläufen geschaffen (nur noch gültige Übergänge sind anlegbar) und zum anderen muss der Anwender sich mit den ungültigen Beziehungen nicht mehr beschäftigen, sie folglich auch gar nicht mehr kennen.

Verhindern des Speichervorgangs bei invaliden Modellen. Bei der Speicherung des Modells können zwei Vorgehensweisen gewählt werden um mit vorhandenen Fehlern umzugehen:

1. Zulassung des Speichervorgangs, solange der Ladevorgang dadurch nicht unmöglich wird.
2. Verhinderung des Speichervorgangs bei jedem relevanten Fehler.

In unserem Prototyp haben wir die zweite Möglichkeit gewählt, denn wir arbeiten hier in der Regel auf einer bereits vorhanden V-Modell-Instanz, die bereits anderweitig weiterverwendet wird. Somit wäre die Zulassung von Fehlern im serialisierten Modell ausgehend von unserem Prototyp womöglich problematisch, denn vorhandene Fehler könnten unbekannt sein oder auf dieser Art und Weise einfach übersehen werden.

4.4.1 Validierung und Petri-Netze

Petri-Netze [BF09] stellen eine formale Beschreibungsform für Projektdurchführungsstrategien dar. Sie definieren eine formale Semantik des Ablaufs und verfügen zudem über Konsistenzbedingungen.

Aufbau von Petri-Netzen. Die Implementierung von Petri-Netzen im Prototyp gestaltet sich wie folgt (siehe Abbildung 4.2).

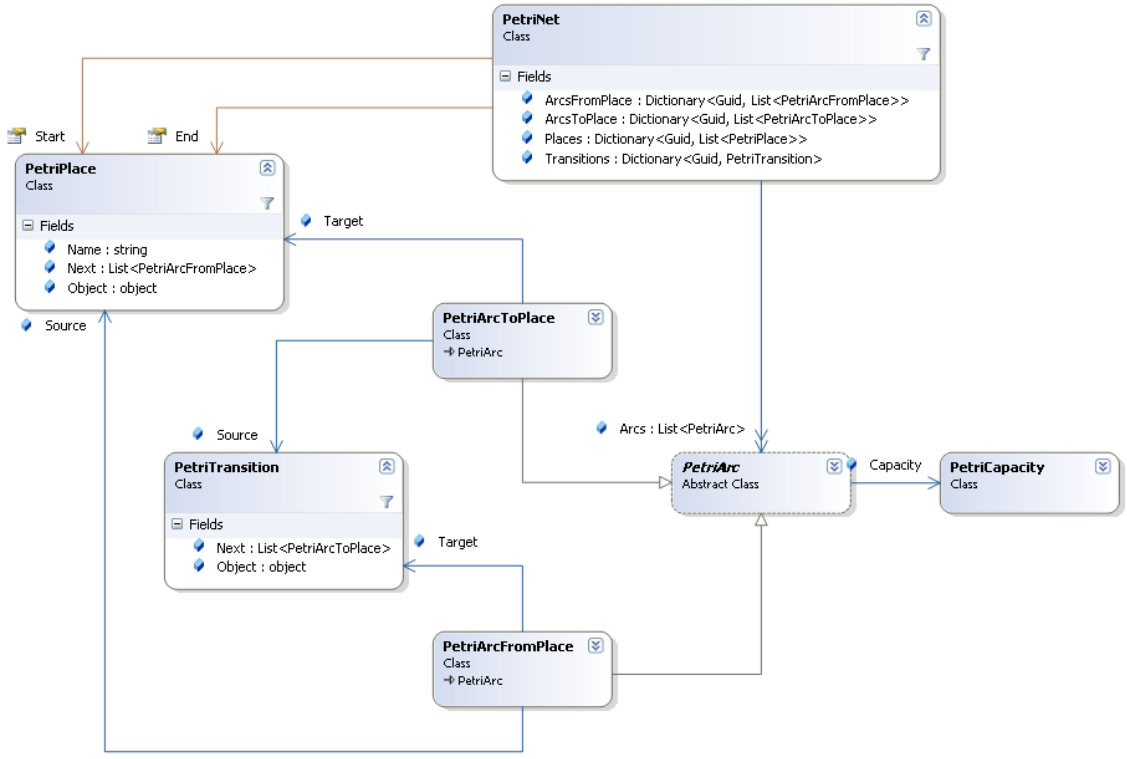

Abbildung 4.2: Petri-Netz Modell im Prototyp

Jedes Petri-Netz wird über die Klasse *PetriNet* dargestellt, die neben den notwendigen Elementen *Start* und *End* noch zusätzlich die Eigenschaften: *Arcs*, *ArcsFromPlace*, *ArcsToPlace*, *Places* und *Transitions* zur Erleichterung der Definition von Algorithmen enthält.

Start und *End* stehen dabei für den Startpunkt und den Endepunkt des Ablaufs, der durch ein PetriNetz beschrieben wird. Beide sind vom Typ *PetriPlace* und sind für den Aufbau eines Petri-Netzes essenziell. Ein *PetriPlace* steht dabei für die einzelnen Ablaufelemente wie Startpoint, Endpoint, Split und Join Ports sowie Procedure Specification und Procedure Decision Gate Point. Diese werden bei einem *PetriPlace* über die Eigenschaften *Object* und *Name* hinterlegt. Dabei drückt der Name aus um welche Elementenart es sich handelt, was nicht rein durch die Speicherung der zugehörigen Klasseninstanz beim Objekt erbracht werden kann, da beispielsweise die Integration von Ablaufbausteinen mittels zwei Elemententypen im Petri-Netz beschrieben wird (durch *SpecificationPointEntry* und *SpecificationPointExit*), die es in unserem Domänenmodell nicht gibt.

Zur Definition des Ablaufs wird ausgehend von einem *PetriPlace* ein Übergang mittels *PetriArcFromPlace* zu einer *PetriTransition* angelegt. Dieser Übergang wird beim *PetriPlace* in der *Next*-Eigenschaft hinterlegt, die alle derartig vorhandenen Übergänge aufweist. Zur Fortsetzung des Ablaufs wieder zu einem *PetriPlace* wird der Übergang *PetriArcToPlace* eingesetzt, der eine Verbindung zwischen einer *PetriTransition* und einem *PetriPlace* herstellt und bei der *Next*-Eigenschaft der *PetriTransition* hinterlegt wird. Wichtig hierbei ist, dass alle Übergänge vom Typ *PetriArc* abgeleitet sind und somit über die Eigenschaft *Capacity* die Kardinalität der Verbindung festlegen können. Auf dieser Art und Weise werden parallele Prozesse, die durch Splits und Joins definiert werden, auch im Petri-Netz beschrieben.

Somit wird ein Petri-Netz prinzipiell wie folgt aufgebaut: Ausgehend von *Start* s wird ein Übergang mittels *ArcFromPlace* a_1 zu einer *PetriTransition* t angelegt. Ausgehend von t wird nun ein Übergang *ArcToPlace* a_2 angelegt, der t mit einem weiteren *PetriPlace* p_0 verbindet. Dieser Vorgang wird nun solange wiederholt, bis es ein gewisses $i \geq 0$ gibt mit $p_i = e$ und e ist *End*.

Zusammengefasst werden die einzelnen Elemente eines Petri-Netzes bei den zugehörigen Eigenschaften hinterlegt. So sind alle vorhandenen Elemente vom Typ *PetriPlace* der Eigenschaften *Places* zugeordnet, während alle vorhandenen Übergänge bei *Arcs* zusammengefasst und bei den typisierten *ArcsFromPlace* und *ArcsToPlace* spezifisch sortiert werden. Weiterhin finden sich alle *PetriTransitions* bei der Eigenschaft *Transitions* wieder.

Beispiele. Im Folgenden geben wir einige Beispiele für Petri-Netze an, die aus modellierten Ablaufbausteinen und Projektdurchführungsstrategien ermittelt wurden.

