(19) Republik Österreich Patentamt

(11) Nummer: AT 002 970 U2

(12) GEBRAUCHSMUSTERSCHRIFT

(21) Anmeldenummer: 232/98

(51) Int.Cl.[6] : G06F 17/60

(22) Anmeldetag: 2. 4.1998

(42) Beginn der Schutzdauer: 15. 6.1999

(45) Ausgabetag: 26. 7.1999

(73) Gebrauchsmusterinhaber:

CFC INFORMATIONSSYSTEME ENTWICKLUNGSGMBH
A-1010 WIEN (AT).

(54) VERFAHREN ZUR GENERATIVEN FERTIGUNG VON OBJEKTORIENTIERTER SOFTWARE AUF BASIS VON METAMODELLINFORMATION

(57) Das hier beschriebene Verfahren dient zur generativen Fertigung von objektorientierter Client/Server-Software auf Basis von Metamodellinformation. Das hier beschriebene Verfahren unterscheidet sich wie folgt von den anderen, herkömmlichen Verfahren zur Entwicklung solcher Systeme (nämlich die Verwendung einer 4GL Sprache oder eines Frameworks):

In dem Verfahren wird die gewünschte Funktionalität in einem sogenannten Modell beschrieben. Die formale Sprache, die verwendet wird, um das Modell zu beschreiben, ist das Metamodell. Das Verfahren ist dadurch gekennzeichnet, daß ein eigenes Metamodell, das genau für den Zweck der Modellierung solcher Systeme entworfen wurde, zur Beschreibung des Modells verwendet wird. Weiters ist man durch die Anwendung des Verfahrens in der Lage, die zur Implementierung des unter Verwendung des Metamodells beschriebenen Systems notwendige Programmlogik mit Hilfe von Generatoren automatisch zu erstellen. Durch die Verwendung des Metamodells wird gewährleistet, daß sich die gewünschte Programmlogik wirklich generieren läßt und man dadurch die gewünschte Qualitäts- und Produktivitätssteigerung erhält.

Die Beschreibung des Verfahrens gliedert sich in die folgenden Abschnitte. Zuerst wird das Metamodell beschrieben, das zur Modellierung der Funktionalität verwendet wird. Danach wird beschrieben, welche Programmlogik aufgrund des Modells generiert werden kann, und wie diese Generierung vonstatten geht.

AT 002 970 U2

Beschreibung

Die Beschreibung des Verfahrens gliedert sich in die folgenden Abschnitte. Zuerst wird das Metamodell beschrieben, daß zur Modellierung der Funktionalität verwendet wird. Danach wird beschrieben, welche Programmlogik aufgrund des Modells generiert werden kann, und wie diese Generierung vonstatten geht.

DAS METAMODELL

Es existieren heute bereits eine Reihe von Metamodellen für die Modellierung von objektorientierten Systemen (z.B. UML von Booch, Rumbaugh, Jacobson [Literaturempfehlung: UML Distilled von Martin Fowler, Addison Wesley 1997] oder OOA von Coad, Yourdon [Object-Oriented Analysis von Coad und Yourdon, Prentice-Hall 1991]). Das Metamodell unseres Verfahrens unterscheidet sich von diesen anderen Metamodellen dadurch, daß die Bedeutung (die Semantik) der einzelnen Sprachelemente genau definiert ist und es dadurch gewährleistet ist, daß sich bei Verwendung des Modells die gewünschte Programmlogik vollständig und richtig generieren läßt, während die Verwendung von herkömmlichen Metamodellen nur die Generierung von Programmgerüsten erlaubt, und die eigentliche Programmlogik manuell erstellt werden muß.
Das Metamodell bedient sich der folgenden Begriffe: Entity, ModelObject, ValueHolder, ValueHolderWithString, Relationship, Attribute, Reference, Inverse, Aspect, AspectRelationship, Interaction, CheckExistence Interaction, Read Interaction, ReadAndModify Interaction, Create Interaction, Delete Interaction, Search Interaction, ModelAspect, SearchAspect, Subaspect.
Zeichnung 3 zeigt die Beziehungen und Zusammenhänge zwischen den einzelnen Begriffen auf. Diese Begriffe werden im folgenden definiert.

Entity

Entities dienen zusammen mit den Relationships zur Beschreibung des statischen Teils des Modells. Der statische Teil bildet die Grundlage für die dynamischen Aspekte des Modells, die die eigentlichen Funktions-Abläufe beschreiben. Alle Entitäten des Modells werden durch Entities ausgedrückt. Entity ist der Oberbegriff für ModelObject und ValueHolder. Entities haben Beziehungen (Relationships) zu anderen Entities. Entities sind insbesondere dadurch gekennzeichnet, daß sie unvollständig, inkonsistent oder ungültig sein können. Durch diese scheinbar einfache und naheliegende Erweiterung der Semantik werden eine Reihe von Funktionalitäten erst möglich; diese sind: Verschiedene Funktionen, die sich auf die selben Entities beziehen, benötigen unterschiedliche Ausschnitte aus dem Modell. Diese werden durch Aspects definiert. Damit dies überhaupt möglich ist, ist es notwendig, ein Entity nur teilweise, also unvollständig zu instanzieren.
Moderne, objektorientierte Benutzerschnittstellen sind dadurch gekennzeichnet, daß der Benutzer die Reihenfolge der Aktionen im Zuge der Interaktion mit dem System festlegt; dadurch ist es möglich, daß ein Entity temporär in einen inkonsistenten Zustand gerät (z.B. kann eine Person mehrere, jedoch mindestens eine Adresse haben. Wird die letzte Adresse gelöscht, und dann eine neue Adresse angelegt, so hat die Person temporär keine Adresse. Wenn man diesen inkonsistenten Zustand nicht zuläßt, so wird der Benutzer unnötigerweise gezwungen, zuerst die neue Adresse anzulegen, und erst dann die alte Adresse zu löschen).

ModelObject

Ein ModelObject ist ein Entity, das mehrere Beziehungen zu anderen Entities aufweist. Person oder Adresse sind Beispiele für ModelObjects. ModelObjects haben mehrere Attribute-Beziehungen zu ValueHolders, bei einer Person z.B. deren Name und Vorname, und mehrere Attribute- oder Reference-Beziehungen zu anderen ModelObjects. ModelObjects sind dadurch gekennzeichnet, daß sie eine Identität besitzen, d.h. daß sie durch einen Schlüssel (Key) identifiziert werden können. Sie sind des weiteren persistent, d.h. sie behalten ihren Zustand über die Zeit ihrer Existenz (vom Zeitpunkt der Erzeugung bis zum Zeitpunkt des Löschens) hinweg; der Zustand wird in einer Datenbank gespeichert. Die Existenz von ModelObjects wird durch die Art der Beziehung, deren Teil sie sind, eingeschränkt: ist das ModelObject Teil einer Attribute-Beziehung, so wird es beim Löschen des übergeordneten Objekts implizit mitgelöscht. Ist das ModelObject nicht Teil einer Attribute-Beziehung, so wird es PrimaryObject genannt, und kann nur explizit gelöscht werden.

