Alf Borrmann

Rational Rose und UML

Galileo Computing

Inhalt

Vorbemerkungen

Vorwort

Muss man Idealist sein, um ein Buch zu schreiben? Nein, aber es hilft doch sehr. Diese Abwandlung des Spruchs »Muss man verrückt sein, hier zu arbeiten ...« fiel mir ein, als ich darüber nachdachte, weshalb ich dieses Projekt begonnen habe. Nun, mein Antrieb, dieses Buch zu machen, kommt tatsächlich daher, dass ich meine, dass das Ergebnis und der Ablauf von Softwareprojekten nach wie vor stark verbesserungswürdig und auch -fähig ist. Eine Möglichkeit, diese sehr komplexe Arbeit produktiver und besser strukturiert zu gestalten, ist der Einsatz von CASE-Tools.

In meiner Arbeit habe ich mich daher schon vor Jahren dafür entschieden, Rational Rose einzusetzen. Hiermit habe ich meine Ideen dokumentiert und mir eine Möglichkeit geschaffen, durch die Stück-für-Stück-Erstellung einzelner Modellelemente den Prozess der Konstruktion meiner Software zu strukturieren.

Ziemlich schnell fiel mir dann auf, dass es für die Verbesserung des Arbeitsablaufs bei der Softwareentwicklung nicht reicht, wenn ich als Einziger im Projekt Tools einsetze und damit umgehen kann. Und weil das neben mir auch Entwickler in anderen Teams so sahen, bekam ich die ersten Anfragen, Schulungen zu geben.

Da Schulungsunterlagen im Jahre 1996 zu Rose oder der Objektorientierung praktisch nicht vorhanden waren, habe ich diese eben selbst erstellt, über die Jahre hinweg immer wieder erweitert und an neue Versionen, Sprachen und Techniken angepasst. Schließlich hatten diese Unterlagen fast 200 Seiten Umfang und von dort war es dann – scheinbar – nur ein kleiner Schritt, daraus ein Buch zu machen.

Um es allerdings gleich vorweg zu sagen: Dieses Buch kann und will – wie alle anderen Bücher in dem Umfeld auch – keine Anleitung dazu sein, wie Softwareprojekte »automatisch« zum Erfolg zu führen sind. Es will Ihnen eine Anleitung geben, wie Sie Rational Rose in Ihren Projekten so einsetzen, dass Sie den optimalen Nutzen aus seiner Anwendung ziehen. Daneben sind aber viele weitere Dinge bei der Erstellung von objektorientierter Software zu beachten, und auf diese geht das vorliegende Buch ganz bewusst nicht ein. Statt dessen verweise ich auf die inzwischen in großer Zahl und in guter Qualität vorliegenden Titel zu Themen wie Objektorientierung, UML und Projektmanagement.

Einfach die Idee zu haben, ein Buch zu schreiben, reicht natürlich nicht, um es zu realisieren. Meine Idee fiel aber in eine Zeit – Beginn des Jahres 2001 – zu der auch andere Leute erkannt haben, dass es an der Zeit ist, praxisgerechte Unterlagen für den Einsatz von Rose zur Verfügung zu haben, und ich hatte das Glück, diese Leute auch zu finden.

Normalerweise stehen an dieser Stelle in einem Buch Danksagungen, meist an den Lebenspartner oder die Familie. Dann kommt der Hinweis, dass es eine riesige Arbeit gewesen ist, das Buch zu schreiben und eine Aufzählung von Personen, denen man zu Dank verpflichtet ist. Ich möchte Ihnen den Hinweis auf die ungeheure Arbeit ersparen. Denn in Wirklichkeit hat es mir riesigen Spaß gemacht, dieses Buch zu schreiben und mit den Coautoren und den anderen Beteiligten zu arbeiten. Ich hoffe, etwas von der Energie und dem Schwung, den wir alle bei der Arbeit hatten (denn – ja, Arbeit war es natürlich auch), wird sicht- und fühlbar in dem, was Sie nun zwischen den Pappdeckeln vorfinden.

Damit bin ich dann aber doch dabei, den Beteiligten zu danken. Als erstes fällt mir dabei Stefan Komnick von der Firma TERACOM in Berlin ein. Er hat mir in glanzvoller Manier die Arbeit des Projektmanagements abgenommen und hat damit den vielleicht größten Anteil am Gelingen des ganzen Vorhabens. Ich kann nur jedem Autor, der seine Arbeit mit mehreren Coautoren koordinieren muss, raten, sich jemanden wie Stefan zu suchen. Ebenfalls sehr glücklich bin ich darüber, mit Gunnar Landgrebe von der TK Hamburg *den* Spezialisten in Deutschland für die Erweiterungsmöglichkeiten von Rose gefunden zu haben. Er hat die sehr fundierten Kapitel zum Rose REI und zu den Stereotypen geschrieben. Mit viele Liebe zum Detail hat auch Manfred »raezz« Rätzmann von der Rätzmann GmbH in Berlin an seinen Kapiteln zur Objektorientierung, zur UML und zu den Analysetätigkeiten gearbeitet. Ihm danke ich für seine Ideen zur Abrundung des Buches mit den Überblickskapiteln. Mit Jörg Sauer aus Köln habe ich einen Spezialisten für die C++-Generatoren gefunden, der sich richtig 'reingekniet hat, um sein komplexes Thema umfassend zu beschreiben. Ich danke ihm für sein ungeheures Engagement und den unbedingten Willen, dieses wichtige Kapitel korrekt und informativ zu schreiben. Mit Jan Matèrne von der Finanzverwaltung NRW schließlich, als letztem schreibend Beteiligten, hatte ich das besondere Glück, dass er sich freiwillig gemeldet hat, um das Java-Kapitel zu erstellen. Besonders danke ich ihm dafür, dass er es noch zusätzlich übernommen hat, das Fallbeispiel so aufgearbeitet zu haben, dass es »rund« geworden ist. Melanie Schröter von der Firma Freund+Dirks danke ich für ihre Unterstützung bei der Erstellung des speziellen Indexes und beim Korrekturlesen des

Manuskripts. Ebenfalls danke ich meiner Freundin Annette Muß, die sich viel Arbeit mit der sprachlichen »Glättung« der Texte gemacht hat. Schließlich danke ich Volker Kopetzky und Dirk Geuenich von der Firma Rational, die mir viele Informationen zu Rose zur Verfügung gestellt und die die Grundzüge der Kapitel zu Team-Development und Visual Basic geliefert haben.

alf borrmann

Warum sollten Sie dieses Buch lesen?

Ja, warum halten Sie dieses Buch in den Händen? Ich hoffe, Sie versprechen sich Informationen darüber, wie Sie das Tool Rational Rose einsetzen können. Sie werden diese Informationen natürlich nicht gleich in diesem Teil des Buches erhalten, aber Sie sollten wissen, was Sie weiter hinten erwartet.

Uns Autoren ging es vor allem darum, Ihnen eine Hilfestellung in der täglichen Arbeit mit dem CASE-Tool Rational Rose zu geben. Diese Informationen sind vorgesehen für *Projektleiter* und *Softwareentwickler*, die mit Rose arbeiten, aber auch für *Systemanalytiker*, die Ihre Informationen geordnet ablegen wollen. Es beinhaltet Teile, die für *Konfigurationsmanager*, *Supporter* und *Schulungsleiter* interessant sind und gibt Hinweise für *Entscheider*, die den Einsatz von Rose in ihrem Unternehmen oder Projekt erwägen.

Alle Informationen sind *herstellerunabhängig* zusammengetragen worden, sie enthalten keine Marketingaussagen, sondern Informationen, die aus der eigenen Erfahrung und Arbeit der Autoren stammen. Es ist das erste (und bisher einzige) in Deutsch geschriebene Buch zu Rose, das Ihnen eine verständliche und umfassende Information zu allen Teilbereichen dieses mächtigen Produktes gibt. Im Einzelnen haben wir dabei die nachfolgenden Schwerpunkte gelegt.

Legen der Grundlagen

Wir geben Ihnen im ersten Teil eine kurze Einführung in die Objektorientierung an sich und die UML. Diese Kapitel sind dazu vorgesehen, Ihnen bei der Arbeit mit Rose das umständliche Suchen nach Erläuterungen in anderen Büchern zu ersparen. Sie können und wollen die speziellen Titel natürlich nicht ersetzen.

Geben einer Entscheidungshilfe

Das Kapitel zu den Möglichkeiten und Fähigkeiten von Rose gibt Ihnen einen Einblick in die grundlegende Funktionsweise des Tools. Damit erhalten Sie viele Hinweise, wie Sie es in Ihrem Projekt oder Unternehmen nutzen können, was es Ihnen bietet und was es ggf. nicht bietet. Außerdem finden Sie auf der CD eine voll funktionsfähige Version von Rational Rose, mit der Sie alle Funktionen ausprobieren können. Wir haben für Sie außerdem die Modelle für ein komplettes Fallbeispiel auf die CD kopiert, mit deren Hilfe Sie direkt die beschriebenen Arbeitsweisen von Rose testen können.

Hilfe bei der Arbeit

Wenn Sie Rose bereits einsetzen, so finden Sie im mittleren Teil des Buches (Kapitel *3 Der schnellste Weg zum ersten Modell* und *4 Die UML-Konstrukte im Detail*) eingehende Beschreibungen der einzelnen UML-Konstrukte und wie sie in Rose zu nutzen sind. Sie finden Hinweise auf die Zusammenhänge der Modellelemente und wie Sie sie am besten verwenden. Wir haben die Kapitel so gestaltet, dass Sie auch als Nachschlagewerk bei Fragen zur Bedienung geeignet sind. Wir haben uns außerdem bemüht, die kleinen Tricks anzugeben, die wir in unserer täglichen Arbeit mit Rose wie selbstverständlich anwenden. Diese Tricks haben wir für Sie speziell markiert, damit Sie schnell zu Produktivitätsverbesserungen kommen.

Um es aber auch hier noch einmal zu sagen: Wir können Ihnen keine »Kochrezepte« für die zusammenhängende Nutzung der Funktionen von Rose in Ihrem Projekt geben. Dazu sind die einzelnen Projekte zu unterschiedlich und die Funktionen, mit denen Rose Sie unterstützt, zu vielfältig einsetzbar. Sie werden aber einen Überblick darüber bekommen, was die Funktionen leisten.

Wir sind sicher, dass Sie genügend Ideen für deren Einsatz in Ihrem Projekt bekommen.

Tiefgehende Beschreibung aller Funktionen

Alle Funktionen, die Rose bietet, haben wir mit ihren Auswirkungen und Zusammenhängen beschrieben. Sie finden in dem speziellen Index eine Auflistung aller Rose-Funktionen, die über die Oberfläche erreichbar sind, sodass Sie bei Fragen zu einem Bedienelement dieses sofort im Buch finden und sich über dessen Bedeutung informieren können. Wir sind

überzeugt, dass Ihnen das Buch allein schon deswegen zur unverzichtbaren Hilfe im Arbeitsalltag wird.

Daneben halten wir nicht hinter dem Berg, wenn eine Funktion in Rose nicht so tut, wie sie soll oder wenn sie nicht gründlich implementiert ist. So erledigt sich die zeitraubende Frage »Ist es ein Bug oder ist es ein Feature?« und Sie erhalten wichtige Informationen, wie Sie Klippen sicher umschiffen.

Beschreibung der wichtigsten Generatoren

Wie haben die wichtigsten Codegeneratoren komplett behandelt und geben Ihnen sowohl einen Einblick in deren Arbeitsweise und eine Einschätzung, wozu Sie Roundtrip-Engineering einsetzen können als auch genaue Informationen zur effektiven Nutzung der Generatoren.

Tipps zum unternehmensweiten Einsatz

Sie finden fundierte Beschreibungen zu den wichtigen Themen *Team-Development*, *Konfiguration und Erweiterung*. Damit sind Sie in der Lage, Rose in Ihrer speziellen Arbeitsumgebung so anzupassen, dass es Ihnen den größtmöglichen Nutzen bringt.