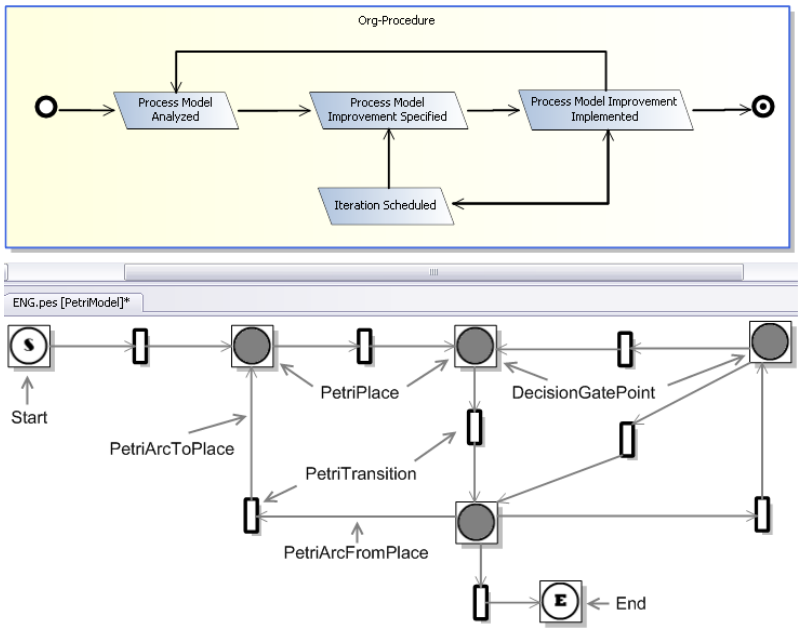

Abbildung 4.3: Petri-Netz zum Ablaufbaustein „Org-Procedure“

Ein einfaches Beispiel für ein Petri-Netz lässt sich Abbildung 4.3 entnehmen. Dort wird der fla-

che Ablaufbaustein „Org-Procedure" dargestellt, dessen innerer Ablauf ausschließlich Ablaufentscheidungspunkten (Procedure Decision Gate) aufweist, sodass keine Integration von weiteren Ablaufbausteinen (Hierarchie) notwendig ist. Die Darstellung hierbei ist noch gut erfassbar, so dass sie auch vom menschlichen Bearbeiter schnell nachvollzogen werden kann.

In der Regel sind Petri-Netze allerdings deutlich komplexer, wie Abbildungen 4.4 und 4.5 zeigen. Beiden Abbildungen liegt dabei jeweils ein Ablaufbaustein zugrunde, der über Joins und Splits verfügt, so dass wir auch in den zugehörigen Petri-Netzen parallele Prozesse beschreiben. Hierdurch wird die Darstellung deutlich komplexer.

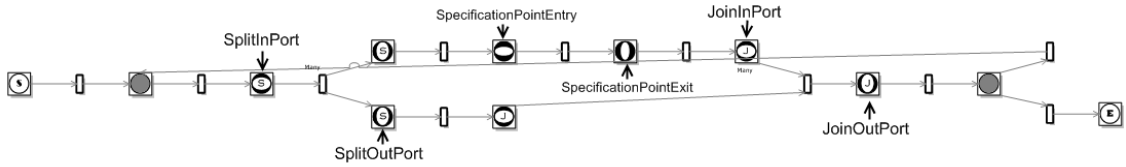

Abbildung 4.4: Petri-Netz zum Ablaufbaustein „Detailed Design and Realization of System Elements", ohne Integrationen

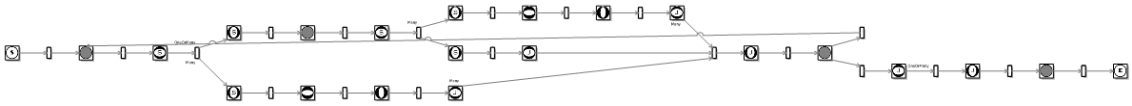

Abbildung 4.5: Petri-Netz zum Ablaufbaustein „Component-Based System Development", ohne Integrationen

Zuletzt geben wir noch eine sehr komplizierte Beschreibung in Form der AN-Strategien (siehe Abbildung 4.6). Hier wird das Petri-Netz mit allen möglichen Ablaufbausteinen, die integriert werden können, generiert, sodass die Darstellung sehr komplex und vom Menschen kaum mehr verstanden werden kann.

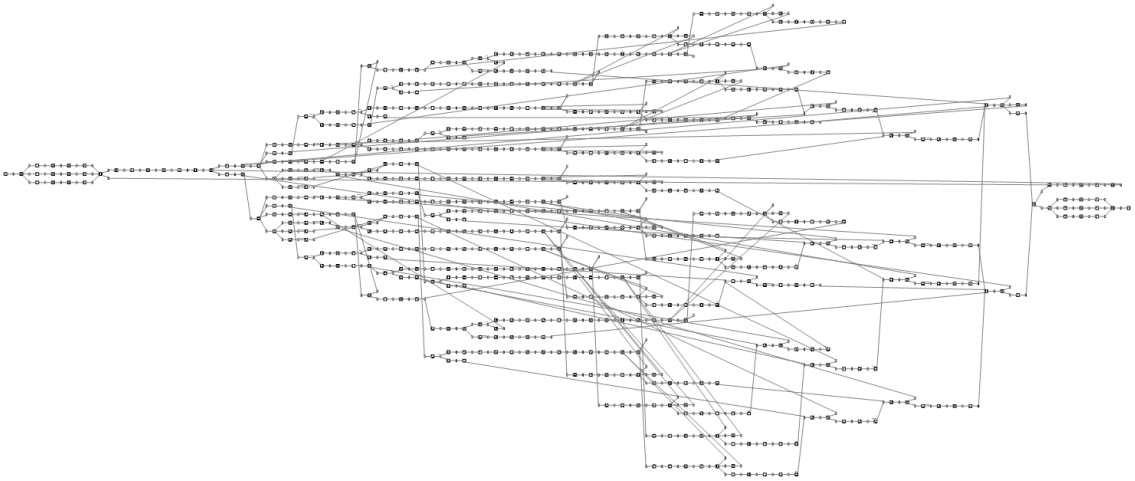

Abbildung 4.6: Petri-Netz zu den AN-Strategien mit allen möglichen Integrationen

Einsatz bei der Validierung. Im Prototyp setzen wir Petri-Netze bei der Validierung ein, um die Konsistenz von Projektdurchführungsstrategien zu überprüfen. Jedes Petri-Netz, wie oben dargestellt, beschreibt dabei einen Ablauf, der allerdings sehr kompliziert sein kann. Da wir aber bei den Konsistenzprüfungen auf automatische Algorithmen setzen und somit auf manuelle Überprüfungen verzichten, ist der Umstand der Komplexität nicht derartigen ausschlaggebend.

Vorteilhaft bleibt zudem, dass die Beschreibung durch Petri-Netze einer erweiterten Konsistenzprüfung gleicht, die sowohl modular als auch auf integrierte, wie auf alle möglichen Strategien anwendbar ist. So lässt sie sich anwenden auf einzelne

- flache Ablaufbausteine.
- hierarchische Ablaufbausteine, hierbei werden mögliche Integrationen nicht berücksichtigt und nur der einzelne Ablaufbaustein untersucht.
- Ablaufbausteine mit allen Integrationen gemäß einer vorhandenen Konfiguration, sodass nur Ablaufbausteine bei der Integration berücksichtigt werden, die in der zugehörigen Projekttypvariante festgelegt sind. Auf dieser Art und Weise lässt sich eine Projektdurchführungsstategie, die zu einem bestimmten Projekttyp gehört, untersuchen.
- Ablaufbausteine mit allen möglichen Integrationen, sodass die Konsistenz aller theoretisch wählbarer Konfigurationen sichergestellt wird.

Im Prozess der Validierung wird die Erstellung der Petri-Netze auch erst nach der Überprüfung und Einhaltung der *Soft Constraints* möglich. Generell spiegelt ein generiertes Petri-Netz bereits die Einhaltung viele Konsistenzkriterien wieder. Wir setzen die Petri-Netze weiterhin insbesondere zur Validierung der Kardinalität paralleler Prozesse ein, die ohne eine hierzu geeignete Beschreibungsform der Abläufe nicht ohne weiteres durchführbar wäre. Diese Validierung kann allerdings aufgrund der Komplexität erlaubter Parallelitäten nicht immer automatisch entscheiden ob es sich um einen Fehler handelt oder nicht, sodass die hierbei festgestellten Fehler eher als Warnungen zu deuten sind und so auch im Editor dargestellt werden.

Die Petri-Netz-Idee erlaubt weiterhin die Validierung von Projektplänen bezüglich der Ableitbarkeit aus einer Strategie. Dazu – vorausgesetzt entsprechende Algorithmen werden eingeführt – sind Projektpläne als Petri-Spuren gegen die Projektdurchführungsstrategien als Petri-Netze zu validieren.

4.4.2 Ergebnisse der Validierung

In der Einführung dieses Berichts (Kapitel 1.1) haben wir einige Problembereiche in der derzeitigen Modellierung mit dem V-Modell XT Editor aufgelistet. Nun wollen wir diese wieder aufgreifen und explizit Vergleiche führen zwischen den Validierungsmöglichkeiten des V-Modell XT Editors und unseres Prototyps. Das Ziel hierbei liegt in der Feststellung, dass die genannten Probleme in unserem Prototyp behoben wurden und somit eine Verbesserung in der Modellierung der Strategien gegenüber dem V-Modell XT Editor für den Prozessingenieur gegeben ist.

Bei unseren Vergleichen wollen wir zunächst eine Fehlermöglichkeit betrachten und folglich die Fehlerfindungs- bzw. die Fehlerbehebungsaßnahmen feststellen.