ValueHolder

Ein ValueHolder ist ein einfaches Entity, das zur Speicherung eines Namens, einer Zahl, eines Boolean-Wertes, eines Datums oder eines Zeitpunktes verwendet wird. Ein ValueHolder ist normalerweise über eine One-Attribute-Beziehung an ein ModelObject gebunden. Dieses wird auch als der Eigentümer (Owner) des

ValueHolders bezeichnet. Es sind aber auch alle anderen Arten von Beziehungen zu einem ValueHolder möglich. Der ValueHolder selbst kann keine weiteren Beziehungen zu anderen Entities haben.
In manchen anderen Metamodellen werden ValueHolder auch Attribute genannt. In unserem Metamodell sind die Konzepte der Struktur (Komplexe Objekte vs. einfache Attribute) und der Beziehung (Aggregation vs. Relationship) klar und streng orthogonal (d.h. unabhängig voneinander und uneingeschränkt kombinierbar) getrennt.

ValueHolderWithString

Ein ValueHolderWithString ist eine erweiterte Art von ValueHolder, der zusätzlich zu seinem Wert auch eine textuelle Repräsentation seines Werts beinhaltet. Durch die explizite Darstellung in zwei unterschiedlichen Repräsentationen ist es möglich, die textuelle Darstellung an spezielle Anforderungen wie z.B. Normen, vereinfachte Benutzereingabe, Sprach- und Länderspezifika anzupassen, ohne die Programmlogik zu beeinflussen.

Relationship

Eine Relationship stellt eine Beziehung zwischen zwei Entities her. Relationship ist der Oberbegriff zu Attribute, Reference und Inverse. Jede dieser Beziehungen gibt es in den Kardinalitäten One und Many. Die Kardinalität der Beziehung sagt über die Anzahl der in Bezug genommenen Entities aus. Eine Many-Relationship von Person zu Adresse sagt z.B. aus, daß jede Person mehrere Adressen haben kann. Die Kardinalität der Beziehung ist weiters genauer spezifiziert durch ein Kennzeichen das besagt ob die Beziehung obligatorisch (Mandatory) ist. Das Gegenteil von Mandatory wird Optional genannt. Eine Mandatory One-Beziehung muß immer auf ein gültiges Entity weisen, bei einer Optional One-Beziehung ist dies nicht der Fall. Eine Mandatory Many-Relationship muß auf mindestens ein Entity verweisen; eine Optional Many-Relationship kann auch leer sein, d.h. auf gar keine Entities verweisen.
Relationships haben, äquivalent zu Entities, die Eigenschaft, temporär unvollständig oder ungültig sein zu können, mit den selben wie bei den Entities beschriebenen Vorteilen.

Attribute

Eine Attribute-Beziehung sagt aus, daß das Ziel-Entity der Beziehung ein integraler Bestandteil des übergeordneten ModelObjects ist. Wenn das übergeordnete ModelObject gelöscht wird, wird immer auch das untergeordnete Entity gelöscht.
Jedes Entity kann nur Attribut genau eines Quell-ModelObjects sein.
In anderen Metamodellen wird diese Art der Beziehung manchmal Aggregation genannt.
Unser Metamodell unterscheidet sich von anderen Metamodellen dadurch, daß sowohl ModelObjects als auch ValueHolders als Ziel einer Attribute-Beziehung fungieren können.

Reference

Eine Reference-Beziehung impliziert, daß das referenzierte ModelObject vom referenzierenden ModelObject existentiell unabhängig ist. Wenn das referenzierende ModelObject gelöscht wird, bleibt das referenzierte ModelObject dennoch bestehen. Es ist eigenständig. Ein ModelObject kann mit beliebig vielen anderen ModelObjects über Reference-Beziehungen verbunden sein. Wird ein ModelObject gelöscht, so werden die Integritätsregeln der Reference-Beziehungen, deren Teilnehmer es ist, geprüft und das Löschen möglicherweise verhindert. Wird keine Integritätsregel durch das Löschen verletzt, so wird das ModelObject automatisch aus den Referenzen entfernt und sodann gelöscht.

Inverse

Eine Inverse-Beziehung kann für jede Attribute- und Reference-Beziehung definiert sein. In diesem Fall stellt die Inverse-Beziehung die Rückbeziehung vom untergeordneten ModelObject zum übergeordneten ModelObject dar. Dadurch kann das untergeordnete ModelObject auf das übergeordnete ModelObject zugreifen. Es kann sozusagen die Attribute- oder Reference-Beziehung rückwärts entlanggehen.

Aspect

Aspects beschreiben einen Ausschnitt aus dem durch Entities und Relationships gebildeten Objektmodell, der für eine bestimmte Anwendungsfunktionalität benötigt wird. Nur die notwendigen, durch den Aspect beschriebenen Daten werden vom Server aus der Datenbank gelesen und über das Netzwerk an den Client bzw. umgekehrt vom Client an den Server geschickt. Die Daten werden nicht auf einmal, sondern in Stücken geschickt. Die Reihenfolge, in der die Stücke gesendet werden, ist nicht festgelegt, sondern hängt von der dynamischen Verfügbarkeit der Daten ab. Dies ist insbesondere von Vorteil, wenn die Daten aus mehreren, verschiedenen Datenquellen stammen. Der Empfänger erhält die Daten so früh wie möglich, und kann sie bereits verwendet, ohne die vollständige Übertragung abwarten zu müssen. Das Konzept, nur Ausschnitte aus dem Modell zu übertragen ermöglicht es, daß sehr sparsam mit Speicher- und Netzwerkressourcen umgegangen

wird. Wenn z.B. zum Zweck der Anzeige nur der Name einer Person benötigt wird, wird dafür ein Personen-Aspekt verwendet, in dem nur der Name und der Vorname, nicht aber die Adressen enthalten sind.
Zu der Beschreibung der Aspects gehört neben den verwendeten Entities und Relationships auch die Verwendung der Daten. Die Verwendung der Daten wird einerseits definiert durch AspectRelationships, die für jede Relationship angeben, wofür (lesen, ändern, oder beides) die Beziehung verwendet wird, und andererseits durch die Angabe, für welche Interactions der Aspect verwendet wird.
Die durch die AspectRelationships definierte Information wird verwendet um festzulegen, in welcher Richtung Daten übertragen werden (vom Server zum Client, vom Client zum Server, oder beides).
Durch die Angabe der möglichen Interactions wird nur die notwendige Programmlogik generiert; dabei werden sowohl die Programmlogik für den Client als auch für den Server generiert, und zwar

AspectRelationship

Jeder Aspekt setzt sich aus einzelnen AspectRelationships zusammen. Eine AspectRelationship bezieht sich auf eine Relationship des Objektmodells und enthält zusätzliche Informationen. Es wird z.B. angegeben, ob die Relationship im Aspekt nur angesehen oder auch geändert werden darf. Bei Referenzen wird festgelegt, ob auch die Menge aller möglichen zuordenbaren ModelObjects Teil des Aspekts sein und im Speicher gehalten werden soll.

