INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Konzeption und prototypische Implementierung eines Werkzeugs zur Unterstützung der MNM-Dienstmodellierungsmethodik

David Schmitz

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Igor Radisic Harald Rölle

Abgabetermin: 30. September 2002

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Konzeption und prototypische Implementierung eines Werkzeugs zur Unterstützung der MNM-Dienstmodellierungsmethodik

David Schmitz

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Igor Radisic Harald Rölle

Abgabetermin: 30. September 2002

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 30. September 2002

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

In dieser Diplomarbeit wird eine Werkzeugunterstützung für das MNM-Dienstmodell entworfen und prototypisch implementiert. Ziel dabei ist es, den Benutzer bei der Anwendung der Methodik zur Erstellung einer Instanz des MNM-Dienstmodells zu einem gegebenem Anwendungsfall zu unterstützen. Das Werkzeug sollte als Erweiterung eines bestehenden OO/CASE-Werkzeuges entwickelt werden.

Zunächst wurden die MNM-Dienstmethodik und die für sie notwendigen Benutzerinteraktionen analysiert. Auf Basis dieser Analyse wurde ein erster Entwurf für die Architektur des zu entwickelnden Werkzeuges entwickelt. Aus dieser Architektur wurden Anforderungen abgeleitet, die einerseits zur Auswahl des bestehenden OO/CASE-Werkzeuges, das als Grundlage für das zu entwickelnde Werkzeug dienen sollte, und andererseits für den Entwurf des zu entwickelnden Werkzeuges selbst verwendet wurden. Bei der Auswahl des als Grundlage dienenden OO/CASE-Werkzeuges fiel die Wahl auf die Meta-Modellierungsumgebung DoME.

Auf Basis von DoME unter Berücksichtigung der vorher entwickelten Architektur und der deren Anforderungen wurde das Werkzeug entworfen und prototypisch implementiert. Hierbei wurde eine Workflowunterstützung geschaffen, die sowohl Spezifikation als auch Abarbeitung von praktisch beliebigen Workflows innerhalb von DoME ermöglicht. Diese generische Unterstützung wird speziell für die Spezifikation und Ausführung der MNM-Dienstmethodik verwendet. Weiterhin wurde die Repräsentation der Dienstmodell-Daten in DoME selbst ermöglicht. Hierbei wurden alle für das Dienstmodell benötigten Datentypen, wie z.B. UML-Diagramme, Freitext, Standard- und Dokumentreferenzen, berücksichtigt.

Die Tauglichkeit des Werkzeugs wurde mittels bereits vorhandener Dienstmodell-Instanzen getestet.

Das Werkzeug unterstützt die gesamte workfloworientierte Methodik des MNM-Dienstmodells. Hierfür werden Interaktionen mit dem Benutzer und – soweit möglich – automatische Workflow-Verarbeitungsschritte durchgeführt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Im heutigen Informatik-Umfeld spielen Dienstleistungen eine immer größere Rolle. Die Anzahl und Vielfalt der angebotenen Dienste hat hierbei in den letzten Jahren vor allem in Bereichen wie der Telekommunikation und dem Internet stark zugenommen. Damit einhergehend ist auch die Bedeutung des Dienstmanagements gestiegen. Um ein kundenorientiertes Management für einen Dienst zu gewährleisten, reicht in der Regel ein Management der Netze und Komponenten wie beim klassischen Netz- und Systemmanagement (siehe [HAN 99]) nicht mehr aus. Dies liegt einerseits an der großen Vielfalt, andererseits aber vor allem an der Komplexität der Dienste. Denn bei modernen Diensten müssen viele Abhängigkeiten, wie z.B. QoS-Parameter oder die Auslagerung von Teilen des Dienstes zu anderen Anbietern, berücksichtigt werden.

Beim Dienstmanagement wird daher ein Dienst selbst einschließlich aller seiner Abhängigkeiten, und nicht nur die zu seiner Erbringung benötigten Komponenten, betrachtet. Es gibt bereits eine Reihe von Ansätzen für das Dienstmanagement, die jedoch jeweils einige der folgenden Schwächen aufweisen (vgl. [GHH+ 01]), [GHK+ 01]):

- Ausrichtung nur auf spezielle Dienste (z.B. ATM, Multimedia), da kein konsistenter Top-Down-Aufbau
- Keine klare, einheitliche Terminologie
- Eine implementierungsorientierte, statt eine dienstorientierter Sicht, so dass eine gemeinsame Sichtweise von Dienstleister und Dienstnehmer nicht berücksichtigt wird
- Keine Berücksichtigung von organisatorischen Aspekte, wie z.B. Provider-Hierarchien
- Unvollständige Berücksichtigung des Dienst-Lebenszykluses

Um diese Schwierigkeiten beim Management von Diensten zu lösen, wurde am Lehrstuhl von Professor Hegering das MNM-Dienstmodell als ein universell anwendbares Dienstmodell zum Beschreiben von Diensten entwickelt. Es beschreibt alle wesentlichen Aspekte eines Dienstes und hat keine der oben genannten Schwächen. Zum Dienstmodell selbst wurde auch bereits eine Anwendungsmethodik erstellt, die vorgibt, wie das Dienstmodell auf einen bestimmten, gebebenen Fall anzuwenden ist.

Aber die Anwendung der Methodik gestaltet sich zum Teil schwierig: Die Methodik bestimmt Workflows aus mehreren aufeinanderfolgenden Aktivitäten, die jeweils Ein- und Ausgabe-Artefakte besitzen. Als Artefakte kommen hierbei z.B. UML-Diagramme, Freitext oder Artefakte zur Strukturierung anderer Artefakte vor. Die Ausgabe-Artefakte einer Aktivität sind oft Eingabe-Artefakte von später nachfolgenden Aktivitäten. Das heißt die Anwendungsmethodik des MNM-Dienstmodells erfordert eine workflow-orientierte Verwaltung vieler zum Teil komplexer Ein- und Ausgaben. Daher ist eine Werkzeugunterstützung für diese Anwendungsmethodik sinnvoll und wünschenswert.

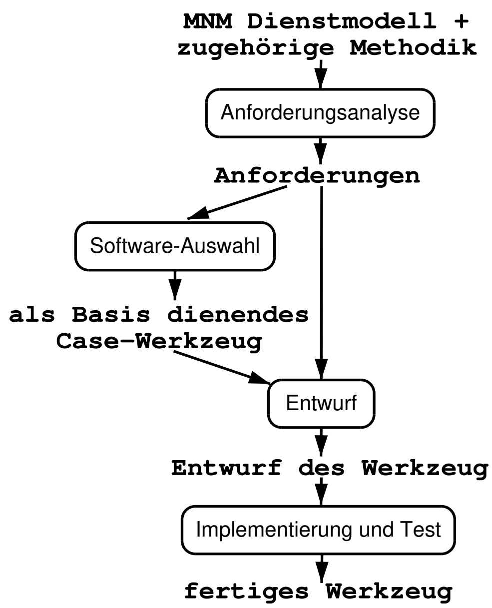

Abbildung 1.1: Überblick über die Vorgehensweise

1.1 Aufgabenstellung

In dieser Diplomarbeit soll eine Werkzeugunterstützung für die Anwendung des MNM-Dienstmodell geschaffen werden. Ziel des zu entwickelnden Werkzeuges ist die Unterstützung der in [GHH+ 02] spezifizierten Methodik zur Anwendung des Dienstmodells. Das Werkzeug soll entworfen und (möglichst vollständig) implementiert werden.

Dabei sollen alle für die Methodik beschriebenen Workflows, bestehend aus Aktivitäten und eingehenden bzw. erzeugten Informationen, sogenannten Artefakten, unterstützt werden. Als Artefakte kommen dabei Daten in unterschiedlicher Form, vor allem UML-Diagramme, z.B. Klassen-, Use-Case-, Aktivitäts- und Kollaborationsdiagramme, aber auch Beschreibungen/Anforderungen in nicht näher spezifizierter Form (Listen, reiner Text, ...), vor.

Es muss jeweils festgelegt werden, wie die Artefakte verwaltet und dargestellt (GUI) werden. Insbesondere ist die automatische Übernahme von Artefakten, die in vorangegangenen Aktivitäten erstellt wurden und die für eine spätere Aktivität als Eingabe dienen, in die spätere Aktivität wünschenswert.

Einerseits wurde das Dienstmodell selbst nach dem Rational Unified Process (RUP, siehe [Kruc 00])) entworfen. Andererseits wird eine Instanz des Dienstmodells zum Großteil mit Hilfe von UML (siehe [OMG 01-02-14]) spezifiziert, d.h. das Werkzeug muss insbesondere UML unterstützen. Es bietet sich daher an, das Werkzeug als Erweiterung eines bestehenden OO/CASE-Werkzeuges zu entwickeln.

1.2 Vorgehensweise

Der prinzipielle Ansatz ist die Erweiterung eines bestehenden OO/CASE-Werkzeuges, um die Erstellung von Instanzen des Service Modells nach der Modellierungsmethodik zu unterstützen. Dabei gliedert sich das Vorgehen grob in die Abschnitte Anforderungsanalyse, Software-Auswahl, Entwurf, Implementierung und Test (siehe Abb. 1.1). Diese werden nun genauer erläutert.

Anforderungsanalyse: Zu Beginn sollen die Anforderungen an das zu entwickelnde Werkzeug identifiziert werden. Dazu wird einerseits die Anwendungsmethodik selbst betrachtet, da das Werkzeug diese vollständig unterstützen bzw. enthalten soll. Weiterhin werden hierfür aber auch die sich aus der Anwendungsmethodik ergebenden Interaktionen des Anwendungs-Modellierer mit dem Werkzeug analysiert.

Analyse und Auswahl der zu erweiternden Software: Vorhandene Software, d.h. OO/CASE-Werkzeuge, die als Grundlage für das Werkzeug dienen könnten, werden betrachtet und miteinander verglichen. Mit Hilfe des in der Anforderungsanalyse gefundenen Anforderungskatalogs wird dann das am besten geeignetste ausgewählt. Dieses wird dann im weiteren Verlauf der Diplomarbeit erweitert, um die Dienstmodellierung zu unterstützen.

Entwurf: Nachdem eines der vorhandenen CASE-Werkzeuge als Grundlage für das Werkzeug ausgewählt worden ist, soll das Werkzeug genau konzipiert und entworfen werden. Dies beinhaltet einerseits den Entwurf des Anwendungskerns inkl. Datenstrukturen für Workflows, Aktivitäten und Artefakte, etc. Andererseits wird hier auch die Benutzerschnittstelle, d.h. die Art und Weise der Dateneingabe durch den Benutzer bzw. die Darstellung und Präsentation aller Daten dem Benutzer gegenüber, entworfen.

Implementierung und Test: Anhand des vorausgegangenen Entwurfs wird dann, prototypisch der Anwendungskern sowie die GUI des Werkzeugs implementiert und getestet. Zum Testen werden bereits früher am Lehrstuhl entwickelte Instanzen des Dienstmodells mit Hilfe des Werkzeuges erstellt.

1.3 Gliederung der Ausarbeitung

Diese Ausarbeitung ist folgendermaßen gegliedert:

Zunächst wird in Kapitel 2 ein Überblick über das MNM-Dienstmodell sowie die zugehörige Anwendungsmethodik gegeben. Außerdem werden die Freiräume der Methodik, d.h. die Aspekte die die Methodik nicht vollständig festlegt, besprochen.

Dann werden in Kapitel 3 Anwendungsmethodik sowie die dafür notwendigen Benutzerinteraktionen analysiert. Auf Basis dieser Analyse wird dann ein erster Entwurf für die Architektur des Werkzeugs entwickelt. Daran anschließend werden aus dieser Architektur Anforderungen abgeleitet, die das OO/CASE-Tool, das als Grundlage für das Werkzeug dient, erfüllen muss.

Das Kapitel 4 befasst sich mit der Auswahl des zur Entwicklung verwendeten bzw. erweiterten OO/CASE-Werkzeuges. Hierbei werden die OO/CASE-Werkzeuge bezüglich der Anforderungen geprüft, die im vorangegangenen Kapitel identifiziert wurden. Die Meta-Modellierungsumgebung *DoME* erweist sich hierbei als am besten geeignet und wird daher als Grundlage für das Werkzeug gewählt.

Anschließend wird in Kapitel 5 der Entwurf des Werkzeuges behandelt. Hierbei wird zunächst auf *DoME* allgemein und dann auf das Design des Werkzeugs selbst eingegangen. Insbesondere wird die Umsetzung der in 3 entwickelten Werkzeugarchitektur innerhalb von *DoME* betrachtet.

Über die prototypische Implementierung, die auf dem vorangegangenen Entwurf aufbaut, wird dann in Kapitel 6 ein Überblick gegeben. In Anhang A werden die Details der Implementierung genauer betrachtet.

Auf den Benutzung sowie den Test des prototypisch implementierten Werkzeuges wird abschließend in Kapitel 7 eingegangen. Hier werden einige der Tests beispielhaft besprochen. Abschließend wird in Kapitel 8 eine Zusammenfassung der Arbeit sowie ein Ausblick gegeben.

Kapitel 2

MNM-Dienstmodell und seine Anwendungsmethodik

Zunächst werden nun die Eigenschaften, Anforderungen und der Aufbau des MNM-Dienstmodell vorgestellt. Danach wird auch die zum Dienstmodell zugehörige Anwendungsmethodik beschrieben, die zeigt, wie man eine Instanz des MNM-Dienstmodells zu einem gegeben Fall erstellt. Abschließend wird betrachtet, welche Freiräume diese Methodik erlaubt, d.h. welche Aspekte in der Methodik nicht fest vorgegeben sind. Diese geben Anhaltspunkte für notwendige Designentscheidungen in den nachfolgenden Kapiteln.

Die ursprüngliche Beschreibung des Dienstmodells findet sich in in [GHH+ 01] und [GHK+ 01]. Die zugehörige Anwendungsmethodik wurde erstmals in [GHH+ 02] beschrieben.

2.1 MNM-Dienstmodell

Bisherige Ansätze zur Modellierung von Diensten weisen viele Schwächen auf. Oft sind sie z.B. nur für spezielle Szenarien konzipiert, haben keine durchgängige Terminologie oder können bestimmte Abhängigkeiten wie Provider-Hierarchien oder QoS-Parameter nicht adäquat beschreiben (vgl. [GHH+ 01]), [GHK+ 01]).

Daher wurde am Lehrstuhl von Professor Hegering das sogenannte MNM-Dienstmodell entwickelt mit dem Ziel eine klare Terminologie einzuführen und für jeden gegebenen Dienst eine generische, universelle Beschreibung und damit ein allgemeines Verständnis (vor allem zwischen Kunde und Anbieter) zu ermöglichen.

Im Folgenden wird dieses Dienstmodell genauer beschrieben und erläutert.

2.1.1 Übersicht über das MNM-Dienstmodell

Das MNM-Dienstmodell bietet die Möglichkeit Dienste universell zu modellieren. Zu jedem bestimmten, gegebenen Dienst kann eine Instanz des MNM-Dienstmodells erstellt werden, die den gegebenen Dienst beschreibt.

Diese Dienstbeschreibung kann zwischen Dienstnehmerseite (Kunde) und Dienstleisterseite (Anbieter, Diensterbringer) vereinbart bzw. ausgetauscht werden, um ein gemeinsames, einheitliches Verstehen aller wesentlichen Dienstbestandteile — soweit für die jeweilige Seite jeweils notwendig — zu ermöglichen.

Auch bei Provider-Hierarchien, wenn ein Diensterbringer selbst zum Dienstnehmer eines anderen Diensterbringers wird, nämlich indem er den Dienst des anderen Diensterbringers zur Erbringung seines eigenen

Dienstes nutzt, kann die einheitliche Dienstbeschreibung unter den beteiligten Diensterbringern ein gemeinsames Verständnis des Dienstes schaffen.

Es wurden bei der Entwicklung des Dienstmodells folgende Anforderungen berücksichtigt:

- Es soll eine allgemeine und abstrakte Dienstdefinition möglich sein, d.h. für beliebige Dienste anwendbar sein. Es sollte nicht auf bestimmte Bereiche wie etwa Multimedia, ATM oder Telekommunikation beschränkt sein.
- Eine Reihe verschiedener organisatorischer und funktionaler Abhängigkeiten, wie z.B. QoS-Parameter oder Provider-Hierarchien sollte in mit dem Modell beschreibbar sein.
- Es sollte eine klare Trennung zwischen dem Dienst selbst, d.h. dem Dienst aus der Sicht des Kunden, und der Dienstrealisierung, d.h. der detaillierten Beschreibung, wie der Dienst genau erbracht wird (z.B. welche Ressourcen oder anderen Provider werden genutzt), möglich sein.
- Das Management des Dienstes sollte beschreibbar sein. Eine Unterscheidung zwischen dem Management des Dienstes und dem Dienst an sich, d.h. der eigentlichen Funktionalität des Dienstes, sollte möglich sein.
- Das Modell sollte alle Phasen des gesamten Dienstlebenszykluses berücksichtigen. Ähnlich wie es im Bereich der Software-Entwicklung verschiedenen Lebenszyklus-Phasen der Entwicklung gibt, so unterscheidet man bei einem Dienst die Lebenszyklusphasen Entwurf (design), Verhandlung (negotiation), Bereitstellung (provisioning), Nutzung (usage) und Deinstallation (deinstallation).

Für das MNM-Dienstmodell wurde eine klare, einheitliche, aber auch allgemein anwendbare Terminologie der vorkommenden Begriffe eingeführt: Vor allem gibt es bei jedem Dienst im wesentlichen zwei beteiligte Seiten - den Dienstleister (provider) und Dienstnehmer (customer) - die sich gegenüberstehen. Während der ganzen Lebensdauer des Dienstes finden Interaktionen zwischen diesen beiden Seiten statt. Da es sehr viele, unterschiedliche Interaktionen während eines gesamten Dienstlebenszykluses geben kann, klassifiziert man die Interaktionen zunächst sowohl nach den Phasen des Lebenszyklus sowie nach einer Prozessklassifikation, die auf der Telecom Operations Map (TOM, siehe [TOM1.1]) und den OSI System Management Functional Areas (OSI SMFA's) (siehe [ISO 10040]) basiert. Grob lassen sich dabei die Prozessklassen in die beiden Bereiche (Dienst-)Nutzung und (Dienst-)Management einteilen. In Abb. 2.1 sind die Prozessklassen und ihre Zuordnung zu Lebenszyklus-Phasen genauer dargestellt.

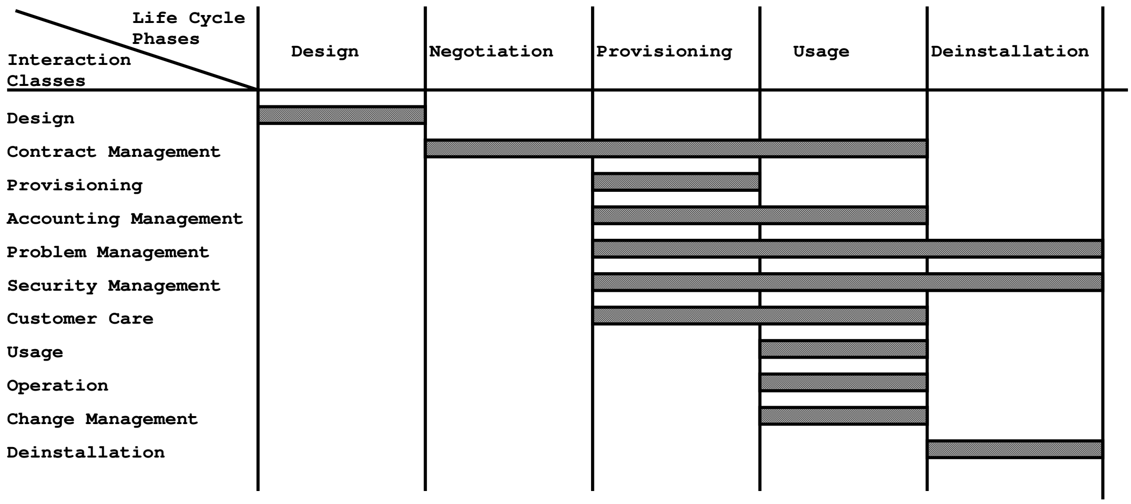

Abbildung 2.1: Lebenszyklus und Dienstklassen

Bei jeder der Interaktionen tritt sowohl die Seite des Dienstleisters als auch die des Dienstnehmers in einer bestimmten Rolle auf. Grundsätzlich lassen sich hier die Rollen Nutzer (user) und Kunde (customer) auf der

Seite des Dienstnehmers und die Rolle Anbieter (provider) auf der Seite des Dienstleisters unterscheiden. Dabei tritt die Dienstnehmerseite im Fall des Managements als Kunde und im Fall der reinen Nutzung als Nutzer auf. Diese Unterscheidung kann ggf. verfeinert werden.

Das Dienstmodell bietet überdies die Möglichkeit einer rekursiven Anwendung. Verwendet ein erster Dienstleister nämlich zur Erbringung des Dienstes die Dienste eines zweiten Dienstleisters, sognannte Sub-Dienste, so tritt in diesem Fall der erste Dienstleister in der Rolle des Nutzers bzw. Kunden, d.h. als Dienstnehmer, gegenüber dem anderen Dienstleister auf. Auch der zweite Dienstleister kann wiederum Sub-Dienste eines dritten Dienstleisters verwenden und so weiter. So eine Konstellation der Dienstleister nennt man Provider-Hierarchie. Das MNM Service Model wurde konzipiert, dass es durch rekursive Anwendung der Modellierung eine derartige Abhängigkeit der Dienste untereinander passend darstellt.

Im Weiteren wird nun der genaue Aufbau des MNM-Dienstmodells eingehend beschrieben.

Zur Übersichtlichkeit und um der geforderten Trennung von Dienstrealisierung und dem Dienst aus der Sicht des Kunden gerecht zu werden, wird das Dienstmodell in drei verschiedene Teile bzw. Views, nämlich das Basis-Modell, die Dienstsicht und die Realisierungssicht, aufgeteilt:

Das Basis-Modell dient als grober Überblick über den Dienst und zeigt den Aufbau des Dienstes aus Sub-Diensten und die auftretenden Rollen in Bezug auf den Dienst und die Sub-Dienste. Diese globale Sicht wird dann in zwei weiteren Sichten, der Dienstsicht als gemeinsame Sicht von Dienstleister- und Dienstnehmerseite und der Realisierungssicht als dienstleister-interne Sicht, verfeinert.

Im Folgenden werden diese drei verschiedenen Teile bzw. Sichten des MNM-Dienstmodell genauer behandelt.

2.1.2 Basis-Modell

Das Basis-Modell (Basic Model) (Abb. 2.2) stellt eine allgemeine Übersicht über den modellierten Dienst, den sogenannten Hauptdienst, dar. Es zeigt den Aufbau des Hauptdienstes aus Sub-Diensten und die Beziehung zwischen den in verschiedenen Rollen bezüglich des Haupt- und der Sub-Dienste auftretenden Entitäten (Dienstnehmer und Dienstleister) untereinander.

Die auftretenden Entitäten werden in der Realität von juristischen Personen (legal entities) wahrgenommen.

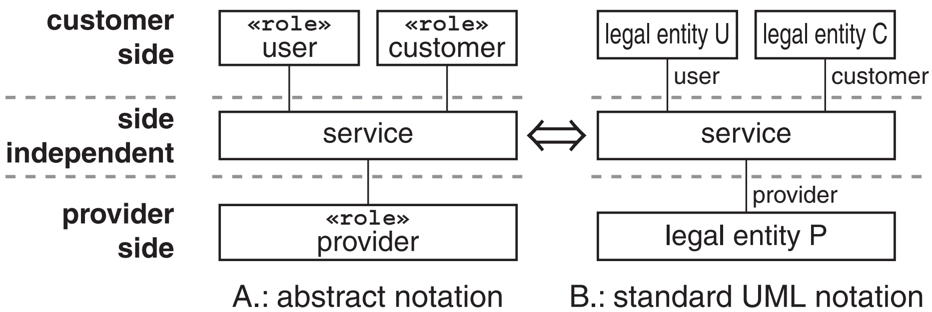

Abbildung 2.2: Basis-Modell (Basic Model)

Das Basis-Modell enthält also aus dem Hauptdienst und seine Sub-Dienste, sowie alle in bestimmten Rollen auftretetenden Entitäten. Sowohl für den Haupt- wie auch jeden Sub-Dienst treten dann Entitäten in den fundamentalen Rollen user, customer oder provider in Bezug auf den jeweiligen Dienst auf. Die Rollen user und customer stellen dabei jeweils die Dienstnehmerseite des jeweiligen Dienstes dar, die Rolle des providers die Dienstleisterseite.

Dargestellt wird das Basis-Modell in Form eines Klassendiagramm der Unified Modeling Language (UML). Die Haupt- und Sub-Dienste und auftretende Entitäten werden hierbei als Klassen modelliert.

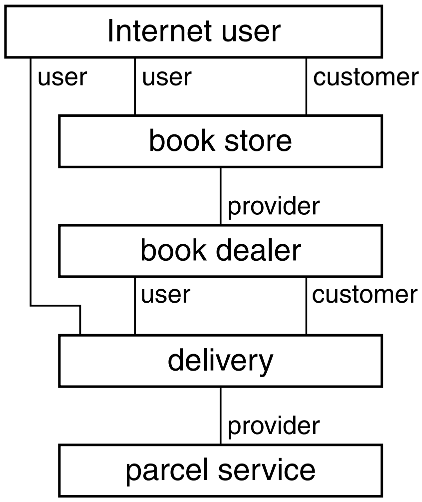

Abbildung 2.3: Beispiel für ein Basis-Modell – Internet-Buchhändler

Zwischen den Diensten und Entitäten zeigen Assoziationen an, das die Entität in einer Rolle bezüglich dem Dienst auftritt.

Für das Basis-Modell gibt es eine abstrakte Form (abstract notation), bei der die Rollen bezüglich eines Dienstes noch nicht an bestimmte Entities gebunden sind, und eine nicht abstrakte, instanziierte Form (instantiated notation), bei der jede Rolle von einem bestimmten Entity wahrgenommen wird.

In der abstrakten Form werden die Rollen eines Dienstes als eigene Klassen modelliert und durch das zusätzlich eingeführte Stereotyp <<role>> markiert. In der instanziierten Form werden dagegen die Entitäten als Klassen dargestellt und die Rollen in der sie bezüglich eines Dienstes auftreten – wie in UML üblich – als Rollennamen an die Assoziationsenden der Entitäten gestellt. Beide Darstellungen sind bezüglich der Rollen, in der Entities bezüglich eines Dienstes auftreten können, äquivalent.

In Abb. 2.3 ist ein Beispiel für ein Basis-Modell, dass einen Internet-Buchhändler beschreibt. Dabei nutzt der Buchhändler selbst einen Paket-Dienst als Sub-Dienst, tritt also in der Rolle des Nutzers und Kunden gegenüber dem Paket-Dienst auf. Der Kunde des Buchhändlers tritt auch in der Rolle des Nutzers gegenüber dem Paket-Dienst auf, da er über diesen das Buch geliefert bekommt.

Das grobe Basis-Modell wird durch die Dienstsicht (Service View), die die gemeinsame Sicht von Dienstnehmer und Dienstleister auf den Hauptdienst modelliert, und die Realisierungssicht (Realization View), die die dienstnehmer-interne Realisierung des Hauptdienstes zeigt, verfeinert. In den folgenden beiden Abschnitten werden diese Views behandelt.

2.1.3 Dienstsicht

Beim MNM-Dienstmodell dient die Dienstsicht (Service View) (Abb. 2.4) als gemeinsame Sicht von Dienstleister- und Dienstnehmerseite dazu, ein gemeinsames Verständnis zwischen beiden Seiten über den spezifizierten (Haupt)-Dienst zu erlangen. Hierbei sind nur realisierungs-unabhängige Aspekte wichtig, da die Realisierung des Dienstes für die Dienstnehmerseite transparent sein soll.

Bei der Dienstsicht wird strikt zwischen der Dienstnehmerseite, der Dienstleisterseite und der Beschreibung der Dienst selbst, der unabhängig von den beiden anderen Seiten soll, unterschieden.

Die Spezifikation des Dienst selbst besteht hier im Wesentlichen aus fünf Bestandteilen: Nutzfunktionalität, Managementfunktionalität, Dienstzugangspunkt, Customer-Service-Management-Zugangspunkt und QoS-Parameter.

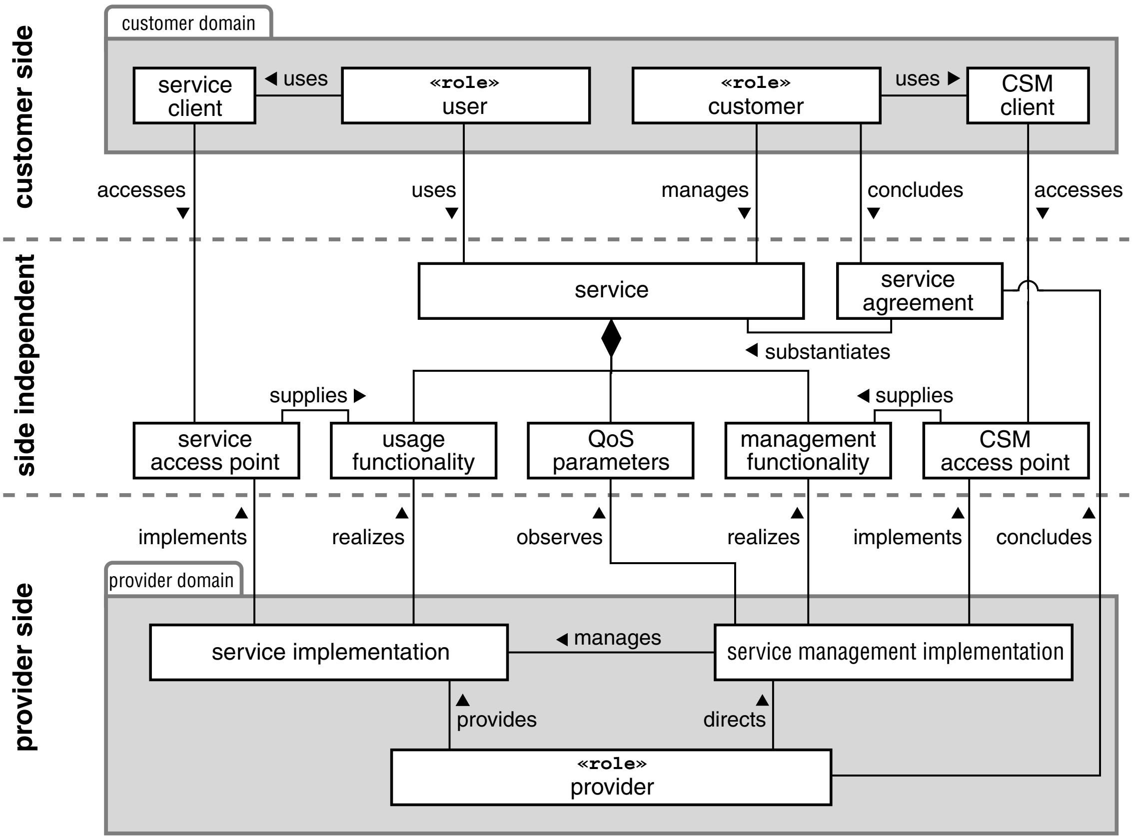

Abbildung 2.4: Dienstsicht (Service View)

Der Dienst gliedert sich grob in die Nutzung des Dienstes und das Dienstmanagement. Beide Teile werden auf der Dienstleisterseite vom Anbieter in Form der Dienstimplementierung (service implementation) und der Dienstmanagement-Implementierung (service management implementation) bereitgestellt bzw. gesteuert. Das Dienstmanagement managt dabei die Dienstnutzung. Diese Implementierungen werden hier jedoch nicht detaillierter betrachtet.

Sowohl für die (Dienst)-Nutzung als auch für das (Dienst)-Management wird getrennt die Funktionalität beschrieben — für die Dienstnutzung die Nutzfunktionalität (usage functionality) und für das Management die Managementfunktionalität (management functionality).

Die Funktionalität beschreibt jeweils alle konkreten Interaktionen zwischen Dienstleister und Dienstnehmerseite, die die Nutzung bzw. das Management des Dienstes ausmachen.

Damit die Dienstnehmerseite jeweils auf diese Funktionalität zugreifen kann, werden jeweils Schnittstellen — für die Nutzung ein Dienstzugangspunkt (service access point) und für das Management ein Customer-Service-Management-Zugangspunkt (customer service management access point — spezifiziert. Customer Service Management wird im Weiteren mit CSM abkürzt. Die Beschreibungen der Schnittstellen kann je nach Dienst aus Dienstprimitiven, Protokollen aber auch Hardware-Schnittstellen oder gar Steckerbelegungen bestehen.

Die Dienstimplementierung realisiert die Nutzfunktionalität und implementiert den Dienstzugangspunkt, der die Nutzfunktionalität bereitstellt. Analog realisiert die Dienstmanagement-Implementierung die Managementfunktionalität und implementiert den CSM-Zugangspunkt, der die Managementfunktionalität bereitstellt.

Auf Dienstnehmerseite werden jeweils Clients definiert, die auf die Zugangspunkte zugreifen können. Die Clients liegen dabei im Gegensatz zu den Schnittstellen im Verantwortungsbereich der Dienstnehmerseite und nicht der Dienstleisterseite. Für die Nutzung gibt es einen Dienst-Client (service client), der vom Nutzer verwendet wird, um über den Dienstzugangspunkt auf den Nutzdienst zuzugreifen. Genauso existiert ein ein CSM-Client mit dem der Kunde über CSM-Zugangspunkt auf das Dienstmanagement zugreifen kann.

Dies heißt insbesondere auf der Dienstnehmerseite werden die eigentliche Dienstnutzung von der Rolle des Nutzers und das Dienstmanagement von der Rolle des Kunden „verwendet“. Der Nutzer nutzt den (Nutz)-Dienst, der Kunde managt den Dienst.

Weiterhin werden QoS-Parameter festgelegt, die von der Dienstmanagement-Implementierung zu überwachen sind, um die Qualität des Dienstes zu sichern. Der Dienstnehmerseiter wird (je nach festgelegter Managementfunktionalität) die Möglichkeit gegeben, über das Dienstmanagement die Einhaltung dieser QoS-Parameter zu überprüfen. Hier werden jedoch noch keine konkreten Werte, sondern nur die Typen der QoS-Parameter festgelegt.

Um den Dienst dann zwischen Dienstleister- und Dienstnehmerseite, genauer zwischen der Rolle des Anbieters und der des Kunden, vertraglich festzulegen, dient die Dienstvereinbarung (service agreement). Hierin werden alle oben beschriebenen fünf Bestandteile inkl. der Clients zusammengefasst und der spezifizierte Dienst vollständig beschrieben. Das heißt für QoS-Parameter und andere Werte werden genaue Angaben gemacht und sonstige rechtliche, organisatorische Einzelheiten exakt festgelegt. Die Dienstleisterseite muss insbesondere die festgelegten QoS-Parameter einhalten.

2.1.4 Realisierungssicht

In der Realisierungssicht (Realization View) (Abb. 2.5) wird die dienstleisterabhängige Realisierung des (Haupt-)Dienstes beschrieben. Hier wird die Dienstimplementierung und die Dienstmanagement-Implementierung der Dienstleisterseite, die in der Dienstsicht nur erwähnt werden, detaillierter dargestellt. Es wird spezifiziert, wie der (Haupt-)Dienst mit Hilfe von eigenen Ressourcen des Anbieters und Sub-Diensten anderer Anbieter erbracht wird.

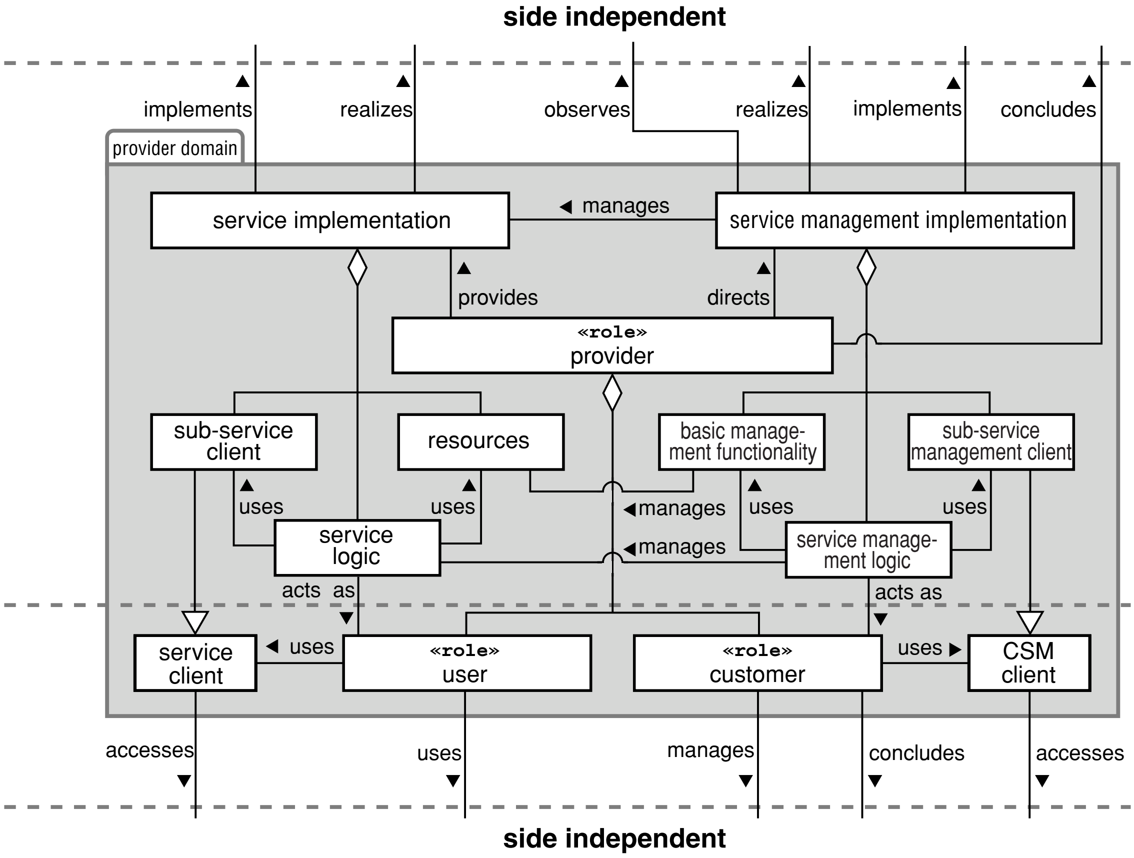

Abbildung 2.5: Realisierungssicht (Realization View)

In dieser Sicht tritt der Anbieter (des Hauptdienstes) einerseits bezüglich des Hauptdienstes in der Rolle des Anbieters gegenüber der Dienstnehmerseite des Hauptdienstes auf. Andererseits tritt der Anbieter, wenn er auf Sub-Dienste anderer Anbieter zurückgreift, in der Rolle des Nutzers bzw. Kunden auf der Dienstnehmerseite des Sub-Dienstes gegenüber dem anderen Anbieter auf.

Die Realisierungssicht (des Hauptdienstes) besteht daher einerseits aus der detaillierten Dienstleisterseite des Hauptdienstes, passend zur Dienstsicht dieses (Haupt-)Dienstes. Andererseits besteht sie für jeden verwendeten Sub-Dienst auch aus einer Dienstnehmerseite für diesen Sub-Dienst, für die jeweils wieder (rekursiv) eine passende Dienstsicht erstellt werden kann. Die Realisierungssicht stellt daher sowohl die anbieter-internen Vorgänge als auch die Verbindung zu Anbietern von Sub-Diensten detailliert dar.

Wie bei der Dienstsicht werden Dienstnutzung und Dienstmanagement (des Hauptdienstes) jeweils getrennt betrachtet. Beide werden durch Nutzung von anbieterinternen Ressourcen und ggf. Sub-Diensten erbracht. Im Fall der Dienstnutzung sind dies interne Ressourcen wie z.B. Personal, Know-How, Hardware, Software etc. Für das Dienstmanagement wird intern die sogenannte Basic Management Functionality (BMF), also Netz-, System- und Anwendungs-Management (siehe [HAN 99]) zum Management der eigenen Ressourcen verwendet.

Jeweils können aber auch Sub-Dienste zur Diensterbringung verwendet werden. Diese werden im Fall der Dienstnutzung durch Sub-Dienst-Clients, im Fall des Managements durch Sub-Dienstmanagement-Clients angesprochen. Der Sub-Dienst-Client stellt dabei eine Spezialisierung des Dienst-Clients der Dienstnehmerseite des Sub-Dienstes, der Sub-Dienstmanagement-Client entsprechend eine Spezialisierung des CSM-Clients dar. Der Anbieter selbst tritt bei der Dienstnutzung des Sub-Dienstes in der Rolle des Nutzers auf, beim Dienstmanagement des Sub-Dienstes in der Rolle des Kunden.

Die Spezifikation der Dienstimplementierung (des Hauptdienstes) setzt sich nun aus den eigenen Ressourcen, den verwendeten Sub-Clients und der Dienstlogik (service logic), die alle anbieter-internen Abläufe zur Erbringung des Nutzdienstes aus Ressourcen und Sub-Diensten beschreibt, zusammen. Ebenso wird die Dienstmanagement-Implementierung durch Festlegung der BMF, den verwendeten Sub-Management-Clients und einer Dienstmanagement-Logik (service management logic) zur Beschreibung der internen Abläufe zur Erbringung des Dienstmanagements bestimmt.

Basis-Modell, Dienstsicht und Realisierungssicht zusammen bilden das gesamte MNM-Dienstmodell.

Im Folgenden wird die Methodik zur Anwendung des Dienstmodells erklärt.

2.2 Anwendungsmethodik des MNM-Dienstmodells

Das MNM-Dienstmodell ermöglicht eine Beschreibung von beliebigen Diensten. Jedoch stellt sich die Frage, wie man zu einem gegebenen Dienst eine passende Instanz des Dienstmodells erhält, also wie man das Dienstmodell auf einen gegebenen Fall anwendet. Am Lehrstuhl wurde daher zu dem MNM-Dienstmodell bereits basierend auf gesammelten Erfahrungen und dem Rational Unified Process (RUP, siehe [JBR 99], [Kruc 00]) eine Methodik für die Anwendung des Modells entwickelt.

In diesem Abschnitt wird nun diese Methodik dargestellt (vgl. [GHH+ 02]).

2.2.1 Übersicht über die Anwendungsmethodik

Mit der Anwendungsmethodik kann man das Dienstmodell auf einen beliebigen, gegebenen Dienst anwenden und eine zum gegebenen Dienst passende Instanz des Dienstmodells erstellen.

Es gibt prinzipiell drei verschiedene Fälle für die Anwendungs des Dienstmodell (Modellierungsfälle, modeling cases):

- Reverse Engineering
- Ausschreibung (bid invitation)
- Angebot (offering)

Beim Reverse Engineering wird eine bereits bestehende Realisierung eines Dienstes modelliert. Dies ist sinnvoll z.B. bei Outsourcing-Entscheidungen oder Neu-Organisation des Dienstes. Bei einer Ausschreibung gibt der Kunde den Dienst, also die Dienstsicht des Dienstes vor, den er gerne hätte. Beim Angebot gibt der Anbieter den Dienst vor, den er bereitstellen möchte.

Je nachdem, ob existierende Realisierungen des zu modellierenden Dienstes vorhanden sind bzw. bei einem neu-einzuführenden Dienst bereits vorhandenen Komponenten verwendet werden oder ob ein Dienst von Grund auf neu konzipiert wird, ist zu entscheiden, ob die Modellierung mit einer Bottom-Up- oder einer Top-Down-Strategie durchgeführt wird. Bei der Bottom-Up-Strategie wird zunächst die Realisierungssicht erstellt und dann daraus durch Abstraktion die Dienstsicht gewonnen. Dagegen bei der Top-Down-Strategie wird zuerst die Dienstsicht erstellt und diese dann verfeinert zur Realisierungssicht. Im Fall des Reverse Engineerings verwendet man daher die Bottom-Up-Strategie. Bei der Ausschreibung wird dagegen die Top-Down-Strategie genutzt, da man zunächst nur eine Dienstsicht benötigt. Beim Angebot sind beide Vorgehensweisen möglich. Die Entscheidung hängt hier z.B. davon ab, ob bereits vorhandene Komponenten für den Dienst verwendet werden oder der Dienst von Grund auf neu entwickelt wird.

Die Anwendungsmethodik gibt für beide Vorgehensweisen jeweils drei aufeinanderfolgende Workflows an, um die Instanz des Dienstmodells zum gegebenen Dienst zu erstellen.

Bei jedem der Workflows wird einer der Teile bzw. Sichten des MNM-Dienstmodells erzeugt: Stets wird zuerst im Basis-Modell-Workflow (basic model workflow) das Basis-Modell erstellt. Dann folgen

Dienst- und Realisierungssicht - die Reihenfolge der beiden hängt von der Vorgehensweise ab. Beim Top-Down-Vorgehen wird zunächst im Dienstsicht Top-Down-Workflow (service view top-down workflow) die Dienstsicht, danach im Realisierungssicht Top-Down-Workflow (realization view top-down workflow) die Realisierungssicht generiert. Umgekehrt beim Bottom-Up-Vorgehen, hier wird im Realisierungssicht Bottom-Up-Workflow (realization view bottom up workflow) die Realisierungssicht und dann die Dienstsicht im Dienstsicht Bottom-Up-Workflow (service view bottom-up workflow) erstellt (siehe Abb. 2.6).

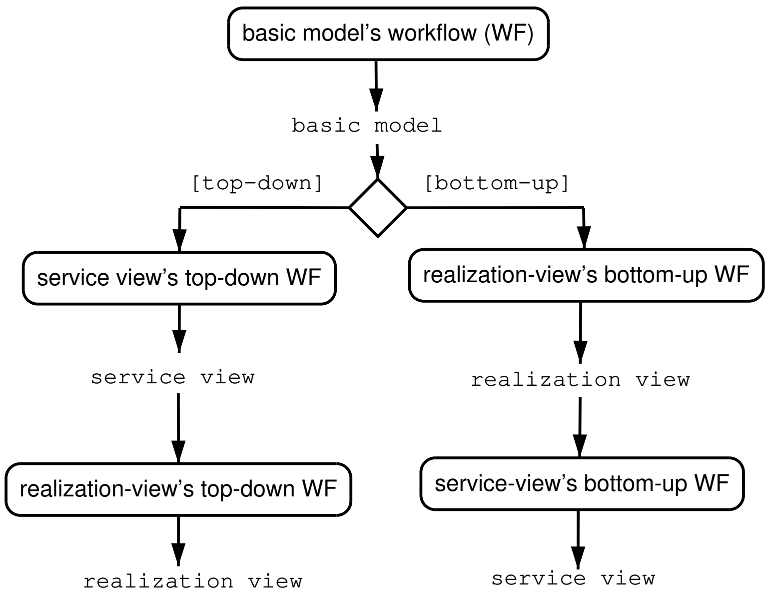

Abbildung 2.6: Workflow-Übersicht

Jeder Workflow besteht selbst aus mehreren Aktivitäten. Jede Aktivität hat sogenannte Artefakte als Ein- und Ausgabe. Artefakte, die Ausgabe einer Aktivität sind, können in einer späteren Aktivität wieder als Eingabe dienen. Es gibt es zwei verschiedene Typen von Artefakten bezüglich der Eingabe. Einerseits gibt es Artefakte die nur der Strukturierung der ausgegebenen Artefakte dienen. Sie sind weitgehend unabhängig vom speziellen Dienst. Beispiele hierfür sind die Phasen des Dienstlebenszyklus oder eine Prozessklassifikation. Die anderen Artefakte sind i.A. dienstspezifisch, wie z.B. UML-Use-Case-Diagramme, Beschreibungen in Freitext, Referenzen zu Standards. Diese Artefakte stellen im Endeffekt die erstellte Instanz des Dienstmodells dar.

In den folgenden Abschnitten werden die verschiedenen Workflows bzw. Aktivitäten mit ihren Artefakten näher betrachtet.

2.2.2 Basis-Modell-Workflow

In jedem Fall wird zu Beginn mit dem Basis-Modell-Workflow (basic model workflow) (Abb. 2.7) eine erste Version des Basis-Modells erstellt. Diese muss evtl. später vor allem bei der Top-Down-Strategie noch erweitert werden.

Wie in Abschnitt 2.1.2 beschrieben, ist das Basis-Modell ein UML-Klassendiagramm, bestehend aus dem Hauptdienst, Sub-Diensten und Entitäten, die in bestimmten (evtl. mehreren) Rollen bzgl. der Dienste auftreten.

Der Basis-Modell-Workflow besteht aus zwei aufeinanderfolgenden Aktivitäten, nämlich dem Identifizieren der Rollen (role identification) und dem Benennen der Dienste (service naming).

Zunächst wird das Identifizieren der Rollen besprochen.

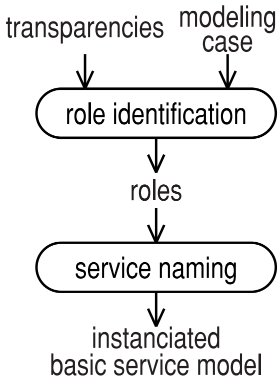

Abbildung 2.7: Basis-Modell-Workflow (basic model's workflow)

Identifizieren der Rollen Beim Identifizieren der Rollen werden alle Rollen identifiziert, die am zu modellierenden Dienst beteiligt sind.

Wichtig ist hierbei die Sichtbarkeit (transparency) der einzelnen Rollen — vor allem der Rollen bezüglich eines Sub-Dienstes — aus der Sicht der Dienstnehmerseite.

Tritt eine Entität in einer Rolle bzgl. eines Dienstes auf, so ist dieser Dienst für diese nicht-transparent, ansonsten transparent.

Eine Entität kann in einer Rolle bezüglich eines Dienstes auf der Dienstnehmer- oder der Dienstleisterseite auftreten. Es muss hierbei noch unterschieden werden, ob eine Entität, die bzgl. eines Dienstes in einer Rolle auf der Dienstleister- oder der Dienstnehmerseite auftritt, für Entitäten, die in einer Rolle bzgl. des gleichen Dienstes auf der anderen Seite auftreten, transparent ist oder nicht.

Die Kenntnis und Sichtbarkeit der zu modellierenden Rollen und Dienste ist auch vom Modellierungsfall (modeling case) abhängig: Beim Reverse Engineering sind i.a. alle Rollen bekannt, da von einer vorhandenen Realisierung des Dienstes ausgegangen wird. In den anderen Fällen ist dies z.T. nicht möglich. Sub-Dienste und zugehörige Rollen, die erst später in einer Realisierung sichtbar werden, können auch erst zu diesem Zeitpunkt erkannt werden.

Die Aktivität „Identifizieren der Rollen“ hat also als Eingabe-Artefakte den Modellierungsfall und die Sichtbarkeit (transparency) der Rollen. Ausgabe-Artefakt sind die identifizierten Rollen.

Die zweite Aktivität des Basis-Modell-Workflows ist die Benennung der Dienste.

Benennung der Dienste Bei dem Benennen der Dienste (service naming) werden der Hauptdienst und alle identifizierten Sub-Dienste benannt und Beziehungen zu den vorher identifizierten Rollen hergestellt und entsprechend bezeichnet.

Damit ist das instanziierte Basis-Modell — zumindest in seiner ersten Version, die ggf. später erweitert wird — soweit vollständig spezifiziert und der Basis-Modell-Workflow abgeschlossen.

Nun folgen die Workflows zum Erstellen der Dienst- und der Realisierungssicht. Reihenfolge und Aufbau der Workflows sind abhängig davon, ob top-down oder bottom-up vorgegangen wird.

2.2.3 Top-Down-Vorgehen

Beim Top-Down-Vorgehen wird zunächst im Dienstsicht Top-Down-Workflow die Dienstsicht, also die für Dienstleister und Dienstnehmer gemeinsame Sicht des Dienstes, sozusagen ein Entwurf des Dienstes, erstellt. Diese Vorgehensweise wird z.B. bei Erstellung einer Ausschreibung verwendet.

Evtl. wird sogar nach Erstellung der Dienstsicht aufgehört, da man die Implementierung zu einem anderen Provider auslagern möchte. Beispiele hierfür wären ausgelagerte Sub-Dienste, die man bei der rekursiven Anwendung der Methodik selbst modellieren könnte.

Hat man die Dienstsicht erstellt, so kann sie zu einer Realisierungssicht verfeinert werden. Dies geschieht im Realisierungssicht Top-Down-Workflow.

Im Folgenden werden die Top-Down-Workflows im Einzelnen behandelt.

2.2.4 Dienstsicht Top-Down-Workflow

Der in Abb. 2.8 dargestellte Dienstsicht Top-Down-Workflow (service view top-down workflow) gliedert sich in vier aufeinanderfolgende Aktivitäten: die Definition der Funktionalität, die QoS-Definition, die Definition von Clients und Dienstzugangspunkten sowie die Festlegung des Service Agreements.

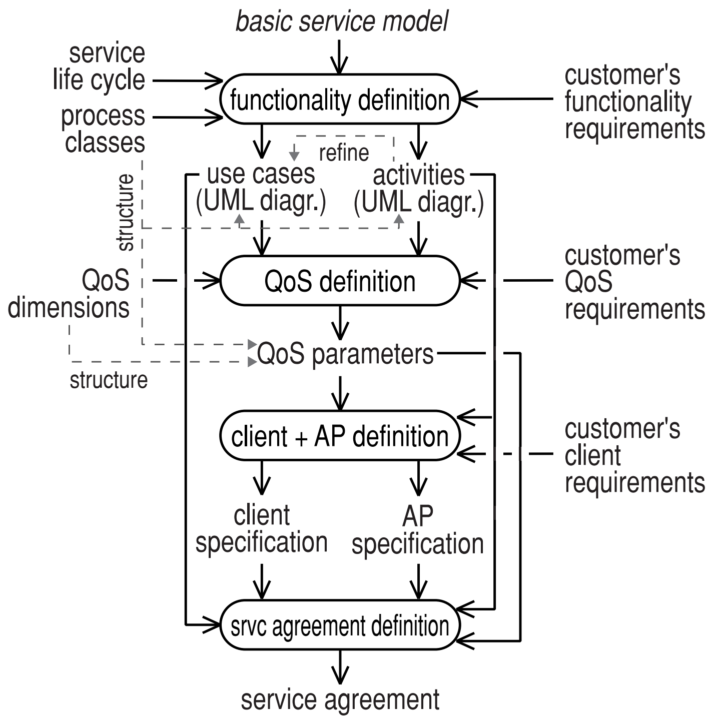

Abbildung 2.8: Dienstsicht Top-Down-Workflow (service view's top-down workflow)

Festlegung der Funktionalität Bei der Definition der Funktionalität (functionality definition) (inkl. der Managementfunktionalität) des Dienstes werden als Eingabe das im Basis-Modell-Workflow erstellte Basis-Modell, die Phasen des Service-Lebenszyklus und eine Prozessklassifizierung sowie die Anforderungen des Kunden an die Dienstfunktionalität verwendet. Die Lebenszyklus-Phasen und die Prozessklassen dienen dazu, alle zur Dienstfunktionalität gehörenden Prozesse zu identifizieren. Ausgabe-Artefakte sind Use-Case- und Aktivitätsdiagramme, jeweils strukturiert nach Prozessklassen und Lebenszyklus-Phasen, die die Funktionalität beschreiben. Die Use-Case-Diagramme zeigen Abhängigkeiten zwischen den identifizierten Prozessen und welche Entitäten (in welchen Rollen, evtl. Verfeinerung der 3 groben Rollen Nutzer, Kunde und Provider) an welchen Prozessen beteiligt sind. Dabei stellen die Entitäten mit ihren Rollen die Akteure und die Prozesse die Use-Cases der Use-Case-Diagramme dar. Für jeden Prozess, d.h. jeden Use-Case, wird durch eines der Aktivitätsdiagramme die Struktur des Prozesses aus mehreren Aktivitäten näher beschrieben.

QoS-Definition Als nächste Aktivität folgt die Definition der Dienstgüte(QoS definition). Eingabe-Artefakte hierfür sind die schon vorher verwendete Prozessklassifizierung, eine Menge von QoS-Dimensionen, die Anforderungen des Kunden an den QoS, sowie die vorher erstellten Use-Case- und Aktivitätsdiagramme. Ausgabe-Artefakt ist eine Liste von QoS-Parametern, die durch die Prozessklassen und die QoS-Dimensionen strukturiert wird.

Vor allem Anfang und Ende eines durch ein Aktivitätsdiagramm modellierten Prozesses können Hinweise über QoS-Parameter liefern. Genaueres hierzu ist in [Schm 01] zu finden.

Clients und Dienstzugangspunkte Nächste Aktivität des Dienstsicht Top-Down-Workflows ist die Festlegung der Clients und der Dienstzugangspunkte (client and access point definition), sowohl für die Dienstnutzung selbst, als auch für das Management des Dienstes.

Eingabe-Artefakte sind hierbei die Aktivitätsdiagramme aus der Funktionalitätsdefinition, die QoS-Parameter sowie die Anforderungen des Kunden an die Clients. Ausgabe-Artefakt ist eine Spezifikation der Clients und der Dienstzugangspunkte. Diese Beschreibung sind verbunden mit einem oder mehreren der Prozesse als textuelle Beschreibung, formale Spezifikation oder eine Referenz zu einem Standard.

Service Agreement Die letzte Aktivität ist die Festlegung der Dienstvereinbarung (service agreement definition) Diese besteht im wesentlichen aus den Use-Case- und Aktivitätsdiagrammen der Funktionalitätsdefinition, den QoS-Parametern und den Client/Dienstzugangspunkt-Festlegungen.

Ergänzt wird sie durch konkrete Werte-Festlegungen vor allem für QoS-Parameter, gesetzlich-vertragliche Vereinbarungen und sonstige zur genaueren Instanziierung des tatsächlichen Dienstes notwendigen Informationen.

Nach dieser Aktivität ist die gesamte Dienstsicht beim Top-Down-Vorgehen spezifiziert. Nun folgt der Realisierungssicht Top-Down-Workflow, in dem die Realisierungssicht, also eine interne Sicht auf den Dienst für die Dienstleisterseite selbst, durch Verfeinerung der Dienstsicht erstellt wird.

2.2.5 Realisierungssicht Top-Down-Workflow

Der Realisierungssicht Top-Down-Workflow (realization view top-down workflow) (Abb. 2.9) gliedert sich in diese aufeinanderfolgenden Aktivitäten: Festlegung des Provider-Service-Managements-Prozesses, Outsourcing-Festlegungen und Ressourcenidentifikation.

Provider-interner Service-Management-Prozess Erste Aktivität ist die Definition des provider-internen Service-Management-Prozesses (definition of provider's service management process). Die Abkürzung SM-Prozess steht im Folgenden für Service-Management-Prozess.

Hier werden die Use-Case- und Aktivitätsdiagramme aus der Funktionalitätsdefinition des Dienstsicht Top-Down-Workflows verfeinert, um für den Provider die Funktionalität zur Verwaltung der internen Ressourcen und Prozesse zu beschreiben. Zu den in der Funktionalitätsdefinition des Dient-Sicht-Workflows identifizierten Prozessen können weitere hinzugenommen werden und außerdem die vorhandenen Prozesse durch weitere Aktivitäten, die provider-intern sind, erweitert werden. Die entstehenden Use-Case- und Aktivitätsdiagramme beschreiben dann die gesamte Dienstlogik. Zur weiteren Strukturierung der Prozesse wird hier eine erweiterte Klassifikation von Prozessen nach TOM (Telecom Operations Map) verwendet.

Outsourcing-Entscheidungen Als nächstes werden Outsourcing-Entscheidungen (outsourcing decision) getroffen. Hierbei wird anhand der gegebenen Aktivitätsdiagramme entschieden, welche der zuvor festgelegten Prozesse ausgelagert werden.

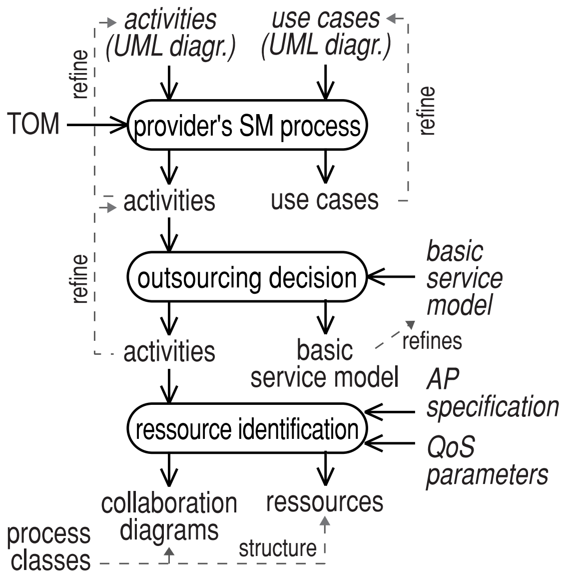

Abbildung 2.9: Realisierungssicht Top-Down-Workflow (realization view's top-down workflow)

Das Basis-Modell wird dabei pro ausgelagertem Prozess um einen neuen Sub-Dienst erweitert. Hierbei müssen wieder, wie beim Basis-Modell-Workflow, alle damit verbundenen Transparenzen der Rollen betrachtet und verarbeitet werden. Ist ein Sub-Dienst von der Dienstnehmerseite des Hauptdienstes aus transparent, so wird er als intern markiert. Dies muss das Basis-Modell als Stereotyp unterstützen.

Außerdem wird für jeden ausgelagerten Dienst ein Sub-Dienst-Client bzw. ein Sub-Dienstmanagement-Client festgelegt. Zudem werden die Aktivitätsdiagramme für die ausgelagerten Prozesse entsprechend erweitert, z.B. um den Zugriff auf den ausgelagerten Sub-Dienst zu beschreiben.

Insbesondere diese Informationen können bei rekursiver Anwendung der Methodik für den Sub-Dienst als Teil der Kundenanforderungen benutzt werden.

Identifikation von Ressourcen und BMF Die letzte, abschließende Aktivität ist die Identifikation von provider-internen Ressourcen bzw. Basic Management Functionality (BMF) (resource identification), die für den Dienst benötigt werden.

Eingabe-Artefakte sind hier die erweiterten Aktivitätsdiagramme aus der vorigen Aktivität, die Client/Dienstzugangspunkt-Spezifikation und die QoS-Parameter aus dem Dienstsicht-Workflow sowie die Prozessklassifizierung.

Anhand der Aktivitätsdiagramme werden zunächst die internen Ressourcen bzw. die Basic Management Functionality, strukturiert nach Prozessklassen identifiziert.

Ebenso werden in den Aktivitätsdiagrammen die Zugangspunkte für die Ressourcen/Basic Management Functionality und Sub-Dienst/Sub-Dienstmanagement-Clients zu den Prozessen gefunden.

Diese Zugangspunkte werden dann den internen Ressourcen/Basic Management Functionality (BMF) bzw. den Sub-(Management)-Clients zugeordnet.

Um die Interaktionen zwischen den internen bzw. externen Ressourcen/BMF schließlich genauer zu beschreiben, werden Kollaborationsdiagramme, strukturiert nach Prozessklassen verwendet.

Auch werden den QoS-Parameter aus der Dienstsicht mess- bzw. beobachtbare Werte zugewiesen.

Abschließend wird die Dienstarchitektur, die die Implementierung des Dienstes ermöglicht, angegeben.

Damit ist die Realisierungssicht, also das ganze Dienstmodell, beim Top-Down-Vorgehen vollständig.

2.2.6 Bottom-Up-Vorgehen

Das Bottom-Up-Vorgehen wird vor allem dann verwendet, wenn der zu modellierende Dienst oder Teile davon schon bestehen, d.h. wenn Information über die Realisierung bzw. Implementierung bereits vorhanden ist.

Bei diesem Vorgehen wird zunächst im Realisierungssicht Bottom-Up-Workflow die Realisierungssicht des Dienstes erstellt. Danach wird im Dienstsicht Bottom-Up-Workflow aus der Realisierungssicht durch Abstraktion von für den Kunden irrelevanten, provider-internen Gegebenheiten die Dienstsicht gewonnen.

2.2.7 Realisierungssicht Bottom-Up-Workflow

Der Realisierungssicht-Bottom-Up-Workflow (realization view bottom-up workflow) (Abb. 2.10) besteht aus drei Aktivitäten: der Identifikation der Use-Cases, der Identifikation von Ressourcen bzw. Basic Management Functionality und der Dienstlogik-Definition.

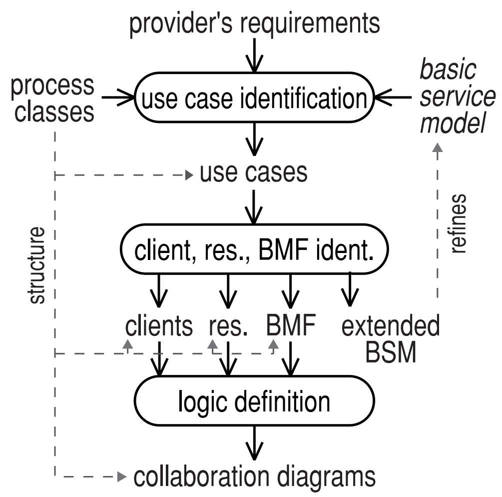

Abbildung 2.10: Realisierungssicht Bottom-Up-Workflow (realization view's bottom-up workflow)

Use-Case-Identifikation Als erstes werden bei der Use-Case-Identifikation (use case identification) aus den Anforderungen der Providers und dem im Basis-Modell-Workflow erstellten Basis-Modell Use-Cases für die grobe Darstellung der Dienstfunktionalität abgeleitet und in Use-Case-Diagrammen aus Entitäten und Prozessen. Der Ablauf dabei ist ähnlich zu der Funktionalitätsdefinition beim Top-Down-Vorgehen, nur werden hier auch provider-interne Prozesse betrachtet. Die Prozesse werden außerdem mit einer Prozessklassifizierung strukturiert.