Vergleich 1: Löschen eines Übergangs

Übergänge dienen der Definition von Abläufen innerhalb der Ablaufbausteine. Folglich bilden sie eine naheliegende Fehlerquelle, denn sie müssen bei den meisten Änderungen an den Strategien angepasst oder bei neuen Strategien neu modelliert werden. Wie bereits in der Einführung dieses Berichts erwähnt, behandelt der V-Modell XT Editor alle Elemente gleich, somit auch Übergänge wie Joins oder Splits. Eine spezielle Modellierungsweise oder Darstellungsweise für die Übergänge fehlt an dieser Stelle. Wir wollen nun betrachten, wie der V-Modell XT Editor mit Modellierungsfehlern bei den Übergängen umgeht, und löschen dazu einen beliebigen Übergang.

Achtung

Das Löschen eines Übergangs stellt keinen syntaktischen Fehler dar. Genauer genommen, ist rein durch die Strukturen des Metamodells des V-Modell XT nicht vorgegeben, dass ein Ablaufbaustein über ausschließlich gültige innere Abläufe verfügen muss. (Diese legen wir für unsere DSL über Soft Constraint 3.4 fest und erweitern diese Definition über Hard Constraints 3.1, 3.2 und 3.4).

Nach dem Löschen eines Übergangs im V-Modell XT Editor starten wir folgend den Export, der auch erwartungsgemäß fehlerfrei durchgeführt wird. Wir wissen aber, dass es bei einer Strategie einen Fehler geben muss und suchen diesen in der Prozessdokumentation. Wir finden diesen Fehler in einem der generierten Bilder im Teil 3: Tailoring (Projekttypvarianten), siehe Abbildung 4.7.

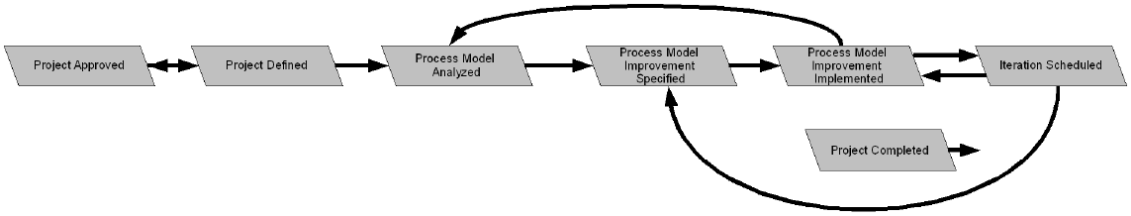

Abbildung 4.7: Fehler nach dem Löschen eines Übergangs in der Dokumentation des V-Modell XT

Die Fehlerfindung ist an dieser Stelle nicht immer unproblematisch. Wir konnten diesen Fehler relativ leicht finden, weil wir genau wussten **wo** wir diesen gemacht haben. Wird hingegen ein Fehler unbewusst modelliert, so ist dieser relativ schwer aufzufinden, da dazu entweder im Planungsmodul des Projektassistenten Beispielplanungen aller möglichen Ablaufkombinationen geplant werden müssen oder in der dazugehörigen Dokumentation explizit danach gesucht werden muss. Ein weiterer problematischer Aspekt ist das Auffinden der genauen Quelle. Die Bilder in der Dokumentation stellen bereits integrierte Strategien dar, sodass nicht immer direkt klar ist, wo sich die eigentliche Fehlerquelle befindet. Somit müssen hier gegebenenfalls mehrere zusammenhängende Ablaufbausteine untersucht werden.

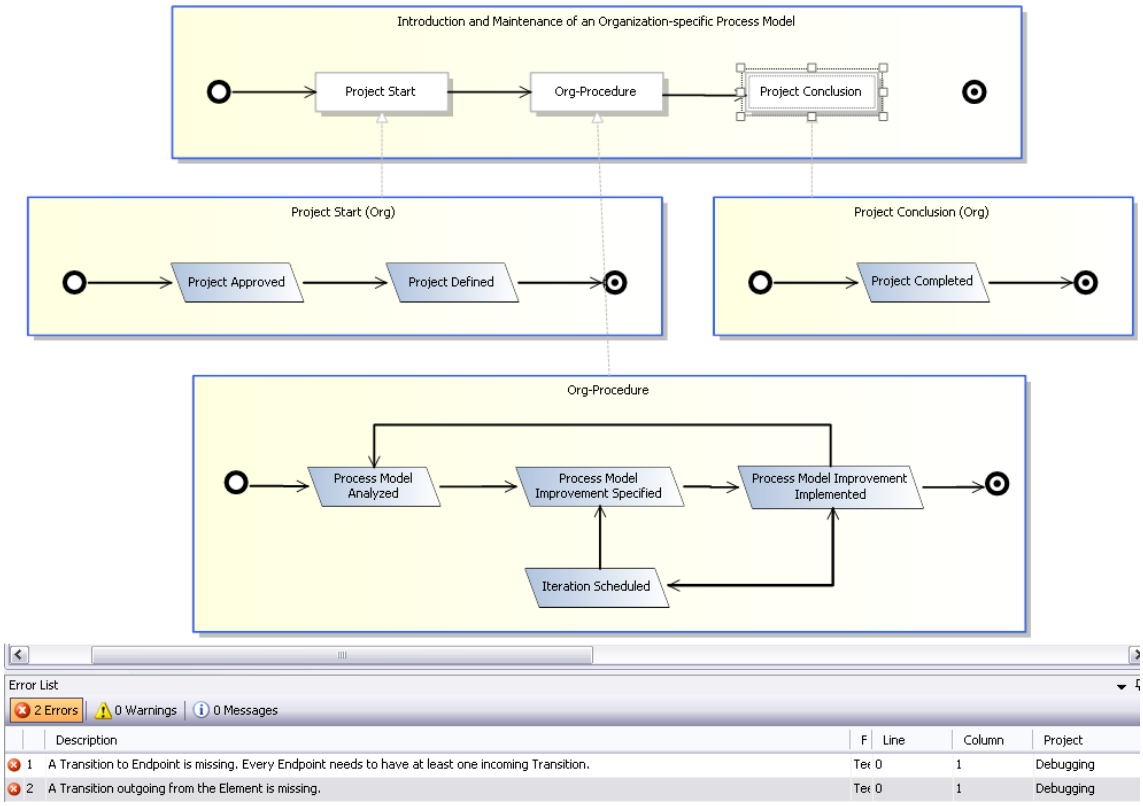

Abbildung 4.8: Fehler nach dem Löschen eines Übergangs im Prototyp

Im Prototyp gehen wir mit diesen Problemen über zwei Möglichkeiten zur Fehlerfindung um (siehe auch Abbildung 4.8):

1. Anhand der grafischen Darstellung lassen sich Fehler direkt ablesen. Das ist allerdings nur möglich falls das fehlerverursachende Element tatsächlich auf der Designeroberfläche dargestellt wird.
2. Die Validierung findet Fehler zur Design-Zeit und listet dieselben mit der Quelle und der Ursache auf. Insbesondere lässt sich direkt von der Fehlerauflistung zur Fehlerquelle springen, sodass hier eine schnelle Fehlerbehebung gegeben ist.

Somit ist die Fehlerfindung sowie die Fehlerbehebung im Prototyp erheblich einfacher. Es entfallen unnötige Exportvorgänge, die Konsistenz wird zur Design-Zeit überprüft und ihre Einhaltung erzwungen.

Hinweis: Die Validierung liefert in Abbildung 4.8 genau zwei Fehler. Das liegt daran, dass ausgehend vom Ablaufbausteinpunkt (*Procedure Specification Point*) sowie eingehend beim Endpunkt (*Endpoint*) jeweils Übergänge vorhanden sein müssen.

Vergleich 2: Anlegen nicht erlaubter Übergänge

Bei der Modellierung von Abläufen innerhalb der Ablaufbausteine werden Übergänge angelegt. Wie oben bereits gezeigt wurde, ist diese Modellierung nicht immer unproblematisch. In diesem Beispiel wollen wir nun einen fehlerhaften Übergang erstellen und die Folgen daraus beim V-Modell XT Editor und bei unserem Prototyp vergleichen. Bei diesem Übergang handelt es sich dabei um einen Übergang, der zwischen Elementen unterschiedlicher Ablaufbausteine definiert wird. Generell ist so ein Übergang nicht erlaubt, denn dieselben dürfen nur innerhalb und somit nicht zwischen den modularen Ablaufbausteinen modelliert werden. Allerdings filtert der V-Modell XT Editor die möglichen Teilnehmer, die Quelle und das Ziel eines Übergangs, auch nicht für einen Ablaufbaustein spezifisch, sodass theoretisch Elemente aller vorhandener Ablaufbausteine ausgewählt werden können (siehe Abbildung 4.9).