Finden von Informationen

Wir haben eine spezielle Kombination aus Glossar und Index entwickelt, die Ihnen zu wichtigen Themen eine kurze Erläuterung gibt und die Sie direkt zu den Stellen im Buch führt, wo Sie eine weitere Behandlung des Themas finden oder wo Sie angrenzende Informationen erhalten. In diesem Index zusammengefasst haben wir noch einmal die wichtigsten Funktionen der Kapitel: Sie finden sie unter den Kategorien *»Rose-Funktionen«*, *»C++«*, *»Java«*, *»Visual Basic«*, *»REI«* und *»Skripte«*.

Wir hoffen, Ihnen gefällt der Aufbau dieses – unseres Wissens nach einzigartigen – »Lexikon-Teils«.

Typografie und Sprache

Rose ist nur in Englisch erhältlich. Sollten Sie es jedoch auf einem deutschen Windows-System installieren, so erhalten Sie teilweise Dialoge, in denen die Schaltflächen deutsche Beschriftungen haben. Wir haben in diesem Buch jedoch die englischen Beschriftungen verwendet. Wo wir uns auf Texte beziehen, die Sie z. B. im Menü finden, haben wir diese in

Anführungszeichen gestellt. Die Codebeispiele für Visual Basic, Java oder C++, die wir abgedruckt haben, sind speziell als `Codebeispiel` formatiert.

Bei der Nennung von Tastaturkürzeln haben wir diese im Text ebenfalls in Anführungszeichen gesetzt. Wir haben die Beschriftungen der deutschen Tastatur verwendet und bei Tastenkombinationen, also dort, wo Sie mehrere Tasten gleichzeitig drücken müssen, das »+«-Zeichen dazwischen gesetzt.

Sie finden im Index bzw. Glossar und im Anhang weitere Informationen zur deutschen Bedeutung der englischen Begriffe.

Spezielle Hinweise im Buch sind mit Icons gekennzeichnet:

Beschreibt eine Stelle, an der Sie besonders aufmerksam sein müssen, damit die beschriebene Funktion ihren Zweck erfüllt oder damit Sie nicht versehentlich Schaden anrichten.

Dies sind von uns als Fehler im Programm angesehenes Verhalten. An einzelnen Stellen geben wir Ihnen Hinweise, ob diese Fehler in speziellen Versionen von Rose behoben sind.

Hier laufen Sie Gefahr, Informationen zu verlieren, wenn Sie die beschriebene Funktion anwenden.

Wir haben uns bemüht, Ihnen möglichst viele Hinweise zu Ihrer Arbeit zu geben, die aus der originalen Dokumentation und der Hilfe von Rose nicht so deutlich wird bzw. völlig fehlt. Hier finden Sie auch »Workarounds« zu unvermutetem Verhalten von Rose.

Da dies ein deutsches Buch zur Verwendung von Rose ist, haben wir uns darum bemüht, auch in »gutem« Deutsch zu schreiben. Wir haben uns bei der Übersetzung von Begriffen an den Vorgaben des System-Bauhaus orientiert (s. a. *9.2 UML-Übersetzungstabelle*). Allerdings sind inzwischen viele englische Begriffe der Softwareentwicklung einfach in die deutsche Fachsprache übernommen worden. So wird in der Regel im Deutschen sowohl von »Use-Case« als auch von »Anwendungsfall« gesprochen. Insofern haben wir uns erlaubt, im Sinne einer besseren Lesbarkeit auch die eingedeutschten Begriffe zu verwenden und uns nicht krampfhaft an deutsche Worte zu halten.

1 Die Welt in Objekten

Die Anwendung der Unified Modeling Language (UML) als Sprache zur Modellierung in objektorientierten Projekten hat sich allgemein durchgesetzt. Diese Sprache ist jedoch ohne die Nutzung von CASE-Tools nicht sinnvoll einsetzbar. Im diesem Umfeld ist Rose eines der Tools, die eine längere Entwicklungszeit vorweisen können. Es bietet eine fast schon überwältigende Vielfalt an Funktionen, um den Entwickler bei der Anwendung der UML zu unterstützen. Dieses Buch gibt Ihnen einen Überblick – aber auch detaillierte Hilfestellungen – um die UML und Rational Rose anzuwenden. Es hilft Ihnen so, dieses mächtige Werkzeug in Ihren Projekten sinnvoll einzusetzen und sie so zum Erfolg zu führen.

1.1 Warum Objektorientierung?

Muss man den Leser eines Praxishandbuchs zu Rational Rose zunächst von den Vorzügen der Objektorientierung überzeugen? Die meisten von Ihnen wahrscheinlich nicht. Wenn hier zu Beginn dieses Buchs trotzdem von den Vorzügen der Objektorientierung die Rede ist, soll auch das ganz im Sinne eines Berichts aus der Praxis verstanden werden. Um nicht ausschließlich aus eigenen Erfahrungen schöpfen zu müssen, habe ich eine kleine Umfrage unter befreundeten Entwicklern durchgeführt. Gefragt war nach den eigenen, persönlichen Gründen für die Entscheidung, objektorientiert zu programmieren. Die Ergebnisse sind natürlich völlig unrepräsentativ und alle daraus abgeleiteten Gedanken nach wie vor rein subjektiv. Vielleicht erkennt der ein oder andere von Ihnen seine eigenen Erfahrungen aber darin wieder.

1.1.1 Ein Paradigma kommt in die Jahre

Als die Objektorientierung Mitte der 90er Jahre so richtig populär wurde, hatte die Idee bereits knapp 30 Jahre hinter sich. Diverse objektorientierte Sprachen waren entstanden, seit Beginn der 90er erschienen verstärkt Fachbücher zu objektorientierter Analyse und Design. Autoren wie Grady Booch und James Rumbaugh beschrieben in ihren Büchern komplette Methoden des objektorientierten Designs (vgl. [Booch 94] und [Rumbaugh 91]), in Fachzeitschriften wurden die Vorzüge und eventuellen Nachteile objektorientierter Programmierung heftig debattiert

(Performance!), es gab viel zu lesen, aber wenig konkrete Erfahrungen. Das änderte sich, als nach und nach alle Hersteller von Entwicklungsumgebungen auf den Zug aufsprangen und neue, zumindest teilweise objektorientierte Versionen ihrer Programme herausbrachten. Nun konnte man auch die Brot-und-Butter-Projekte des Programmieralltags objektorientiert angehen, ohne allzu große Risiken einzugehen. Mit den ersten praktischen Erfahrungen kam die Euphorie. Nüchterne Menschen kamen ins Schwärmen wenn sie von den Möglichkeiten der Vererbung berichteten, von der Aufteilung komplexer Aufgaben auf spezialisierte Klassen, von der Zusammenarbeit unabhängiger Objekte zur Laufzeit. Die Rede war von einem völlig neuen Programmierparadigma, das alle unsere bisherigen Erfahrungen und Denkgewohnheiten über den Haufen schmeißen würde.

Die Euphorie ist abgeklungen, der Katzenjammer aber ist ausgeblieben. Das ehemals neue Programmierparadigma »Objektorientierung« hat sich bewährt und ist inzwischen in vielen Bereichen zu einer Selbstverständlichkeit geworden. Die Objektorientierung ist in ein Arbeitsstadium übergegangen. Debattiert wird nicht mehr das Ob, sondern nur noch das Wie.

Bei der Erstellung einer grafischen Oberfläche kommt man heute häufig ohne Objekte gar nicht mehr aus, sind doch die von der Entwicklungsumgebung angebotenen Oberflächenelemente wie Fenster, Schaltflächen, Rollbalken, Optionsgruppen und so weiter lauter Objekte, deren Eigenschaften eingestellt werden müssen und die auf Ereignisse (Auslöser) reagieren, indem sie die zugeordneten Methoden abarbeiten. Leider sind aber auch viele Entwickler auf diesem Stand der »Objektorientiertheit« stehen geblieben. Das ist schade, denn gerade das objektorientierte Design der unter der Oberfläche liegenden Arbeitsschicht bringt den größten Nutzen bei der Bewältigung komplexer Programmieraufgaben. Wenn Sie dieses Buch lesen, weil Sie mehr wollen als Eigenschaften setzen und Klick-Ereignisse ausfüllen, dann herzlich willkommen! Wenn Sie noch nicht so richtig wissen, ob, oder ob nicht, lassen Sie sich von der folgenden Zusammenfassung der Antworten zu meiner kleinen Umfrage dazu verleiten, Ihre Programme in Zukunft nicht nur an der Oberfläche, sondern auch im Innern objektorientiert aufzubauen.

Eine umfassende Einführung in die Objektorientierte Softwareentwicklung können wir in diesem Buch allerdings nicht bieten. Wir wollen uns ja schließlich mit Rational Rose befassen – und das nicht zu knapp. Im Anhang dieses Buches finden Sie jedoch ausführliche Literaturhinweise

zur Objektorientierung, zur UML und zu spezifischeren Themen wie Projektmanagement, Designmustern und vielem mehr.

1.1.2 Objektorientierung ist einfach

»But first: Are you expirienced?« (J. Hendrix)

Die Grundkonzepte der objektorientierten Softwareentwicklung sind einfach. Wirklich! Damit ist nicht gemeint, dass die Tätigkeit eines objektorientierten Analytikers oder eines Designers einfach wäre. Ganz im Gegenteil. Diese Tätigkeiten erfordern ein hohes Abstraktionsvermögen, Kenntnisse im Fachgebiet der Anwendung, Ordnungssinn, Kreativität und viel Erfahrung. Also lauter Dinge, die Ihnen dieses Buch nicht vermitteln kann. Die grundlegenden Bausteine, mit denen der Systemanalytiker und Designer arbeitet, sind in Zahl und Inhalt jedoch überschaubar.

Versuchen wir uns also zumindest an einer stichwortartigen Darstellung der Grundkonzepte objektorientierten Denkens. Damit möchte ich auch einige Begriffe einführen, die anschließend bei der Arbeit mit Rational Rose des Öfteren verwendet werden. Die folgenden Erläuterungen stammen zugegebenermaßen eher aus der Welt der Programmierer. Wer objektorientierte Geschäftsmodelle entwickelt wird das ein oder andere sicher anders sehen.

Objekte

Objekte (auch als »Instanz« bezeichnet) bestehen aus Daten und Prozeduren, die auf diesen Daten arbeiten. Die Daten (engl. *data*) eines Objektes bezeichnet man auch als *Attribute*, die Prozeduren als *Operationen*. Die konkreten Werte aller Attribute eines Objektes ergeben zusammen den aktuellen *Status* oder *Zustand* des Objektes. Wenn der Status eines Objektes auch das Beenden des Programms übersteht, bezeichnet man das Objekt als *persistent*, ansonsten als *transient* bzw. als kurzlebiges Objekt. Außerdem wird zwischen *aktiven* und *passiven* Objekten unterschieden. Objekte, die zwar vorhanden sind (zum Beispiel gespeichert in der Datenbank), aber zur Laufzeit des Programms gerade nicht aktiv sind, sind eben passiv.

Objekte haben per se eine Identität (existenz-basierte Identität). Diese ist unabhängig vom Status des Objektes. Das Objekt *KundeMeier* bleibt mit sich selbst identisch, auch wenn der Kunde Meier seinen Namen ändert und jetzt Müller heißt. Das nächste Mal, wenn Sie diesem Objekt begegnen, werden Sie es wahrscheinlich *KundeMüller* nennen. Es ist aber immer noch das gleiche Objekt.

Kapselung, Eigenschaften, Methoden

Attribute und Operationen können nach außen hin verborgen werden. Die veröffentlichten Attribute und Operationen können von anderen Objekten genutzt werden. Die veröffentlichten Attribute werden häufig als *Eigenschaften* des Objekts bezeichnet, die veröffentlichten Operationen als *Methoden*. Beide zusammen bilden die *Schnittstelle* des Objektes.