Sub-Client, Ressourcen, BMF-Identifikation Als nächstes werden aus den zuvor erstellten Use-Cases die für die Dienstrealisierung notwendigen Sub-Clients, Ressourcen und Basic Management Functionality (BMF) gefunden und dabei das Basis-Modell evtl. um weitere Sub-Dienste erweitert (subclient, resource and BMF identification).

Wird ein neuer Sub-Dienst eingeführt, so müssen — wie im Basis-Modell-Workflow bzw. im Top-Down-Vorgehen bei den Outsourcing-Entscheidungen — alle Transparenzen der Rollen bzgl. diesem Sub-Dienst bestimmt und im Basis-Modell entsprechend festgehalten werden.

Die identifizierten Sub-Clients, Ressourcen und Basic Management Functionality werden durch die schon vorher verwendete Prozessklassifizierung strukturiert.

Für die Art ihrer Darstellung gelten die gleichen Aussagen, wie beim Top-Down-Vorgehen für die Aktivitäten Outsourcing-Entscheidungen und Ressourcenidentifikation.

Dienstlogik-Definition Die Realisierungssicht beim Bottom-Up-Vorgehen wird vervollständigt durch die Dienstlogik-Definition (service logic definition).

Aus den vorher gefundenen Sub-Clients, Ressourcen und BMF und der Prozessklassifizierung werden nun pro Use-Case Kollaborationsdiagramme erstellt. Sie legen detailliert die Dienstlogik sowie Dienstmanagement-Logik fest.

Außerdem werden für den Hauptdienst, sowohl für den Dienst selbst als auch sein Management, Dienstzugangspunkte definiert. Diese werden in die Kollaborationsdiagramme mitaufgenommen.

Ähnlich wie bei der Realisierungssicht des Top-Down-Vorgehens muss wohl auch hier zur Vervollständigung noch die den Dienst unterstützende Dienstarchitektur festgelegt werden. Diese kann man in diesem Fall der bereits vorhandenen Realisierung des Dienstes entnehmen.

Nach dieser Aktivität ist die Realisierungssicht vollständig beschrieben.

Es folgt nun der Dienstsicht Bottom-Up-Workflow zur Generierung der Dienstsicht aus der Realisierungssicht.

2.2.8 Dienstsicht Bottom-Up-Workflow

Beim Bottom-Up-Vorgehen wird im Dienstsicht Bottom-Up-Workflow (service view bottom-up workflow) (Abb. 2.11) aus der Realisierungssicht die Dienstsicht abgeleitet. Dieser Workflow umfasst vier aufeinanderfolgende Aktivitäten: die Funktionalitätsdefinition, QoS-Spezifikation, Dienstzugangspunkt-Beschreibung und Service Agreement-Spezifikation.

Definition der Funktionalität Zuerst werden bei der Definition der Funktionalität (functionality definition) aus den Kollaborationsdiagrammen des Realisierungssicht-Workflows Aktivitätsdiagramme abgeleitet, die die Aktivitäten der Implementierungselemente allein aufzeigen. Zu jedem der im Realisierungssicht-Workflow erarbeiteten Use-Case-Diagramme (oder einer Auswahl davon, die für den Service View ausreicht) wird je ein Aktivitätsdiagramm erstellt.

Hierbei sollten alle Aktivitäten und Prozesse, die nur intern den Provider betreffen, entfernt werden, und somit für die Dienstnehmerseite nicht sichtbar sein.

Strukturiert werden die Use-Case- und Aktivitätsdiagramme nach Service-Lebenszyklus-Phasen und der schon im Realisierungssicht-Workflow verwendeten Prozessklassen.

QoS-Definition Bei der nachfolgenden QoS-Spezifikation (QoS specification) werden aus der Basic Management Functionality des Realisierungssicht-Workflows, den Use-Case- und den vorher erstellten Aktivitätsdiagrammen die QoS-Parameter, strukturiert nach Prozessklassen und gegebenen QoS-Dimensionen, identifiziert. Die QoS-Parameter werden dazu textuell oder als Tabelle beschrieben und dem jeweils zugehörigen Use-Case zugeordnet.

Zugangspunkt-Definition Bei der Zugangspunkt-Definition (access point definition) wird aus den Kollaborationsdiagrammen der Realisierungssicht die Beschreibung der Dienstzugangspunkte, sowohl des Dienstes selbst als auch des Managements davon, übernommen.

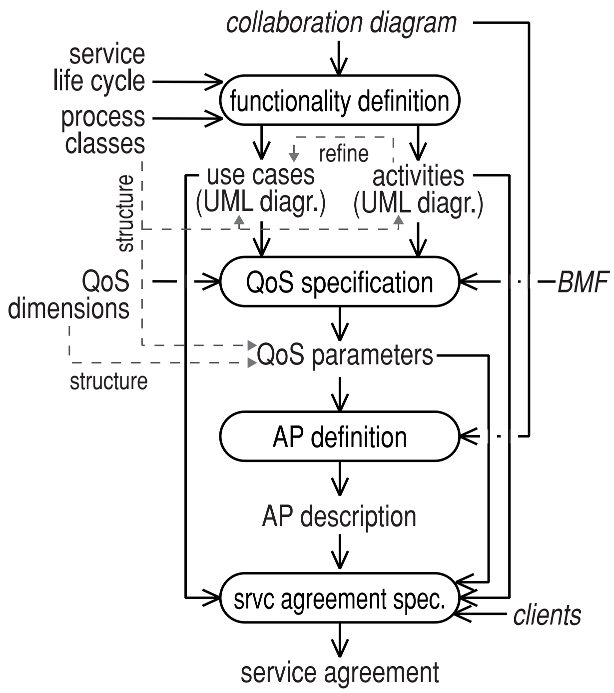

Abbildung 2.11: Dienstsicht Bottom-Up-Workflow (service view's bottom-up workflow)

Service-Agreement Die letzte Aktivität ist Spezifikation einer Dienstvereinbarung (service agreement definition), bestehend aus den Use-Case und Aktivitätsdiagrammen, den QoS-Parametern und der Dienstzugangspunkt-Beschreibung.

Hier müssen auch noch die Clients für Dienst selbst und Management festgelegt werden. Feste Werte und sonstige den Dienst näher konkretisierende Details werden noch hinzugefügt (vergleiche [Schm 01]). Diese könnten z.B. aus der vorhandenen Dienstrealisierung abgelesen werden.

Damit ist die Realisierungssicht und das ganze Dienstmodell beim Bottom-Up-Vorgehen fertiggestellt.

2.3 Freiräume der Anwendungsmethodik

Im Folgenden werden die Freiräume, d.h. nicht die vollständig festgelegten Aspekte, der vorher besprochenen Anwendungsmethodik besprochen.

Vor allem werden in der Methodik eine ganze Reihe von Artefakten aufgeführt, für die nicht exakt festgelegt wird, in welcher Form sie zu spezifizieren sind. Es wird oft nur beispielhaft gesagt welche Möglichkeiten für die Darstellung existieren. Unter diesen Möglichkeiten befindet sich die Darstellung als reiner Text oder als eine Liste. Auch Standards können referenziert werden. Aber selbst in diesen Fällen ist nicht immer klar, wie genau die jeweiligen Artefakte strukturiert sein sollen.

Zusammenfassend, werden die betreffenden Artefakte und ihre jeweiligen Freiräume nun aufgeführt:

- Für die Spezifikation der Funktionalitäts-, QoS- und Client-Anforderungen des Kunden beim Dienstsicht Top-Down-Workflow oder die Provider-Anforderungen beim Realisierungssicht Bottom-Up-Workflow wird keine genaue Form angegeben.
- Für die QoS-Parameter wird nicht die genaue Struktur angegeben, sondern nur gesagt, dass sie nach QoS-Dimensionen eingeteilt werden, als Text oder Tabelle zu spezifizieren sind und jeweils einem Use-Case zugeordnet sind.

- Über die Form der Clients und Zugangspunkt-Spezifikationen wird nur gesagt, dass es bloßer Text oder eine formale Spezifikation sein kann in Verbindung mit Referenzen zu Standards. Dabei kann die Beschreibung der Schnittstellen mit einem oder mehreren der identifizierten Prozesse verbunden sein. Wie genau aber z.B. eine solche formale Spezifikation aussieht, wird nicht festgelegt.
- Die genaue Form der Dienstvereinbarung, d.h. die Form der zusätzlichen vertraglichen Regelungen bzw. der Informationen, die den Dienst genauer beschreiben, wird auch nicht genannt.
- Die genaue Beschreibungsform von Ressourcen bzw. BMF's in der Realisierungssicht wird nicht genau erklärt. Ressourcen bzw BMF werden zwar nach Prozessklassen strukturiert und mit den Kollaborationsdiagrammen in Verbindung gebracht, aber welche Struktur sie selbst haben sollen, wird nicht genannt.
- Die genaue Form der Angabe der Dienstarchitektur bei der Ressourcenidentifikation beim Realisierungssicht Top-Down-Workflow wird auch nicht erwähnt.

Eine Werkzeugunterstützung für die Methodik, die möglichst viel automatisieren soll, verlangt nach möglichst strukturierten Artefakten. Da die Methodik in einigen Fällen Freiräume lässt, muss bei der Entwicklung der Werkzeugunterstützung darüber nachgedacht werden, wie genau die verschiedenen Artefakte spezifiziert und strukturiert werden können.

2.4 Zusammenfassung

In diesem Kapitel wurde das MNM-Dienstmodell erläutert. Zunächst wurde der Aufbau des Basis-Modells, der Dienstsicht und der Realisierungssicht erklärt.

Anschließend wurde die Methodik zur Anwendung des Modells auf einen gegebenen Dienst betrachtet. Die Methodik beschreibt Workflows, die selbst aus Aktivitäten mit Ein/Ausgabe-Artefakten besteht. Dabei gibt es prinzipiell zwei grundlegende Vorgehensweisen: Top-Down oder Bottom-Up, die sich in Reihenfolge und Aufbau der Workflows unterscheiden.

Als letztes wurden die nicht genauer von der Methodik festgelegten Aspekte besprochen: Vor allem bei der Struktur und der Darstellung einiger Artefakte lässt die Methodik Spielräume, für die beim Entwurf des Werkzeugs genauere Festlegungen getroffen werden müssen.

Kapitel 3

Analyse der Methodik und der Benutzerinteraktionen

In diesem Kapitel wird die Anwendungsmethodik des MNM-Dienstmodells detailliert analysiert. Dies dient einerseits zur Entwicklung einer groben Architektur für das zu entwickelnde Werkzeug, andererseits um Anforderungen an das zu entwickelnde Werkzeug bzw. an das bestehende, zu erweiternde OO/CASE-Werkzeug zu identifizieren.

In Abschnitt 3.1 wird zunächst ein Überblick gegeben. Abschnitt 3.2 geht dann auf die Analyse der Anwendungsmethodik allgemein ein. Die für die Realisierung der Anwendungsmethodik notwendigen Benutzerinteraktionen dagegen werden in Abschnitt 3.3 betrachtet. Dann wird in Abschnitt 3.4 ein erster Entwurf der Architektur für das Werkzeug vorgestellt. Daraus wird abschließend ein Katalog von Anforderungen für die Wahl des OO/CASE-Werkzeuges, das als Grundlage für das zu entwickelnde Werkzeug dient, abgeleitet.

3.1 Einleitung

Das zu entwickelnde Werkzeug muss einerseits die Anwendungsmethodik vollständig enthalten und andererseits alle dafür notwendigen Interaktionen mit dem Benutzer durchführen.

Hierbei gilt als allgemeines Ziel, die Interaktionen mit dem Benutzer zu minimieren, d.h. möglichst vieles zu automatisieren. Eine hohe Automatisierung verlangt jedoch, dass die Daten — in diesem Fall die Artefakte der Anwendungsmethodik — ausreichend strukturiert sind.

Die Anforderungen an das zu entwickelnde Werkzeug bzw. das dafür als Grundlage dienende OO/CASE-Werkzeug lassen sich somit grob in drei Klassen teilen:

- Anforderungen, die sich aus der Anwendungsmethodik selbst ergeben
- Anforderungen, die sich aus den Interaktionen zwischen Werkzeug und Benutzer identifizieren lassen
- Sonstige Anforderungen, die sich zusätzlich aus der Aufgabenstellung ergeben

Die Anforderungen an das Werkzeug werden daher im wesentlichen aus der Analyse der Anwendungsmethodik und den sich daraus ergebenden Interaktionen mit dem Modellierungsanwender abgeleitet (siehe Abb. 3.1).

Im Weiteren werden die Anforderungsklassen einzeln genauer betrachtet. In Abschnitt 3.2 werden die Methodik allgemein und in Abschnitt 3.3 die Benutzerinteraktionen der Methodik analysiert. Daraus wird

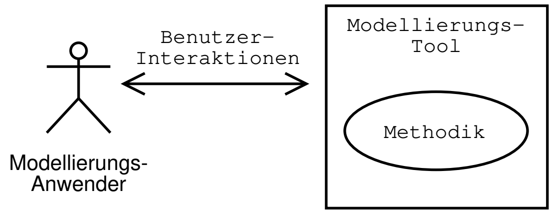

Abbildung 3.1: Skizze der Analyse

dann in Abschnitt 3.4 eine erste, grobe Architektur für das Werkzeug entworfen und daraus ein Anforderungskatalog für die verwendete Software erstellt. Hierbei werden dann auch die sonstigen Anforderungen, die sich zusätzlich aus der Aufgabenstellung ergeben, berücksichtigt.

3.2 Analyse der Anwendungsmethodik

Das Werkzeug soll die Methodik zur Erstellung einer Instanz des MNM-Dienstmodells zu einem gegebenen Dienst unterstützen.

Da die Methodik Workflows mit vielen verschiedenen Ein- und Ausgabe-Artefakten spezifiziert, muss das Werkzeug grundlegend eine Workflowsteuerung für die Einrichtung, Verwaltung und Abarbeitung der verschiedenen Workflows der Methodik beinhalten sowie die Erstellung und Verwaltung der verschiedenen Artefakte ermöglichen.

Die Artefakte lassen sich in zwei Gruppen teilen: Zum einen gibt es strukturierende Artefakte, wie z.B. die Phasen des Dienstlebenszyklus, die i.A. vom Dienst unabhängig sind. Alle sonstigen Artefakte sind dienstspezifisch und müssen pro modelliertem Dienst eigens verwaltet werden.

Zu den dienstspezifischen Artefakten zählen eine Reihe von UML-Diagrammtypen, genauer Klassendiagramme (vor allem für das Basis-Modell), Use-Case-, Aktivitäts- und Kollaborationsdiagramme. Das Werkzeug muss daher eine Unterstützung für die Erstellung, Veränderung, Verwaltung und ggf. teilweise Auswertung dieser Diagrammtypen erlauben.

Für die übrigen dienstspezifischen Artefakte muss zunächst genau festgelegt werden, wie sie aufgebaut sind bzw. welche evtl. verschiedenen Möglichkeiten für ihren Aufbau existieren. Solche Artefakte können z.B. aufgebaut sein als Listen, als Rekords, als freier Text oder Kombinationen davon. Für jedes Artefakt muss dann je nach seinem spezifizierten Aufbau die Erzeugung und Verwaltung bzw. Zugriff auf einzelne Bestandteile (bei Listen, Rekords, etc.) vom Werkzeug unterstützt werden.

Unter diesen Artefakten gibt es einige, die selbst als UML-Elemente in UML-Diagrammen auftauchen, so kommen z.B. die Prozesse als Use-Cases in den Use-Case-Diagrammen oder aber die Ressourcen und Sub-Clients in den Kollaborationsdiagrammen vor. Bei der Spezifikation des Aufbaus dieser Artefakte muss dieser Sachverhalt berücksichtigt. Das Werkzeug sollte die Korrespondenz zwischen diesen Artefakten selbst und ihrer Darstellung in den UML-Diagrammen entsprechend speichern bzw. verarbeiten.

Bei einigen Artefakten können Referenzen zu Standards gemacht werden. Hierfür muss das Werkzeug zusätzlich Unterstützung bieten. (Dokumentenmanagement)

Im Folgenden wird eine Zusammenfassung aller genannten Anforderungen gegeben:

- Workflowsteuerung: die für jeden zu modellierten Dienst, die passenden Workflows anlegt, verwaltet und durchführt.
- Artefakt-Verwaltung: dabei wird unterschieden zwischen dienstunabhängigen Artefakten zur Strukturierung und dienstabhängigen Artefakten, die für jeden modellierten Dienst eigens angelegt und verwaltet werden müssen.

aus Sicht der realisierten Aktivität:
Eingabe-Interaktion
Verarbeitungs-Interaktion
nach Automatisierungsgrad:
nicht automatisierte Interaktion
halb automatisierte Interaktion
voll automatisierte Interaktion
nach den vorkommenden bzw. verwendeten Artefakttypen:
Basiseinstellungs-Interaktion
Strukturierungs-Definitions-Interaktion
UML-Interaktion
nicht-UML-Interaktion:
einfache nicht-UML-Interaktion
nicht-UML-Interaktion mit Strukturierung
nicht-UML-Interaktion ohne Strukturierung mit UML
nicht-UML-Interaktion mit Strukturierung und UML

Tabelle 3.1: Klassifizierung der Benutzerinteraktionen

- Bei den dienstabhängigen Artefakten muss vor allem die Erstellung, Änderung, Verwaltung und ggf. Auswertung von UML-Klassen, Use-Case, Aktivitäts- und Kollaborationsdiagrammen unterstützt werden.
- Für die sonstigen dienstabhängigen Artefakten muss zunächst deren genaue Struktur spezifiziert werden. Dann muss die Erzeugung, die Verwaltung und der Zugriff auf Bestandteile dieser Artefakte bezüglich der spezifizierten Struktur unterstützt werden. Insbesondere Berücksichtigung von Beziehungen zwischen diesen Artefakten und bestimmten Elementen in den UML-Diagrammen muss gesondert berücksichtigt werden.
- Auch die Referenzierung von Standards muss unterstützt werden.

3.3 Analyse der Benutzerinteraktionen

3.3.1 Einführung

Das Werkzeug muss dem Modellierungsanwender erlauben, zu einem gegebenen Anwendungsfall eine Instanz des MNM-Dienstmodells zu erstellen. Dafür müssen die Workflows der Anwendungsmethodik realisiert werden.

Zur Abarbeitung eines Workflows muss das Werkzeug eine Reihe aufeinanderfolgender Aktivitäten realisieren.

Da jede Aktivität Artefakte als Eingabe und Artefakte als Ausgabe hat, muss das Werkzeug die Eingabe neuer Artefakte durch den Benutzer bzw. die automatische Übernahme von früher erstellten Artefakten unterstützen. Für die Eingabe neuer Artefakte bzw. die Auswahl/Bestätigung beim Übernehmen schon bestehender Artefakte, sind Interaktionen zwischen dem Werkzeug und dem Benutzer durchzuführen.

Das Werkzeug muss als wesentliche Anforderung alle Interaktionen geeignet unterstützen.

Diese Interaktionen ergeben sich aus der Betrachtung der Anwendungsmethodik, d.h. vor allem durch Betrachtung der Ein- und Ausgabe-Artefakte der Aktivitäten.

Die verschiedenen Workflows werden daher betrachtet und aus ihren Aktivitäten und Artefakten Benutzerinteraktionen abgeleitet. Für die Interaktionen wurde dann zur Übersichtlichkeit eine Klassifizierung vorgenommen aus der dann die Anforderungen abgeleitet werden.

Grundlegend muss das Werkzeug dem Benutzer erlauben für jeden neu zu modellierenden Dienst eigene Instanzen der drei nacheinander auszuführenden Workflows anzulegen.

Da das MNM-Dienstmodell rekursiv auf Sub-Dienste eines modellierten (Haupt-)Dienstes anwendbar ist, wäre es wünschenswert, wenn das Werkzeug dem Benutzer eine solche rekursive Modellierung erleichtert. Das Werkzeug könnte automatisch entsprechende Artefakte aus der Modellierung des (ursprünglichen) Hauptdienstes übernehmen und neue Workflows für den (bisherigen) Sub-Dienst einrichten. Dies könnte evtl. parallel zu den noch laufenden Workflows des Hauptdienstes sein, soweit dies die Modellierung zulässt. Das Werkzeug muss dafür Übernahme-Möglichkeiten von Artefakte zwischen Haupt- und Sub-Diensten ermöglichen.

Was die Eingabe-Artefakte einer Aktivitäten betrifft, so sollten Artefakte, die in frühren Aktivitäten erstellt wurden, automatisch übernommen werden. Neue Artefakte werden entweder ganz manuell vom Benutzer eingegeben oder aber das Werkzeug bietet dem Benutzer Vorgaben, die der Benutzer annehmen oder ändern bzw. erweitern kann.

Aus Sicht der Realisierung einer Aktivität, kann man zwei Typen von Interaktionen unterscheiden:

- **Eingabe-Interaktionen**, die neue Eingabe-Artefakte erstellen, sei es manuell durch den Benutzer oder in Zusammenarbeit von Werkzeug und Benutzer.
- **Verarbeitungs-Interaktionen**, die gegebene Eingabe-Artefakte verarbeiten, d.h. Zwischenergebnisse oder Ausgabe-Artefakte bzw. Teile davon erzeugen.

Vor allem aber lassen sich die Benutzerinteraktionen nach dem Grad der Automatisierung bezüglich der Erstellung neuer Eingabe-Artefakte (Artefakte, die noch in einer Aktivität vorher erstellt wurden) bzw. der Verarbeitung der Eingabe-Artefakte, d.h. der Erstellung der Ausgabe-Artefakte in die folgenden Klassen teilen:

- **Voll-Automatisierte Interaktionen:** der Benutzer muss höchstens noch die Durchführung bestätigen, die Ausführung wird vom Werkzeug voll übernommen.
- **Halb-Automatisierte Interaktionen:** das Werkzeug stellt dem Benutzer Vorgaben bzw. mögliche Alternativen als Eingabe, die der Benutzer akzeptieren oder aber verändern oder erweitern kann.
- **Nicht-Automatisierte Interaktionen:** der Benutzer muss alle Eingaben selbst treffen, das Werkzeug erlaubt ihm dabei die Eingabe der Daten bzw. Artefakte, gibt jedoch keinerlei mögliche Vorgaben.

Wie bereits in Abschnitt 2.3 erwähnt lassen sich in der Anwendungsmethodik drei verschiedene Sorten von Artefakten unterscheiden: Strukturierende Artefakte, UML-Diagramme und sonstige Artefakte, die wie die UML-Diagramme dienstabhängig sind. Diese sonstigen, dienstspezifischen Artefakte werden im folgenden kurz nicht-UML-Artefakte genannt.

Die Typen der Artefakte bieten eine weitere Klassifizierung der Interaktionen. Man kann damit die Interaktionen nach den Typen der Artefakte, die bei einer Interaktion erstellt bzw. verändert werden, sowie nach den Typen der Artefakte, die zur Erstellung bzw. Veränderung verwendet werden, unterscheiden:

- **Strukturierungs-Definitions-Interaktionen (kurz Strukturierungs-Definitionen):** Interaktionen, in denen die zur Strukturierung verwendeten Artefakte definiert bzw. erweitert werden. Da sie weitgehend dienstunabhängig sind, kann das Werkzeug diese Artefakte bereits vorgeben. Der Benutzer sollte jedoch bei Bedarf eine Änderung oder Verfeinerung durchführen können. Das heißt diese Interaktionen sind halb-automatisch.
- **UML-Diagramm-Interaktionen (kurz UML-Interaktionen):** Interaktionen, in denen UML-Diagramme erstellt bzw. erweitert werden. In den UML-Diagrammen treten auch Elemente auf, die Artefakte repräsentieren, die selbst zu den nicht-UML-Artefakten zählen. Beispiele hierfür sind z.B. die Dienstzugangspunkte oder Ressourcen in Kollaborationsdiagrammen. Bei alle vorkommenden UML-Diagramme werden außerdem strukturierende Artefakte zur Einteilung verwendet.
- **Nicht-UML-Artefakt-Interaktionen (kurz nUML-Interaktionen):** Interaktionen, in denen nicht-UML-Artefakte erstellt bzw. erweitert werden. Derartige Artefakte haben z.B. die Form von Listen,

Rekords, reinem Text oder Kombinationen davon. Diese Interaktionen lassen sich weiter danach einteilen, ob dabei strukturierende Artefakte zur Einteilung verwendet werden und ob für die Erzeugung bzw. Änderung der Artefakte auf UML-Diagramme zugegriffen wird:

  - **Einfache nUML-Interaktionen:** Interaktionen, bei denen nicht-UML-Artefakte ohne Strukturierung und ohne Zugriff auf UML-Diagramme erstellt oder verändert werden.
  - **nUML-Interaktionen mit Strukturierung:** Interaktionen, bei denen nicht-UML-Artefakte mit Strukturierung, aber ohne Zugriff auf UML-Diagramme erstellt oder verändert werden.
  - **nUML-Interaktionen ohne Strukturierung mit UML(-Zugriff):** Interaktionen, bei denen nicht-UML-Artefakte ohne Strukturierung und mit Zugriff auf UML-Diagramme erstellt oder verändert werden.
  - **nUML-Interaktionen mit Strukturierung und UML(-Zugriff):** Interaktionen, bei denen nicht-UML-Artefakte mit Strukturierung und mit Zugriff auf UML-Diagramme erstellt oder verändert werden.

- **Basiseinstellungs-Interaktionen (kurz Basiseinstellungen):** Interaktionen, in denen Basiseinstellungen für die Modellierung eines Dienstes festgelegt werden, die in der Methodik selbst nicht als Artefakte vorkommen. Derartige Interaktionen finden hauptsächlich zu Beginn der Modellierung statt. Ein Beispiel für ein solche Einstellung ist die Entscheidung, ob bei der Modellierung Top-Down oder Bottom-Up vorgegangen wird.

Die Tabelle 3.1 zeigt eine Übersicht über alle genannten Klassifizierungsmöglichkeiten für Interaktionen.

Im Weiteren werden zunächst die aus der Methodik abgeleiteten Interaktionen pro Workflow und Aktivität im Einzelnen behandelt und nach den drei oben genannten Kriterien klassifiziert. Die Klassifikation wird dabei jeweils in der Auflistung der Interaktionen einer Aktivität vor Beschreibung der Interaktion genannt. Die Klassifikation wird hierbei in der Reihenfolge Automatisierungsgrad, Interaktionstyp nach Artefakttypen, Eingabe/Verarbeitung dargestellt.

Danach wird eine zusammenfassende Übersicht der Interaktionen nach den Klassen erstellt, um zusammenfassend darzustellen, welche Benutzerinteraktionen das Werkzeug realisieren sollte.

Abb. 3.2 gibt zunächst einen Überblick über die Benutzerinteraktionen, der sich aus der Abfolge und dem grobem Aufbau der Workflows aus Aktivitäten ergibt.

3.3.2 Interaktionen für Basiseinstellungen

Zu Beginn der Modellierung, bevor die eigentlichen Workflows der Methodik durchgeführt werden, müssen einige grundlegende Einstellungen, hier als Basiseinstellungen bezeichnete Interaktionen mit dem Benutzer durchgeführt werden:

Vor allem muss hier festgelegt werden, ob bei der Modellierung Top-Down oder Bottom-Up vorgegangen wird. Diese Eingabe könnte mit der Angabe des Modellierungsfalles (Reverse Engineering, Ausschreibungs-Erstellung, Angebots-Erstellung) verbunden wederden, denn beim Reverse-Engineering und der Ausschreibungserstellung liegt die Entscheidung Top-Down/Bottom-Up bereits fest. Das auch für den Basis-Modell-Workflow benötigte Artefakt des Modellierungsfalles könnte automatisch von hier übernommen werden.

Hier könnte auch bei der Rekursion der Modellierung, wenn also ein Sub-Dienst eines (Haupt-)Dienstes nun selbst als Hauptdienst modelliert wird, diese Abhängigkeit zwischen den beiden Modellen angegeben werden. Dadurch könnte dem Benutzer durch automatische Übernahme von Teilen des Basis-Modells des ursprünglichen Hauptdienstes Arbeit abgenommen werden. Auch später könnte diese Information verwendet werden, um Artefakte aus der Modellierung des Hauptdienstes in die Modellierung des Sub-Dienstes zu übernehmen.

Damit ergeben sich folgende Interaktionen (siehe Abb. 3.3):

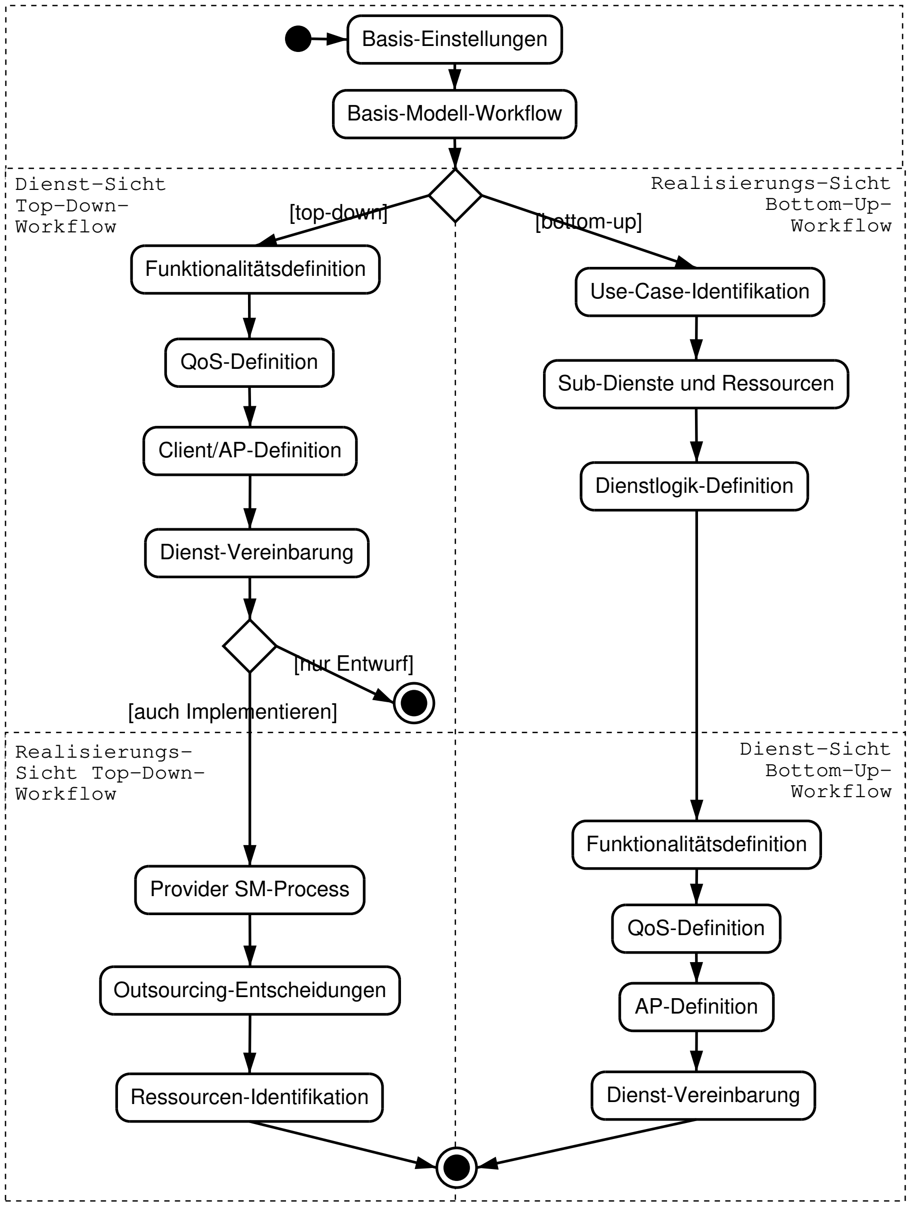

Abbildung 3.2: Überblick über die Benutzerinteraktionen

nicht automatisiert/Basiseinstellung/Eingabe:

- Festlegung des Modellierungsfalles bzw. der Vorgehensweise (Top-Down oder Bottom-Up)
- Evtl Festlegung des Rückgriff auf Workflows früherer Modellierungen bei Rekursion der Modellierung (z.B. bestehende Basis-Modelle übernehmen oder erweitern)
- Evtl. sonstige allgemeine Festlegungen

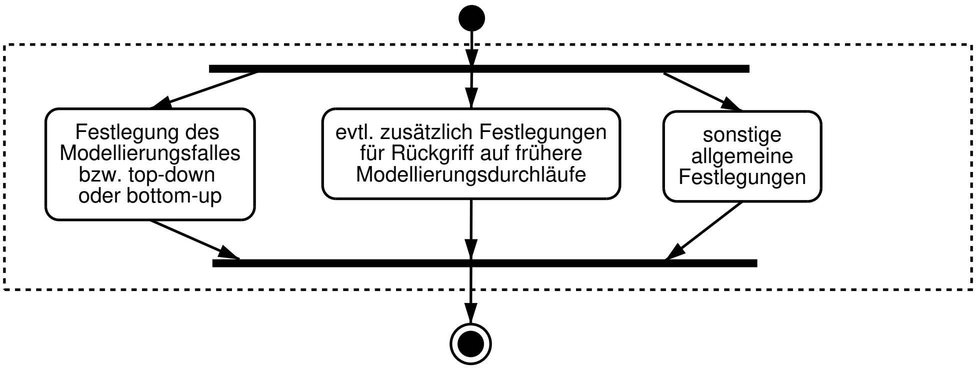

Abbildung 3.3: Benutzerinteraktionen zu Beginn der Modellierung

3.3.3 Interaktionen beim Basis-Modell-Workflow

Beim Basis-Modell-Workflow wird das Basis-Modell — zumindest in einer ersten Version erstellt.

Das heißt das Werkzeug muss dem Benutzer hier ermöglichen, das Basis-Modell als Klassendiagramm aus Diensten (ein Hauptdienst und evtl. Sub-Dienste), Entitäten und zugewiesenen Rollen (transparent oder nicht-transparent) bezüglich der Dienste zu erstellen.

Eine Unterstützung beider Darstellungsformen für Rollen, d.h. die abstrakte und die instanziierte, wäre möglich.

Wünschenswert wäre, wenn dem Benutzer durch automatische Generierung von Teilen dieses Klassendiagramms aus den gegebenen Transparenzen und dem Anwendungsfall Arbeit abgenommen werden könnte.

Vor Beendigung des Basis-Modell-Workflows muss vom Werkzeug überprüft werden, ob alle Dienste benannt sind und ob zu allen Diensten benannte Rollen zugewiesen sind.

Außerdem muss das Werkzeug ermöglichen, das hier erstellte Basis-Modell in späteren Workflows ggf. zu erweitern.

Als Interaktionen des Werkzeugs mit dem Modellierer ergeben sich (siehe Abb. 3.4):

- *nicht automatisiert/einfache nUML-Interaktion/Eingabe:*
  Festlegen der Haupt- und Sub-Dienste (soweit bekannt), Entitäten und Transparenzen zwischen diesen
- *nicht oder halb-automatisiert/UML-Interaktion/Verarbeitung:*
  Basis-Modell — Klassendiagramm aus Haupt/Sub-Diensten, Entitäten, Rollen, Transparenzen — erstellen
- *halb-automatisiert/UML-Interaktion/Verarbeitung:*
  nicht-transparente Dienste benennen, Rollen zu Diensten zuweisen und entsprechend benennen

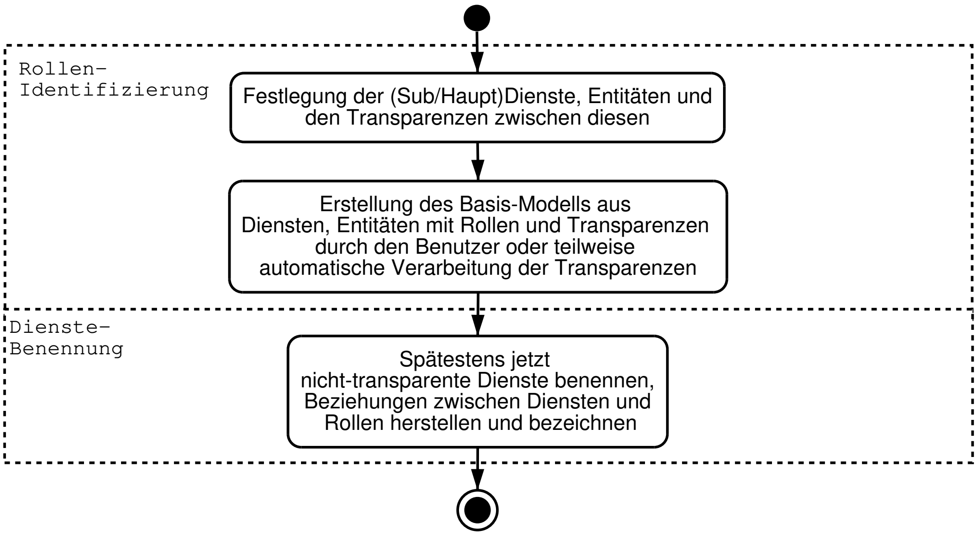

Abbildung 3.4: Benutzerinteraktionen beim Basis-Modell-Workflow

3.3.4 Interaktionen für die Top-Down-Workflows

Evtl. wird nach Erstellung der Dienstsicht aufgehört, da man die Implementierung zu einem anderen Provider auslagern möchte. Das Werkzeug muss entsprechend dem Benutzer die Möglichkeit geben, nach Erstellung der Dienstsicht mit der Modellierung aufzuhören. Das heißt keinen Realisierungssicht Top-Down-Workflow anlegen, wenn es nicht notwendig ist. Das Werkzeug könnte den Benutzer nach Abarbeitung des Dienstsicht-Workflows fragen, ob er die Realisierungssicht nun erstellen auch erstellen will oder nicht

Soll der modellierte Dienst dann doch schließlich realisiert werden, muss aber auch die Möglichkeit vom Werkzeug gegeben werden, die Modellierung für die Realisierungssicht fortzusetzen.

3.3.5 Interaktionen beim Dienstsicht Top-Down-Workflow

Eine Übersicht über die Interaktionen für den Dienstsicht Top-Down-Workflow findet sich in Abb. 3.5.

Festlegung der Funktionalität Hier wird die Funktionalität des Dienstes durch Festlegen der Prozesse, Anlegen von Use-Case- und Aktivitätsdiagrammen aus den Prozessen und in den Rollen auftretenden Entitäten festgelegt.

Grundlegend müssen hier zunächst aber die Phasen des Lebenszyklus und eine Prozessklassifizierung spezifiziert werden. Diese sind jedoch i.a. dienstunabhängig und vom Werkzeug vorgegeben werden. Wünscht der Benutzer jedoch eine Änderung oder Verfeinerung davon, so wird ihm die Möglichkeit dazu gegeben.

Insgesamt ergeben sich folgende Interaktionen mit dem Benutzer:

- *halb-automatisiert/Strukturierungs-Eingabe/Eingabe:*
  - Festlegung bzw. Erweiterung (Verfeinerung) von Prozessklassifizierung
  - Festlegung bzw. Erweiterung (Verfeinerung) der Phasen des Dienstlebenszyklus
- *nicht-automatisiert/einfache nUML-Interaktion/Eingabe:*
  Eingabe der Kunden-Funktionalitäts-Anforderung

- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukturierung/Verarbeitung:*
  Festlegung und Zuordnung von Prozessen zu Klassen und Lebenszyklus-Phasen, evtl automatische Verarbeitung der Kunden-Funktionalität
- *halb- oder nicht-automatisiert/UML-Interaktionen/Verarbeitung:*
  Erstellen von Use-Case-Diagrammen, in denen die Prozesse und die Entitäten mit (evtl. verfeinerten) Rollen auftauchen
- *halb- oder nicht-automatisiert/UML-Interaktionen/Verarbeitung:*
  zu jedem Prozess Erstellung eines Aktivitätsdiagramms aus Aktivitäten; Unterstützung von Sequenzen, Parallelen Pfaden, Schleifen, Bedingungen, Signale, etc.

Die erstellten Use-Case- und Aktivitätsdiagramme müssen im weiteren Verlauf der Modellierung erweitert werden. Das Werkzeug muss also eine Erweiterbarkeit ermöglichen und es muss darüber nachgedacht werden, wie die verschiedenen Versionen der einzelnen Diagramme verwaltet werden.

Beispielsweise stellt sich, falls ein Aktivitätsdiagramm später nicht nur verfeinert, sondern stark geändert wird, das Problem, dass die weniger feine, frühere Version eigentlich entsprechend mit-angepasst werden müsste. Am einfachsten wäre vermutlich nur eine Version zuzulassen, die einfach erweitert wird oder aber man verändert einfach eine Kopie als neue Version und überprüft keine Konsistenzen zwischen verschiedenen Versionen.

QoS-Definition Hier werden QoS-Parameter bzw. die Typen der QoS-Parameter für den Dienst und zwar strukturiert nach QoS-Dimensionen und Prozessklassen festgelegt.

Der Benutzer muss daher als erstes die Gelegenheit erhalten, die QoS-Dimensionen einzugeben bzw. Vorgaben dafür zu erweitern/ändern. Die Prozessklassifikation wird automatisch von der vorigen Aktivität übernommen.

Dann müssen die QoS-Parameter an Hand der Prozessklassifizierung, der Dimensionen und vor allem der Use-Case- und Aktivitätsdiagramme identifiziert und eingeordnet werden.

Zusammengefasst, muss das Werkzeug die folgenden Interaktionen durchführen:

- *halb-automatisiert/Strukturierungs-Eingabe/Eingabe:*
  Eingabe von QoS-Dimension
- *nicht automatisiert/einfache nUML-Interaktion/Eingabe:*
  Eingabe der Kunden-QoS-Anforderungen
- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  Eingabe und Zuordnung der QoS-Parameter, hierbei speziell Zugriff auf Use-Case- und Aktivitätsdiagramme, evtl. automatische Verarbeitung der QoS-Kundenanforderung

Wie das Werkzeug hier genau den Benutzer unterstützt, ist noch zu klären. Auf jeden Fall muss eine Ansicht oder evtl. sogar eine Änderung der Use-Case- und Aktivitätsdiagramme, die ja auch nach Prozessklassen eingeteilt sind, möglich sein.

In der Beschreibung der Methodik wird erwähnt, dass vor allem Anfang und Ende eines durch ein Aktivitätsdiagramme modellierten Prozesses dem Benutzer über QoS-Parameter Hinweise liefern kann. Unter Umständen kann das Werkzeug speziell diese Punkte der Diagramme dem Benutzer einfach präsentieren, um das Auffinden der QoS-Werte zu erleichtern. Auch ist bei dieser Aktivität noch nicht klar, in welcher Weise die Kunden-QoS-Anforderungen eingegeben werden und inwieweit sie automatisch vom Werkzeug verarbeitet werden können.

Clients und Dienstzugangspunkte Hier geschieht die Festlegung des Dienstzugangspunkte und der Clients für den Nutzdienst bzw. das Dienstmanagement. Hierzu muss das Werkzeug dem Benutzer Zugriff auf

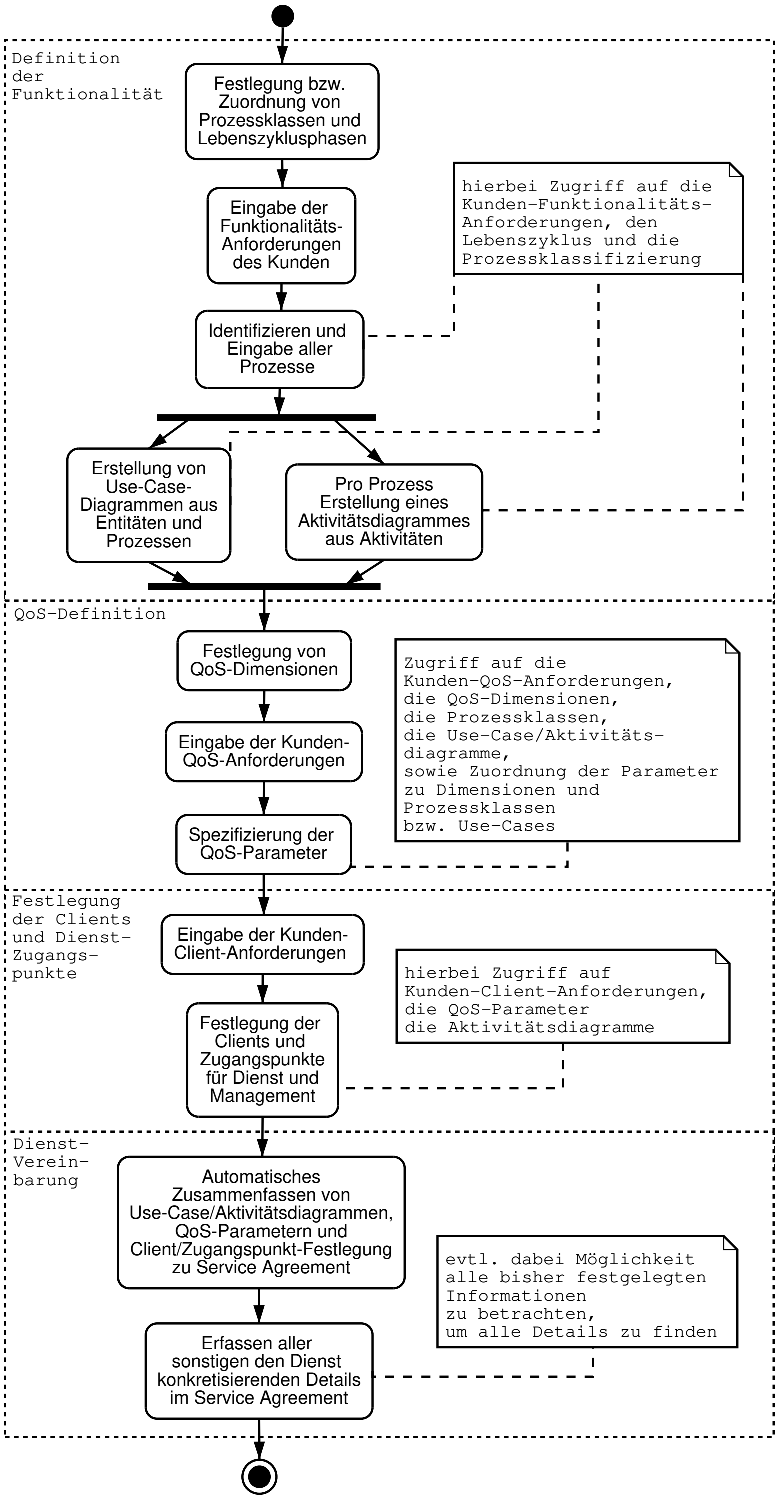

Abbildung 3.5: Benutzerinteraktionen beim Dienstsicht Top-Down-Workflow

die Aktivitätsdiagramme, die QoS-Parameter und die Kunden-QoS-Anforderungen gestatten. Inwieweit dies automatisch geschehen kann, ist noch ungewiss.

Notwendige Interaktionen mit dem Benutzer sind insgesamt:

- *nicht automatisiert/einfache nUML-Interaktion/Eingabe:*
  Eingabe der Client-Anforderungen des Kunden
- *halb- oder nicht automatisiert/nUML-Interaktion mit UML/Verarbeitung:*
  Festlegen des Client Spezifikation und der Dienstzugangspunkte, sowohl für Dienst selbst als auch das Management, dabei Zugriff auf die Aktivitätsdiagramme. Evtl. automatische Verarbeitung der Anforderungen

Ähnlich wie bei den vorigen Aktivitäten stellt sich hier die Frage, in welcher Form die Kundenanforderungen vorliegen und verarbeitet werden. Auch die Struktur der Ausgabe selbst ist noch näher festzulegen. Auf jeden Fall beschreibt diese jeweils für den Nutzdienst selbst als auch für das Management, den Dienstzugangspunkt als auch den Client dafür.

In der Beschreibung der Methodik wird festgelegt, dass diese Beschreibung verbunden ist mit einem oder mehreren der Prozesse als textuelle Beschreibung, formale Spezifikation oder eine Referenz zu einem Standard.

Festlegung der Dienstvereinbarung Hier sollte das Werkzeug alle benötigten Eingaben – soweit möglich – automatisch zu einer Dienstvereinbarung zusammensetzen und dann dem Benutzer ermöglichen, konkretisierende Festlegungen für Werte, etc. zu machen.

Hier sind notwendige Benutzerinteraktionen demnach:

- *voll-automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  Automatisches Zusammenfassen der verschiedenen Informationen
- *nicht oder halb-automatisiert/einfache nUML-Interaktion/Verarbeitung:*
  Spezifizieren konkreter Werte und sonstiger Informationen zur Konkretisierung

Wie genau diese zusätzlichen Festlegungen aussehen bzw. wie genau sie strukturiert sind, muss noch bestimmt werden. Man könnte sie z.B. zumindest nach Klassen wie etwa Dienstparameter, QoS-Parameter, Client/Dienstzugangspunkt-Parameter, Gesetzlich-Vertragliches, etc, einteilen oder sogar genauer Elementen der erstellten Diagramme zuordnen.

3.3.6 Interaktionen beim Realisierungssicht Top-Down-Workflow

Abb. 3.6 zeigt eine Übersicht über die Interaktionen des Realisierungssicht Top-Down-Workflows.

Provider-interner Service-Management-Prozess Zur Definition des provider-internen SM-Prozesses werden die Prozesse und die entsprechenden Use-Case- und Aktivitätsdiagramme der Dienstsicht erweitert.

Das Werkzeug sollte daher an dieser Stelle gestatten, die Liste der Prozesse zu erweitern, die Use-Case- und Aktivitätsdiagramme für vorhandene Prozesse zu verfeinern und für neue Prozesse neue Aktivitätsdiagramme (und ggf. neue Use-Case-Diagramme) zu erstellen.

Die zur Strukturierung verwendetet TOM-Prozessklassifizierung kann vom Werkzeug vorgegeben oder durch den Benutzer ggf. erweitert erden.

Also ergeben sich folgende Interaktionen:

- *halb-automatisiert/Strukturierungs-Eingabe/Eingabe:*
  TOM-Prozessklassifizierung evtl. erweitern (z.B. nach ITIL)

- *nicht automatisiert/nUML-Interaktion mit Strukturierung/Verarbeitung:*
  vorhandene Prozesse neu einordnen, neue Prozesse festlegen, einordnen
- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  - Use-Case-Diagramme erweitern, um neue Prozesse zu beschreiben, evtl. Vorgaben anhand der Klassifizierung der neuen Prozesse machen
  - vorhandene Aktivitätsdiagramme erweitern bzw. für neue Prozesse durch neue Aktivitätsdiagramme beschreiben, evtl. Vorgaben anhand der Klassifizierung der neuen Prozesse machen

Outsourcing-Entscheidungen Hier werden Prozesse bestimmt, die auszulagern sind und dann das Basis-Modell entsprechend erweitert.

Wie genau das Werkzeug bei der Identifikation der auszulagernden Prozesse helfen kann, ist noch unklar.

Hier ergeben sich nach der Methodik folgende Interaktionen:

- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  bei Zugriff auf die Aktivitätsdiagramme Auswahl auszulagernder Prozesse
- *halb-automatisiert/UML-Interaktion/Verarbeitung:*
  pro ausgelagertem Prozess Basis-Modell erweitern
- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  pro ausgelagertem Prozess ggf. Aktivitätsdiagramm des Prozesses erweitern; evtl. anhand der Klassifizierung des Prozesses mögliche Vorgaben bieten
- *halb- oder nicht automatisiert/einfache nUML-Interaktion/Verarbeitung:*
  pro ausgelagertem Prozess Clients für Sub-Dienst und Management davon festlegen; evtl .Vorgaben nach Klassifizierung des Prozesses bieten

Ist ein Sub-Dienst von der Kundenseite des Hauptdienstes aus transparent, so wird er als intern markiert. Dies muss das Basis-Modell als Stereotyp unterstützen.

Identifikation von Ressourcen und BMF In dieser Aktivität werden zunächst Ressourcen und BMF sowie Zugangspunkte zu den Prozessen für diese identifiziert. Weiterhin wird durch Kollaborationsdiagramme die Dienstlogik von Nutzdienst und Management beschrieben, sowie einige abschließende Festlegungen für die Realisierung des Dienstes getroffen.

Aus der Methodik ergeben sich die folgende Reihe von Interaktionen mit dem Benutzer:

- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  anhand der Aktivitätsdiagramme wird die Liste der internen Ressourcen/Basic Management Functionality, geordnet nach den Prozessklassen, gefunden
- *nicht automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  Finden der Zugangspunkte für interne und externe Ressourcen/BMF zu den Prozessen
- *nicht automatisiert/nUML-Interaktion mit Strukturierung/Verarbeitung:*
  Zuordnung von Zugangspunkten und internen bzw. externen Ressourcen/BMF
- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  Erstellen von Kollaborationsdiagrammen zur Modellierung der Interaktionen zwischen Prozessen und internen bzw. externen Ressourcen/BMF
- *nicht automatisiert/einfache nUML-Interaktion/Verarbeitung:*
  - Abbildung von QoS-Parametern auf messbare Werte
  - Spezifizierung der Dienstarchitektur

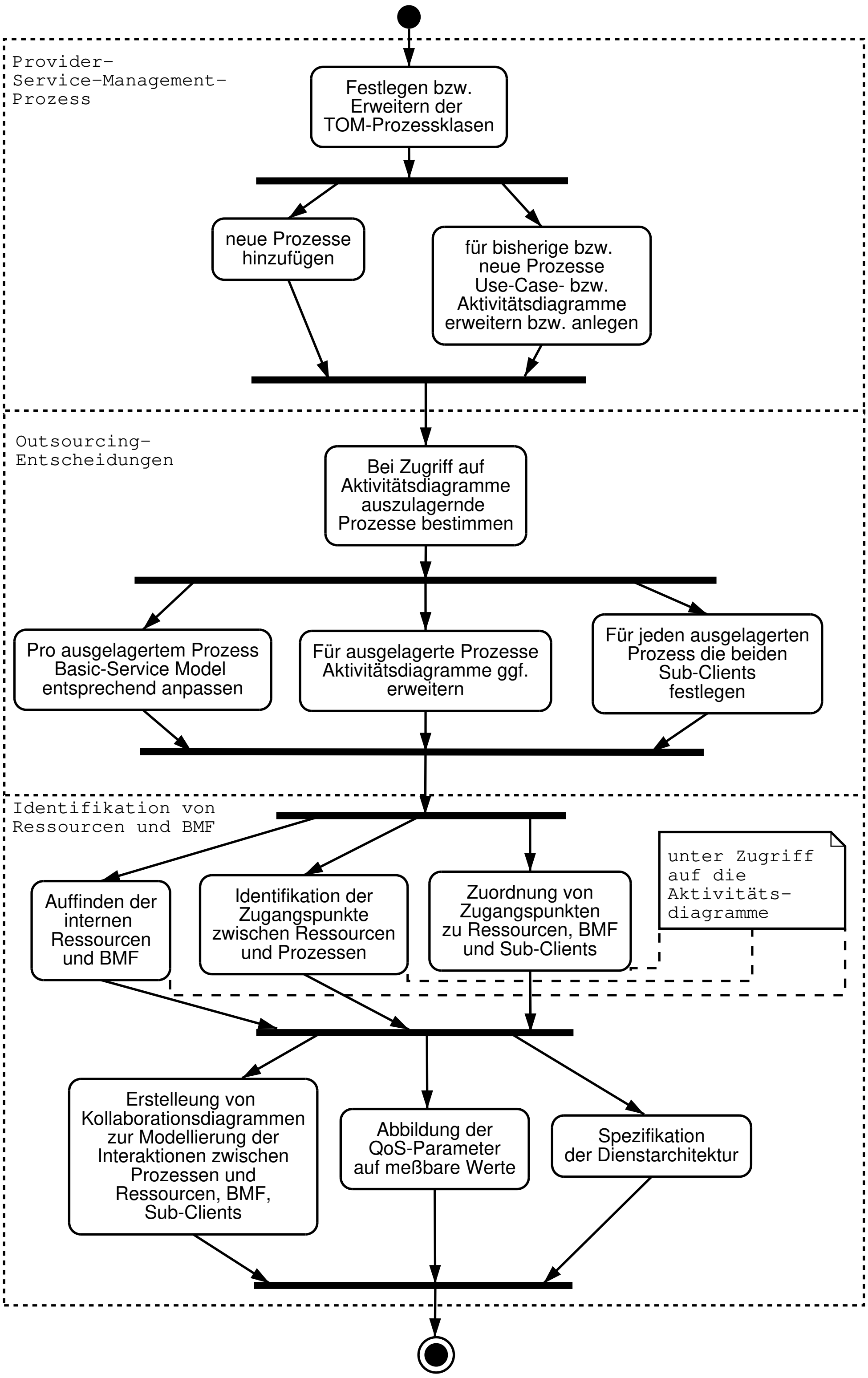

Abbildung 3.6: Benutzerinteraktionen beim Realisierungssicht Top-Down-Workflow

3.3.7 Interaktionen beim Realisierungssicht Bottom-Up-Workflow

In Abb. 3.7 wird eine Übersicht über die Interaktionen des Realisierungssicht Bottom-Up-Workflows gegeben.

Use-Case-Identifikation Bei dieser Aktivität werden Prozesse identifiziert und dafür Use-Case-Diagramme erstellt. Dies läuft ähnlich zur Funktionalitätsdefinition beim Top-Down-Vorgehen ab, allerdings werden hier auch provider-interne Prozesse betrachtet und keine Aktivitätsdiagramme erstellt.

Es ergeben sich damit folgenden Interaktionen:

- *halb automatisiert/Strukturierungs-Eingabe/Eingabe:*
  Eingabe bzw. Erweiterung der Prozessklassen
- *nicht automatisiert/einfache nUML-Interaktion/Eingabe:*
  Spezifikation der Provider-Anforderungen
- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukturierung/Verarbeitung:*
  Eingabe der Prozesse und Zuordnung zu Prozessklassen, evtl. automatische Verarbeitung der Provider-Anforderungen
- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  Erstellung von Use-Case-Diagrammen zu den Prozessen

Sub-Client, Ressourcen, BMF-Identifikation Hier werden interne Ressourcen bzw. BMF, sowie verwendete Sub-Dienste spezifiziert. Das Basis-Modell wird ggf. entsprechend erweitert.

Folgende Benutzerinteraktionen sind daher notwendig:

- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  unter Zugriff auf die Use-Cases Erweiterung des Basis-Modells um neue Sub-Dienste
- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  unter Zugriff auf die Use-Cases Identifikation der internen Ressourcen, BMF und Sub-Dienst/Sub-Dienstmanagement-Clients, strukturiert nach Prozessklassen

Für die Art ihrer Darstellung bzw. Verarbeitung gelten im wesentlichen die gleichen Aussagen, wie im Top-Down-Fall bei den Aktivitäten Outsourcing-Entscheidungen und Ressourcenidentifikation. Das Werkzeug sollte aber z.B. feststellen, ob zu jedem Sub-Dienst auch die entsprechenden Sub-Clients definiert wurden.

Dienstlogik-Definition Die Dienstlogik von Nutzdienst und Management wird festgelegt, im wesentlichen durch Angabe von Kollaborationsdiagrammen.

Daraus ergeben sich folgende Interaktionen:

- *nicht automatisiert/einfache nUML-Interaktion/Verarbeitung:*
  Definition von Dienstzugangspunkten für den Hauptdienst
- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  Pro Use-Case, Erstellung eines Kollaborationsdiagrammes unter Einbeziehung der Sub-Clients, Ressourcen und BMF, Dienstzugangspunkten
- *nicht automatisiert/einfache nUML-Interaktion/Verarbeitung:*
  Festlegung der Dienstarchitektur

In wieweit all dies automatisch vom Werkzeug erledigt werden kann, hängt von der Form der Eingaben ab und liegt noch nicht fest.

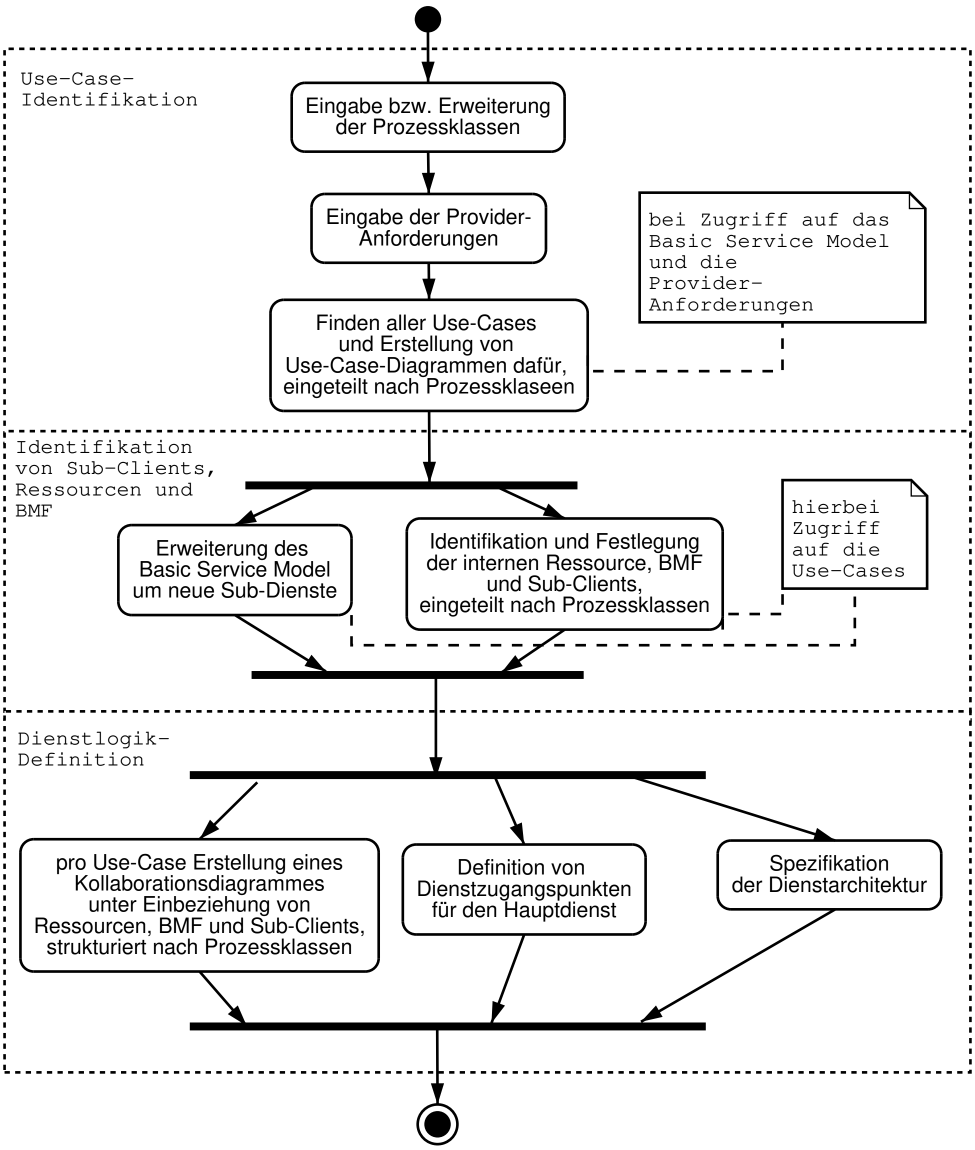

Abbildung 3.7: Benutzerinteraktionen beim Realisierungssicht Bottom-Up-Workflow

3.3.8 Interaktionen beim Dienstsicht Bottom-Up-Workflow

Eine Übersicht über die Interaktionen des Dienstsicht Bottom-Up-Workflows wird in Abb. 3.8 gegeben.

Definition der Funktionalität Zur Funktionalitätsdefinition des Dienstes werden aus den Prozessen bzw. den mit diesen verbundenen Use-Case-Diagrammen alle Aspekte entfernt, die für Dienstnehmerseite nicht sichtbar sein sollen. Für die verbleibenden Prozesse werden aus ihren Kollaborationsdiagrammen Aktivitätsdiagramme erstellt.

Hiermit ergeben sich nachfolgende Interaktionen mit dem Benutzer:

- *halb-automatisiert/Strukturierungs-Eingabe/Eingabe:*
  - Eingabe bzw. Erweiterung der Service-Lebenszyklus-Phasen
  - Übernahme bzw. Auswahl der Prozessklassifizierung aus den beim Realisierungssicht-Workflow verwendeten Prozessklassen
- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukturierung/Verarbeitung:*
  Übernahme bzw. Auswahl aus den Prozessen bzw. entspr. Use-Case-Diagrammen der Realisierungssicht
- *halb- oder nicht automatisiert/UML-Interaktion/Verarbeitung:*
  zu jedem Use-Case Ableitung eines Aktivitätsdiagrammes aus den Kollaborationsdiagrammen evtl. teilweise automatisch

Eine Frage ist, inwieweit das Werkzeug den Benutzer bei dem Entfernen von provider-internen Prozessen unterstützen kann. Strukturiert werden die Aktivitätsdiagramme nach Service-Lebenszyklus-Phasen und der schon im Realisierungssicht-Workflow verwendeten Prozessklassen.

QoS-Definition Die QoS-Parameter werden an dieser Stelle unter Zuhilfenahme der Use-Case, Aktivitätsdiagramme und den BMF abgeleitet.

Es gibt folgende Benutzerinteraktionen:

- *halb-automatisiert/Strukturierungs-Eingabe/Eingabe:*
  Auswahl bzw. Eingabe von QoS-Dimensionen
- *halb- oder nicht automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  Unter Zugriff auf die Use-Case- und Aktivitätsdiagramme, sowie auf die im Realisierungssicht-Workflow ermittelten BMF Definition von QoS-Parametern, strukturiert nach Prozessklassen und QoS-Dimensionen

Zugangspunkt-Definition Einzige Interaktion ist hier die Übernahme der Spezifikation der Zugangspunkte aus der Realisierungssicht *(voll-automatisiert/einfache nUML-Interaktion/Verarbeitung)*. Diese kann automatisch erfolgen.