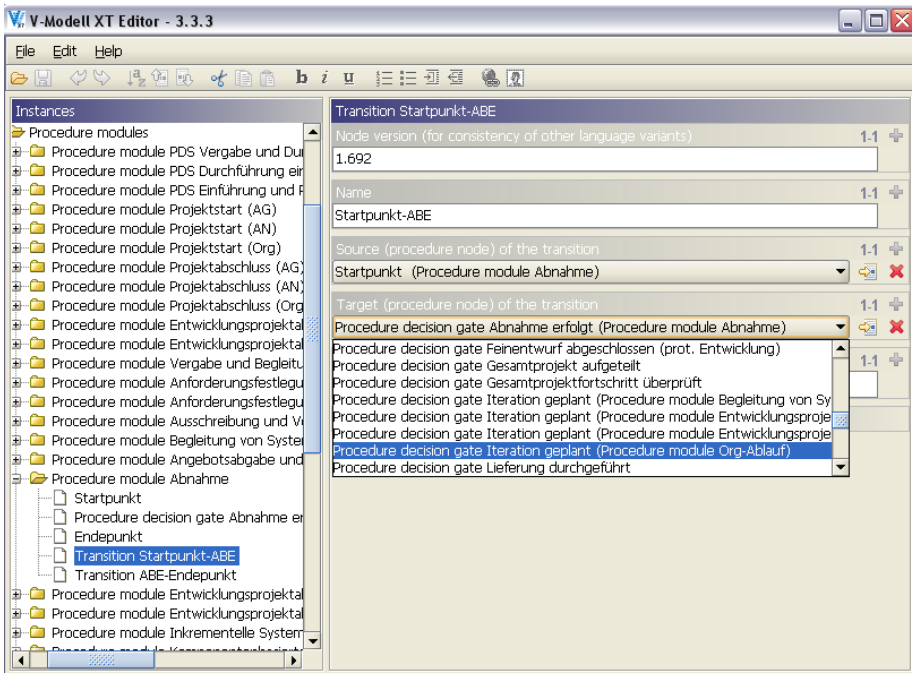

Abbildung 4.9: Modellierung von Übergängen (Quelle und Ziel) im V-Modell XT Editor

Wir modellieren folgend einen Übergang, der als Quelle und Ziel Elemente unterschiedlicher Ablaufbausteine aufweist und starten den Export. Dieser schlägt erwartungsgemäß fehl und wir bekommen die Fehlermeldung aus Abbildung 4.10. Die erhaltene Fehlermeldung ist sehr problematisch, denn sie weist uns weder auf einen genaue Fehlerquelle hin, noch nennt sie die tatsächliche Fehlerursache, sodass wir die Suche einschränken könnten. Der Fehler kann, rein basierend auf der Meldung, beispielsweise auch bei fehlenden Referenzen oder im Tailoringmodell liegen. Somit erhalten wir hier nicht mal die garantierte Bestätigung, dass der Fehler bei einer Projektdurchführungsstrategie liegt noch bei welchem Ablaufbaustein er passiert ist, geschweige denn welcher Übergang tatsächlich für die Meldung verantwortlich ist. Folglich müssen zur Fehlerbehebung große Teile des V-Modells oder – falls man die Einschränkung auf die Strategien trifft – zumindest die einzelnen Ablaufbausteine im Detail analysiert werden, um den Fehler zu identifizieren. Dieser Umstand ist natürlich problematisch und wird im Prototyp dementsprechend gelöst.

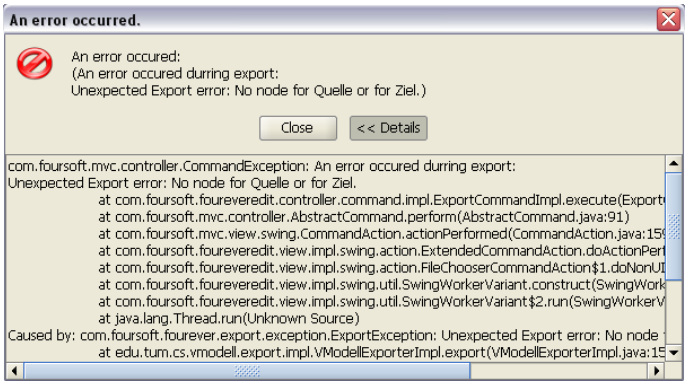

Abbildung 4.10: Fehlermeldung beim Export nach der Modellierung eines fehlerhaften Übergangs

Im Prototyp werden Übergänge über grafische Verbindungskanten visualisiert und können auch nur bei Elementen angelegt werden, die auf der Designeroberfläche sichtbar sind. Für einen Anlegevorgang wird dabei zunächst der zugehörige Verbindungstyp aus der Visual Studio Toolbox ausgewählt, um folglich ausgehend von einem Quellelement mittels einer Ziehbewegung zum Zielelement den Übergang anzulegen. Um die obige Problematik in den Griff zu bekommen, ist die Auswahl des Quell- und des Zielelements ausschließlich auf die Elemente des Ablaufbausteins beschränkt, in dem sich das Quellelement befindet. Hierdurch wird das obige Problem verhindert, indem grundsätzlich keine derartig fehlerhaften Übergänge modelliert werden können.

Zusammenfassung. Zusammengefasst lässt sich festhalten, dass der Prototyp fehlerhaften Modellierungen vorbeugt oder dieselben über eine Fehlerliste mit Quelle und Ursache aufzeigt. Somit entfallen früher notwendige manuelle Konsistenzprüfungen. Zusätzlich unterstützt der Prototyp die Modellierung, indem entsprechende Mechanismen eingesetzt werden, die diese erleichtern (z. B. Erstellung von Übergängen). Hierdurch wird die Unterstützung bei der Erstellung oder Verwaltung der Projektdurchführungsstrategien für den Prozessingenieur deutlich verbessert.

4.5 Evaluierung der DSL-Tools

In diesem Abschnitt wollen wir eine kurze Evaluierung des DSL-Tools Framework geben, die sich auf die während der Implementierung des Prototyps gewonnenen Erkenntnisse stützt.

Die Hauptvorteile der DSL-Tools können im Abstraktionsnivieau sowie in der Flexibilität und der Konsistenz des Frameworks gesehen werden. Dadurch lassen sich domänenspezifische Sprachen leichter implementieren und erweitern. Die Flexibilität des Frameworks geht vor allem auf die automatische Generierung von *partiellen* Klassen zurück, sodass anwendungsspezifische Anpassungen ermöglicht werden. Zusätzlich lassen sich große Teile des Frameworks auch durch das Überschreiben von vorhandenen Methoden hinsichtlich gewünschter Eigenschaften und Prozesse modifizieren, sodass sehr viele Features implementiert werden können.

Die Erstellung von DSLs mit dem DSL-Tools Designer verfügt zudem über eine Überprüfungsfunktionalität, die wie die Validierungsmaßnahmen im hierbei erzeugten Editor, über Fehlerlisten die Beseitigung der vorhandenen Fehler vorschlägt. Das erlaubt eine einfache und weniger fehleranfällige Erstellung von DSLs.

Die DSL-Tools integrieren sich selbst in das Visual Studio und arbeiten basierend auf dem .NET Framework, sodass zum einen eine mächtige Entwicklungsumgebung zur Verfügung steht und zum anderen objektorientierte Hochsprachen zur Anpassung bzw. Erweiterung des automatisch generierten Quellcodes verwendet werden können. Weiterhin wird der erstellte Editor auch im Visual Studio ausgeführt, sodass eine hohe Steigerung in der Produktivität erreicht werden kann.

Eine der Hauptnachteile der DSL-Tools ist die recht steile Lernkurve. Der Einstieg fällt zwar verhältnismäßig leicht – will man jedoch erweiterte Konzepte nutzen und das Modell mit Eigenschaften versehen, die nicht „out-of-the-box" unterstützt werden, steigt die Komplexität sehr schnell

an. So war selbst das Dekompilieren von CLR-Code erforderlich, um einige fortgeschrittenere Zusammenhänge zu verstehen und Anpassungen zu implementieren. Darüber hinaus fällt das Fehlen von einigen sinnvollen Standardfeatures (Copy&Paste) negativ auf, die allerdings auch durch eigenen Quellcode und Eingriffe in das Framework implementiert werden können. In der kommenden Version der DSL-Tools (Visual Studio 2010 SDK, siehe [Pri09]) werden viele dieser Features umgesetzt sowie um einige neue interessante Konzepte erweitert.

Abschließend lässt sich anhand unserer Erfahrung festhalten: Viele der genannten Hürden und Schwierigkeiten sind auf das recht frühe Entwicklungsstadium der DSL-Tools zurückzuführen. Es ist zu vermuten (und im Visual Studio 2010 SDK bereits erkennbar), dass diese mit der weiteren Reifung des Produkts behoben werden. Aufgrund der Flexibilität, der mächtigen Entwicklungsumgebung und der Freiheit, das Modell durch eigenen Code anzureichern, sind die DSL-Tools für beinahe beliebige Modelle geeignet und erlauben es, komplexe Sachverhalte durch grafisch dargestellte Sprachen zur besseren Handhabung umzusetzen.