Assoziation

Objekte können miteinander verbunden werden. Die Verbindung bedeutet, dass das eine Objekt das andere kennt und also dessen Eigenschaften und Methoden benutzen kann. Verbindungen zwischen (zwei) Objekten werden als *binäre Assoziationen* (engl. *binary assiciation*) bezeichnet. Wenn Objekt A eine Methode von Objekt B aufruft, sagt man auch, dass A eine *Nachricht* an B sendet. Objekt A wird in dem Zusammenhang als »Sender«, Objekt B als »Empfänger« bezeichnet. Assoziationen heißen *bidirektional*, wenn beide Objekte sich wechselseitig kennen, sie heißen *unidirektional* wenn nur das eine Objekt das andere kennt.

Aggregation, Komposition

Objekte können ineinander verschachtelt werden. Ein Objekt, das andere Objekte enthält, bezeichnet man als *Container* oder als *Sammlung* (engl. *collection*), die Tatsache des Enthalten-Seins als *Aggregation* oder *Komposition* (engl. *composition*). Wenn das enthaltene Objekt nur innerhalb des Containers existieren kann, alleine also keinen Sinn macht, liegt eine Komposition vor. Wenn das enthaltene Teil auch außerhalb des Containers existieren kann, liegt eine Aggregation vor. Die Materialien in einer Stückliste bilden also eine Aggregation, die Positionen in einem Beleg eine Komposition.

Klassen

Objekte gleichen Typs bilden eine *Klasse* (engl. *class*). In der objektorientierten Programmierung wird nicht der Aufbau eines einzelnen Objektes im Sourcecode beschrieben, sondern immer der Aufbau einer Klasse. Zur Laufzeit werden dann, basierend auf der Klassenbeschreibung, die Objekte erstellt. Das geschieht entweder durch einen an das Laufzeitsystem gerichteten Programmbefehl, zum Beispiel `CreateObject(<Klassen Name>)`, oder durch Aufruf einer speziellen Klassenmethode, der Konstruktormethode, wie in dem Beispiel `meinObjekt := meineKlasse.New()`. Dabei ist »`New()`« die Konstruktormethode der Klasse »`meineKlasse`«.

Klassen, aus denen zur Laufzeit Objekte erstellt werden sollen, bezeichnet man als *konkrete* Klassen (engl. *concrete class*), solche aus denen keine Objekte erstellt werden sollen, als *abstrakt*. Eine abstrakte Klasse ist immer als die gemeinsame Elternklasse mehrerer konkreter Klassen vorgesehen (siehe Vererbung).

Vererbung

Klassen lassen sich auseinander ableiten. Abgeleitete Klassen *erben* alle Eigenschaften und Methoden ihrer Elternklasse. Interne Attribute und Operationen können von der Elternklasse wahlweise vererbt werden. Abgeleitete Klassen können neue, eigene Attribute und Operationen den ererbten hinzufügen. Diese stehen dann nur in der abgeleiteten Klasse, nicht in der Elternklasse zur Verfügung. Ererbte Attribute und Operationen können von abgeleiteten Klassen nicht verworfen werden. In einigen Programmiersprachen können sie jedoch verborgen werden, sodass verbundene Objekte diese Attribute oder Operationen nicht mehr benutzen können. Ererbte Operationen können von einer abgeleiteten Klasse überschrieben werden, wenn die Elternklasse dies zulässt.

Abstraktion

Abstraktion ist das Prinzip, die Aspekte eines Sachverhalts zu ignorieren, die für den aktuellen Zweck nicht von Bedeutung sind. Damit kann man sich auf die wichtigsten Aspekte konzentrieren.

Der Zweck dabei ist, ein Verständnis für die anstehende Aufgabe zu erlangen. Um dieses zu erreichen, können nun die Details eines »Dings« (z. B. eines Objektes) zunächst ignoriert werden. Es genügt im ersten Ansatz ggf. bereits, ein Ding (engl. *thing*) nur zu benennen, um dessen Rolle im Informationssystem klar zu machen. Ein Objekt in der objektorientierten Arbeit kann zunächst als Abstraktion eines Dings innerhalb der zu lösenden Aufgabenstellung verstanden werden.Polymorphie

Die Namen der Attribute müssen nur jeweils innerhalb einer Klasse eindeutig sein. Operationen einer Klasse dürfen den selben Namen haben, wenn sie sich in Art und/oder Anzahl der zu übergebenen Parameter unterscheiden. Verschiedene Klassen können unter demselben Methodennamen spezifisch implementierte Operationen anbieten.

Die Möglichkeit, unter dem gleichen Namen Verschiedenes zu tun, bezeichnet man als *Polymorphie*.

Interfaces

Eine Gruppe von Methodennamen kann separat als gemeinsame Schnittstelle unterschiedlicher Klassen beschrieben werden. Eine solche Gruppe von Methodennamen bezeichnet man als *Interface* (deutsch: Schnittstelle).

Alles klar?

Aus diesen wenigen und einfachen Konzepten lassen sich beliebig komplexe Systeme zusammenbauen. Das ist die gute Nachricht. Die schlechte lautet: In diesen objektorientierten Systemen können Sie auch beliebig schnell den Überblick verlieren. Böse Zungen behaupten sogar: gerade dort! Denn objektorientierte Systeme gleichen weniger einem Fließband, auf dem Ihre Daten von einer Prozedur zur nächsten weiter gereicht wer-

den, bis sie fertig sind. Ein objektorientiertes System hat viel mehr Ähnlichkeit mit einem Netzwerk, in dem Alles mit Allem verknüpft sein kann. So wie im wirklichen Leben halt.

Die Kunst besteht also zum großen Teil darin, den Überblick zu behalten. Das ist mithilfe von einfachen Diagrammen eher machbar als ohne. Die Betonung liegt dabei auf »einfache« Diagramme! Zur Erstellung dieser Diagramme hat sich mit der UML inzwischen eine einheitliche Beschreibungssprache durchgesetzt. Mehr über die UML finden Sie weiter unten im Kapitel *2.2.1 Die Modellierungssprache von Rational Rose*.

1.1.3 Objektorientierung macht Spaß

»We work for fun and money«

Dass Programmieren Spaß macht, dürfte den meisten Lesern dieses Buches bekannt sein. Objektorientiert zu programmieren erhöht den Spaß an der Sache. Das hat mehrere Gründe. Zum einen sind Programmierer faul (wobei ich natürlich von mir auf andere schließe). Das Ziel der Wiederverwendbarkeit von Objekten kommt der Programmiererdenke also sehr entgegen. Nichts ist langweiliger, als mehrmals hintereinander das Gleiche zu tun. Im »normalen« Programmiererhirn triggert ein doppelt eingetippter Sourcecode sofort den Wunsch nach einem Unterprogramm, das das Thema ein für alle Mal erledigt. Die Erfüllung dieses Wunsches ist in der objektorientierten Entwicklung zum Prinzip geworden. Aber auch die anderen Prinzipien der Objektorientierung sind so, wie wir als Entwickler uns das Leben vorstellen.

Erben oder Delegieren?

Die mit der objektorientierten Programmierung eingeführte Möglichkeit, Funktionalität zu vererben, kommt, wie gesagt, der Faulheit des normalen Softwareentwicklers sehr entgegen. Hier hat er oder sie erstmals die Möglichkeit, eine Änderung an genau einer Stelle durchzuführen – in der Elternklasse eben – und schon erben alle daraus abgeleiteten Klassen die neue oder geänderte Funktionalität. Vorbei sind die Tage, in denen eine gar nicht so schwierige Verbesserung im Programm nur deshalb unterlassen wurde, weil diese Änderung an mehreren Stellen durchzuführen war. Und wer macht schon gerne mehrfach das selbe (siehe oben). Bei etwas Glück (oder passendem Design) wird heute die Elternklasse geändert und die Sache ist erledigt. Das macht die Vererbung zum bevorzugten Designmittel eines jeden Neulings in der Objektorientierung. Im Laufe der Zeit merkt er oder sie allerdings, dass sich durch intensive Nutzung der Vererbungsmechanismen neue Probleme – Designprobleme diesmal – erge-

ben. Aber auch hier bietet die Objektorientierung mit der Delegation eine Lösungsmöglichkeit.

Delegation bedeutet, Teilaufgaben kleinen Maschinen zu übertragen, die darauf spezialisiert sind – anderen Objekten also. Wenn ein solches Maschinchen einmal funktioniert, braucht sich der Entwickler darüber keine Gedanken mehr zu machen. Wo auch immer diese spezielle Aufgabe anfällt, wird jetzt die kleine Spezialmaschine eingesetzt. Dass diese Vorgehensweise in vielen Fällen eine Alternative zur Vererbung sein kann, leuchtet schnell ein. Eine Funktionalität wird nicht im Vererbungsbaum herunter gereicht, sondern in ein gesondertes Objekt ausgelagert. Vor allem bei Funktionen, die wenig miteinander zu tun haben und für die man eigentlich zwei Vererbungslinien gleichzeitig bräuchte, kann man durch Delegation und Einfachvererbung ein sauberes, unproblematisches Design erreichen und auf Mehrfachvererbung verzichten.

Polymorphie nutzen

Polymorphie, übersetzbar mit »Vielgestaltigkeit« ist eines der wichtigsten Prinzipien der Objektorientierung. Ganz grob bedeutet sie, dass sich hinter einem bestimmten Methodennamen ganz unterschiedliche Algorithmen verbergen können. Wozu braucht man das? Stellen Sie sich vor, in Ihrer Anwendung gibt es ein Druckerobjekt, das dafür zuständig ist, aufbereitete Druckseiten an den physischen Drucker weiter zu geben, dessen Status zu überwachen, mit dem Anwender wegen neuem Papier zu verhandeln etc. Über dieses Druckerobjekt wollen Sie nun die unterschiedlichsten Dokumente und Listen ausdrucken. Sie schicken also alles, was gedruckt werden soll, an dieses Druckerobjekt – Rechnungen, Lieferscheine, Berichte, Grafiken usw. – lauter Objekte, die eines gemeinsam haben, nämlich eine Methode, die zum Beispiel `erstelleDruckseiten` heißt. Das Druckerobjekt ruft diese Methode auf und bekommt die aufbereiteten Druckseiten zurück geliefert, die es an den Drucker weiter leitet. Was sich hinter `erstelleDruckseiten` verbirgt, ist bei jedem Objekttyp etwas anderes. Die Methode `erstelleDruckseiten` kann also viele unterschiedliche Gestalten annehmen, je nachdem, welcher Typ von Objekt sie anbietet. Dieses Verhalten nennt man Polymorphie.

Polymorphie hilft unter anderem, Verantwortlichkeiten im Programm geeignet zu verteilen und dadurch seitenlange IF – oder CASE Statements zu vermeiden. Das Druckerobjekt von vorhin muss also gar nicht wissen, welche Art von Objekt ihm zum Druck übergeben wurde. Anstatt die Aufbereitung der Druckseiten pro Objekttyp in einem gesonderten CASE abzuhandeln, wird die Verantwortlichkeit dafür dem zu druckenden Objekt übertragen.

Interna kapseln

Programmierer sind eigenwillig. Wie sie eine Aufgabe lösen entscheiden sie gerne selber. Auch hierbei kommt die objektorientierte Entwicklung dem Programmierer mit dem Prinzip der Kapselung sehr entgegen. Was zählt, ist die korrekte Implementierung der Schnittstelle eines Objektes. Der interne Aufbau soll den Benutzern des Objektes dagegen verborgen bleiben. Hier kann der Entwickler seine ganze Kreativität entfalten. Auch dem Wunsch, einen Algorithmus umzustrukturieren und zu optimieren steht – außer Zeit- und Budgetbeschränkungen – nichts im Wege, solange die Schnittstelle nach außen konstant bleibt. Dieses Prinzip der Vorläufigkeit und die Lust am Ändern wurde als »Refactoring« im eXtreme Programming gar zum bevorzugten Arbeitsstil erhoben.