Dienstvereinbarung Diese Aktivität ist ähnlich wie die Festlegung der Dienstvereinbarung beim Top-Down-Vorgehen. Hier werden zusätzlich noch die Clients definiert.

Damit gibt es hier folgende Benutzerinteraktionen:

- *voll-automatisiert/nUML-Interaktion mit Strukt. und UML/Verarbeitung:*
  automatische Zusammenfassung aller vorher definierten Informationen
- *nicht automatisiert/einfache nUML-Interaktion/Eingabe:*
  Spezifikation der Clients

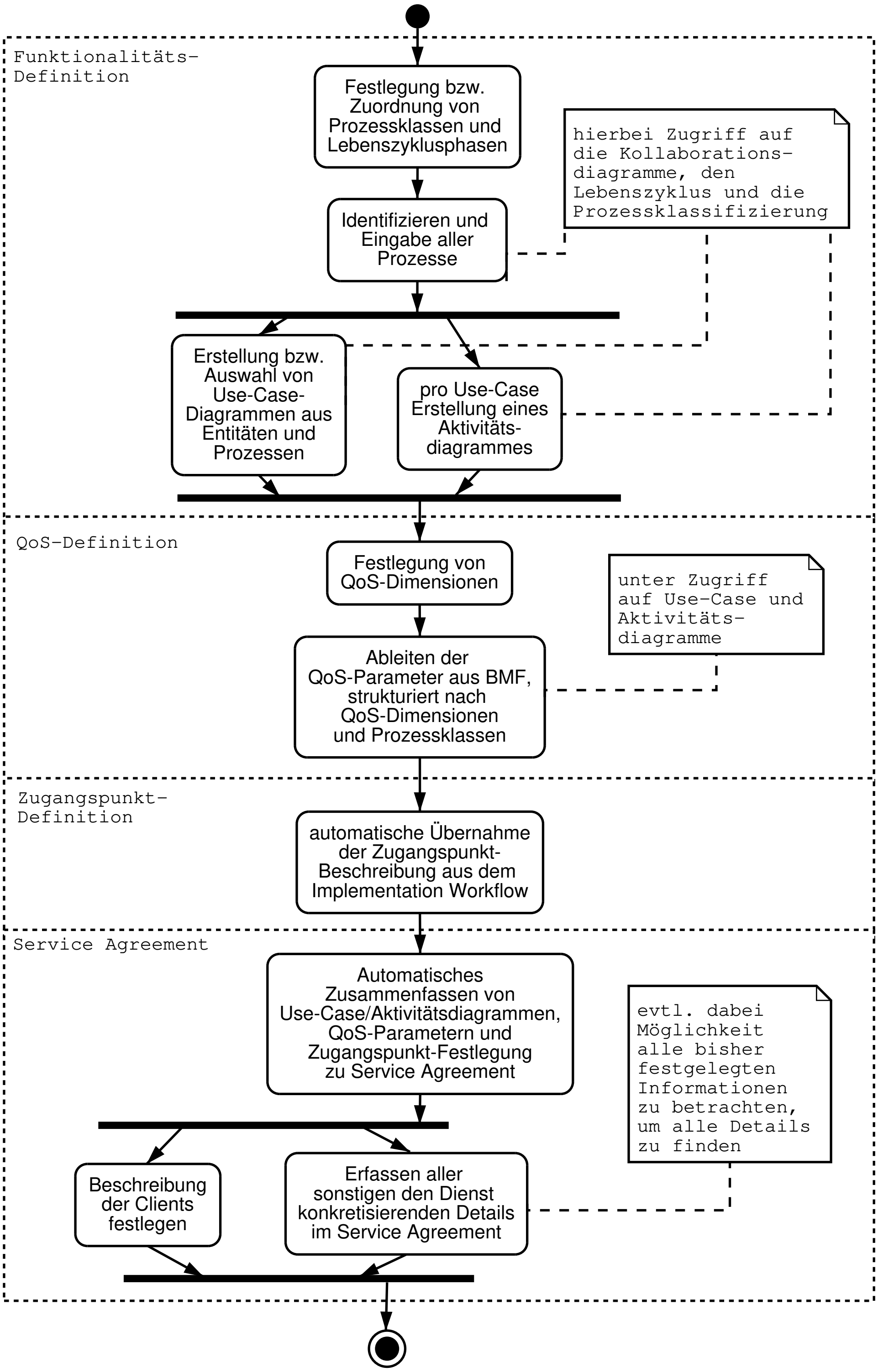

Abbildung 3.8: Benutzerinteraktionen beim Dienstsicht Bottom-Up-Workflow

- *halb- oder nicht automatisiert/einfache nUML-Interaktion/Verarbeitung:*
  Festlegung aller sonstigen den Dienst konkretisierenden Details

3.3.9 Zusammenfassende Betrachtungen

Insgesamt ist festzustellen, das bei einer Reihe von Artefakte noch nicht genau feststeht, in welcher Form sie in das Werkzeug eingegeben werden. Auch wie und ob das Werkzeug viele Artefakte (teilweise) automatisch weiterverarbeiten kann, ist noch zu bestimmen. Daher ist für eine ganze Reihe von Interaktionen auch noch nicht klar, ob sie halb- oder nicht-automatisiert sind.

Simpelster Ansatz wäre jeweils reiner Text, den nur der Werkzeug-Benutzer selbst als Anhaltspunkt verwendet. Einige der Artefakte haben im wesentlichen die Form einer Liste, wobei die Struktur eines Listenelementes selbst noch nicht feststeht. Zumindest eine Strukturierung nach z.B. Prozessklassen ist in diesem Fall bereits von der Methodik schon vorgegeben. In wieweit hier mehr strukturierte Information dem Werkzeug zur automatische Verarbeitung gegeben werden kann, muss in diesen Fällen noch festgestellt werden.

Innerhalb der verschiedenen UML-Diagramme wären auch spezielle Flags zu den einzelne UML-Elementen denkbar, die vom Werkzeug automatisch später weiterverwendet werden könnten. So könnte z.B. beim Realisierungssicht Top-Down-Workflow aus den Aktivitätsdiagrammen an Hand eines Flags (Stereotyp oder ähnliches) namens „Ressourcen-Zugangspunkt“ für Prozess-Aktivitäten automatisch eine Liste von Zugangspunkten für Ressourcen/BMF/Sub-Clients vom Werkzeug vorgeschlagen werden.

3.3.10 Klassifizierungsübersicht über die Interaktionen

Im Folgenden wird nun eine Übersicht über die Interaktionen, klassifiziert nach dem Grad der Automatisierungs und den Typen der vorkommenden Artefakte gegeben:

Es gibt eine kleine Zahl von Interaktionen, die voll-automatisiert abgearbeitet werden können:

- *Dienstsicht Top-Down, Dienstvereinbarung/ voll-automatisiert/Verarbeitung:*
  automatisches Zusammenfassen aller vorher definierten Informationen
- *Dienstsicht Bottom-Up, Dienstvereinbarung/ voll-automatisiert/Verarbeitung:*
  automatische Zusammenfassung aller vorher definierten Informationen
- *Dienstsicht Bottom-Up, Zugangspunkte/ voll-automatisiert/Verarbeitung:*
  Übernahme der Spezifikation der Zugangspunkte aus der Realisierungssicht

Alle sonstigen Interaktionen sind halb- oder nicht automatisiert, d.h. der Benutzer muss allein oder in Zusammenarbeit mit dem Werkzeug die Artefakte eingeben bzw. verarbeiten. In vielen Fällen muss noch genau festgelegt werden, ob und wie das Werkzeug durch automatische Aktionen hier den Benutzer zusätzlich unterstützt:

- Basiseinstellungen:
  - *nicht automatisiert/Eingabe:*
    Festlegung des Modellierungsfalles bzw. der Vorgehensweise (Top-Down oder Bottom-Up)
  - *nicht automatisiert/Eingabe:*
    Festlegung des Rückgriffs auf Workflows früherer Modellierungen
  - *nicht automatisiert/Eingabe:*
    sonstige allgemeine Festlegungen
- Strukturierungs-Eingaben:
  - *Dienstsicht Top-Down, Funktionalität/ halb-autom./Eingabe:*
    Festlegung bzw. Erweiterung von Prozessklassifizierung

- *Real.-Sicht Bottom-Up, Use-Case-Identif./ halb autom./Eingabe:*
  Eingabe bzw. Erweiterung der Prozessklassifizierung
- *Real.-Sicht Top-Down, SM-Prozess/ halb-autom./Eingabe:*
  TOM-Prozessklassifizierung eingeben bzw. erweitern
- *Real.-Sicht Bottom-Up, Funktionalität/ halb-autom./Eingabe:*
  Übernahme bzw. Auswahl der Prozessklassifizierung aus dem beim Realisierungssicht-Workflow
- *Dienstsicht Top-Down, Funktionalität/ halb-autom./Eingabe:*
  Festlegung bzw. Erweiterung des Dienstlebenszyklus
- *Real.-Sicht Bottom-Up, Funktionalität/ halb-autom./Eingabe:*
  Eingabe bzw. Erweiterung der Dienstlebenszyklus-Phasen
- *Dienstsicht Top-Down, QoS-Definition/ halb-autom./Eingabe:*
  Eingabe von QoS-Dimension
- *Real.-Sicht Bottom-Up, QoS-Definition/ halb-autom./Eingabe:*
  Eingabe von QoS-Dimensionen

• UML-Interaktionen:

- *Basis-Modell, Rollen-Identif./ nicht oder halb-autom./Verarbeitung:*
  Basis-Modell als Klassendiagramm aus Diensten und Entitäten mit Rollen und Transparenzen erstellen, evtl. dabei automatische Verarbeitung der Transparenzen
- *Basis-Modell, Dienste-Benennung/ halb-autom./Verarbeitung:*
  nicht-transparente Dienste benennen, Rollen zu Diensten zuweisen und entsprechend benennen
- *Real.-Sicht Top-Down, Outsourcing/ halb-autom./Verarbeitung:*
  pro ausgelagertem Prozess Basis-Modell erweitern, der Sub-Dienst kann automatisch angelegt werden, die zugehörigen Rollen müssen aber durch den Benutzer gegeben werden
- *Real.-Sicht Bottom-Up, Ress.+Sub-Clients/ halb- oder nicht autom./Verarbeitung:*
  unter Zugriff auf die Use-Cases Erweiterung des Basis-Modells um neue Sub-Dienste
- *Dienstsicht Top-Down, Funktionalität/ halb- oder nicht autom./Verarbeitung:*
  Erstellen von Use-Case-Diagrammen aus Prozessen und Entitäten mit Rollen evtl. anhand der Klassifizierung des Prozesses mögliche Vorgaben bieten
- *Real.-Sicht Bottom-Up, Use-Case-Identif./ halb- oder nicht autom./Verarbeitung:*
  Erstellung von Use-Case-Diagrammen zu den Prozessen, evtl. anhand der Klassifizierung des Prozesses mögliche Vorgaben bieten
- *Real.-Sicht Top-Down, SM-Prozess/ halb- oder nicht autom./Verarbeitung:*
  Use-Case-Diagramme mit neuen Prozessen erweitern, evtl. anhand der Klassifizierung der neuen Prozesses mögliche Vorgaben bieten
- *Dienstsicht Top-Down, Funktionalität/ halb- oder nicht-autom./Verarbeitung:*
  pro Prozess Erstellung eines Aktivitätsdiagramms, anhand Klassifizierung des zugehörigen Prozesses mögliche Vorgaben geben
- *Dienstsicht Bottom-Up, Funktionalität/ halb- oder nicht autom./Verarbeitung:*
  zu jedem Use-Case Ableitung eines Aktivitätsdiagrammes aus Kollaborationsdiagramm
- *Real.-Sicht Top-Down, SM-Prozess/ halb- oder nicht autom./Verarbeitung:*
  vorhandene Aktivitätsdiagramme erweitern bzw. neue Prozesse durch neue Aktivitätsdiagramme beschreiben, anhand der Klassifizierung der neuen Prozesse mögliche Vorgaben geben

- *Real.-Sicht Top-Down, Outsourcing/ halb- oder nicht autom./Verarbeitung:*
pro ausgelagertem Prozess ggf. Aktivitätsdiagramm des Prozesses erweitern, evtl. anhand der Klassifizierung des Prozesses mögliche Vorgaben bieten
- *Real.-Sicht Top-Down, Ressourcen/ halb- oder nicht autom./Verarbeitung:*
Erstellen von Kollaborationsdiagrammen zur Modellierung der Interaktionen zwischen Prozessen und internen bzw. externen Ressourcen/BMF
- *Real.-Sicht Bottom-Up, Dienstlogik-Definition/ halb- oder nicht autom./Verarbeitung:*
Pro Use-Case, Erstellung eines Kollaborationsdiagrammes unter Einbeziehung von Ressourcen/BMF/Sub-Clients und Zugangspunkten

• einfache nUML-Interaktionen:

- *Basis-Modell, Rollen-Identif./ nicht autom./Eingabe:*
Festlegen der Haupt- und Sub-Dienste, Entitäten und Transparenzen zwischen diesen
- *Dienstsicht Top-Down, Funktionalität/ nicht-autom./Eingabe:*
Eingabe der Kunden-Funktionalitäts-Anforderung
- *Dienstsicht Top-Down, QoS-Definition/ nicht autom./Eingabe:*
Eingabe der Kunden-QoS-Anforderungen
- *Dienstsicht Top-Down, Zugangspunkte/ nicht autom./Eingabe:*
Eingabe der Client-Anforderungen des Kunden
- *Real.-Sicht Bottom-Up, Use-Case-Identif./ nicht autom./Eingabe:*
Spezifikation der Provider-Anforderungen
- *Dienstsicht Top-Down, Dienstvereinbarung/ nicht oder halb-autom./Verarbeitung:*
Spezifizieren von sonstigen den Dienst konkretisierenden Informationen
- *Dienstsicht Bottom-Up Dienstvereinbarung/ halb- oder nicht autom./Verarbeitung:*
Festlegung von sonstigen den Dienst konkretisierenden Details
- *Real.-Sicht Top-Down, Ressourcen/ nicht autom./Verarbeitung:*
Abbildung von QoS-Parametern auf messbare Werte
- *Real.-Sicht Bottom-Up, Dienstlogik-Definition/ nicht autom./Verarbeitung:*
Definition von Dienstzugangspunkten für den Hauptdienst
- *Dienstsicht Bottom-Up, Dienstvereinbarung/ nicht autom./Eingabe:*
Spezifikation der Clients
- *Real.-Sicht Top-Down, Outsourcing/ halb- oder nicht autom./Verarbeitung:*
pro ausgelagertem Prozess Clients für Sub-Dienst und dessen Management festlegen, evtl .Vorgaben nach Klassifizierung des Prozesses bieten
- *Real.-Sicht Bottom-Up, Dienstlogik-Definition/ nicht autom./Verarbeitung:*
Festlegung der Dienstarchitektur
- *Real.-Sicht Top-Down, Ressourcen/ nicht autom./Verarbeitung:*
Spezifizierung der Dienstarchitektur

• nUML-Interaktionen mit Strukturierung:

- *Dienstsicht Top-Down, Funktionalität/ halb- oder nicht autom./Verarbeitung:*
Festlegung und Zuordnung von Prozessen zu Klassen und Lebenszyklus-Phasen, evtl. automatische Verarbeitung der Kunden-Funktionalität
- *Real.-Sicht Bottom-Up, Use-Case-Identifikation/ halb- oder nicht autom./Verarbeitung:*
Eingabe der Prozesse und Zuordnung zu Prozessklassen, evtl. automatische Verarbeitung der Provideranforderungen

  - *Real.-Sicht Top-Down, SM-Prozess/ nicht autom./Verarbeitung:*
    vorhandene Prozesse neu einordnen, neue Prozesse festlegen und einordnen
  - *Dienstsicht Bottom-Up, Funktionalität/ halb- oder nicht autom./Verarbeitung:*
    Auswahl aus den Prozessen bzw. entspr. Use-Case-Diagrammen der Realisierungssicht, evtl. bereits durch bestimmte Flags als intern markierte Prozesse automatisch weglassen
  - *Real.-Sicht Top-Down, Ressourcen/ nicht autom./Verarbeitung:*
    Zuordnung von Zugangspunkten und internen bzw. externen Ressourcen/BMF

- nUML-Interaktionen ohne Strukturierung mit UML:
  - *Dienstsicht Top-Down, Zugangspunkte/ halb- oder nicht autom./Verarbeitung:*
    Festlegen der Client-Spezifikation und der Dienstzugangspunkte, Zugriff auf Aktivitätsdiagramme

- nUML-Interaktionen mit Strukturierung und UML:
  - *Dienstsicht Top-Down, QoS-Definition/ halb- oder nicht autom./Verarbeitung:*
    Eingabe und Zuordnung der QoS-Parameter, Zugriff auf Use-Case- und Aktivitätsdiagramme, evtl. automatische Verarbeitung der QoS-Kundenanforderung
  - *Dienstsicht Bottom-Up, QoS-Definition/ halb- oder nicht autom./Verarbeitung:*
    Definition von QoS-Parametern, strukturiert nach Prozessklassen und QoS-Dimensionen, Zugriff auf Use-Case- und Aktivitätsdiagramme, BMF
  - *Real.-Sicht Top-Down, Outsourcing/ halb- oder nicht autom./Verarbeitung:*
    Auswahl auszulagernder Prozesse, Zugriff auf die Aktivitätsdiagramme
  - *Real.-Sicht Top-Down, Ressourcen/ nicht autom./Verarbeitung:*
    Finden der Zugangspunkte für interne und externe Ressourcen/BMF zu Prozessen
  - *Real.-Sicht Bottom-Up, Ress.+Sub-Clients/ halb- oder nicht autom./Verarbeitung:*
    Festlegen der Ressourcen, BMF und Sub-Dienst/Sub-Dienstmanagement-Clients, strukturiert nach Prozessklassen, Zugriff auf Use-Case-Diagramme
  - *Real.-Sicht Top-Down, Ressourcen/ halb- oder nicht autom./Verarbeitung:*
    Ressourcen/BMF spezifizieren, geordnet nach Prozessklassen, Zugriff auf Aktivitätsdiagramme

3.4 Architektur des Werkzeuges

Aufbauend auf den vorangegengenen Analysen wird nun ein erster Entwurf der Architektur des zu entwickelnden Werkzeuges vorgestellt.

Das zu entwickelnde Werkzeug muss die gesamte Anwendungsmethodik mit all den im vorangegangenen Abschnitt betrachteten Benutzerinteraktionen realisieren. Zusätzlich dazu wäre es wünschenswert, wenn das Werkzeug in Bezug auf zukünftige Änderungen bzw. Erweiterungen der Methodik oder des Dienstmodells leicht anpassbar wäre. Vor allem die Workflowunterstützung muss daher flexibel sein. Daher bietet sich als erster Entwurf die folgende Architektur an:

Diese Architektur sieht für das Werkzeug im wesentlichen drei spezifische, funktionale Komponenten vor (siehe Abb. 3.9):

- **Methodik-Spezifikations-Komponente:**
  Dient zur Spezifikation der Methodik, d.h. der Workflows mit all ihren Aktivitäten.
- **Methodik-Ausführungs-Komponente:**
  Dient der Ausführung der Methodik, die durch die Methodik-Spezifikations-Komponente spezifiziert

wurde; Hiermit wird die Erstellung, Verwaltung und Ausführung der Workflows mit ihren Aktivitäten und Artefakten gesteuert.

- **Datenbasis-Komponente:**
  Wird verwendet zur Verwaltung und Speicherung der Artefakte der Dienstmodellmethodik, d.h. der Daten zur Repräsentation der jeweiligen Instanz des Dienstmodells (UML-Diagramme, Plaintext, etc.).

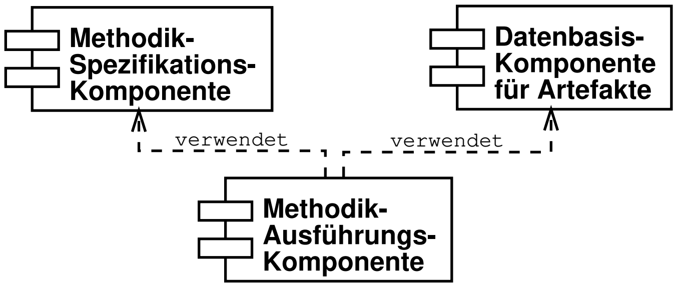

Abbildung 3.9: erster Entwurf der Architektur für das zu entwickelnde Werkzeug

Die Trennung in Methodik-Spezifikations- und Methodik-Ausführungs-Komponente soll hierbei eine möglichst einfache Anpassung der realisierten Methodik innerhalb des Werkzeugs ermöglichen: Änderungen der Methodik erfordern innerhalb des Werkzeugs nur eine Anpassung der Methodik-Spezifikation. Die Methodik-Ausführungs-Komponente verwendet zur Abarbeitung der geänderten Methodik automatisch die geänderte Methodik-Spezifikation, ohne selbst angepasst werden zu müssen.

Für das Werkzeuges kommen somit zwei grundsätzlich unterschiedliche Arten von Daten vor (siehe Abb. 3.10):

- **Methodik-Spezifikationsdaten:**
  Daten zur Spezifikation der Methodik aus ihren Workflows und Aktivitäten; Diese werden von der Methodik-Spezifikations-Komponente bereitgestellt und von der Methodik-Ausführungs-Komponente während der Abarbeitung der Methodik verwendet.

- **Workflow-Verwaltungsdaten:**
  Daten zur Verwaltung von Workflows, die derzeit abgearbeitet werden: Hiermit wird einerseits verwaltet, welche Workflows zur Zeit ausgeführt werden. Andererseits wird für diese zur Zeit ausgeführten Workflows der aktuelle Ausführungsstatus gespeichert.

- **MNM-Dienstmethodik-Artefaktdaten:**
  Daten, die die Artefakte der Dienstmethodik, d.h. die jeweilige Instanz des Dienstmodell selbst, darstellen; Sie werden von der Datenbasis-Komponente verwaltet und während der Abarbeitung der Methodik durch die Methodik-Ausführungs-Komponente genutzt.

Aus dieser ersten Architektur lassen sich die Anforderungen an das zu entwickelnde bzw. an das als Grundlage dienende Werkzeug vollständig identifizieren.

Zusätzlich zu den Anforderungen, die sich direkt aus der Anwendungsmethodik ableiten lassen, d.h. vor allem die Unterstützung von UML-Diagrammen und einer workfloworientierter Verarbeitung, (siehe Abschnitt 3.2) und der Anforderung, alle in dem vorangegangenen Abschnitt behandelten Benutzerinteraktionen zu unterstützen, lassen sich weiter die folgenden Anforderungen ableiten:

- Allgemein: leichte Anpassbarkeit und Erweiterbarkeit:
  Die obige Architektur aus ihren drei Komponenten muss unterstützt werden. Vor allen die Komponenten zur Spezifikation und Abarbeitung der Methodik müssen dabei realisierbar sein.

- Einfacher, einheitlicher Zugriff auf die Modellierungsdaten:
  Da das Werkzeug für das Dienstmodell unterschiedlichste Daten (strukturierende Daten, UML, Freitext, etc.) verwaltet, wäre eine möglichst einheitliche Schnittstelle für den Zugriff auf diese Daten

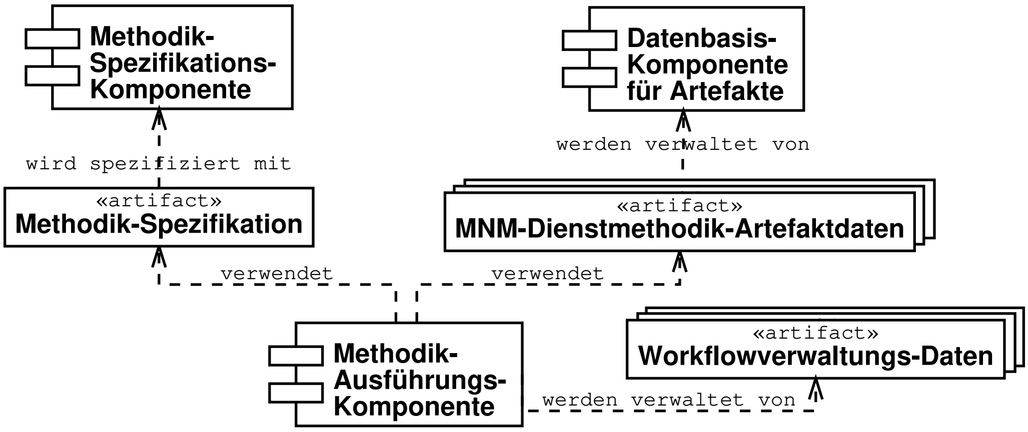

Abbildung 3.10: erster Entwurf der Architektur für das zu entwickelnde Werkzeug mit Artefakten

sowohl innerhalb des Werkzeuges sowie auch nach außen sinnvoll. Eine solcher, einheitlicher Zugriff wäre z.B. durch eine Skriptsprache möglich.

- Möglichkeiten zum Constraint-Checking, um die Konsistenz der Modellierungsdaten zu überprüfen: Da im Dienstmodell eine Reihe von bestimmten Beziehungen innerhalb der Daten besteht (z.B. Sub-Client gehört zu bestimmten Sub-Dienst, Prozess ist bestimmter Prozessklasse zugeordnet, etc.) ist zur Unterstützung des Benutzers bei der Dienstmodellierung eine automatische Konsistenzüberprüfung der Daten wichtig.

- Automatisierung, vor allem das Kopieren oder Verschieben von Modellierungsdaten innerhalb der Modellierung:
  Ebenso müssen vor allem für die Unterstützung der Workflows der Modellierungsmethodik Kopier- bzw. Verschiebeaktionen automatisierbar sein. Dies ist vor allem notwendig für Artefakte, die von der Dienstsicht in die Realisierungssicht (oder umgekehrt) übernommen werden, z.B. Prozessklassen und Prozesse (inkl. der sie beschreibenden UML-Use-Case- und Aktivitätsdiagramme). Denn für die Dienst- und Realisierungssicht sollten jeweils eigene Versionen dieser Artefakte existieren, da eine erstellte Instanz des Dienstmodells auch beide Sichten getrennt enthalten soll (z.B. nicht nur die erweiterten Aktivitätsdiagramme der Realisierungssicht für den Dienstleister, sondern auch die Aktivitätsdiagramme der Dienstsicht für die gemeinsame Sichtweise von Dienstnehmer und Dienstleister).

- Gute Produktunterstützung:
  Es muss eine gute Dokumentation, Unterstützung durch Newsgroups, o.ä. für das zur Entwicklung verwendete bzw. erweiterte OO/CASE-Werkzeug vorhanden sein.

3.5 Zusammenfassung

In diesem Kapitel wurden zunächst die Anwendungsmethodik des MNM-Dienstmodells selbst sowie die für sie notwendigen Benutzerinteraktionen analysiert. Hieraus wurde ein erster Entwurf der Architektur für das zu entwickelnde Werkzeug entworfen. Von dieser wurden Anforderungen abgeleitet, die einerseits für die Auswahl des als Grundlage dienenden OO/CASE-Werkzeugs und andererseits für die Konzeption des zu entwickelnden Werkzeugs selbst verwendet werden. Im folgenden Kapitel 4 wird die Auswahl des verwendeten OO/CASE-Werkzeugs und in Kapitel 5 der Entwurf des Werkzeugs behandelt.

Kapitel 4

Auswahl der verwendeten Software

Das zu entwickelnde Werkzeug wird durch die Erweiterung eines vorhandenen OO/CASE-Werkzeuges entwickelt. Daher werden zunächst eine Reihe verschiedener auf dem Markt befindlicher CASE-Werkzeuge betrachtet, um unter diesen das Werkzeug zu finden, das sich am besten als Grundlage für das zu entwickelnde Werkzeug eignet.

In Abschnitt 4.1 wird zunächst der für die Bewertung der Produkte verwendete Anforderungskatalog vorgestellt. Auf die einzelnen Produkte wird dann in Abschnitt 4.2 eingegangen. Die letztendliche Bewertung und Auswahl wird dann in Abschnitt 4.3 durchgeführt.

4.1 Anforderungskatalog

Zur Bewertung der verschiendenen, betrachteten OO/CASE-Werkzeuge werden jeweils die im vorhergehenden Kapitel identifizierten Anforderungen an das zu entwickelnde Werkzeug betrachtet:

- **(A1) Unterstützung von UML-Konzepten:**
  Die Erstellung, die Darstellung, die Veränderung, die Verwaltung und der Zugriff für alle notwendigen UML-Diagrammtypen (Klassen-, Use-Case-, Sequenz- und Kollaborationsdiagramme) muss unterstützt werden — in einer Form, die leicht verwaltet und weiterverarbeitet werden kann. Reine Bildinformationen, z.B. reichen nicht aus, da diese nur der Darstellung für den Benutzer dienen. Das heißt die verwendeten Datenstrukturen müssen genügend semantische Informationen enthalten, um sie weiterzuverwenden.

- **(A2) Gute Erweiterbarkeit:**
  Das vorhandene Werkzeug muss in vielerlei Hinsicht erweiterbar sein:

  - Insgesamt soll vor allem die in Abschnitt 3.4 beschriebene Architektur aus Methodik-Spezifikations-Komponente, Methodik-Ausführungs-Komponente und Datenbasis-Komponente realisierbar sein.

  - Eine komplette Unterstützung der Methodik, — sowohl deren Spezifikation als auch deren Ausführung — muss realisierbar sein. Dies schließt alle in Abschnitt 3.3 identifizierten Benutzerinteraktionen ein.

  - Alle Artefakte der Methodik, d.h. auch die nicht-UML-Artefakte, wie z.B. Daten zur Strukturierung, aber auch die Elemente des Dienstmodelles selbst (z.B. QoS-Parameter etc.) müssen spezifiziert und verwaltet werden können. Hierbei sollten Referenzen zu Dokumenten oder Standards gemacht werden können.

- Die UML-Unterstützung muss zusätzliche Erweiterungsmöglichkeiten (Stereotypen oder ähnliches) bieten, um die Elemente des Dienstmodells in den UML-Diagrammen passend zu repräsentieren, soweit dies notwendig ist.

Um all diese Erweiterungsmöglichkeiten — vor allem bzgl. des Workflows sowie der Modellierung der Dienstmodellelemente selbst — zu unterstützen, bieten sich z.B. die Konzepte der domänenorientierten Metamodellierung (siehe [DomainModeling]) an.

- **(A3) Konsistenzprüfung, Automatisierung und einheitlicher Datenzugriff:**
  Um den Benutzer bei der Modellierung zu unterstützen, sollte die Konsistenz der verwalteten Daten — vor allem der Elemente des Dienstmodelles selbst und die Beziehungen untereinander — automatisch vom Werkzeug überprüft werden. Auch sollten Kopier- und Verschiebevorgänge innerhalb der Modellierung automatisch ablaufen (z.B. Übernahme von Prozessen und Prozessklassen inkl. beschreibende UML-Diagramme zwischen Dienst- und Realisierungssicht). Eine einheitliche Schnittstelle für den Zugriff auf die verwalteten Daten innerhalb des Werkzeuges könnte die Realisierung dieser Anforderungen erleichtern. Für den externen Zugriff wäre ebenfalls eine Schnittstelle, — wenn möglich die gleiche wie intern — sinnvoll, damit die Daten aus dem Werkzeug exportiert werden können.

- **(A4) Gute, vorhandene Produktunterstützung:**
  Es sollte eine gute Dokumentation bzw. Unterstützung durch Newsgroups, Mailinglisten o.ä. existieren.

4.2 Vorstellung der betrachteten CASE-Werkzeuge

Im Weiteren werden nun die betrachteten CASE-Werkzeuge sowie die Erfüllung der gestellten Anforderungen im Einzelnen vorgestellt.

Unter diesen gibt es sowohl kommerzielle Produkte, als auch Open-Source-Entwicklungen. Die letzteren haben wegen des bekannten, offenliegenden Quellcodes den Vorteil, dass nicht erfüllte Anforderungen notfalls durch Erweiterung des Source-Codes kompensiert werden könnten, was jedoch einen hohen Aufwand bedeuten würde. Die Anforderungen – vor allem die der Erweiterbarkeit – werden bei der Bewertung dieser Produkte daher nur nach den jeweils vorgesehenen Möglichkeiten und nicht nach der direkten Erweiterung im Source-Code betrachtet.

4.2.1 Microsoft Visio, Version 2002

Visio (Homepage `http://www.microsoft.com/office/visio/default.asp`, siehe [visio]) ist ein bekanntes Programm zur Erstellung von Diagrammen verschiedenster Typen. Es wird von Microsoft entwickelt und ist nur für die verschiedenen Windows-Plattformen verfügbar. Es gibt *Visio* in drei verschiedenen Ausgaben:

- *Business Edition*: Ist vor allem für Geschäftsleute gedacht: Diese Edition enthält z.B. Diagrammtypen für Organisations-Modellierung und Office-Zwecke.
- *Professional Edition*: Ist eine Erweiterung der Business Edition, die vor allem für Software-Entwickler konzipiert ist. Hierin sind alle vor allem alle UML-Diagrammtypen enthalten.
- *Enterprise Edition*: Hierbei handelt es sich um eine spezielle Erweiterung der Professional Edition, die u.a für die Netzwerk-Modellierung geeignet ist.

Visio bietet viele Erweiterungen und Schnittstellen (OLE) zu anderen Programmen an. Auch Code-Generierung für Visual Basic, Java und C++ wird unterstützt. Erweiterbar ist Visio durch Visual Basic-Makros (VBA= Visual Basic for Applications, siehe [vba]) in den Diagrammdokumenten selbst oder durch

Standalone-Executables, COM (= Component Object Model)-Addins oder DLL's (sogenannte VSL's = Visio Solution Libraries) mit Hilfe von C++ oder sogar Delphi (siehe [delphi]).

Von den gestellten Anforderungen erfüllt *Visio* vor allem die UML-Unterstützung (A1) sehr gut. Es ist auch — z.B. mit seiner Skriptsprache Visual Basic — vielseitig erweiterbar (A2) und bietet damit auch Mechanismen für Konsistenzprüfung (A3).

4.2.2 Dia, Version 0.88.1

Dia (Homepage `http://www.lysator.liu.se/~alla/dia/`, siehe [dia]) ist ein Open-Source-Projekt in Anlehnung an *Visio*. Geschrieben ist es in C. Es ist ein reines Diagrammerstellungs-Programm. Im Prinzip ist es nur eine Art Vektor-Zeichenprogramm — ähnlich wie *xfig* (siehe [xfig]) —, das spezielle Unterstützung für bestimmte Diagrammtypen, u.a. UML-Diagramme, enthält. Dia bietet hierfür die Möglichkeit, die Knoten- und Kantentypen der unterstützten Diagrammtypen in Dokumenten zu verwenden. Assoziationen und andere Kanten-Verbindungen zwischen gleichen/verschiedenen Diagrammelementen sind möglich und nicht an die Position der Elemente gebunden, d.h. werden bei Verschiebung der Elemente mitverschoben. Dabei wird jedoch keinerlei Typüberprüfung durchgeführt, d.h. Diagrammelemente verschiedener Modelle sind miteinander verbindbar.

Als internes Dateiformat für Diagramme verwendet *Dia* XML (siehe [xml]). *Dia* erlaubt Code-Generierung für C++ und Java mit Hilfe des eigenständigen Tools *dia2code* (siehe [dia2code]). *Dia* ist durch Plugins — vor allem für Ein- und Ausgabe-Filter oder die Unterstützung neuer Diagrammtypen — erweiterbar.

In der derzeitigen Version werden bei UML-Aktivitätsdiagrammen keine Swimlanes unterstützt.

Insgesamt betrachtet, sind die Informationen in den Diagrammen sehr an die Darstellung gebunden bzw. in erster Linie für die Darstellung gedacht. Die vorhandenen Erweiterungsmöglichkeiten sind nicht geeignet für eine Workflowunterstützung oder die Modellierung aller Elemente des Dienstmodells und der Beziehungen zwischen diesen. Auch Möglichkeiten für Konsistenzprüfung oder Automatisierung sind keine vorhanden. Damit wird die Anforderung (A1) erfüllt, die Anforderungen (A2) und (A3) aber nicht.

4.2.3 Toolkit for Conceptual Modeling (TCM), Version 2.0.1

TCM (Homepage `http://wwwhome.cs.utwente.nl/~tcm/`, siehe [tcm]) ist ein Werkzeug zur Modellierung bzw. Erstellung von verschiedenen Diagrammtypen, u.a. der meisten UML-Diagrammtypen. Es ist ein Open-Source-Projekt. Hierbei sind alle Diagrammtypen jeweils einzeln in C++ geschrieben. Für jeden Diagrammtyp wird ein gewisses Type Checking durchgeführt.

Derzeit sind nicht alle UML-Diagrammtypen unterstützt, genauer gesagt keine Sequenz- und keine Kollaborationsdiagramme. Hiermit wird die Anforderung (A1) nicht erfüllt.

TCM bietet selbst keine Möglichkeit, Informationen verschiedener Diagramme miteinander zu verbinden. Erweiterungsmöglichkeiten, vor allem Workflowintegration, sind (außer der direkten Erweiterung im Source-Code) nicht vorhanden. Damit sind auch die Anforderungen (A2) und (A3) nicht erfüllt.

4.2.4 Rational Rose, Version 2002

Rational Rose (Homepage siehe `http://www.rational.com/products/rose/index.jsp`, siehe [rrose]) ist eine grafische OO-Modellierung- und Entwicklungsumgebung für UML bzw. die verwandten Notationen Boochs und OMT. Entwickelt wird es von der Firma Rational Software Corporation.

Erhältlich ist *Rational Rose* für die Plattformen Windows, Solaris, HP-UX, Aix, Irix, Linux. Für Windows bzw. Linux ist eine Evaluations-Version verfügbar.

Alle UML-Diagrammtypen werden voll unterstützt. Code-Generierung, Reverse- und Round-Trip-Engineering für Java, C, C++, Corba IDL (siehe [CORBA 2.4]), Ada, (Smalltalk, Visual Basic) sind möglich. In der aktuellsten Version ist auch eine Integration mit Version Control-Werkzeugen für Team-Unterstützung enthalten.

Erweiterbar ist *Rational Rose* durch die interne, eigene Skriptsprache (Rose Scripting) oder durch sogenannte Addins (Erweiterungs-Libraries). Hiermit ist Anpassung bzw. Steuerung der Oberfläche sowie Zugriff bzw. Modifikation des zu erstellenden Modells möglich.

Damit erfüllt *Rational Rose* alle Anforderungen, denn es hat einen sehr gute UML-Unterstützung (A1), ist erweiterbar (A2) und hat mit seiner Skriptsprache eine einheitliche Datenschnittstelle für Konsistenzprüfung und Automatisierung (A3).

4.2.5 Aonix Software Through Pictures (STP), Millennium Edition

Von der Firma Aonix stammt das Programm *STP* (*Software Through Pictures*, Homepage `http://www.aonix.com/content/products/stp/uml.html`, siehe [stp]). Es ist eine grafische objektorientierte Modellierungsumgebung zur automatisierten Analyse, Entwurf und Implementierung von OO-Anwendungen. *STP* ist für die Architekturen Windows NT/2000, Solaris, HP-UX, Aix und Irix erhältlich.

Eine wesentliche Eigenschaft ist die Multi-User-Unterstützung, d.h. die Unterstützung von shared Repositories und anderen Synchronisationsmechanismen. *STP* unterstützt den gesamten Lebenszyklus objektorientierter Anwendungsentwicklung. Es erlaubt das Erfassen der gesamten zu entwickelnden Anwendung in UML. Hierbei bietet es eine sehr umfassende Unterstützung aller UML-Diagrammtypen. Weiterhin sind Syntax- und Semantik-Checking der erfassten Informationen möglich. Die Dokumentation wird während der Entwicklung automatisch erstellt.

Code-Generierung und Reverse/Round-Trip-Engineering von C++, Java, ADA und Corba IDL (siehe [CORBA 2.4]), Visual Basic werden unterstützt. Diese ist durch ACD (= Architecture Component Development zur Festlegung der Abbildung des abstrakten Modells auf eine konkrete Programmiersprache) erweiterbar.

Allgemein erweiterbar ist *STP* durch die interne, als Skriptsprache dienende Query- und Report-Sprache QRL. *STP* unterstützt das OMG-Austauschformat XMI (siehe [OMG 98-09-03]).

Ähnlich wie *Rational Rose* erfüllt *STP* alle gestellten Anforderungen: die notwendige UML-Unterstützung (A1), die Erweiterbarkeit (A2) und die einheitliche Datenschnittstelle bzw. Mechanismen zur Konsistenzprüfung (A3).

4.2.6 uml, Version 1.0.3-1

Das open-source-Projekt namens *uml* (Homepage `http://uml.sourceforge.net/`, siehe [umlprog]) hat ein UML-Modellierungswerkzeug entwickelt. Dabei ist die grafische Benutzeroberfläche der von *Rational Rose* nachempfunden. Geschrieben ist uml in C++ für *KDE*. Unterstützt werden Klassen-, Use-Case-, Sequenz- und Kollaborationsdiagramme, aber keine Aktivitätsdiagramme.

Die UML-Unterstützung ist daher nicht ausreichend für das Werkzeug, d.h. Anforderung (A1) nicht erfüllt. Außerdem fehlen jegliche Erweiterungsmöglichkeiten und Mechanismen zur Konsistenzprüfung. Also sind auch die Anforderungen (A2) und (A3) nicht erfüllt.

4.2.7 ArgoUML, Version 0.9.8

ArgoUML (Homepage `http://argouml.tigris.org/`, siehe [argouml]) ist ein Open-Source-Projekt, geschrieben in Java, zur Entwicklung eines UML-Modellierungswerkzeuges. Es basiert u.a. auf der open-source-Library *nsuml* von Novasoft (siehe [nsuml]).

Es ist projektorientiert, d.h. erlaubt eine übergreifende Modellierung mit mehreren Diagrammen. Dabei werden alle wesentlichen UML-Diagrammtypen unterstützt. Weitere Eigenschaften sind Typüberprüfung und ein Ratgeber-Modus bei der Diagrammerstellung. Viele semantische Informationen über die Diagrammelemente sind erfassbar. Code-Generierung für Java wird unterstützt.

ArgoUML kann durch Module erweitert werden und unterstützt das OMG-Austauschformates XMI (siehe [OMG 98-09-03]).

Die Anforderung der UML-Unterstützung (A1) ist erfüllt. Erweiterungen sind mit Modulen möglich. Somit ist Anforderung (A2) erfüllt. Mit der Typüberprüfung und dem Ratgeber-Modus ist die Anforderungen (A3) weitgehend erfüllt, wenn gleich auch eine einheitliche, interne Datenschnittstelle fehlt.

4.2.8 kuml, Version 0.5.1

Ähnlich wie *ArgoUML* ist das Programm *kuml* (Homepage `http://www.informatik.fh-hamburg.de/~kuml/`, siehe [kuml]) ein Werkzeug zum Erstellen von UML-Diagrammen bzw. zur UML-Modellierung. Es ist ebenfalls open-source und in C++ geschrieben. Code-Generierung für Java bzw. C++ ist geplant.

Derzeit werden aber nicht alle UML-Diagrammtypen unterstützt: nur Klassen- und Use-Case-Diagramme. Es ist bei weitem nicht so weit entwickelt wie *ArgoUML*. Hiermit ist Anforderung (A1) nicht erfüllt.

Erweiterungsmöglichkeiten und Konsistenzprüfungsmechanismen existieren noch keine, daher werden die Anforderungen (A2) und (A3) nicht erfüllt.

4.2.9 gaphor, Version 0.0.1

Ein weiteres Open-Source-Projekt zur Entwicklung einer UML-Modellierungsumgebung ist *gaphor* (Homepage `http://gaphor.sourceforge.net/`, siehe [gaphor]). Entwickelt wird es in Python und C, basierend auf *gnome2* und *diacanvas2-Library* (siehe [diacanvas]). Dies soll vor allem eine hohe Erweiterbarkeit ermöglichen. Das Projekt befindet sich jedoch noch im Anfangsstadium und ist daher als Grundlage für das Werkzeug nicht zu verwenden.

4.2.10 ObjectDomain DomainR3

Von der Firma ObjectDomain gibt es das Programm *DomainR3* (Homepage siehe `http://objectdomain.com/domain/`, siehe [domainr3]). Es ist eine in Java geschriebene UML-Modellierungs- und Entwicklungsumgebung . Unterstützt werden sämtliche (z.B. auch statische Objektdiagramme) UML-Diagrammtypen. Es hat eine Python-Schnittstelle (siehe [Python]) für Zugriff auf Modell und Diagramme.

Die Anforderungen erfüllt *DomainR3* alle: Die UML-Unterstützung ist sehr gut, Anforderung (A1) also erfüllt. Mit seiner Schnittstelle für Python ist *DomainR3* gut erweiterbar (A2) und bietet eine einheitliche Zugriffsmöglichkeiten und Konsistenzprüfungsmechanismen (A3).

4.2.11 Honeywell Domain Modelling Environment (DoME), Version 5.3i

DoME (Homepage `http://www.htc.honeywell.com/dome/`, siehe [domehome], [dome53]) ist ein Werkzeug zum Modellieren in verschiedenen Diagrammnotationen, d.h. zum Erstellen verschiedener Diagrammtypen. Genauer gesagt ist es ein Meta-Modeling-Werkzeug, beim die Definition eines Modells bzw. Diagrammtyps, mit dem Werkzeug selbst entwickelt werden kann. *DoME* wurde von der Firma Honeywell entwickelt und ist open-source. Geschrieben ist es in Smalltalk (siehe [OMG 99-07-65], genauer Smalltalk für *VisualWorks 5i.4* von Cincom, siehe [vwsupport], [cincom1])

Unterstützt werden bereits eine Reihe verschiedenster Typen von Diagrammen, wie z.B. UML-, Datenfluss-, Zustandsübergangsdiagramme oder Petrinetze. Alle für das Dienstmodell notwendigen UML-Diagrammtypen werden unterstützt, insbesondere können sie weiterentwickelt werden. Das heißt die Anforderung der UML-Unterstützung (A1) wird erfüllt.

Weitere Diagramm- bzw. Modelltypen können entworfen werden. Hiermit sind Modelltypen für Workflowspezifikation, Workflowausführung und Modelltypen für die Dienstmodell-Daten selbst entwickelbar, d.h. die Komponenten der in 3.4 entwickelten Architektur sind mit *DoME* alle realisierbar. Damit wird die Anforderung (A2) der Erweiterbarkeit unter allen betrachteten Produkten am besten erfüllt.

Die Scheme-artige Skriptsprache Alter erlaubt Zugang und Transformation (auch einfache Dokument- bzw. Code-Generierung) der in den Diagrammen enthaltenen Daten. Damit sind Mechanismen für Konsistenzprüfung, sowie für Automatisierung von Kopier- bzw. Verschiebeabläufen vorhanden. Der Zugriff auf die verschiedenen Modelle bzw. Diagramme erfolgt bei Alter mit Hilfe einer einheitlichen Klassenstruktur und Operationsaufrufen dafür. Damit steht eine einheitliche Daten-Schnittstelle für die Modell- bzw. Diagrammdaten intern und extern zur Verfügung. Insgesamt ist damit auch die Anforderung (A3) sehr gut erfüllt.

Es ist außerdem eine umfassende Dokumentation des Systems selbst, der Skriptsprache und auch für *Visual Works* vorhanden. Damit wird auch die Anforderung (A4) erfüllt.

Damit erfüllt *DoME* alle Anforderungen. Insbesondere in Bezug auf die Erweiterbarkeit für eine klare Modellierung des Dienstmodells und einer in der Zukunft leicht anpassbaren Workflowsteuerung ist es mit seiner Meta-Modellierung allen anderen Produkten überlegen. Die erste, entworfene Architektur für das Werkzeug (siehe 3.4) ist mit *DoME* vollständig realisierbar. Es wurde daher als Grundlage für das in dieser Diplomarbeit zu entwickelnde Werkzeug ausgewählt.

4.3 Abschließende Bewertung

Hier sollen die betrachteten CASE-Werkzeuge und die von ihnen erfüllten Anforderungen miteinander verglichen werden. Hierzu sind in Tabelle 4.1 die CASE-Werkzeuge mit den jeweils erfüllten oder nicht erfüllten Anforderungen aufgelistet. Die Bewertung verwendet dabei drei Stufen:

- -: nicht erfüllt
- +: erfüllt
- ++ : gut erfüllt

Die kommerzielen Produkte *Visio*, *Rational Rose* und *STP* weisen eine gute UML-Unterstützung und viele Erweiterungsmöglichkeiten und einheitliche Schnittstellen für den Datenzugriff auf. Meist wird hier eine Code-Generierung für verschiedene Programmiersprachen unterstützt, die jedoch für das zu entwickelnde Werkzeug nicht benötigt wird.

Fast alle Open-Source-Produkte (bis auf *dia*, *ArgoUML* und *DoME*) sind dagegen bezüglich der UML-Unterstützung unvollständig bzw. haben in der Regel keine ausreichenden Erweiterungsmöglichkeiten vorgesehen. Es fehlt jeweils eine einheitliche Schnittstelle. Auch sind keine Konsistenzprüfungsmechanismen vorhanden, jedenfalls keine bei denen die geprüften Bedingungen frei definierbar wären, d.h. um z.B. eine Überprüfung der Beziehungen zwischen den Dienstmodellelementen zu gestatten.

Ganz anderst verhält es sich mit *DoME*: Es erfüllt die gestellten Anforderungen, vor allem die der Erweiterbarkeit, Konsistenzprüfung und Automatisierung, am besten:

Produkt:	(A1) UML-Unterstützung:	(A2) Erweiterungs-möglichkeiten:	(A3) einheitlicher Zugriff, Konsistenz, Automatisierung:
kommerziell:			
Visio	++	+	+
Rational Rose	++	+	+
STP	++	+	+
DomainR3	++	+	+
open-source:			
Dia	+	-	-
TCM	-	-	-
uml	-	-	-
ArgoUML	+	+	+
kuml	-	-	-
DoME	+	++	++

Tabelle 4.1: Vergleich der CASE-Werkzeuge bezüglich der erfüllten Anforderungen

- **UML-Unterstützung:**
  Alle notwendigen UML-Diagrammtypen werden unterstützt. Auch ist eine praktisch beliebige Anpassung (z.B. Hinzufügen neuer, spezieller Attribute, etc.) möglich.
- **Erweiterbarkeit:**
  Praktische beliebige Modelle bzw. Diagramme werden Dank der Metamodellierung unterstützt. Hiermit lassen sich alle Elemente des Dienstmodells und die speziellen Beziehungen vollständig modellieren. Spezifikationen als Freitext, Referenzen zu Dokumenten oder Standards können in eigenen speziellen Modelltypen realisiert werden. Die in 3.4 entworfenen Architektur-Komponenten lassen sich jeweils alle mit eigenen *DoME*-Modelltypen realisieren. Damit ist insbesondere eine auch in der Zukunft leicht anpassbare bzw. erweiterbare Workflowsteuerung inkl. Artefaktverwaltung realisierbar.
- **Einheitliche interne/externe Schnittstelle u.a. für Konsistenzprüfung und Automatisierung:**
  Mit der Skriptsprache Alter kann auf alle Daten, sowohl die Workflow-Steuerungsdaten als auch die Artefakte inkl. UML-Diagramme und andere Spezifikationen (Freitext, Standard-Referenzen, etc.) einheitlich innerhalb des Werkzeuges wie auch extern zugegriffen werden. Alter ermöglicht insbesondere auch Konsistenzprüfungen und Automatisierung, z.B. von Kopiervorgängen während der Modellierung.
- **Guter Support:**
  Es steht eine Dokumentation des *DoME*-Systems selbst, der Skriptsprache Alter und auch für *Visual Works* zu Verfügung.

4.4 Zusammenfassung

In diesem Kapitel wurden die bestehenden OO/CASE-Werkzeuge, die als Grundlage für das zu entwickelnde Werkzeug in Frage kamen, betrachtet. Die Wahl fiel auf *DoME*. Im Folgenden wird *DoME* als Basis zur Entwicklung und prototypischen Implementierung des Werkzeug verwendet. Das folgende Kapitel 5 geht genauere auf den Entwurf des Werkzeuges mit *DoME* als Grundlage ein. Die prototypische Implementierung wird dann anschließend in Kapitel 6 behandelt.

Kapitel 5

Entwurf des Werkzeugs

Dieses Kapitel ist dem Entwurf des entwickelten Werkzeugs gewidmet. Alle wesentlichen, für das Werkzeug getroffenen Designentscheidungen und ihre Gründe werden hier besprochen.

Zunächst wird in Abschnitt 5.1 die Metamodellierungs-Umgebung *DoME*, die als Grundlage für das Werkzeug dient, genauer vorgestellt. Daran anschließend wird in Abschnitt 5.2 ein Überblick über das Design des Werkzeugs aufbauend der in Abschnitt 3.4 entwickelten Architektur betrachtet. Nachfolgend werden die Aspekte des Designs einzeln behandelt. Zunächst wird in Abschnitt 5.3 das entworfene und mit *DoME* umgesetzte Workflowkonzept erklärt. Hierbei wird auf Spezifikation, Erstellung und Ausführung von Workflows für das Werkzeug allgemein eingegangen. Als nächstes wird — damit im Zusammenhang stehend — in Abschnitt 5.4 auf die Realisierung der Artefakte der Workflows, u.a. der UML-Diagrammtypen, eingegangen. Abschließend wird in Abschnitt 5.5 die Umsetzung der MNM-Dienstmodellierungsmethodik-Workflows selbst innerhalb von *DoME* mit den vorher entwickelten Mitteln beschrieben.

5.1 Grundlagen der Entwicklungs- und Ausführungsumgebung

Als Entwicklungs- und Ausführungsumgebung für das zu entwickelnde Werkzeug wurde die Metamodellierungs-Umgebung *DoME* gewählt (siehe Kapitel 4). Im Folgenden werden *DoME* und seine Konzepte näher vorgestellt. In diesem Zusammenhang sei auch auf den *DoME*-Guide ([DOMEGuide]) und die Alter-Manual ([AlterManual]) verwiesen. Beide Dokumentationen werden jeweils mit *DoME* mitgeliefert.

Zunächst werden Begriffe und Konzepte von *DoME* eingeführt und erläutert. Abschnitt 5.1.1 führt die grundlegenden Konzepte von *DoME* und die damit verbundenen Begriffe Modell, Modelltyp und Metamodell ein. In Abschnitt 5.1.2 wird der prinzipielle Aufbau von Modell und Metamodell eingeführt. Eine Übersicht über existierenden Modelltypen wird in Abschnitt 5.1.3 gegeben.

5.1.1 Das Grundkonzept: Modell, Modelltyp und Metamodell

DoME ist ein Werkzeug zum Erstellen und Verarbeiten von Modellen bzw. Graphen verschiedenster Typen, wobei die mögliche Struktur eines Modells durch ein Metamodell vorgegeben wird.

Grundelegendes Konzept ist das **Modell** oder **Graphmodell**. Jedes Modell ist ein Graph, der im Wesentlichen wie üblich aus Knoten und Kanten besteht und entsprechend dargestellt wird. In einem Modell in *DoME* sind aber verschiedene Typen von Knoten und Kanten, die sich in Darstellung und Verhalten und zusätzlichen Eigenschaften unterscheiden, möglich.

Jedes Modell gehört einem sogenannten ***DoME*-Modelltyp (*DoME* model type)** (oder kurz **Modelltyp**) an. Dieser legt vor allem die verschiedenen Knoten- und Kantentypen und die möglichen Beziehungen zwischen diesen innerhalb des Modells fest. So gibt es z.B. in *DoME* bereits einen Modelltypen für Petrinetze. Jedes Modell dieses Modelltyps repräsentiert ein Petrinetz-Diagramm. Gehört ein Modell einem gegebenen Modelltyp an, so wird es auch als eine **Instanz dieses Modelltyps** bezeichnet.

Jeder Modelltyp wird festgelegt mit einem sogenannten **Metamodell**. Ein Metamodell ist selbst ein Modell und wird damit auch durch einen Graphen beschrieben (näheres siehe Abschnitt 5.1.2). Das Metamodell gibt dabei aber nur den prinzipiellen Rahmen für Struktur und Verhalten eines Modells vor.

Um Struktur bzw. Verhalten dann vollständig anzugeben, ist zusätzlich Programm-Code — vor allem für die verschiedenen Typen von Knoten und Kanten — festzulegen. Hierbei gibt es zwei prinzipielle Methoden:

1. Zusätzliche Spezifikation der Knoten- und Kantentypen direkt als Smalltalk-Klasse: Jeder Knoten- und Kantentyp in einem Modelltyp ist hierbei eine Smalltalk-Klasse, deren grundlegende Struktur und Verhalten durch das Metamodell vorgegeben wird und durch weiteren Smalltalk-Code erweitert wird Auf diese Weise realisierte Modelltypen werden im Weiteren als **Smalltalk-basierte Modelltypen** bezeichnet. Diese ist die ursprüngliche der beiden Methoden. Sie ist mächtiger als die zweite, da hier die Vorgaben für das Modell durch *DoME* beliebig erweiterbar ist. Auch kann hier *DoME* direkt um weitere Features erweitert werden.

2. Zusätzliche Spezifikation der Knoten- und Kantentypen mit der Erweiterungssprache **Alter**: Alter ist ein objekt-orientierter Scheme-Dialekt (siehe [Scheme]), der als Teil von *DoME* selbst in Smalltalk implementiert ist. Zur Laufzeit wird der Alter-Code des Modelltyps, dem ein Modell angehört, interpretiert. Modelltypen, die so realisiert werden, heißen **protodome-basierte Modelltypen** oder kurz **Protodome-Modelltypen**.

   Protodome ist sozusagen selbst ein Modelltyp, dessen Klassen direkt in Smalltalk implementiert sind (nach erster Methode). Er ist aber nicht an ein bestimmtes Metamodell gebunden, sondern interpretiert sowohl ein vorgegebenes Metamodell als auch zugehörigen Alter-Code, wobei er Alter auf das interpretierte Metamodell bzw. auf das daraus erstellte Modell in Form von erstellten Alter-Klassen pro Kanten- bzw Knotentyp Zugriff erlaubt. Die erstellten Klassen können beliebig erweitert werden, um Darstellung und Verhalten des zugehörigen Knoten und Kanten näher zu bestimmen.

   Mit Alter kann auch auf Smalltalk-basierte Modelle zugegriffen werden, um diese zu erstellen, ändern (transformieren) oder in beliebiger Weise zu verarbeiten (z.B. Dokument/Code-Generierung). Die dann ebenfalls Alter zur Verfügung gestellten Knoten- bzw. Kantentypen sind jedoch nicht erweiterbar, da sie direkt in Smalltalk implementiert sind.

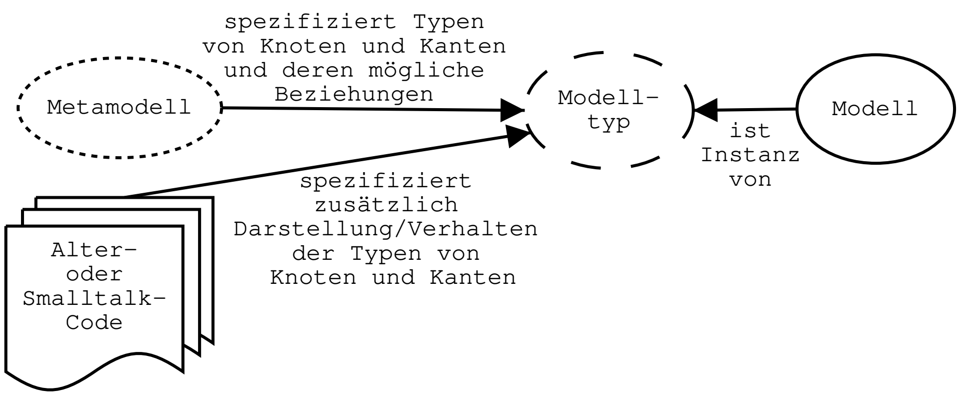

Abbildung 5.1: Zusammenhang von Metamodell, Modelltyp und Modell in *DoME*

Zusammengefasst betrachtet, gehört jedes Modell in *DoME* genau einem Modelltyp an, der die Typen von

Knoten und Kanten, deren Verhalten, sowie mögliche Beziehungen zwischen diesen, spezifiziert. Jeder Modelltyp wird dazu selbst durch ein Metamodell bestimmt, dessen genaues Verhalten durch zusätzlichen Smalltalk- oder Alter-Code näher spezifiziert werden kann (siehe Abb. 5.1).

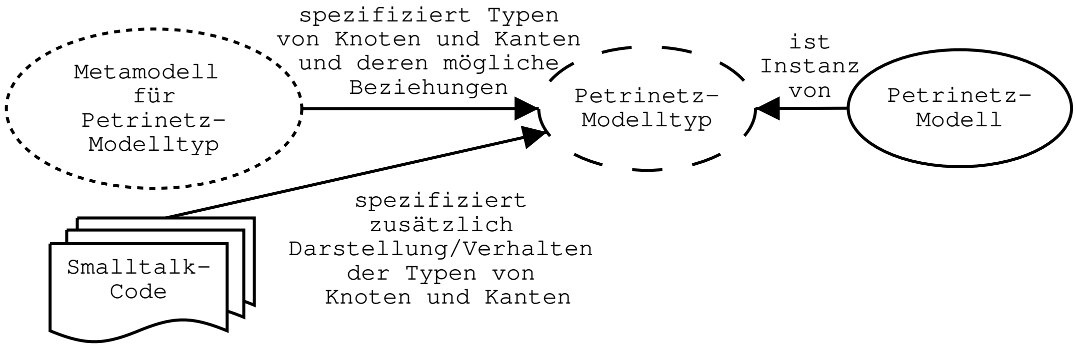

Abbildung 5.2: Der Petrinetz-Modelltyp als ein Beispiel für einen *DoME*-Modelltyp

Ein in *DoME* bereits vorhandener Modelltyp ist z.B. der Petrinetz-Modelltyp für Petrinetz-Diagramme (siehe Abb. 5.2). Er ist Smalltalk-basiert. Dieser Modelltyps definiert zwei Knotentypen, nämlich Plätze und Transitionen und einen einzigen Typ von Kante. Ein Modell dieses Typs enthält dann als Knoten Plätze und Transitionen, die (in der für Petrinetze üblichen Weise) durch Kanten verbunden werden können. Ein Beispiel für ein solches Petrinetz-Modell ist Abb. 5.3 zu sehen. (für das Metamodell des Petrinetz-Modelltyp siehe Abschnitt 5.1.2 und Abb. 5.4).

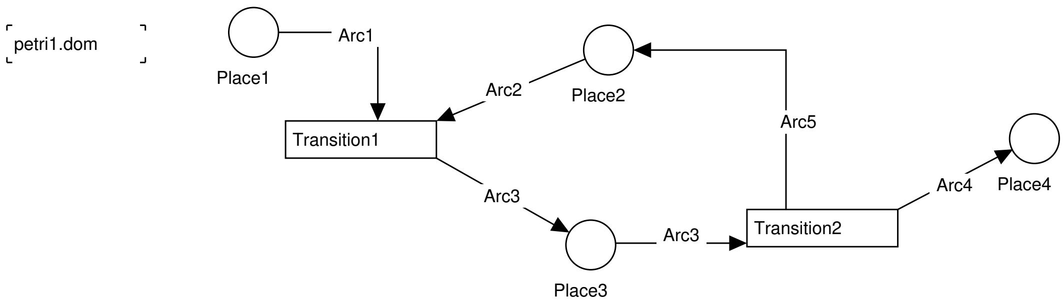

Abbildung 5.3: Beispiel für ein Modell des Petrinetz-Modelltyps

Für das zu entwickelnde Werkzeug wurde eine Reihe von Modelltypen entworfen und implementiert. Zur Realisierung dieser Modelltypen wurde Methode 2 gewählt (für Workflow und Artefakte wie UML-Diagramme etc.), da diese Methode eine wesentlich einfachere Anpassung bzw. Erweiterung des gesamten Werkzeugs im Vergleich zu Methode 1 ermöglicht. Das heißt, zur Entwicklung des Werkzeugs wurden Metamodelle spezifiziert und dafür notwendiger Alter-Code erstellt.

5.1.2 Der Aufbau von Modellen und Metamodellen

Jedes Modell in *DoME* gehört einem Modelltyp an, dessen Struktur mit einem Metamodell festgelegt wird (siehe vorigen Abschnitt). Modelle und Metamodelle sind Graphen, die im Wesentlichen aus Knoten und Kanten bestehen. Knoten und Kanten werden im Weiteren unter dem Begriff **Graphobjekt** zusammengefasst.

In einem Modell in *DoME* gehört jedes Graphobjekt jeweils einem bestimmten Typ an (ähnlich wie das Modell einem bestimmten Modelltyp angehört). Die verschiedenen Typen von Graphobjekten eines Modells werden in dessen Modelltyp bzw. genauer dessen Metamodell festgelegt. Und zwar wird jeder Typ eines Graphobjektes innerhalb des Metamodells durch einen eigenen Knoten, einem sogenannten Spezifikationsknoten, dargestellt bzw. spezifiziert. Zwischen den Spezifikationsknoten wird mit Kantenverbindungen festgelegt, welche Typen von Knoten in einer Instanz des spezifizierten Modelltyps über Kanten (und mit welchem Typ von Kanten) zueinander in Beziehung gesetzt werden können.