5 Zusammenfassung und Ausblick

In diesem Bericht wurde eine domänenspezifische Sprache zur Modellierung und Validierung von Projektdurchführungsstrategien vorgestellt. Diese basiert auf den Metamodellstrukturen des V-Modell XT, die in das Domänenmodell der DSL-Tools abgebildet wurden. Der hierbei erstellte Editor (siehe Abbildung 5.1) verfügt über eine grafische Darstellung für die Projektdurchführungsstrategien und erlaubt folglich die Modifikation bzw. die Erfassung der Zusammenhänge der einzelnen Elemente über eine grafische Visualisierung, sodass die komplexen Abläufe einfacher nachvollzogen werden können. Die grafische Darstellung hat zudem noch den Vorteil, dass Fehler direkt aus der Darstellung erkannt und behoben werden können. Dieses wird zusätzlich über die Validierung unterstützt, die zu Designzeit stattfindet und den Prozessingenieur auf die vorhanden Fehler hinweist, indem explizit aufgezeigt wird, wo sich ein Fehler befindet und warum es sich um einen Fehler handelt. Somit wird die Modellierung bzw. die Modifikation von Projektdurchführungsstrategien erheblich erleichtert.

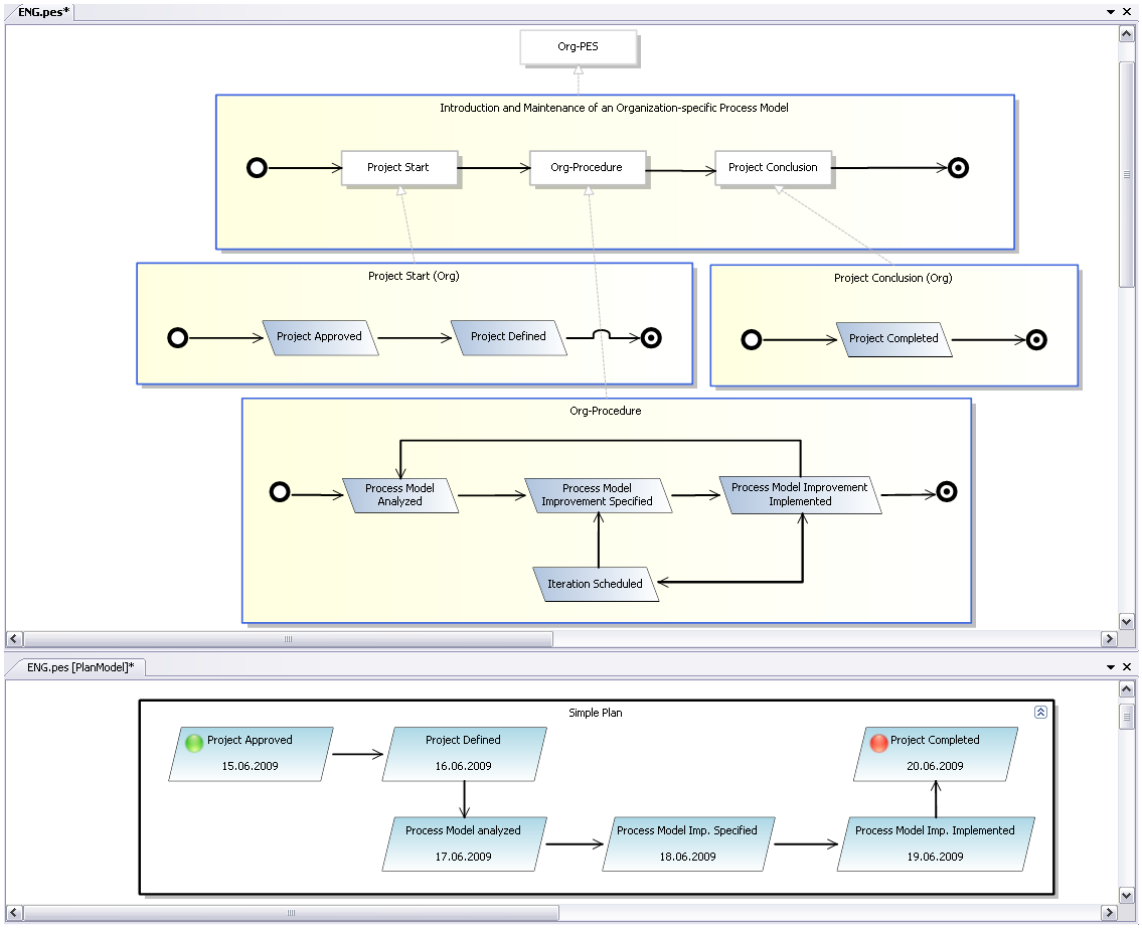

Abbildung 5.1: Einfaches Beispiel des Prototyps

Weiterhin erlaubt unsere domänenspezifische Sprache die Definition von Projektplänen, die über Referenzen zur zugehörigen Projektdurchführungsstrategie (Meilenstein zum Entscheidungspunkt) verfügen und folglich auch bezüglich der Anordnung der Meilensteine im Projektplan und der Entscheidungspunkte in der Strategie validiert werden können. Somit ist feststellbar, ob ein Projektplan aus der zugrunde liegenden Projektdurchführungsstrategie ableitbar ist oder nicht. Weiterhin lassen sich gegebenenfalls Information über nicht enthaltene Entscheidungspunkte oder Abläufe dem Prozessingenieur zurückgeben, sodass dieser seine Fehlersuche an den

relevanten Stellen fortsetzen kann. An dieser Stelle könnte man in der Zukunft auch Methoden und Algorithmen einführen und in den Prototyp integrieren, die in der Lage sind, sinnvolle Vorschläge bezüglich der Behebung solcher Fehler zu liefern, sodass ein gegebener Projektplan aus der zugehörigen Strategie ableitbar wird.

Ein Projektplan, der mit unserer domänenspezifischen Sprache erstellt wird, bedarf einer Referenz auf einen Entscheidungspunkt für jeden einzelnen Meilenstein. Dies ist für die Validierungsmethoden wichtig. Alle Pläne, die nicht über diese Information verfügen, werden daher zurzeit ausgeschlossen. Gelöst werden könnte diese Problematik zum einen durch die Einführung einer Erweiterung zur Zuweisung der fehlenden Referenzen oder zum anderen durch Algorithmen zum automatischen Auffinden von fehlenden Referenzen (z.B. durch Nachbarschaftsanalyse). Weiterhin öffnet die Möglichkeit zur Validierung von Projektplänen gegen die zugehörigen Projektdurchführungsstrategien auch eine neue Sichtweise auf den Prozess selbst. Derzeit verwenden wir eine Projektdurchführungsstrategie, um daraus einen Plan abzuleiten. Die umgekehrte Richtung, nämlich die Verwendung eines Plans zur Generierung einer Strategie, erscheint auch als eine interessante Möglichkeit, die in der Zukunft untersucht werden kann. Hierduch ließen sich erfolgreiche Projektpläne in zugehörige Projektdurchführungsstrategien transformieren. Der Prototyp ließe sich entsprechend erweitern um derartige Vorgänge zu unterstützen.

Der implementierte Prototyp dient als Konzeptbeweis. Doch auch diese Implementierung war bereits sehr aufwändig. Implementierungsaufwand entstand vor allem bei der Anpassung des Editors zur Unterstützung der gewünschten Vorgänge wie in Kapitel 4 beschrieben. Die Erweiterung des Prototyps zur Unterstützung des gesamten V-Modell XT und nicht nur der Projektdurchführungsstrategien wäre mit dieser Vorgehensweise sehr aufwendig. Bei jeder Änderung am V-Modell Metamodell wären Anpassungen an vielen unterschiedlichen Stellen im Domänenmodell notwendig. Hier ist ein anderer Ansatz sinnvoller, der eine abstraktere Vorgehensweise wählt und das gesamte V-Modell XT Metamodell einschließt.

A How To: Benutzerdefinierte Serialisierung

Die Serialisierung des Modells wird standardmäßig durch die hierzu automatisch generierten Klassen übernommen. Für sehr viele Modelle, wie auch in unserem Fall, muss sie angepasst werden, um bereits bestehende Daten mit einzubeziehen. Im Folgenden wollen wir die Implementierung der Serialisierung in unserem Prototyp vorstellen, wobei wir hier nur die Lade- und Speichervorgänge der Modelldaten anpassen und die Serialisierung der grafischen Informationen beibehalten.