ModelAspect

Ein ModelAspect hält ein einzelnes ModelObject aus Hauptobjekt. Er wird zum Lesen, Erzeugen, Ändern und/oder Löschen eines ModelObjects verwendet.
Ein ModelAspect kann die folgenden, weiter unten beschriebenen Interactions durchführen: CheckExistence Interaction, Read Interaction, ReadAndModify Interaction, Create Interaction, Delete Interaction. Die genaue funktionale Definition der Interactions werden nachfolgend genau beschrieben.

SearchAspect

Ein SearchAspect enthält mehrere ModelObjects, die das Ergebnis einer Suche darstellen.
Zusätzlich definiert ein SearchAspect die für die Suche zu verwendenden Suchkriterien.
Ein SearchAspect ist in der lage, eine Search Interaction durchzuführen. Die Suche funktioniert dabei in Tranchen: Der Client kann selbst die gewünschte Anzahl von Objekten einer Tranche anfordern. Somit ist ein "vorwärtsblättern" im Suchergebnis möglich, und der Client ist gegen einen Speicherüberlauf geschützt.

Subaspect

Die Programmlogik des zugrundeliegenden Objektmodells unterstützt Events nach dem gängigen Model/View/Controller-Konzept (MVC). Die in unserem System eingesetzte Benutzerschnittstelle zeigt eine ganze Reihe von Attributen in einem View an. Dadurch entsteht das Problem, daß viele einzelne Änderungen in einem (Model-)Aspect viele Änderungen am View auslösen, und der Overhead des Aktualisierens des Views das System substantiell verlangsamt. Die Lösung des Problems sind Subaspects. Ein Subaspect bezeichnet einen Teilausschnitt eines Aspects, der die Granularität für das automatische Update definiert. Wenn sich ein Entity verändert, sollen diese Änderungen automatisch in den Views dargestellt werden. Wenn jedoch mehrere Änderungen eines Entities kurz hintereinander durchgeführt werden, würden sich die Views mehrmals updaten, was auf der einen Seite zu Performance-Problemen führt und andererseits ein Flimmern des Bildschirms verursacht. Subaspects vermeiden das, indem sie trotz mehrerer Änderungen nur einen Update der Benutzerschnittstelle durchführen. Sie "sammeln" die einzelnen Updates und lösen zu kontrollierten Zeitpunkten einen einzigen Update aus.

Interaction

Mit Interaction wird eine Konversation zwischen Client und Server bezeichnet, bei dem die Daten eines Aspects zwischen Client und Server ausgetauscht werden. Eine Interaction definiert ein fixes Kommunikations-Protokoll. Interaction ist der abstrakte Oberbegriff für die folgenden konkreten Protokolle: CheckExistence Interaction, Read Interaction (Replicated oder Precise), ReadAndModify Interaction, Create Interaction (WithKey oder WithoutKey), Delete Interaction, Search Interaction (Replicated oder Precise).
Die nachfolgenden Beschreibungen der einzelnen Interactions beinhalten das Kommumnikations-protokoll zwischen Client und Server. In diesen Beschreibungen bedeutet → eine Nachricht (Anforderung) vom Client zum Server, und ← eine Nachricht (Antwort) vom Server zum Client. Hinter den Pfeilen steht der Inhalt der Nachricht. Die Kommunikations-Protokolle, die bei unserem Verfahren zur Anwendung kommen, sind dadurch gekennzeichnet, daß sie die angeforderten Daten vom Server zum Client nicht auf einmal, sondern in einzelnen Teilpaketen gesendet werden. Der Client ist somit in der Lage, die bereits angekommenen, jedoch noch unvollständigen

Daten bereits zu verarbeiten (z.B. dem Benutzer anzuzeigen). Der Client ist durch diese Eigenschaft des Kommunikations-Protokolls in der Lage, unmittelbar auf Anfragen zu reagieren, und lange dauernde, unvollständige Abfragen zu stornieren.
Alle Interactions werden vom Client aus gestartet (d.h. der Server ist passiv, im Gegensatz zu sogenannten aktiven Servern).
In der Beschreibung wird manchmal zwischen Precise und Replicated-Interactions unterschieden. Diese Unterscheidung ist nur bei einer dreistufigen Architektur relevant. Siehe Zeichnung 1: Unterscheidung zwischen 2-stufiger und 3-stufiger Architektur.
Bei einer zweistufigen Architektur kommt nur die Precise-Interaction zur Anwendung. In einer dreistufigen Architektur wird bei einer Replicated-Interaction auf den dezentralen Server zugegriffen. Bei einer Precise-Interaction wird auf den zentralen Server zugegriffen.

CheckExistence Interaction

Eine CheckExistence Interaction prüft, ob ein ModelObject am Server existiert. In einer dreistufigen Architektur wird die Prüfung Precise, also in der zentralen Datenhaltung ausgeführt.
Kommunkationsprotokoll:
→***key***
←**exists** oder **doesNotExist** oder **fehler**
Der Client sendet den Schlüssel des zu überprüfenden ModelObjects. Der Server sendet exists wenn das ModelObject existiert, doesNotExist wenn das ModelObject nicht existiert, oder eine Fehlermeldung.

Read Interaction

Diese Interaction veranlaßt den Server, einen Aspect von der Datenhaltung zu lesen und an den Client zu senden. Diese Interaction gibt es im Fall der dreistufigen Architektur in zwei Ausprägungen, nämlich Read-Precise und Read-Replicated. Die Precise Interaction liest einen Aspect von der zentralen Datenhaltung. Die Replicated Interaction liest den Aspect von der dezentralen Datenhaltung. Die beiden Arten der Interaction unterscheiden sich vom Protokoll her nur durch die möglichen Fehlermeldungen.
Variante 1 (Normalfall):
→***key***
←***data***
←***data***
←**readCompleted** oder **fehler**
Der Client sendet den Schlüssel des zu Lesenden Aspects. Der Server sendet die einzelnen Datenpakete. Nach erfolgreicher Übertragung sendet der Server readCompleted. Im Fehlerfall wird die Übertragung durch das Senden einer Fehlermeldung abgebrochen.
Variante 2 (der Vorgang wird abgebrochen, bevor das Objekt zur Gänze gelesen wurde):
→***key***
←***data***
→**cancelRead**
←**readCanceled**
Der Client kann von sich aus die laufende Übertragung abbrechen.
Variante 3 (Fehlerfall):
→***key***
←**fehler**
Im Fehlerfall sendet der Server eine Fehlermeldung.