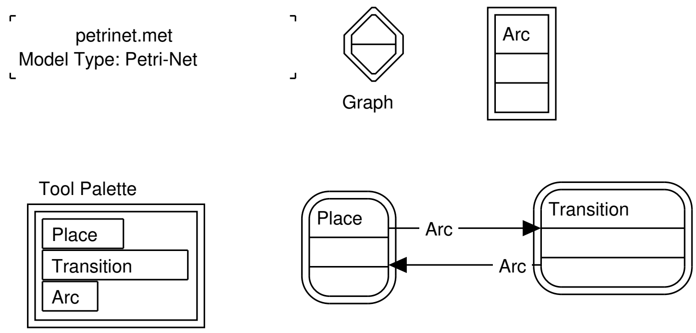

Abbildung 5.4: Metamodell des Petrinetz-Modelltyps

Als Beispiel ist in Abb. 5.4 das Metamodell des Petrinetz-Modelltyps dargestellt. In *DoME*-Metamodellen repräsentieren abgerundete Klassenboxen die Knotentypen und eckige Klassenboxen die Kantentypen. Im Fall des Petrinetz-Modelltyps gibt es nur die beiden Knotentypen Platz (place) und Transition (transition) und einen Kantentyp (arc). Der Modelltyp (auch Graphtyp) selbst wird durch ein rautenförmige Klassenbox dargestellt. Außerdem wird in einem Metamodell die Toolbar (Tool Palette) definiert, die bei der Erstellung bzw. Änderung eines Modells des definierten Modelltyps dem Benutzer zur Verfügung steht, festgelegt. Kanten zwischen den (runden) Klassenboxen für Knotentypen legen die möglichen Kantenverbindungen in einem Modell des Modelltyps zwischen den Knoten der repräsentierten Knotentypen fest.

Jedes Graphobjekt in *DoME* kann beliebig viele sogenannte **Submodelle** an sich gebunden haben. Submodelle sind selbst vollwertige Modelle (also Graphen). Auf diese Weise kann in *DoME* mit Hierarchien von Modellen bzw. Graphen modelliert werden.

Weiterhin kann jedes Graphobjekt weitere Attribute, sogenannte **Graphobjekt-Properties** (oder kurz **Properties**), besitzen. Properties sind Wertzuweisungen, die selbst nicht in der Graphdarstellung angezeigt werden. Sie können aber die Darstellung und das Verhalten der Graphobjekte zu denen sie gehören (über Smalltalk/Alter-Code) beeinflussen. Als Datentypen für Properties sind dabei Grundtypen wie z.B. Integer, String, Boolean sowie verschachtelte Strukturen wie z.B. Listen oder assoziative Arrays, aber auch Referenzen zu beliebigen anderen Modellen oder Graphobjekten möglich.

Sowohl der Modelltyp von Submodelle, als auch die Properties (der Name und Datentyp einer Property) eines Graphobjekttyps werden im Metamodell eines Modelltyps im Spezifikationsknoten des jeweiligen Graphobjekttyps festgelegt.

In Abb. 5.5 ist die allgemeine Struktur eines *DoME*-Modells und in Abb 5.6 ist die Struktur eines *DoME*-Metamodells jeweils in UML dargestellt. *DoME* kennt zusätzlich zu den bisher beschriebenen Konzepten noch weitere. Für detailliertere Informationen über die Struktur von Modellen bzw. Metamodellen in *DoME* siehe Anhang A.1.1 bzw. Anhang A.1.2.

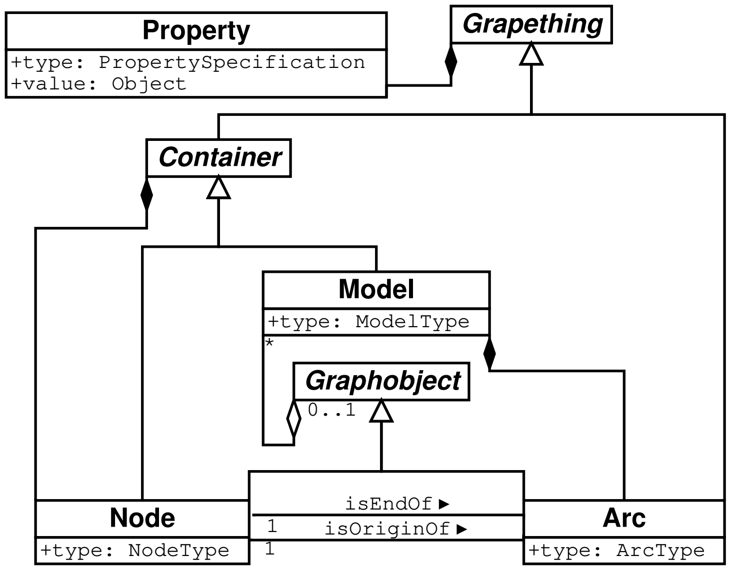

Abbildung 5.5: Struktur eines *DoME*-Modells dargestellt mit UML

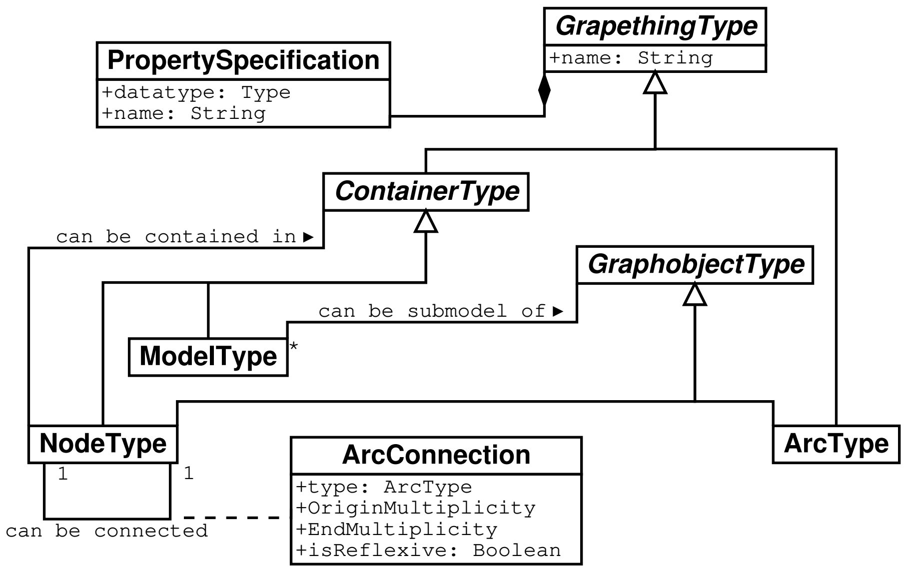

Abbildung 5.6: Struktur eines *DoME*-Metamodells dargestellt mit UML

5.1.3 Überblick über existierende Modelltypen

In *DoME* gibt es bereits eine Reihe von vorhanden Modelltypen. Die folgende Liste gibt eine kurz Übersicht über alle existierenden Modelltypen:

1. Smalltalk-basiert:
   - Verschiedene objekt-orientierte Diagrammnotationen: Colbert OOSD, Coad-Yourdon OOA und IDEF-0
   - Einfache Petrinetze, Datenfluss- und einfache Zustandsübergangsdiagramme
   - Eigener Diagrammtyp, um Alter-Code grafisch darzustellen bzw. zu erstellen (genannt Projektor)
   - Modelltyp für Metamodelle: Metadome
   - der generische Protodome-Modelltyp zur Umsetzung von protodome-basierten Modelltypen mittels Alter. Zusätzlich zu Protodome gibt es auch Cyberdome, bei dem die Spezifikation des Metamodells bei der Erstellung einer Instanz des Protodome-Modells selbst automatisch erweitert werden kann.
2. Protodome-basiert:
   - Verschiedene Beispiele für die Verwendung von Protodome, u.a. z.B. Protodome-Realisierung für bereits in Smalltalk direkt realisierte Modelle wie Petrinetze und Datenflussdiagramme
   - UML-Klassen, Use-Case-, Sequenz-, Kollaborations-, Zustands- und Aktivitätsdiagramme: hierbei fehlt jedoch meist die vollständige Realisierung des UML-Metamodells, daher wurde für das Werkzeug von den vorhandenen Metamodellen eine eigene Spezifikation für alle (zumindest alle für die Anwendungsmethodik des MNM-Dienstmodells notwendigen) UML-Diagrammtypen erstellt (siehe Abschnitt 5.4.2).
   - Entity-Relationship-Diagramme (nicht mit *DoME* selbst mitgeliefert, aber von der *DoME*-Webseite aus erhältlich)

5.2 Design des Werkzeugs

In diesem und den folgenden Abschnitten dieses Kapitels wird auf das Design des Werkzeugs aufbauend auf der in Abschnitt 3.4 beschriebenen Architektur innerhalb von *DoME* genauer eingegangen.

In diesem Abschnitt selbst wird zunächst ein grober Überblick über den Entwurf gegeben. Danach wird in den nachfolgenden Abschnitten auf die Aspekte des Entwurfs im Einzelnen eingegangen. Abschnitt 5.3 geht hierfür auf die Spezifikation und Abarbeitung von Workflows innerhalb des Werkzeugs allgemein ein. Abschnitt 5.4 bespricht die Repräsentation der Dienstmethodik-Artefakte innerhalb des Werkzeugs. Auf den Ergebnissen der Abschnitte 5.3 und 5.4 aufbauend wird in Abschnitt 5.5 die Spezifikation der MNM-Dienstmethodik selbst innerhalb des Werkzeugs beschrieben.

Das Werkzeug für die Anwendung der Methodik der MNM-Dienstmodellierung wurde innerhalb von *DoME* als eine Reihe von *DoME*-Modelltypen entworfen. Die in Abschnitt 3.4 entwickelte Architektur wurde hierbei vollständig umgesetzt. Diese Architektur für das Werkzeug sieht drei wesentliche Komponenten vor: eine **Methodik-Spezifikations-Komponente** zur Festlegung der Methodik mit all ihren Workflows, eine **Methodik-Ausführungs-Komponente** zur Abarbeitung der vorher spezifizierten Methodik und eine **Datenbasis-Komponente** für die Speicherung und Verwaltung der Dienstmodellierungs-Artefakte während der Abarbeitung der Methodik (vgl. Abb. 3.9). In der allgemeinen Architektur wurden bereits drei verschiedene Arten von Daten unterschieden: die **Methodik-Spezifikationsdaten** zur Festlegung der Methodik, die **Workflow-Verwaltungsdaten** zur Verwaltung der aktuell abgearbeiteten Workflows und die **Dienstmethodik-Artefaktdaten**, d.h. die Daten zur Repräsentation der jeweiligen Dienstmodell-Instanz selbst (vgl. Abb 3.10). Die Methodik-Spezifikationsdaten werden von der Methodik-Spezifikations-Komponente erstellt bzw. verwaltet. Die Workflow-Verwaltungsdaten der aktuell ausgeführten Workflows werden von der Methodik-Ausführungs-Komponente und die Dienstmethodik-Artefaktdaten von Datenbasis-Komponente verwaltet.

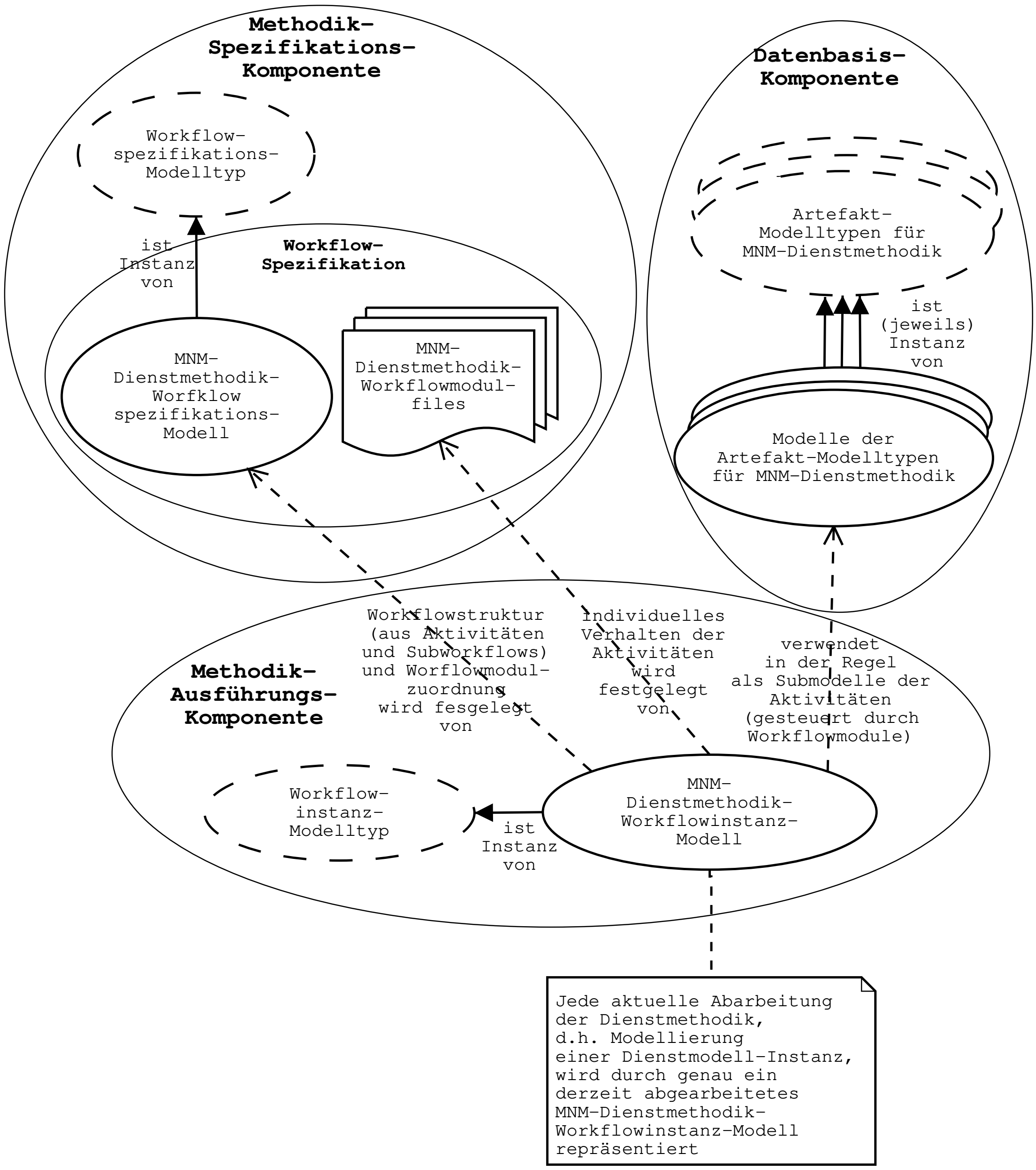

Abbildung 5.7: Skizze des Werkzeugsentwurfs mit *DoME*-Modelltypen und *DoME*-Modellen

Jede Architekturkomponente wird innerhalb von *DoME* durch einen oder mehrere Modelltypen realisiert. Die Daten, die von einer Komponente jeweils verwaltet werden, sind hierbei weitgehend in Modellen der zugehörigen Modelltypen enthalten. Das heißt jede Komponente wird durch einen (oder mehrere) Modelltypen realisiert, ihre zugehörigen Daten sind in den Modellen ihres (oder ihrer) Modelltypen enthalten. Es wurden für die Methodik-Spezifikations-Komponente der **Workflowspezifikations-Modelltyp**, für die Methodik-Ausführungs-Komponente der **Workflowinstanzen-Modelltyp** und für die Datenbasis-

Komponente mehrere Modelltypen, die unter dem Begriff **Artefakt-Modelltypen** zusammengefasst werden, entworfen.

Die zu realisierenden, in der MNM-Dienstmodellierungsmethodik spezifizierten Workflows, im Folgenden kurz als **Methodik-Workflows** bezeichnet, müssen in der Zukunft möglicherweise an weitere Entwicklungen angepasst werden. Die Modelltypen für Methodik-Spezifikation und Methodik-Ausführung, d.h. der Workflowspezifikations-Modelltyp und der Workflowinstanzen-Modelltyp wurden daher sehr flexibel gestaltet: Mit dem ersteren können praktisch beliebige, rekursiv verschachtelte Workflows spezifiziert werden. Jedes Modell des Workflowspezifikations-Modelltyps spezifiziert einen rekursiv verschachtelten Workflow. Insbesondere für die MNM-Dienstmodellierungsmethodik und alle enthaltenen Methodik-Workflows wurde ein einziges Modell dieses Modelltyps entworfen, das die gesamte Methodik spezifiziert. Dieses wird im weiteren als **MNM-Dienstmethodik-Workflowspezifikations-Modell** bezeichnet.

Der Workflowinstanz-Modelltyp dagegen dient zur Abarbeitung von Workflows, die mit dem Workflowspezifikations-Modelltyp spezifiziert wurden. Genauer gesagt werden Modelle des Workflowinstanz-Modelltyps aus einem Modell des Workflowspezifikations-Modelltyps erstellt, d.h. die in dem Workflowspezifikations-Modell festgelegte Workflowstruktur wird übernommen und der darin enthaltene Workflow kann dann ausgeführt werden. Jedes Modell des Workflowinstanz-Modelltyps stellt genau eine Abarbeitung des mit dem entsprechenden Workflowspezifikations-Modell spezifizierten, rekursiv verschachtelten Workflows dar. Insbesondere kommen für das zu entwickelnde Werkzeug nur Workflowinstanz-Modelle vor, die aus dem speziell für die Dienstmethodik entworfenen MNM-Dienstmethodik-Workflowspezifikations-Modell generiert werden. Sie werden im Folgenden als **MNM-Dienstmethodik-Workflowinstanz-Modelle** bezeichnet. Das spezifische Verhalten von Aktivitäten innerhalb eines Workflows, der durch ein Workflowspezifikations-Modell beschrieben und mit Workflow-Instanz-Modellen abgearbeitet wird, wird in eigenen Alter-Codefiles, sogenannten **Workflowmoduldateien** festgelegt. Jedem Workflowspezifikations-Modell sind dabei eigene Workflowmoduldateien zugeordnet. Auch für das MNM-Dienstmethodik-Workflowspezifikations-Modell wurden eigene Workflowmoduldateien (**MNM-Dienstmethodik-Workflowmoduldateien** genannt) entworfen. Diese MNM-Dienstmethodik-Workflowmoduldateien zusammen mit dem MNM-Dienstmethodik-Workflowspezifikations-Modell spezifizieren die Dienstmethodik vollständig, stellen also innerhalb der entworfenen Architektur die Methodik-Spezifikationsdaten dar. Der in den Workflowmoduldateien enthaltene Alter-Code wird jedoch bei Abarbeitung eines MNM-Methodik-Workflowinstanz-Modells ausgeführt, d.h. Workflowmodule stellen auch einen Teil der Methodik-Ausführungs-Komponente dar. Jedes Workflowinstanz-Modelle enthält insbesondere den aktuellen Zustand des darin gerade abgearbeiteten Workflows. Die MNM-Dienstmethodik-Workflowinstanz-Modell stellen somit innerhalb der Architektur die Workflow-Verwaltungsdaten dar.

Die Datenbasis-Komponente wird durch die bereits erwähnten Artefakt-Modelltypen realisiert. Jeder Modelltyp steht hierbei für eine bestimmte Art von Artefakten (z.B. für UML, für Dienstlebenszyklus, etc.). Die Dienstmethodik-Artefaktdaten, d.h. die Daten zur Repräsentation der jeweiligen Dienstmodell-Instanz selbst, sind dann in Modellen dieser Modelltypen enthalten. Die Methodik-Artefaktdaten innerhalb der Architektur werden also durch die Modelle der Artefakt-Modelltypen repräsentiert. Und zwar enthält jedes MNM-Dienstmethodik-Workflowinstanz-Modell entsprechend seinem momentanen Ausführungsstatus Modelle der Artefakt-Modelltypen als Submodelle. Dies heißt vor allem auch, dass jede Instanz der Modellierungsmethodik, d.h. jedes MNM-Dienstmethodik-Workflowinstanz-Modell, seine eigenen Dienstmethodik-Artefaktdaten besitzt.

Diese Beziehungen zwischen den entworfenen Modelltypen bzw. ihren Modellen sowie ihre Relation zur entworfenen Architektur sind in Abb. 5.7 grafisch dargestellt.

Zusammengefasst betrachtet wurden für das Werkzeug die verschiedenen, erwähnten Modelltypen sowie das spezielle MNM-Dienstmethodik-Workflowspezifikations-Modell und die zu diesem gehörigen MNM-Dienstmethodik-Workflowmoduldateien entworfen.

In Abschnitt 5.3 ist der Entwurf der Modelltypen für die Spezifikation und Abarbeitung von Workflows allgemein, also von Workflowspezifikations- und Workflowinstanz-Modelltyp, behandelt. In Abschnitt 5.4 wird eine Übersicht über die speziell für die Abarbeitung des Workflows für die MNM-Dienstmethodik benötigten Artefakt-Modelltypen, also die Modelltypen für die Repräsentation des Dienstmodells selbst, gegeben. In den Abschnitten 5.4.1 bis 5.4.5 werden diese Modelltypen dann einzeln betrachtet. Abschließend wird in Abschnitt 5.5 der speziell für die MNM-Dienstmodellierung entworfene Workflow, d.h. das MNM-Dienstmethodik-Workflowspezifikations-Modell bzw. die mit seiner Hilfe generierten MNM-Dienstmethodik-Workflowinstanz-Modelle, selbst beschrieben.

5.3 Spezifikation und Ausführung von Workflows innerhalb des Werkzeuges

Das Werkzeug benötigt zur Realisierung der MNM-Dienstmodellierungsmethodik eine flexible Workflowunterstützung. Dessen Entwurf wird nachfolgend erläutert.

In der in Abschnitt 3.4 entworfenen Architektur für das Werkzeug werden die Spezifikation und die Abarbeitung der Methodik einzeln berücksichtigt, d.h. es gibt eine Methodik-Spezifikations-Komponente, mit der die Methodik (d.h. alle darin enthaltenen Workflows) spezifiziert wird, und eine Methodik-Ausführungs-Komponente, die die vorher spezifizierte Methodik abarbeitet. Dieser Ansatz ermöglicht eine leichte Anpassbarkeit der Methodik innerhalb des Werkzeugs für denn Fall, dass die Methodik zukünftig verfeinert oder geändert wird.

Für die Umsetzung der Methodik-Spezifikations- und der Methodik-Ausführungs-Komponente innerhalb von *DoME* wurde jeweils ein *DoME*-Modelltyp entworfen: der **Workflowspezifikations-Modelltyp** für die Methodik-Spezifikations-Komponente und der **Workflowinstanz-Modelltyp** für die Methodik-Ausführungs-Komponente. Beide Modelltypen zusammen realisieren ein sehr allgemein gehaltenes, flexibles Workflowkonzept. Anstatt die vorhandene MNM-Dienstmodellierungsmethodik mit allen ihren derzeitigen Workflows und Aktivitäten direkt in den beiden entwickelten Modelltypen zu realisieren, wurde ein indirekter Ansatz verwendet. Der Workflowspezifikations-Modelltyp und der Workflowinstanz-Modelltyp sind prinzipiell nicht nur für die Spezifikation bzw. Ausführung der MNM-Dienstmethodik-Workflows, sondern für praktisch beliebige Workflows geeignet. Der Workflowspezifikations-Modelltyp dient zur Spezifikation von praktisch beliebigen Workflows, d.h zur Festlegung ihrer Workflowstruktur. Jedes Modell des Workflowspezifikations-Modelltyps legt dabei einen rekursiv verschachtelten Workflow fest. Das Verhalten der Aktivitäten des darin spezifizierten Workflows wird nicht im Modell selbst festgelegt, sondern in eigenen Altercodefiles, den sogenannten **Workflowmoduldateien**. Der Workflowinstanz-Modelltyp wird dagegen zur Abarbeitung von Workflows, die vorher mit einem Modell des Workflowspezifikations-Modelltyps spezifiziert wurden verwendet. Modelle des Workflowinstanz-Modelltyps werden automatisch mit Hilfe eines gegebenen Modell des Workflowspezifikations-Modelltyps, das einen rekursiv verschachtelten Workflow (dessen Struktur) spezifiziert, erstellt und können dann abgearbeitet werden (d.h. vor allem ihre Aktivitäten, deren genaues Verhalten in den Workflowmoduldateien festgelegt ist). Jedes Modell des Workflowspezifikations-Modelltyps stellt also eine Instanz des rekursiv verschachtelten Workflows, der mit dem Workflowspezifikations-Modelltyp spezifiziert wurde, dar.

Für die tatsächliche Unterstützung der MNM-Modellierungsmethodik wurde ein Modell des Workflowspezifikations-Modelltyps (**MNM-Dienstmethodik-Workflowspezifikations-Modell** genannt) entworfen, das alle in der MNM-Dienstmodellierungsmethodik beschriebenen Methodik-Workflows beinhaltet. Hierfür wurden eigene Workflowmoduldateien (**MNM-Dienstmethodik-Workflowmoduldateien** genannt) zur Festlegung des Verhaltens der Aktivitäten des Workflows entwickelt. Abgearbeitet wird die darin spezifizierte Methodik dann mit dem Workflowinstanz-Modelltyp. Jedes der dabei erstellten Modelle des Workflowinstanz-Modelltyps (**MNM-Dienstmethodik-Workflowinstanz-Modelle** genannt), d.h. der Modelle des Workflowinstanz-Modelltyps, deren Workflowstruktur von MNM-Dienstmethodik-Workflowspezifikation vorgegeben wurde, ist damit eine Instanz der ausgeführten Dienstmethodik.

Der Workflowspezifikations-Modelltyp entspricht wie bereits erwähnt der Methodik-Spezifikations-Komponente und der Workflowinstanz-Modelltyp der Methodik-Ausführungs-Komponente. Das spezielle MNM-Dienstmethodik-Workflowspezifikations-Modell zusammen mit den zugehörigen MNM-Dienstmethodik-Workflowmoduldateien spezifiziert die MNM-Dienstmethodik vollständig und entspricht in der Architektur den Methodik-Spezifikationsdaten. Der in den MNM-Dienstmethodik-Workflowmoduldateien enthaltene Alter-Code (bzw. von dort aus rekursiv aufgerufener Alter-Code) dient der Festlegung des Verhaltens der Aktivitäten des spezifizierten Workflows und wird daher bei der Ausführung des Workflows in Form eines MNM-Dienstmethodik-Workflowinstanz-Modells benutzt. Somit stellen die MNM-Dienstmethodik-Workflowmoduldateien ebenfalls einen Teil der Methodik-Ausführungs-Komponente dar. Die MNM-Dienstmethodik-Workflowinstanz-Modelle enthalten den aktuellen Zustand der von ihnen jeweils repräsentierten Instanz der Dienstmethodik, sie stellten bezüglich der entwickelten Architektur also die Workflow-Verwaltungsdaten dar

Nachfolgend wird auf die Spezifikation und Abarbeitung von Workflows innerhalb des Workflows allgemein eingegangen. Das spezielle für die MNM-Dienstmethodik entwickelte MNM-Dienstmethodik-Workflowspezifikations-Modell und seine zugehörigen MNM-Dienstmethodik-Workflowinstanz-Modelle werden dagegen in Abschnitt 5.5 behandelt. Im Abschnitt 5.3.1 wird die Spezifikation und im Abbschnitt 5.3.2 die Abarbeitung von Workflows besprochen.

5.3.1 Modelltyp für Workflowspezifikationen

Für die Festlegung von Workflows innerhalb des Werkzeuges wurde ein eigener *DoME*-Modelltyp entworfen, der sogenannte **Workflowspezifikations-Modelltyp**. Modelle dieses Modelltyps werden im Folgenden als **Workflowspezifikations-Modelle** oder kurz **Workflowspezifikationen** bezeichnet.

Der Benutzer legt in einem solchen Modell die Struktur eines Workflows fest. Ein Workflow besteht hierbei aus **Subworkflowschritten** und **Aktivitäten**. Die Subworkflowschritte beinhalten jeweils selbst einen eigenen, untergeordneten Workflow, einen sogenannten **Subworkflow**. Hiermit ist eine hierarchische Strukturierung von Workflows möglich. Aktivitäten dagegen sind die eigentlich auszuführenden Einheiten des Workflows, deren Verhalten eigens durch eigenen Alter-Code außerhalb des Workflowspezifikations-Modells festgelegt wird. Das Alter-Codefile für eine Aktivität, wird als **Workflowmoduldatei** oder kurz **Workflowmodul** bezeichnet. In einem Workflowspezifikations-Modell wird nur der Name des Workflowmoduls festgelegt.

Die hierarchische Strukturierung von Workflows aus Subworkflowschritten und Aktivitäten innerhalb des Werkzeugs wurde als Konzept gewählt, um in einem einzigen spezifizierten Workflow (d.h. in einem einzigen Workflowspezifikations-Modell) die gesamte Methodik aus ihren verschiedenen Methodik-Workflows und Methodik-Aktivitäten (z.B. auch Top-Down und Bottom-Up-Aufteilung) zu spezifizieren. Die Methodik gibt Methodik-Workflows aus Methodik-Aktivitäten vor. Die Methodik-Aktivitäten wiederum können in einzelne Benutzerinteraktionen aufgeteilt werden (siehe Abschnitt 3.3). Zur Spezifikation der Methodik innerhalb des Werkzeugs werden Methodik-Workflows und Methodik-Aktivitäten als Subworkflows und weitere Aufteilungen der Methodik-Aktivitäten (d.h. weitgehend die Benutzeraktionen der Methodik-Aktivitäten) als Aktivitäten innerhalb eines Top-Level-Workflows modelliert, dir mit einem Workflowspezifikations-Modell (MNM-Dienstmethodik-Workflowspezifikations-Modell genannt) spezifiziert und mit entsprechenden Workflowinstanz-Modellen (MNM-Dienstmethodik-Workflowinstanz-Modelle genannt) abgearbeitet wird. Eine detaillierte Betrachtung des MNM-Dienstmethodik-Workflows innerhalb des Werkzeugs wird in 5.5 gegeben.

Subworkflowschritte und Aktivitäten werden unter dem Begriff **Workflowschritt** zusammengefasst. Die Workflowschritte werden durch **Transitionen** verbunden, wobei Parallelität erlaubt ist. Außerdem sind Auswahlpunkte für bedingte Verzweigung möglich. Komplexere strukturelle Konzepte, wie z.B. Schleifen, sind von der Workflowstruktur selbst nicht vorgesehen. Für die Methodik des Dienstmodells sind sie nicht notwendig, außer ggf. für die rekursive Modellierung. Diese bezieht sich aber weitgehend nur auf die Übernahme von Artefakten und weniger auf die Struktur des Workflows. Derartige komplexe Konzepte müssen in den Aktivitäten bzw. deren Workflowmodulen selbst implementiert werden.

Außerdem können **Hyperlink-Knoten** für Workflowartefakte innerhalb des Workflows platziert werden.

Diese erleichtern für den Benutzer im aktiven Workflow, also später bei Ablauf des Workflow, die Navigation zu den Workflowartefakten.

Die Abb. 5.8 zeigt den prinzipiellen Aufbau eines Workflows, der durch ein Workflowspezifikations- bzw dessen zugehörige Workflowinstanz-Modelle repräsentiert wird in UML.

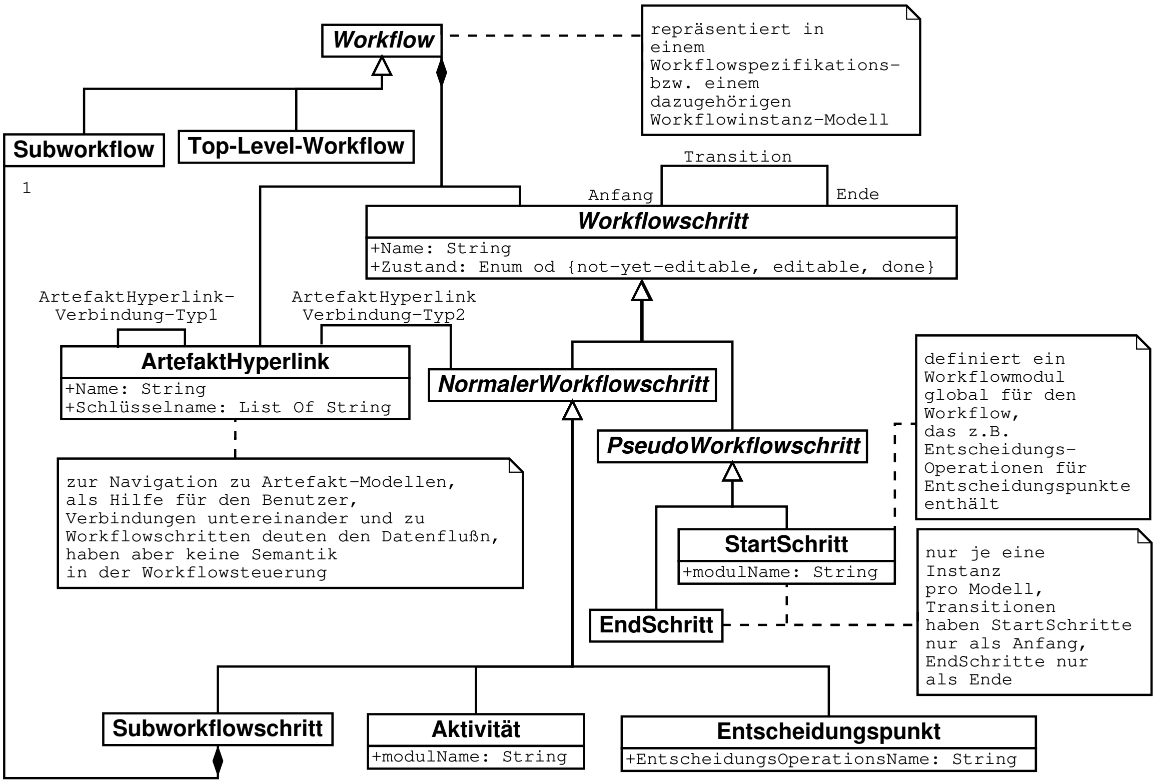

Abbildung 5.8: prinzipielle Struktur eines Workflows, der durch ein Workflowspezifikations- bzw. ein Workflowinstanz-Modell repräsentiert wird — dargestellt in UML

5.3.2 Modelltyp für Workflowinstanzen

Mit den im letzten Abschnitt beschriebenen Workflowspezifikations-Modellen werden Workflows für das Werkzeug spezifiziert. Um einen so beschriebenen Workflow tatsächlich innerhalb des Werkzeuges abzuarbeiten, wurde ein eigener *DoME*-Modelltyp entworfen: Der **Modelltyp für Workflowinstanzen**.

Ein Modell des Workflowinstanz-Modelltyps, also ein **Workflowinstanz-Modell** (kurz: eine **Workflowinstanz**), wird automatische mit Hilfe eines gegebenen Workflowspezifikations-Modells erstellt. Das gegebene Workflowspezifikations-Modell wird als **Workflowspezifikation der Workflowinstanz** bezeichnet. Diese ist natürlich zu unterscheiden vom Metamodell für Workflowinstanz-Modelle. Da in *DoME* das Metamodell (bzw. der Modelltyp) eines Modells auch als Spezifikation des Modells bezeichnet wird, ist im Zusammenhang mit Workflowinstanz-Modellen der Begriff „Spezifikation“ zweideutig. Bei Workflowinstanz-Modellen werden daher um Verwechslungen zu vermeiden immer die genaueren Begriffe Workflowspezifikation bzw. Metamodell (bzw. Modelltyp) verwendet.

Wie im vorigen Abschnitt beschrieben, bestehen Workflows innerhalb des Werkzeugs aus Subworkflowschritten zur hierarchischen Strukturierung und Aktivitäten, die die eigentlich auszuführenden Einheiten des Workflows darstellen. Das individuelle Verhalten einer Aktivität wird jeweils durch Alter-Code in einer eigenen Workflowmoduldatei spezifiziert. Die Struktur eines Workflows und die Zuordnung von

Aktivitäten zu Workflowmoduldateien wird in einem Workflowspezifikations-Modell spezifiziert. Diese Struktur wird zur tatsächlichen Ausführung des Workflows in ein Workflowinstanz-Modell übernommen. In diesem werden die Aktivitäten (und die Subworkflowschritte) verwaltet und abhängig von der Workflowstruktur ausgeführt (siehe Abb. 5.9 allgemein für Workflows innerhalb des Workflows, vgl. auch Abb. 5.7, für die spezifische Realisierung der MNM-Dienstmethodik mit diesem allgemeinen Prinzip). Ganz grob betrachtet besteht ein Workflow innerhalb des Werkzeugs also aus Aktivitäten mit individuellem Verhalten, die von einer globalen Workflowsteuerung gemanagt werden.

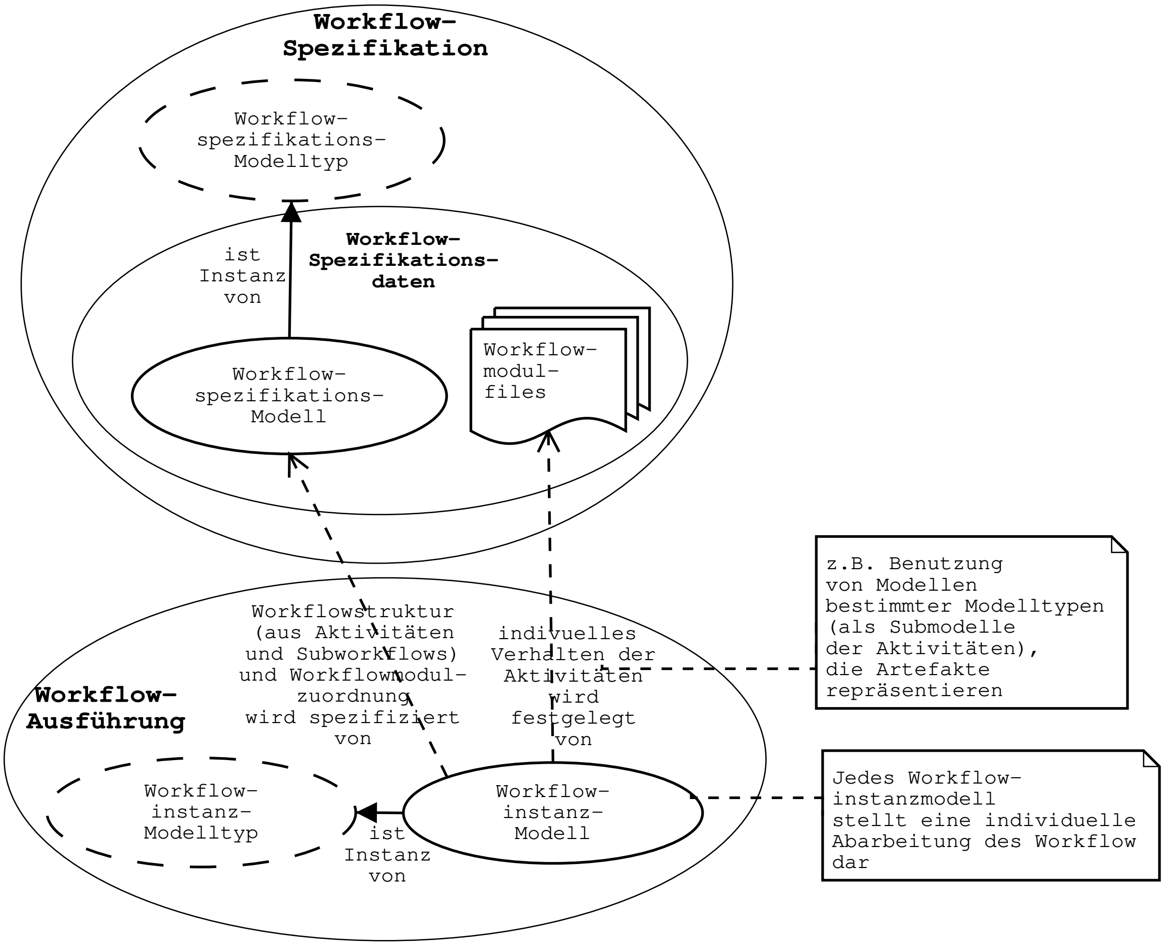

Abbildung 5.9: Skizze der Workflowrealisierung in *DoME* allgemein

Eine aus einer Workflowspezifikation erstellte Workflowinstanz lässt den Benutzer den Workflow selbst nicht editieren, sondern erlaubt einerseits eine (hierarchische) Navigation über den Workflow und andererseits die Ausführung des Workflows, d.h. die Verwaltung und Iteration des Bearbeitungsstatus von Workflowschritten, sowie die Ausführung der Aktivitäten selbst.

Dazu hat zu jedem Zeitpunkt während der Ausführung sowohl jeder Workflowschritt (jeder Subworkflowschritt und jede Aktivität) als auch jeder einem Subworkflowschritt untergeordneter Subworkflow einen **Workflow- oder Bearbeitungsstatus**, der einen der drei folgenden Werte annehmen kann:

- **Noch nicht bearbeitbar (not yet editable):** Der Schritt bzw. Subworkflow ist noch nicht bearbeitbar, da vorhergehende Schritte noch nicht abgeschlossen sind.
- **Bearbeitbar (editable):** Der Schritt bzw. Subworkflow wird derzeit bearbeitet.
- **Abgeschlossen (done):** Der Schritt bzw. Subworkflow ist vollständig bearbeitet und abgeschlossen.

Das tatsächliche Verhalten der Aktivitäten des Workflows wird in Alter-Codefiles festgelegt, in sogenannten **Workflowmoduldateien** oder kurz **Workflowmodulen**. Der darin spezifizierte Code dient vor allem zum Ausführen von individuellen Aktionen, wenn sich der Workflowstatus der Aktivität ändert (z.B. wenn die Aktivität bearbeitbar wird), oder zum Überprüfen, ob die Aktivität abgeschlossen werden kann. Auch können damit Aktionen festgelegt werden, die gleich bei Erstellung des Workflows stattfinden.

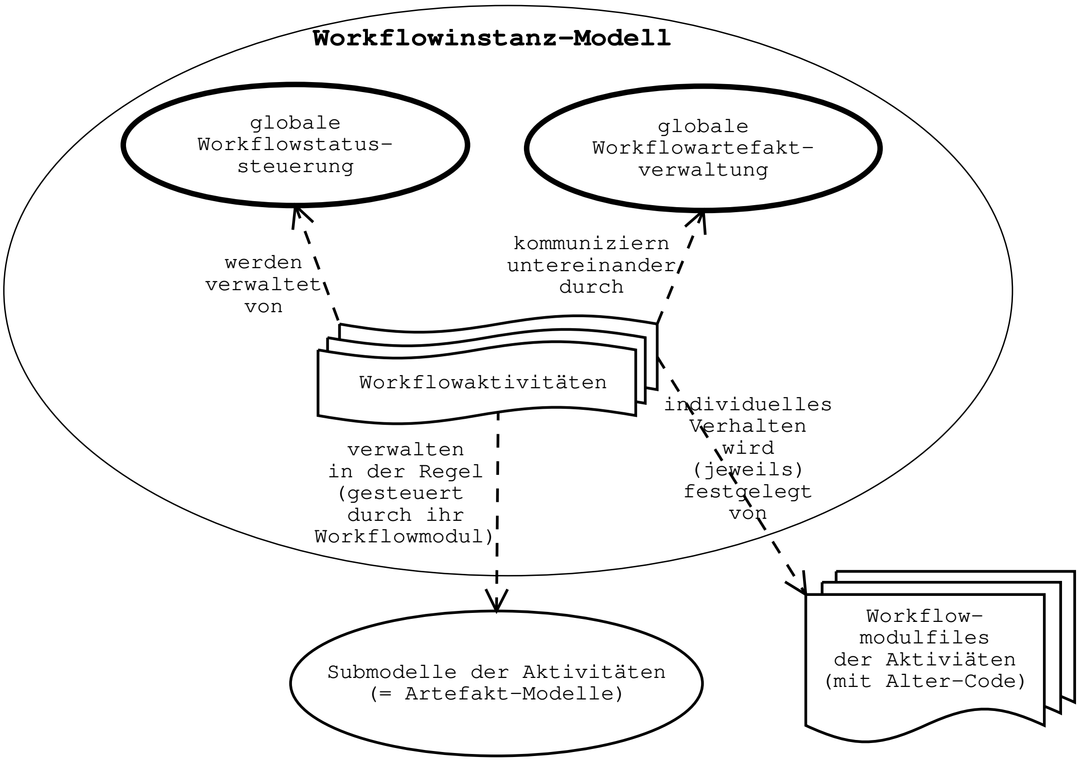

Abbildung 5.10: Workflowaktivität als Schnittstelle zwischen Workflow und Artefakt-Modellen

Die Aktivitäten haben vor allem die Möglichkeit, beliebige Submodelle als ihre Artefakte zu verwenden. Für die Realisierung des Dienstmodellierungsmethodik z.B. wurden eine Reihe verschiedener Modelltypen für Artefakte (kurz: **Artefakt-Modelltypen**) entwickelt (siehe Abschnitt 5.4).

Der Workflowinstanz-Modelltyp unterstützt außerdem eine globale Artefaktverwaltung. Hierfür können Aktivitäten (oder Submodelle davon) beim Workflowinstanz-Modell Artefakte (z.B. ihre Submodelle oder andere beliebige Alter-Objekte) unter bestimmten Schlüsselnamen abspeichern (bzw. referenzieren) oder diese abfragen, sowie Interesse bei Änderung dieser registrierten Artefakte anmelden. Auch die bereits im Workflowspezifikations-Modell festgelegten Artefakt-Hyperlinks benutzen die globale Workflowartefaktverwaltung. Jeder solche Hyperlink registriert sich gleich bei Erstellung des Workflows für ein Workflowartefakt. Der Name des Artefakts wird in der Workflowspezifikation festgelegt.

Eine Aktivität muss jedoch nicht unbedingt Submodelle erstellen und verwalten, sie kann (spezifiziert in der ihr zugeordneten Workflowmoduldatei als Alter-Code) jeden beliebigen Alter-Code ausführen (z.B. direkte Interaktion mit dem Benutzer durch Menüs, Dialogfenster, Aufrufen von externen Programmen, Kommunikation über Netzdienste, sogar Ausführung von praktisch beliebigem Smalltalk-Code von Alter aus möglich). Da *DoME* jedoch modell- bzw. graphenorientiert ist, bietet sich dieses Vorgehen an und es wird zur Umsetzung der MNM-Dienstmethodik auch stets so vorgegangen (siehe Abschnitt 5.5).

Zusammengefasst betrachtet, besteht ein Workflow innerhalb des Werkzeugs aus Aktivitäten mit unterschiedlichem Verhalten, die von einer globalen Workflowsteuerung gemanagt werden. Die Aktivitäten haben ein individuelles Verhalten, das durch Alter-Code in Workflowmoduldateien festgelegt wird. Jede

Aktivität kann bei ihrer spezifischen Ausführung verschiedene Workflowartefakte (z.B. als ihre Submodelle) erstellen und verwalten. Die Aktivitäten können diese Workflowartefakte über eine globale Workflowartefaktverwaltung untereinander austauschen. Die Workflowaktivitäten stellen somit die Schnittstelle zwischen dem Workflow, d.h. dem Workflowinstanz-Modell, und den Workflowartefakten dar (siehe Abb. 5.10).

5.4 Modelltypen für Artefakte der MNM-Dienstmethodik

Die in Abschnitt 3.4 entworfene Architektur enthält eine Datenbasis-Komponente zur Speicherung und Verwaltung der MNM-Dienstmethodik-Artefaktdaten. Die davon verwalteten Artefaktdaten sind die eigentlichen Daten, die das Dienstmodell (bzw. eine Instanz davon) darstellten bzw. aus denen es gewonnen wird.

Zur Repräsentation der verschiedenen Teile des Dienstmodells, also der MNM-Dienstmethodik-Artefaktdaten, innerhalb des Werkzeugs wurden eine Reihe von *DoME*-Modelltypen entworfen. Jeder dieser Modelltypen repräsentiert einen bestimmten Teil bzw. Aspekt der MNM-Dienstmethodik-Artefaktdaten. Sie werden alle unter dem Begriff **MNM-Dienstmethodik-Artefakt-Modelltypen** (kurz **Artefakt-Modelltypen**) zusammengefasst. Innerhalb der Architektur repräsentieren diese Modelltypen die Datenbasis-Komponente. Die zu ihnen jeweils gehörigen Modelle (**MNM-Dienstmethodik-Artefakt-Modelle** oder kurz **Artefakt-Modelle**) stellen dabei die MNM-Dienstmethodik-Artefaktdaten dar.

Die MNM-Dienstmethodik-Artefakt-Modelle werden innerhalb des Werkzeugs zur Realisierung der Dienstmethodik als Submodelle von Aktivitäten eines MNM-Dienstmethodik-Workflowinstanz-Modells (also einer ausgeführten Instanz der Dienstmethodik innerhalb des Werkzeugs) verwendet. Die Verwendung eines Artefakt-Modells als Submodell einer Aktivität wird hierbei durch den spezifischen Alter-Code der zur Aktivität gehörenden Workflowmoduldatei bestimmt. (vgl. Abschnitt 5.3.2).

Hier werden die Artefakt-Modelltypen und ihre Beziehungen untereinander selbst betrachtet. Ihre Verwendung (als Submodell) bei Aktivitäten eines MNM-Dienstmethodik-Workflowinstanz-Modells wird dagegen bei genauer Betrachtung des speziell für die MNM-Dienstmethodik entwickelten Workflows (dem MNM-Dienstmethodik-Workflowspezifikations-Modell) in Abschnitt 5.5) behandelt.

Die MNM-Dientmethodik-Artefakt-Modelltypen lassen sich folgendermassen einteilen:

- Modelltypen für Strukturierung, d.h. für Dienstlebenszyklus, Prozessklassifizierung, QoS-Dimensionen und Rollen eines Dienstes.
- Modelltyp für UML (alle notwendigen UML-Diagrammtypen)
- Modelltyp für beliebige, generische Beschreibungen
- Servicemodel-Modelltyp (kurz SM-Modelltyp) für die eigentlichen Elemente des Dienstmodells und Beziehungen darunter
- Modelltyp für Basiseinstellungen (d.h. Modellierungsfall und andere, allgemeine Einstellungen)

In Abschnitt 5.4.1 werden die Modelltypen für Strukturierung besprochen. Abschnitt 5.4.2 geht auf den UML-Modelltyp, der zur Realisierung von sämtlichen UML-Diagrammen innerhalb des Werkzeugs verwendet wird, ein. In Abschnitt 5.4.3 wird der Modelltyp für generische Beschreibungen erklärt. Er ermöglicht das Beschreiben von beliebigen Begriffen und Konzepten in verschiedenen Formen wie z.B. Freitext, als Liste, mit Dokument- und Standard-Referenzen sowie mit *DoME*-Modellen eines beliebigen Modelltyps. Insbesondere kann mit der zuletztgenannten Möglichkeit z.B. ein Modell des UML-Modelltyps oder auch ein Modell eines in *DoME* bereits existierenden Modelltyps (auch Modelltypen, die nicht explizit für das Werkzeug entwickelt wurden; unabhängig davon ob Smalltalk- oder protodome-basiert) innerhalb eines Modells des Modelltyps für generische Beschreibung benutzt werden.

Der sogenannte SM-Modelltyp dient der Spezifikation der eigentlichen Elemente des Dienstmodells und den spezifischen Beziehungen unter diesen. (z.B. Dienste, Prozesse, Entitäten, QoS-Parameter, Ressourcen, Clients, etc. aber auch Beziehungen untereinander wie z.B. „Client nutzt Schnittstelle“ etc.) Die Modelle des SM-Modelltyps verwenden als Submodelle einerseits Modelle der Strukturierungs-Modelltypen zur Strukturierung der in ihnen definierten Dienstmodellelemente und andererseits Modelle des UML-Modelltyps und Modelle des Modelltyps für generische Beschreibungen, um die in ihnen definierten Dienstmodellelemente zusätzlich zu beschreiben. Das heißt für Dienstmodellelemente, für die die Methodik klar festlegt, wie sie vor allem mit UML zu beschreiben sind (z.B. Prozesse durch UML-Aktivitäts- bzw. Kollaborationsdiagramme), wird diese Beschreibung mit Hilfe von entsprechenden Modells des UML-Modelltyps realisiert. Bei Dienstmodellelementen für die keine bestimmte Form der Beschreibung von der Methodik vorgegeben wird, werden Modelle des Modelltypen für generische Beschreibung verwendet. Der SM-Modelltyp wird Abschnitt 5.4.4 genauer betrachtet. Abschließend wird in Abschnitt 5.4.5 der eigens für Methodik-Basiseinstellungen (siehe Abschnitt 3.3.2 entwickelte Modelltyp behandelt.

Eine Übersicht über die Artefakt-Modelltypen findet sich in Abb. 5.11. Diese zeigt auch die Beziehungen zwischen dem SM-Modelltyp und den anderen von ihm verwendeten Modelltypen.

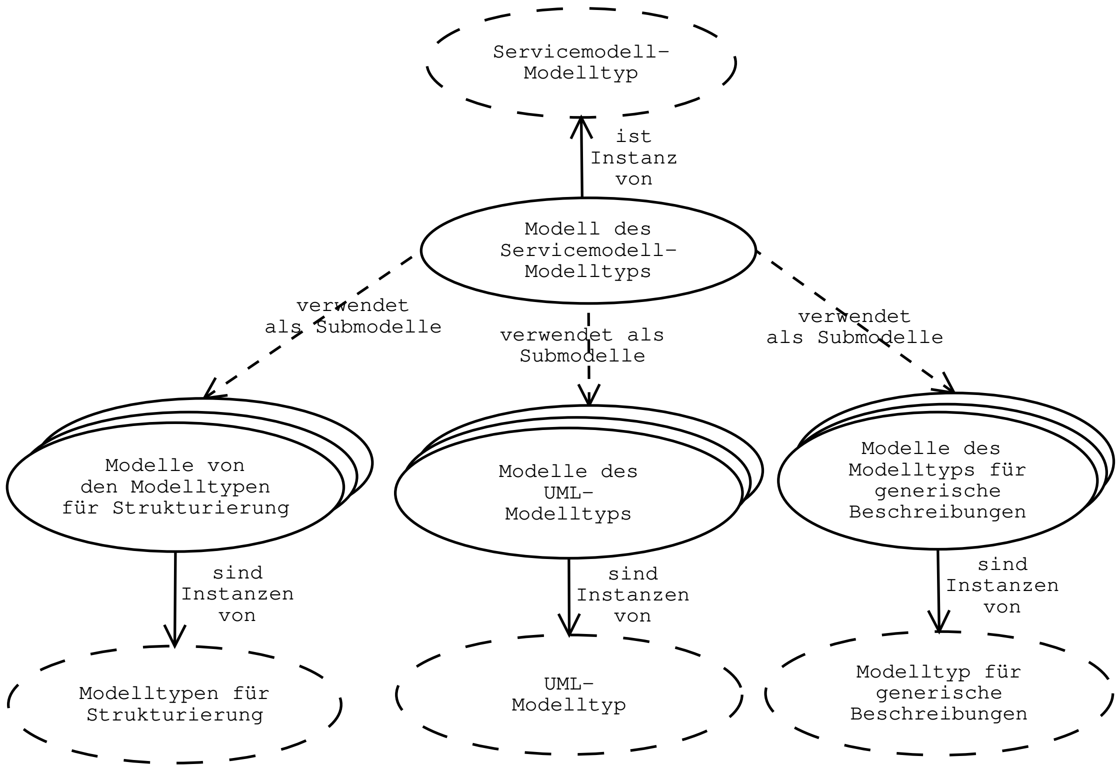

Abbildung 5.11: Artefakt-Modelltypen für das MNM-Dienstmodell bzw. seine Methodik

5.4.1 Modelltypen für Strukturierung

Speziell für die in der Methodik vorgegebenen Artefakte zur Strukturierung andrer Artefakte (z.B. Lebenszyklus) wurden eigene Modelltypen entworfen. Die strukturierenden Artefakte werden also stets selbst in eigenen Modellen dieser Modelltypen spezifiziert. Dies erhöht die Übersichtlichkeit bei der Modellierung. Insbesondere können spezifische Beziehungen unter den strukturierenden Artefakten selbst (z.B. Reihenfolge der Lebenszyklusphasen, „Prozessklasse tritt auf in Lebenszyklusphase“) getrennt von eigentlichen Dienstmodell-Daten, die durch die strukturierenden Artefakte klassifiziert werden, behandelt werden.

In der Methodik des Dienstmodells kommen explizit drei verschiedene Artefakte für die Strukturierung anderer Artefakte vor:

- Die Phasen des Dienstlebenszyklus für die Dienstsicht
- Die Prozessklassifizierungen sowohl für Dienstsicht als auch für die Realisierungssicht (verfeinert gegenüber dem der Dienstsicht)
- Die QoS-Dimensionen für die QoS-Parameter der Dienstsicht

Für die Spezifikation jedes dieser strukturierenden Artefakte wurde jeweils ein eigener Modelltyp entworfen.

Ein Modell für den Dienstlebenszyklus besteht dabei aus Knoten, die je eine Phase des Lebenszyklus darstellen. Die Reihenfolge der Phasen wird durch Kantenverbindungen zwischen den entsprechenden Knoten festgelegt.

Die Modelltypen für Prozessklassifizierung und QoS-Dimensionen sind ähnlich aufgebaut. Zugehörige Modelle enthalten jeweils Knoten, die Prozessklassen oder QoS-Dimensionen darstellen. Optional ist mit Kantenverbindungen eine Vererbungsbeziehung zwischen Prozessklassen bzw. QoS-Dimensionen darstellbar. Bei den Prozess-Klassen werden jedoch Klassen für Dienstnutzung (usage class) und Dienstmanagement (management class) unterschieden.

Außerdem wurde für die Spezifikation von Rollen, in denen legale Entitäten bezüglich eines Dienstes auftreten können, auch ein eigener Modelltyp (kurz **Rollen-Modelltyp** genannt) entwickelt. Denn die Methodik des Dienstmodells erlaubt eine Verfeinerung der grundlegenden Rollen Kunde (customer), Nutzer (user) und Anbieter (provider). Diese grundlegenden Rollen sollen im Folgenden als **Basisrollen** bezeichnet werden. Der Modelltyp ist ähnlich zu den Modelltypen für Prozessklassifizierung bzw. QoS-Dimensionen. **Rollen** werden durch Knoten im Modell dargestellt, wobei jede Rolle genau einer der Basisrollen zugeordnet ist. Die Verfeinerung von Rollen ist durch eine mittels Kanten dargestellter Vererbungsbeziehung zwischen Rollenknoten, die der gleichen Basisrolle zugeordnet sind, möglich.

Mit dem Rollen-Modelltyp wird also nur eine Verfeinerung der Basisrollen definiert. Die dabei spezifizierten, allgemeinen Rollen sind unabhängig von einem bestimmten Dienst. Tritt bei einem Dienst nun eine bestimmte Rolle auf, d.h. tritt eine legale Entität in der Rolle bezüglich des Dienstes auf, so wird diese Rolle einer der (mit dem Rollen-Modelltyp festgelegten) allgemeinen Rollen zugeordnet. Um Verwechslungen zwischen der allgemeinen, mit dem Rollen-Modelltyp festgelegten Rolle und Rolle bezüglich des Dienstes, die der allgemeinen Rolle zugeordnet ist, zu vermeiden, wird die Rolle bezüglich des Dienstes genauer als **Rolleninstanz**, also als Instanz der allgemeinen Rolle, bezeichnet.

Insgesamt treten also legale Entitäten in einer Rolleninstanz bezüglich eines Dienstes auf. Die Rolleninstanz ist dabei einer (allgemeinen) Rolle und diese Rolle ist selbst einer der drei Basisrollen zugeordnet.

5.4.2 Modelltyp für UML

In *DoME* gibt es bereits eine Reihe von Modelltypen für die verschiedenen UML-Diagrammtypen. Diese realisieren zumindest die rein grafischen Aspekte des UML-Standards vollständig. Die UML-Spezifikation (siehe [OMG 01-02-14]) sieht jedoch bei vielen UML-Elementen Eigenschaften und Beziehungen vor, für die zwar keine direkte Darstellung in den UML-Diagrammen gefordert bzw. vorgeschlagen wird, die jedoch zur vollständigen Modellierung mit UML definiert wurden. Die UML-Spezifikation definiert das sogenannte UML-Metamodell (nicht mit Metamodellen in *DoME* zu verwechseln) zur generischen Spezifikation aller UML-Elemente. Das UML-Metamodell ist aufgebaut aus sogenannten UML-Metaklassen die sich aus Metaattributen und Metareferenzen zu anderen Metaobjekten (sozusagen Metaobjekt-wertige Attribute) zusammensetzten und beschreibt die mögliche Struktur von UML-Modellen, die dann mittels UML-Diagrammen dargestellt werden können.

Da das MNM-Dienstmodell UML-Diagramme zur Spezifikation für viele Elemente des Dienstmodells vorsieht, also zum Großteil mit UML-Konzepten modelliert wird, ist eine UML-Modellierung mit möglichst vollständig realisiertem UML-Metamodell innerhalb des Werkzeuges wünschenswert. Da die in *DoME* bisher für UML-Diagramme existierenden Modelltypen aber diese Anforderung nicht erfüllen, wurde für das Werkzeug ein eigener Modelltyp für UML-Diagramme ausgehend von den vorhandenen Modelltypen entwickelt. Die bereits realisierten, grafischen UML-Konzepte wurden dabei aus den vorhandenen Modelltypen in den neuen Modelltyp übernommen.

Das UML-Metamodell besitzt viele, unterschiedliche Metaklassen, die viele verschiedene Beziehungen untereinander haben. Es wurde daher für einen ersten Entwurf bzw. die prototypische Implementierung statt mehrerer Modelltypen (etwa für jeden UML-Diagrammtyp einen, wie dies bei den vorhandenen Modelltypen der Fall ist) ein einziger Modelltyp (**UML-Modelltyp** genannt) entworfen, der alle Aspekte von UML beinhaltet und zur Repräsentation für jede Art von UML-Diagramm (zumindest alle für die MNM-Dienstmodellierung notwendigen) verwendet werden kann. Für das MNM-Dienstmodell sind die folgenden UML-Diagrammtypen relevant: Klassendiagramme, Use-Case-Diagramme, Kollaborationsdiagramme und Aktivitätsdiagramme (vgl. Abschnitt 3.2).

Die einzelnen Anforderungen an den Entwurf und die Entwicklung des UML-Modelltyps sind:

- Möglichst vollständige Realisierung des UML-Metamodells:
  - Realisierung aller UML-Metaklassen, die für Klassen-, Use-Case-, Kollaborations- und Aktivitätsdiagramme notwendig sind (d.h. nicht unbedingt notwendig die für Komponentendiagramme oder allgemeine Zustandsdiagramme)
  - Pro UML-Metaklasse: möglichst alle Metaattribute und Metareferenzen
  - Pro UML-Metaklasse: möglichst alle Darstellungsmöglichkeiten, die die UML-Spezifikation anbietet
- Kopplung zwischen Elementen des Dienstmodells, die außerhalb von UML definiert sind (und nur durch UML genauer spezifiziert werden) und den UML-Modellelementen, die Dienstmodellelemente repräsentieren

Speziell grafische Konzepte für die vollständige Darstellung von (binären) UML-Assoziationen (z.B. zwischen Use-Cases und Aktoren), mussten entworfen und entwickelt werden.

Da einerseits ein einziger Modelltyp für alle UML-Diagrammtypen verwendet wird und andererseits alle Details des UML-Metamodells integriert werden sollten, d.h. auch Details für die von der UML-Spezifikation keine Darstellung vorgeschlagen wird, sind Modelle des UML-Modelltyps hierarchisch aufgebaut: Es wird jeweils eine Hierarchie von Modellen des UML-Modelltyps verwendet um alle Elemente von UML grafisch darzustellen. Dies schließt auch die Elemente bzw. Details ein, für die die UML-Spezifikation keine (eigenständige) grafische Repräsentation vorsieht bzw. die grafisch nur innerhalb eines anderen Elementes angezeigt werden sollen. (z.B. Parameter einer Operation, Guard oder Action einer Transition). Derartige Elemente, für die die UML-Spezifikation keine eigene Darstellung vorsieht werden einerseits in der von der UML-Spezifikation vorgeschlagenen Weise innerhalb anderer UML-Elemente angezeigt. Andererseits werden sie aber selbst in einem Submodell (in der Regel einem Submodell des UML-Elements, dem sie semantisch zugeordnet sind, z.B. ein Parameter seiner Operation, eine Aktion oder ein Guard seiner Transition, etc.) definiert, stellen dort also grafisch ein eigenes Objekt dar.

Modelle des UML-Modelltyps stellen somit jeweils einen oder mehrere (rekursiv enthaltene) UML-Diagramme bestimmter Typen (inkl. aller Details) dar.

5.4.3 Modelltyp für generische Beschreibung

Die Methodik des MNM-Dienstmodells gibt bei einer Reihe von Artefakten nicht vor, in welcher Form sie zu genauer zu beschreiben sind. Es werden nur Hinweise gegeben, wie eine Beschreibung aussehen könnte: z.B. als Freitext, als Liste, mit Referenzen zu Standard etc. (siehe Abschnitt 2.3). Um die Möglichkeiten und Freiräume der Beschreibung innerhalb des Werkzeugs für solche Artefakte nicht unnötig einzuschränken, wurde deren Beschreibung ein generisch gehaltener Modelltyp entwickelt. Er wird im Weiteren **Modelltyp für generische Beschreibungen** genannt.

Diese Artefakte der Methodik, die mit Modellen dieses Modelltyps näher beschrieben werden können, lassen sich grob in zwei Gruppen teilen (vgl. Abschnitt 2.3):

- Artefakte für die Spezifikation verschiedener Anforderungen. Diese sind stets eine Eingabe der Methodik-Workflows.
- Artefakte, die in den Workflows erstellt werden: Sie spezifizieren bestimmte Elemente bzw. Aspekte innerhalb der Dienstmodell-Instanz (die nicht oder nur teilweise durch UML spezifiziert werden) genauer. Die Methodik schlägt für diese Artefakte meist verschiedene Formen wie z.B. Freitext oder Listenform vor. Außerdem sollen Referenzen zu Standards gemacht werden können.