Faszination Modell

Die Entwicklung eines Programms von der ersten Analyse der Aufgabenstellung bis zum fertigen, voll funktionsfähigen Produkt, ist eine faszinierende Sache. Wie andere Leute auch, sind Softwareentwickler häufig begeistert von dem, was sie geschaffen haben. Wenn das eigene »Baby« die gestellte Aufgabe perfekt und obendrein elegant löst, ist der Erbauer mit sich zufrieden und das Leben ist gut.

Diese Begeisterung erfährt der objektorientierte Entwickler nicht erst am Ende eines langen Entwicklungszeitraums, sondern bereits mittendrin, wenn ihr etwa ein elegantes Designstück gelungen ist oder wenn sie eine neue Klasse entwickelt hat, die perfekt und nützlich ihren Dienst versieht.

Mit der objektorientierten Softwareentwicklung ist eine weitere Tätigkeit für viele Leute zu einem Faszinosum geworden. Die Rede ist vom Modellieren. Modelle und Diagramme gab es auch schon vor der Objektorientierung; ihnen haftete aber häufig der Geruch einer Pflichtübung an. Mit dem objektorientierten Blick auf die Welt scheinen wir der Realität ein ganzes Stück näher gerückt zu sein, sodass ein Modell heute viel eher als »Welt im Kleinen« empfunden wird. Die davon ausgehende Faszination kennt jeder Hobby-Modellbauer und jeder, der als Kind einmal mit einer elektrischen Eisenbahn oder einer Puppenstube gespielt hat. Unter den Vorläufern der objektorientierten Modellierung war die Entity-Relationship-Modellierung vielleicht deshalb so erfolgreich, weil sie dieser »Welt im Kleinen« nahe kam.

Roundtrip-Engineering

Was dem Modellbauer und Softwareentwickler keinen Spaß macht, ist doppelte Arbeit. Deswegen wird in der Praxis immer dort auf das Modellieren verzichtet, wo der automatische Übergang vom Modell zum Sourcecode und umgekehrt fehlt. Wenn alles das, was als Modell erdacht und erschaffen wurde, zur Realisierung im Sourcecode erneut eingetippt werden muss oder wenn umgekehrt jede Strukturänderung des Codes im

Modell nachvollzogen werden muss um beide in Übereinstimmung zu halten, hört der Spaß bald auf. Die Hersteller der Modellierungswerkzeuge reagieren darauf mit der Möglichkeit des Roundtrip-Engineering. Aus dem Modell kann zumindest die Klassenstruktur als Code automatisch generiert werden und umgekehrt kann das Modell als Dokumentation der Strukturen im Code automatisch generiert werden. Dies funktioniert jedoch mehr schlecht als recht. Wie im Teil *7 Generatoren und Reverse-Engineering* noch zu besprechen sein wird, ist der Weg vom Sourcecode zurück zum Modell mit den vorhandenen Tools nur sehr mühselig begehbar.

Im Grunde handelt es sich dabei um eine Notlösung. Gefragt sind eigentlich die Hersteller der Entwicklungsumgebungen. Jede moderne Entwicklungsumgebung verfügt heute über einen eigenen GUI-Builder. Warum ist es dann nicht möglich, die benötigten Klassen in der Entwicklungsumgebung selbst in Form von Klassendiagrammen anzulegen? Warum kann eine Entwicklungsumgebung Aufrufsequenzen ausgewählter Objekte nicht gleich als UML-Sequenzdiagramme darstellen?

1.1.4 Objektorientierung hilft

»Divide et impera!«

Viele Entwickler haben das Gefühl, eine gestellte Aufgabe durch eine objektorientierte Zerlegung besser in den Griff zu bekommen. Woran liegt das? Wahrscheinlich, weil hier das Wort »Zerlegung« erst richtig Sinn macht. Objektorientierte Verfahren bewältigen Komplexität immer durch Aufteilung. Der Entwickler sieht sich also sehr bald schon nicht mehr einer riesengroßen, undurchschaubaren Sache gegenüber, sondern einer Vielzahl von kleinen, überschaubaren Einheiten.

Diese kleinen Einheiten können separat entwickelt und – genau so wichtig – separat getestet werden. Angepasste Testverfahren für das Unit-Testing orientieren sich vor allem an der Schnittstelle des Objekts. Speziell zum Unit-Testing von objektorientierten Programmen wurden Testframeworks entwickelt. Das sind Klassenbibliotheken zur Erstellung von automatischen Testläufen. Testfälle und Testsuites werden aus vorbereiteten Klassen abgeleitet. Ein TestRunner-Objekt führt die Testfälle aus. Das von Kent Beck und Erich Gamma entwickelte Testframework JUnit für Java Unit Tests ist von vielen Anderen für weitere Programmiersprachen adaptiert worden. Eine aktuelle Liste von frei verfügbaren Testframeworks finden Sie unter http://www.xprogramming.com/software.htm

In den Fachzeitschriften und Fachbüchern hat sich inzwischen eine große Menge so genannter »Best Practices« angesammelt. Erprobte Vorgehensweisen also, die dem Entwickler dabei helfen, objektorientierte Konzepte anzuwenden, sich in neue Aufgabengebiete einzuarbeiten oder bislang ungeübte Techniken zu erlernen (s. a. [Beck 99], [Kruchten 99]).

1.1.5 Das macht man heute halt so

»It's cOOl, man!«

Objektorientierte Softwareentwicklung ist heute »State of the art«, alle modernen Entwicklungsumgebungen und Sprachen sind objektorientiert aufgebaut. Zu jedem Aspekt der objektorientierten Softwareentwicklung gibt es Fachliteratur, Diskussionsforen im Internet, Seminare und Workshops, sowie die unterschiedlichsten Meinungen. Objektorientierung ist also aus dem Stadium des Hypes längst heraus und als allgemeine Arbeitsgrundlage der Softwareentwicklung akzeptiert.

Natürlich wird die Entwicklung nicht bei der Objektorientierung stehen bleiben. Dass aber die Technik von objektorientierter Analyse, Design und Programmierung demnächst wieder von einem völlig neuen Paradigma abgelöst werden – wie es den strukturierten Methoden passiert ist – ist ziemlich unwahrscheinlich. Anders als die strukturierten Methoden können objektorientierte Methoden ohne Bruch im gesamten Entwicklungszyklus angewandt werden. Und anders als abstrakte Strukturen sind die aus dem Design hervorgehenden Objekte etwas Konkretes, das der Entwickler anfassen und weiter verwenden kann.

Als Beleg für diese These könnte der aktuelle Trend im Bereich Java angesehen werden: hier wird unter dem Begriff »Enterprise JavaBeans« eine stärker *komponentenorientierte* Entwicklung angestrebt. Diese Komponenten selbst werden aber nach den Prinzipien der Objektorientierung entwickelt.

1.2 UML

1.2.1 Die Modellierungssprache von Rational Rose

Objektorientierte Softwareentwicklung ist weniger eine Art zu programmieren als vielmehr eine Analyse und Designmethodik. Aus einem prozedural denkenden Entwickler wird kein objektorientierter Entwickler nur dadurch, dass er seine Funktionen mit einer Deklaration à la `Define class` ... und `Enddefine` umkleidet. Daher ist auch der Glaubenskrieg zwischen

Anhängern rein objektorientierter Sprachen auf der einen Seite und Benutzern so genannter Hybrid-Sprachen auf der anderen Seite eher uninteressant. Bei der Frage, ob eine Software sich mit Fug und Recht objektorientiert nennen darf, kommt es letztlich nicht auf die Programmiersprache, sondern auf das Design der Anwendung an.

In diesem Kapitel beschreiben wir kurz die Diagrammtypen der Unified Modeling Language (UML), die als Modellierungssprache in Rational Rose verwendet wird. Zur UML gibt es inzwischen unendlich viel Literatur, deshalb hier nur der Hinweis auf zwei Bücher, die in dieses Kapitel eingeflossen sind. Zum einen das »UML Toolkit« von Hans-Erik Eriksson und Magnus Penker [Erik Penker 98], zum anderen das deutsche Standardwerk zu Analyse und Design mit der UML von Bernd Oestereich [Oestereich 98].

»Design first!«

»Design first«, diese Forderung haben sich die Verfechter der objektorientierten Softwareentwicklung schon sehr früh auf die Fahnen geschrieben. Deshalb entstanden neben objektorientierten Programmiersprachen sehr bald Designmethoden und Notationsweisen wie die Booch-Methode, OMT von James Rumbaugh, OOSE von Ivar Jacobson, OOSA von Shlaer/Mellor, OOA von Coad/Yourdon und einige andere (vgl. [Booch 94], [CoadYourdon 91A], [CoadYourdon 91D], [Jacobson 92], [Rumbaugh 91] und [ShlaerMellor 88]). Um einen unsinnigen Methodenstreit zu vermeiden, haben sich die als »drei Amigos« bezeichneten Autoren Grady Booch, James Rumbaugh und Ivar Jacobson 1995 unter dem Dach der Firma Rational zusammengetan, um eine »Unified Method« (UM) zu entwickeln. Dieses ehrgeizige Vorhaben wurde bald darauf auf die Entwicklung der »Unified Modeling Language« (UML) eingeschränkt. Das zu einer vollständigen Methode gehörende umfassende Vorgehensmodell für alle Phasen des Softwarelebenszyklus' wurde zunächst ausgeklammert und 1998 als »Rational Unified Process« (RUP) nachgeliefert. Geschadet hat dies der Verbreitung der UML ganz und gar nicht. Ganz im Gegenteil. Die UML hat sich inzwischen auf breiter Front als Notationsstandard für objektorientierte Modelle durchgesetzt.

Iterativ vorgehen

»Design first« bedeutet nicht, dass vor Beginn der Codierung das gesamte zu entwickelnde Programm in Form von UML-Diagrammen vorliegen soll. Alle objektorientierten Vorgehensmodelle sind iterativ (auch als »evolutionäres oder inkrementelles Vorgehen« bezeichnet) ausgelegt. Dabei wird nur die nächste Iteration detailliert geplant und nur die in der nächsten Iteration zu entwickelnden Programmteile werden detailliert designed. Auch die Vorstellung, alle Programmteile der nächsten Iteration zunächst als UML-Diagramme darstellen zu müssen, stammt wohl eher

vom alten Wasserfall-Denken. Das Modell kann und soll nicht als Schaltplan gedacht sein, der dann in der Codierung nur noch realisiert werden muss. »Design first« kann nur heißen, dass man sich vor der Codierung Gedanken zur Struktur der zu entwickelnden Klassen, zu notwendigen Abläufen und Aufrufsequenzen, zum Schnittstellendesign und Ähnlichem machen sollte. Das Ergebnis dieser Überlegungen hält man dann in Textform und Diagrammen fest. Ob dabei der Text das Diagramm erläutert oder das Diagramm den Text veranschaulicht ist wohl eher eine Frage der persönlichen Sicht der Dinge als ein ernsthaft zu diskutierendes Thema.

Modellieren ist kein Selbstzweck

Eine wichtige Frage ist, wie umfassend und genau die Designüberlegungen sein sollten. Hier kann es nur eine richtige Antwort geben: So umfassend und genau wie nötig! Wie viel nötig ist, hängt wesentlich von der Größe des Entwicklerteams ab. Manchem Einzelkämpfer reicht schon die kurze Skizze einer Idee, bevor er diese in die Tat umsetzt, sprich codiert, wenn er nicht sogar vollständig auf ein Modell verzichtet und seine Klassen gleich in der Entwicklungsumgebung anlegt. Dagegen ist nur einzuwenden, dass man seinem eigenen Programm nach einer gewissen Zeit selbst als Fremder gegenüber steht. Und der Sourcecode ist denkbar schlecht geeignet, einen Überblick zu gewinnen bzw. zu behalten.

Andere nutzen das Nachdenken am Modell schon intensiver, gehen dabei jedoch in sehr kurzen Iterationen vor. »Design ein bisschen, codier' ein bisschen, teste ein bisschen«, diese Vorgehensweise eignet sich vor allem für kleinere Teams.