ReadAndModify Interaction

Die ReadAndModify Interaction dient zum Ändern von bereits existierenden Aspects. Der Server liest einen Aspect und sendet ihn zum Client. Der Aspect wird am Client verändert und zum Schreiben an den Server zurückgeschickt.
Variante 1 (Normalfall):
→***key***
←***data***
←***data***
←**readCompleted**
→ ***data***
←**modifyCompleted** oder **fehler**
Der Client sendet den Schlüssel des zu ändernden Aspects. Der Server sendet den Aspect, und danach die Bestätigung readCompleted. Der Client sendet die geänderten Daten des Aspects. Der Server beendet das Protokoll durch Senden vom modifyCompleted oder einer Fehlermeldung.
Variante 2 (der Vorgang wird abgebrochen, bevor das Objekt zur Gänze gelesen wurde):
→***key***
←***data***
→**cancelRead**
←**readCanceled**
Variante 3 (der Vorgang wird abgebrochen, nachdem das Objekt gelesen wurde):
→***key***
←***data***
←***data***
←**readCompleted**
→**cancelModify**
←**modifyCanceled**
Variante 4 (Fehlerfall):
→***key***
←**fehler**

CreateWithKey Interaction

Diese Interaction erzeugt ein neues ModelObject, wobei der Schlüssel vom Client vorgegeben wird. Diese Interaction wird angewendet bei dezentraler Vergabe von Schlüsseln oder manueller Organisation mit Schlüssellisten.
Kommunkationsprotokoll:
→***key + data***
←**createCompleted** oder **fehler**

Der Client sendet den Schlüssel und die Daten des ModelObjects.

CreateWithoutKey Interaction

Eine CreateWithoutKey Interaction erzeugt ein neues ModelObject, wobei der Schlüssel vom Server vorgegeben wird.
Kommunkationsprotokoll:
→*data*
←**createCompleted** + *key* oder **fehler**
Der Client sendet die Daten des ModelObjects. Der Server sendet im Erfolgsfall den Schlüssel des erzeugen ModelObjects, oder eine Fehlermeldung.

Delete Interaction

Die Delete Interaction dient zum Löschen von bereits existierenden ModelObjects. Es kommen bei der Verwendung dieser Interaction automatisch die impliziten Löschregeln zur Anwendung. Das bedeutet, daß zuerst geprüft wird, ob das ModelObject in Beziehungen (Relationships) teilnimmt, deren Integrität durch das Löschen nicht mehr gewährleistet ist. Ist die Gefahr von Inkonsistenzen nicht gegeben, so wird das ModelObject und alle seine Attribute (und alle Attribute seiner Attribute und so weiter) gelöscht.
Kommunkationsprotokoll:
→*key*
←**deleteCompleted** oder **fehler**

Search Interaction

Die Suche nach Aspects wird durch die Angabe eines Suchkriteriums eingeschränkt. Ein Suchkriterium wird gleich behandelt wie ein gewöhnliches ModelObject.
Die Suche wird quantitativ eingeschränkt. Dies wird durch die Verwendung von Such-Tranchen realisiert. Der Client gibt an, wieviele Suchergebnisse er empfangen kann/will. Nach dem Empfang der angegebenen Anzahl an Daten hat der Client die Möglichkeit, die nächste Tranche der Suchergebnisse anzufordern.
Die Search Interaction gibt es in zwei Unterarten, Precise und Replicated. Die Unterscheidung erfolgt analog zur Read Interaction durch die Quelle der durchsuchten Daten.
Variante 1 (Normalfall):
→*search criteria*
←**searchStarted** (optional: + *count*)
→**fetch next** *nnn*
←*data*
←*data*
←**fetchCompleted**
→**fetch next** *nnn*
←*data*
←*data*
←**searchCompleted**
Variante 2 (der Suchvorgang wird abgebrochen):
→*search criteria*
←**searchStarted** (optional: + *count*)
→**fetch next** *nnn*
←*data*
→**cancel search**
←**searchCanceled**
Variante 3 (Fehlerfall):
→*search criteria*
←**fehler**

Die Generierte Programmlogik

Mit dem hier beschriebenen Metamodell können ca. 95% des Source-Codes für den Filialserver und für den Zentralserver und ca. 50% des Source-Codes für den Client generiert werden. Die Anteile des Anwendungs-Sourcecodes am Gesamtumfang hängt von der Art und dem Funktionsumfang des verwendeten Frameworks ab. Ein umfangreiches Framework erfordert wenig generierten Sourcecode und umgekehrt. Zeichnung 2 verdeutlicht diesen Sachverhalt.
Das Metamodell wurde so entworfen, das nur ein wenig umfangreiches und mächtiges Framework notwendig ist, um dafür Sourcecode zu generieren. Dadurch ist gewährleistet, daß ein Umstieg auf ein anderes Framework leicht möglich ist.
In der Folge werden die einzelnen Generatoren und die von ihnen generierte Programmlogik beschrieben.

ModelGenerator

Dieser Generator setzt ein Framework voraus, das eine abstrakte Superklasse (im Sinn der objektorientierten Programmiersprache) für die generierten Klassen zur Verfügung stellt.
Mit diesem Generator werden aus den ModelObjects Klassen generiert. Für die Klassen werden Instanzvariablen (members) für die Relationships generiert. Weiters werden Zugriffsmethoden (get und set) für die einzelnen Relationships generiert. Ebenfalls generiert wird die Programmlogik zum Instanzieren von neuen Relationships. Die generierte Programmlogik wird sowohl im Client als auch im Server verwendet.

AspectGenerator

Dieser Generator setzt ein Framework voraus, das eine abstrakte Superklasse für die generierten Aspect-Klassen zur Verfügung stellt.
Mit dem Aspect-Generator werden aufgrund der Metainformation über ModelObjects, Relationships und Aspects die Aspect-Klassen generiert. Weiters werden Methoden generiert, die

zum Instanzieren von Aspects verwendet werden. Es wird dabei auf die vom ModelGenerator generierten Methoden zurückgegriffen. Der generierte Source-Code wird sowohl im Client als auch im Server verwendet.

AspectPathGenerator

Mit diesem Generator werden Methoden erzeugt, mit denen man Aspect-Ausschnitte, sogenannte AspectPaths, über die Client-Server-Kommunikationsschnittstelle in beide Richtungen senden kann. Dabei wird darauf geachtet, daß nur die minimal notwendige Information gesendet wird, um Probleme mit der Antwortzeit zu vermeiden. Der generierte Source-Code wird sowohl im Client als auch im Server verwendet.

TableGenerator

Damit werden die Datenbank-Scripts aus der Metainformation über Entities und Relationships erzeugt, mit deren Hilfe man das notwendige Datenbankschema für die dezentrale und zentrale Datenbank erstellen kann. Derzeit können die Scripts für die Datenbanken DB2 und Oracle generiert werden. Die zugrundeliegende Logik ist jedoch für alle relationalen Datenbanken gleich. Die generierten Datenbank-Scripts werden nur im Server verwendet.

RDBViewGenerator

Dieser Generator erzeugt aufgrund der Metainformation über ModelObjects, Relationships und Aspects Zugriffs-Methoden für die Datenbank, die die einzelnen Aspects benötigen. Diese Methoden greifen auf die vom TableGenerator generierten Datenbanktabellen zu. Der generierte Source-Code wird nur im Server verwendet.