In der Methodik kommen die folgenden Anforderungs-Beschreibungen vor:

- Spezifikation der Transparenzen beim Basis-Modell-Workflow.
- Kundenanforderungen für Funktionalität, QoS und Clients beim Top-Down-Vorgehen im Dienstsicht-Workflow
- Provider-Anforderungen bei Bottom-Up-Vorgehen im Realisierungssicht-Workflow

Genauere Beschreibungen sind notwendig für die folgenden Elemente des Dienstmodells, da sie keine vorgegebene Beschreibung durch UML haben:

- QoS-Parameter
- Clients bzw. Schnittstellen für Dienst-Nutzung und Dienst-Management
- Den Dienst konkretisierende Informationen für die Dienstvereinbarung
- Sub-Clients
- Ressourcen/BMF, sowie Ressourcen-Zugangspunkte
- Dienstarchitektur

Im Weiteren wird nun der Modelltyp für generische Beschreibungen selbst erklärt. Er ist wie bereits erwähnt sehr allgemein, um möglichst jede Beschreibungsform, insbesondere die von der Methodik vorgeschlagen, zu unterstützen: Modelle dieses generischen Modelltyps enthalten einen oder mehrere Knoten, von denen jeder eine eigene Teilbeschreibung innerhalb der Gesamtbeschreibung durch das ganze Modell darstellt. Diese Knoten werden im Folgenden als **Beschreibungselemente** des Modells bezeichnet. Es gibt dabei verschiedene Typen von Beschreibungselementen, um möglichst alle denkbaren Möglichkeiten zu erfassen:

- **Plaintext-Beschreibung oder Plaintext-Listen-Beschreibung:** reiner Freitext oder eine Liste von Texten
- **Struktur-Beschreibung:** generisches Struktur-Konzept, das sich aus beliebigen Komponenten zusammensetzt, in Anlehnung an strukturierte Datentypen aus klassischen imperativen Programmiersprachen. Für jede Komponente können der Name, der Datentyp, die Multiplizität sowie der Datenwert jeweils als Freitext angegeben werden.
- **Dateiliste:** zur Referenzierung von externen Dateien; Es können hierbei mehrere Dateien durch ihren Dateinamen sowie die Angabe ihres Dateityps (in Form von Freitext) referenziert werden.
- **Normale Sub-Beschreibung:** zur hierarchischen Strukturierung von Modellen des Modelltyps für generische Beschreibung; Eine normale Sub-Beschreibung hat als Submodell selbst ein Modell für generische Beschreibungen.
- **Allgemeine Sub-Beschreibung:** generisches Konzept für Verwendung von beliebigen Submodellen als Beschreibung (nicht nur Modelle der Modelltypen, die speziell für das Dienstmodell entwickelt

wurden, sondern Modelle von beliebigen Modelltypen, die in *DoME* vorhanden sind); Eine allgemeine Sub-Beschreibung kann ein beliebiges Modell als Submodell haben.

- **Dokumentreferenzen-Liste:** zur Referenzierung von Dokumenten nach der Art von *Bibtex* (zu *TeX* bzw. *LaTeX* gehöriges Programm, siehe [Lamp 86]); Hierbei können in einer Dokumentreferenz alle Teile eines *Bibtex*-Eintrags einschließlich des *Bibtex*-Keys und der *Bibtex*-Datei selbst gespeichert werden. Eine Dokumentreferenz-Liste kann mehrere Referenzen enthalten. Der Import von *Bibtex*-Einträgen aus *Bibtex*-Dateien wird rudimentär unterstützt.

  Als Erweiterung zu den reinen Dokumentreferenzen gibt es Standardreferenzen zur Referenzierung von Standards. Diese enthalten jeweils zusätzliche Attribute für die explizite Angabe des Standards, der im referenzierten Dokument spezifiziert ist:

  - Der Herausgeber des Standards (developer)
  - Die Dokumentbezeichnung des Standards durch den Herausgeber
  - Der Status (z.B. für Werte wie „draft", „stable" „obsolete" oder „anerkannt durch ISO", etc.) des Standards.
  - Eine textuelle Beschreibung der durch den Standard spezifizierten Sache bzw. des Konzepts

Außerdem können zu jedem Beschreibungselement Aspekte (bezüglich des mit dem Modell insgesamt spezifizierten Begriffs/Konzepts, z.B. einer Ressource der Preis als Aspekt) festgelegt werden, auf die sich das Beschreibungselement genauer bezieht.

Weiterhin sind mit Hilfe von Kanten Beziehungen zwischen Beschreibungselementen definierbar — ähnlich zu UML-Dependencies. Hiermit kann z.B. festgelegt werden, auf welche Dokument-Referenzen in einem als Freitext spezifizierten Text Bezug genommen wird. Allgemeiner kann hiermit beschrieben werden, dass ein Beschreibungselement ein anderes detaillierter beschreibt.

Modelle des Modelltyps für generische Beschreibung kommen in Modellen des SM-Modelltyps (siehe Abschnitt 5.4.4) jeweils als Submodelle der darin enthaltenen Dienstmodellelemente, die keine sonstige, genauere Beschreibung durch UML haben, vor. Hiermit ist eine generische Beschreibung aller, während der Modellierung definierten Dienstmodellelemente möglich.

Die Abb. 5.12 zeigt die grobe Struktur eines Modells des Modelltyps für generische Beschreibung als UML-Klassendiagramm.

5.4.4 Der Servicemodel-Modelltyp

Der Modeltyp für generische Beschreibungen erlaubt eine generische Beschreibung von beliebigen Objekten oder Begriffen, auch für die eigentlichen Elemente des Dienstmodells. Er kennt und unterscheidet jedoch die jeweiligen Objekte bzw. Begriffe nicht selbst, sondern bietet nur generische Möglichkeiten für Beschreibungen.

Um innerhalb des Werkzeugs die verschiedenen Elemente des Dienstmodells selbst sowie ihre speziellen Beziehungen untereinander zu definieren wurde ein spezieller Modelltyp entworfen, der im Folgenden als **Servicemodel-Modelltyp** oder kurz **SM-Modelltyp** bezeichnet wird.

Eine Instanz des SM-Modelltyps definiert die verschiedenen Dienstmodellelemente und die spezifischen Beziehungen unter diesen. Jedes Dienstmodellelement wird in der Regel durch einen eigenen Knoten im Modell dargestellt. Um aber die definierten Elemente genauer zu beschreiben werden als Submodelle Instanzen der Modelltypen für UML oder generische Beschreibung verwendet. Außerdem werden Instanzen der Modelltypen für Rollen, Dienstlebenszyklus, Prozessklassifizierung und QoS-Dimensionen zur Strukturierung der definierten Elemente benutzt.

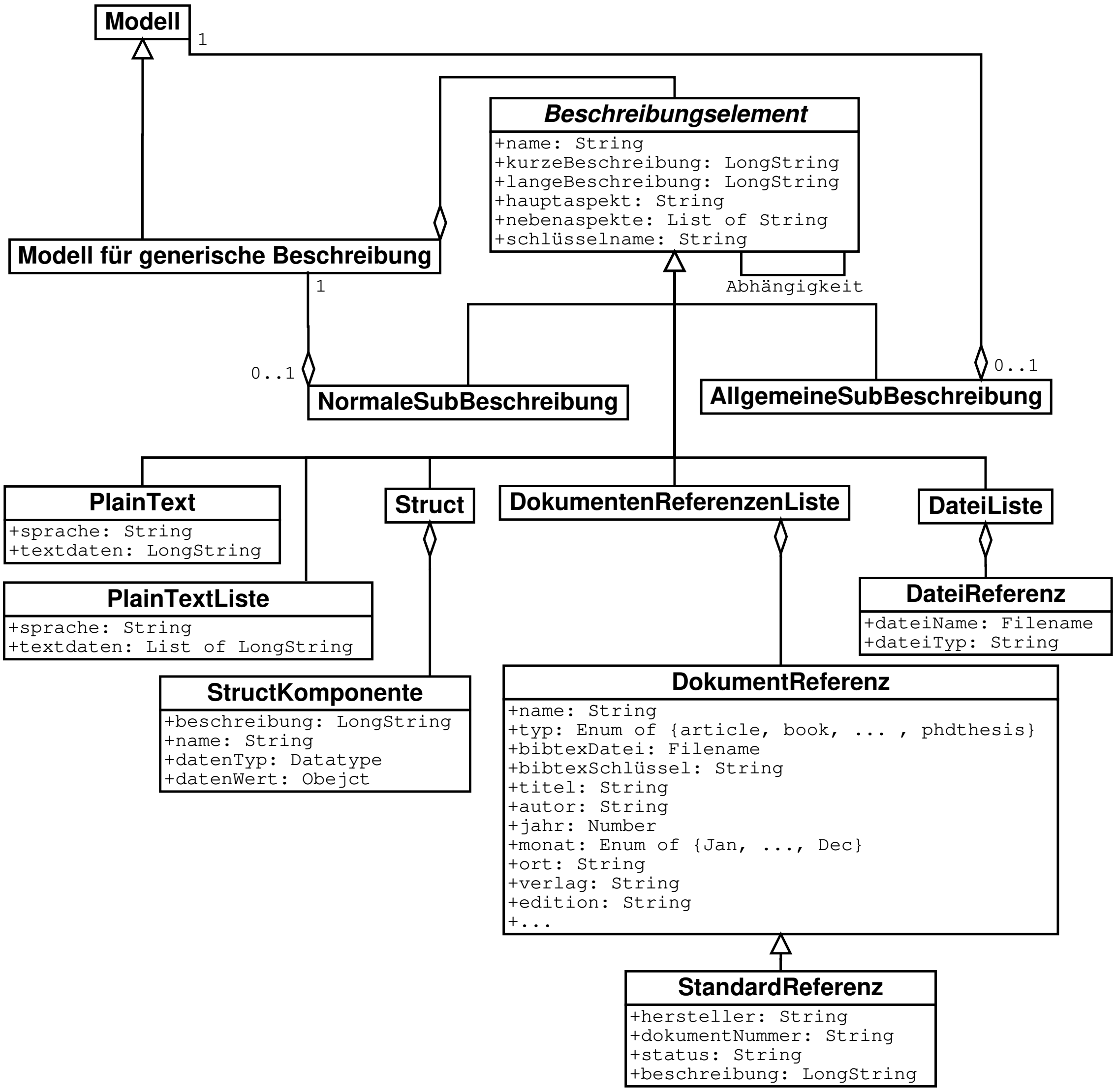

Abbildung 5.12: Struktur eines Modells für generische Beschreibung — dargestellt in UML

Das MNM-Dienstmodell besteht aus den drei Sichten Basis-Modell, Dienstsicht und Realisierungssicht. Der SM-Modelltyp erlaubt die Definition von Elementen aus allen drei Sichten. Die Elemente (bzw Beziehungen zwischen diesen) des Basis-Modell sind (vgl. Abschnitt 2.1):

- Entitäten (entities)
- Haupt- und Sub-Dienste (main and sub services), sowie Kanten zwischen diesen, die anzeigen, dass ein Sub-Dienst für die Erbringung eines anderen Dienstes verwendet wird.
- Rolleninstanzen (role instances), die bezüglich eines Haupt- oder Sub-Dienstes von Entitäten eingenommen werden können. Jede Rolleninstanz ist dabei genau einem Dienst zugeordnet und referenziert eine der mit dem Rollen-Modelltyp deklarierten Rollen.
- Entitäten werden durch Kanten den Rolleninstanzen zugeordnet, um anzuzeigen, in welcher Rolleninstanz bezüglich einem Dienst eine Entität auftritt.

Für die Dienstsicht kennt der SM-Modelltyp folgende Elemente:

- Prozesse (processes), die Prozessklassen zugeordnet sind; Genauer beschrieben werden Prozesse — wie von der Methodik vorgegeben — durch UML-Aktivitätsdiagramme: Das heißt sie haben Modelle des UML-Modelltyps, die Aktivitätsdiagramme enthalten, als Submodelle. Beziehungen zwischen den Prozessen werden durch UML-Use-Case-Diagramme, in denen die Prozesse als Use-Cases und die Entitäten als Aktoren auftreten können beschrieben. Im Modell gibt es dafür einen eigenen Knotentyp, die sogenannte Use-Case-Diagrammliste. Jeder Knoten dieses Typs kann beliebig viele Use-Case-Diagramme, d.h. Modelle des UML-Modelltyps, die Use-Case-Diagramme darstellen, beinhalten.
- QoS-Parameter (QoS parameter), die QoS-Dimensionen und Prozessen zugeordnet werden können.
- Dienst-Clients (service clients) und Dienst-Zugangspunkte (service access points), die einander durch Kanten-Beziehungen zugeordnet werden können. Es ist jeweils festlegbar, ob der Client bzw. der Zugangspunkt zur Dienstnutzung oder zum Management verwendet wird. Genauer kann jeweils auch die Rolleninstanz ausgewählt werden, bei der der Client bzw. der Zugangspunkt eingesetzt wird.
- Legale Rahmenvereinbarungen (legal frames) und SLA-Spezifikationen (SLA specifications), die als Teil der Dienstvereinbarung den modellierten Dienst näher konkretisieren.

Die Dienstvereinbarung besteht insgesamt aus allen Teilen der Dienstsicht, d.h. aus der Beschreibung der Dienstfunktionalität (Prozesse), den QoS-Parametern, den Client und Zugangspunkten. Zusätzlich dazu sind Informationen notwendig, die den Dienst näher konkretisierenden. In [Schm 01] wird ein Vorschlag für die Struktur einer Dienstvereinbarung und damit auch für die Informationen zur genaueren Konkretisierung des Dienstes gegeben. Beim SM-Modeltyp wird ein sehr ähnlicher Ansatz für diese konkretisierenden Informationen benutzt: Es einerseits eine legale Rahmenvereinbarungen, die den rein rechtlichen Teil des Vertrages zwischen Kunde und Anbieter festlegt, sowie sogenannte SLA-Spezifikationen, in denen für QoS-Parameter oder andere Parameter des Dienstes (die z.B. in den Aktivitäts-Diagrammen der Prozesse definiert wurden) exakte Werte festgelegt werden. In der Regel wird eine Dienstvereinbarung eine Rahmenvereinbarung und eine oder mehrere SLA-Spezifikationen enthalten. Jede SLA-Spezifikation definiert eine eigene Ausprägung des Hauptdienstes (z.B. verschiedene QoS-Parameter-Werte pro SLA-Spezifikation).

Sowohl für Rahmenvereinbarung, als auch für SLA-Spezifikation werden spezielle Submodelle des SM-Modelltyps verwendet, um ihren Inhalt genauer festzulegen. Die Knoten für Rahmenvereinbarung und SLA-Spezifikation stellen also selbst nur einen Gruppierungsmechanismus für die eigentlichen Dienst-Konkretisierungs-Informationen dar.

Der Inhalt einer Rahmenvereinbarung ist nicht vollständig festgelegt, sondern nur rudimentär vorgegeben. Bisher enthalten ihre speziellen SM-Modelltyp-Submodelle nur einen Typ von Knoten, die sogenannten legalen Entitäten. Diese Knoten beschreiben die beteiligten Vertragspartner, d.h. typischerweise mindestens einen Anbieter und einen Kunden. Legale Entitäten werden den Entitäten des Basismodells zugeordnet.

Darüberhinaus kann der Gerichtsstand bei jeder legalen Entität festgelegt werden. Für weitere Informationen in der Rahmenvereinbarung müsste der SM-Modelltyp um entsprechende Knotentypen erweitert werden.

Das Submodell einer SLA-Spezifikation kann die folgendenen Dienst-Konkretisierungs-Elemente enthalten:

- QoS-Parameter-Konkretisierung (QoS parameter instantiations), die einem der QoS-Parameter einen konkreten Wert zuweist.
- Konkretisierung von anderen Dienst-Parametern (other service parameter instantiations), z.B.Variablen, die in den Aktivitätsdiagrammen der Prozesse definiert wurden.
- Beschreibung von Reports, die den QoS-Parameter-Konkretisierungen zugeordnet werden.

Auch diese Möglichkeiten für den Inhalt einer SLA-Spezifikation sind nicht unbedingt vollständig. Der SM-Modelltyp kann ggf. erweitert werden, um weitere Informationen für die SLA-Spezifikation aufzunehmen.

Für die Realisierungssicht gibt es im SM-Modelltyp die folgenden Elemente:

- Prozesse (processes), ähnlich wie bei der Dienstsicht, nur das hier als UML-Modelltyp-Submodelle möglich sind, die Kollaborationsdiagramme darstellen. In den Kollaborationsdiagrammen ist es möglich, andere Dienstmodellelemente, wie z.B. die Dienst-Zugangspunkte, Ressourcen etc., zu referenzieren.
- Ressourcen (resources) und Basic Management Functionality für die Beschreibung von internen Betriebsmitteln.
- Sub-Clients (subclients), die Sub-Diensten aus dem Basismodell zugeordnet werden. Zusätzlich zum Sub-Dienst wird der Typ (Dienst-Nutzung oder Dienst-Management) und ggf. die genaue Rolleninstanz angegeben.
- Dienstarchitektur-Spezifikationen (service architectures)
- QoS-Parameter-Konkretisierungen (QoS parameter instantiations), die für QoS-Parameter konkrete Werte festlegen. Diese werden vor allem bei der Ressourcen-Definition im Realisierungssicht Top-Down-Workflow verwendet.

Um der Verteilung des Dienstmodells innerhalb des Workflows gerecht zu werden, können Modelle des SM-Modelltyps in verschiedenen Ausprägungen, die den verschiedenen Teilen des Dienstmodells entsprechen, vorkommen. In jeder Ausprägung sind nur bestimmte Elemente erstellbar.

Weiterhin sind zur Sicherung der Datenkonsistenz Überprüfungsmechanismen von allen spezifischen Beziehungen zwischen den Dienstmodellelementen notwendig.

Außerdem werden innerhalb eines Modells des SM-Modelltyps Hyperlinks auf andere Modelle des SM-Modelltyps unterstützt, um dem Benutzer die Navigation zu erleichtern.

5.4.5 Modelltyp für Basiseinstellungen

Der Basiseinstellungen-Modelltyp dient der Festlegung von Basiseinstellungen der MNM-Dienstmethodik (siehe Abschnitt 3.3.2). Basiseinstellungen sind vor allem die Vorgehenweise (top-down oder bottom-up) und der Modellierungsfall.

Weiterhin sind mit diesem Modelltyp Festlegungen für die Rekursion der Dienstmodellierung bei Providerhierarchien definierbar. Eine bereits vorher erstellte Instanz des Dienstmodells (d.h. ein Workflowinstanz-Modell für die MNM-Dienstmodellierungsmethodik, dessen Bearbeitung bereits abgeschlossen ist) kann hierzu als Submodell in einem Basiseinstellungs-Modell referenziert werden. Zusätzlich zu der Referenz zu der vorhandenen Dienstmodell-Instanz wird festgelegt welcher Sub-Dienst aus der bereits vorhandenen Instanz in der neu erstellten Instanz als Hauptdienst modelliert wird.

5.5 Der Workflow für die MNM-Dienstmodellierung

In den vorangegangenen Abschnitten wurde allgemein erklärt, wie für das entwickelte Werkzeug Workflows definiert werden können, wie diese ablaufen und welche Artefakt-Modelltypen für den Workflow existieren. Im Weiteren soll nun auf den Aufbau des mit diesen allgemeinen Mitteln entworfenen Dienstmethodik-Workflows innerhalb des Werkzeugs, also das MNM-Dienstmethodik-Workflowspezifikations-Modell mit den zugehörigen MNM-Dienstmethodik-Workflowmoduldateien bzw. die aus dem MNM-Dienstmethodik-Workflowspezifikations-Modell generierten, ausführbaren MNM-Dienstmethodik-Workflowinstanz-Modelle, genauer eingegangen werden (vgl. Abschnitt 5.3). Das speziell für die MNM-Dienstmethodik entworfene Workflowspezifikations-Modell, nämlich das MNM-Dienstmethodik-Workflowspezifikations-Modell und die zu ihm gehörenden Workflowmoduldateien (MNM-Dienstmethodik-Workflowmoduldateien) stellen zusammen innerhalb der Architektur für das Werkzeug (siehe Abschnitt 3.4) die Methodik-Spezifikationsdaten dar. Der in den MNM-Dienstmethodik-Workflowmoduldateien enthaltene Alter-Code wird bei der Abarbeitung der Methodik in Form von MNM-Dienstmethodik-Workflowinstanz-Modellen zur Realisierung desindividuellen Verhaltens der Aktivitäten der Workflowinstanz-Modelle ausgeführt und stellt somit zusätzlich einen Teil der Methodik-Ausführungs-Komponente der Architektur dar.

Die Methodik beschreibt Workflows bestehend aus Aktivitäten. Diese sollen hier im Folgenden genauer als Methodik-Workflows und Methodik-Aktivitäten bezeichnet werden. Die Begriffe Workflow und Aktivität allein dagegen stehen für Workflows bzw. Aktivitäten innerhalb der des Workflowspezifikations- bzw. der Workflowinstanz-Modelle. In Abs 3.3 wurden die Methodik-Aktivitäten weiter in Benutzerinteraktionen zerlegt. Innerhalb des Werkzeugs, d.h. im MNM-Dienstmethodik-Workflowspezifikations-Modell bzw. in den MNM-Dienstmethodik-Workflowinstanz-Modellen, werden sowohl die Methodik-Workflows als auch deren Methodik-Aktivitäten in Form von Subworkflowschritten (mit zugehörigen Subworkflows) hierarchisch verschachtelt repräsentiert. Die Subworkflows für Methodik-Aktivitäten bestehen dann aus Aktivitäten (im Sinne des Werkzeugs wie oben angegeben). Aktivitäten sind die eigentlich auszuführenden Einheiten des Workflows, ihr spezifisches Verhalten wird durch Alter-Code in ihren Workflowmoduldateien definiert (siehe Abschnitt 5.3) Die Aufteilung der Subworkflows der Methodik-Aktivitäten in Aktivitäten wurde an Hand der in Abschnitt 3.3 durchgeführten Analyse der Benutzerinteraktionen durchgeführt. Zum Teil entsprechen Aktivitäten einzelnen Benutzerinteraktionen. Es gibt jedoch auch Aktivitäten, die mehr als eine der identifizierten Benutzerinteraktionen in sich enthalten.

Eine Instanz des Dienstmodells ist bekanntlich in drei Sichten aufgeteilt. Im Werkzeug wird jede Sicht durch eigene Artefakt-Modelle dargestellt. Jede Sicht wird dabei im wesentlichen durch ein oder mehrere Instanzen des SM-Modelltyps (siehe Abschnitt 5.4.4) dargestellt. Diese definieren die eigentlichen Dienstmodellelemente (z.B. Dienste, Rolleinstanzen, Prozesse, QoS-Parameter, Ressourcen, etc.) sowie die spezifischen Beziehungen unter diesen. Hinzu kommen jeweils Modelle für strukturierende Artefakte (Dienstlebenszyklus, etc.) sowie zur genaueren Beschreibung Modelle des UML-Modelltyps bzw. des Modelltyps für generische Beschreibungen, die von den Modellen des SM-Modelltyps genutzt werden. Jede Aktivität im MNM-Dienstmethodik-Workflowspezifikations-Modell verwaltet dazu eines oder mehrere dieser Artefakte.

Die für die Methodik hierbei entwickelten Workflowmoduldateien (MNM-Dienstmethodik-Workflowmoduldateien) und damit die ihnen zugeordneten Aktivitäten lasen sich prinzipiell in zwei Gruppen einteilen: Aktivitäten der ersten Gruppe erzeugen selbst eigene Artefakt-Modelle als Submodelle, die der zweiten Gruppe verwenden bereits vorhandene Artefakt-Modelle. Damit die Artefakt-Modelle für Aktivitäten der zweiten Gruppe auffindbar sind, speichern die Aktivitäten der ersten Gruppe die neu erstellten Artefakt-Modelle in der Regel bei der globalen Workflowartefakt-Verwaltung unter speziellen hierarchischen Schlüsselnamen ab (vgl. Abschnitt 5.3.2). Neu erstellte Artefakt-Modelle werden von der erstellenden Aktivität entsprechend initialisiert und ggf. gesteuert. Verwenden nun Aktivitäten der zweiten Gruppe vorhandene Artefakt-Modelle, so ändern sie den Status dieser Modelle entsprechend und steuern sie ggf. Abb. 5.13 zeigt eine Skizze über die Umsetzung der MNM-Methodik mit dem MNM-Dienstmethodik-Workflowspezifikations- bzw dessen MNM-Dienstmethodik-Workflowinstanz-Modellen.

Der Aufbau der Methodik-Aktivitäten aus Aktivitäten sowie die Beschreibung dieser Aktivitäten bzw. der sie steuernden Workflowmoduldateien wird nun im Einzelnen beschrieben.

Basiseinstellungen

In einem eigenen,kleinen Subworkflow vor Ablauf der Methodik-Workflows werden die Basiseinstellungen mit Hilfe einer Instanz des Modelltyps für Basiseinstellungen (siehe Abschnitt 5.4.5) festgelegt. Dieses Modell wird im Workflow unter der Schlüssel-Liste `(basic-settings main)` als Artefakt registriert.

Der Subworkflow besteht aus einer einzigen Aktivität („Basiseinstellungen (basic settings)“, [Workflowmodul „basic-settings1“]).

Wesentlich ist hier die Auswahl des Vorgehens (top-down oder bottom-up), was den genauen Ablauf der Methodik-Workflows später beeinflusst.

Basis-Modell-Workflow

In diesem Workflow wird eine erste Version des Basis-Modells erstellt. Repräsentiert wird das Basis-Modell selbst durch eine einzige Instanz des SM-Modelltyps. Diese beinhaltet eine Instanz des Modelltyps für generische Beschreibungen zur Spezifikation von Transparenzen, sowie eine Instanz des Rollen-Modelltyps für die Festlegung der Rollen. Das SM-Modell für das Basis-Modell wird in den späteren Workflows erweitert.

Die erste Methodik-Aktivität „Rollen-Identifikation (role identification)“ wird durch mehrere Aktivitäten realisiert:

- Parallel ablaufend:
  - Aktivität „Transparenzen-Spezifikation (transparencies specification)“:
    [Workflowmodul „basic0“]
    Zur Festlegung der Transparenzen von Diensten und ihren Rolleninstanzen wird hier ein Modell für generische Beschreibungen als Artefakt-Model (registriert beim Workflow unter der Schlüssel-Liste `(basic-model transparencies-specification)`) verwendet.
  - Aktivität „Rollen-Definition (role definition)“:
    [Workflowmodul „basic1“]
    Mit einer Instanz des Rollen-Modelltyps (beim Workflow registriert unter der Schlüssel-Liste `(basic-model role-definition)`) werden hier die bei den Diensten verwendeten Rollen definiert. Für jede der drei Basis-Rollen wird bereits eine generische Rolle vorgegeben.
- Aktivität „erste Version Basis-Modell (basic model first version)“:
  [Workflowmodul „basic2“]
  Hier wird das Basis-Modell selbst, d.h. die spezielle Instanz des SM-Modelltyps (registriert unter `(basic-model main)`) zumindest teilweise — soweit die Informationen schon bekannt sind — festgelegt.

Vervollständigt wird die erste Version des Basis-Modell dann in der darauffolgenden Methodik-Aktivität „Dienste-Benennung (service naming)“. Diese enthält nur eine Aktivität („Basis-Modell (basic model)“, [Workflowmodul „basic3“]). Diese nimmt das bereits erstellte Basis-Modell und gestattet bzw. überprüft seine Vervollständigung.

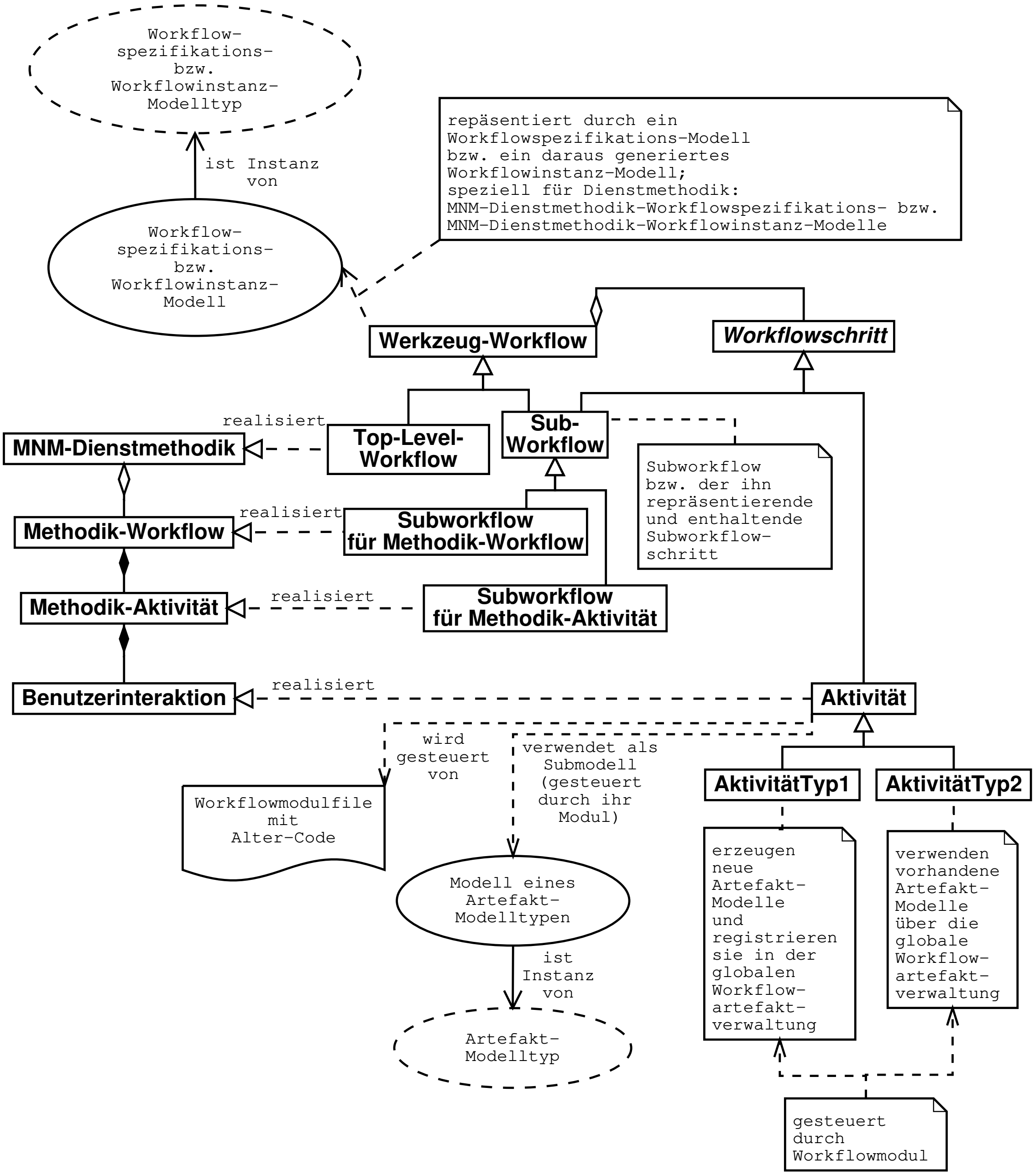

Abbildung 5.13: Skizze der Umsetzung der MNM-Methodik mit Workflowspezifikations- bzw Workflowinstanz-Modellen und darin enthaltenen Aktivitäten, die durch Workflowmodule gesteuert ihre Artefakte verwalten

Dienstsicht Top-Down-Workflow

Beim Top-Down-Vorgehen wird kundenorientiert zuerst im Dienstsicht Top-Down-Workflow die Dienstsicht erstellt und dann daraus die Realisierungssicht im Realisierungssicht Top-Down-Workflow abgeleitet.

Die Dienstsicht wird im Werkzeug in verschiedene Teile aufgegliedert die sich aus dem workflow-orientierten Vorgehen ergeben. In jeder der vier Aktivitäten des Dienstsicht Top-Down-Workflow wird einer der Teile erstellt. Jeder Teil wird durch eine Instanz des SM-Modelltyps repräsentiert.

Die erste Methodik-Aktivität „Funktionalitäts-Definition (functionality definition)" besteht aus folgenden Aktivitäten:

- Parallel ablaufend:
  - Aktivität „Kunden-Funktionalitäts-Anforderungen (customer's functionality requirements)":
    [Workflowmodul "td-sv-fnd0"]
    Dient zur Eingabe der Anforderungen des Kundens bezüglich der Dienst-Funktionalität. Diese werden durch ein Modell für generische Beschreibungen (registriert unter `(service-view customer-functionality-requirements)`) dargestellt.
  - Aktivität „Dienstlebenszyklus (service life cycle)":
    [Workflowmodul „td-sv-fnd1"]
    Hier wird der Dienstlebenszyklus durch eine entsprechende Instanz des Modelltyps für Dienstlebenszyklen festgelegt (registriert unter `(service-view lifecycle)`). Dem Benutzer wird hierbei ein Lebenszyklus mit den Phasen Design (design), Verhandlung (negotiation), Bereitstellung (provisioning). Benutzung (usage) und Deinstallation (deinstallation) vorgegeben, den er ggf. anpassen kann.
  - Aktivität „Dienstsicht-Prozessklassen (service view process classes)":
    [Workflowmodul „td-sv-fnd2"]
    Aufbauend auf der Definition des Lebenszyklus werden mit einer Modell für Prozessklassifizierungen (unter `(service-view process-classes)` registriert) die Prozessklassen der Dienstsicht spezifiziert. Auch hier wird eine grobe Einteilung in die Klassen Design (design), Vertrags-Management (contract management), Bereitstellung (provisioning), Abrechnungs-Management (accounting management), Sicherheits-Management (security management), Problem-Management (problem management), Change-Management (change management), Customer Care (customer care), Benutzung (usage), Betrieb (operation), und Deinstallation (deinstallation) inkl. der Zuordnungen zu entsprechenden Standard-Lebenszyklus-Phasen (siehe oben) bereits als Vorlage gegeben.
- Dienstsicht-Prozesse (service view processes):
  [Workflowmodul „td-sv-fnd3"]
  Mit Hilfe der vorher spezifizierten Prozessklassen werden nun im eigentlichen Modell für die Dienst-Funktionalität, einer SM-Modelltyp-Instanz (registriert unter `(service-view functionality)`), die Prozesse für Dienst-Nutzung und Dienst-Management festgelegt. Jeder Prozess wird selbst durch ein Aktivitätsdiagramm, d.h. einem UML-Modell, das ein Aktivitätsdiagramm enthält, als Submodell, beschrieben. Beziehungen zwischen den Prozessen werden in Form von Use-Case-Diagrammen, d.h. Submodellen vom UML-Modelltyp, die Use-Case-Diagramme enthalten, dargestellt.

Die Methodik-Aktivität „QoS-Definition (QoS definition)" ist mit diesen Aktivitäten realisiert:

- Parallel ablaufend:
  - Aktivität „Kunden-QoS-Anforderungen (customer's QoS requirements)":
    [Workflowmodul „td-sv-qos0"]
    Mit einem Modell für generische Beschreibungen (registriert unter `(service-view customer-qos-requirements)`) Festlegung der QoS-Anforderungen des Kunden.

- Aktivität „QoS-Dimensionen (QoS dimensions)“:
  [Workflowmodul „td-sv-qos1“]
  Spezifikation von QoS-Dimensionen mit einer Instanz des Modelltyps für QoS-Dimensionen (registriert unter `(service-view qos-dimensions)`).

- Aktivität „QoS-Parameter (QoS parameters)“:
  [Workflowmodul „td-sv-qos2“]
  Die QoS-Parameter werden in einer eigenen Instanz des SM-Modelltyps (registriert unter `(service-view qos-parameters)`) festgelegt. (Die Abbildung auf konkrete Werte erfolgt jedoch erst in der Dienstvereinbarung).

Die Methodik-Aktivität „Definition der Clients/Zugangspunkte (client/access point definition)“ besteht aus den folgenden Aktivitäten:

- Aktivität „Kunden-Client-Anforderungen (customer's client requirements)“:
  [Workflowmodul „td-sv-ap0“]
  Festlegung der Anforderungen des Kunden an die Dienst-Clients mit einer Instanz des Modelltyps für generische Beschreibungen (registriert unter `(service-view customer-client-requirements)`).

- Aktivität „Dienst-Client/Zugangspunkt-Definition (service client/access point definition)“:
  [Workflowmodul „td-sv-ap1“]
  Die Spezifikation der Dienst-Clients und Dienst-Schnittstellen selbst sowie ihre Zuordnung zueinander erfolgt hier in einer eigenen Instanz des SM-Modelltyps (registriert unter `(service-view access-point-specification)`).

In der letzten Methodik-Aktivität „Definition der Dienstvereinbarung (service agreement definition)“ wird der modellierte Dienst genauer konkretisiert. Hierfür werden z.B. für QoS-Parameter Abbildungen auf konkrete Werte gemacht. Die hierfür notwendigen Konkretisierungs-Informationen werden in einer eigener Instanz des SM-Modelltyps (registriert unter `(service-view service-agreement)`) durch eine legale Rahmenvereinbarung bzw. SLA-Spezifikationen (siehe Abschnitt 5.4.4) repräsentiert. Die Eingabe dafür erfolgt in einer einzigen Aktivität („Dienstvereinbarungs-Definition (service agreement definition)“,[Workflowmodul „td-sv-agreement1“]).

Realisierungssicht Top-Down-Workflow

Die vorher erstellte Dienstsicht wird im Realisierungssicht Top-Down-Workflow zu zur Realisierungssicht verfeinert.

Die Realisierungssicht wird hierbei dargestellt durch mehrere Instanzen vom SM-Modelltyp, die ergänzt werden durch Modelle für Strukturierende Artefakte und Modelle für UML oder generische Beschreibung zur genaueren Beschreibung. Aufgeteilt ist die Realisierungssicht hier in Prozess-Beschreibung (inkl. Aktivitäts, Use-Case- und Kollaborationsdiagramme), Beschreibung von internen und externen Ressourcen, Spezifikation der Dienstarchitektur und Abbildung der QoS-Parameter der Dienstsicht auf Werte, die durch die Basic Management Functionality messbar bzw. beobachtbar sind.

Speziell die Beschreibungen für Prozesse und externe bzw. interne Ressourcen werden hierbei nicht auf einmal spezifiziert, sondern in mehreren Aktivitäten schrittweise festgelegt. Auch das im Basis-Modell-Workflow definierte Basis-Modell wird hier erweitert.

Die Methodik-Aktivität „Provider-SM-Prozess (provider's SM process)“ wird durch diese Aktivitäten realisiert:

- Aktivität „Realisierungssicht-Prozess-Klassen (realization view process classes)“:
  [Workflowmodul „td-rv-sm1“]
  Spezifikation der Prozessklassen der Realisierungssicht mit einer Instanz des Modelltyps für Prozes-

sklassifizierungen (registriert unter `(realization-view process-classes)`) Die Prozessklassen können hierbei aus der Prozessklassifizierung der Dienstsicht übernommen werden.

- Aktivität „Realisierungssicht-Prozesse (realization view processes)“:
  [Workflowmodul „td-rv-sm2“]
  Definition der Prozesse, strukturiert nach den vorher festgelegten Klassen, in einer Instanz des SM-Modelltyps (registriert unter `(realization-view processes)`). Die Prozesse der Dienstsicht können hierbei übernommen werden und die Beschreibungen der Prozesse durch Aktivitäts- und Use-Case-Diagramme entsprechend erweitert werden. Diese SM-Modell für Prozesse wird in den beiden folgenden Aktivitäten erweitert.

Die Methodik-Aktivität „Outsourcing-Entscheidungen (outsourcing decisions)“ gliedert sich in folgenden Aktivitäten:

- Parallel ablaufend:
  - Aktivität „Basis-Modell-Erweiterung (basic model extension)“:
    [Workflowmodul „td-rv-out1“]
    Erweiterung des Basis-Modells, d.h. des SM-Modells, das das Basis-Modell beschreibt, um neue Sub-Dienste mit zugehörigen Rolleninstanzen und neue Entitäten.
  - Aktivität „Prozess-Spezifikations-Erweiterung (process specification extension)“:
    [Workflowmodul „td-rv-out2“]
    Erweiterung der Prozess-Beschreibungen, d.h. des SM-Modells, das die Prozesse beschreibt; Dabei Auswahl von ausgelagerten Prozessen und Zuordnung zu Sub-Diensten.
  - Aktivität „Sub-Client-Spezifikation (subclient specification)“:
    [Workflowmodul „td-rv-out3“]
    zur Spezifikation von Sub-Clients und Zuordnung zu Sub-Diensten und ggf. Rolleninstanzen; Als Zugangspunkte zu externen Ressourcen werden Sub-Clients zusammen mit internen Ressourcen in einer eigenen Instanz des SM-Modelltyps (registriert unter `(realization-view all-ressources)`) beschrieben. In dieser Aktivität können hier nur die Sub-Clients und noch keine internen Ressourcen festgelegt werden. Die internen Ressourcen werden erst in der nächsten Aktivität hinzugefügt.

Die letzte Methodik-Aktivität „Ressourcenidentifikation (resource identification)“ zerfällt in folgenden Aktivitäten:

- Aktivität „Ressourcenidentifikation (resource identification)“:
  [Workflowmodul „td-rv-ress1“]
  Erweiterung des SM-Modells für Ressourcen um die internen Ressourcen inkl. Basic Management Functionality.
- Parallel ablaufend:
  - Aktivität „Prozess-Spezifikation mit Kollaborationsdiagrammen (process specification by collaboration diagrams)“:
    [Workflowmodul „td-rv-ress2“]
    Beschreibung der Prozesse mit Kollaborationsdiagrammen, d.h. Erweiterung des SM-Modells für Prozesse. Jeder Prozess wird nun zusätzlich durch ein Kollaborationsdiagramm, d.h. ein Submodell vom UML-Modelltyp, das ein Kollaborationsdiagramm enthält, spezifiziert.
  - Aktivität „Spezifikation der Dienstarchitektur (specification of service architecture)“:
    [Workflowmodul „td-rv-ress3“]
    Festlegung der Dienstarchitektur mit einer eigenen Instanz des SM-Modells (registriert unter `realization-view service-architecture)`).
  - Aktivität „QoS-Parameter-Konkretisierung (QoS parameter instantiation)“:
    . [Workflowmodul “td-rv-ress4“]

Abbilden der QoS-Parameter auf mess- bzw. beobachtbare Werte innerhalb einer Instanz des SM-Modelltyps (registriert unter `(realization-view qos)`).

Realisierungssicht Bottom-Up-Workflow

Beim Bottom-Up-Vorgehen wird im Gegensatz zum Top-Down-Vorgehen zunächst die Realisierungssicht im Realisierungssicht Bottom-Up-Workflow entworfen und daraus im Dienstsicht Bottom-Up-Workflow die Dienstsicht als Abstraktion für den Kunden gewonnen.

Die Darstellung der Realisierungssicht innerhalb des Werkzeuges ist ähnlich wie Top-Down-Vorgehen. Hier gliedert sich die Realisierungssicht in die Teile Prozess-Spezifikation, Beschreibung interner und externer Ressourcen, Spezifikation des Dienstzugangspunktes, Spezifikation der Dienstarchitektur. Jeder Teil wird ebenfalls durch eine Instanz des SM-Modelltyps (inkl. zusätzlichen Submodellen) dargestellt.

Die erste Methodik-Aktivität „Use-Case-Identifikation (use case identification)“ setzt sich folgendermaßen zusammen:

- Parallel ablaufend:
  - Aktivität „Provider-Anforderungen (provider's requirements)“:
    [Workflowmodul „bu-rv-uc0“]
    Spezifikation von Anforderungen des Providern in Form einer Instanz des Modelltyps für generische Beschreibungen (unter `(realization-view provider-requirements)` registriert).
  - Aktivität „Realisierungssicht-Prozessklassen (realization view process classes)“:
    [Workflowmodul "bu-rv-uc1"]
    Festlegung von Prozessklassen mit einer Instanz des Prozessklassifizierungs-Modelltyps. (registriert unter `(realization-view process-classes)`). Hierbei Vorgabe wie beim Dienstsicht Top-Down-Workflow.
- Aktivität „Use-Case-Identifikation (use case identification)“:
  [Workflowmodul „bu-rv-uc2“]
  Festlegung der Prozesse in einer Instanz des SM-Modelltyps (registriert unter `(realization-view processes)`). Beziehungen zwischen den Prozessen werden durch Use-Case-Diagramme, d.h. Submodelle vom UML-Modelltyp, die Use-Case-Diagramme darstellen, beschrieben. Diese Prozess-Beschreibungen werden später für jeden Prozess einzeln durch Kollaborationsdiagramme ergänzt.

Die nächste Methodik-Aktivität „Client, Ressourcen/BMF-Identifikation (client, resources/BMF identification)“ enthält die folgenden Aktivitäten:

- Parallel ablaufend:
  - Aktivität „Basis-Modell-Erweiterung (basic model extension)“:
    [Workflowmodul „bu-rv-ress1“]
    Erweiterung des Basis-Modells, d.h. des SM-Modells, das das Basis-Modell beschreibt, um neue Sub-Dienste mit zugehörigen Rolleninstanzen und neue Entitäten.
  - Aktivität „Ausgelagerte Prozesse identifizieren (outsourced processes identification)“:
    [Workflowmodul „bu-rv-ress2“]
    Bei den Prozess-Beschreibungen, Auswahl von ausgelagerten Prozessen und Zuordnung zu Sub-Diensten.
  - Aktivität „Ressourcen/BMF und Sub-Clients (resources, BMF and subclients)“:
    [Workflowmodul „bu-rv-ress3“]
    Spezifikation von externen und internen Ressourcen, d.h. Sub-Clients, Ressourcen und

Basic Management Functionality in einer Instanz des SM-Modelltyps (registriert unter `(realization-view all-ressources)`).

Die letzte Methodik-Aktivität „Logik-Definition (logic definition)“ ist wie folgt aufgebaut:

- Parallel ablaufend:
  - Aktivität „Prozess-Spezifikation durch Kollaborationsdiagramme (process specification by collaboration diagrams)“:
    [Workflowmodul „bu-rv-logic1“]
    Beschreibung der Prozesse mit Kollaborationsdiagrammen, d.h. Erweiterung des SM-Modells für Prozesse. Jeder Prozess wird nun zusätzlich durch ein Kollaborationsdiagramm, d.h. ein Submodell vom UML-Modelltyp, das ein Kollaborationsdiagramm enthält, spezifiziert.
  - Aktivität „Dienstzugangspunkt-Spezifikation (service access point specification)“:
    [Workflowmodul „bu-rv-logic2“]
    Festlegung der Zugangspunkte für den Dienst mit einer eigenen Instanz des SM-Modells (registriert unter `(realization-view access-point-specification)`).
  - Aktivität „Dienstarchitektur-Spezifikation (service architecture specification)“:
    [Workflowmodul „bu-rv-logic3“]
    Festlegung der Dienstarchitektur mit einer eigenen Instanz des SM-Modells (registriert unter `(realization-view service-architecture)`).

Dienstsicht Bottom-Up-Workflow

Nach Erstellung der Realisierungssicht wird daraus unter Abstraktion von provider-internen Details die Dienstsicht gewonnen.

Dargestellt wird die Dienstsicht sehr ähnlich wie im Top-Down-Fall. Sie besteht – gemäß den Workflowaktivitäten – aus vier Teilen, von denen jeder in einer eigenen Instanz des SM-Modelltyps beschreiben wird.

Die Aktivitäten der Methodik-Aktivität „Funktionalitäts-Definition (functionality definition)“ sind:

- Parallel ablaufend:
  - Aktivität „Dienstlebenszyklus (service life cycle)“:
    [Workflowmodul „bu-sv-fnd1“]
    Festlegung des Dienstlebenszyklus durch eine entsprechende Instanz des Modelltyps für Dienstlebenszyklen festgelegt (registriert unter `(service-view lifecycle)`); Vorgaben wie beim Top-Down-Vorgehen.
  - Aktivität „Dienstsicht-Prozessklassen (service view process classes)“:
    [Workflowmodul „bu-sv-fnd2“]
    Auf Basis des Lebenszyklus werden mit einer Modell für Prozessklassifizierungen (unter `(service-view process-classes)` registriert) die Prozessklassen der Dienstsicht spezifiziert. Hierbei können Prozessklassen der Realisierungssicht übernommen werden.
- Aktivität „Dienstsicht-Prozesse (service view processes)“:
  [Workflowmodul „bu-sv-fnd3“]
  Festlegung der Prozesse der Dienstsicht mit einer SM-Modelltyp-Instanz (registriert unter `(service-view functionality)`). Die Prozesse der Realisierungssicht können übernommen und angepasst werden. Jeder Prozess wird nun durch ein Aktivitätsdiagramm, d.h. einem UML-Modell, das ein Aktivitätsdiagramm enthält, als Submodell, beschrieben. Beziehungen zwischen den Prozessen werden in Form von Use-Case-Diagrammen, d.h. Submodellen vom UML-Modelltyp, die Use-Case-Diagramme enthalten, dargestellt.

Die Methodik-Aktivität „QoS-Definition (QoS definition)“ enthält folgenden Aktivitäten:

- Aktivität „QoS-Dimensionen (QoS dimensions)“:
  [Workflowmodul „bu-sv-qos1“]
  Spezifikation von QoS-Dimensionen mit einer Instanz des Modelltyps für QoS-Dimensionen (registriert unter `(service-view qos-dimensions)`).

- Aktivität „QoS-Parameter (QoS parameters)“:
  [Workflowmodul „bu-sv-qos2“]
  Die QoS-Parameter werden in einer eigenen Instanz des SM-Modelltyps (registriert unter `(service-view qos-parameters)`) festgelegt.

In der Methodik-Aktivität „Definitions der Dienst-Zugangspunkte (client/access point definition)“ werden nur die in der Realisierungssicht definierten Dienstzugangspunkte in die Dienstsicht kopiert. Hierfür gibt es eine einzige Aktivität (Aktivität „Dienst-Zugangspunkt-Definition (service access point definition)“,[Workflowmodul „bu-sv-ap1“]), bei der in eine Instanz des SM-Modelltyps (registriert unter `(service-view access-point-specification)`) die Zugangspunkt-Spezifikation übernommen wird. Dieses Modell wird in der nächsten Aktivität um entsprechende Clients erweitert.

Die abschließende Methodik-Aktivität „Dienstvereinbarungs-Definition (service agreement definition)“ hat zwei Aktivitäten:

- Parallel ablaufend:

- Aktivität „Spezifikation der Dienst-Clients (service client specification)“:
  [Workflowmodul „bu-sv-agreement1“]
  Erweiterung des SM-Modells mit den Dienstzugangspunkten um passende Clients.

- Aktivität „Dienstvereinbarungs-Definition (service agreement definition)“:
  [Workflowmodul „bu-sv-agreement2“]
  Festlegung von den Dienst näher konkretisierenden Informationen innerhalb einer eigenen Instanz des SM-Modelltyps (registriert unter `(service-view service-agreement)`) wie beim Top-Down-Vorgehen.

5.6 Zusammenfassung

In diesem Kapitel wurde zunächst *DoME* genauer betrachtet. Anschließend wurde für den Entwurf der Workflowsteuerung auf die Struktur (d.h. Definition und Erstellung) und auf die allgemeine Ausführung (Aktivitäten und Artefakte allgemein) von Workflows innerhalb des Werkzeugs näher eingegangen. Schließlich wurde der Entwurf der unterschiedlichen Workflowartefakt-Modelltypen, die für die Workflows der MNM-Dienstmodellierung benötigt werden, betrachtet. Hiermit wurde abschließend der Entwurf der Workflows für die der MNM-Dienstmodellierung selbst besprochen.

Im nächsten Kapitel wird auf die prototypische Implementierung des Werkzeugs eingegangen.

Kapitel 6

Prototypische Implementierung

In diesem Kapitel wird die protypische Implementierung des Werkzeuges beschrieben. Hierfür wird in Abschnitt 6.1 zunächst auf die für die Implementierung getroffenen Datei- und Namenskonventionen eingegangen und die Verknüpfung von Metamodell und Alter-Code genauer erläutert. Dann wird auf die Implementierung der einzelnen, entwickelten Modelltypen eingegangen. Zunächst werden in Abschnitt 6.2 die Modelltypen für Spezifikation und Ausführung von Workflows betrachtet. Anschließend wird in Abschnitt 6.3 die Implementierung der Modelltypen für Artefakte besprochen. Zuletzt wird in Abschnitt 6.4 die Realisierung des MNM-Dienstmodellierung-Workflows beschrieben.

Über die Implementierung wird in diesem Kapitel nur ein Überblick gegeben. Detailliertere Informationen zur Implementierung, die auch eine Anpassung des Werkzeugs an zukünftige Änderungen der Methodik unterstützen, finden sich in Anhang A.

6.1 Implementierung der Modelltypen allgemein

In diesem Abschnitt wird das allgemeine Vorgehen zur Implementierung der verschiedenen entworfenen Modelltypen für das Werkzeug beschrieben. In Abschnitt 6.1.1 werden die Konventionen zu Datei- und Verzeichnisstrukturen, sowie andere Namenskonventionen von *DoME* allgemein erklärt. Darauf aufbauend werden in Abschnitt 6.1.2 die für die Implementierung des Werkzeuges getroffenen Datei- und Namenskonventionen beschrieben. In Abschnitt 6.1.3 wird dann das verwendete Konzept zur Verknüpfung der Metamodelle und der Alter-Codefiles zur tatsächlichen Realisierung der Modelltypen selbst erklärt.

6.1.1 Datei- und Namenskonventionen in DoME allgemein

Alle Modelle, sowohl Metamodelle als auch die Instanzen von Metamodellen werden von *DoME* in speziell formatierten ASCII-Dateien gespeichert. Darin ist die gesamte Information über das Modell (hierarchisch mit enthaltenen Submodellen) mit allen darin enthaltenen Graphobjekten und dazugehörigen Properties enthalten.

In *DoME* enthält das sogenannte **Tool-Verzeichnis** standardmäßig jeweils pro Modelltyp ein Unterverzeichnis, das in der Regel nach dem Modelltyp benannt ist. Ein solches **modelltyp-spezifisches Verzeichnis** enthält eine Reihe von weiteren Unterverzeichnissen, die das zugehörige Metamodell, ggf. verwendeten Alter-Code und andere modelltyp-spezifische Informationen beinhalten. Metamodelle von protodome-basierten Modelltypen befinden sich hierbei im Unterverzeichnis `specs/`. Metamodelle von Smalltalk-basierten Modelltypen sind im Unterverzeichnis `dev/specs/` abgelegt. Alter-Code Ist im Unterverzeichnis `lib/` zu finden. Einige grundsätzliche Einstellungen wie z.B. für Dokumentgenerierung werden in einigen Dateien im Unterverzeichnis `etc/` getroffen. Darüber hinaus gibt es vor allem Unterverzeichnisse für Hilfetexte (`help/` und `doc/`).

Tatsächlich ordnet *DoME* die modellspezifischen Verzeichnisse aber den in ihnen definierten Modelltypen nicht explizit einander zu, sondern durchsucht einfach das gesamte Tool-Verzeichnis. Zum Beispiel eine Library, die Alter-Code enthält, wird in allen `lib/`-Unterverzeichnissen aller modellspezifischen Unterverzeichnisse gesucht, Protodome-basierte Modelle werden durch die Durchsuchung der `specs/`-Unterverzeichnisse aller modellspezifischen Unterverzeichnisse gefunden. Die tatsächliche Zuordnung zwischen Modelltyp und Position einer Datei innerhalb des Tool-Verzeichnisses muss in aller Regel durch den Filenamen (z.B. bei Alter-Codefiles) oder ggf. im Inhalt einer Datei (z.B. Dateien in den modelltypspezifischen `etc/`-Unterverzeichnissen) festgelegt werden.

Um in *DoME* die verschiedenen Modelltypen geeignet zu unterscheiden, werden pro Modelltyp eigene Klassen für das Modell (den Graphen) selbst und alle zugehörigen Graphobjekte definiert. Hierfür wird im Metamodell jedes Modelltyps ein **Klassennamenpräfix** definiert. Der Klassenname des Modells selbst ist dann das Klassennamenpräfix konkateniert mit dem Namen `Graph`. Die Klassennamen der Graphobjekte ergeben sich aus dem Klassennamenpräfix erweitert um den im Modellmodell festgelegten, spezifischen Namen des Graphobjekts. Für Smalltalk-basierte Modelltypen sind diese Klassen Smalltalk-Klassen, die direkt in Smalltalk mit zusätzlichen Verhalten erweitert werden können. Für protodome-basierte Modelle stellen diese Klassen Alter-Klassen dar. Innerhalb von Smalltalk sind das Modell bzw. die Graphobjekte Instanzen der Smalltalk-Klassen `ProtoDoMEGraph`, `ProtoDoMENode`, `ProtoDoMEArc` und `ProtoDoMEElement`, also unabhängig vom Metamodell, fest vorgegebene Smalltalk-Klassen. Bei Klassennamen in Smalltalk ist die Groß/Kleinschreibung wesentlich, bei Klassennamen in Alter dagegen ist sie unerheblich.

Alter kennt ein Namensraum-Konzept. Die Namensräume heißen **Pakete**. Sie sind vor allem dazu geeignet Namenskonflikte zwischen verschiedenen Modelltypen zu verhindern. Hierfür wird in einem Metamodell (für einen protodome-basierten Modelltyp) der Paket-Name angegeben. Die oben beschriebenen Klassennamen sind innerhalb dieses Paketes sichtbar. Weiterer Alter-Code, der genaueres Verhalten der Klassen für den jeweiligen Modelltyp spezifiziert, kann dann auch innerhalb dieses Paketes definiert werden. In Alter sind auch die Namen von Paketen nicht von der Groß/Kleinschreibung abhängig.

6.1.2 Datei- und Namenskonventionen für das entwickelte Werkzeug

Das Werkzeug wurde als eine Reihe von protodome-basierten Modelltypen entwickelt. Dabei wurde für alle erstellten Dateien ein eigenes Unterverzeichnis namens `mnm/` im Tool-Verzeichnis von *DoME* benutzt. Wie bei *DoME* üblich wurden alle erstellten Metamodelle im `specs/`-Unterverzeichnis (also relativ zum Tool-Verzeichnis `mnm/specs/`) und jedes Alter-Codefile im `lib/`-Unterverzeichnis (also insgesamt `mnm/lib/`) abgelegt.

Da *DoME* keine explizite Zuordnung der modellspezifischen Unterverzeichnisse im Tool-Verzeichnis durchführt, wurden, um Namenskollisionen mit anderen Modelltypen zu vermeiden, Die Dateinamen jeweils mit einem stets verwendeten Präfix versehen. Die Dateien der Metamodelle erhielten alle das Präfix „MNM“, Alter-Codefiles das Präfix „mnm-“.

Das Klassennamenpräfix eines erstellten Metamodells wurde grundsätzlich konsistent zum Dateinamen des Metamodells gewählt. Der Modelltyp, der z.B. in der Datei `MNMWf.met` gespeichert wurde, erhielt das Präfix `MNMWf` (in *DoME* wird die Datei-Endung „met“ typischerweise für Metamodelle verwendet). Da die Dateinamen aller erstellten Modelltypen den Präfix „MNM“ haben, haben auch die Klassennamenpräfixe „MNM“ als Präfix.

In der Regel wurde für das Werkzeug in Alter das Paket (Namensraum) `mnmservice` verwendet. Das bedeutet, dass der Namensraum aller erstellten Modelltypen diesen Wert hat. Somit sind in den verschiedenen Modelltypen für das Werkzeug gegenseitig die Definitionen für den Modelltyp (Klassennamen, Operationen etc.) sichtbar und Konflikte mit anderen Modelltypen werden vermieden.

Pro Modelltyp gibt es ein Alter-Codefile, das sogenanntes Alter-Supportfile, dessen Name im Metamodell spezifiziert ist. Der Name des Supportfiles setzt sich dabei stets zusammen aus dem Namen des Modelltyps in Kleinbuchstaben (und evtl. darin zur Strukturierung eingeschobene „-“-Zeichen) und dem Suffix „-support.lib“. Ein Supportfile enthält in der Regel nur Anweisungen zum Laden von weiteren Alter-Codefiles, in denen die tatsächlichen Alter-Definitionen gemacht werden.

Unter diesen Alter-Codefiles gibt es generische, d.h. modelltyp-unabhängige, und modelltyp-spezifische. In der Regel werden die generischen Alter-Codefiles von allen definierten Modelltypen durch deren Supportfile eingebunden. Eine Liste aller entwickelten, generischen Alter-Codefiles findet sich in Anhang A.2.

Die meisten Modelltypen haben nur ein modelltyp-spezifisches Alter-Codefile, dieses hat den gleichen Namen wie das Supportfile, allerdings mit dem Suffix „-general.lib“ statt „-support.lib“.

Alter-Codefiles, die als Präfix „mnm-wf“ haben, sind spezifisch für Modelltypen, die die Workflowsteuerung des Werzeugs bereitstellen. Alter-Codefiles die mit „mnm-uml-“ beginnen, sind spezifisch für die UML-Unterstützung des Werkzeugs.

6.1.3 Verknüpfung von Metamodellen und Alter-Codefiles

Die Spezifikation eines protodome-basierten Modelltyps besteht aus zwei Komponenten: dem Metamodell und Alter-Code. Mit dem Metamodell wird die grundsätzliche Struktur eines Modells des spezifizierten Modelltyps, d.h. die Typen von Knoten und Kanten und mögliche Beziehungen zwischen diesen in dem Modell, angegeben. Um aber die Darstellung und das Verhalten des Modells selbst, sowie der verschiedenen Knoten- und Kantentypen genauer festzulegen wird zusätzlich Alter-Code verwendet (siehe Abschnitt 5.1.1).

Der Alter-Code für den Modelltyp wird hierbei in Form von sogenannten **Protodome-Methoden** pro Graphobjekttyp (Knoten- bzw. Kantentyp) festgelegt. Es gibt jeweils für verschiedene Aspekte eines Graphobjektes Protodome-Methoden. Grob einteilen kann man sie in Methoden für die Anpassung der Darstellung des Graphobjekts, Methoden für Constraint-Checking und Methoden für die Definition von zusätzlichem Verhalten bei Erstellung oder Löschung eines Graphobjektes des festgelegten Graphobjekttyps.

Ein Beispiel für eine Protodome-Methode ist die „Protodome-Methode für die Erstellungs-Post-Aktion“ eines Knotens. Mit ihr kann man pro Knotentyp festlegen, was in einem Modell bei Erstellung eines Knotens des entsprechenden Typs als zusätzliche Aktion durchgeführt werden soll. Zum Beispiel können dabei der Knoten selbst oder Beziehungen zu anderen Graphobjekten entsprechend initialisiert werden. Diese Protodome-Methode stellt somit eine Art Konstruktor für den Knoten dar. Auch für jeden Kantentyp, sowie für das Modell selbst existieren derartige Protodome-Methoden zur Initialisierung. Um zusätzliches Verhalten bei Löschung zu spezifizieren, gibt es ebenfalls Protodome-Methoden.

Aber auch zur genaueren Regelung der Darstellung von Knoten und Kanten existieren Protodome-Methoden. So gibt es z.B. die „Protodome-Methode für die Berechnung des Objekt-Label-Textes“ eines bestimmten Knotentyps. Mit ihr kann die Knotenbeschriftung als beliebig berechneter Text vorgegeben werden. Andere Protodome-Methoden existieren um die Form und das genaue Aussehen von Knoten und Kanten der verschiedenen Typen festzulegen (für nähere Information siehe Anhang A.1.3).

Ein protodome-basierter Modelltyp wird also entwickelt, indem man zunächst in einem Metamodell Knoten- und Kantentypen und prinzipielle Beziehungen zwischen diesen festlegt und dann für jeden dieser Knoten- und Kantentypen Protodome-Methoden für die verschiedenen Aspekte spezifiziert. Diese Festlegung der Protodome-Methoden erfolgt selbst im Metamodell. Der hierbei festgelegt Alter-Code kann jedoch beliebigen Code, d.h. auch Alter-Code, der außerhalb des Metamodells definiert ist, aufrufen. Bei Modelltypen mit vielen Knoten- und Kantentypen ist diese Art der Codespezifikation aber sehr unübersichtlich bzw. schwer überschaubar, zumal das Metamodell selbst ein Graph ist und als solcher vom Metamodell-Entwickler grafisch erstellt wird und dabei der Code für Protodome-Methoden in eigenen Edit-Fenstern eingegeben wird.

Bei der Implementierung der Modelltypen für das Werkzeug, wurde daher der eigentliche Code für die Protodome-Methoden nicht innerhalb des Metamodells selbst, sondern stets außerhalb, in Alter-Codefiles festgelegt: Der Code im Metamodell für die jeweiligen Protodome-Methoden ist stets ein nur einziger, standardisierter (unabhängig vom Modelltyp) Wrapper-Aufruf von Code, der in Alter-Codefiles (abhängig vom Modelltyp) definiert ist.

So wird z.B. für die „Protodome-Methode für Erstellungs-Post-Aktion“ bei jedem Knoten- und Kantentyp als Code der standardisierte Aufruf `(creation-cleanup self)` verwendet. Dieser ruft die Alter-Operation `creation-cleanup` (self ist dabei als Argument der neu erstellte Knoten bzw. die neu erstellte Kante) auf, die in den Alter-Codefiles pro Knoten- und Kantentyp implementiert ist.

Genauso wird für die „Protodome-Methode für Berechnung des Objekt-Label-Textes“ eines Knotens für alle Knotentypen stets der standardisierte Aufruf `(layout-object-label-text self)` benutzt. Für alle Knotentypen ist die hierbei aufgerufene Alter-Operation `layout-object-label-text` eigens in den Alter-Codefiles implementiert.

Für jede in *DoME* vorhandene Protodome-Methode wurde jeweils ein standardisierter Alter-Operationsaufruf definiert, der als Code für die Protodome-Methoden im Metamodell verwendet wird. Weitere Informationen über den standardisierten Code für Protodome-Methoden und die dafür entwickelten Alter-Operationen finden sich in Anhang A.3.