Hinweis: Die DSL-Tools generieren für unseren Prototyp für die Serialisierung die Klasse *PESEditorSerializationHelper*. Hier können Methoden zur Anpassung der Serialisierung hinsichtlich des Datenmodells und der grafischen Information überschrieben werden. Wesentlich dabei sind die Methoden:

- LoadModelAndDiagram
- SaveModelAndDiagram

Diese werden beim Laden und Speichern durch die Klasse *PESEditorDocData* aufgerufen, die selbst ein PESEditor-Dokument repräsentiert.

Zielsetzung. Wir wollen mit der Anpassung der Serialisierung in der Lage sein, bereits bestehende Daten aus dem V-Modell zu laden und die folgend modifizierten Daten dort wieder abzuspeichern. Weiterhin soll das Planmodell unabhängig vom V-Modell (dort kommen die entsprechend hierzu notwendigen Strukturen nicht vor) serialisierbar sein. Insofern verwenden wir das Standarddokument des Prototyps (**.pes*) zum Speichern der Datenelemente des Planmodells sowie einer Referenz auf die zugehörige V-Modell-Datei.

Die grafischen Informationen können wir hierbei nach wie vor über die automatisch generierten Klassen und Methoden serialisieren.

Laden des Modells. Zur Anpassung des Ladevorgangs des Datenmodells überschreiben wir die Methode *LoadModel* der *PESEditorSerializationHelper* Klasse (siehe Listing A.1). Im Wesentlichen lässt sich hier der automatisch generierte Quellcode als Rahmen fast vollständig wiederverwenden, es muss nur eine Anpassung hinsichtlich des Laden der einzelnen Datenelemente erfolgen. Hierzu haben wir eine neue Klasse angelegt (*DomainSerializier*), die verantwortlich ist für

- das Laden der V-Modell XT Datenelemente.
- das Laden der Elemente des Planmodells.

Diese Klasse liefert folglich auch die Instanz des *PESEditorModel* mit allen geladenen Daten.

Listing A.1: Laden des Datenmodells

```
public override PESEditorModel LoadModel(SerializationResult serializationResult,
    Partition partition, string fileName, ISchemaResolver schemaResolver,
    ValidationController validationController)
{
    // Check Parameters
    // ...

    // Ensure there is a transaction for this model to Load in.
    if (!partition.Store.TransactionActive)
    {
        throw new global::System.InvalidOperationException(PESEditorDomainModel.
            SingletonResourceManager.GetString("MissingTransaction"));
    }

    using (Transaction t = partition.Store.TransactionManager.BeginTransaction("Load
        ␣Model␣from␣" + fileName, true))
    {
        // setup moniker
        global::System.Collections.Generic.IDictionary<global::System.Guid,
            IMonikerResolver> resolvers = this.GetMonikerResolvers(partition.Store);
```

```
            foreach (global::System.Collections.Generic.KeyValuePair<global::System.Guid
                , IMonikerResolver> pair in resolvers)
            {
                IMonikerResolver existingMonikerResolver = partition.Store.
                    FindMonikerResolver(pair.Key);
                if (existingMonikerResolver == null)
                {
                    existingMonikerResolver = pair.Value;
                    partition.Store.AddMonikerResolver(pair.Key, existingMonikerResolver
                        );
                }
            }

            // load model
            PESEditorModel pesModel = CustomCode.View.Serialization.DomainSerializer.
                Load(partition, fileName);

            if (t.IsActive)
                t.Commit();

            return pesModel;

        }
    }
```

Speichern des Modells. Bei der Anpassung des Speichervorgangs für das Datenmodell gehen wir sehr ähnlich zum Laden vor. Hierzu überschreiben wir die Methode *SaveModel* der *PESEditorSerializationHelper* Klasse (siehe Listing A.2) und rufen die Methoden unserer Serialisierungsklasse (*DomainSerializier*) auf, die basierend auf der übergebenen Instanz des Datenmodells sowie eines Dateipfades die entsprechende Serialisierung durchführt. Der Dateipfad hierbei steht für das PESEditor-Dokument (**.pes*).

Listing A.2: Speichern des Datenmodells

```
public override void SaveModel(SerializationResult serializationResult,
    PESEditorModel modelRoot, string fileName, Encoding encoding, bool
    writeOptionalPropertiesWithDefaultValue)
{
    // Check Parameters
    // ...

    CustomCode.View.Serialization.DomainSerializer.Save(modelRoot, fileName);
}
```

DomainSerializier. Der DomainSerializier ist für das Laden und Speichern der Daten des V-Modell verantwortlich. Insofern werden hier Klassen des .Net Frameworks für die Serialisierung von XML-Daten verwendet.

B How To: Implementierung von benutzerdefinierten Shapes

Im Folgenden wird beschrieben, wie im DSL-Tools Framework Shapes mit benutzerdefinierter Darstellungsinformation implementiert werden können. Dies wird anhand der Darstellung des *Decision Gate Point* Elements (*DecisionGateShape*) durchgeführt, der über eine geometrisch orientierte Form mit einem Gradientenverlauf als Hintergrund verfügt (siehe Abbildung B.1).

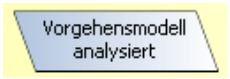

Abbildung B.1: Grafische Darstellung des DecisionGatePoint Elements

Einstellungen im DSL-Tools Designer. Zunächst müssen gewisse Eigenschaften im DSL-Tools Designer für das *DecisionGateShape* gesetzt werden, die zum einen die Implementierung ermöglichen und zum anderen diese auch direkt gestalten. Dazu zählen:

- *Generates Double Derived*: Hierdurch wird die zum Shape gehörende Klasse so generiert, dass gewisse Basiseigenschaften und Methoden überschrieben werden können, die für die von uns gewünschte grafische Darstellung notwendig sind.
- *Fill-, Outline, Text Color*: Wir benutzen nach wie vor die für die Darstellung im DSL-Tools Designer setzbaren Farben.
- *Geometry*: An dieser Stelle lassen sich *(Rounded) Rectangle*, *Circle* und *Ellipse* wählen. Das führt später zu einigen Vor- und Nachteilen bei der Darstellung, auf die wir auch eingehen werden. Zunächst entscheiden wir uns für *Rectangle*, verwenden folglich die hierbei festgelegten Zeichenmethoden als Basis.

Hinweis: Für die Darstellung eines Shapes bei den DSL-Tools sind bei der Anpassung im Wesentlichen zwei Klassen relevant:

- *ShapeGeometry*: Definition der geometrischen Form
- *AreaField*: Definition der notwendigen Informationen für den Gradienten

Die Anpassung dieser beiden Klassen geschieht über das Überschreiben der entsprechenden Eigenschaften bzw. Methoden der Shape-Klasse.

Quellcode Anpassung. Die zum Shape gehörende Klasse wurde als partielle Klasse generiert, so dass wir in der Lage sind einige grundlegende Eigenschaften und Methoden anzupassen und zu implementieren. Diese sind auch im Listing B.1 dargestellt. Wichtig hierbei sind vor allem

- *ShapeGeometry*: Hierdurch wird die geometrische Form festgelegt.
- *CreateBackgroundGradientField*: Hierbei wird der Gradient gezeichnet. In unserem Fall muss das neben der Anpassung der *ShapeGeometry* modifiziert werden, da ansonsten der Gradient die Standarddarstellung bezüglich der geometrischen Form aufweist und womöglich die zuvor definierte Form in der Darstellung zumindest teilweise überdeckt.

Hätten wir anstatt von *Rectangle* beispielsweise *Circle* als Eigenschaft für *Geometry* gewählt, so wären an dieser Stelle keine Anpassungen zu erstellen (die Standarddarstellung weist hier keine Probleme auf). Allerdings hätten wir dann auch keinen Gradientenverlauf als Hintergrund.

Listing B.1: Anpassung der DecisionGateShape Klasse

```
// ...
public partial class DecisionGateShape
{
    public override ShapeGeometry ShapeGeometry
    {
```

```
        get
        {
            DecisionGateShapeGeometry shapeGeometry = new DecisionGateShapeGeometry
                ();
            return shapeGeometry;
        }
    }

    public override bool HasBackgroundGradient
    {
        get
        {
            return true;
        }
    }
    public override bool HasShadow
    {
        get
        {
            return false;
        }
    }

    protected override AreaField CreateBackgroundGradientField(string fieldName)
    {
        return new DecisionGateShapeAreaField(fieldName);
    }
}
// ...
```

Im Folgenden wollen wir die Klassen zur Anpassung der geometrischen Form sowie des Gradientenverlaufs genauer anschauen. Das Zeichnen der geometrischen Darstellung für das *DecisionGateShape* in der angepassten Klasse *ShapeGeometry* lässt sich dem Listing B.2 entnehmen. Wichtig hierbei ist die Anpassung des *GraphicsPath,* worüber sich unterschiedliche geometrische Formen realisieren lassen.