Die Architektin einer größeren Anwendung wird einige kritische Strukturen und Abläufe in ausgefeilten Klassen- bzw. Sequenzdiagrammen darstellen und dem Entwicklerteam Vorgaben zum Aufbau von Schnittstellen etc. machen wollen. Dabei wird sie sich aber zunächst auf die grundlegende Architektur der zu erstellenden Anwendung beschränken. Das detaillierte Design wird dann erstellt, wenn die jeweilige Programmfunktion realisiert werden soll. So vermeidet sie einen hohen Designaufwand für Funktionen, die erst in späteren Iterationen realisiert werden sollen.

Ein Team, das über einen längeren Zeitraum mit der Entwicklung einer Softwaresystemfamilie beschäftigt ist, tut gut daran, das aus der Domainanalyse entstandene Basisdesign der Anwendung vorab umfassend und genau zu dokumentieren, alleine schon, um neuen Teammitgliedern den Einstieg zu erleichtern. Häufig wird zur Entwicklung einer Softwaresystemfamilie ein darauf ausgelegtes Framework erstellt oder zugekauft. Auch ein solches Framework muss natürlich sehr umfassend und genau dokumentiert sein.

Für alle diese Herangehensweisen und Ziele ist die UML bestens geeignet. Dies liegt daran, dass die UML zum einen den für ausführliche und genaue Modelle notwendigen Detailreichtum aufweist, zum anderen aber auch mit wenigen einfachen Mitteln verständliche und häufig ausreichende Diagramme ermöglicht. Dabei gilt, dass jedes Diagramm um so verständlicher ist, je mehr es sich auf das Wesentliche beschränkt. Die UML zwingt Sie als Modellierer nicht dazu, jedes kleinste Detail zu spezifizieren. Exaktheit und Verständlichkeit sind die konkurrierenden Gewichte an der Waage, die jeder Modellierer für sich austarieren muss. Zwischen dem skizzenhaften »Nachdenken am Modell« und dem ausgearbeiteten Konstruktionsplan gibt es viele Stufen. Welche für die jeweilige Aufgabe sinnvoll ist, muss jeder Modellierer selbst entscheiden.

1.2.2 Diagrammtypen der UML

Die UML 1.4 kennt folgende Diagrammtypen (vgl. [UML 14]):

- Anwendungsfalldiagramm (engl. *use case diagram*)
  Ein Anwendungsfalldiagramm zeigt Akteure und Anwendungsfälle, also wer das Informationssystem benutzt und was er, sie oder es damit tut. Akteure können Personen oder andere, externe Systeme sein, sie repräsentieren damit immer die externe Sicht auf das System. Ein einfaches Rechteck symbolisiert die Systemgrenzen. Akteure können »rechts und links« vom System angesiedelt werden, um zum Beispiel zwischen firmeninternen und -externen Personen zu differenzieren. Zwischen Akteuren ist auch eine Vererbungshierarchie möglich.

  Die Anwendungsfälle geben alle Arten der Systembenutzung wieder. Anwendungsfälle werden textlich beschrieben. Zu einem Anwendungsfalldiagramm wird es also immer mehrere Dokumente geben, die die dargestellten Anwendungsfälle beschreiben.

  Die Erstellung von Anwendungsfalldiagrammen mit Rational Rose wird im Kapitel *3.3.3 Anwendungsfalldiagramme erstellen* ausführlich beschrieben.

- Aktivitätsdiagramm (engl. *activity diagram*)
  Aktivitätsdiagramme beschreiben Abläufe im System oder bei dessen Benutzung an Hand von Aktivitäten. Häufig werden Aktivitätsdiagramme dazu eingesetzt, Anwendungsfälle detaillierter darzustellen.

  Aktivitätsdiagramme ähneln den prozeduralen Flussdiagrammen. Sie können in Verantwortlichkeitsbereiche aufgeteilt werden (sog. *swimlanes*). Ausgehend von einem Anfangszustand des Systems werden die Aktivitäten durchlaufen und führen zu einem Endzustand. Dabei ist es

möglich, parallele Abläufe darzustellen. Aktivitäten können auch mehrere Ausgänge haben, die von Bedingungen abhängen. Darüber hinaus kann man Bedingungen auch unabhängig von einer Aktivität für eine Verzweigung des Ablaufs angeben.

Zusätzlich können Aktivitäten Objektzustände hervorrufen, die wiederum bei einer nächsten Aktivität vorausgesetzt werden können.

Mit der Erstellung von Aktivitätsdiagrammen in Rational Rose beschäftigen wir uns detailliert in den Kapiteln *3.3.7 Anwendungsfälle detaillieren* und *3.4.2 Abläufe detaillieren: Aktivitätsdiagramm*.

- Klassendiagramm (engl. *class diagram* oder *static structure diagram*)
  Ein Klassendiagramm stellt die im System vorkommenden Klassen und deren Beziehung zueinander dar. Das ist die wohl am häufigsten verwendete Diagrammform der UML. Die Beziehungen zwischen den Klassen können Vererbung (*generalization*), Benutzung (*association*), Enthalten-sein (*composition*, *containment*, *aggregation*) oder Abhängigkeit (*dependency*) beschreiben. Jeder Beziehung kann eine Kardinalität mitgegeben werden, um die minimale oder maximale Anzahl von Objekten anzugeben, die an einer Beziehung beteiligt sein können.

  Klassendiagramme sind statische Diagramme. Anders als Aktivitätsdiagramme, Sequenzdiagramme oder Kollaborationsdiagramme enthalten sie keine Aussage darüber, zu welcher Zeit Objekte erstellt werden, einander aufrufen, wieder freigegeben werden usw.

  Wie Klassendiagramme in Rational Rose erstellt werden beschreiben die Kapitel *4.5 Klassen* und *4.7 Klassenbeziehungen*.

- Sequenzdiagramm (engl. *sequence diagram*)
  Ein Sequenzdiagramm zeigt Objekte und deren Aufrufe untereinander zur Laufzeit. Wichtig ist hierbei die vertikale Zeitachse, die durch eine gestrichelte Lebenslinie unter jedem Objekt dargestellt wird. Die Aufrufe können als Methodenaufrufe (*procedure calls*), als synchrone oder asynchrone Nachrichten oder auch als Aufrufe eigener Methoden entlang der Lebenslinie ausgelegt werden. Je nach gewünschtem Detaillierungsgrad können auch Aufrufparameter und Rückgabewerte angegeben werden.

  Sequenzdiagramme sind das Thema des Kapitels *3.4.3 Zeit kommt ins Spiel: Sequenzdiagramm*.

- Kollaborationsdiagramm (engl. *collaboration diagram*)
  Ein Kollaborationsdiagramm zeigt, ähnlich wie ein Sequenzdiagramm, welche Objekte bei einer bestimmten Systemaktivität zusammenarbeiten. Die zeitliche Abfolge der Aufrufe kann hier durch vorangestellte

Reihenfolgenummern angegeben werden. Kollaborationsdiagramme werden aber vorrangig benutzt um darzustellen, welche Objekte überhaupt in einer bestimmten Situation zusammenarbeiten. Die Zeitachse spielt hier also eine untergeordnete Rolle.

Kollaborationsdiagramme werden im Kapitel *3.4.4 Zusammenarbeit darstellen: Kollaborationsdiagramm* näher behandelt.

- Zustandsdiagramm (engl. *state diagram*)
  Ein Zustandsdiagramm zeigt, in welchen Zuständen (*states*) ein Objekt sich während seiner Lebensdauer befindet. Das hier beschriebene Objekt ist dabei meistens kein konkretes Objekt, sondern ein allgemeines Objekt als Repräsentant seiner Klasse, also Ein_Kunde (als Repräsentant der Klasse »Kunden«) nicht Kunde_Meier (als konkretes Objekt). Die dargestellten Zustandsübergänge sind Aufrufe von Methoden dieses Objektes oder eintretende Ereignisse (auch: »stimuli«). Methodenaufrufe und Ereignisse können mit Bedingungen verknüpft werden, unter denen das Objekt dann in den einen oder anderen Zustand wechselt.

  Wie Zustandsdiagramme in Rational Rose erstellt werden erläutert Kapitel *4.8 Zustandsmodell*.

- Komponentendiagramm (engl. *component diagram*)
  Komponentendiagramme zeigen den physischen oder logischen Aufbau eines Systems. Physische Einheiten werden häufig als Komponenten (engl. *component*) dargestellt. Das kann eine ausführbare Datei, eine Klassenbibliothek, eine DLL oder Ähnliches sein.

  Logische Einheiten werden dagegen meistens als Pakete (engl. *packages*) dargestellt. Diese Unterscheidung ist allerdings nicht zwingend. Die UML bezeichnet als Package einfach eine Gruppierung von Modellelementen.

  Die Erstellung von Komponentendiagrammen in Rational Rose ist Thema des Kapitels *4.9 Komponenten*.

- Verteilungsdiagramm (engl. *deployment diagram*)
  Verteilungsdiagramme zeigen die Verteilung der physischen Systemresourcen. Knoten werden durch Quader dargestellt und können zusätzlich zu ihrem Namen noch einen Typ angeben. Knoten können Komponenten oder Objekte enthalten.

  Verteilungsdiagramme in Rational Rose erläutert Kapitel *4.10 Knoten und Verteilung*.

1.2.3 Modelle strukturieren

Ein komplettes System, das nicht ganz trivial ist, lässt sich kaum in einem Diagramm beschreiben. Deshalb besteht ein Modell in der Regel aus mehr als einem Diagramm. Da es unser wichtigstes Ziel ist, den Überblick zu behalten, müssen also auch alle diese Diagramme irgendwie organisiert werden. Die UML bietet dazu Sichten (engl. *view*) und Pakete (engl. *package*) an.

Sichten

Sichten stellen verschiedene Aspekte des gleichen Systems dar. Eine Sicht ist kein Diagramm, sondern eine Sammlung von Diagrammen und Paketen. Zur Organisation des Modells können folgende Sichten benutzt werden:

1. Anforderungen (Use Case View)
   Hier wird die Funktionalität des Systems aus der Sicht eines Benutzers dargestellt. In dieser Sicht werden vorwiegend Anwendungsfalldiagramme benutzt. Zur detaillierten Darstellung einzelner Anwendungsfälle können Aktivitätsdiagramme, Sequenzdiagramme, Kollaborationsdiagramme und Klassendiagramme benutzt werden.
2. Logik (Logical View)
   Diese strukturelle Sicht zeigt die Funktionalität des System von innen. Die statische Struktur wird hauptsächlich mit Klassendiagrammen beschrieben. Zur Darstellung des dynamischen Verhaltens benutzt man Aktivitäts-, Sequenz- und Kollaborationsdiagramme.
3. Komponenten (Component View)
   Die Komponentensicht stellt dar, aus welchen physischen Code-Komponenten (ausführbare Programme, Link-Libraries, Skripts etc.) das System besteht. Hier werden nur Komponentendiagramme benutzt.
4. Prozesse (Concurrency View)
   Dies ist die Sicht auf gleichzeitig ablaufende Prozesse im System. Kommunikations- und Synchronisationsfragen werden in Form von Komponenten-, Verteilungs-, Sequenz- und Aktivitätsdiagrammen beschrieben.
5. Verteilung (Deployment View)
   Hier wird die Verteilung des Systems auf die vorhandene Hardware gezeigt. Rechner, Drucker, Ein- und Ausgabegeräte und anderes werden als Knoten (node) in Verteilungsdiagrammen dargestellt.

Pakete

Innerhalb der Sichten können Pakete gebildet werden, um das Modell weiter zu strukturieren. Den Paketen wird von der UML kein bestimmter Inhalt zugewiesen. Pakete sind lediglich eine Zusammenfassung anderer

Modellelemente. Pakete können Anwendungsfälle, Akteure, Klassen, Schnittstellen, Diagramme, Komponenten oder andere Pakete enthalten.