ProcessMethodGenerator

Process-Methoden dienen dazu, am Server die für eine bestimmte Interaction eines Aspects benötigten Daten aus der Datenbank zu lesen oder in die Datenbank zu schreiben. Sie sind der Kern der eigentlichen Serverfunktionalität und für das Verfahren von eminenter Bedeutung. In den Process-Methoden werden die vom RDBViewGenerator generierten Zugriffsmethoden verwendet, und die vom AspectPathGenerator generierten Kommunikations-Methoden verwendet. Außerdem werden die Daten je nach Interaction vom Client gelesen oder zum Client geschickt, analog zu den in den vorigen Kapiteln beschriebenen Kommunikationsmustern. Der generierte Source-Code wird nur im Server verwendet.

SubaspectGenerator

Damit werden Methoden generiert, die Subaspects implementieren. Diese werden von der Benutzerschnittstelle verwendet, um sich auf Änderungen im Objektmodell zu registrieren. Die Benutzerschnittstelle wird von den Subaspects benachrichtigt, wenn sie sich updaten soll, wobei die Subaspects darauf achten, daß bei kurz aufeinander folgenden Änderungen des Objektmodells nur eine Benachrichtigung erfolgt, sodaß keine Performance-Probleme entstehen. Der generierte Source-Code wird nur im Client verwendet.

Ansprüche

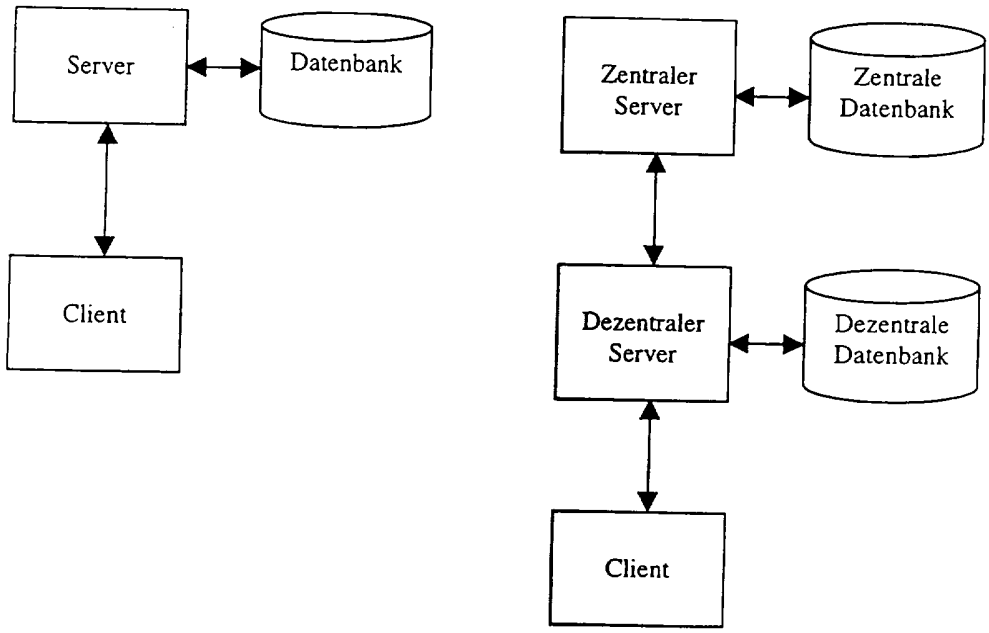
1. Verfahren zur Spezifikation von Client/Server-Software, wobei die Beschreibung der Funktionalität (das Modell) ausgedrückt wird durch die Begriffe Entity und Relationship, gekennzeichnet dadurch, daß:
   die Beschreibung der Funktionalität zusätzlich ausgedrückt wird durch die Begriffe
   Aspect,
   Interaction, und
   Subaspect.
2. Verfahren nach Anspruch 1, gekennzeichnet dadurch, daß:
   der Begriff Entity eine Klasse von Objekten ist, der weiter verfeinert wird durch die Varianten (Subklassen) ModelObject, ValueHolder und ValueHolderWithString, wobei ModelObject eine Klasse ist, die Beziehungen zu anderen Entities haben kann, ValueHolder eine Klasse ist, die keine weiteren Beziehungen zu anderen Entities haben kann und die einen Wert (value) hat und ValueHolderWithString eine ValueHolder-Klasse ist, die zusätzlich zu seinem Wert eine textuelle Repräsentation seines Werts hat.
3. Verfahren nach Anspruch 1, gekennzeichnet dadurch, daß:
   der Begriff Relationship eine Verbindung (Beziehung) zwischen zwei Entities ist, der weiter verfeinert wird durch die Subklassen Attribute, Reference, und Inverse, wobei Relationships gerichtet sind, das heißt sie haben eine Quelle (source) und ein Ziel (target); wobei Attribute bedeutet, daß das Ziel von der Quelle existentiell abhängt und wird die Quelle gelöscht, so wird das Ziel automatisch gelöscht; wobei Relationship bedeutet, daß keine existentielle Abhängigkeit zwischen Quelle und Ziel existiert, wobei es jedoch eine Konsistenzregel geben kann, die das Löschen des Ziels verhindert, wenn es Teil einer Relationship ist; wobei Inverse eine Relationship bedeutet, die symmetrisch zu einem Attribute oder einer Reference, jedoch in der anderen Richtung (Quelle und Ziel vertauscht) angelegt ist, wobei Inverse nur dann verwendet werden, wenn das Ziel die Notwendigkeit hat, zu seiner Quelle zu gelangen.
4. Verfahren nach Anspruch 1, gekennzeichnet dadurch, daß:
   der Begriff Aspect einen Ausschnitt aus dem Modell darstellt, der den Datenumfang (und somit den Funktionsumfang) einer Interaction definiert wobei ein Aspect bezeichnet, welche Entities und welche Relationships an der Interaction teilnehmen.
5. Verfahren nach Anspruch 1, gekennzeichnet dadurch, daß:
   ein Aspect ausgedrückt wird durch eine Liste von AspectRelationships wobei ein AspectRelationship eine Relationship bezeichnet (somit die Quelle und das Ziel der Relationship), und die Verwendung der Relationship und der zugehörigen Entities eine Interaction; wobei die Verwendung ausgedrückt wird durch die Kennzeichen Lesen und Schreiben, wobei auch beide Kennzeichen gegeben sein können; wobei Lesen bedeutet, daß die die Relationship repräsentierenden Daten vom Server zum Client übertragen werden und Schreiben bedeutet, daß die die Relationship repräsentierenden Daten vom Client zum Server übertragen werden.
6. Verfahren nach Anspruch 1, gekennzeichnet dadurch, daß:
   der Begriff Aspect verfeinert wird in die beiden Subklassen ModelAspect und SearchAspect wobei ein ModelAspect die Interactions CheckExistence Interaction, Read Interaction, ReadAndModify nteraction, Create Interaction, und Delete Interaction ausführen kann; wobei die Definition des ModelAspects die Information beinhaltet, welche der genannten Interactions unterstützt werden; wobei ein SearchAspect eine Search Interaction ausführen, also Entities suchen kann; wobei die Definition des SearchAspects ein Suchkriterium beinhaltet; wobei ein Suchkriterium analog zu einem ModelAspect spezifiziert wird und wobei die AspectRelationships dabei jedoch immer nur das Schreiben-Kennzeichen haben.
7. Verfahren nach Anspruch 1, gekennzeichnet dadurch, daß:
   der Begriff Interaction ein Kommunikations-protokoll zwischen Client und Server zum Zwecke der jeweiligen Funktionalität definiert und verfeinert wird durch die Begriffe CheckExistence Interaction, Read