Listing B.2: Anpassung der DecisionGateShape Klasse

```
// ...

public class DecisionGateShapeGeometry : RectangleShapeGeometry
{
    public override GraphicsPath GetPath(IGeometryHost geometryHost)
    {
        RectangleF targetRect = RectangleD.ToRectangleF(geometryHost.
            GeometryBoundingBox);

        GraphicsPath path = base.UninitializedPath;
        path.Reset();

        float intent = 0.13f;

        // draw shape
        path.AddLine(new PointF(0, 0), new PointF(intent, targetRect.Height));
        path.AddLine(new PointF(intent, targetRect.Height), new PointF(targetRect.
            Width, targetRect.Height));
        path.AddLine(new PointF(targetRect.Width, targetRect.Height), new PointF(
            targetRect.Width - intent, 0));
        path.AddLine(new PointF(targetRect.Width - intent, 0), new PointF(0, 0));

        // calculate the bounds of the drawn character, calculate whether we should
            scale it.
        RectangleF currentBounds = path.GetBounds();
        float scaleX = targetRect.Width / currentBounds.Width;
        float scaleY = targetRect.Height / currentBounds.Height;

        // scale it
        Matrix scaleTransform = new Matrix();
        scaleTransform.Scale(scaleX, scaleY);
        path.Transform(scaleTransform);

        // calculate the bounds of the scaled character, calculate the offset and
            apply this.
        currentBounds = path.GetBounds();
```

```
            Matrix translationTransform = new Matrix();
            translationTransform.Translate(targetRect.Left - currentBounds.Left,
                targetRect.Top - currentBounds.Top);
            path.Transform(translationTransform);

            // return the path.
            return path;
        }
    }

    // ...
```

Damit lassen sich geometrische Formen vieler unterschiedlicher Art implementieren.

Schwieriger ist an dieser Stelle die Anpassung bzw. Implementierung des Gradientenverlaufs, den wir als Hintergrund nutzen wollen. Das liegt daran, dass einige Methoden in der Klasse *AreaField* als private markiert sind und somit nicht überschrieben und angepasst werden können. Aus diesem Grund wurde hier basierend auf der Klasse *AreaField* aus dem DSL-Tools Framework eine neue Klasse angelegt[1] und an den entscheidenden Stellen angepasst (wesentlich hier ist die Funktion *GetBorderPath*, in der wie schon zuvor der *GraphicsPath* anzupassen ist). Diese Klasse ist im Listing B.3 dargestellt.

Listing B.3: Anpassung der DecisionGateShape Klasse

```
    // ...

namespace Tum.PESEditor.CustomCode.View.Shapes
{
    // Information aus Microsoft.VisualStudio.Modeling.Diagrams.dll
    public class DecisionGateShapeAreaField : AreaField
    {
        private static GraphicsPath borderPath;

        public DecisionGateShapeAreaField(string fieldName)
            : base(fieldName)
        { }

        // Microsoft.VisualStudio.Modeling.Diagrams.dll
        private bool IsGradientBrushRequested{ // ... }

        // Microsoft.VisualStudio.Modeling.Diagrams.dll
        private static GraphicsPath BorderPath{ // ... }

        // Anpassung
        private GraphicsPath GetBorderPath(ShapeElement parentShape)
        {
            RectangleF targetRect = RectangleD.ToRectangleF(parentShape.
                GeometryBoundingBox);

            RectangleF rectangleF = RectangleD.ToRectangleF(base.GetBounds(parentShape))
                ;
            GraphicsPath graphicsPath = DecisionGateShapeAreaField.BorderPath;

            graphicsPath.Reset();

            float intent = 0.13f;

            // draw shape
            graphicsPath.AddLine(new PointF(0, 0), new PointF(intent, targetRect.Height)
                );
            graphicsPath.AddLine(new PointF(intent, targetRect.Height), new PointF(
                targetRect.Width, targetRect.Height));
            graphicsPath.AddLine(new PointF(targetRect.Width, targetRect.Height), new
                PointF(targetRect.Width - intent, 0));
            graphicsPath.AddLine(new PointF(targetRect.Width - intent, 0), new PointF(0,
                 0));

            return graphicsPath;
        }

        // Microsoft.VisualStudio.Modeling.Diagrams.dll
```

1 Die Information zu dieser Klasse wurde über das Decompilieren der Framework DLL „Microsoft.VisualStudio.Modeling.Diagrams.dll“ dem DSL-Tools Framework entnommen.

```
        public override void DoPaint(DiagramPaintEventArgs e, ShapeElement parentShape){
             // ... }

        // Microsoft.VisualStudio.Modeling.Diagrams.dll
        internal class DrawHelper{ // ... }
    }
}

    // ...
```

Auf diese Art und Weise lassen sich Shapes mit benutzerdefinierter Darstellungsinformation in den DSL-Tools implementieren.

C How To: Drag&Drop vom ModelExplorer und der Visual Studio Toolbox

Im Folgenden wird beschrieben, wie man Drag&Drop-Mechanismen ausgehend vom ModelExplorer sowie der Visual Studio Toolbox implementiert und die so transportierten Daten zur Erstellung von Elementen nutzt. Die Vorgehensweise sieht dabei wie folgt aus

1. Anpassen des ModelExplorers, so dass ausgehend von diesem Drag&Drop-Vorgänge möglich werden.
2. Anpassen der Erstellung der Visual Studio Toolbox Items, damit wir in der Lage sind festzustellen, welches Toolbox Item am Drag&Drop-Vorgang beteiligt ist.
3. Anpassen der Drag&Drop-Routinen in den zulässigen Shape-Elementen, die als Container andere Elemente enthalten können (z.B. *Diagram*).

C.1 Anpassungen für den ModelExplorer

Der ModelExplorer implementiert das gesamte instanziierte Modell in einer baumartigen Struktur. Für diese muss das Drag&Drop aktiviert werden, was an dieser Stelle durch das Anpassen der Klasse *PESEditorExplorer*[1] realisiert werden kann. Hierzu muss in der Methode *CreateElementVisitor*, die während der Initialisierung der ModelExplorer Klasse aufgerufen wird und Modifikationen an ihren Kernelementen zulässt, das Ereignis *ItemDrag* des Elements *ObjectModelBrowser* implementiert werden. Der *ObjectModelBrowser* stellt die baumartige Struktur des ModelExplorers dar (über die Klasse *TreeView*), so dass bei der Implementierung nur noch lediglich die Drag&Drop-Routine zu aktivieren ist. (siehe Listing C.1).

Listing C.1: Anpassung der ModelExplorer Klasse

```
internal partial class PESEditorExplorer
{
    // ...

    /// <summary>
    /// Executed when the model explorer creates the tree element visitor.
    /// </summary>
    /// <returns>The tree element visitor.</returns>
    protected override IElementVisitor CreateElementVisitor()
    {
        this.ObjectModelBrowser.AllowDrop = true;

        this.ObjectModelBrowser.ItemDrag += new System.Windows.Forms.
            ItemDragEventHandler(ObjectModelBrowser_ItemDrag);


        return base.CreateElementVisitor();
    }

    void ObjectModelBrowser_ItemDrag(object sender, System.Windows.Forms.
        ItemDragEventArgs e)
    {
        DoDragDrop(e.Item, DragDropEffects.Move);
    }

    // ...
}
```

1 Die Klasse *PESEditorExplorer* implementiert den ModelExplorer in unserem Prototyp. Sie ist abgeleitet von der Klasse *ModelExplorerTreeContainer*, die ihrerseits die baumartige Struktur und das Darstellungsfenster im Visual Studio repräsentiert.

C.2 Anpassungen für die Visual Studio Toolbox

Der Visual Studio Toolbox wird durch die Methoden der Klasse *PESEditorToolboxHelper*[2] angepasst, so dass hier die für den Prototyp wesentlichen Toolbox Elemente erstellt und gegliedert in Tabs (Tabs gruppieren Elemente) der Toolbox hinzugefügt werden. Wesentlich hier ist die Anpassung bei der Erstellung der Elemente um gewisse Zusatzinformationen, die es uns später erlauben festzustellen, dass ein Element bzw. genau welches Element über einen Drag&Drop-Vorgang hinzugefügt werden soll.

Hinweis: Bei der Standarderstellung der Toolbox Elemente durch die Klasse *PESEditorToolboxHelper* wird bei jedem einzelnen Element Zusatzinformation in Form von *ElementGroupPrototype* hinterlegt. Hierdurch lässt sich feststellen, welches Element neu hinzugefügt werden soll (über die *DomainClassId* der Elementenklassen). Existieren allerdings mehrere Shapes für eine Elementenklasse, so wird es an dieser Stelle schwer zu entscheiden welches Shape genau verwendet werden soll.

Ferner existiert noch ein Feld *UserData*, welches allerdings nach der Erstellung nur gelesen werden kann. Insofern muss die Erstellung der einzelnen Elemente angepasst werden, um genau dieses *UserData*-Feld mit der entsprechenden Zusatzinformation zu beschreiben (in unserem Fall mit dem Namen des Toolbox Elements).