Die Protodome-Methoden — vor allem die für Erstellungs-Post-Aktion und die für das Constraint-Checking — bieten wichtige Initialisierungs- und Synchronisationsmöglichkeiten innerhalb eines Modells. Die Ausführung des Codes ist hierbei aber statisch auf ganz bestimmte Zeitpunkte bezüglich des Graphobjekts, für das die Protodome-Methode aufgerufen wird, z.B. auf dessen Erstellungszeitpunkt, festgelegt. Zusätzlich zu Protodome-Methoden gibt es in *DoME* daher noch die sogenannten Graphobjekt-Interest-Handler. Mit ihnen können dynamisch Code-Handler festgelegt werden, die bestimmten Code dann aufrufen, wenn sich Aspekte eines vorher bestimmten Graphobjekts ändern. Hierzu kann ein Graphobjekt bei einem anderen Graphobjekt zu einem bestimmten Aspekt einen Code-Handler einrichten oder auch wieder aufheben. Das Einrichten solcher Code-Handler wird meist innerhalb der Ausführung der „Protodome-Methode für Erstellungs-Post-Aktion“ durchgeführt. (für weitere Informationen siehe Anhang A.1.4, Anhang A.6.7).

6.2 Workflow- und Artefaktsteuerung allgemein

In diesem Abschnitt wird auf die Implementierung der in Abschnitt 5.3 entworfenen Workflowsteuerung für das Werkzeug eingegangen. Hierbei werden Workflows allgemein betrachtet. Die Implementierung des Workflows für die MNM-Dienstmodellierungsmethodik wird dagegen in Abschnitt 6.4 behandelt.

Innerhalb des Werkzeugs werden beliebige Workflows mit Hilfe von Workflowspezifikations-Modellen spezifiziert und als Workflowinstanz-Modelle, die aus einem gegebenen Workflowspezifikations-Modell generiert werden, abgearbeitet. Workflows bestehen dabei aus Subworkflowschritten, die hierarchisch jeweils einen Subworkflow beinhalten, und Aktivitäten, die die eigentlich ausführbaren Einheiten im Workflow sind. Subworkflowschritte und Aktivitäten, die gemeinsam als Workflowschritte bezeichnet werden, sind dabei durch Transitionen verbunden.

Die Abarbeitung eines Workflows als Workflowinstanz-Modell beinhaltet die Navigation über den Workflow (für den Benutzer), die Iteration des Workflowbearbeitungsstatus (noch nicht bearbeitbar/bearbeitbar/abgeschloßen) für jeden Workflowschritt und jeden Subworkflow sowie die Ausführung der Workflowaktivitäten (gemeinsam mit dem Benutzer). Das individuelle Verhalten einer Aktivität wird jeweils in einem eigenen Alter-Codefile, der sogenannten Workflowmoduldatei oder kurz Workflowmodul, durch Alter-Code festgelegt.

Innerhalb der Workflows können außerdem bei der Workflowspezifikation sogenannte Hyperlink-Knoten für Workflowartefakte platziert werden, die dem Benutzer eine erleichterte Navigation zu Workflowartefakten erlauben.

Für die allgemeine Workflowrealisierung innerhalb des Werkzeugs wurden also zwei Modelltypen implementiert: der Workflowspezifikations-Modelltyp und der Workflowinstanz-Modelltyp.

Das Metamodell für Workflowspezifikationen ist in der Datei `MNMWfSpec.met` im *DoME*-Tool-Verzeichnis gespeichert und verwendet das Klassennamenpräfix `MNMWfSpec`. Das Metamodell für Workflowinstanzen ist in der Datei `MNMWf.met` festgelegt und benutzt das Klassennamenpräfix `MNMWf`.

Der Alter-Code für den Workflowspezifikations-Modelltyp ist vollständig in der Datei `mnm-wf-spec-general.lib` enthalten. Für den Workflowinstanz-Modelltyp war wesentlich mehr Alter-Code erforderlich, vor allem um die tatsächliche Ausführung eines Workflow zu unterstützen. Dieser Code befindet sich in den Codefiles `mnm-wf-general.lib`, `mnm-wf-iteration.lib`, `mnm-wf-artifact.lib`, `mnm-wf-module.lib`, `mnm-wf-activity.lib` und `mnm-wf-activity-utilities.lib`. Vor allem musste dabei für die Workflowmodul-Unterstützung (für das individuelle Verhalten von Aktivitäten) eine geeignete Verhaltensspezifikations-Schnittstelle definiert werden.

6.2.1 Realisierung der Workflowmodule

Das tatsächliche Verhalten der Aktivitäten des Workflows wird in den Workflowmoduldateien in Form von Alter-Code festgelegt. Im Workflowspezifikations-Modell werden die Workflowmodule den Aktivitäten zugeordnet. Für jede Aktivität ist hierfür der Name eines Workflowmoduls festlegbar. Die Namen der Alter-Codefiles für Workflowmodule setzen sich wie folgt zusammen: Das Präfix „mnm-wfm-“, gefolgt von dem eigentlichen Namen des Moduls, der im Workflowspezifikations-Modell festgelegt ist und die Endung „.lib“. In diesen Codefiles werden speziell festgelegte Alter-Prozeduren definiert. Diese stellen die Schnittstelle zur Definition des Verhaltens der zugehörigen Aktivität dar. Um dabei verschiedene Workflowmodule zu unterscheiden wird jeweils ein eigenes Alter-Paket (ein eigener Namensraum) verwendet. Der Name des Pakets besteht hierbei aus dem Präfix „MNMService-wfm-“ und dem eigentlichen Modulnamen.

Vor allem muss das Verhalten der Aktivität definiert werden, wenn sich der Bearbeitungsstatus der Aktivität ändert. Ein Beispiel hierfür ist die Prozedur `action-editable`, die aufgerufen wird, wenn eine Aktivität editierbar wird. Es gibt für jeden Bearbeitungsstatus-Wert eine zugehörige Prozedur, die ausgeführt wird wenn der jeweilige Wert erreicht wird. Auch existieren Prozeduren für die Initialisierung bei Erzeugung des Workflows sowie Möglichkeiten zur Festlegung von seitens der Aktivität, ob sie abgeschlossen werden kann oder nicht. Die Liste aller definierten Alter-Prozeduren, die in einer Workflowmoduldatei zu spezifizieren sind, ist in Anhang A.5.1 zu finden. Einige Hilfs-Operationen, die vor allem innerhalb der Workflowmodul-Prozeduren verwendet werden können, sind in der Datei `mnm-wf-activity-utilities.lib` definiert.

6.2.2 Workflowartefaktverwaltung

Die Aktivitäten haben in ihren Modulprozeduren vor allem die Möglichkeit beliebige Submodelle als Artefakte zu erschaffen und zu verwalten. Der Workflowinstanz-Modelltyp unterstützt hierfür eine globale Artefaktverwaltung. Mit ihr können Aktivitäten (oder Submodelle davon) beim Workflowinstanz-Modell Artefakte (z.B. ihre Submodelle oder andere beliebige Alter-Objekte) unter bestimmten Schlüsselnamen abspeichern (bzw. referenzieren) oder diese abfragen, sowie Interesse bei Änderung dieser registrierten Artefakte anmelden. Die Schlüsselnamen der Workflowartefakte sind hierarchisch aufgebaut, um eine gute Strukturierung zu erlauben. Die Operationen für den Zugriff auf diese Artefaktverwaltung sind in Anhang A.5.2 beschrieben. Weiter Hilfs-Operationen sind in Anhang A.6.8 enthalten.

Auch die bereits im Workflowspezifikations-Modell festgelegten Artefakt-Hyperlinks benutzen die globale Workflowartefaktverwaltung. Jeder solche Hyperlink registriert sich gleich bei Erstellung des Workflows für ein Workflowartefakt. Der Name des Artefakts wird in der Workflowspezifikation festgelegt. Wird nun einem Hyperlink gemeldet, dass der Wert dieses Artefakts zum ersten mal auf ein Modell (in der Regel ein Submodell einer Aktivität) gesetzt wird, so erstellt der Hyperlink automatisch einen Navigationslink bei sich zu diesem Modell.

6.3 Modelltypen für Artefakte des MNM-Dienstmodellierungs-Workflows

Die Implementierung der Modelltypen für die Repräsentation des Dienstmodells, deren Modelle als Artefakte im Workflow für die MNM-Dienstmodellierungsmethodik verwendet werden (daher auch als Artefakt-Modelle bezeichnet), soll nun näher besprochen werden.

In Abschnitt 6.3.1 wird dabei auf Modelltypen für strukturierende Artefakte, in Abschnitt 6.3.2 auf den UML-Modelltyp, in Abschnitt 6.3.3 auf den Modelltyp für generische Beschreibungen, in Abschnitt 6.3.4 den SM-Modelltyp und in Abschnitt 6.3.5 auf den Modelltyp für Basiseinstellungen eingegangen.

6.3.1 Modelltypen für Strukturierung

Die in Abschnitt 5.4.1 entworfenen Modelltypen für Rollen-Definition, Dienstlebenszyklus, Prozessklassifizierung und QoS-Dimensionen wurden alle implementiert.

Die Metamodelle für sind in dabei den Dateien `MNMSMRoles.met`, `MNMSMLifeCycle.met`, `MNMSM-ProcClf.met` und `MNMSMQosDim.met` festgelegt und verwenden die zugehörigen Klassennamenpräfixe `MNMSMRoles`, `MNMSMLifeCycle`, `MNMSMProcClf` und `MNMSMQosDim`.

Jeder dieser Modelltypen ist relativ einfach strukturiert und es wird daher jeweils nur ein Alter-Codefile verwendet: Die zugehörigen Codefiles sind `mnm-sm-roles-general.lib`, `mnm-sm-lifcycle-general.lib` `mnm-sm-proc-clf-general.lib` und `mnm-sm-qos-dim-general.lib`.

Der Prozessklassifizierungs-Modelltyp erlaubt sowohl die Definition einer Prozessklassifizierung für die Dienstsicht als auch für die Realisierungssicht. Für die Übernahme von vorhandenen Prozessklassen aus der jeweils anderen Teilsicht während der Workflowabarbeitung wurden spezielle Mechanismen integriert.

6.3.2 Modelltyp für UML

Hier wird die prototypische Implementierung des UML-Modelltyps, dessen Entwurf in Abschnitt 5.4.2 behandelt wurde, besprochen.

Der UML-Modelltyp dient der Repräsentation aller UML-Diagrammtypen innerhalb des Werkzeugs. Ein Modell dieses Modelltyps stellt einen oder mehrere (rekursiv) enthaltene UML-Diagramme bestimmter UML-Diagrammtypen dar. Eine Anforderung an die Entwicklung des UML-Modelltyps war die möglichst vollständige Realisierung des UML-Metamodells der UML-Spezifikation (siehe [OMG 01-02-14], vgl. Abschnitt 5.4.2).

Das Metamodell für den UML-Modelltyps ist in der Datei `MNMUML.met` enthalten und verwendet als Klassennamenpräfix `MNMUML`.

Der Alter-Code ist aufgrund der enormen Komplexität des Modelltyps auf viele Dateien verteilt. Darunter gibt es generische Codefiles, die allgemeine Alter-Operationen für den gesamten Modelltyp bereitstellen, und spezifische Codefiles, die bestimmte UML-Metaklassen implementieren und dabei die allgemeinen Operationen nutzen. Die generischen Codefiles sind:

- `mnm-uml-general.lib`: allgemeine Definitionen bzw. Standards für Operationen zur Realisierung der Protodome-Methoden

- `mnm-uml-graph.lib`: Handhabung des Modells selbst, u.a. Verwaltung der hierarchischen Schachtelung von Modellen des UML-Modelltyps
- `mnm-uml-label.lib`: Konzepte für einfache Angabe des Objekt-Label-Textes für UML-Elemente
- `mnm-uml-formatting.lib`: allgemeine Operationen für Textformatierung
- `mnm-uml-custom-button.lib`: Steuereung der Buttonleiste
- `mnm-uml-custom-menuitem.lib`: Steuerung der Custom-Menüs
- `mnm-uml-reference.lib`: spezielles Referenzierungskonzept für UML-Elemente (z.B. für rekursive Nummerierung bei UML-Messages in Kollaborationsdiagrammen verwendet)
- `mnm-uml-arc.lib`: allgemeine Hilfsfunktionen für Kanten innerhalb des UML-Modelltyps
- `mnm-uml-name.lib` und `mnm-uml-namespace.lib`: Namens- und UML-Namesraum-Verwaltung
- `mnm-uml-property.lib`: Handhabung von allgemeinen Grapething-Properties für UML-Modellelemente

Die spezifische Codefiles und die in ihnen realisierten UML-Metaklassen (jeweils Auswahl) sind dagegen die folgenden:

- `mnm-uml-core.lib`: Comment, Constraint, TaggedValue, Stereotyp
- `mnm-uml-dependency.lib`: Dependency, Abstraction, Usage, Permission und Flow
- `mnm-uml-generalization.lib`: Generalization
- `mnm-uml-classifier.lib`: Class, Interface, Datatype
- `mnm-uml-association.lib`: Association
- `mnm-uml-association-end.lib`: AssociationEnd
- `mnm-uml-feature.lib`: Attribute, Operation, Reception
- `mnm-uml-parameter.lib`: Parameter
- `mnm-uml-package.lib`: Package, Model
- `mnm-uml-usecase.lib`: UseCase, Actor, Include, Extend
- `mnm-uml-action.lib`: Action
- `mnm-uml-collaboration.lib`: Collaboration
- `mnm-uml-classifier-role.lib`: ClassifierRole
- `mnm-uml-association-role.lib`: AssociationRole, AssociationEndRole
- `mnm-uml-message.lib`: Message
- `mnm-uml-state.lib`: StateVertex, PseudoState, State (zumindest soweit, wie für Aktivitätsdiagramme notwendig)
- `mnm-uml-event.lib`: Event
- `mnm-uml-transition.lib`: Transition, Guard
- `mnm-uml-activitygraph.lib`: ActivityGraph, ActionState, ObjectFlowState

Die bisherige prototypischen Implementierung enthält noch nicht alle Aspekte des UML-Metamodells, die für die MNM-Dienstmodellierung wünschenswert wären. Vor allem sind sämtliche UML-Metaklassen für

Instanzen (d.h. UML-Objekte und damit in Verbindung stehende Konzepte wie etwa Links) nicht realisiert. Es wird in Bezug auf Instanzen derzeit nur das abstraktere Konzept des Classifiers (Klasse, Interface, Assoziation etc.) umgesetzt. Daher sind Kollaborationsdiagramme nur in der abstrakten Form (mit ClassifierRoles) und nicht in der instanziierten Form möglich. Eine Erweiterung des Metamodells sowie die zugehörige Spezifikation von Code wäre mit entsprechendem Zeitaufwand möglich, wobei sich hierbei die bereits entwickelten Konzepte für den abstrakteren Fall (Klassen, Assoziationen) auf den Instanzen-Fall übertragen ließen. Die noch fehlenden Aspekte im Einzelnen sind:

- Classifier-Instanzen, d.h. Object, Link, Stimulus, etc.
- Assoziation n-är (nicht nur binär)
- Kollaborationsdiagramme in Instanzenform (d.h. mit Objekten)

Weiterhin könnte der UML-Modelltyp um folgende UML-Konzepte bzw. UML-Diagrammtypen, die für das MNM-Dienstmodell derzeit nicht unbedingt sind, erweitert werden:

- Komponenten- und Verteilungsdiagramme
- Allgemeine Zustandsdiagramme
- Sequenzdiagramme

Eine zusätzliche Erweiterung wäre außerdem die Realisierung eines generischer Zugriffes, der alle UML-Elemente eines Modells des UML-Modelltyps in generischer UML-Metaobjektform (Metaklassenname, Metaattribute/Referenzen-Werte), d.h. unabhängig von Darstellung, präsentiert. Dies wäre z.B. für Export-Mechanismen (XMI [OMG 98-09-03] oder ähnliches) sinnvoll.

6.3.3 Modelltyp für generische Beschreibung

In Abschnitt 5.4.3 wurde der Modelltyp für generische Beschreibungen entworfen. Er erlaubt eine Beschreibung von beliebigen Konzepten oder Begriffen in Form von Text- bzw. Textlistenform, Dateireferenzen, Dokument- und Standardreferenzen, als generisches „Struct"-Konzept oder durch beliebige Submodelle von *DoME*.

Verwendung finden Modelle dieses Modelltyps innerhalb des MNM-Dienstmodellierungs-Workflows zur Spezifikation von Anforderungen oder für die Spezifikation der eigentlichen Dienstmodellelemente, die nicht durch UML beschrieben sind.

Das Metamodell ist in der Datei `MNMGenSpec.met` definiert und benutzt das Klassennamenpräfix `MNMGenSpec`. Der Code für diesen Modelltyp weitgehend in der Datei `mnm-gen-spec-general.lib` enthalten.

Nur speziell für den Import von Einträgen aus *Bibtex*-Dateien wurde in der Datei `mnm-bibtex-file.lib` eine entsprechende Unterstützung geschaffen. Diese könnte aber noch weiter ausgebaut werden (z.B. Ausdehnung auf Standards etc.). Auch wären Mechanismen für den Export von Dokumentreferenzen in Bibtex-Files denkbar.

6.3.4 Servicemodel-Modelltyp

Der Entwurf des Servicemodel-Modelltyps (kurz SM-Modelltyp) ist in Abschnitt 5.4.4 beschrieben. Dieser Modelltyp ist für die Definition der eigentlichen Elemente des Dienstmodells und den spezifischen Beziehungen unter diesen gedacht. Die genauere Beschreibung der hierin definierten Elemente wird mit Submodellen vom UML-Modelltyp oder vom Modelltyp für generische Beschreibung realisiert. Außerdem werden zur Strukturierung als Submodelle Modelle für strukturierenden Artefakte verwendet.

Das Metamodell für den SM-Modelltyp ist in der Datei `MNMSMGeneral.met` gespeichert und benutzt das Klassennamenpräfix `MNMSMGeneral`. Der Alter-Code verteilt sich auf mehrere Dateien, nämlich `mnm-sm-general-general.lib`, `mnm-sm-general-graph.lib`, `mnm-sm-general-init.lib`, `mnm-sm-general-property.lib`, `mnm-sm-general-artifact.lib`, `mnm-sm-general-uml-extobj.lib`, `mnm-sm-general-completion-check.lib`, `mnm-sm-general-copy.lib` und `mnm-sm-general-custom-tools.lib`.

Um der Verteilung des Dienstmodells innerhalb des Workflows gerecht zu werden, können Modelle des SM-Modelltyps in verschiedenen Ausprägungen, die den verschiedenen Teilen des Dienstmodells entsprechen, vorkommen. In jeder Ausprägung sind nur bestimmte Elemente erstellbar. Die notwendigen Mechanismen für die Verteilung des Dienstmodells bzw. die Workflowsteuerungs-Anbindung sind weitgehend in der Datei `mnm-sm-general-artifact.lib` festgelegt. Die Anbindung von UML-Diagrammen, d.h. genauer gesagt die Festlegungen, wie Dienstmodellelemente, die in einem Modell des SM-Modelltyps definiert sind, in Submodellen des UML-Modelltyps durch UML-Elemente repräsentiert werden (z.B. Entities oder Rolleninstanzen durch Aktoren, Prozesse durch Use-Cases), sind in der Datei `mnm-sm-general-uml-extobj.lib` enthalten. Mechanismen zur Konsistenzprüfung speziell für Modelle des SM-Modelltyps (in den möglichen verschiedenen Ausprägungen) sind in der Datei `mnm-sm-general-completion-check.lib` definiert. Diese sind bisher nur für das Basismodell und die Dienstsicht, aber nicht für die Realisierungssicht implementiert.

In den Dateien `mnm-sm-general-init.lib`, `mnm-sm-general-graph.lib` und `mnm-sm-general-copy.lib` befinden sich Mechanismen zur Modell-Initialisierung, u.a. Mechanismen zur Übernahme von Teilen des Dienstmodells (vor allem Prozessklassen und der sie genauer beschreibenden UML-Diagramme zwischen Dienst- und Realisierungssicht) in andere Methodik-Aktivitäten während der Workflowabarbeitung. Hierin enthalten ist auch die Regelung von Hyperlinks auf andere Modelle des SM-Modelltyps, die den Benutzer bei der Navigation für das über den Workflow verteiltem Dienstmodell unterstützen.

In der Datei `mnm-sm-general-property.lib` ist die Handhabung von Grapething-Properties für die einzelnen Dienstmodellelemente geregelt. Die Datei `mnm-sm-general-custom-tools.lib` enthält die Steuerung von Buttonleiste und Custom-Menüeinträgen.

6.3.5 Modelltyp für Basiseinstellungen

Mit einem Basiseinstellungen-Modell werden Basiseinstellungen der MNM-Dienstmethodik repräsentiert. Der Entwurf dieses Modelltyps wird in Abschnitt 5.4.5 besprochen.

Das Metamodell für die Basiseinstellungen selbst ist in der Datei `MNMSMSettings.met` festgelegt und hat als Klassennamenpräfix `MNMSMSettings`. Sämtlicher Alter-Code für den Modelltyp befindet sich in Datei `mnm-sm-settings-general.lib`.

Basiseinstellungen sind vor allem die Vorgehenweise (top-down oder bottom-up) und der Modellierungsfalls. Weiterhin sind mit diesem Modelltyp Festlegungen für die Rekursion der Dienstmodellierung bei Providerhierarchien definierbar. Eine bereits vorher erstellte Instanz des Dienstmodells kann hierzu als Submodell in einem Basiseinstellungs-Modell verwendet werden. In *DoME* können Submodelle für ein Graphobjekte entweder direkt an ein Graphobjekt gebunden werden. Dann werden sie in der gleichen Datei gespeichert wie das Graphobjekt. Es besteht aber auch die Möglichkeit bereits erstellte Modelle (deren Editor gerade geöffnet ist) nur als Submodell zu referenzieren. In diesem Fall wird nur eine Referenz in der Datei, die das Graphobjekt enthält, auf das Modell gespeichert. Diese Referenzierungs-Methode wird hier für die Modellierungs-Rekursion benutzt. Zusätzlich zu der Referenz zu der vorhandenen Dienstmodell-Instanz wird festgelegt welcher Sub-Dienst aus der bereits vorhandenen Instanz in der neu erstellten Instanz als Hauptdienst modelliert wird.

6.4 Der MNM-Dienstmodellierungs-Workflow

Hier wird die Implementierung des in Abschnitt 5.5 entworfenen MNM-Modellierungsmethodik-Workflows behandelt. Für den Entwurf dieses Workflows wurde ein Workflowspezifikations-Modell hauptsächlich bestehend aus Subworkflowschritten, Aktivitäten und Transitionen erstellt. Dieses Modell ist in der Datei `wf-specification.dom` im Unterverzeichnis `model/` des mnm-spezifischen Unterverzeichnisses des *DoME*-Tool-Verzeichnisses abgespeichert. Es wird als Grundlage für alle erstellten Workflowinstanzen des Dienstmodells verwendet.

Für die Realisierung des Workflows muss jeder im Modell festgelegten Workflowaktivität ein Workflowmodulname zugeordnet werden, sowie jede der zugehörigen Workflowmoduldateien, d.h. alle Workflowmodul-Prozeduren zur Beschreibung des Aktivitätsverhaltens (vgl. Abschnitt 6.2), implementiert werden. (für die Zuordnung von Modulname und Moduldatei siehe Abschnitt 6.2)

Wie in Abschnitt 5.5 entworfen, teilen sich die Aktivitäten dabei in zwei Gruppen: Aktivitäten der ersten Gruppe erschaffen selbst Submodelle als Artefakte und registrieren sie global beim Workflow, die Aktivitäten der zweiten Gruppe verwenden dagegen von anderen Aktivitäten erschaffene Modelle und verarbeiten bzw. steuern sie weiter. Für den Umgang mit den Artefakten müssen daher die bereits beim Entwurf festgelegten globalen Artefakt-Schlüsselnamen in den jeweiligen Workflowmodulen entsprechend verwendet werden.

Alle für den MNM-Dienstmodellierungsmethodik-Workflow realisierten Workflowmodule verwenden außerdem einige Alter-Definitionen, die in der speziellen Datei `mnm-wf-sm.lib` gemacht werden.

6.5 Zusammenfassung

Dieses Kapitel befasste sich mit der prototypischen Implementierung des Werkzeugs. Es wurde zunächst das Vorgehen zum Realisieren von entworfenen Modelltypen allgemein beschrieben. Darauf aufbauend wurde auf die einzelnen Modelltypen eingegangen. Hierbei wurde zunächst die Realisierung von Workflows allgemein betrachtet. Anschließend wurde die Implementierung der Modelltypen für die Repräsentation des Dienstmodells besprochen Zuletzt wurde die Realisierung des MNM-Modellierungsmethodik-Workflows selbst behandelt.

Kapitel 7

Die werkzeugunterstützte Anwendung der Methodik und Test

In diesem Kapitel wird die Anwendung der MNM-Modellierungsmethodik mit dem entwickelten Werkzeug beispielhaft gezeigt. Außerdem wird hierbei gleichzeitig auf den Test des Werkzeugs eingegangen.

Um zu überprüfen, ob das entwickelte Werzeug die gestellten Anforderungen erfüllt, wurde es an bereits vorhandenen Instanzen des MNM-Dienstmodells bzw. Ausschnitten davon getestet, d.h. die Modellierung für den jeweils beschriebenen Dienst wurde mit dem Werkzeug nochmals durchgeführt. Diese Tests und ihre Ergebnisse sollen hier erläutert werden.

Da das MNM-Dienstmodell auf Basis neuester Forschungserkenntnisse entwickelt wurde, gibt es bisher jedoch keinen konkreten Dienst, auf den die MNM-Dienstmodellierungsmethodik vollständig angewendet wurde, d.h. es existiert keine vollständige, konkrete Instanz des MNM-Dienstmodells. Derzeit gibt es nur eine Reihe von Beispielen, die jeweils einen bestimmten Ausschnitt des MNM-Dienstmodells betrachten. Diese Beispiele sind in [GHH+ 01], [GHK+ 01], [GHH+ 02], [Roel 02] sowie [Schm 01] beschrieben. Mit ihnen wurde das Werkzeug tatsächlich getestet.

Die Dateien der hierbei erstellten Dienstmodell-Instanzen wurden alle im Unterverzeichnis `tests/` des mnm-spezifischen Unterverzeichnisses im *DoME*-Tool-Verzeichnis (vgl. Abschnitte 6.1.1 und 6.1.2) abgelegt.

Es wurden die folgenden Dienstmodell-Instanzen (bzw. die davon vorhandenen Ausschnitte) getestet:

1. E-Commerce-Dienst (nur Basis-Modell), beschrieben in [GHH+ 01]
2. User-Help-Desk-Dienst, beschrieben in [GHK+ 01] und [GHH+ 02]
3. Extranet-Dienst, beschrieben in [Roel 02] bzw. in [Schm 01]

Diese Dienste werden im Folgenden einzeln betrachtet.

7.1 E-Commerce-Dienst

Der E-Commerce-Dienst ist in [GHH+ 01] beschrieben. Für diesen Dienst wurde nur das Basis-Modell angegeben. Dieses Basis-Modell wurde mit dem Werkzeug erneut entworfen. Die Abb. 7.1 zeigt die Werkzeug-Version des Basis-Modells, d.h. das Modell des Servicemodel-Modelltyps (siehe Abschnitt 5.4.4), das das Basis-Modell repräsentiert.

Hierbei stellen die großen viereckigen Rechtecke im oberen Teil des Modells Hyperlinks zu den Modellen für die Spezifikation der Transparenzen (ein Modell für generische Beschreibungen, siehe Abschnitt 5.4.3,

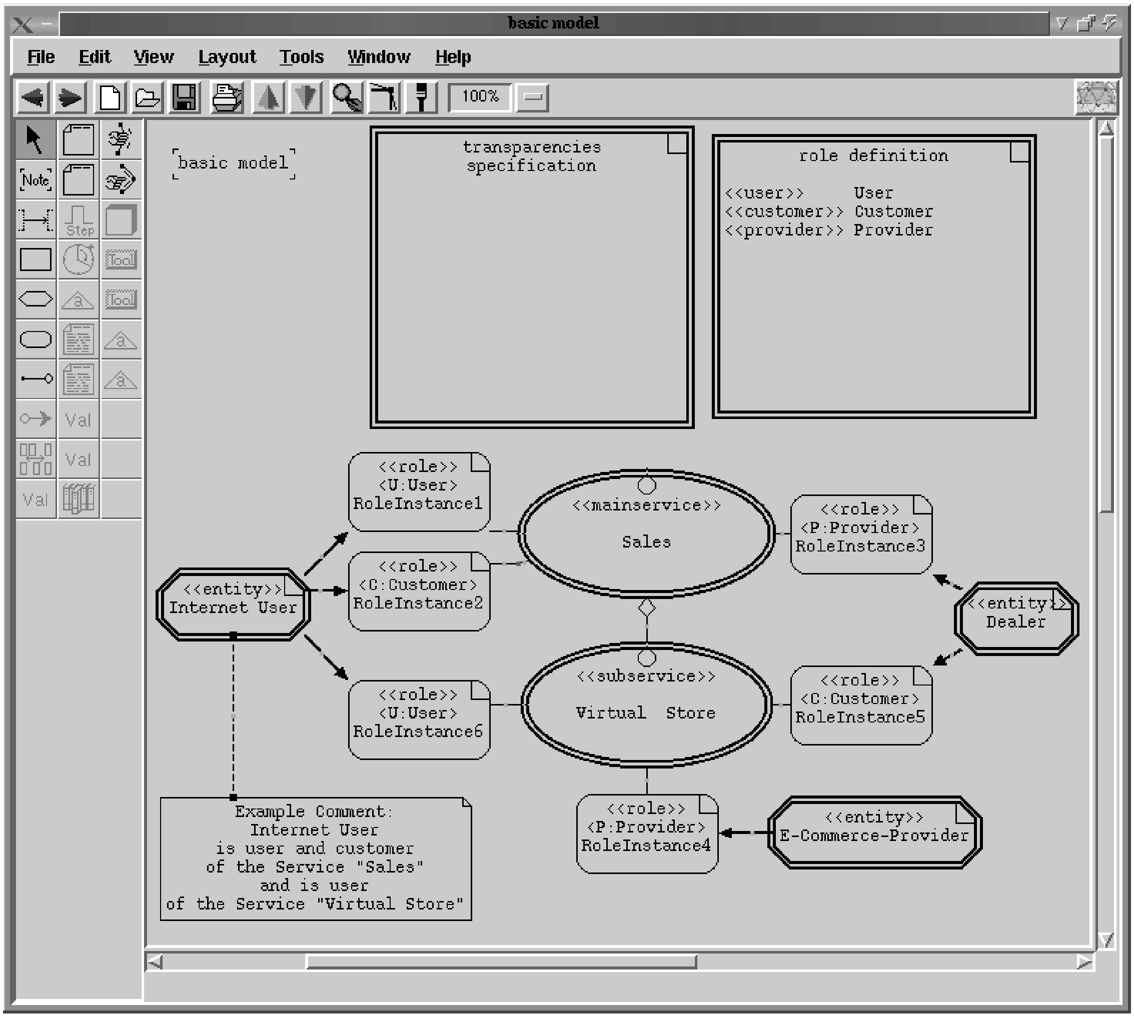

Abbildung 7.1: Screenshot eines *DoME*-Fensters für ein Modell des SM-Modelltyps (Basis-Modell des des E-Commerce-Dienst)

wird hier nicht verwendet) und der Rollendefinition (siehe Abschnitt 5.4.1, hier wird pro Basisrolle jeweils nur eine Rolle definiert) dar.

Im unteren Teil des Modells ist das eigentliche Basismodell — bestehend aus Diensten (ellipsenförmig), Entitäten (rechteckig mit schiefen Ecken) und Rolleninstanzen (rechteckig mit abgerundeten Ecken) — zu sehen. Bei jeder Rolleninstanz wird in eckigen Klammern die Rolle angegeben, der die Rolleninstanz zugeordnet ist. Die einzelnen Typen von Dienstmodellelementen sind jeweils auch — ähnlich wie bei UML — an Schlüsselwörtern in Guillemetten unterscheidbar.

Abschließend ist zu sagen, dass sich der E-Commerce-Dienst mit dem Werkzeug vollständig modellieren ließ.

7.2 User-Help-Desk-Dienst

In [GHK+ 01] und [GHH+ 02] werden für einen User-Help-Desk-Dienst Teile des Dienstmodells angegeben. Auch sie wurden zum Test mit dem Werkzeug nochmals erstellt.

In den Abbildungen 7.2 und 7.3 sind Beispiele für Workflowinstanz-Modelle (siehe Abschnitt 5.3.2) zu sehen. Die Abb. 7.2 zeigt den Top-Level-Workflow zur MNM-Dienstmethodik, der alle Methodik-Workflows enthält (vgl. Abschnitt 5.5). Der Dienstsicht Top-Down-Workflow ist als Beispiel in Abb. 7.3 dargestellt. Der Beginn und die Enden eines Workflows werden hierbei wie bei UML-Zustandsdiagrammen durch einen kleinen, ausgefüllten Kreis bzw. kleine Kreise mit einem Punkt in der Mitte dargestellt. Die Workflowschritte werden durch Rechtecke repräsentiert. Aktivitäten sind dabei mit dem Schlüsselwort „activity“ und Subworkflowschritte mit dem Schlüsselwort „workflow“ gekennzeichnet. Transitionen zwischen den Workflowschritten werden durch gerichtete Kanten (im Bild in dunkler, schwarzer Farbe) dargestellt. Hyperlinks für Artefakte sind ebenfalls in beiden Modellen zu sehen. Sie sind nur durch einen Text dargestellt, der von keiner eigenen Umrandung begrenzt wird. Diese Hyperlinks für Artefakte sind untereinander bzw. mit den Workflowschritten ebenfalls durch Kanten, sogenannten Artefakt-Hyperlink-Verbindungen, verbunden (im Gegensatz zu den Kanten für Transitionen: grau und gestrichelt/gepunktet). Die Artefakt-Hyperlink-Verbindungen haben innerhalb des Workflows keine Ausführungs-Semantik, sondern sollen nur dem Modellierungsbenutzer Anhaltspunkte für die Verwendung bzw. die Bedeutung der Artefakte geben (vgl. Abschnitte 5.3.1, 5.5). Beim Modell für den Top-Level-Workflow ist in der rechten, oberen Ecke ein Kommentar zu sehen. Solche Kommentare können bei der Workflow-Spezifikation im Workflow beliebig platziert werden und haben keine weitere Semantik.

In Abb. 7.4 ist das Basis-Modell des User-Help-Desk-Diensts — erstellt mit dem Werkzeug — zu sehen. Es hat prinzipiell den gleichen grafischen Aufbau wie das Basis-Modell des E-Commerce-Dienstes (vgl. Abschnitt 7.1).

Die Abb. 7.5 zeigt ein Modell des Servicemodel-Modelltyps (siehe Abschnitt 5.4.4) für die Dienst-Client/Zugangspunkt-Definition des User-Help-Desk-Dienstes. Ähnliche Modelle des Servicemodel-Modelltyps gibt es auch für die anderen Teile des Dienstmodells, wie z.B. die Festlegung der Dienstfunktionalität und der QoS-Parameter.

Die Abb. 7.6 zeigt ein Modell des Servicemodel-Modelltyps für die Übersicht über die Dienstsicht des User-Help-Desk-Dienstes. Dieses Modell enthält nur Hyperlinks auf die anderen Modelle des Servicemodel-Modelltyps, die die verschiedenen Teile der Dienstsicht (Funktionalität, QoS-Parameter, Clients/Zugangspunkte) darstellen. Auch der User-Help-Desk-Dienst ließ sich mit dem Werkzeug vollständig modellieren.

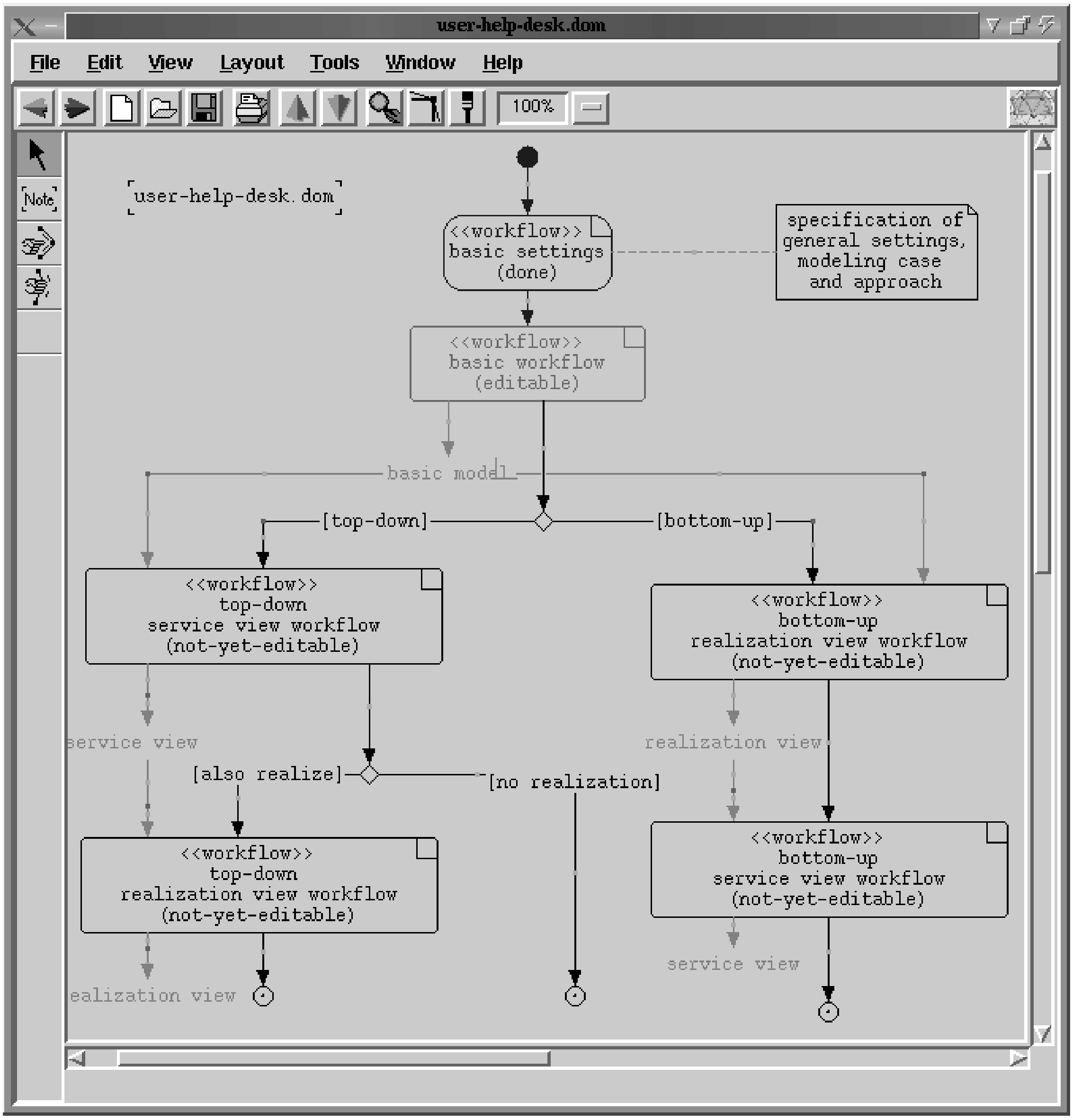

Abbildung 7.2: Screenshot eines *DoME*-Fensters für ein Workflowinstanz-Modell (Top-Level-Workflow der MNM-Dienstmethodik)

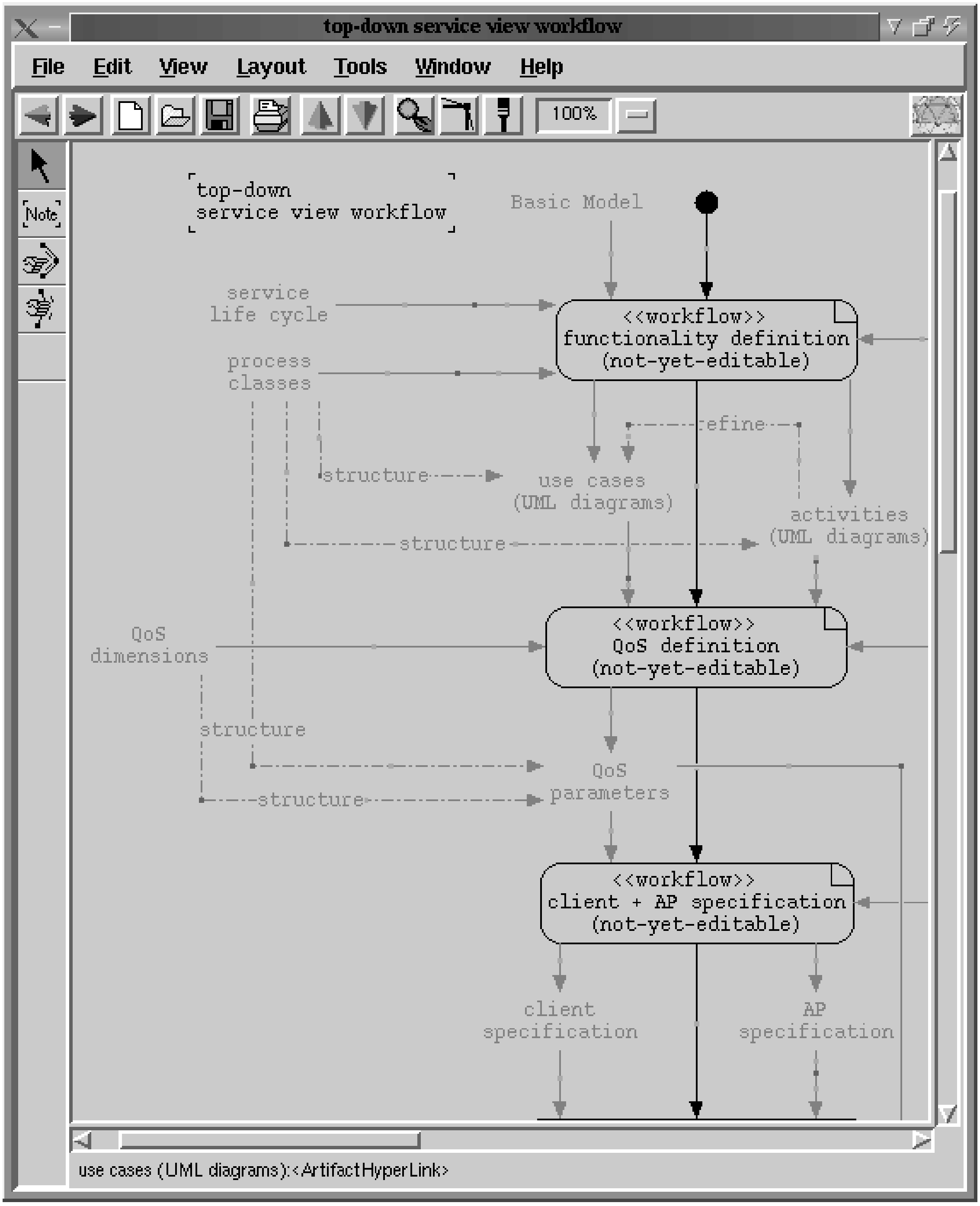

Abbildung 7.3: Screenshot eines *DoME*-Fensters für ein Workflowinstanz-Modell (Dienstsicht Top-Down-Workflow der MNM-Dienstmethodik)

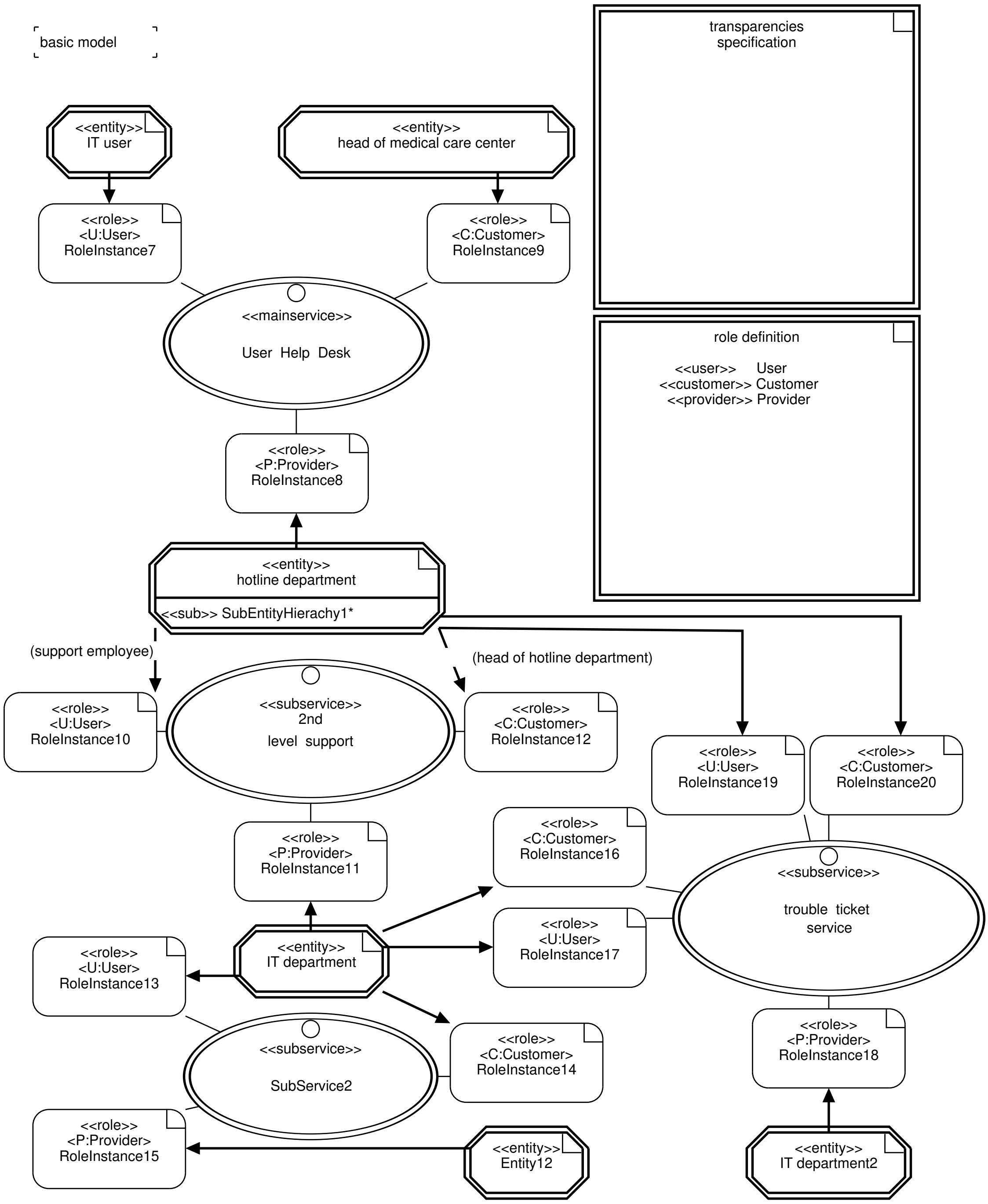

Abbildung 7.4: Ausdruck des Modells des SM-Modelltyps für das Basis-Modell des User-Help-Desk-Dienst

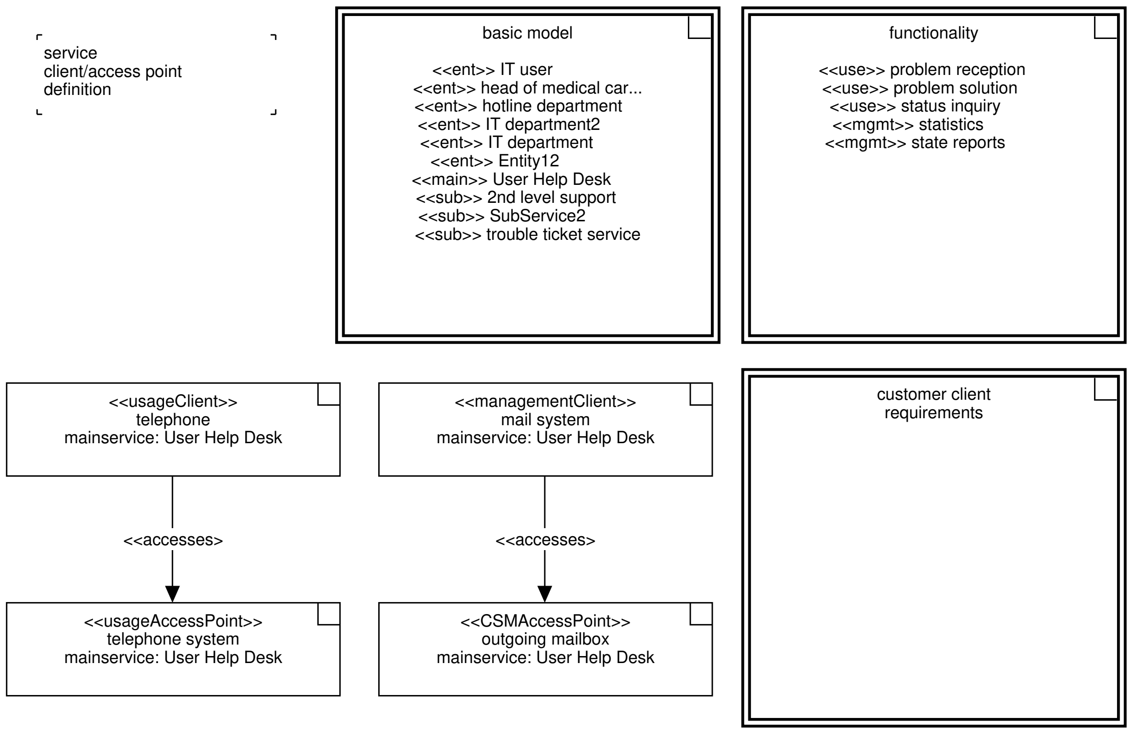

Abbildung 7.5: Ausdruck des Modells des SM-Modelltyps für die Dienstzugangspunkt-Definition des User-Help-Desk-Diensts

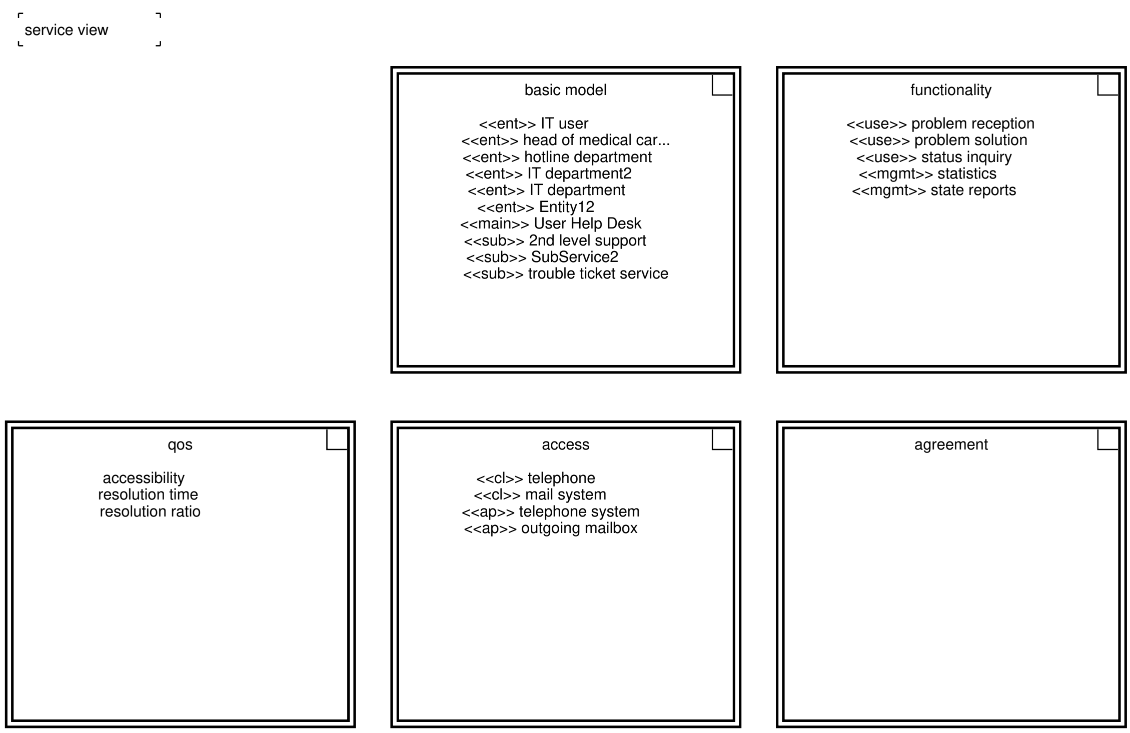

Abbildung 7.6: Ausdruck des Modells des SM-Modelltyps zur Übersicht über die Dienstsicht des User-Help-Desk-Diensts

7.3 Extranet-Dienst

In [Roel 02] bzw. in [Schm 01] wird ein Extranet-Dienst mittels des Dienstmodells beschrieben. Dieses Beispiel für eine Instanz des Dienstmodells ist wesentlich umfangreicher als die beiden anderen betrachteten Beispiele. Auch der Extranet-Dienst wurde mit dem Werkzeug zum Test erneut vollständig modelliert.

Es wird hier wegen des großen Umfangs nur ein einziger, kleiner Ausschnitt betrachtet: Die Abb. 7.7 zeigt ein Modell (bzw. einen Ausschnitt davon) des UML-Modelltyps (siehe Abschnitt 5.4.2) mit einem Aktivitätsdiagramm, das einen Problem-Management-Prozess für den Extranet-Dienst beschreibt.

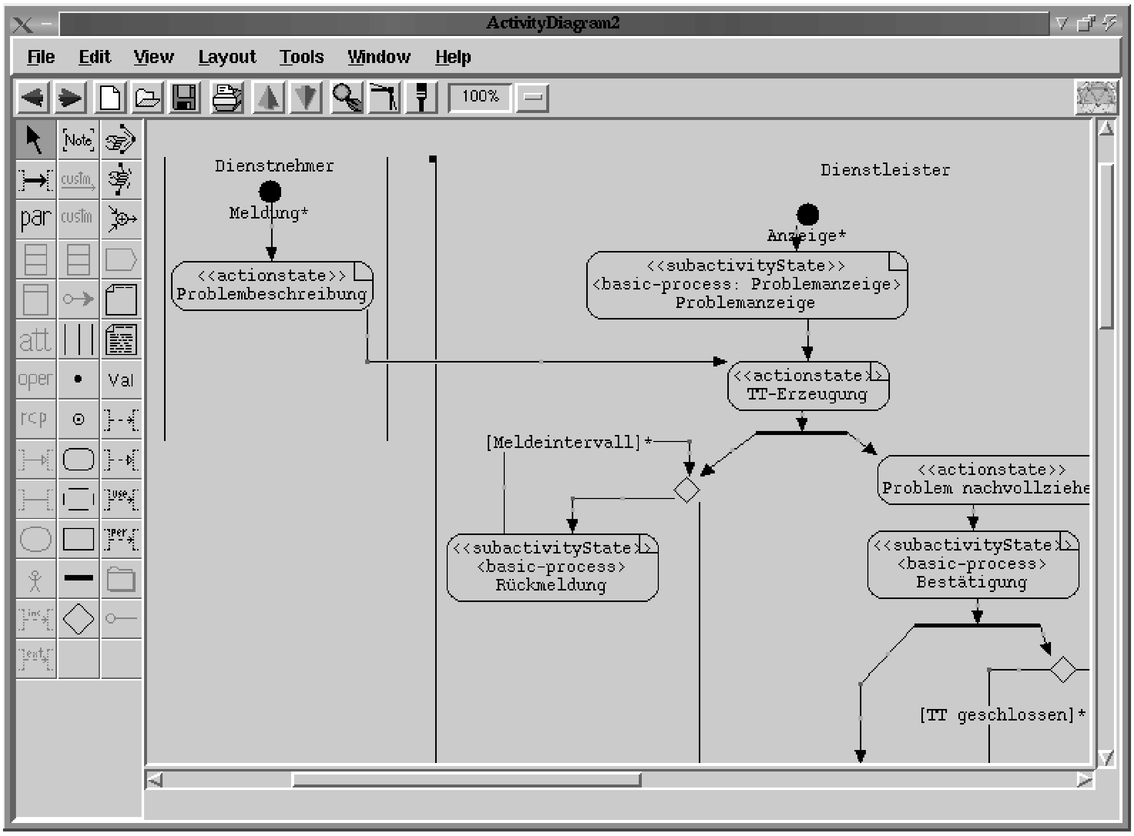

Abbildung 7.7: Screenshot eines *DoME*-Fensters für ein Modell des UML-Modelltyps (Aktivitätsdiagramm für einen Problem-Management-Prozess des Extranet-Dienstes)

7.4 Zusammenfassung

Dieses Kapitel betrachtete die Verwendung des Werkzeugs zur Dienstmodellierung sowie den Test des Werkzeugs. Hierzu wurde das Werkzeug beispielhaft an einigen bestehenden Instanzen des Dienstmodells (bzw. Ausschnitten davon) erprobt. Mit dem Werkzeug war es für jeden der betrachteten Dienste möglich die jeweils vorhandenen Teile des Dienstmodells erneut zu erstellen. Das Werkzeug kann also zur Modellierung mit dem MNM-Dienstmodell bzw. dessen MNM-Dienstmethodik tatsächlich eingesetzt werden.

Im folgenden Kapitel wird abschließend eine Zusammenfassung der gesamten Arbeit sowie ein Ausblick dafür gegeben.

Kapitel 8

Zusammenfassung und Ausblick

Das MNM-Dienstmodell und die zu ihm gehörige MNM-Dienstmethodik stellen einen modernen und generischen Ansatz zur Modellierung von Diensten dar. Bei der Anwendung dieser Dienstmodellierung kommen jedoch viele verschiedene Artefakte, wie z.B. UML-Diagramme vor. Eine Werkzeugunterstützung für die MNM-Dienstmethodik, die den Benutzer bei der Modellierung zum großen Teil unterstützt, ist daher wünschenswert und sinnvoll.

In dieser Diplomarbeit wurde eine Werkzeugunterstützung für das MNM-Dienstmodell bzw. seine Dienstmethodik entworfen und prototypisch implementiert. Ziel dabei war es, den Benutzer bei der Anwendung der MNM-Dienstmethodik zur Erstellung einer Instanz des MNM-Dienstmodells zu einem gegebenem Anwendungsfall zu unterstützen. Das Werkzeug wurde als Erweiterung eines bestehenden OO/CASE-Werkzeuges entwickelt.

Zunächst wurden die MNM-Dienstmethodik sowie die für sie notwendigen Benutzerinteraktionen analysiert und daraus eine Architektur für das Werkzeug aufgestellt. Bei dieser Architektur wurde Wert darauf gelegt, dass das Werkzeug an zukünftige Änderungen der Dienstmethodik leicht anpassbar ist. Dies wird durch eine Trennung in Methodik-Spezifikations-Komponente und Methodik-Ausführungs-Komponente erreicht. Änderungen der Dienstmethodik müssen nur in der Methodik-Spezifikations-Komponente durchgeführt werden. Von der Methodik-Ausführungs-Komponente werden sie automatisch übernommen. Die MNM-Dienstmethodik ist ein sehr junger Vorschlag zum Vorgehen bei der Dienstmodellierung. Sie wird daher in der Zukunft möglicherweise weiterentwickelt oder an weitere Forschungen angepasst werden. Daher ist eine leichte Anpassung des Werkzeugs an eine geänderte bzw. erweiterte Dienstmethodik sehr wichtig. Aus der entwickelten Architektur wurden Anforderungen abgeleitet, die einerseits zur Auswahl des bestehenden OO/CASE-Werkzeuges, das als Grundlage für das zu entwickelnde Werkzeug dient, und andererseits für den Entwurf des zu entwickelnden Werkzeuges selbst dienen. Anhand dieser Anforderungen wurden ca. 10 bestehende OO/Case-Werkzeuge verglichen. Unter diesen erfüllte die Meta-Modellierungsumgebung *DoME* die Anforderungen am besten und wurde daher als Grundlage für das Werkzeug verwendet. *DoME* erlaubt die Modellierung in verschiedensten Diagrammnotationen. Hierbei können neue Diagrammnotationen entworfen werden und bestehende Notationen können erweitert werden. Damit erlaubt *DoME* eine komplette Umsetzung der entwickelten Architektur des Werkzeugs inkl. des vorgesehenen Workflowkonzepts, das eine Trennung von Spezifikation und Ausführung von Workflows vorsieht.

Die Architektur für das Werkzeugs wurde dann mit *DoME* umgesetzt und dabei das Werkzeug entworfen sowie prototypisch implementiert. Hierbei wurde eine Workflowunterstützung geschaffen, die sowohl Spezifikation als auch Abarbeitung von praktisch beliebigen Workflows innerhalb von *DoME* ermöglicht. Insbesondere ist damit die von der Architektur geforderte Trennung in eine Spezifikation und Ausführung von Workflows erfüllt. Diese generische Unterstützung wird speziell für die Spezifikation und Ausführung der MNM-Dienstmethodik verwendet. Weiterhin wurde die Repräsentation der Dienstmodell-Daten in *DoME* selbst ermöglicht. Hierbei wurden alle für das Dienstmodell benötigten Datentypen, wie z.B. jeder

UML-Diagrammtyp, der für das MNM-Dienstmodell benötigt wird, aber auch andere Beschreibungsformen wie z.B. Freitext, Standard- und Dokumentreferenzen, berücksichtigt.

Das entwickelte Werkzeug unterstützt somit die gesamte MNM-Dienstmethodik inkl. aller darin vorkommenden Artefakte, die letzten Endes die Instanz des MNM-Dienstmodells, die den jeweils betrachteten und modellierten Dienst repräsentiert, darstellen.

Das Werkzeug bzw. die dafür entworfene Architektur bietet einerseits eine Vielzahl von Erweiterungsmöglichkeiten und ist aber andererseits auch für andere Einsatzgebiete verwendbar. Das Werkzeug wurde zur Umsetzung der workfloworientierten MNM-Dienstmodellierungsmethodik entwickelt. Die Architektur des Werkzeugs (vgl. Abschnitte 3.4 und 5.2) — insbesondere die Spezifikation und Abarbeitung von Workflows — wurde jedoch hierbei sehr generisch gehalten, so dass sich mit dieser Architektur praktisch beliebige workfloworientierte Modellierungsmethodiken unterstützen lassen, bei denen die Modellierung auf der Basis von Graphdiagrammen (z.B. UML-Diagramme, Zustandsdiagramme, Datenflussdiagramme, Petrinetze) beruht. Zum Beispiel kann jede jemals entwickelte, workfloworientierte Methodik, deren Modellierung mit Prozessmodellen arbeitet, mit dieser Architektur umgesetzt werden. Hierbei können jeweils weite Teile des entworfenen Werkzeugs ohne große Anpassung wiederverwendet werden.

Die für das Werkzeug entwickelten *DoME*-Modelltypen (vgl. Abschnitt 5.2) — vor allem die für die Artefakte der MNM-Dienstmethodik entworfenen *DoME*-Modelltypen (vgl. Abschnitt 5.4) —, lassen sich noch bezüglich vieler Aspekte, die für die MNM-Dienstmethodik nicht unbedingt notwendig sind, erweitern. Insbesondere können diese *DoME*-Modelltypen damit auch für andere Modellierungsmethodiken, bei denen diese zusätzlichen Aspekte berücksichtigt werden müssen, wiederverwendet werden:

- Der entwickelte UML-Modelltyp (siehe Abschnitt 5.4.2) kann um einen generischen (unabhängig von Darstellung) Zugriff auf die UML-Modellelemente nach Art des UML-Metamodells (nach Meta-Klassen, Attributen und Referenzen) erweitert werden. Darauf aufbauend kann ein UML-Daten-Export (z.B. mit XMI, siehe [OMG 98-09-03]) für das Werkzeug entwickelt werden.
- Der entwickelte UML-Modelltyp (siehe Abschnitt 5.4.2) kann erweitert werden, um auch die folgenden, nicht für die MNM-Dienstmethodik notwendigen Diagrammtypen zu repräsentieren: Komponenten/Verteilungsdiagramme, allgemeine UML-Zustandsdiagramme und UML-Sequenzdiagramme.
- Das entwickelte Workflowkonzept kann noch um Wiederholungskonzepte wie z.B. Schleifen erweitert werden (vgl. Abschnitt 5.3.1).

Weiterhin gibt es noch einige Aspekte des Werkzeugs in Bezug auf die MNM-Dienstmodellierungsmethodik selbst, die noch genauer betrachtet bzw. behandelt werden können:

- Der UML-Modelltyp muss noch bezüglich einiger Aspekte (siehe Abschnitt 6.3.2) vervollständigt werden.
- Die Daten für die rekursive Dienstmodellierung, die zu den Basiseinstellungen für die MNM-Dienstmethodik (siehe Abschnitt 5.4.5) zählen, könnten tatsächlich zur rekursiven Modellierung innerhalb von geeigneten Aktivitäten der Workflows (z.B. beim Basis-Modell-Workflow) verwendet werden. Bisher werden sie nur definiert und noch von keiner Workflowaktivität verwendet.
- Allgemein kann die automatische Anordnung von Graphobjekten sowie die Textformatierung in den für das Werkzeug entwickelten *DoME*-Modelltypen noch verbessert werden.

Abschließend betrachtet wurde in dieser Diplomarbeit eine Werkzeugunterstützung für das MNM-Dienstmodell bzw. die zugehörige MNM-Dienstmethodik entwickelt, die den Modellierungsanwender bei der Durchführung der Dienstmethodik unterstützt. Das Werkzeug ist hierbei an Änderungen der Dienstmethodik leicht anpassbar. Die für das Werkzeug entwickelte Architektur ist jedoch nicht auf die MNM-Dienstmethodik beschränkt, sondern kann auf beliebige andere workfloworientierte Modellierungsmethodiken, deren Modellierung auf Graphdiagrammen basiert, übertragbaren werden. Insbesondere können viele Teile des Werkzeugs hierfür ohne große Änderung übernommen werden.

Anhang A

Details zu DoME und der Werkzeugimplementierung

Dieser Anhang ist der Beschreibung von Implementierungsdetails gewidmet. Es wird hierbei auf wesentliche Teile des entwickelten Werkzeuges genauer eingegangen.