Interaction, ReadAndModify Interaction, Create Interaction, Delete Interaction und Search Interaction wobei eine CheckExistence Interaction überprüft, ob das gesuchte ModelObject existiert; wobei eine Read Interaction die Daten eines ModelAspects liest; wobei eine ReadAndModify Interaction die Daten eines ModelAspects liest und es erlaubt, die veränderten Daten zu speichern; wobei eine Create Interaction es erlaubt, einen ModelAspect zu erzeugen; wobei eine Delete Interaction einen ModelAspect löscht und wobei eine Search Interaction zum Suchen von SearchAspects nach einem komplexen Suchkriterium dient und die Daten in Tranchen liefert.

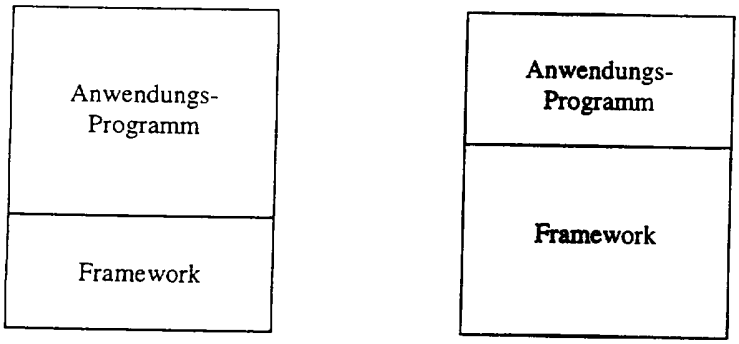
8. Verfahren zum Generieren der Klassendefinitionen aus den ModelObjects und der Programmlogik gekennzeichnet dadurch, daß:
   es zum Instanzieren der ModelObjects und seiner unmittelbaren Relationships aufgrund eines Modells dient, das durch ein Metamodell entsprechend den Ansprüchen 1 - 7 ausgedrückt ist.
9. Verfahren zum Generieren von Definitionen, gekennzeichnet dadurch, daß:
   es Aspektklassendefinitionen für eine im Framework vorhandene Aspect-Superklasse aus dem Modell entsprechend den Ansprüchen 1 – 7 generiert.
10. Verfahren zum Generieren der Programmlogik gekennzeichnet dadurch, daß:
    es den Aspect und seine zugeordneten Entities und Relationships aus dem Modell entsprechend den Ansprüchen 1 – 7 instanziert.
11. Verfahren zum Generieren der Programmlogik gekennzeichnet dadurch, daß:
    es AspectPaths aus dem Modell entsprechend den Ansprüchen 1 - 7 durch Entlangwandern (graph traversal) der Relationships und direkter Umsetzung in entsprechende Methoden für jede Relationship generiert.
12. Verfahren zum Generieren von Definitionen gekennzeichnet dadurch, daß:
    der Tabellendefinitionen des Datenbankschemas aus dem Modell entsprechend den Ansprüchen 1 – 7 generiert werden, wobei zuerst die Namen der ModelObjects und Relationships auf die Namenskonvention der Zieldatenbank normiert weden, danach für jedes ModelObject eine Tabelle angelegt wird, für alle One-Attributes zu einem ValueHolder ein Tabellenattribut des dem ValueHolder entsprechenden Typs angelegt wird, für alle One-Attributes zu einem ModelObject ein Tabellenattribut für den Schlüssel des Zielobjekts angelegt wird, für alle Many-Attributes eine eigene Zwischentabelle angelegt wird mit dem Schlüssel des Quellobjekts und einem Tabellenattribut für den ValueHolder oder dem Schlüssel des Zielobjekts, für alle One-References ein Tabellenattribut für den Schlüssel des Zielobjekts angelegt wird und für alle Many-References eine Zwischentabelle mit dem Schlüssel des Quellobjekts und dem Schlüssel des Zielobjekts angelegt wird.
13. Verfahren zum Generieren von Definitionen. gekennzeichnet dadurch, daß:
    Viewdefinitionen des Datenbankschemas aus dem Modell entsprechend den Ansprüchen 1 – 7 generiert werden, wobei alle ModelObjects, die in einer Vererbungshierarchie stehen, in einem Datenbankview zusammengefaßt werden, die konkrete Klasse des ModelObjects durch ein eigenes Attribut erweitert wird, das automatisch mit dem originären Tabellennamen des ModelObjects verknüpft wird und für jeden Aspect eigene Views erzeugt werden, die nur die in dem Aspect vorkommenden Datenbank-attribute beinhalten.
14. Verfahren zum Generieren der Programmlogik gekennzeichnet dadurch, daß:
    die Programmlogik für das Serverprogramm generiert wird, das die Funktionalität der einzelnen Interactions für die einzelnen Aspects aus dem Modell entsprechend den Ansprüchen 1 – 7 implementiert, wobei die Interactions dabei logisch in Lese- und Schreib-Operationen aufgetrennt werden; wobei der Modellausschnitt, der durch den Aspect vorgegeben ist, analysiert (graph traversal) wird und für jedes involvierte ModelObject die entsprechende Lese- beziehungsweise Schreib-Operation aufgerufen wird; wobei die für den Aspect optimierten Datenbankviews verwendet werden, falls diese vorhanden sind, ansonsten die allgemeinen, generischen Datenbankviews verwendet werden und die überflüssigen Datenbankattribute verworfen werden; wobei die gelesenen Daten in AspectPaths umgeformt und über ein Kommunikationsframework verschickt

(data-Pakete) werden, wobei die Programmlogik berücksichtigt, daß eine Interaction storniert werden kann, indem periodische Abfragen des Stornokennzeichens generiert werden.