Der Vorgang der Anpassung der Erstellung von Toolbox Elementen (siehe Listing C.2) gliedert sich wie folgt

1. Standarderstellung der einzelnen Elemente
2. Erstellung einer Kopie dieser Elemente, wobei hier das Feld UserData beim *ElementGroupPrototype* entsprechend gesetzt wird
3. Rückgabe der kopierten Elemente

Listing C.2: Anpassung der Toolbox Elemente

```
public partial class PESEditorToolboxHelper
{
    // ...

    public override IList<ModelingToolboxItem> CreateToolboxItems()
    {
        IList<ModelingToolboxItem> l_modelingTollboxItemList = base.CreateToolboxItems()
            ;
        List<ModelingToolboxItem> l_modelingTollboxItemListNew = new List<
            ModelingToolboxItem>();

        // Add user data field with the id value to the toolbox elements
        using (Store store = new Store(this.ServiceProvider))
        {
            store.LoadDomainModels(typeof(CoreDesignSurfaceDomainModel),
                typeof(global::Tum.PESEditor.PESEditorDomainModel));

            using (Transaction t = store.TransactionManager.BeginTransaction("
                CreateToolboxItems"))
            {
                foreach (ModelingToolboxItem item in l_modelingTollboxItemList)
                {
                    if (item.Prototype != null)
                        if (item.Prototype.ProtoElements != null)
                            if (item.Prototype.ProtoElements.Count == 1)
                                l_modelingTollboxItemListNew.Add(new ModelingToolboxItem
                                    (item.Id, item.Position,
                                    item.DisplayName, item.Bitmap, item.TabNameId, item.
                                        TabName, item.ContextSensitiveHelpKeyword,
                                    item.Description,
                                    CreateElementToolPrototype(store, item.Prototype.
                                        ProtoElements[0].DomainClassId, item.Id),
                                    item.Filter));
                            else
                                l_modelingTollboxItemListNew.Add(item);
                        else
                            l_modelingTollboxItemListNew.Add(item);
                    else
                        l_modelingTollboxItemListNew.Add(item);
                }
```

2 Die Klasse *PESEditorToolboxHelper* erweitert die Visual Studio Toolbox um weitere Elemente, die bei der Verwendung des Prototyps notwendig sind.

```
                t.Rollback();
            }
        }

        // ...

        return l_modelingTollboxItemListNew;
    }

    private ElementGroupPrototype CreateElementToolPrototype(Store store, Guid
        domainClassId, string id)
    {
        ModelElement element = store.ElementFactory.CreateElement(domainClassId);
        ElementGroup elementGroup = new ElementGroup(store.DefaultPartition);
        elementGroup.AddGraph(element, true);
        elementGroup.UserData = id;

        return elementGroup.CreatePrototype();
    }

    // ...
}
```

C.3 Anpassungen der Drag&Drop-Routinen

Die Implementierung der Drag&Drop-Routinen wollen wir exemplarisch anhand der *Diagram*-Methoden vorstellen. Hier wird das Erstellen der *Specification* und *Module* Elemente realisiert, so dass auch Überprüfungen notwendigerweise implementiert werden müssen, die feststellen, ob die jeweiligen Elemente hinzugefügt werden dürfen oder nicht. Die Drag&Drop-Routine läuft hier wie folgt ab

1. *OnDragOver*: Feststellen ob das jeweilige Element überhaupt hinzugefügt werden darf. In diesem Fall geht das nur für die *Specification* und *Module* Elemente.
2. *OnDragDrop*: Hinzufügen der Elemente

Um festzustellen, ob ein bestimmtes Element hinzugefügt werden können oder nicht, müssen wir zunächst die jeweiligen Formate betrachten, die zum Transport der Daten während dem Drag&Drop-Vorgang verwendet werden. Diese sehen wie folgt aus

- Vom ModelExplorer: Jedes Element wird über *ModelElementTreeNode* transportiert. Diese Klasse enthält einen Verweis auf das jeweilige Element über die Eigenschaft *ModelElement*. Somit lassen sich hier leicht die Klasse und die Daten des jeweiligen Elements feststellen.
- Von der Visual Studio Toolbox: Hier wird ein Element über *ElementGroupPrototype* transportiert. Im Wesentlichen ergibt sich hier die Zuordnung eines Elements zur Klasse und dem zu verwendenen Shape über die *DomainClassId* sowie die *UserData* Felder.

Die Implementierung in unserem Prototyp in der *Diagram* Klasse lässt sich dem Listing C.3 entnehmen.

Listing C.3: Drag&Drop-Routine Methode in der Diagram Klasse

```
// Drag&Drop from the Visual Studio Toolbox and elements can be added onto the
    diagram ?
private bool IsAddingNewElement(DiagramDragEventArgs e)
{
    if (e.Prototype == null)
        return false;

    if (e.Prototype.ProtoElements == null)
        return false;

    if (e.Prototype.ProtoElements.Count == 1 && e.Prototype.UserData != null)
    {
        if (e.Prototype.UserData != null && e.Prototype.UserData.ToString() !=
            string.Empty)
        {
            if (e.Prototype.ProtoElements[0].DomainClassId == Module.DomainClassId
                &&
                (e.Prototype.UserData.ToString() == "ElementToolPESToolboxItem" ||
                 e.Prototype.UserData.ToString() == "ElementToolModuleToolboxItem"))
```

```
                {
                    return true;
                }
                else if
                {
                    // ...
                }
                else
                    return false;
            }
        }
        return false;
    }

    public override void OnDragOver(DiagramDragEventArgs e)
    {
        // ...
        if (IsAddingNewElement(e))
            e.Effect = DragDropEffects.Copy;
        else if (e.Data.GetDataPresent("Microsoft.VisualStudio.Modeling.Shell.
            ModelElementTreeNode"))
        {
            ModelElementTreeNode treeNode
                = e.Data.GetData("Microsoft.VisualStudio.Modeling.Shell.
                    ModelElementTreeNode") as ModelElementTreeNode;

            if (treeNode.ModelElement == null)
                base.OnDragOver(e);
            else
                if (((treeNode.ModelElement is PESSpecification || treeNode.ModelElement
                     is PMSpecification)
                    && e.HitDiagramItem.Shape is PESEditorDiagram)
                    ||
                    // ....
                    )
                {
                    ModelElement elem = treeNode.ModelElement;

                    // check if already added onto the diagram
                    foreach (PresentationElement pel in PresentationViewsSubject.
                        GetPresentation(treeNode.ModelElement))
                    {
                        base.OnDragOver(e);
                        return;
                    }

                    e.Effect = System.Windows.Forms.DragDropEffects.Move;
                }
                else
                    base.OnDragOver(e);
        }
        else
            base.OnDragOver(e);
    }

    public override void OnDragDrop(DiagramDragEventArgs e)
    {
        // ...

        if (IsAddingNewElement(e))
        {
            // drag & drop from the vs toolbox
            // ...

        }
        else if (e.Data.GetDataPresent("Microsoft.VisualStudio.Modeling.Shell.
            ModelElementTreeNode"))
        {
            // drag and drop from the model explorer
            // ...

        }
        else
            base.OnDragDrop(e);
    }
```

Literaturverzeichnis

[BEJ06] BUSCHERMÖHLE, R., H. ECKHOFF und B. JOSKO: *Success – Erfolgs- und Misserfolgsfaktoren bei der Durchführung von Hard- und Softwareentwicklungsprojekten in Deutschland*. Nummer ISBN 978-3-8142-2035-2. BIS-Verlag der Carl von Ossietzky Universität Oldenburg, 2006.

[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.

[FHKS08] FRIEDRICH, JAN, ULRIKE HAMMERSCHALL, MARCO KUHRMANN und MARC SIHLING: *Das V-Modell XT - Für Projektleiter und QS-Verantwortliche kompakt und übersichtlich*. Nummer ISBN: 978-3-540-76403-8 in *Informatik im Fokus*. Springer, 2. Auflage, aug 2008. available at http://www.springer.com/computer/programming/book/978-3-540-76403-8.

[Int04] INTERNATIONAL, STANDISCH GROUP: *Chaos Reports*. Online, 2004. `http://www.standishgroup.com/chaos_resources/index.php`.

[KTF09] KUHRMANN, MARCO, THOMAS TERNITÉ und JAN FRIEDRICH: *Das V-Modell XT anpassen*. Informatik im Fokus. Springer, 2009. (to appear).

[Pri09] PRIEUR, JEAN-MARC: *Announcing the Visual Studio 2010 DSL SDK Beta 1*. http://blogs.msdn.com/jmprieur/archive/2009/05/22/announcing-the-visual-studio-2010- dsl-sdk-beta-1.aspx, 2009.

[RG09] RECCHIA, P. und A. GUEROT: *Multiply Dsl points of view*. http://www.netfxfactory.org/blogs/papers/archive/2009/01/13/multiply-dsl-points-of-view.aspx, 2009.

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

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