1.2.4 Welche Diagramme wann?

Die UML ist eine grafische Notation zur objektorientierten Spezifikation, Konstruktion und Dokumentation von Systemen. Sie ist also von ihren Erfindern nicht nur für Softwaresysteme konzipiert worden. Das zeigt sich am besten daran, dass UML-Diagramme bereits in der Analysephase eingesetzt werden können. Dort geht es ja noch nicht darum, das zu erstellende Softwaresystem zu beschreiben, sondern das Geschäftsumfeld in Form eines Geschäftsmodells. Auch für ein solches Geschäftsmodell können bereits alle in der UML verfügbaren Diagrammtypen sinnvoll eingesetzt werden.

Aufgabenbeschreibung

Die Aufgabenbeschreibung entsteht aus den Vorgaben des Auftraggebers. In ihr wird in natürlicher Sprache (d. h. nicht formalisiert) festgehalten, welche Zielsetzung das zu entwickelnde Programm hat, in welcher Umgebung es eingesetzt werden soll und welche Ergebnisse erwartet werden. Neben den rein sprachlichen Dokumenten sollte ein erstes Modell (Geschäftsmodell) Bestandteil der Aufgabenbeschreibung sein. Das Geschäftsmodell wird als Klassenmodell ohne Klassenmethoden erstellt und kann bei Bedarf um Aktivitätsdiagramme, Sequenzdiagramme oder andere Systemdiagramme ergänzt werden.

In einem Geschäftsmodell werden Entitäten (Kunde, Lieferant, Artikel etc.) und deren Beziehungen zueinander aufgeführt. Der Modellierer muss sich an dieser Stelle also noch keine Gedanken über benötigte Attribute machen, sondern einfach festhalten, wie die für das Programm wichtigen Verhältnisse im modellierten Geschäftsbereich aussehen. Häufig werden vom Interviewpartner aber bereits Angaben zu benötigten Daten gemacht. Diese können als Attribute den Klassen im Geschäftsmodell zugeordnet werden.

Die Beziehungen werden mit aussagekräftigen Verben bezeichnet, die die Aktivität der einen Geschäftsklasse in Bezug auf die andere Geschäftsklasse beschreiben, zum Beispiel: Kunde <erteilt> Auftrag, Auftrag <umfasst> Artikel.

Anwendungsfallanalyse

In der Anwendungsfallanalyse wird ermittelt, wie der zukünftige Anwender das System sieht, das heißt, welche Eingaben er durchführt und welche Ergebnisse er erwartet. Die Anwendungsfallanalyse stützt sich im Wesentlichen auf Interviews mit den zukünftigen Anwendern. Ausge-

hend von der Aufgabenbeschreibung werden die Anwendungsfälle zunächst ermittelt und grob beschrieben. Die Verfeinerung der Anwendungsfälle findet meistens erst beim detaillierten Design statt. Ergebnis der Analyse sind Anwendungsfälle als Dokumente und Anwendungsfalldiagramme.

Systemdesign Hier werden erste Designentscheidungen auf einer hohen Abstraktionsebene getroffen. Eventuell wird das System zunächst in verschiedene Pakete aufgeteilt. Um eine genauere Beschreibung des zu erstellenden Systems zu erhalten, werden komplexe Anwendungsfälle durch Aktivitätsdiagramme oder Sequenzdiagramme ergänzt.

Designmodell Anhand der Aufgabenbeschreibung, der Anwendungsfallanalyse und des Systemdesigns wird ein weiter gehendes Klassenmodell der gewünschten Anwendung erstellt. Dazu wird das oben erwähnte Geschäftsmodell unter Berücksichtigung des entstandenen Systemdesigns in ein Designmodell überführt. Das Designmodell enthält die im Geschäftsmodell aufgeführten Entitäten sowie Klassen, die sich aus der internen Programmorganisation ergeben, und vergibt Zuständigkeiten in Form von Klassenmethoden. Die Klassenmethoden umfassen alle in der Anforderungsanalyse und im Systemdesign aufgeführten Funktionen. Das Designmodell legt aber noch nicht fest, aus welchen Basisklassen die im Programm benötigten Klassen abgeleitet werden sollen, da diese von der Sprache und vom verwendeten Framework etc. abhängen.

Das Designmodell beschreibt ebenfalls Abhängigkeiten und Beziehungen zwischen den gefundenen Klassen. Dazu werden Kollaborations- oder Sequenzdiagramme verwendet. Wenn die Lebensdauer bzw. der Werdegang einzelner Objekttypen wichtig ist (ähnlich Datenfluss), wird dies in Zustandsdiagrammen (engl. *state diagram*) festgehalten. Für jeden zu dokumentierenden Objekttyp wird dabei ein eigenes Zustandsdiagramm angefertigt.

Implementationsmodell Der letzte Schritt ist das Implementationsmodell. Hier wird das Designmodell um Klassen erweitert, die für die Implementation in einer bestimmten Programmiersprache benötigt werden. Auch die Entwicklung mit einem bestimmten Framework und die aus dem Framework benutzten Klassen werden erst im Implementationsmodell festgelegt.

1.2.5 Erweiterungsmöglichkeiten

Um die UML nicht zu komplex werden zu lassen, haben die »drei Amigos« einige Konzepte weggelassen, die in anderen Modellierungssprachen vorhanden sind. Dafür machten sie die UML erweiterbar, damit sie an die Bedürfnisse des Benutzers, des Teams oder der verwendeten Methodik angepasst werden kann. Was Sie an der Oberfläche von Rational Rose sehen, sind im Allgemeinen nicht die puren UML-Elemente, sondern bereits erweiterte Versionen davon.

Es gibt drei Erweiterungsmechanismen in der UML, nämlich »Tagged values«, »Constraints« und »Stereotypes«.

- Tagged Values
  Ein »Tag« ist ein kleines Schild, das man irgendwo anheften kann. Tagged Values (deutsch: Eigenschaftswert) sind also Werte, die man an andere UML-Elemente anheften kann. Vor- und Nachbedingungen einer Klassenoperation oder auch ein Dokumentationstext sind Beispiele für solche Tagged Values.
- Constraints
  Constraints (deutsch: Einschränkung) sind Regeln, die das Verhalten eines oder mehrerer UML-Elemente einschränken. Constraints können für Klassen oder Objekte und auch für Beziehungen zwischen diesen gelten. Einschränkungen lassen sich mithilfemithilfe der Object Constraint Language (OCL) definieren. Deren Anwendung wird in der UML-Spezifikation 1.4 Teil 6 beschrieben.
- Stereotypes
  Die Stereotypes sind der ausgefeilteste Erweiterungsmechanismus der UML. Ein Stereotyp erweitert die Bedeutung eines Modell Elements oder schränkt diese ein. Ein typischer Stereotyp ist die Angabe, dass eine Klasse <<abstrakt>> ist (siehe oben). Die Bedeutung des Modellelements »Klasse« wird durch den Stereotyp <<abstrakt>> eingeschränkt, der aussagt, dass von dieser Klasse keine Objekte erstellt werden können (sondern nur von ihren Unterklassen, siehe auch *2.1 Warum Objektorientierung?*). Dieses Beispiel zeigt auch, wie solche Stereotypes in der einfachsten Form dargestellt werden, nämlich als <<Wort>>, eingeschlossen in spitze Klammern.

In Rational Rose sind die wichtigsten Tagged Values, Constraints und Stereotypes bereits vordefiniert. Wie Sie eigene Erweiterungen definieren können, erfahren Sie im Kapitel *5.6 Konfiguration und Erweiterung*.

1.3 Die Rolle von Rational Rose

1.3.1 Sinn und Umfeld

Die Größe von OO-Modellen kann leicht mehrere hundert Klassen – um nur diese zu betrachten – umfassen. Damit verbietet sich das manuelle Erstellen der entsprechenden Diagramme praktisch von selbst: das Versammeln aller Klassen auf einem einzigen Diagramm ist allein zeichentechnisch nicht möglich, und die Koordination von Änderungen über mehrere Diagramme »per Hand« ist aufwendig und fehlerträchtig. Dementsprechend dürfte diese Arbeit in einem Projekt schnell liegen bleiben und sich die gezeichneten Diagramme vom eigentlichen Modell entfernen.

Hier setzt Rational Rose an: Es führt eine komplette Datenbank über *alle* in einem Modell verwendeten Konstrukte wie Klassen, ihre Beziehungen, Use-Cases etc. und erlaubt deren Darstellung auf verschiedenen Diagrammen. Dabei werden alle Diagramme konsistent gehalten, wenn Sie z. B. den Namen einer Klasse ändern. Ein neuer Name wird von Rose also automatisch auf allen Diagrammen, die die umbenannte Klasse enthalten, gezeigt. Die Darstellung eines einzelnen Elements auf mehreren unterschiedlichen Diagrammen ermöglicht aber noch mehr: so kann jeweils ein funktionaler Ausschnitt aus dem gesamten Modell auf einem Diagramm im Zusammenhang dargestellt werden, ohne dass das Diagramm zu komplex wird und zu einem »Schnittmusterbogen« verkommt.

Ein Diagramm ist dann übersichtlich, wenn es sich auf eine DIN-A4-Seite drucken lässt und noch lesbar ist (*Managerregel*).

So ist es z. B. üblich, auf einem separaten Diagramm nur die Vererbungshierarchie der Klassen in einem Paket zu zeigen. Außer zur Dokumentation von objektorientierten Modellen dient Rose aber auch als zentrales Tool, um die technische Organisation von Projekten zu unterstützen. Hier fließen alle Informationen zusammen: die Anforderungen, die in Use-Cases und anhand von Akteuren erfasst wurden, die Darstellung des Ablaufs anhand von Interaktions- oder Aktivitätsdiagrammen, die Verteilung der Arbeit auf einzelne Entwicklungsteams und die zu erstellenden oder beteiligten Komponenten, sprich: Dateien mitsamt ihren Abhängigkeiten.

Durch die Verknüpfung der Elemente in dem Modell erlaubt Rose es Ihnen, ausgehend von einer Teilansicht eines Problems durch das gesamte Modell zu navigieren und sich damit Stück für Stück einen umfassenden Überblick zu verschaffen.

Ist Ihre Aufgabenstellung das Implementieren, so bietet Rose Ihnen die Möglichkeit, aus den im Modell erstellten Klassen automatisch ein Codegerüst zu erzeugen, das Sie dann mit den semantischen Anweisungen füllen können. Alle Deklarationen von Attributen und von Operationen mitsamt ihrer Argumente und Rückgabewerte können Sie aus dem Modell erzeugen lassen.

Dabei dient Rose nicht nur als »Vorstufe« zur Codeerzeugung, sondern bietet auch eine Hilfestellung bei der Konstruktion. So können Sie sich mithilfe von Zustandsdiagrammen einen Überblick über die möglichen Zustände einer komplexen Klasse verschaffen und entdecken beim Aufbau eines solchen Diagramms und beim Durchgehen der Zustände und Transitionen noch zusätzlich notwendige Operationen, um spezielle Zustandsübergänge zu ermöglichen oder finden Verhalten in Ihren Klassen, das unerwünscht ist und das Sie daher mithilfe einer speziellen Implementation beseitigen müssen.

1.3.2 Einblick in die Historie

Rational Rose erblickte das Licht der Welt Ende 1991 als das erste nicht in ADA realisierte Projekt von Rational Software. Seine Anfänge liegen in den Händen von Grady Booch, der 1989 den ersten Prototypen auf der R1000 Hardware von Rational in ADA schrieb. Bald danach folgte eine Smalltalk-Version auf Intel Hardware (einem Laptop). Unsere Anfrage bei Rational Software führte zu einem langen archäologischen Mail-Verkehr: voll von ‚Ach, damals ...' und ‚Weißt Du noch ...?'. Das Entstauben und Sortieren der verschiedenen Funde zeigt uns die folgende Historie – sicherlich nicht vollständig und sehr wahrscheinlich teilweise glorifiziert:

Die erste Version von Rose entstand 1991/92 bei Rational. Es war die Version 1.0 und sie wurde für Unix erstellt. Sie wurde in Smalltalk geschrieben und es war gleichzeitig das erste Projekt für Rational, das nicht mit Ada entwickelt wurde. Rose 1.0 benutzte eine objektorientierte Versant-Datenbank zum Speichern des Modells und unterstützte nur die Booch-Notation. Das war insofern kein Wunder, da dieses Programm auf einem von Grady Booch selbst erstellten Tool aufsetzte, das es ermöglichte, die Struktur von Ada-Progammen als Diagramm zu zeigen.