Zunächst wird aber ausführlichher auf einige Begriffe und Konzepte von *DoME* eingegangen. Insbesondere werden einige Begriffe eingeführt die in den darauffolgenden Detailbeschreibungen für das Werkzeug benutzt werden. Zusätzlich werden einige ergänzende Informationen zu Alter und zu *DoME* gegeben, die nicht in der Alter-Manual ([AlterManual]) enthalten sind.

Anschließend wird zunächst ausführlich auf die Realisierung der Protodome-Methoden für Modelltypen des entwickelten Werkzeuges eingegangen. Dann werden Details für die Workflowsteuerung genauer erklärt. Als nächstes wird eine Beschreibung der wichtigsten, implementierten Alter-Operationen in den generischen Alter-Codefiles gegeben.

Mit der Kenntnis all dieser in diesem Anhang gegebenen Informationen wird eine Anpassung bzw. Erweiterung des Werkzeuges, d.h. vor allem der realisierten Workflows und ihrer Artefakte, erleichtert.

A.1 DoME, Alter und Protodome – ausführlich

A.1.1 Modelle in DoME – ausführlich

Ein Modell in *DoME* ist ein **Graph**, besteht also grundsätzlich aus **Knoten** und **Kanten**, die jeweils zwei der Knoten zueinander (oder einen Knoten mit sich selbst) in Beziehung setzen. Dargestellt werden Modelle wie bei Graphen üblich als Graphdiagramme, d.h. jeder Knoten stellt jeweils ein eigenes grafisches Objekt dar und eine jede Kante wird als Verbindungslinie zwischen den Knoten, die sie zueinander in Beziehung setzt, gezeichnet. Im Folgenden werden die Begriffe Modell, Graph, Graphmodell, und Graphendiagramm synonym gebraucht.

In einem Modell in *DoME* sind aber verschiedene Typen von Knoten und Kanten, die sich in Darstellung und Verhalten und zusätzlichen Eigenschaften unterscheiden, möglich. Jedes Modell gehört einem bestimmten **Modelltyp** an, welcher vor allem die verschiedenen Knoten- und Kantentypen und die möglichen Beziehungen zwischen diesen festlegt. Gehört ein Modell einem Modelltyp an, so wird es auch eine **Instanz dieses Modelltyps** genannt.

Es sind mehr Beziehungen als bei reinen Graphen möglich: So können in einem Modell Knoten in sich selbst entweder andere Knoten oder sogenannte **Listelemente** enthalten. Von den Listelementen gibt es wie bei Knoten und Kanten verschiedene Typen. Sie können im Gegensatz zu Knoten (zumindest graphisch) selbst keine weiteren Objekte enthalten. So wie Knoten können sie jedoch sowohl untereinander als auch mit Knoten durch Kanten verbunden werden.

Listelemente wären z.B. für die Realisierung von Attributen oder Operationen innerhalb einer UML-Klasse geeignet. Knoten innerhalb eines anderen Knotens könnten z.B. für die Realisierung von Sub-Zuständen in UML-Zustandsdiagrammen benutzt werden.

Sowohl Kanten als auch Knoten können weiterhin an sie gebundende Knoten in ihrer Umgebung besitzen (sogenannte **Accessories**). Diese Accessory-Knoten sind normale Knoten, können also insbesondere auch durch Kanten mit anderen Knoten oder Listelementen verbunden werden.

Zum Beispiel werden Accessories in *DoME* für Label-Beschriftungen von Kanten oder spezieller für UML-Constraints, die nur an ein UML-Element gebunden werden oder Messages bei Kollaborationsdiagrammen, verwendet.

Knoten, Kanten und Listelemente werden in *DoME* zusammengefasst als **Graphobjekte** (also die Objekte die (zumindest graphisch) in einem Modell enthalten sind) bezeichnet. Graphobjekte und (Graph-)Modelle werden zusammengefasst unter Begriff **Grapething**. (GrapE ist der Name des Diagramm-Engines in *DoME*).

Jedes Graphobjekt (also jeder Knoten, jede Kante, jedes Listelement) kann beliebig viele sogenannte **Submodelle** (auch als **Subgraphen** oder **Subdiagramme** bezeichnet) beliebiger Modelltypen haben. Submodelle sind selbst vollwertige Modelle. Modelltypen, bei denen mindestens ein Typ eines Graphobjekt Submodelle besitzen darf, werden in *DoME* als **hierarchische Modelltypen** bezeichnet.

Außerdem kann jedes Grapething (also jedes Graphobjekt und jedes Modell für sich, d.h. ohne die in ihm enthaltenen Graphobjekte zu betrachten) weitere Eigenschaften, sogenannte **Properties**, besitzen. Diese können Werte von Grund-Typen wie z.B. Integer, String, Boolean oder verschachtelten Strukturen wie Listen oder Dictionaries (assoziative Arrays) annehmen. Auch Referenzen zu beliebigen Graphmodellen oder Graphobjekten sind als Werte möglich. Die Properties können die Darstellung und das Verhalten der Graphobjekte und Graphmodelle zu denen sie gehören beeinflussen.

Abb. A.1 zeigt den Aufbau eines Modells in *DoME* vollständig, d.h. im Gegensatz zu Abb. 5.5 aus Kapitel 5 inkl. der Accessory-Enthaltenseinsbeziehung und Listelementen in einem UML-Klassendiagramm.

Vor allem können hierarchische Modelle beliebigen Typs in *DoME* eine Einheit bilden. In einem Modell können die verschiedenen Typen von Graphobjekten Submodelle verschiedenster Modelltypen haben (auch unabhängig davon, welche der beiden Methoden für die tatsächliche Realisierung verwendet wurde).

DoME interagiert mit dem Benutzer standardmäßig über Edit-Fenster, wobei jedes Edit-Fenster ein Modell zeigt bzw. editieren lässt. Die Edit-Fenster der Submodelle bzw. das Edit-Fenster des Parent-Modells eines Submodelle sind dabei über Navigations-Buttons erreichbar.

Verschiedene Modelle, die hierarchisch zusammenhängen werden von *DoME* in einer einzigen Datei (inkl. aller Properties der enthaltenen Grapethings) gespeichert. Es besteht aber auch die Möglichkeit Modelle aus anderen Dateien nur zu referenzieren.

Als Erweiterungs-Sprache dient Alter:

- Objekt-orientierter Scheme-Dialekt als Teil von *DoME* in Smalltalk implementiert
- Möglichkeiten für Zugriff bzw. Veränderung oder Verarbeitung von (hierarchischen) Modellen, d.h. pro Typ eines Grapethings eine Alter-Klasse

Alter kann für die folgenden Aufgaben verwendet werden:

- Verarbeitung oder sogar Veränderung beliebiger, bestehender Modelle: Unterstützung von (SGML-artige) Dokumentgenerierung, Druckertreiber oder beliebige andere Auswertung oder auch Modifikation eines Modells (z.B. Code-Generierung)

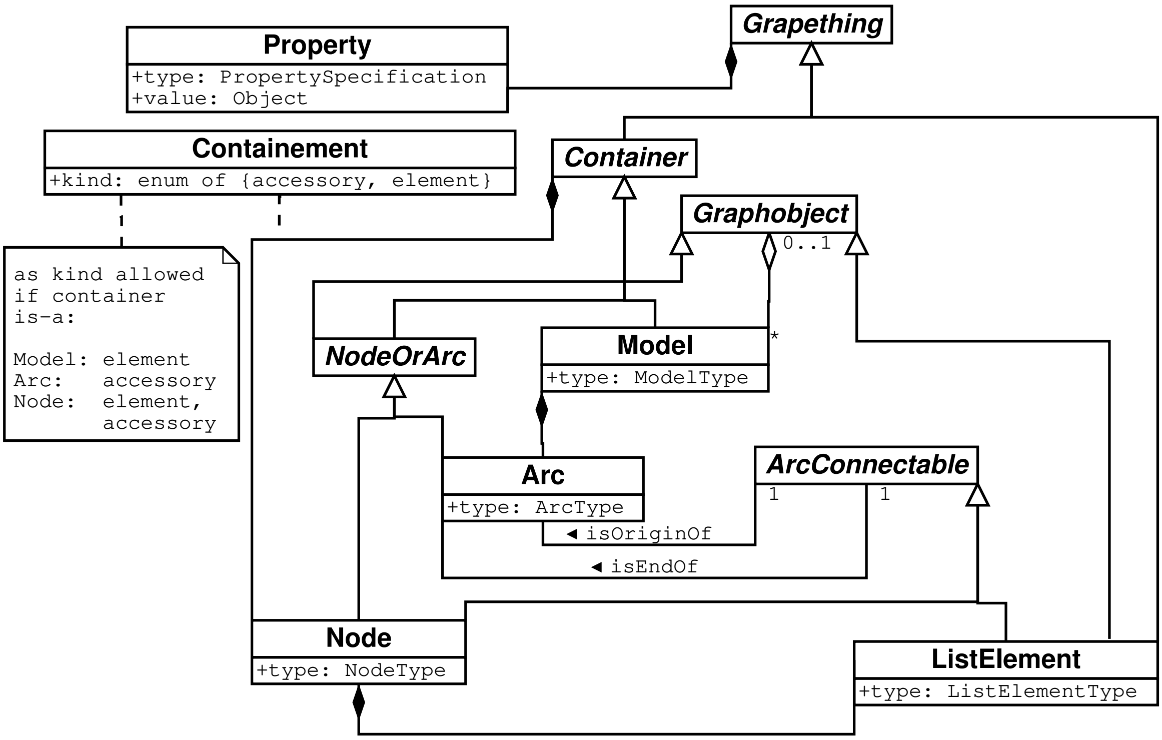

Abbildung A.1: Struktur eines *DoME*-Modells dargestellt mit UML — vollständig inkl. Accessory-Beziehung und Listelementen

- Für Protodome-Modelle (siehe oben): zur Erweiterung der Darstellung, der Struktur und des Verhaltens des Modells selbst

A.1.2 Aufbau und Struktur eines Metamodells – ausführlich

Im Folgenden soll kurz die Struktur eines Metamodells beschrieben werden. Das Metamodell ist selbst ein Modell. Es legt grafisch fest, welche Typen von Graphobjekten in Instanzen des Modelltyps vorkommen können und welche Beziehungen zwischen ihnen möglich sind.

Für jeden Typ eines Graphobjekts des spezifizierten Modelltyps (also eines Graphobjekts, das in einer Instanz des Metamodells vorkommen kann) gibt es im Metamodell einen Knoten, eine sogenannte **(Graphobjekt-)Spezifikation**.

In den Spezifikationsknoten wird im wesentlichen mithilfe von Properties das Aussehen der spezifizierten Graphobjekte sowie die Modelltypen für Submodelle festgelegt.

Durch Listelemente innerhalb der Spezifikationsknoten sind Properties der spezifizierten Graphobjekte definierbar.

Zwischen Spezifikationsknoten für Knoten und/oder Listelemente der Modelltyp-Instanzen legen spezielle Kanten, sognannte **Verbindungs-Constraints**, fest, welche Kantenbeziehungen in einer Modelltyp-Instanz möglich sind. Ein Verbindungs-Constraint erlaubt in einer Instanz des Modelltyps Kantenverbindungen zwischen Instanzen der durch das Verbindungs-Constraint verbundenen Graphobjektspezifikationen. Der hierbei erlaubte Kantentyp wird im Verbindungs-Constraint als zusätzliche Property festgelegt.

Weiterhin wird mit speziellen Kanten im Metamodell zwischen Graphobjektspezifikationen festgelegt, welche Knotentypen oder Listelement-Typen in einem anderen Knotentyp (grafisch) innerhalb einer

Modelltyp-Instanz enthalten sein können. Genauso wird spezifiziert, welche Knotentypen als Accessory eines Knoten- oder Kantentyps in einer Modelltyp-Instanz möglich sind.

Für das das Graphmodell selbst gibt es im Metamodell einen eigenen Spezifikationsknoten ähnlich den Spezifikationsknoten für Graphobjekte. Dieser dient vor allem der Festlegung von Properties, die das Modell selbst (nicht seine enthaltenen Graphobjekte) besitzt.

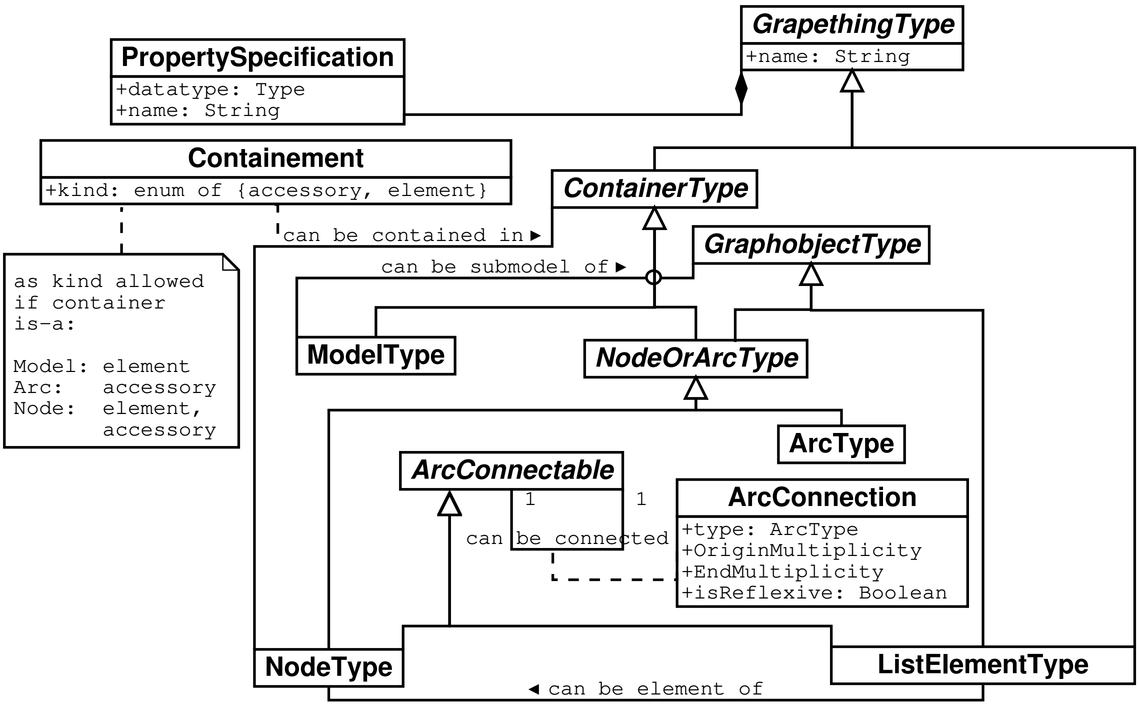

Abbildung A.2: Struktur eines *DoME*-Metamodells dargestellt mit UML — vollständig inkl. Accessory-Beziehung und Listelementen

In Abb. A.2 ist der Aufbau eines Metamodells in *DoME* vollständig, d.h. im Gegensatz zu Abb. 5.6 aus Kapitel 5 inkl. der Accessory-Enthaltenseinsbeziehung und Listelementen mit einem UML-Klassendiagramm dargestellt.

Im Metamodell sind auch Aspekte für die Interaktion des Benutzers (über das Edit-Fenster) mit einer Instanz des Metamodells definierbar. Vor allem die so sogenannte Toolbar, die aus Buttons für die Erstellung von Graphobjekten (Creation-Buttons) oder für den Aufruf spezieller Aktionen (Custom-Buttons) besteht, kann hier genau festgelegt werden. Auch Erweiterungen der *DoME*-Standard-Menüs für das Edit-Fenster einer Instanz des Metamodells sind im Metamodell festlegbar. Ein Metamodell wird in *DoME* daher auch oft als eine **DoME-Tool-Spezifikation** bezeichnet.

A.1.3 Protodome-Methoden allgemein

Auszuführende Blöcke von Code, denen bei Aufruf Parameter übergegeben werden können, heißen in Alter (wie in Scheme üblich) **Prozeduren**. Je nach Definition der Prozedur kann eine verschiedene Anzahl von Parametern bei Aufruf übergegeben werden.

Alter ist zudem objekt-orientiert, kennt also verschiedene Typen (= Klassen) und eine Vererbung zwischen diesen. Um dies bei der Ausführung von Code geeignet zu unterstützen, gibt es in Alter auch das Konzept des Dynamischen Bindens (Dynamic Binding) bei **Operationen**. Eine Operation wird aufgerufen wie eine Prozedur. Sie kann aber für verschiedene Typen verschiedene Methoden, d.h. verschienden Prozeduren, haben. Erst bei Aufruf der Operation wird anhand des Typs des ersten Parameters die auszuführende Methode, d.h. die tatsächlich auszuführende Prozedur, bestimmt.

Um die vom Metamodell vorgegebenen Klassen der verschiedenen Graphobjekte und des Graphmodells selbst mit Alter-Code zu erweitern, sind im Metamodell bei jedem Spezifikationsknoten sowohl bei allen Graphobjektspezifikationen als auch beim Graphmodellspezifikationsknoten spezielle Alter-Methoden definierbar. Diese speziellen Methoden werden im Folgenden unter dem Begriff **Protodome-Methoden** zusammengefasst.

Je nach Art (Knoten, Kante, Listelement oder Graphmodell selbst) des spezifizierten Grapethings gibt es hierbei verschiedene Protodome-Methoden. Grob einteilen kann man diese in Methoden für die Anpassung der Darstellung des Grapething-Typs, Methoden für (zusätzliches) Constraint-Checking bzw. Definition von (zusätzlichem) Verhalten bei Erstellung oder Löschung eines Grapething des spezifizierten Typs.

Bei der Spezifikationen für Knoten gibt es für folgende Aspekte Methoden:

1. Für die Darstellung:

    - **Objekt-Label-Text (Object Label Text):**
      zur Berechnung des Labels des Knoten; Standardmäßig wird nur der Name des Knoten als Label angezeigt. Wird nicht verwendet, falls (in der Spezifikation) festgelegt ist, dass Knoten dieses Typs keinen Namen besitzen.

    - **Linienstärke (Line Width):**
      legt Zeichenstärke der Linien für die Knotenumrandung fest

    - **Linienstil (Line Style):**
      legt Zeichenstil (durchgezogen, gestrichelt, gepunktet, etc.) der Linien der Knotenumrandung fest

    - **Linienanzahl (Line Count):**
      Zahl der Linien der Knotenumrandung; Bei Anzahl größer Eins wird innerhalb der Knotenumrandung dieselbe in geringem Abstand entsprechend oft wiederholt (jeweils entsprechend verkleinert) gezeichnet. Wird nicht verwendet, falls in der Knotenspezifikation „Custom Rectangular“ oder „Custom Circular“ für die Form eingestellt ist.

    - **Füll-Muster (Paint Pattern):**
      „Pinselstärke“ für Zeichnung des gesamten Knotens (inkl. Label-Text): entweder „Voll“ (solid) oder „50%“

    - **Poly-Linien-Stil (Poly Line Style):**
      Auswahl einer von Protodome’s vorgegebenen Punktlisten für Formen der Knotenumrandung; Wird nur verwendet, falls in der Knoten-Spezifikation „Polyline“ für die Form eingestellt ist.

    - **Poly-Linien-Punkte-Array (Polyline Point Array):**
      eigene Angabe der Punktliste für Spezifikation der Form der Knoten-Umrandung; Wird nur verwendet, falls in der Knoten-Spezifikation „Polyline“ für die Form eingestellt ist.

    - **Namensposition (Name Position):**
      gibt Position des Labels innerhalb der Knoten-Umrandung an.

    - **Knotenform/Knotenausdehnung (Node Shape/Node Bounds):**
      volle Spezifikation der Form (d.h. Zeichenbefehle für Linien, Rechtecke, Kreise und sogar Texte) der Knotenumrandung in Alter, anstatt der auch relativ vielfältigen Möglichkeiten der festen Angabe der Form (Rechteck mit verschiedenen Darstellungsmöglichkeiten für die Ecken, Ellipse oder direkte Angabe einer Punkt-Liste) in der Knotenspezifikation selbst; Wird nur verwendet, falls in der Knotenspezifikation „Custom Rectangular“ oder „Custom Circular“ für die Form eingestellt ist.

- **Exzentrität (Eccentricity):**
  Angabe eines Höhe-Länge-Verhältnisses bei Rechtecks oder Ellipsen-Form der Knoten-Umrandung; Wird nur verwendet, falls in der Knotenspezifikation „Rectangular“ oder „Circular“ für die Form eingestellt ist.

- **Eckentyp (Corner Type):**
  Typ der Ecke (gerundet, schief, etc.) bei Rechtecks-Form der Knoten-Umrandung; Wird nur verwendet, falls in der Knotenspezifikation „Rectangular“ für die Form eingestellt ist.

- **Ecken-Radius-Faktor (Corner Radius Factor):**
  Angabe der Ausdehnung einer Ecke bei Rechtecks-Form; Wird nur verwendet, falls in der Knotenspezifikation „Rectangular“ für die Form eingestellt ist.

- **Ansatzpunkt einer Kante (Clipping a Connector):**
  Berechnung des genauen Ansatzpunktes für eine Kante an der Umrandung des Knotens

2. Für Constraints/Aktionen bei Erstellung/Löschung:

- **Erstellungsprüfung (Creation Check):**
  prüft zusätzlich zu den Angaben im Metamodell selbst, ob ein Knoten (vom Benutzer) erstellt werden darf; bekommt als Parameter den Container (= enthaltenden Knoten oder das Graphmodell selbst) und eine Liste, die nur die Position innerhalb des Containers enthält; Wird auch auch bei Verschiebung (durch den Benutzer) verwendet.

- **Erstellungs-Post-Aktion (Creation Cleanup):**
  wird unmittelbar nach Erstellung des Knotens aufgerufen mit neuem Knoten als Parameter, für weitere Initialisierung des Knotens eine Art Konstruktor-Erweiterung.

- **Zerstörungsprüfung (Deletion Check):**
  prüft zusätzlich zu den Angaben im Metamodell selbst, ob ein Knoten (vom Benutzer) gelöscht werden darf; bekommt als Parameter den Knoten

- **Zerstörungs-Post-Aktion (Deletion-Cleanup):**
  wird nach Löschung eines Knotens aufgerufen; bekommt als Parameter den Container (= enthaltenden Knoten oder das Graphmodell selbst)

Beim Spezifikationsknoten eines Listelement-Typs ist per Methode für die Darstellung nur Label-Text anzupassen. Die Methoden für Erstellung und Löschung sind ähnlich wie bei Knoten:

1. Für die Darstellung:

- **Objekt-Label-Text (Object Label Text):**
  zur Berechnung des Texts des Listelements; Standardmäßig wird nur der Name des Listelements angezeigt.

2. Für Constraints/Aktionen bei Erstellung/Löschung:

- **Erstellungsprüfung (Creation Check):**
  prüft zusätzlich zu den Angaben im Metamodell selbst, ob das Listelement (vom Benutzer) erstellt werden darf; bekommt als Parameter den Container (= enthaltenden Knoten) das Graphmodell selbst) und eine Liste, die nur die Position innerhalb des Containers enthält; Wird auch auch bei Verschiebung (durch den Benutzer) verwendet.

- **Erstellungs-Post-Aktion (Creation Cleanup):**
  unmittelbar nach Erstellung des Listelements aufgerufen mit ihm als Parameter; für weitere Initialisierung des Grapethings; eine Art Konstruktor-Erweiterung

- **Zerstörungsprüfung (Deletion Check):**
  prüft ob ein Listelement (vom Benutzer) gelöscht werden darf; bekommt als Parameter das Listelement

- **Zerstörungs-Post-Aktion (Deletion-Cleanup):**
  wird nach Löschung des Listelements aufgerufen; bekommt als Parameter den Container (= enthaltenden Knoten)

Die Spezifikation von Kantentypen ist durch folgende Methoden erweiterbar:

1. Für die Darstellung:
   - **Label-Präsenz (Label Presence):**
     gibt an, ob ein Label für die Kante angezeigt wird; Das Label einer Kante ist ein spezielles Accessory. Wird nicht verwendet, falls (in der Spezifikation) festgelegt ist, dass Kanten dieses Typs keinen Namen besitzen.
   - **Objekt-Label-Text (Object Label Text):**
     zur Berechnung des Labels der Kante; Standardmäßig wird nur der Name des Kante als Label angezeigt; Das Label einer Kanten ist ein spezielles Accessory. Wird nicht verwendet, falls (in der Spezifikation) festgelegt ist, dass Kanten dieses Typs keinen Namen besitzen oder falls es mit der Methode für „Label Presence“ verhindert wird.
   - **Linien-Stärke (Line Width):**
     Zeichen-Stärke der Kantelinie; Wird nur verwendet, falls in der Kanten-Spezifikation für die Linien-Stärke „Custom“ eingestellt ist.
   - **Linien-Stil (Line Style):**
     Zeichen-Stil (durchgezogen, gestrichelt, gepunktet, etc.) der Kanten-Linie; Wird nur verwendet, falls in der Kanten-Spezifikation für den Linien-Stil „Custom“ eingestellt ist.
   - **Linien-Anzahl (Line Count):**
     Zahl der Linien der Kante; bei Anzahl größer Eins werden statt einer Linie der angegebenen Zahl entsprechend viele, parallele Linien in geringem Abstand für die Kante gezeichnet; Wird nur verwendet, falls in der Kanten-Spezifikation für die Linien-Anzahl „Custom“ eingestellt ist.
   - **Füll-Muster (Paint Pattern):**
     „Pinselstärke“ für Zeichnung der Kantenlinie (inkl. Label-Text): entweder „Voll“ (solid) oder „50%“; Wird nur verwendet, falls in der Kanten-Spezifikation für das Füll-Muster „Custom“ eingestellt ist.
   - **Ursprungs-Spitzen-Presence (Origin Head Presence):**
     gibt an, ob am Beginn der Kantenlinie eine Spitze angezeigt wird; Wird nur verwendet, falls in der Kanten-Spezifikation für die Präsenz der Spitze (am Ursprung) „Custom“ eingestellt ist.
   - **Ursprungs-Spitzen-Stil (Origin Head Style):**
     wählt Form der Spitze am Beginn der Kantenlinie aus; Wird nur verwendet, falls in der Kanten-Spezifikation für die Form der Spitze (am Ursprung) „Custom“ eingestellt ist und eine nur dann, wenn überhaupt eine Spitze am Beginn angezeigt werden soll.
   - **End-Spitzen-Presence (Destination Head Presence):**
     gibt an, ob am Ende der Kantenlinie eine Spitze angezeigt wird; Wird nur verwendet, falls in der Kanten-Spezifikation für die Präsenz der Spitze (am Ende) „Custom“ eingestellt ist.
   - **End-Spitzen-Stil (Destination Head Style):**
     wählt Form der Spitze am Ende der Kantenlinie aus; Wird nur verwendet, falls in der Kanten-Spezifikation für die Form der Spitze (am Ende) „Custom“ eingestellt ist und eine nur dann, wenn überhaupt eine Spitze am Ende angezeigt werden soll.
2. Für Constraints/Aktionen bei Erstellung/Löschung:
   - **Erstellungsprüfung (Creation Check):**
     prüft zusätzlich zu den Angaben im Metamodell selbst, ob eine Kante (vom Benutzer oder von Alter) erstellt werden darf; bekommt als Parameter den Container (= stets das Modell), sowie

eine Liste, die Ursprung und Ende (d.h. jeweilige Knoten oder Listelemente) enthält; Wird auch auch bei „Verschiebung“ (Änderung von Ursprung oder Ende) (durch den Benutzer) verwendet.

- **Erstellungs-Post-Aktion (Creation Cleanup):**
  unmittelbar nach Erstellung der Kante aufgerufen mit ihr als Parameter; für weitere Initialisierung; eine Art Konstruktor-Erweiterung

- **Zerstörungsprüfung (Deletion Check):**
  prüft ob eine Kante (vom Benutzer) gelöscht werden darf; bekommt als Parameter die Kante

- **Zerstörungs-Post-Aktion (Deletion-Cleanup):**
  wird nach Löschung einer Kante aufgerufen; bekommt als Parameter den Container (= enthaltenden Knoten), sowie Ursprung und Ende (jeweilige Knoten oder Listelemente)

Auch der Spezifikationsknoten des Graphmodells kann mit Methoden erweitert werden. Graphen selbst benötigen aber keine Darstellungs-Methoden:

- **Erstellungs-Post-Aktion (Creation Cleanup):**
  unmittelbar nach Erstellung des Modells aufgerufen mit ihm als Parameter; für weitere Initialisierung; eine Art Konstruktor-Erweiterung

- **Zerstörungsprüfung (Deletion Check):**
  prüft ob ein Modell bzw. sein Edit-Fenster geschlossen werden kann

Für die Spezifikationen von Properties innerhalb der Spezifikationsknoten von Graphobjekten oder Graphmodell können ebenfalls Methoden angegeben werden:

- **Gültige Werte (Valid Values):**
  gibt Liste der Werte an, die der Benutzer für die Property auswählen kann; Diese Protodome-Methode wird nur bei Properties benutzt, die als Wert-Typ eine benutzer-definierte Aufzählung, einen Graphobjekt-Typ oder den allgemeinen Typ `Object` eingestellt haben. Die Liste sollte nur Werte enthalten, die dem in der Property-Spezifikation festgelegten Daten-Typ entsprechen.

- **Wächter-Bedingung (Guard Condition):**
  prüft, ob der Wert der Property verändert werden darf; bekommt als Parameter das Grapething, dem die Property gehört, die Definition der Property selbst und den neu gewünschten Wert; Wird vor jeder Änderung des Property-Wertes aufgerufen, nicht nur wenn die Änderung durch den Benutzer verursacht wurde.

- **Post-Aktion (Post Action):**
  wird unmittelbar nach Änderung des Wertes der Property aufgerufen; bekommt als Parameter das Grapething, dem die Property gehört, die Definition der Property selbst, sowie den neu und den alten Wert der Property; Wird vor jeder Änderung des Property-Wertes aufgerufen, nicht nur wenn die Änderung durch den Benutzer verursacht wurde.

Zusätzlich zu den Protodome-Methoden kann in einem Metamodell noch der Name eines Alter-Codefile angegeben werden, das bei der Erstellung eines neuen Modells bzw. beim Laden (aus einer Datei) eines vorhandenen Modells geladen wird. Dieses Codefile wird im Folgenden als **Alter-Supportfile** des Modelltyps bezeichnet. Es kann beliebigen Alter-Code ausführen, vor allem für das neu erstellte bzw. geladene Modell, d.h. für dessen Modelltyp, Prozeduren und Operationen definieren. Die darin definierten Prozeduren und Operationen können z.B. auch von den Protodome-Methoden aus aufgerufen werden. Auch können andere Alter-Codefiles vom Supportfile aus geladen werden.

Außerdem kann man im Metamodell noch eine Methode definieren, die beim Laden eines vorhandenen Modells aufgerufen wird, d.h. wenn ein vorher in einer Datei abgespeichertes Modell wieder in den Speicher geladen wird:

- **Post-Laden (Post Loading):**
  wird direkt nach Laden eines Modells, d.h. genauer nach dem Laden der Graphmodellstruktur aus seiner und Datei und dem Laden des Alter-Supportfiles, aufgerufen mit dem geladenen Graphmodell

als Argument; Hier sind z.B. die Registrierung von zusätzlichen Laufzeit-Beziehungen zwischen den enthaltenen Graphobjekten möglich, diese Methode wird nur für den hierarchisch höchsten Teil (top-level-graph) eines Modells, d.h. nicht für Submodelle, aufgerufen.

Ein Metamodell enthält zusätzlich zu den Spezifikationen für Grapethings und den Beziehungen unter diesen die Definition der Button-Toolbar, die in Editor-Fenstern von Modellen des spezifizierten Modelltyps verwendet wird. Diese Toolbar kann zusätzlich zu **Standard-Buttons** für Verschieben, Objekt-Anwählen etc. sowohl sogenannte **Creation-Buttons**, die speziell für die Erstellung einer bestimmten Graphobjekt-Klasse bestimmt sind, als auch **Custom-Buttons**, die für beliebige Aktionen gedacht sind, enthalten. Das Verhalten der Creation- und Custom-Buttons kann dabei ebenfalls durch einige Protodome-Methoden angepasst werden:

- **Zulassung von Creation-Buttons (Enabling bei Creation-Buttons):**
  wird aufgerufen, um zu bestimmen, ob der jeweilige Creation-Button vom Benutzer verwendet werden kann, ob also mit dem zugehörigen Button Graphobjekte der dem Button zugeordneten Graphobjekt-Klasse erschaffen werden können; bekommt als Parameter nur das Graphmodell; Als Rückgabe sollte ein entsprechender boolscher Wert zurückgegeben werden.

- **Zulassung von Custom-Buttons (Enabling bei Custom-Buttons):**
  wird aufgerufen, um zu bestimmen, ob der jeweilige Custom-Button vom Benutzer verwendet werden kann; bekommt als Parameter nur das Graphmodell; Als Rückgabe sollte ein entsprechender boolscher Wert zurückgegeben werden.

- **Aktion von Custom-Buttons (Action bei Custom-Buttons):**
  wird bei Auslösung des zugehörigen Custom-Buttons ausgeführt; kann beliebige Aktion durchführen; Parameter sind das Graphmodell, sowie die Position des Mauszeigers innerhalb des Graphen.

Weiterhin können im Metamodell Menüs definiert werden, die innerhalb der Edit-Fenster eines Modells entweder den globalen Standard-Menüs oder den Menüs für die verschiedenen Graphobjekt-Klassen zugeordnet werden. Die Menüeinträge können ähnlich zu Custom-Buttons beliebige Aktionen auslösen. Um diese Einträge genauer festzulegen, gibt es ebenfalls einige Protodome-Methoden:

- **Zulassung (Enabling):**
  wird aufgerufen, um zu bestimmen, ob der jeweilige Menüeintrag vom Benutzer verwendet werden kann; Bekommt als einzigen Parameter das Graphmodell, falls der Menüeintrag von einem globalen Menü aus aufgerufen wurde, bzw. das zugehörige Graphobjekt, falls der Menü-Eintrag von dem Menü des Graphobjekts aus aufgerufen wurde. Als Rückgabe sollte ein entsprechender boolscher Wert zurückgegeben werden.

- **Aktion (Action):**
  wird bei Auslösung des zugehörigen Menüeintrages ausgeführt; kann beliebige Aktion durchführen; Einziger Parameter ist das Graphmodell, falls der Menüeintrag von einem globalen Menü aus aufgerufen wurde, bzw. das zugehörige Graphobjekt, falls der Menüeintrag von dem Menü des Graphobjekts aus aufgerufen wurde. Die Position des Mauszeigers innerhalb des Graphen wird nicht übergeben.

A.1.4 Grapething-Interests in DoME

Außer den Protodome-Methoden bietet *DoME* noch als eine weitere wichtige Synchronisationsmöglichkeit sogenannte **Grapething-Interest** an. Ein beliebiges Grapething kann hierbei bei einem beliebigen anderen Grapething ein Interesse für die Änderung eines bestimmten **(Grapething-Interest-)Aspekts** anmelden. Alter hält die Operation `add-interest` für das Anmelden und entsprechend die Operation `remove-interest` für das Abmelden bereit (siehe auch [AlterManual]). Die Aspekte sind pro Typ des Grapethings, bei sich das andere Grapething anmeldet, vorgegeben.

Bei der Anmeldung durch `add-interest` wird eine Alter-Prozedur (der sogenannte **Grapething-Interest-Handler** oder kurz **Interest-Handler**) übergeben, die für das interessierte Grapething bei Änderung des Aspekts aufgerufen wird. Dieser erhält bei Aufruf zusätzlich zu dem interessierten Grapething als Parameter das Grapething, bei dem die Anmeldung erfolgte, den Namen des geänderten Aspektes und einen weiteren Wert, der vom Aspekt abhängt. Der letzte, aspektabhängige Parameter wird im Folgenden als **(Grapething-Interest)-Handler-Argument** bezeichnet.

Der Aufruf der Interest-Handler erfolgt i.A. nach erfolgter Änderung des jeweiligen Aspekts, nicht vorher. Das Handler-Argument enthält in der Regel den alten bzw. den neuen Wert des Aspekts vor bzw. nach der Änderung.

Für Grapethings allgemein gibt es die folgenden Aspekte (Aspekte sind in Alter als Strings dargestellt, d.h.sie sind case-sensitiv):

- **name:** Änderung des Namens
- **displaySignificantProperty:** Änderung des Wertes einer Property, für die im Metamodell Anzeigesignifikanz definiert wurde, (d.h. bei ihrer Änderung wird die Darstellung ihres Grapethings erneuert); das Handler-Argument enthält den Namen der geänderten Property
- **addedComponent:** bei Hinzufügung einer Komponente des Grapethings, d.h. bei Graphmodellen: Kante oder Knoten hinzugefügt, bei Knoten: Listelement oder Knoten als Inhalt oder Accessory hinzugefügt, bei Kanten: Knoten als Accessory hinzugefügt; das Handler-Argument ist eine Referenz auf die neue Komponente
- **removedComponent:** bei Entfernung einer Komponente (vgl.„addedComponent“) des Grapethings; wird vor der Entfernung aufgerufen; das Handler-Argument ist die zu löschende Komponente
- **delete:** bei Löschung des Grapethings selbst; wird vor der Löschung aufgerufen

Bei Graphobjekten allgemein kommen Aspekte hinzu:

- **position:** Änderung der Position des Grapethings; Handler-Argument ist die neue Position (`point` in Alter)
- **bounds:** Änderung der Ausdehnung des Grapethings; Handler-Argument ist die neue Ausdehnung (`rectangle` in Alter)
- **container:** Änderung des enthaltenden Containers (Graphmodell oder Knoten bzw. Kante)
- **addedSubdiagram:** Hinzufügen eines Submodells; Handler-Argument ist das hinzugefügte Submodell
- **removedSubdiagram:** bei Entfernung eines Submodells; wird vor der Entfernung aufgerufen; Handler-Argument ist das zu entfernende Submodell
- **color:** bei Änderung der Farbe des Grapethings; Handler-Argument ist der neue Farb-Wert

Speziell Kanten haben noch die folgenden Aspekte:

- **originConnection:** Änderung des Kanten-Ursprungs, d.h. des Graphobjekts mit dem der Anfang der Kante verbunden ist; Handler-Argument ist der alte Kanten-Ursprung
- **destinationConnection:** Änderung des Kanten-Endes, d.h. des Graphobjekts mit dem das Ende der Kante verbunden ist; Handler-Argument ist das alte Kanten-Ende

Auch für Graphmodelle selbst gibt es einige spezielle Interest-Aspekte:

- **filename:** Änderung des Namens der Datei, in der das Graphmodell gespeichert ist
- **modified:** Änderung des Änderungs-Status des Graphmodells; die Anzeige das Graphmodell stimmt nicht mehr mit dem Inhalt der Datei, in der es gespeichert ist, überein

Für jedes Property eines Grapethings gibt es auch jeweils einen Interest-Aspekt. Der Aspekt ist dabei gleich dem Namen der Property.

A.1.5 spezielle Properties in DoME

In *DoME* werden bei den verschiedenen Arten von Grapethings einige spezielle,interne Properties verwendet. Die Namen dieser sollten nicht für eigene, benutzer-definierte Properties benutzt werden. Daher werden diese speziellen Properties kurz vorgestellt.

Für Grapethings allgemein gibt es die folgenden speziellen Properties:

- description
- rationale
- anchor
- uplink
- specification
- propertySchemaFiles

Graphobjekte haben weiterhin:

- frozencolor
- overlays

Knoten und Listelemente haben zusätzlich diese Properties:

- flippedHorizontally
- defaultBinding

Kanten haben zusätzlich noch folgende spezielle Properties:

- showOriginHead
- showDestinationHead

Auch Graphmodelle selbst haben einige intern verwendete Properties:

- initialOpenGraphs
- originOffset
- windowBounds
- overlays
- activeOverlays
- showCommented
- userSettings
- formatVersion
- configurations
- usePrintSize
- printScale
- printFormat
- printToFile

- printer
- printSize
- printLandscape
- printName
- printSubdiagrams

Aufgrund der Internas von *DoME* sollten außerdem die Name `type` und `name` als Property-Name vermieden werden.

A.1.6 Details zur Modellierung mit Protodome und Alter

Die Modellierung mittels Protodome und Alter weist einige kleinere Aspekte bzw. Probleme auf, die vor allem die Übersichtlichkeit des Metamodells betreffen. Da die Kenntnis dieser Aspekte eine Anpassung oder Erweiterung des Werkzeuges erleichtern, sollen sie hier kurz besprochen werden.

In einem Metamodell für einen protodome-basierten Modelltyp kann bei jeder Protodome-Methode statt von Alter-Code auch ein Smalltalk-Operationsaufruf (d.h. der Smalltalk-Selektor einer Smalltalk-Operation für das betreffenden Grapething) angegeben werden. Hierbei werden nur der Name der Smalltalk-Operation (der Selektor) und keine zusätzlichen Operationsparameter verwendet.

Besonders sinnvoll kann dies bei der Protodome-Aktions-Methode eines Custom-Button eingesetzt werden: Verwendet man dort den Smalltalk-Selektor `newArc` (mit der gegebenen Groß/Kleinschreibung, da in Smalltalk diese eine Rolle spielt) anstatt von Alter-Code, so können mit diesem Custom-Button Kanten unterschiedlicher Klassen erzeugt werden (im Gegensatz zu normalen Creation-Buttons). Die Auswahl der Kanten-Klasse erfolgt jeweils nachdem der Benutzer (bei Verwendung dieses Custom-Buttons) sowohl Ziel- als auch End-Punkt der zu erstellenden Kante ausgewählt hat und zwar anhand der Graphobjekt-Klassen von Ziel und End-Punkt. Gibt es mehrere, mögliche Kanten-Klassen, so richtet sich die Auswahl dabei nach der Reihenfolge, in der die Kanten-Klassen-Spezifikationen im Metamodell erstellt wurden. Diese Erstellungsreihenfolge der Kanten kann vom Benutzer beispielsweise bei der Wahl der Kanten-Klasse für ein Verbindungs-Constraint abgelesen werden.

Verwendet man also `newArc` bei einem Custom-Button, so ist ggf. auf die Reihenfolge der Erstellung von Kanten-Klassen Rücksicht zu nehmen. Unter Umständen muss man die bisherigen Kanten-Klassen-Spezifikationen löschen und neu in der richtigen Reihenfolge erstellen.

Bei einem Verbindungs-Constraints in einem Metamodell wird mit angegeben, für welche Kanten-Klasse es gilt. Die Vererbungsbeziehung von Kanten-Klassen wird hierbei berücksichtigt, aber jeweils nur eine Vererbungsebene weit, d.h. auf direkte Unterklassen der gewählten Kanten-Klasse trifft das Verbindungs-Constraint auch zu, nicht aber auf die indirekten Unterklassen. Das heißt ggf. müssen für indirekte Unterklassen eigene Verbindungs-Constraints festgelegt werden. Dies macht unter Umständen aber das Metamodell unübersichtlicher.

Ein ähnlicher Aspekt ist der folgende: Für Knoten und Listelemente gibt es keine eigene Spezifikationsknoten-Oberklasse, welche die Verbindungs-Constraints allgemein zu ihnen erlaubt. Das heißt Verbindungs-Constraints müssen jeweils eigens für Knoten und Listelement festgelegt werden. Es sind also jeweils zwei eigene Verbindungs-Constraints nötig. Auch dies kann die Übersichtlichkeit des Metamodells beeinträchtigen.

A.2 Entwickelte, generische Alter-Codefiles für das Werkzeug

Die folgendende Liste zeigt eine Aufstellung aller generischen Alter-Codefiles:

- **mnm-support.lib:** nur ein Wrapper, lädt alle anderen generischen Codefiles

- **mnm-defines.lib:** einige grundsätzlich Definitionen für Datei- und Verzeichnis-Namen (alle relativ zum Tool-Verzeichnis)
- **mnm-utilities.lib:** allgemeine Hilfsfunktionen
- **mnm-math.lib:** mathematische Funktionen, wie z.B. für Mengen, Intervalle
- **mnm-geometry.lib:** Funktionen für Punkte, Rechtecke und andere Geometrieberechnungen
- **mnm-string.lib:** Stringberechnungen und -Formatierungen
- **mnm-file.lib:** Hilfsfunktionen für Dateizugriffe
- **mnm-default-name.lib:** Berechnung von Namen für neu erstellte Graphobjekte
- **mnm-grape.lib:** allgemeine Funktionen für Zugriff/Änderung von Modellen, vor allem bezogen auf Graphobjekte
- **mnm-graphmodel.lib:** Funktionen für Verwaltung von Modellen und Submodellen: Erstellung, Navigation etc.
- **mnm-properties.lib:** allgemeine Funktionen für Zugriff auf Properties von Modellen und Graphobjekten
- **mnm-general.lib:** allgemeine Funktionen für die Kopplung von im Metamodell eingestellten Alter-Definitionen und den in den Codefiles enthaltenen Definitionen, die sich mit Constraints und Handlern für das Erstellen und das Löschen von Modellen und Graphobjekten befassen. Die hier definierten Methoden werden ggf. von spezifischen Modelltypen überschrieben.
- **mnm-layout.lib:** allgemeine Funktionen für die Kopplung von im Metamodell spezifizierten Alter-Definitionen und den in den Codefiles enthaltenen Definitionen, die das Layout von Graphobjekten näher beschreiben. Hier definierte Methoden werden ggf. in den spezifischen Modelltypen überschrieben.
- **mnm-interests.lib:** Funktionen für das Einrichten und Löschen von Interest-Handlern (zur Reaktion auf Änderung verschiedener Aspekte innerhalb eines Graphobjektes bzw. eines Modells selbst) zwischen Graphobjekten
- **mnm-arc.lib:** allgemeine Hilfsfunktionen für Kanten
- **mnm-message.lib:** Funktionen für Message-Boxen
- **mnm-grapheditor.lib:** Funktionen für Anpassung des Editors eines Modells
- **mnm-custom-button.lib:** einige Hilfsfunktionen für Custom-Tool-Buttons
- **mnm-myview.lib:** spezielles Konzept für Views von Graphobjekten
- **mnm-generalization.lib:** einige Hilfsfunktionen für Kanten/Knoten-Beziehungen die eine Generalisierung modellieren (z.B. in UML)

A.3 Protodome-Methoden-Realisierung im Werkzeug

Für die in Spezifikationsknoten für Grapethings bzw. deren Properties definierbaren Alter-Protodome-Methoden (siehe Anhang A.1.3) wird bei allen Modelltypen für das Werkzeug ein standardisiertes Schema verwendet: In der Regel wird der eigentliche Alter-Code für eine Protodome-Methode nicht im Metamodell selbst, sondern in den durch das Supportfiles geladenen Alter-Codefiles definiert. Im Metamodell ist als Alter-Code für die Protodome-Methode nur der Aufruf einer Prozedur oder einer weiteren Methode, die in einem der Alter-Codefiles definiert wird, enthalten.

Im Folgenden werden die Protodome-Methoden sowie einige sie weiter unterstützende Operationen im Einzelnen besprochen. Für Protodome-Methoden wird jeweils der Alter-Code genannt, der jeweils im Metamodell anzugeben ist. Hierbei (evtl. indirekt) aufgerufene Operationen werden genauer erläutert.

Protodome-Methoden und Operationen für das Graphmodell selbst (alle Operationen sind hierbei in der Regel im Alter-Codefile `mnm-general.lib` definiert):

- **Protodome-Methode für Erstellungs-Post-Aktion (Creation Cleanup):**
  Code: `(creation-cleanup self)`
- **Operation `(creation-cleanup self)`:**
  kann für jeden Graphmodelltyp durch eine eigene Methode implementiert werden, um den Graphen weiter zu initialisieren (bestimmte Knoten automatische erstellen etc.) und um ähnlich wie beim Post-Loading Grapething-Interest-Handler zwischen beliebigen Graphobjekten zu registrieren.
- **Protodome-Methode für Post-Laden (Post Loading):**
  Code `(post-load self)`
- **Operation `(post-load self)`:**
  gedacht zur Registrierung von Grapething-Interest-Handlern zwischen Graphobjekten; Standard ist für den Graphen selbst die Methode `register-dependencies` aufzurufen.
- **Operation `(post-load-sub self)`:**
  Operation, die standardmäßig als Wrapper für den Aufruf von `post-load` dient; Vor dem Aufruf von `post-load` wird jedoch ggf. das Supportfile des Modells geladen, da dies von Protodome nicht automatisch gemacht wird.
- **Operation `(register-dependencies self)`:**
  u.a. im Codefile `mnm-interests.lib` definiert; Standard ist für alle Kanten und alle (direkt) enthaltenen Knoten die Methode `register-dependencies` selbst aufzurufen, die speziell für die (rekursive) Registrierung von Grapething-Interest-Handlern bei Laden oder Erstellung des Modells gedacht ist.
- **Operation `(deregister-dependencies self)`:**
  u.a im `mnm-interests.lib` definiert; zur Abmeldung von Grapething-Interest-Handlern; Es sollten hierbei alle Interest-Handler zu anderen Grapethings (die vor allem durch `register-dependencies` erstellt wurden) wieder gelöscht werden; Standard ist die Operation rekursiv auf enthaltenen Graphobjekten auszuführen.

Protodome-Methoden und Operationen bei Graphobjekten für Erzeugungs/Zerstörungsprüfungen bzw. Aktionen (alle Operationen sind hierbei in der Regel im Alter-Codefile `mnm-general.lib` definiert):

- **Operation `(register-dependencies self)`:**
  speziell für die (rekursiven) Registrierung von Grapething-Interest-Handlern bei Laden oder Erstellung des Modells gedacht; Standard ist für alle enthaltenen Knoten (oder Listelemente), sowie die Methode `register-dependencies` selbst aufzurufen. Standardmäßig werden keine eigenen Registrierungen gemacht.
- **Operation `(deregister-dependencies self)`:**
  zur Abmeldung von Grapething-Interest-Handlern; Es sollten alle Interest-Handler zu anderen Grapethings (die vor allem durch `register-dependencies` erstellt wurden) wieder gelöscht werden; Standard ist die Operation rekursiv auf enthaltenen Graphobjekten und Submodellen auszuführen.
- **Protodome-Methode für Erstellungs-Post-Aktion (Post Creation):**
  Code: `(creation-cleanup self)`
- **Operation `(creation-cleanup self)`:**
  initialisiert das neue Graphobjekt; Standard ist einige weitere Operationen aufzurufen: `init-name`

, `init-pre`, `register-dependencies` und `init-childs`. wird in der Regel selbst nicht überschrieben, sondern statt dessen die aufgerufenen Methoden

- **Operation `(init-name self)`:**
  zur Festlegung eines Standardnamens, der neuem Graphobjekt zugewiesen wird, falls das Graphobjekt einen Namen besitzen kann. Verwendet hierzu standardmäßig `default-name`.
- **Operation `(default-name self)`:**
  zur Ermittlung eines Standardnamens
- **Operation `(init-pre self)`:**
  zur Initialisierung des Graphobjekts, bevor Grapething-Interests angemeldet werden (`register-dependencies` aufgerufen wird); z.B. Festlegung von Property-Werten; Standard ist nicht zu tun.
- **Operation `(init-childs self)`:**
  zur Initialisierung des Graphobjekts, nachdem Grapething-Interest-Handler angemeldet wurden (`register-dependencies` aufgerufen wurde), z.B. Erstellung von enthaltenen Knoten, Accessory-Knoten oder Submodellen; Standard ist nicht zu tun.
- **Protodome-Methode für Zerstörungsprüfung (Deletion Check):**
  Code: `(deletion-check-and-prepare self)`
- **Operation `(deletion-check-and-prepare self)`:**
  standardmäßig als Wrapper für `deletion-check` und `deletion-prepare` gedacht; `deletion-check` wird dabei aufgerufen, um die tatsächlichen Zerstörungsprüfung durchzuführen; erlaubt dieses dann anhand des boolschen Rückgabestatus die Zerstörung, so wird für Vorbereitungen noch `deletion-prepare` aufgerufen, ansonsten die Zerstörung abgelehnt; Wird im allgemeinen nicht überschrieben.
- **Operation `(deletion-check self)`:**
  führt die tatsächliche Zerstörungsprüfung durch; Standard ist die Zerstörung zu erlauben (true zurückzugeben).
- **Operation `(deletion-prepare self)`:**
  für Vorbereitungen vor der Zerstörung des Graphobjekts; z.B. dem Abmelden von Grapething-Interest-Handlern; Standard ist nur `deregister-dependencies` aufzurufen.
- **Protodome-Methode bei Knoten für Erstellungsprüfung (Creation Check):**
  Code: `(creation-check-node container <classname> args)`, wobei `<classname>` die Knoten-Klasse in Form eines Alter-Symbols.
- **Operation `(creation-check-node container classname args)`:**
  Prüfung ob ein Knoten erstellt werden darf; `classname` beschreibt als Alter-Symbol die Klasse des zu erstellenden Knoten, `args` ist eine Liste die nur die Position innerhalb des Containers beschreibt; Standard ist die Erstellung zu erlauben (true zurückzugeben).
- **Protodome-Methode bei Listelementen für Erstellungsprüfung (Creation Check):**
  Code: `(creation-check-listelement container <classname> args)`, wobei `<classname>` die Listelement-Klasse in Form eines Alter-Symbols angibt.
- **Operation `(creation-check-listelement container classname args)`:**
  Prüfung, ob ein Listelement erstellt werden darf; `classname` beschreibt als Alter-Symbol die Klasse des zu erstellenden Listelements, `args` ist eine Liste die nur die Position innerhalb des Containers beschreibt; Standard ist die Erstellung zu erlauben (true zurückzugeben).
- **Protodome-Methode bei Kanten für Erstellungsprüfung (Creation Check):**
  Code: `(creation-check-arc container <classname> args)`, wobei `<classname>` die Kanten-Klasse in Form eines Alter-Symbols angibt.
- **Operation `(creation-check-arc container classname args)`:**
  Prüfung, ob eine Kante erstellt werden darf; `classname` beschreibt als Alter-Symbol die Klas-

se der zu erstellenden Kante, `args` ist eine Liste die Ursprung und Ende (jeweilige Knoten- oder Listelement-Objekte) enthält; Standard ist die Erstellung zu erlauben (true zurückzugeben).

- **Protodome-Methode bei Knoten für Zerstörungs-Post-Aktion (Deletion Cleanup):**
Code: `(deletion-cleanup-node container <classname>)`, wobei `<classname>` die Knoten-Klasse in Form eines Alter-Symbols angibt.

- **Operation `(deletion-cleanup-node container classname)`:**
Aktion, die nach Zerstörung eines Knoten durchgeführt wird; `classname` beschreibt als Alter-Symbol die Klasse des zu erstellenden Knoten; Standard ist nichts zu tun.

- **Protodome-Methode bei Listelementen für Zerstörungs-Post-Aktion (Deletion Cleanup):**
Code: `(deletion-cleanup-listelement container <classname>)`, wobei `<classname>` die Listelement-Klasse in Form eines Alter-Symbols angibt

- **Operation `(deletion-cleanup-listelement container classname)`:**
Aktion, die nach Zerstörung eines Listelements durchgeführt wird; `classname` beschreibt als Alter-Symbol die Klasse des zu erstellenden Listelements; Standard ist nichts zu tun.

- **Protodome-Methode bei Kanten für Zerstörungs-Post-Aktion (Deletion Cleanup):**
Code: `(deletion-cleanup-arc container <classname> (list old-origin old-destination)`, wobei `<classname>` die Kanten-Klasse in Form eines Alter-Symbols angibt

- **Operation `(deletion-cleanup-arc container classname args)`:**
Aktion, die nach Zerstörung einer Kante durchgeführt wird; `classname` beschreibt als Alter-Symbol die Klasse der zu erstellenden Kante, `args` ist eine Liste die Ursprung und Ende (jeweilige Knoten- oder Listelement-Objekte) enthält; Standard ist nichts zu tun.

Die Standards der Operationen für die Darstellung von Graphobjekten sind alle im Codefile `mnm-layout.lib` definiert und werden von den Definitionen für die einzelnen Modelltypen ggf. ganz oder teilweise überschreiben.

Protodome-Methoden für die Darstellung von Knoten:

- **Protodome-Methode für Objekt-Label-Text (Object Label Text):**
Code: `(layout-object-label-text self)`, Standard ist den Namen des Knoten zurückzugeben, falls er einen besitzt.

- **Protodome-Methode für Linien-Stärke (Line Width):**
Code: `(layout-line-width self)`, Standardmäßig wird `1` für normale Linienstärke zurückgegeben.

- **Protodome-Methode für Linien-Stil (Line Style):**
Code: `(layout-line-style self)`, Standardmäßig wird das Alter-Symbol `normal` für normale durchgezogene Linie zurückgegeben.

- **Protodome-Methode für Linien-Anzahl (Line Count):**
Code: `(layout-line-count self)`, Standardmäßig wird `1` als Anzahl zurückgegeben.

- **Protodome-Methode für Füll-Muster (Paint Pattern):**
Code: `(layout-paint-pattern self)`; Standardmäßig wird das Alter-Symbol `solid` für normale Pinsel-Stärke zurückgegeben.

- **Protodome-Methode für Poly-Linien-Stil (Poly Line Style):**
Code: `(layout-polyline-style self)`, Standardmäßig wird das Alter-Symbol `default` für normale Rechtecksform zurückgegeben.

- **Protodome-Methode für Poly-Linien-Punkte-Array (Polyline Point Array):**
Code: `(layout-polyline-point-array self)`, Standardmäßig wird eine leere Liste zurückgegeben.

- **Protodome-Methode für Namens-Position (Name Position):**
  Code: `(layout-name-position self)`, Standardmäßig wird das Alter-Symbol `inside-top` (für Darstellung des Labels im Inneren des Knotens oben, zentriert) zurückgegeben.

- **Protodome-Methode für Knoten-Ausdehnung (Node Bounds):**
  Code: `(layout-node-bounds self)`, Standardmäßig wird nichts (`nil`) zurückgegeben.

- **Protodome-Methode für Knoten-Form (Node Shape):**
  Code: `(layout-node-shape self context b-bounds pos)`, Standardmäßig wird nichts gezeichnet oder getan.

- **Protodome-Methode für Exzentrität (Eccentricity):**
  Code: `(layout-eccentricity self)`, Standardmäßig wird `1` für normales Höhe/Längen-Verhältnis zurückgegeben.

- **Protodome-Methode für Ecken Typ (Corner Type):**
  Code: `(layout-corner-type self)`, Standardmäßig wird das Alter-Symbol `square` für normale Ecken zurückgegeben.

- **Protodome-Methode für Ecken-Radius-Faktor (Corner Radius Factor):**
  Code: `(layout-corner-radius-factor self)`, Standardmäßig wird `0.375` zurückgegeben.

- **Protodome-Methode für Ansatzpunkt einer Kante (Clipping a Connector):**
  Code: `(layout-clipping-a-connector self connector vertex b-bounds)`, Standardmäßig wird nichts zurückgegeben.

Protodome-Methoden für die Darstellung von Listelementen:

- **Protodome-Methode für Objekt-Label-Text (Object Label Text):**
  Code: `(layout-object-label-text self)`, Standard ist den Namen des Listelements zurückzugeben.

Protodome-Methoden für die Darstellung von Kanten:

- **Protodome-Methoden für Label-Präsenz (Label Presence):**
  Code: `(layout-label-presence self)`, gibt standardmäßig false (`#f`) zurück.

- **Protodome-Methode für Objekt-Label-Text (Object Label Text):**
  Code: `(layout-object-label-text self)`, Standard ist den Namen der Kante zurückzugeben, falls sie einen besitzt.

- **Protodome-Methode für Linien-Stärke (Line Width):**
  Code: `(layout-line-width self)`, Standardmäßig wird `1` für normale Linienstärke zurückgegeben

- **Protodome-Methode für Linien-Stil (Line Style):**
  Code: `(layout-line-style self)`, Standardmäßig wird das Alter-Symbol `normal` für normal durchgezogene Linie zurückgegeben.

- **Protodome-Methode für Linien-Anzahl (Line Count):**
  Code: `(layout-line-count self)`, Standardmäßig wird `1` als Anzahl zurückgegeben.

- **Protodome-Methode für Füll-Muster (Paint Pattern):**
  Code: `(layout-paint-pattern self)`, Standardmäßig wird das Alter-Symbol `solid` für normale Pinselstärke zurückgegeben.

- **Protodome-Methode für Ursprungs-Spitzen-Presence (Origin Head Presence):**
  Code: `(layout-origin-head-presence self)`

- **Protodome-Methode Ursprungs-Spitzen-Stil (Origin Head Style):**
  Code: `(layout-origin-head-style self)`

- **Protodome-Methode für End-Spitzen-Presence (Destination Head Presence):**
  Code: `(layout-dest-head-presence self)`

- **Protodome-Methode End-Spitzen-Stil (Destination Head Style):**
  Code: `(layout-dest-head-style self)`

Protodome-Methoden für Properties von Graphobjekten bzw. des Graphmodells (die Standards sind im Codefile `mnm-layout.lib` definiert):

- **Protodome-Methode für Gültige Werte (Valid Values):**
  Code: `(property-valid-values self property)`

- **Operation `(property-valid-values self property)`:**
  berechnet die Werte, die der Benutzer der durch `property` beschriebenen Property zuweisen kann; Der Name der Property kann mit der Alter-Operation `name` abgefragt werden. Standardmäßig wird `propbyname-valid-values` aufgerufen, um die Auswahl der gültigen Werte direkt an Hand des Property-Namens treffen zu können.

- **Operation `(propbyname-valid-values self propname)`:**
  wird in der Regel von `property-valid-values` aufgerufen, um direkt an Hand des Namens der Property `propname` die Liste der gültigen Property-Werte zu bestimmen. Standardmäßig werden alle möglichen Property-Werte bestimmt und zurückgegeben.

- **Protodome-Methode für Wächter-Bedingung (Guard Condition):**
  Code: `(property-guard-condition self property value)`

- **Operation `(property-guard-condition self property value)`:**
  zur Überprüfung, ob der Wert `value` für die durch `property` beschriebenen Property tatsächlich zulässig ist und die Property auf diesen Wert gesetzt wird oder nicht; Der Name der Property kann mit der Alter-Operation `name` abgefragt werden. Standardmäßig wird `propbyname-guard-condition` aufgerufen, um die Entscheidung direkt an Hand des Property-Namens treffen zu können.

- **Operation `(propbyname-guard-condition self propname value)`:**
  wird in der Regel von `property-guard-condition` aufgerufen, um direkt an Hand des Namens der Property `propname` Änderung zu erlauben oder abzulehnen; Standardmäßig wird jede Änderung erlaubt.

- **Protodome-Methode für Post-Aktion (Post Action):**
  Code: `(property-post-action self property value oldvalue)`

- **Operation `(property-post-action self property value oldvalue)`:**
  wird aufgerufen, nachdem der Werte der durch `property` beschriebenen Property auf `value` gesetzt wurde; Der Name der Property kann mit der Alter-Operation `name` abgefragt werden. Standard ist `propbyname-post-action` aufzurufen, um die Auswahl der Post-Aktion direkt am Property-Namen treffen zu können.

- **Operation `(propbyname-post-action self propname value oldvalue)`:**
  wird in der Regel von `property-post-action` aufgerufen, um direkt an Hand des Property-Namens die durchzuführende Post-Aktion auszuwählen; Standard ist nichts zu tun.

Protodome-Methoden für Toolbar-Buttons und Menü-Einträge: (die Standards sind im Codefile `mnm-general.lib` definiert):

- **Protodome-Methode für Zulassung von Creation-Buttons (Enabling bei Creation-Buttons):**
  Code: `(creation-button-enabling the-graph class-name)`, wobei `class-name` als Alter-Symbol den Namen der Graphobjekt-Klasse enthält, von der mit dem Button Instanzen erstellbar sind.

- **Operation `(creation-button-enabling the-graph class-name)`:**
  wird aufgerufen, um über die Zulassung eines Creation-Buttons zu entscheiden; Standardmäßig wird jeder Button zugelassen.

- **Protodome-Methode für Enabling von Custom-Buttons (Enabling bei Custom-Buttons):**
  Code: `(custom-button-enabling the-graph button-name)`, wobei `button-name` – z.B. als Alter-Symbol – den Custom-Button identifiziert.

- **Operation `(custom-button-enabling the-graph button-name)`:**
  wird aufgerufen, um über die Zulassung eines Custom-Buttons zu entscheiden; Standardmäßig wird jeder Button zugelassen.

- **Protodome-Methode für Aktion von Custom-Buttons (Action bei Custom-Buttons):**
  Code: `(custom-button-action the-graph location button-name)`, wobei `button-name` – z.B. als Alter-Symbol – den Custom-Button identifiziert.

- **Operation `(custom-button-action the-graph location button-name)`:**
  wird bei Auslösung eines Custom-Buttons aufgerufen; Es kann jede beliebige Aktion durchgeführt werden, z.B. kann mit Hilfe der Position des Mauszeigers `location` das Graphobjekt gefunden werden, über dem sich der Mauszeiger befindet. Standardmäßig wird nicht getan.

- **Protodome-Methode für Enabling von Menü-Einträgen (Enabling bei Menü-Einträgen):**
  Code: `(custom-menuitem-enabling grapething menuitem-name)`, wobei `menuitem-name` – z.B. als Alter-Symbol – den Menü-Eintrag identifiziert. `grapething` ist das zum Menü-Eintrag gehörige Grapething, d.h. bei globalen Menü-Eintragen das Graphmodell und sonst das jeweilige Graphobjekt.

- **Operation `(custom-menuitem-enabling grapething menuitem-name)`:**
  wird aufgerufen, um über die Zulassung eines Menü-Eintrags zu entscheiden; Standardmäßig wird jeder Menü-Eintrag zugelassen.

- **Protodome-Methode für Aktion von Menü-Einträgen (Action bei Menü-Einträgen):**
  Code: `(custom-menuitem-action grapething menuitem-name)`, wobei `menuitem-name` – z.B. als Alter-Symbol – den Menü-Eintrag identifiziert. `grapething` ist das zum Menü-Eintrag gehörige Grapething, d.h. bei globalen Menü-Eintragen das Graphmodell und sonst das jeweilige Graphobjekt.