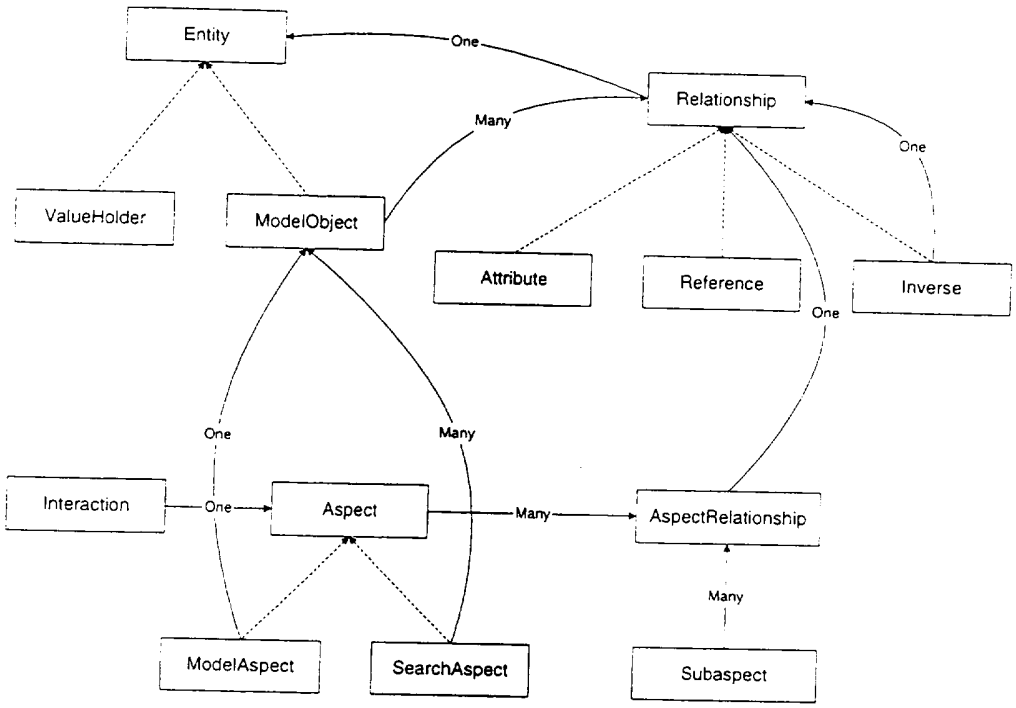
15. Verfahren zum Generieren von Definitionen gekennzeichnet dadurch, daß:
    Subaspects aus dem Modell entsprechend den Ansprüchen 1 – 7 generiert werden, wobei die Subaspect-Definition in die Programmlogik umgesetzt wird, sodaß für jede Relationship des Subaspects die Änderungsnachricht (entsprechend der gängigen Model/View/Controller-Logik) registriert wird und mittels eines Verzögerungszählers (defer count) die Weiterleitung der Änderungsnachricht verzögert zu kontrollierten Zeitpunkten weitergeleitet werden.



Zeichnung 1: Unterscheidung zwischen 2-stufiger und 3-stufiger Architektur.



Zeichnung 2: Zusammenhang zwischen Funktionsumfang des Frameworks und Umfang des generierten Sourcecodes.



Zeichnung 3: Zusammenhänge zwischen den Begriffen des Metamodells.

Anmerkung zum Gebrauchsmusterantrag:
Verfahren zur generativen Fertigung von objektorientierter Software auf Basis von Metamodellinformation

In dem Gebrauchsmusterantrag wird, auf Seite 2, das beschriebene Verfahren mit zwei gängigen Verfahren zur Beschreibung von Software verglichen: UML und OOA.
In dieser Anmerkung sollen die beiden Methoden kurz beschrieben und Literaturhinweise gegeben werden.

Die UML (Unified Modelling Language) entwickelt sich in jüngster Zeit zum de-facto-Standard für Objektmodellierung. Beinahe alle CASE-Tools unterstützen heute die UML-Notation. Es gibt eine große Anzahl an Literatur über die UML. Wir verwenden und empfehlen für den Überblick das Buch "UML Distilled" von Martin Fowler (Addison Wesley 1997).

Eine andere, vor allem in den USA gängige Methode ist OOA (Object-Oriented Analysis) von Coad und Yourdon. Das Standardwerk dazu ist "OOA: Object-Oriented Analysis" von den beiden Erfindern (Prentice-Hall 1991).

Dieser Anmerkung liegen zwei Anlagen bei:

Anlage A ist eine Kopie aus dem Buch "OOA: Object-Oriented Analysis" von Coad und Yourdon und gibt einen Überblick über die Notation der Methode OOA.

Anlage B ist eine Kopie aus dem Buch "UML Distilled" von Martin Fowler und beschreibt den Zweck von Metamodellen und Notationen. Bemerkenswert ist hier die Aussage, daß die Modelle hauptsächlich zur Kommunikation dienen sollen, und es daher notwendig ist, es mit der Semantik nicht so genau zu nehmen ("to bend the language"). Das beantragte Verfahren geht von einem konträren Ansatz aus: Nur wenn man die Semantik genau spezifiziert und auch einhält, läßt sich Software aus den Modellen generieren und sich ein Produktivitätsvorteil erzielen.