Rose 2.0 war 1993 die erste Version, die für Windows- und Unix-Plattformen erstellt wurde. Dies wurde unter Zuhilfenahme einer Architektur ermöglicht, die durch den Kauf von Palladio Software zu Rational kam. Mit der Version 2.0 wurden die bis heute verwendeten ASCII-Dateien als Format für die Speicherung der Modell eingeführt, da sich die Datenbank

als zu langsam herausgestellt hatte. Außerdem wurde hier die Möglichkeit eingebaut, Teile des Modells als »Controlled Unit« zu speichern. Wegen des angebotenen Windows-Produktes wurde diese Version ein erster Verkaufserfolg für Rational.

Nachdem James Rumbaugh in die Firma eingetreten war, wurde mit der Version 3.0 von Rose auch die Darstellung in dessen OMT angeboten. Außerdem kam in dieser 1995 erschienenen Version erstmals die Generatoren und Reverse-Engineering-Tools für C++ hinzu. Das Benutzerinterface wurde mithilfe der Microsoft Foundation Classes (MFC) um den Browser erweitert.

Als letzte Version in diesem Nummernschema erschien 1996 Rose 4.0. Sie erhielt Unterstützung für die von Ivar Jacobson bei Rational eingebrachten Use-Cases, die Unterstützung für Roundtrip-Engineering von C++ wurde verbessert und die ersten Ansätze zur Anbindung von Microsoft Visual Basic wurden integriert. Außerdem war dies die erste Version, die ein intern genutztes API bekam.

Mit Rose98 wurde die Jahreszahl des Erscheinens in die Versionsbezeichnung aufgenommen. Diese Version bekam als erste eine Unterstützung die UML-Notation. Das neu designte API wurde nun auch für Fremdhersteller nutzbar und als das Rose-Extensibility-Interface (REI) veröffentlicht. Dieses REI ermöglicht das Erstellen von Add-Ins mit gleicher Architektur. In einem Service Pack dieser Version wurde dann auch die Erstellung von Aktivitätsdiagrammen ermöglicht. In der Version Rose98 wurde ebenfalls die Unterstützung von Java eingeführt.

Danach folgten die Versionen 2000 und 2001, die vor allem im Detail verbessert wurden: es kamen neue Tools wie der HTML-Generator hinzu und weitere Details zur Abbildung der UML wurden eingebaut. Vor allem wurde das REI von Drittanbietern dazu genutzt, mehr als 100 Add-Ins auf den Markt zu bringen. In der Version 2000 wird die erste Version des Data Modeler angeboten.

In der Version 2001 wird eine Java-IDE-Unterstützung für VisualAge for Java, VisualCafe, Forte for Java und JBuilder integriert und die J2EE-Unterstützung eingebaut.

Die diesem Buch zugrunde liegende Version ist die 2001A Patch 2. Etwa zweimal im Jahr erscheint ein Update von Rose, in dem Fehler beseitigt werden und das neue Funktionen erhält.

1.3.3 Einsatzmöglichkeiten von Rose

Bei der Arbeit mit Rose ist neben der Möglichkeit, UML-Diagramme zu zeichnen, das eigentlich Wichtigere, dass Rose es erlaubt, alle technischen Konstrukte, die für die Erstellung der Software erstellt werden, zu verwalten.

Mithilfe eine Use-Case-gesteuerten Arbeitsweise erstellen Sie ein Modell Ihrer Applikation, von dem ausgehend Sie sich alle Arbeitsergebnisse, die erstellt wurden, wieder erschließen können. Natürlich reicht Rose dazu nicht allein aus, es bietet z. B. keine eigene Möglichkeit zur Speicherung von Anforderungsdokumenten. Aber es bietet als COM-Client Möglichkeiten, beliebige externe Dokumente einzubinden und in dem Modell zu verwalten.

Organisation der Dateien

So können Sie z. B. einen Formularentwurf zu einem Use-Case in Visual Basic als .FRM-Datei erstellen, diese als Referenz zum Use-Case speichern und in dem Fall, dass Sie sie im Zuge Ihrer Arbeiten am Use-Case ansehen oder bearbeiten wollen, starten Sie per Doppelklick auf das Symbol in Rose Visual Basic und haben direkt das Formular auf dem Bildschirm.

Dadurch, dass Rose über diese Wege die Verbindung mit fast beliebigen unter Windows verfügbaren Office- und Entwicklungswerkzeugen erlaubt, bietet es Ihnen eine große Flexibilität bei der Auswahl Ihrer Umgebung und Ihres Programmiersystems. Es ist sogar möglich, in Rose Modelle zu erstellen, deren Einzelteile in verschiedenen Sprachen programmiert werden sollen. So ist denkbar, eine mehrschichtige Architektur zu erstellen, deren Front-End in VB erstellt wird, die Geschäftslogik als Java-Klassen auf dem Server zu erstellen und die Datenspeicherung in einer relationalen Datenbank wie Oracle vorzunehmen, deren Datenschema ebenfalls in Rose spezifiziert werden kann.

Freie Wahl der Arbeitsweise

Auf diese Weise erlaubt Rose, die Änderungen, die sich im Verlauf eines Projektes an den einzelnen Teilmodellen ergeben und die sich über die in der UML festgelegten Beziehungen auf angrenzende Konstrukte auswirken, bereits innerhalb des Modells zu verfolgen und so eine Abschätzung der notwendigen Arbeiten zu machen (s. a. *5.2 Navigation zwischen den Konstrukten*).

Rational Rose gibt keine eigene Arbeitsweise beim Aufbau eines Projektes vor, unterstützt aber sehr stark den Einsatz der UML.

Eine übliche Arbeitsweise sieht so aus, dass Sie zunächst die Akteure und die für diese zu implementierenden Use-Cases erfassen. Damit haben Sie

dann eine Sammlung der Hauptfunktionen, die das Programm dem Kunden bieten soll. Als Nächstes können Sie die Abläufe in den Use-Cases in Form von Szenarios beschreiben. Rose lässt Ihnen dabei die Wahl, ob Sie lieber mit Sequenz- oder mit Kollaborationsdiagrammen arbeiten wollen (s. a. *3.4.1 Szenarios als Instanzen von Anwendungsfällen* und *4.4 Objekte und Interaktionen*). Wenn Sie während Ihrer Arbeit feststellen, dass Sie eine Darstellung in der jeweils anderen Diagrammart benötigen, können Sie diese einfach per Knopfdruck auseinander erzeugen.

Wenn Sie die für die Objekte notwendigen Klassen definieren wollen, können Sie dies nun direkt aus den Szenariodiagrammen heraus tun. Dabei behalten Sie den Überblick, wozu diese Klasse in dem Kontext dieses Szenarios (und des übergeordneten Use-Cases) dienen soll. Nachdem Sie die Klassen definiert haben, können Sie deren Beziehungen ebenfalls basierend auf den Objekten und ihrer Interaktionen festlegen. Rose bietet Ihnen die Möglichkeit, dabei ganz systematisch die Objekte in einem Interaktionsdiagramm durchzuarbeiten und parallel dazu ein Klassendiagramm zu erstellen.

Selbstverständlich erlaubt Rose auch den umgekehrten Weg, nämlich zunächst mit der Erstellung eines Klassenmodells zu beginnen und daraus dann die Objekte zu erzeugen, die in den verschiedenen Szenarios benötigt werden.

In der Praxis ist es natürlich nicht so, dass diese Arbeiten jeweils in der genannten Reihenfolge sequenziell hintereinander erfolgen, sondern es wird ein ständiger Wechsel zwischen diesen beiden Arbeitsrichtungen und den damit verbundenen Diagrammen stattfinden. Da Rose aber beide Richtungen unterstützt, können Sie es so einsetzen, wie es Ihrer Arbeitsweise entspricht.

Einarbeiten in die Möglichkeiten von Rose

Sollten Sie gerade erst dabei sein, sich in die Bedienung von Rose einzuarbeiten, so kann es verwirrend sein, dass es keine festgelegte Arbeitsreihenfolge gibt. In diesem Fall sollten Sie zunächst die Kapitel *3* und *4* durcharbeiten. Diese erläutern Ihnen einen einfachen Weg, mit dem Sie Ihre Arbeit beginnen können. Dabei werden Sie lernen, wie die verschiedenen UML-Konstrukte erreicht werden und Sie werden Ihren Arbeitsstil entwickeln können. Im Kapitel *5* finden Sie dann Informationen zu weiteren Funktionen, die Ihnen in Ihrem Arbeitsalltag helfen.

1.3.4 Die wichtigsten Features

In diesem Buch gehe ich ausschließlich auf die Windows-Version von Rational Rose ein. Zwar gibt es auch eine UNIX-Variante, diese ist funktional und in der Bedienung aber weitestgehend identisch zur Windows-Version. Der Hauptunterschied zwischen den Versionen besteht darin, dass die UNIX-Version nicht als COM-Server aus externen Programmen nutzbar ist, sondern dass hier lediglich interne Skriptmöglichkeiten bestehen. Beiden Versionen von Rose ist gemein, dass sie nur mit einer englischen Bedienerführung erhältlich sind.

Unterstützung der UML

Mithilfe von Rose lassen sich ausnahmslos alle UML-Konstrukte modellieren und dokumentieren. Dabei unterstützt Rose seit der Version 2001A alle wichtigen, wenn auch noch nicht alle Notationsoptionen. So fehlt z. B. die Möglichkeit, Einschränkungen (engl. *constraints*) einzugeben und es ist auch noch nicht möglich, in Interaktionsdiagrammen die korrekte UML-Notation mit Unterscheidung zwischen Nachrichten und Antworten zu verwenden. Die nicht unterstützten UML-Merkmale bedeuten jedoch keine Einschränkung der Praxistauglichkeit von Rose.

Aufbau der Datenbank

Rose speichert alle Informationen zu einem Modell in einer einzigen Datei. Diese Dateien haben die Endung .MDL und haben ein einfach aufgebautes ASCII-Format. Beim Öffnen einer Modell-Datei lädt Rose alle Informationen aus der Modell-Datei in den Speicher und arbeitet ab diesem Moment nur noch mit den Speicherinhalten. Erst wenn Sie Rose ausdrücklich dazu auffordern zu speichern, wird der Speicherinhalt wieder in die Datei geschrieben.

Um bei dieser Funktionsweise mit mehreren Personen gleichzeitig an einem Modell zu arbeiten, bedarf es jedoch einiger Maßnahmen zur Konfiguration (s. a. *5.5 Team-Development*).

Rational Rose erlaubt in einer Arbeitssitzung lediglich *ein Modell zur gleichen Zeit* zu öffnen. Es besitzt also nicht – wie die meisten Office-Produkte – eine mehrdokumentenfähige Benutzerschnittstelle (»MDI« f. engl.: *multiple document interface*). Allerdings erlaubt Rose das gleichzeitige Öffnen beliebig vieler Diagramme zu einem Modell, sodass Sie sich ihre Arbeitsumgebung jeweils aus verschiedenen Ansichten zusammenstellen können und zwischen diesen Ansichten beliebig wechseln können. Dabei ist immer ein Diagramm – nämlich das zuvorderst liegende – das, in dem Sie arbeiten können: neue Elemente mithilfe der Toolbar anlegen oder dessen Ansicht mit den Menübefehlen manipulieren.