- **Operation `(custom-menuitem-action grapething menuitem-name)`:**
  wird bei Auslösung eines Menü-Eintrages aufgerufen; Es kann jede beliebige Aktion durchgeführt werden. Standardmäßig wird nichts getan.

A.4 Standardmäßig im Werkzeug verwendete Grapething-Properties

In den entwickelten Modelltypen wurden für die verschieden Arten von Grapethings eine Reihe von Properties definiert, die in der Regel in allen Modelltypen verwendet werden. Diese werden für unterschiedliche, allgemeine Zwecke eingesetzt.

Bei allen Grapethings (alle Graphobjekte und Graphmodelle selbst) allgemein werden folgende Properties verwendet:

- **init-status:** String, der globalen Initialisierungs-Status (Phase des Lebenszyklus) des Objekts (Erstellung/Zerstörung etc.) anzeigt

- **intern-status:** String, gibt temporären Status für verschiedene Locking-Mechanismen an

- **mylock:** assoziatives Array (Dictionary in Alter), für Locking-Mechanismen

- **mymaster und myview:** für eigenes View-Konzept, meist zwischen den Listelementen in einem ersten Knoten und den Knoten in einem Submodel des ersten Knoten
- **mydata:** assoziatives Array (hierarchische Keys) für beliebige Zwecke; (es wird eine verschachtelte Liste in Alter (Liste aus (Key,value)-Paaren) statt einem Dictionary verwendet, da Referenzen auf Grapethings in einem Dictionary beim Laden eines Modells nicht richtig wiederhergestellt werden)

Für Graphmodelle werden speziell die folgenden Properties verwendet:

- **graph-type und graph-kind:** unterscheidet verschiedene Typen oder Teilaspekte von Modellen des gleichen Modelltyps, d.h. unterschiedliche Anwendungsmöglichkeiten des Modelltyps z.B. in UML die verschiedenen Diagrammtypen
- **artifact-interest:** assoziatives Array ähnlich wie mydata, aber speziell für die Anmeldung von Interesse an speziellen Teilaspekten des Modells; Ist ähnlich zu Grapething-Interests, wird jedoch vom Modelltyp selbst realisiert; wird nicht bei allen Modelltypen verwendet.

Für Graphobjekte gibt es zusätzlich diese Properties:

- **display-status:** Integer-wertige Property, die nur für das Auffrischen der Darstellung gedacht ist. Ändert sich ihr Wert, wird die Darstellung des Graphobjekts erneuert. Dafür muss aber Anzeigesignifikanz im Metamodell bei ihrer Definition aktiviert werden.

A.5 Beschreibung von Operationen bzw. Prozeduren der Workflowsteuerung

A.5.1 Workflowmodule und Workflowmodul-Prozeduren

Die Namen der Alter-Codefiles für Workflowmodule setzen sich wie folgt zusammen: Das Präfix „mnm-wfm-“, gefolgt von dem eigentlichen Namen des Moduls, der im Workflowspezifikations-Modell festgelegt ist und die Endung „.lib“.

In diesen Codefiles werden speziell festgelegte Prozeduren definiert. Um dabei verschiedene Workflowmodule zu unterscheiden wird jeweils ein eigenes Alter-Paket (ein eigener Namensraum) verwendet. Der Name des Pakets besteht hierbei aus dem Präfix „MNMService-wfm-ünd dem eigentlichen Modul-Namen.

Im weiteren werden die zu spezifizierenden Alter-Prozeduren einzeln betrachtet. Jede von ihnen erhält als ersten Parameter die ihr zugeordnete Aktivität und ggf. weitere Argumente.

- **Prozedur `(register-dependencies self)`:**
  zum Registrieren von eigenen, speziellen Grapething-Interest-Handlern; wird von der `register-dependencies`-Operation des Aktivitäts-Knotens aufgerufen bei Erstellung oder Laden des Workflows.
- **Prozedur `(action-initialize self)`:**
  zur anfänglichen Initialisierung der zugehörigen Aktivität; wird einmal nach Erstellung des Workflows aufgerufen.
- **Prozedur `(action-editable self)`:**
  zur Initialisierung, wenn die Aktivität editierbar wird; wird einmal aufgerufen, wenn der Edit-Status der zugehörigen Aktivität auf „bearbeitbar“ gesetzt wird.
- **Prozedur `(action-done self)`:**
  wird einmal aufgerufen, wenn die zugehörige Aktivität abgeschlossen wird, d.h. wenn der Edit-Status der zugehörigen Aktivität auf „fertig“ gesetzt wird.

- **Prozedur `(action-set-to-done-possible self)`:**
  Möglichkeit für das Workflowmodul, den Abschluss einer Aktivität, d.h. das Setzten des Editierungs-Status der ihm zugeordneten Aktivität zu verweigern, falls gewisse Bedingungen noch nicht erfüllt sind; Es sollte eine Liste zurückgeben werden. Diese Liste enthält als erstes Element eine String-Liste, die die schwerwiegenden, nicht erfüllten Bedingungen beschreibt, als zweites Element eine String-Liste, die die die weniger wichtigen, nicht erfüllten Bedingungen beschreibt und als drittes eine String-Liste mit allgemeinen Meldungen. Nur schwerwiegende, nicht erfüllte Bedingungen verhindern den Abschluss der Aktivität, die weniger wichtigen Bedingungen dienen nur dem Benutzer als Anhaltspunkt. Statt der beiden String-Listen für die nicht erfüllten Bedingungen kann jeweils auch nur ein boolscher Wert stehen (true bedeutet: alle Bedingungen erfüllt). Alternativ kann auch nur der boolsche Wert statt der Liste zurückgegeben werden, der das Nicht-Erfüllt-Sein einer schwerwiegenden Bedingung anzeigt.

- **Prozedur `(action-parent-editable self distance)`:**
  ähnlich wie `action-editable`, wird aber bereits dann aufgerufen, wenn ein die Aktivität enthaltender Parent-Workflow editable wird, `distance` gibt dabei die Parent-Tiefe an (0 bedeutet den direkten Parent-Workflow, also das die Aktivität direkt enthaltende Workflowinstanz-Modell).

- **Prozedur `(dynamic-info self)`:**
  bestimmt das Label bzw. den Textinhalt der zugehörigen Aktivität. Dieser Text sollte einen Überblick über den aktuellen Stand der Aktivität bzw. ihrer Artefakte gegen. Es sollte eine Liste von Strings zurückgeliefert werden.. Jeder String wird dann in einer eigenen Zeile dargestellt. Allerdings wird dabei sowohl die Anzahl als auch die Länge der Strings von der aufrufenden Label-Protodome-Operation der Aktivität beschränkt. Die maximalen Werte für schließlich angezeigte Anzahl Strings und deren maximale Länge werden bereits in der Workflowspezifikation festgelegt, um die Anzeige des Workflows bei der Ausführung nicht durcheinanderzubringen. Die Prozedur `dynamic-info` muss sich darum aber nicht kümmern, sondern kann jede beliebige Anzahl String in beliebiger Länge zurückliefern.

- **Prozedur `(get-artifact self key-list)`:**
  bietet eine Möglichkeit zur Kommunikation mit Submodellen der zugehörigen Aktivität, vor allem um den Submodellen auf generische Art Daten zu übergeben. Hierbei ist `key-list` eine hierarchische Schlüsselliste. Hiermit kann eine Aktivität z.B. andere Workflowartefakte oder sonstige Informationen an Submodelle weitergeben

A.5.2 Operationen zur globalen Artefaktsteuerung im Workflow

Der Workflowinstanz-Modelltyp unterstützt außerdem eine globale Artefaktverwaltung. Hierfür können Aktivitäten (oder Submodelle davon) beim Workflowinstanz-Modell Artefakte (z.B. ihre Submodelle oder andere beliebige Alter-Objekte) unter bestimmten Schlüsselnamen abspeichern (bzw. referenzieren) oder diese abfragen, sowie Interesse bei Änderung dieser registrierten Artefakte anmelden. Realisiert wird dies durch die Properties `workflow-artifact` und `workflow-artifact-interest` im Workflowinstanz-Graphmodell sowie eine Reihe von Operationen für das Ändern, Abfragen und die Anmeldung des Interesses. In der Property `workflow-artifact` werden die Artefakte selbst (bzw. Referenzen auf sie) abgespeichert. `workflow-artifact-interest` dagegen enthält die Verwaltung für die Anmeldung des Interesses.

Die Schlüsselnamen der Workflowartefakte sind hierarchisch aufgebaut, um eine gute Strukturierung zu erlauben.

Die Operationen (auf dem Graphmodell) für den Zugriff sind die folgenden (definiert sind alle in der Datei `mnm-wf-artifact.lib`):

- **Operation `(get-wf-artifact self key-list)`:**
  für die Abfrage eines Workflowartefaktes; Gibt als Rückgabe das Artefakt zurück, falls es vorhanden

ist, `self` ist das Workflowinstanz-Modell, `key-list` ist eine Liste von Werten (in der Regel Alter-Symbole), die den hierarchischen Schlüssel bilden.

- **Operation `(set-wf-artifact self key-list value)`:**
  für das Setzen eines Workflowartefaktes; `self` ist das Workflowinstanz-Modell, `key-list` ist eine Liste von Werten (in der Regel Alter-Symbole), die den hierarchischen Schlüssel bilden; `value` ist der neue Wert für das Artefakt.
- **Operation `(unset-wf-artifact self key-list value)`:**
  für das Löschen eines Workflowartefaktes, d.h. überschreiben des aktuellen Wertes durch den Wert `nil`; `self` ist das Workflowinstanz-Modell, `key-list` ist eine Liste von Werten (in der Regel Alter-Symbole), die den hierarchischen Schlüssel bilden.
- **Operation `(add-wf-artifact-interest self key-list proc obj role)`:**
  registriert Interesse an der Änderung eines Artefaktwertes ähnlich wie `add-interest` für Grapething-Interests; `self` ist das Workflowinstanz-Modell, `key-list` ist eine Liste von Werten (in der Regel Alter-Symbole), die den hierarchischen Schlüssel bilden, `proc` ist der Name (als Alter-Symbol) einer Operation, die für `obj` aufgerufen werden soll, wenn sich der Wert des Artefaktes ändert (der Interessen-Handler). Die aufgerufene Operation erhält als Parameter (zusätzlich zu `obj` `key-list`, das Alter-Symbol `set` oder `unset` je nach Art der Änderung und den neuen Wert des Artefaktes. `role` bietet eine zur Möglichkeit zur Unterscheidung von verschiedenen Interessen-Handlern pro `obj`.
- **Operation `(remove-wf-artifact-interest self key-list obj role)`:**
  meldet Interesse an der Änderung eines Artefaktwertes ab; Parameter siehe Operation `add-wf-artifact-interest`
- **Operation `(add-wf-artifact-interest-and-test-now self key-list proc obj role)`:**
  kleine Wrapper-Operation: registriert zunächst mit `add-wf-artifact-interest` den Interest-Handler auf und ruft diesen dann, falls das betroffene Artefakt derzeit einen Wert hat, auch gleich einmal auf.

A.6 Beschreibung von generischen Operationen für das Werkzeug

Der Scheme-Dialekt Alter stellt teilweise selbst nur sehr grundlegende Operationen für Basistypen wie Zahlen, Strings, Listen aber auch Grapethings, zur Verfügung. Diese werden in diesem Kapitel als **Standardoperationen** bezeichnet. Daher wurden für verschiedene Bereiche eine Reihe komplexerer, generischer Operationen entwickelt, auf die an anderen Stellen — vor allem bei der Realisierung der Modelltypen — dann zurückgegriffen werden kann, ohne jedes mal das entsprechende Konzept neu entwickeln zu müssen.

Die wichtigsten dieser entwickelten Operationen sollen nun vorgestellt werden. Denn mit ihrer Hilfe ist eine Anpassung oder Erweiterung der bisher entwickelten Modelltypen oder gar die Entwicklung neuer Modelltypen wesentlich schneller und einfacher möglich, als nur mit den von Alter vorgegebenen Standardoperationen.

A.6.1 Operationen für allgemeine Konvertierungen und Typüberprüfung

Zunächst sollen einige Operationen betrachtet werden, die einfache Konvertierung zwischen verschiedenen Typen ermöglicht (alle definiert in `mnm-utilities.lib`). Diese Operationen werden an verschiedensten Stellen verwendet:

- **Operation `(myobject->string value)`:**
  wandelt das Objekt `value` in einen String um, falls es nicht ein String ist. Dabei werden folgende Möglichkeiten berücksichtigt:

- Der Wert `nil`, das Alter-Symbol `tbd` (dies steht in Grape für „nicht definiert“) werden in einen leeren String umgewandelt
- sonstige Alter-Symbole werden mit der Standardoperation `symbol->string` in Strings umgewandelt
- boolsche Werte werden in den String „true“ bzw. „false“ umgewandelt
- Grapething-Referenzen werden mit der Standardoperation `name` auf ihren Namen abgebildet
- Ansonsten wird `object` mit der Standardoperation `object->string`, die die Alter-String-Repräsentation berechnet, umgewandelt

Rückgabe ist stets das umgewandelte Objekt. Dieses ist stets ein String.

- **Operation `(myobject->int value)`:**
  wandelt das Objekt `value` in eine Integer-Zahl um, falls es nicht schon eine ist. Dabei werden folgende Möglichkeiten berücksichtigt:
  - Der Wert `nil`, das Alter-Symbol `tbd` (dies steht in Protodome für „nicht definiert“) werden in den Integer-Wert Null umgewandelt
  - Zahlen, die keine Integerzahlen sind (`float` oder `fraction`) werden mit Standardoperation `round` gerundet
  - Strings werden mit den Standardoperationen `string->number` und `round` in Zahlen umgewandelt
  - Alter-Symbole werden mit den Standardoperationen `symbol->string`, `string->number` und `round` in Zahlen umgewandelt
  - jedes andere Objekt wird auf den Integer-Wert Null abgebildet

  Rückgabe ist stets das umgewandelte Objekt. Dieses ist stets eine Integer-Zahl.
- **Operation `(myobject->bool value)`:**
  wandelt das Objekt `value` in einen boolschen Wert um, falls es nicht schon einer ist. Dies bedeutet einfach, dass Objekte, die kein boolschen Wert haben auf false abgebildet werden.
- **Operation `(myobject->list value)`:** wandelt das Objekt `value` in eine Liste um, falls es nicht schon eine ist. Dabei werden folgende Möglichkeiten berücksichtigt:
  - Der Wert `nil`, das Alter-Symbol `tbd` (dies steht in Protodome für „nicht definiert“) werden in die leere Liste umgewandelt
  - alle sonstigen Objekte werden in einer Liste verpackt, d.h. auf eine Liste abgebildet, die nur das Objekt enthält

Die folgende Operation wurde entwickelt, um auf verschiedene Arten zu prüfen, ob ein Objekt einem bestimmten Typ angehört:

- **Operation `(myis-a? object type-description)`:**
  ähnlich wie die Standardoperation `is-a?`, nur dass für die Angabe eines Typs allgemeinere Formen möglich sind, als nur einen Typ selbst direkt in Alter anzugeben. Folgende Formen sind für `type-description` möglich:
  - die direkte Angabe eines Typobjekts (d.h. ein Objekt vom Metatyp), dass den jeweiligen Typ beschreibt. In diesem Fall wird die Überprüfung direkt mit der Standardoperation `is-a?` durchgeführt.
  - Angabe des Typs durch ein Alter-Symbol: es wird für den Typ und alle Obertypen von `object` geprüft, ob für einen davon der Name gleich dem Alter-Symbol ist. Diese Angabe kann vor allem dann verwendet werden, wenn nicht klar ist, ob der anzugebende Typ zur Laufzeit dem

Alter-System bekannt (d.h. im wesentlichen ob sein Modelltyp derzeit geladen ist) ist, also dann wenn modelltyp-übergreifendende Typprüfungen notwendig sind.

– Angabe des Typs als Liste von Typen, d.h. die Angabe von mehreren Typen: In diesem Fall wird `myis-a?` für jedes Element der Liste erneut aufgerufen, und überprüft, ob `object` mindestens einem der Typen angehört.

Wenn bekannt ist, dass die Typbeschreibung direkt ein Typobjekt in Alter ist, so ist die Standardoperation `is-a?` `myis-a?` vorzuziehen.

A.6.2 Operationen für Strings

Für die Formatierung von Strings wurden eine Reihe von Operationen (definiert in `mnm-string.lib`) für eine einfache Verwendung — vor allem beim Formatieren des Labels eines Graphobjekts — geschaffen:

- **Operation `(string-with-default s default)`:**
  gibt `s` zurück, falls dieses nicht gleich `nil` oder der leere String ist, sonst `default`

- **Operation `(string-newline s)`:**
  hängt an den String `s` ein Newline-Zeichen an, falls `s` nicht der leere String ist; Rückgabe ist in jedem Fall der geänderte String.

- **Operation `(string-wrap s prefix postfix)`:**
  umgibt den String `s` mit dem Präfix-String `prefix` und dem Postfix-String `postfix`, sofern `s` nicht der leere String ist; Rückgabe ist in jedem Fall der geänderte String.

- **Operation `(string-concat l separator default)`:**
  erstellt aus der Liste von Strings `l` einen zusammenhängenden String, wobei der String `separator` als Zwischen-String zwischen zwei Elementen von `l` eingeschoben wird; Die Elemente von `l` müssen nicht unbedingt Strings sein, sondern müssen von der Operation `myobject->string` (siehe Anhang A.6.1) in Strings konvertierbar sein. Leere Strings werden hierbei ignoriert. Die Rückgabe ist der zusammenhängende String oder der String `default`, falls `l` leer ist bzw. alle Elemente zu einem leeren String konvertiert werden.

- **Operation `(string-concat-newline l)`:**
  ähnlich zu `string-concat`, allerdings ist der Separator-String hier der String der nur aus einem Newline-Zeichen besteht und der Default-String der leere String

- **Operation `(string-concat-newline-list <Liste der Argumente>)`:**
  Wrapper für `string-concat-newline`; Die Strings liegen hier nicht in einer Liste verpackt vor, sondern jeder String ist ein eigenes Argument an die Operation.

- **Operation `(string-concat-space l)`:**
  ähnlich zu `string-concat`, allerdings ist der Separator-String hier der String der nur aus einem Leer-Zeichen besteht und der Default-String der leere String.

- **Operation `(string-concat-space-list <Liste der Argumente>)`:**
  Wrapper für `string-concat-space`; Die Strings liegen hier nicht in einer Liste verpackt vor, sondern jeder String ist ein eigenes Argument an die Operation.

Einige weitere Definitionen für Strings sind die folgenden:

- **Konstante `newline-string`:**
  enthält einen String, der nur aus einem Newline-Zeichen besteht

- **Operation `(string-split-newline s)`:**
  zerlegt den String `s` in Teilstrings, wobei die Teilstrings in `s` durch Newline-Zeichen getrennt waren. Die Newline-Zeichen kommen in den zurückgegebenen Teilstrings selbst nicht vor. Kommen in `s`

mehrere Newline-Zeichen hintereinander oder ein Newline-Zeichen ganz am Anfang vor, so wird dieses als ein leerer Teilstring berücksichtigt. Die Rückgabe ist die Liste der Teilstrings.

- **Operation `(line-count s)`:**
  gibt die Anzahl Zeilen, d.h. Anzahl der vorkommenden Newline-Zeichen erhöht um Eins, des Strings `s` zurück

- **Operation `(is-number-string? s)`:**
  überprüft, ob der String `s` eine natürliche Zahl repräsentiert, d.h. ob er nur aus den Ziffern und Leerzeichen oder Tabulator-Zeichen besteht; Rückgabe ist ein entsprechender boolscher Wert.

A.6.3 Mathematische Operationen

Im Folgenden sollen einige der entwickelten, generischen Operationen für mathematische bzw. geometrische Fragestellungen erläutert werden.

Für die Realisierung von Mengen wurden einige Operationen (in `mnm-math.lib`) definiert. Diese betrachten jeweils eine Liste von Elementen als eine Menge, d.h. im wesentlichen, dass sie Duplikate entfernen bzw. nicht berücksichtigen. Solche speziell interpretierten Listen werden daher als **Listsets** bezeichnet. Die Gleichheit von Elementen wird hierbei jeweils mit den Alter-Operationen `member` bzw. `equal`, die auch rekursive Gleichheitsprüfung für komplexere Strukturen wie Strings, Listen, etc. durchführen, geprüft. Die Reihenfolge von Elementen innerhalb eines Listsets ist beliebig, alle definierten Operationen verändern jedoch eine vorgegebene Reihenfolge im wesentlichen nicht. Daher können die Operationen für Listsets nicht nur für die Realisierung von Mengen (d.h. Sammlung ohne Duplikate mit beliebige Reihenfolge), sondern auch für die Realisierung von Sammlungen ohne Duplikate aber mit bestimmter Reihenfolge verwendet werden:

- **Operation `(listset-unify s)`:**
  macht ein gegebenes Listset `s` kanonisch, d.h entfernt alle Duplikate; Dabei bleibt jeweils das erste Vorkommen eines Elementes in der Liste erhalten, alle weiteren werden entfernt. Rückgabe ist das kanonisierte Listset.

- **Operation `(listset-contains s el)`:**
  prüft, ob das Objekt `el` im Listset `s` vorhanden ist; Rückgabe ist ein entsprechender, boolscher Wert.

- **Operation `(listset-subset? s1 s2)`:**
  prüft, ob das Listset `s1` eine Teilmenge des Listsets `s2` ist, d.h. ob alle Elemente von `s1` auch in `s2` vorhanden sind; Rückgabe ist ein entsprechender, boolscher Wert.

- **Operation `(listset-equal? s1 s2)`:**
  prüft, ob die Listsets `s1` und `s2` (als Mengen) gleich sind, d.h. ob alle Elemente von `s1` auch in `s2` vorhanden sind und umgekehrt; Rückgabe ist ein entsprechender, boolscher Wert.

- **Operation `(listset-union s1 s2)`:**
  berechnet die Vereinigung der Listsets `s1` und `s2`, d.h. das Listset das alle Elemente von `s1` und `s2` enthält; Die Rückgabe ist kanonisiert, d.h. hat keine Duplikate. Die Elemente von `s1` kommen dabei als erstes in ihrer ursprünglichen Reihenfolge vor (soweit keine Duplikate entfernt wurden), dann folgen die zusätzlichen Elemente von `s2`.

- **Operation `(listset-difference s1 s2)`:**
  berechnet den Mengen-Unterschied der Listsets `s1` und `s2`, d.h. das Listset, das aus den Elementen von `s1`, die nicht in `s2` enthalten sind, besteht; Die Rückgabe ist kanonisiert. Die Reihenfolge der Elemente ist gleiche wie bei `s1`.

- **Operation `(listset-intersect s1 s2)`:**
  berechnet den Mengen-Durchschnitt der Listsets `s1` und `s2`, d.h. das Listset, das die gemeinsamen Elemente von `s1` und `s2` enthält; Die Rückgabe ist kanonisiert. Die Reihenfolge der Elemente ist gleiche wie bei `s1`.

- **Operation `(listset-product s1 s2)`:**
  berechnet die Produktmenge der Listsets `s1` und `s2`; Die Rückgabe ist kanonisiert und die Elemente sind zunächst gemäß der Reihenfolge in `s1`, dann der in `s2` (also lexikografisch) geordnet.

- **Operation `(listset-potency s)`:**
  berechnet die Potenzmenge des Listsets `s1`; Die Rückgabe ist kanonisiert.

- **Operation `(listset-insert-last s el)`:**
  fügt das Objekt `el` dem Listset `s` (am Ende) hinzu, sofern es noch nicht darin enthalten ist; Die Rückgabe wird nicht kanonisiert. `s` wird an das Ende von `s` angehängt, die Reihenfolge der übrigen Elemente bleibt wie die in `s`.

- **Operation `(listset-insert-first s el)`:**
  fügt das Objekt `el` dem Listset `s` (am Beginn) hinzu, sofern es noch nicht darin enthalten ist; Die Rückgabe wird nicht kanonisiert. `s` wird vor den Beginn von `s` gesetzt, die Reihenfolge der übrigen Elemente bleibt wie die in `s`.

Auch für die Handhabung von UML-Multiplizitäts-Intervallen, d.h. vor allem ihre Kanonisierung, wurden mehrere Operationen erstellt. In UML werden Multiplizitäten durch eine Liste von Multiplizitäten-Ranges beschrieben. Jede solche Multiplizitäten-Range ist dabei ein Paar, wobei die erste Komponente stets eine natürliche Zahl größer gleich Null ist. Die zweite Komponente ist entweder auch eine natürliche Zahl größer gleich Eins oder aber ein Wert der Unendlich repräsentiert. Für Unendlich wurde als interne Darstellung die Zahl -1 gewählt. Eine Multiplizitäten-Range entspricht nun einem bestimmten Intervall innerhalb der natürlichen Zahlen. Falls sich die Intervalle von Multiplizitäten-Ranges nun überschneiden oder aneinander angrenzen, können sie zusammengefasst werden. Daher gibt es die folgende Operation (definiert in `mnm-math.lib`):

- **Operation `(interval-list-unify l)`:**
  interpretiert die Liste `l` von Paaren als eine Liste von Intervallen und vereinigt soweit wie mögliche alle Intervalle und gibt die resultierenden Liste von entsprechenden Paaren zurück; Die erste Komponente eines Paares muss eine natürliche Zahl größer gleich Null sein, die zweite Komponente entweder auch eine natürliche Zahl größer gleich Null oder der Wert -1, der Unendlich repräsentiert.

A.6.4 Dictionary- und Listen-Operationen

Alter kennt einen Datentyp für assoziative Arrays, den Dictionary-Datentyp. Standardoperationen für Dictionaries sind u.a. `dictionary-ref`, `dictionary-set!` und `dictionary-unset!`.

Dictionaries kann man hierarchisch benutzen, d.h. als Werte in einem Dictionary selbst wieder Dictionaries verwenden. Statt eines einzigen Schlüssel-Wertes verwendet man dann eine Liste von Schlüsseln-Werten, jeweils ein Schlüssel-Wert pro Dictionary-Ebene. Hierarchische Dictionaries sind zur besseren Strukturierung sinnvoll. Um Alter-Dictionaries — vor allem als Property-Werte — hierarchisch zu verwenden, wurden folgende Operationen entwickelt (im Alter-Codefile `mnm-utilities.lib` definiert):

- **Operation `(dictionary-ref-recursively dict key-list default)`:**
  gibt den in dem hierarchischen Dictionary `dict` unter der Schlüsselliste `key-list` gespeicherten Wert zurück, falls dieser existiert, ansonsten wird `default` zurückgegeben;

- **Operation `(dictionary-set-recursively! dict key-list value)`:**
  speichert in dem hierarchischen Dictionary `dict` unter der Schlüsselliste `key-list` den Wert `value`.

- **Operation `(dictionary-unset-recursively! dict key-list)`:**
  löscht in dem hierarchischen Dictionary `dict` den Wert, der unter der Schlüsselliste `key-list` gespeichert ist, d.h. entfernt entsprechenden Wert aus dem Dictionary der letzten Dictionary-Ebene;

Dictionaries stellen eine effiziente Möglichkeit für die Realisierung von assoziativen Arrays in Alter dar. Jedoch kann man ein assoziatives Array auch einfach durch eine Liste von Paaren, einer sogenannten

Aliste, implementieren. Verwendet man nämlich einen Dictionary als Wert für eine Property, so werden beim Laden des Modells aus einer gespeicherten Datei die einzelnen im Dictionary gespeicherten Werte nicht wieder richtig hergestellt, wenn es Referenzen zu Grapethings sind. Beliebig verschachtelte Listen als Property-Werte sind aber kein Problem beim Laden des Modells. Alle in der Liste enthaltenen Grapething-Referenzen werden vollständig aufgelöst, d.h. wiederhergestellt.

Daher wurden für Alisten und vor allem die hierarchische Verwendung — analog zu der hierarchischen Verwendung von Dictionaries — eigene Operationen geschaffen (ebenfalls im Alter-Codefile `mnm-utilities.lib` definiert):

- **Operation `(alist-ref alist key default)`:**
  analog zur Standardoperation `dictionary-ref`: ermittelt den Wert, der in der Alist `alist` unter dem Schlüssel-Wert `key` gespeichert; Existiert dieser Wert nicht, so wird statt dessen `default` zurückgegeben.

- **Operation `(alist-key alist)`:**
  analog zur Standardoperation `dictionary-keys`: ermittelt die in Schlüssel-Werte der Alist `alist`;

- **Operation `(alist-set! alist key value)`:**
  analog zur Standardoperation `dictionary-set!`: speichert in der Alist `alist` den Wert `value` unter dem Schlüssel-Wert `key` ab. Ein Unterschied zu `dictionary-set!` liegt auch darin, dass das veränderte `alist` zurückgegeben wird, d.h. es findet keine Änderung der Alist durch Seiteneffekte statt.

- **Operation `(alist-unset! alist key)`:**
  analog zur Standardoperation `dictionary-unset!`: entfernt in der Alist `alist` den Wert `value` unter dem Schlüssel-Wert `key`. Ein Unterschied zu `dictionary-unset!` liegt auch darin, dass das veränderte `alist` zurückgegeben wird, d.h. es findet keine Änderung der Alist durch Seiteneffekte statt.

- **Operation `(alist-ref-recursively alist key-list default)`:**
  analog zur Operation `dictionary-ref-recursively`: ermittelt den Wert in der Alist `alist`, der unter der Schlüsselliste `key-list` gespeichert ist. Falls der Wert nicht existiert, wird statt dessen `default` zurückgegeben.

- **Operation `(alist-set-recursively! alist key-list value)`:**
  analog zur Operation `dictionary-set-recursively!`: speichert in der hierarchischen Alist `alist` den Wert `value` unter der Schlüsselliste `key-list` ab. Das veränderte `alist` wird zurückgegeben, d.h. es findet keine Änderung der Alist durch Seiteneffekte statt.

- **Operation `(alist-unset-recursively! alist key-list)`:**
  analog zur Operation `dictionary-unset-recursively!`: entfernt in der hierarchischen Alist `alist` den Wert unter der Schlüsselliste `key-list`. Das veränderte `alist` wird zurückgegeben, d.h. es findet keine Änderung der Alist durch Seiteneffekte statt.

A.6.5 Operationen für Grapethings

Für den Zugriff auf Grapethings und deren Beziehungen untereinander bietet Alter bereits eine Reihe von Standardoperationen. Die wichtigsten dieser Standardoperationen sind (siehe auch [AlterManual]): `graph`, `components`, `elements`, `accessories`, `nodes`, `arcs`, `container`, `incoming-arcs`, `outgoing-arcs`, `subdiagrams`, `get-property`, `set-property!`, `position`, `set-position!`, `bounds`, `name`, `color`, `set-color!`.

Auch hierfür wurden einige Operationen entwickelt, die auf Basis dieser Standardoperationen den Zugriff noch erleichtern.

Alter bietet mit den oben genannten Standardoperationen bereits einen einfachen Zugriff auf diese strukturellen Beziehungen von Grapething (z.B. Kanten-verbindungen, Enthalten-Sein). Z.B. kann man mit `incoming-arcs` alle eingehenden Kanten zu einem gegebenen Knoten erhalten. In der Regel ist man aber bei einer solchen Abfrage nur an Kanten eines oder mehrere bestimmter Typen interessiert. Daher wurden für die verschieden, strukturellen Beziehungen zwischen Grapethings jeweils zwei Operationen entwickelt, die zusätzlich zur strukturellen Beziehung auch gleich den Typ bei der Abfrage berücksichtigen. Mit der einen Operation erhält man immer alle Grapethings mit dem jeweiligen Typ. Ihr Name hat immer die Form „get-<Struktur-Beziehung>-with-type“, wobei <Struktur-Beziehung> die jeweilige strukturelle Beziehung beschreibt (z.B. „incoming-arcs“). Mit der anderen Operation wird nur ein Grapething (das erste gefundene; die Auswahl erfolgt mit der Standardoperation `detect`) ermittelt, das dem gesuchten Typ entspricht. Diese kann vor allem dann verwendet werden, wenn bekannt ist, dass nur ein Grapething des gewünschten Typs existiert. Ihr Name hat immer die Form „get-first-<Struktur-Beziehung, aber in der Einzahl>-with-type“. <Struktur-Beziehung> wird hier in der Einzahl verwendet, d.h. z.b. „incoming-arc“ statt ïncoming-arcs“. Die Operationen erhalten jeweils als Argument eine Typbeschreibung, die die gleichen Formen wie bei der Operation `myis-a?` (siehe Anhang A.6.1) haben kann. Im einfachsten Fall ist diese Typbeschreibung einfach direkt ein Alter-Typobjekt (d.h. der Alter-Klassenname der gewünschten Grapethings). Die Operationen sind jeweils im Einzelnen:

- **Operation `(get-components-with-type gthing type-descr)`:**
  Abfrage der Komponenten eines Grapethings, die der Typbeschreibung `type-descr` entsprechen. Das heißt Wrapper für die Standardoperation `components`, die für Graphmodelle die enthaltenen Knoten und Kanten, für Knoten die enthaltene Knoten, Listelemente und Accessories und für Kanten die Accessories bestimmt.

- **Operation `(get-first-component-with-type gthing type-descr)`:**
  Abfrage der ersten Komponente eines Grapethings, die der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `components`.

- **Operation `(get-nodes-with-type graph type-descr)`:**
  Abfrage der Knoten eines Graphmodells, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `nodes`.

- **Operation `(get-first-node-with-type graph type-descr:`**
  Abfrage des ersten Knotens eines Graphmodells, der der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `nodes`.

- **Operation `(get-submodel-nodes-with-type gobject type-descr)`:**
  Abfrage der Knoten des ersten Submodells eines Graphobjekts, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `nodes`. Das erste Submodell wird mit der Operation `get-first-child` (siehe unten) ermittelt.

- **Operation `(get-first-submodel-node-with-type gobject type-descr)`:**
  Abfrage des ersten Knotens des ersten Submodells eines Graphobjekts, der der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `nodes`. Das erste Submodell wird mit der Operation `get-first-child` (siehe unten) ermittelt.

- **Operation `(get-arcs-with-type graph type-descr)`:**
  Abfrage der Kanten eines Graphmodells, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `arcs`.

- **Operation `(get-first-arc-with-type graph type-descr)`:**
  Abfrage der ersten Kante eines Graphmodells, die der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `arcs`.

- **Operation `(get-elements-with-type node type-descr)`:** Abfrage der enthaltenen Knoten oder Listelemente eines Knotens, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `elements`.

- **Operation `(get-first-element-with-type node type-descr)`:**
  Abfrage des ersten, enthaltenen Knoten oder Listelementes eines Knotens, der der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `elements`.
- **Operation `(get-accessories-with-type arc-or-node type-descr)`:**
  Abfrage der Accessory-Knoten einer Kante oder eines Knotens, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `accessories`.
- **Operation `(get-first-accessory-with-type arc-or-node type-descr)`:**
  Abfrage des ersten Accessory-Knoten einer Kante oder eines Knotens, der der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `accessories`.
- **Operation `(get-incoming-arcs-with-type gobject type-descr)`:**
  Abfrage der eingehenden Kanten eines Knoten oder Listelements, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `incoming-arcs`.
- **Operation `(get-first-incoming-arc-with-type gobject type-descr)`:**
  Abfrage der ersten, eingehenden Kante eines Knoten oder Listelements, die der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `incoming-arcs`.
- **Operation `(get-outgoing-arcs-with-type gobject type-descr)`:**
  Abfrage der ausgehenden Kanten eines Knoten oder Listelements, die der Typbeschreibung `type-descr` entsprechen. Verwendet die Standardoperation `outgoing-arcs`.
- **Operation `(get-first-outgoing-arc-with-type gobject type-descr`:**
  Abfrage der ersten, ausgehenden Kante eines Knoten oder Listelements, die der Typbeschreibung `type-descr` entspricht. Verwendet die Standardoperation `outgoing-arcs`.

Ist ein Graphobjekt in einem Graphmodell enthalten, so wird das enthaltende Graphmodell als **das zum Graphobjekt gehörige Graphmodell** bezeichnet. Es kann mit der Standardoperation `graph` ermittelt werden. Diese liefert für Graphmodelle selbst (als Argument, statt einem Graphobjekt) diese zurück, d.h. für ein Graphmodell ist das Graphmodell selbst das zum ihm gehörige Graphmodell. Das heißt für jedes Grapething kann das zu ihm gehörige Graphmodell bestimmt werden.

Die bisher besprochenen Operationen betreffen die strukturellen Beziehungen innerhalb eines Graphmodells. Aber auch zwischen Graphmodellen existieren strukturelle Beziehungen, da jedes Graphobjekt mehrere Graphmodelle als Submodell haben kann. Die Submodelle eines Graphobjekts können mit der Standardoperation `subdiagrams` bestimmt werden. Gibt es zu einem Graphmodell ein Graphobjekt, so dass das Graphmodell ein Submodell dieses Graphobjekts ist, so wird das Graphobjekt das **Parent** des Graphmodells genannt. Das Parent kann mit der Standardoperation `parent` bestimmt werden. Zu jedem Graphmodell gibt es höchstens ein Parent.

Die Submodelle eines Graphobjekts sind vom Benutzer über Navigations-Buttons vom Graphobjekt aus erreichbar. Statt der normalen Submodelle eine Graphobjekts, also solchen die als Parent das Graphobjekt haben, können über die Navigations-Buttons auch Graphmodelle erreicht werden, die mit dem Graphobjekt über sogenannte **Modelbindungen** (model bindings) verbunden sind. Jede Modellbindung besteht zwischen einem Graphobjekt und einem Graphmodell und ist außerdem durch einen Namen gekennzeichnet. Nur falls dieser Name gleich der sogenannten Konfiguration des enthaltenden Graphmodells des Graphobjekts ist, ist statt der Navigation zu eigentlichen Submodellen die Navigation zu den durch Modellbindungen verbundenen Graphmodellen möglich.

Die durch Modellbindungen mit einem Graphobjekt verbundenen Graphmodelle sollen hier auch als Submodelle dieses Graphobjekts bezeichnet werden, wobei die tatsächlichen Submodelle (mit parent-Beziehung zum Graphobjekt) genauer als **echte Submodelle** davon unterschieden werden.

Für diese modell-übergreifenden, strukturellen Beziehungen wurden die folgenden Operationen entwickelt:

- **Operation `(parent-graph gthing)`:**
  Ist `gthing` ein Graphmodell, so wird zunächst mit der Standardoperation `parent` das Parent zu dem Graphmodell `gthing`, also das Graphobjekt, von dem `graph` ein Submodell ist, bestimmt und

falls dieses Parent existiert davon mit der Standardoperation `graph` das enthaltende Graphmodell bestimmt. Existiert das Parent nicht, so wird `nil` zurückgegeben.

Ist `gthing` jedoch ein Graphobjekt, so wird zunächst das es enthaltende Graphmodell bestimmt und für dieses `parent-graph` erneut aufgerufen.

Das heißt die `parent-graph` ermittelt stets das Graphmodell, das dem zum Grapething `gthing` gehörigen Graphmodell übergeordnet ist (sofern ein solches existiert).

- **Operation `(graph-parent graphobject)`:**
  einfacher Wrapper: ermittelt für das Graphobjekt `graphobject` zunächst mit der Standardoperation `graph` das zugehörige Graphmodell, dann mit `parent` das Parent davon;
- **Operation `(get-first-real-child graphobject)`:**
  bestimmt das erste echte Submodell des Graphobjekts `graphobject`, d.h. ein das erste Graphmodell, das `graphobject` als Parent hat.
- **Operation `(get-default-modelbinding graphobject)`:**
  bestimmt das Graphmodell, das derzeit eine Modellbindung (mit der aktuellen Konfiguration des Graphmodells) zum Graphobjekt `graphobject` hat. Dieses Graphmodell kann vom Benutzer über die normalen Submodell-Navigations-Buttons wie ein echtes Submodell erreicht werden.
- **Operation `(get-first-child graphobject)`:**
  bestimmt das erste Submodell des Graphobjekts `graphobject`. Dabei wird zunächst versucht dieses mit der Operation `get-first-real-child` als echtes Submodell von `graphobject` zu ermitteln. Existiert kein solches echtes Submodell, wird weiter versucht das Submodell mit `get-default-modelbinding` zu ermitteln.

A.6.6 Operationen für Zugriff aus Grapething-Properties

Für den Zugriff auf Properties, sowohl generisch für beliebige Properties als auch für die speziell verwendeten Properties (siehe Anhang A.4), wurden mehrere Operationen festgelegt (jeweils definiert im Alter-Codefile `mnm-properties.lib`):

- **Operation `(get-all-property-names grapething-or-class)`:**
  ermittelt alle definierten Properties eines Grapethings oder einer Grapething-Klasse mit Hilfe der Standardoperation `get-property-definitions` und gibt eine Liste ihrer Namen zurück.
- **Operation `(get-property-names-with-mnmspecial grapething-or-class)`:**
  wie `get-all-property-names`, lässt jedoch in der zurückgegebenen Liste Properties weg, die von *DoME* speziell verwendet werden (siehe Anhang A.1.5).
- **Operation `(get-property-names grapething-or-grapething-class)`:**
  wie `get-all-property-names`, lässt jedoch in der zurückgegebenen Liste sowohl Properties weg, die von *DoME* speziell verwendet werden (siehe Anhang A.1.5), als auch für das Werkzeug speziell verwendete Properties („mylock“, „myview“, „mymaster“, „intern-status“, „graph-type“ und „graph-kind“; siehe auch Anhang A.4);
- **Operation `(has-property? grapething-or-grapething-class propname)`:**
  prüft mit der Standardoperation `get-property-definition`, ob eine Property mit dem Namen `propname` vorhanden ist; Es wir ein entsprechender boolscher Wert zurückgegeben.
- **Operation `(get-property-string propname obj)`:**
  benutzt die Standardoperation `get-property` für den Zugriff auf den Wert einer Property und konvertiert diese mit `myobject->string` in einen String;
- **Operation `(get-property-int propname obj)`:**
  benutzt die Standardoperation `get-property` für den Zugriff auf den Wert einer Property und konvertiert diese mit `myobject->int` in einen Integer-Wert;

- **Operation `(get-property-bool propname obj)`:**
benutzt die Standardoperation `get-property` für den Zugriff auf den Wert einer Property und konvertiert diese mit `myobject->bool` in einen boolschen Wert;

- **Operation `(get-property-list propname obj)`:**
benutzt die Standardoperation `get-property` für den Zugriff auf den Wert einer Property und konvertiert diese mit `myobject->list` in eine Liste;

- **Operation `(redisplay-label graphobject)`:**
stößt eine Auffrischung der Darstellung des Graphobjekts `graphobject`. Besitzt `graphobject` die Property `display-status`, so wird der Wert dieser verändert. Um ein Auffrischen der Darstellung tatsächlich zu ermöglichen, muss im Metamodell für diese Property Anzeigesignifikanz definiert sein.

- **Operation `(set-lock! grapething name value)`:**
falls das Graphobjekt `graphobject` die Property `mylock` besitzt (diese sollte einen Alter-Dictionary als Wert haben) wird der Wert `value` darin unter dem Dictionary-Schlüssel `name` gespeichert.

- **Operation `(lock-on! grapething name)`:**
Wrapper für `set-lock!`, der als `value` den Wert true verwendet. Typischerweise als Lock-Mechanismus zum Sperren bezüglich des Namens `name` benutzt.

- **Operation `(lock-off! grapething name)`:**
Wrapper für `set-lock!`, der als `value` den Wert false verwendet. Typischerweise als Lock-Mechanismus zum Freigabe bezüglich des Namens `name` benutzt.

- **Operation `(get-lock grapething name)`:**
falls das Graphobjekt `graphobject` die Property `mylock` besitzt (diese sollte einen Alter-Dictionary als Wert haben) wird der Wert, der darin unter dem Dictionary-Schlüssel `name` gespeichert ist, zurückgegeben. Typischerweise als Lock-Mechanismus für die Abfrage des Locking-Status bezüglich des Namens `name` verwendet.

- **Opertation `(do-with-lock grapething name proc)`:**
führt die Prozedur `proc` für das Grapething `grapething` geschützt durch ein Lock mit dem Namen `name` geschützt aus, sofern der Locking-Status für `name` derzeit überhaupt eine Freigabe anzeigt, sonst wird nichts getan (siehe `get-lock`, `lock-on!`, `lock-off!`).

  Wird u.a. zur Vermeidung von Endlos-Schleifen verwendet.

- **Operation `(set-mydata! grapething key-list value)`:**
falls das Grapething die Property `mydata` besitzt (diese sollte Listen als Werte erlauben), so wird darin nach Art der `alist`-Operationen (siehe Anhang A.6.4) der Wert `value` unter dem hierarchischen Schlüsselnamen `key-list` gespeichert.

  Die Property `mydata` dient zum Speichern beliebiger Informationen.

- **Operation `(get-mydata! grapething key-list)`:**
falls das Grapething die Property `mydata` besitzt (diese sollte Listen als Werte erlauben), so wird nach Art der `alist`-Operationen (siehe Anhang A.6.4) der Wert, der unter dem hierarchischen Schlüsselnamen `key-list` darin gespeichert ist, ermittelt und zurückgegeben.

  Die Property `mydata` dient zum Speichern beliebiger Informationen.

- **Operation `(set-intern-status! grapething value)`:**
setzt den Wert der Property `intern-status` des Grapethings `grapething` auf `value`, falls die Property `intern-status` in `graphobject` existiert;

  Die Property `intern-status` dient als Kennzeichnung eines temporären Status.

- **Operation `(get-intern-status grapething)`:**
  gibt den den Wert der Property `intern-status` des Grapethings `grapething` zurück, falls die Property `intern-status` in `graphobject` existiert;

- **Operation `(do-with-intern-status grapething status-value proc)`:**
  führt die Prozedur `proc` für das Grapething `grapething` aus. Während der Ausführung wird der Wert der Property `intern-status` auf den Wert `status-value` von `grapething` gesetzt. Nach der Ausführung wird der ursprüngliche Wert wieder hergestellt.

- **Operation `(set-property-locked! propname grapething value)`:**
  setzt mit der Standardoperation `set-property!` den Wert der Property mit Namen `propname` des Grapethings `grapething` auf den Wert `value`. Während der Ausführung wird der Wert der Property `intern-status` von `graphobject` auf das spezielle Alter-Symbol `update-property-locked` gesetzt.

  Dies ist sinnvoll, wenn die Property mit Namen `propname` in ihre Protodome-Methode für die Wächter-Bedingung eine Änderung nur dann zulässt, wenn der interne Status als speziellen Wert das spezielle Alter-Symbol `update-property-locked` hat. Dies kann z.B. durch die spezielle Prozedur `guard-condition-property-locked` getan werden. Hiermit kann genau kontrolliert werden, wann und wie durch das *DoME*-System der Wert einer Property geändert wird.

- **Operation `(set-init-status! grapething value)`:**
  setzt den Wert der Property `init-status` des Grapethings `grapething` auf `value`, falls die Property `init-status` in `graphobject` existiert;

  Die Property `intern-status` dient als Kennzeichnung des Initialisierungsstatuses eines Grapethings.

- **Operation `(get-init-status grapething)`:**
  gibt den den Wert der Property `init-status` des Grapethings `grapething` zurück, falls die Property `init-status` in `graphobject` existiert;

- **Operationen `(set-graph-type! model val)` und `(set-graph-kind! model val)`:**
  setzt den Wert der Property `graph-type` bzw. `graph-kind` des Grapemodells `model` auf `val`, falls die jeweilige Property für `model` existiert;

- **Operationen `(get-graph-type model)` und `(get-graph-kind model)`:**
  gibt den den Wert der Property `graph-type` bzw. `graph-kind` des Graphmodells `model` zurück, falls jeweilige Property für `model` existiert;

- **Operation `(is-main? model)`:**
  prüft, ob die Property `graph-type` für das Graphmodell `model` als Wert das Alter-Symbol `main` hat.

- **Operation `(is-standalone? model)`:**
  prüft, ob die Property `graph-kind` für das Graphmodell `model` als Wert das Alter-Symbol `standalone` hat.

  Dies zeigt typischerweise an, ob `model` Artefakt eines Workflows ist oder nicht.

- **Operation `(copy-properties src dst)`:**
  kopiert die Werte aller Properties des Grapethings `src` in die entsprechenden Properties des Grapethings `dst`, soweit diese dort vorhanden sind. Die zu kopierenden Properties werden mit `get-property-names` (siehe oben) bestimmt.

A.6.7 Operationen für Grapething-Interests

DoME bietet durch Grapething-Interest-Handlern eine wichtige Synchronisations-Möglichkeit zwischen Graphobjekten. Zum An- und Abmelden davon gibt es die Standardoperationen `add-interest` und `remove-interest` (siehe Anhang A.1.4). Um das An- und Abmelden von Grapething-Interest-Handlern — vor allem die Auswahl des Grapething-Interest-Aspektes — weiter erleichtern wurden eine Reihe von weiteren Operationen aufbauend auf den Standardoperationen definiert. Diese sind alle im Alter-Codefile `mnm-interests.lib` definiert.

Sie werden typischerweise in der Operationen `register-dependencies` und `deregister-dependencies` benutzt.

Dabei gibt es im wesentlichen zwei Sorten: Die eine Sorte ist für Grapethings-Interests zwischen beliebigen Grapethings gedacht. Die Operationen der zweiten Sorte sind jeweils nur Wrapper für Operationen der ersten Sorte. Sie sind für Grapething-Interests, die ein Grapething bei sich selbst anmeldet gedacht. Der Name für Operationen der ersten Sorte ist stets nach dem Schema „add-interest-<aspekt>„ bzw. „remove-interest-<aspekt>„ aufgebaut, wobei <aspekt>, den Grapething-Interest-Aspekt beschreibt. Die Argumente für diese Operationen sind die gleichen wie bei `add-interest` und `remove-interest`, wobei der Name des Aspekts als Argument fehlt, da er bereits durch den Operations-Namen festgelegt wird. Das heißt die Argumente sind das Grapething, bei dem der jeweilige Aspekt an- bzw. abgemeldet wird, der Interest-Handler, das interessierte Grapething und der Rollenname für die Anmeldung.

Bei `add-interest` bzw. `remove-interest` muss der Aspekt durch ein String-Argument festgelegt werden. Dies ist leicht fehler-anfällig. Fehler, d.h. Angaben eines unbekannten Aspektes, werden von *DoME* nicht als Fehler bemerkt bzw. angezeigt.

Bei Operationen der zweiten Sorte hat der Name stets das Schema „add-interest-my<aspekt>“ bzw. „remove-interest-my<aspekt>“, also ähnlich wie bei der ersten Sorte, nur der String „my“ vor der Aspekt-Beschreibung eingeschoben. Sie haben die gleichen Parameter wie Operationen der ersten Sorte, nur der Parameter für das interessierte Grapething fehlt, da dieses gleich dem Grapething ist, bei dem der Aspekt an- bzw. abgemeldet wird.

Die Operationen und die ihnen zugeordneten Aspekte sind die folgenden (siehe auch Anhang A.1.4):

- **`add-interest-name`, `remove-interest-name` und `add-interest-myname`, `remove-interest-myname`:**
  für den Aspekt „name“
- **`add-interest-display-significant-property`, `remove-interest-display-significant-property` und `add-interest-mydisplay-significant-property`, `remove-interest-mydisplay-significant-property`:**
  für den Aspekt „displaySignificantProperty“
- **`add-interest-position`, `remove-interest-position` und `add-interest-myposition`, `remove-interest-myposition`:**
  für den Aspekt „position“
- **`add-interest-bounds`, `remove-interest-bounds` und `add-interest-mybounds`, `remove-interest-mybounds`:**
  für den Aspekt „bounds“
- **`add-interest-container`, `remove-interest-container` und `add-interest-mycontainer`, `remove-interest-mycontainer`:**
  für den Aspekt „container“
- **`add-interest-added-component`, `remove-interest-added-component` und `add-interest-myadded-component`, `remove-interest-myadded-component`:**
  für den Aspekt „addedComponent“

- **`add-interest-removed-component`, `remove-interest-removed-component` und `add-interest-myremoved-component`, `remove-interest-myremoved-component`:**
  für den Aspekt „removedComponent“
- **`add-interest-added-child-model`, `remove-interest-added-child-model` und `add-interest-myadded-child-model`, `remove-interest-myadded-child-model`:**
  für den Aspekt „addedSubdiagram“
- **`add-interest-removed-child-model`, `remove-interest-removed-child-model` und `add-interest-myremoved-child-model`, `remove-interest-myremoved-child-model`:**
  für den Aspekt „removedSubdiagram“
- **`add-interest-origin-connection`, `remove-interest-origin-connection` und `add-interest-myorigin-connection`, `remove-interest-myorigin-connection`:**
  für den Aspekt „originConnection“
- **`add-interest-destination-connection`, `remove-interest-destination-connection` und `add-interest-mydestination-connection`, `remove-interest-mydestination-connection`:**
  für den Aspekt „destinationConnection“
- **`add-interest-delete`, `remove-interest-delete`:**
  für den Aspekt „delete“; nur sinnvoll, falls das interessierte Grapething ungleich dem Grapething ist, bei dem der Interest angemeldet wird, da der Interest-Handler erst nach der Zerstörung des Grapethings aufgerufen wird.
- **`add-interest-display-change`, `remove-interest-display-change` und `add-interest-mydisplay-change`, `remove-interest-mydisplay-change`:**
  für alle Aspekte, die Erneuern der Darstellung eines Graphobjekts betreffen, d,h, für die Aspekte „name„und „displaySignificantProperty“; Wrapper für die entsprechenden weiter oben beschriebenen Operationen der jeweiligen Aspekte.

Für die das An- und Abmelden von Interests-Handlern für Properties bleibt nicht anderes übrig, als die Standardoperationen zu verwenden, da hier der Name der Property den Aspekt angibt.

A.6.8 Operationen für die Workflow- und Artefaktsteuerung

Einige der Artefakt-Modelltypen — vor allem der SM-Modelltyp — verwenden andere Modelltypen, d.h. referenzieren als Property-Werte Graphobjekte von Modellen der anderen Typen. Dabei sind diese anderen Modelle teilweise über den ganzen Workflow verteilt, d.h. sind in Wirklichkeit Submodelle verschiedenster Workflowaktivitäten.

Um den Artefakt-Modellen eine einfach Kommunikation mit dem Workflow — im wesentlichen mit seinen anderen Artefakten — zu ermöglichen wurden in der Datei `mnm-wf-module-utilities.lib` eine Reihe von Hilfs-Operationen definiert. Alle diese Operationen sind generisch auf den allgemeinen Typen für Grapethings definiert, sind also in allen Modelltypen (innerhalb des Alter-Paketes `MNMService`) verwendbar.

- **Operation `(wf-top-graph model)`:**
  ermittelt für das Graphmodell `model` rekursiv das Parent-Graphmodell, das den gesamten Workflow repräsentiert (dies kann `model` auch selbst sein). Existiert kein übergeordneter Workflow, so wird `nil` zurückgegeben.

- **Operation `(wf-interaction grapething)`:**
ermittelt, die Workflowaktivität, unter der sich das Grapething `grapething` evtl rekursiv (in der Regel als Artefakt) befindet. Existiert keine übergeordnete Workflowaktivität, so wird `nil` zurückgegeben.

- **Operationen `get-wf-artifact`, `set-wf-artifact`, `unset-wf-artifact`, `add-wf-artifact-interest`, `add-wf-artifact-interest-and-test-now`:**
diese Operationen sind nicht nur innerhalb eines Workflowinstanz-Modells selbst (siehe Abschnitt 5.3, Abschnitt 6.2, Anhang A.5.1) sondern in beliebigen Modellen verwendbar. Die Parameter sind die gleichen, wie in Anhang A.5.1 beschrieben, bis auf den (ersten) Parameter für den Workflowgraphen. Statt dem Workflowgraphen kann jedes im Workflow (rekursiv) enthaltenen Grapething verwendet werden. `wf-top-graph` wird benutzt, um den Workflow selbst ausfindig zu machen.

- **Operation `(get-artifact grapething key-list)`:**
ähnlich zu `get-wf-artifact`, aber generischer: Operation für die generische Kommunikation zwischen beliebigen Grapethings (vor allem zwischen Artefakt-Modellen des Workflows). `key-list` auch hier eine hierarchische Schlüsselliste, deren einzelne Werte meist Alter-Symbole sind. Kann von allen Grapethings — vor allem Graphmodellen — überladen werden, um ihre Teilartefakte generisch anderen Grapethings zur Verfügung zu stellen. Wie das zurückgegeben Artefakt vom Grapething `grapething` ermittelt wird, bleibt diesem überlassen. Standardmäßig wird `nil` zurückgegeben.

  Die Kommunikation über `get-artifact` ist aber nur zum Auslesen, nicht zum Verändern von Daten gedacht, es gibt also keine Entsprechung zu den Operationen `set-wf-artifact!` und `unset-wf-artifact!`.

  Insbesondere auf die in einem Workflowmodul für eine Aktivität definierte Prozedur mit dem selben Namen `get-artifact` (in eigenem Modul-Alter-Paket), kann über `get-artifact` (im normalen Paket `MNMService`) zugegriffen werden.

  Umgekehrt kann eine Workflowaktivität von einem Submodell mit der speziell festgelegten Schlüsselliste `('wf-artifact-complete?)` anfragen, ob das Artefakt vollständig definiert ist oder nicht.

- **Operation `(get-wf-interaction-artifact gthing key-list)`:**
Wrapper, der zunächst mit `wf-interaction` die dem Grapething `gthing` übergeordnete Aktivität bestimmt und bei dieser dann über `get-artifact` mit der Schlüssenliste `key-list` das zugehörige Artefakt bestimmt.

- **Operation `(wf-interaction-done? gthing)`:**
stellt fest, ob die übergeordnete Aktivität – sofern sie existiert – abgeschlossen (done) ist oder nicht.

  Kann z.B. in einem Artefakt-Modell verwendet werden, um zu entscheiden, ob bestimmte Änderungen noch erlaubt sind oder nicht.

- **Operation `(get-child-artifact gthing key-list)`:**
Wrapper, der `get-artifact` auf dem mit `get-first-child` ermittelten Submodell – sofern es existiert – ausführt.

- **Operation `(get-parent-artifact gthing key-list)`:**
Wrapper, der `get-artifact` auf dem mit `parent` ermittelten Parent-Graphobjekt – sofern es existiert – ausführt.

Weiterhin gibt es für Grapethings, besonders für Graphmodelle, eine einfache Möglichkeit den Status der in ihnen definierten Teil-Artefakte als Übersichts-Text anderen Grapethings, vor allem dem Parent-Knoten, zur Verfügung zu stellen: Hierfür wird die Property mit Namen `dynamic-info` (siehe Anhang A.4) verwendet. Diese sollte als Wert stets eine Liste von Strings enthalten, die einer Liste von angezeigten Textzeilen entspricht. Die Zeilen sollen einzeln vorliegen, um leichtere Formatierung (Begrenzung der Anzahl und Länge) zu ermöglichen.

Grapethings sollten den Wert von `dynamic-info` konsistent zu ihrem internen Zustand halten, wobei sie aber selbst festlegen können was das genau bedeutet. Der enthaltene Text dient nur dem Benutzer als Übersicht. Interessierte Grapething registrieren typischerweise einen Grapething-Interest-Handler für `dynamic-info` und beziehen den Wert von `dynamic-info` in ihre Darstellung mit ein.

A.7 Zusammenfassung

In diesem Anhang wurden wichtige Details der Implementierung des Werkzeugs besprochen. Dabei wurde vor allem auf die wichtigen der entwickelten Operationen eingegangen, um eine leichtere Erweiterung bzw. Anpassung des Workflows zu ermöglichen.

Literaturverzeichnis

[AlterManual] *Alter Manual*, http://www.htc.honeywell.com/dome/AlterManual.pdf .

[argouml] *ArgoUML homepage*, http://argouml.tigris.org/ .

[cincom1] *Cincom Smalltalk*, http://www.cincom.com/scripts/smalltalk.dll/downloads/index.ssp?content=visualwork .

[CORBA 2.4] *The Common Object Request Broker: Architecture and Specification*. OMG Specification v2.4.2, Object Management Group, Februar 2001.

[delphi] *Borland Delphi*, http://www.borland.com/delphi/ .

[dia] *Dia a drawing program*, http://www.lysator.liu.se/˜alla/dia/ .

[dia2code] *Dia2Code Homepage*, http://dia2code.sourceforge.net .

[diacanvas] *DiaCanvas Homepage on SourceForge*, http://diacanvas.sourceforge.net/ .

[DomainModeling] *Knowledge, Software and Domain Modeling*, http://www.htc.honeywell.com/dome/DOMECourse/DOME .

[domainr3] *Domain R3*, http://objectdomain.com/domain/ .

[dome53] *Index of /sens/Software/DOME/DOME-5.3*, http://www.cse.msu.edu/sens/Software/DOME/DOME-5.3/ .

[DOMEGuide] *DoMEGuide*, http://www.htc.honeywell.com/dome/DOMEGuide.pdf .

[domehome] *DOME Home*, http://www.htc.honeywell.com/dome/ .

[gaphor] *Gaphor Homepage on SourceForge*, http://gaphor.sourceforge.net/ .

[GHH+ 01] GARSCHHAMMER, M., R. HAUCK, H.-G. HEGERING, B. KEMPTER, M. LANGER, M. NERB, I. RADISIC, H. ROELLE und H. SCHMIDT: *Towards generic Service Management Concepts – A Service Model Based Approach*. In: PAVLOU, G., N. ANEROUSIS und A. LIOTTA (Herausgeber): *Proceedings of the 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001)*, Seiten 719–732, Seattle, Washington, USA, Mai 2001. IFIP/IEEE, IEEE Publishing, http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/smtf01/smtf01.shtml .

[GHH+ 02] GARSCHHAMMER, M., R. HAUCK, H.-G. HEGERING, B. KEMPTER, I. RADISIC, H. ROELLE und H. SCHMIDT: *A Case–Driven Methodology for Applying the MNM Service Model*. In: STADLER, R. und M. ULEMA (Herausgeber): *Proceedings of the 8th International IFIP/IEEE Network Operations and Management Symposium (NOMS 2002)*, Seiten 697–710, Florence, Italy, April 2002. IFIP/IEEE, IEEE Publishing, http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/ghhk02/ghhk02.shtml .

[GHK+ 01] GARSCHHAMMER, M., R. HAUCK, B. KEMPTER, I. RADISIC, H. ROELLE und H. SCHMIDT: *The MNM Service Model — Refined Views on Generic Service Management*. Journal of Communications and Networks, 3(4):297–306, Dezember 2001, http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/ghkr01/ghkr01.shtml .

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[ISO 10040] *Information Technology – Open Systems Interconnection – Systems Management Overview*. IS 10040, International Organization for Standardization and International Electrotechnical Committee, 1992.

[JBR 99] JACOBSON, I., G. BOOCH und J. RUMBAUGH: *The Unified Software Development Process*. Addison–Wesley, Januar 1999.

[Kruc 00] KRUCHTEN, P.: *Rational Unified Process: An Introduction*. Addison-Wesley Object Technology Series. ADDISON-WESLEY, second Auflage, 2000.

[kuml] *kUML - UML for linux*, http://www.informatik.fh-hamburg.de/˜kuml/ .

[Lamp 86] LAMPORT, LESLIE: *A Document Preparation System LATEX*. Addison-Wesley, 1986.

[nsuml] *NSUML*, http://nsuml.sourceforge.net/ .

[OMG 01-02-14] *OMG UML Specification v. 1.4 (draft*. TC Document ad/01-02-14, Object Management Group, Februar 2001, ftp://ftp.omg.org/pub/docs/ad/01-02-14. with change bars);pdf .

[OMG 98-09-03] *XML Metadata Interchange (XMI) Seattle Presentation*. TC Document ad/98-09-03, Object Management Group, September 1998, ftp://ftp.omg.org/pub/docs/ad/98-09-03.pdf .

[OMG 99-07-65] *Smalltalk Language Specification*. OMG Specification formal/99-07-65, Object Management Group, Juli 1999, ftp://ftp.omg.org/pub/docs/formal/99-07-65.pdf .

[Python] *Python - A interpreted, interactive, object-oriented programming language*. WWW, http://www.python.org/doc .

[Roel 02] ROELLE, H.: *Allgemeines Dienstmanagement — Das MNM-Dienstmodell in Herleitung und Anwendungsmethodik*. In: *1. GI–Fachgespräch Applikationsmanagement*, Karlsruhe, Februar 2002. Gesellschaft für Informatik e.V. (GI).

[rrose] *Rational Rose v200...with Rational Rose*, http://www.rational.com/products/rose/index.jsp .

[Scheme] STEELE, G.-L. und G.-J. SUSSMAN: *Scheme - A LISP dialect*, http://www-swiss.ai.mit.edu/scheme-home.html .

[Schm 01] SCHMIDT, H.: *Entwurf von Service Level Agreements auf der Basis von Dienstprozessen*. Dissertation, Ludwig-Maximilians-Universität München, Juli 2001.

[stp] *Software through Pictures UML*, http://www.aonix.com/content/products/stp/uml.html .

[tcm] *TCM - Toolkit for Conceptual Modeling*, http://wwwhome.cs.utwente.nl/˜tcm/ .

[TOM1.1] *SMART TMN Telecom Operations Map*. Evaluation Version 1.1 GB910, TeleManagement Forum, April 1999, http://www.tmforum.com/pages/publications/books.html#telops .

[umlprog] *UML Object Modeller for Linux*, http://uml.sourceforge.net/ .

[vba] *Microsoft Visual Basic for Applications Homepage*, http://msdn.microsoft.com/vba/ .

[visio] *Microsoft Visio Homepage*, http://www.microsoft.com/office/visio/default.asp .

[vwsupport] *VisualWorks Support*, http://www.parcplace.com/support/ .

[xfig] *XFIG Drawing Program for the X Window System*, http://www.xfig.org/index.php .

[xml] *Extensible Markup Language (XML)*, http://www.w3.org/XML/ .

Index