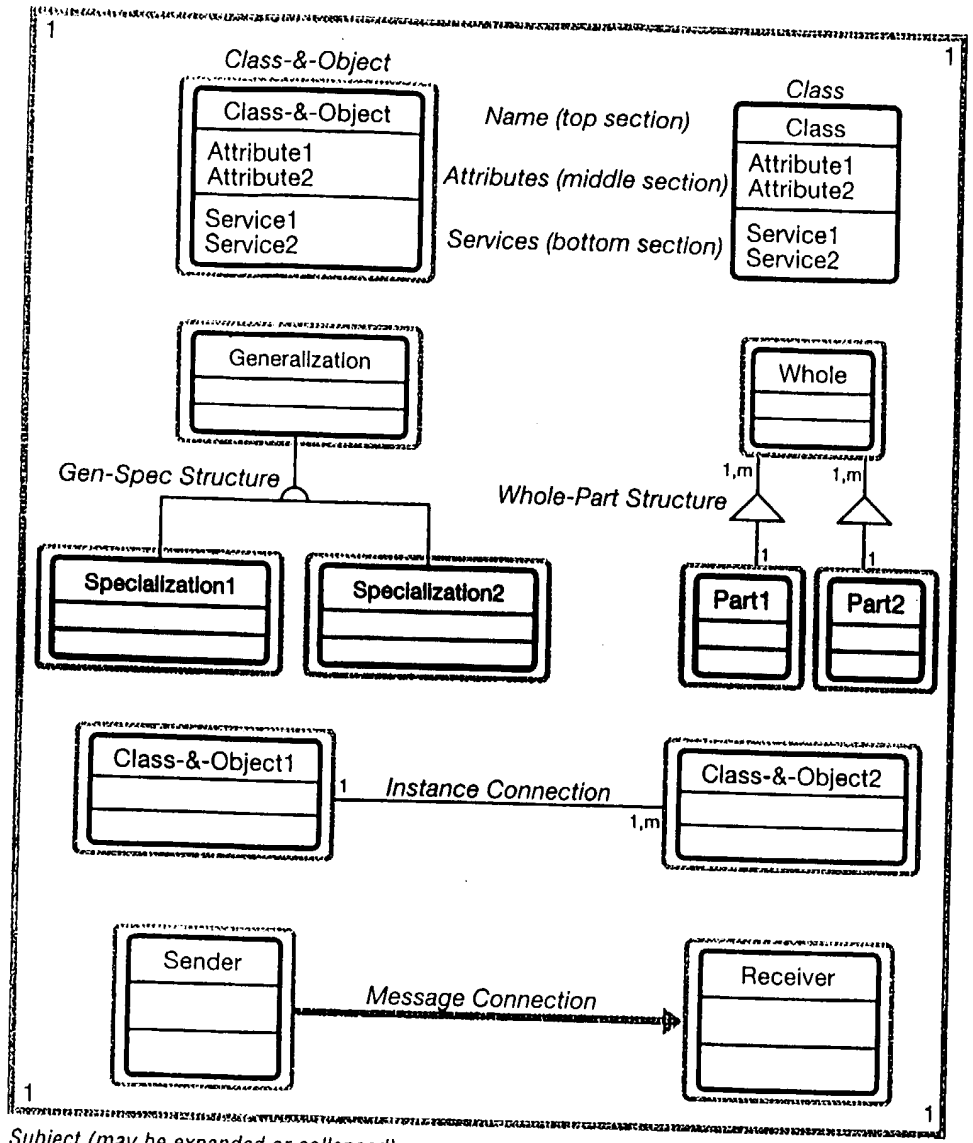


Figure A.1: OOA notations

specification

attribute
attribute
attribute

externalInput
externalOutput

objectStateDiagram

additionalConstraints

notes

service <name & Service Chart>
service <name & Service Chart>
service <name & Service Chart>

and, as needed,

traceabilityCodes
applicableStateCodes
timeRequirements
memoryRequirements

Figure A.2: Class-&-Object specification template

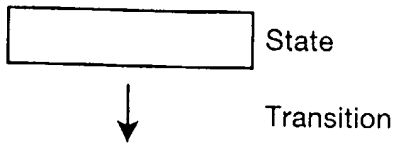


Figure A.3: Object State Diagram notation (used within the template)

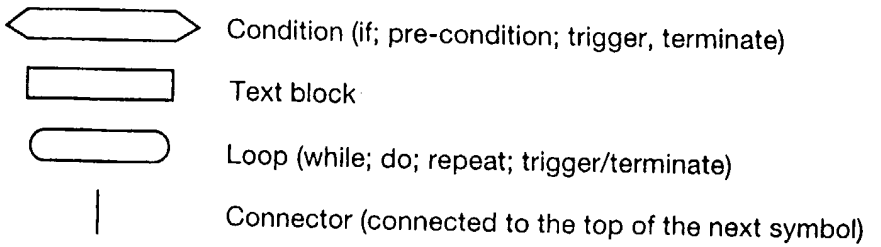


Figure A.4: Service Chart notation (used within the template, for each Service)

Notations and Meta-Models

The UML, in its current state, defines a notation and a meta-model.

The notation is the graphical stuff you see in models; it is the syntax of the modeling language. For instance, class diagram notation defines how items and concepts such as class, association, and multiplicity are represented.

Of course, this leads to the question of what exactly is meant by an association or multiplicity or even a class. Common usage suggests some informal definitions, but many people want more rigor than that.

The idea of rigorous specification and design languages is most prevalent in the field of formal methods. In such techniques, designs and specifications are represented using some derivative of predicate calculus. Such definitions are mathematically rigorous and allow no ambiguity. However, the value of these definitions is by no means universal. Even if you can prove that a program satisfies a mathematical specification, there is no way to prove that the mathematical specification actually meets the real requirements of the system.

Design is all about seeing the key issues in the development. Formal methods often lead to getting bogged down in lots of minor details. Also, formal methods are hard to understand and manipulate, often harder to deal with than programming languages. And you can't even execute them.

Most OO methods have very little rigor; their notation appeals to intuition rather than formal definition. On the whole, this does not seem to have done much harm. These methods may be informal, but many people still find them useful—and it is usefulness that counts.

However, OO methods people are looking for ways to improve the rigor of methods without sacrificing their usefulness. One way to do this is to define a meta-model: a diagram, usually a class diagram, that defines the notation.

Figure 1-1 is a small piece of the UML 1.0 meta-model that shows the relationship among associations and generalization. (The extract is there just to give you a flavor of what meta-models are like. I'm not even going to try to explain it.)

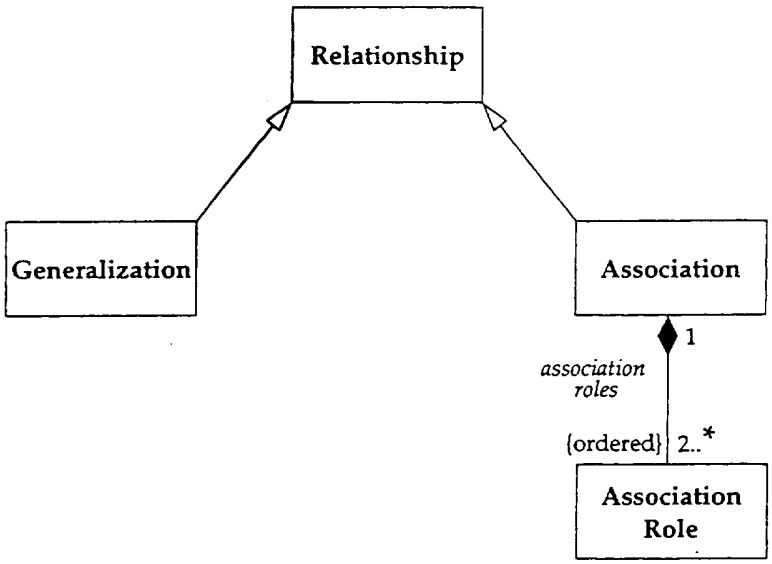


Figure 1-1: *UML 1.0 Meta-Model Extract*

How much does the meta-model affect the user of the modeling notation? Well, it does help define what is a well-formed model—that is, one that is syntactically correct. As such, a methods power user should understand the meta-model. However, most users of methods do not need such deep understanding to get some value out of using the UML notation.

This is why I can write a useful book now, even though the UML meta-model is not completely defined—indeed, it won't reach that point until after the OMG approval process is complete. I will not be rigorous in this book; rather, I will follow the traditional methods path and appeal to your intuition.

How strictly should you stick to the modeling language? That depends on the purpose for which you are using it. If you have a CASE tool that generates code, then you have to stick to the CASE tool's interpretation of the modeling language in order to get acceptable code. If you are using the diagrams for communication purposes, then you have a little more leeway.

If you stray from the official notation, then other developers will not fully understand what you are saying. However, there are times when the official notation can get in the way of your needs. I'll admit that in these cases, I'm not at all afraid to bend the language. I believe that the language should bend to help me communicate, rather than the other way around. But I don't do it often, and I'm always aware that a bend is a bad thing if it causes communication problems. In this book, I mention those places where I'm inclined to do a bit of bending.