Sollten Sie doch einmal zwei oder mehr Modell-Dateien gleichzeitig öffnen wollen, so haben Sie natürlich die Möglichkeit, Rose mehrfach auf Ihrem Rechner zu starten. Dann können Sie in jeder Task ein Modell bearbeiten.

Es ist nicht möglich, Rose zu starten, *ohne* eine Modell-Datei zu öffnen. Selbst wenn Sie den Startdialog, der Sie auffordert, eine neue Datei zu erstellen oder eine vorhandene zu öffnen, ohne Auswahl abbrechen, wird eine leere Datei angelegt. In diesem Falle wird als Name des geöffneten Modells »(untitled)« in der Titelzeile des Fensters von Rose angezeigt.

Alle Modelle in Rose haben von Anfang an eine Einteilung in verschiedene Sichten: die »Use Case View«, die »Logical View«, die »Component View« und die »Deployment View«. Hinzu kommen die »Model Properties« (s. a. *Der Browser* und *5.6.5 Modellproperties*).

Aufteilung eines Modells

Alle Sichten außer der Deployment View (»Verteilungssicht«) können weiter in Pakete unterteilt werden. Dabei erlaubt Rose für die jeweiligen (Unter-) Pakete nicht das Einfügen beliebiger Konstrukte, sondern schränkt die Auswahl ein. So ist es z.B. nicht möglich, Klassen in der Component View zu erstellen oder Komponenten in der »Use Case View« zu verwalten. Grundsätzlich, wenn auch nicht uneingeschränkt, bedeutet dies, dass Sie Ihre Analysearbeit an Akteuren, Use-Cases etc. in der Use-Case-Sicht machen, die Designarbeit an Klassen in der Logischen Sicht und die Implementationsvorbereitung in der Komponentensicht.

Die Deployment View besteht in Rose in Wahrheit aus nur einem Diagramm. Das ist aber insofern sinnvoll, da dieses Diagramm dazu dient, die Verteilung der Prozesse des Gesamtsystems darzustellen. Eine Darstellung auf verschiedenen, sich womöglich widersprechenden Diagrammen würde wohl nur zu Verwirrung führen.

Diese Aufteilung in verschiedene Sichten entspricht dem Modell der »4-1-View« (»vier-plus-eins-Sicht«), die dem Entwurf der UML selbst als Sprache für »Use case driven modeling« (»Anwendungsfallgetriebene Modellierung«) zugrunde liegt. Diese Aufteilung sieht vor, die zu modellierenden Teilaspekte eines Softwaresystems in vier Quadranten aufzuteilen. Diese vier Quadranten dienen dann als Bereiche, in denen die verschiedenen beteiligten Personen die Konstrukte wiederfinden, mit denen sie am meisten zu tun haben oder die sie modellieren sollen.

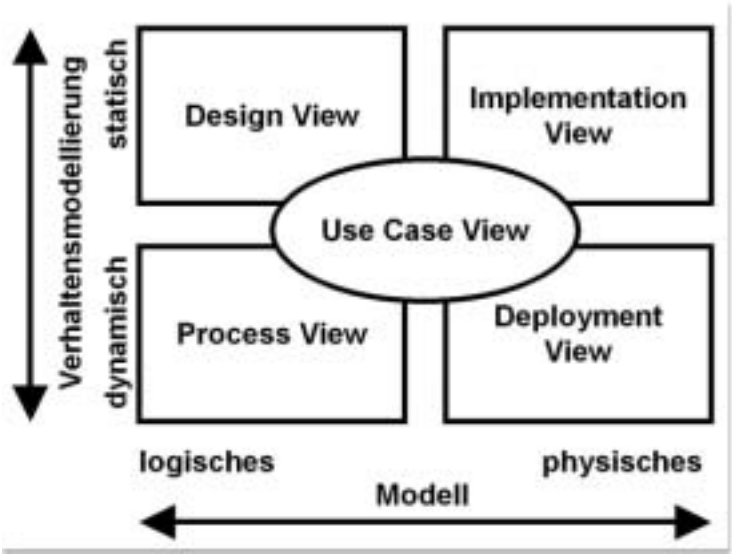

Abbildung 1.1 Schema der 4+1 View

Als Verbindung zwischen diesen Quadranten dient dann die »Use Case View«. Die hierzu gehörigen UML-Konstrukte bieten jeweils den Ausgangspunkt für alle weiteren Arbeiten an einem Modell. Weitere Informationen hierzu finden Sie im Buch von Philippe Kruchten ([Kruchten 99]) oder beim Rational Unified Process [RUP].

Da es möglich und meist auch praktisch ist, viele Diagramme und Dialogfenster gleichzeitig auf dem Bildschirm geöffnet zu haben, ist Rational Rose ein sehr anspruchsvolles Tool, was die maximale Bildschirmauflösung angeht. Eine Arbeit mit Auflösungen von weniger als XGA (1024 * 768 Bildpunkte) ist nicht anzuraten. Wenn Sie dann noch gleichzeitig andere Programme wie die IDE Ihres Programmierwerkzeugs oder Word für die Eingabe von Dokumentation geöffnet haben, so gilt: »viel hilft viel«. Die Anschaffung von guten Monitoren und entsprechenden Grafikkarten, die Auflösungen von 1600 * 1200 Bildpunkten erlauben, machen sich schnell in einer höheren Produktivität bezahlt.

Seit der Version 2001 können Sie Diagrammsets, die sie sich in einer Arbeitssitzung aufgebaut haben, auch als Arbeitsumgebung (*workspace*) abspeichern. Das erlaubt Ihnen das nahtlose Weiterarbeiten an einer entsprechenden Stelle, wenn Sie Ihre Arbeit unterbrechen müssen.

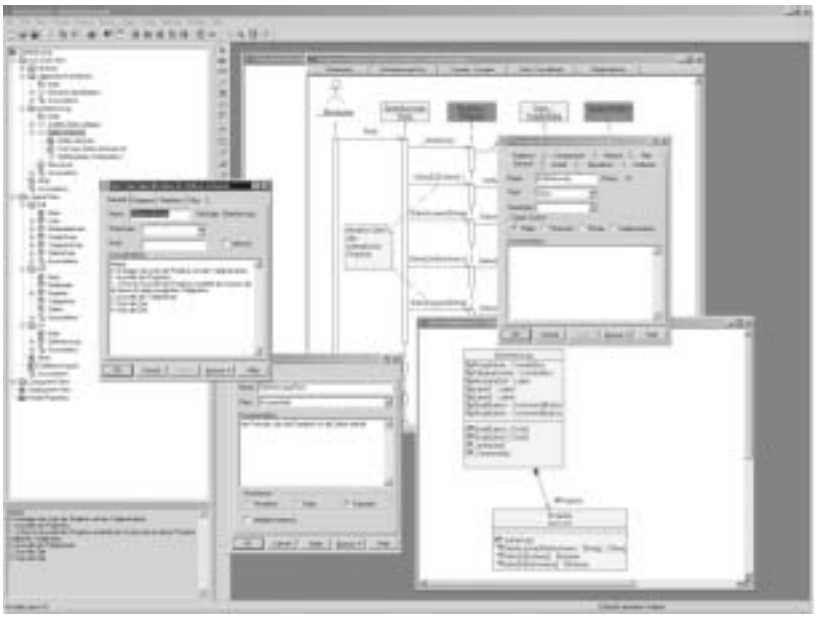

Abbildung 1.2 Die Darstellung auf einem 1600 * 1200 Pixel großen Monitor

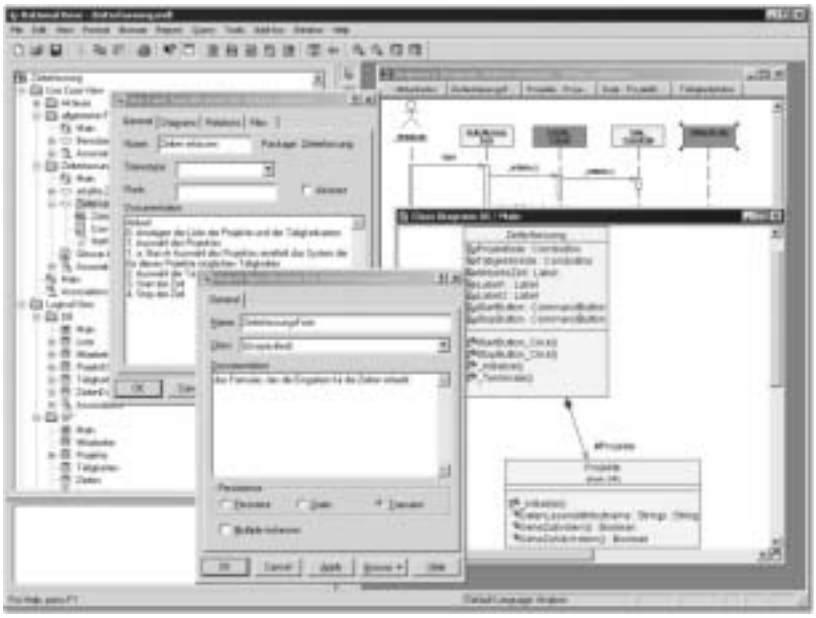

Abbildung 1.3 Das fast gleiche Setup auf Standard-XGA verliert bereits an Übersichtlichkeit

1.3.5 Die Grenzen von Rose

Die Architektur von Rational Rose hat zum Ziel, Rose als spezialisiertes Tool für visuelle Objektmodellierung innerhalb möglichst vieler Arbeitsumgebungen einsetzen zu können. Dies bringt natürlich auch Einschränkungen mit sich. Zunächst möchte ich daran erinnern, dass Rose ein CASE-Tool zur Objektmodellierung ist.

Arbeitsprozess

Es hat daher nicht den Anspruch, Ihnen einen Arbeitsprozess vorzugeben, sondern ist – wie die anderen CASE-Tools dieses Segments auch – dazu designed, sich Ihrem Prozess anzupassen. Sollten Sie noch nicht über einen beschriebenen Arbeitsablauf für Ihre Softwareprojekte verfügen, so wird Rose Ihnen diesen nicht bieten (können). Ein Werkzeug zur visuellen Objektmodellierung übernimmt also nur einen, wenn auch wichtigen Teil bei der Organisation Ihres Projektes. Weiterführende Informationen zur Organisation Ihres Arbeitsprozesses finden Sie beim Rational Unified Process [RUP] oder bei der Beschreibung des ICONIX-Prozesses [Rosenberg 99].

Da Rose auf spezielle Funktionen beispielsweise zum Layout von Use-Case-Dokumenten verzichtet, sind Sie darauf angewiesen, für diese Arbeiten ein separates Tool (zum Beispiel MS Word) einzusetzen. Rose bietet Ihnen also keine eingebaute Funktionalität zur Strukturierung von Use-Cases und zur Erfassung der zugehörigen Texte. Immerhin haben Sie damit die Möglichkeit, für diese Art Dokumente ein Werkzeug Ihrer Wahl zu verwenden.

Anbindung an Programmiersysteme

Ebenso fehlt die Integration in spezielle Programmiersprachen oder -systeme, wie sie andere UML-Tools teilweise bieten. Rose speichert also nur die reinen UML-Informationen zu den Klassen und keinen Quellcode dazu. Damit ist ein direktes Umschalten der Modellansicht in eine Quellcodedarstellung meist nicht ohne weiteres möglich, sondern muss über die Schritte Codegenerierung oder Reverse- bzw. Roundtrip-Engineering gehen. Inzwischen gibt es zwar Erweiterungen, die ein direktes Darstellen und auch Editieren von Quellcode zu einer Klasse erlauben, aber es besteht immer die Gefahr, dass sich das Modell in Rose von Konstrukten im Code unterscheidet. Der Vorteil dieser Aufteilung in Modellierungswerkzeug und Programmiersystem, zu dem ja neben einem Editor für den Code auch Compiler und Debugger gehören, ist aber, dass Sie Modelle erstellen können, die Gebrauch von beliebigen Implementationsprachen machen. Es lassen sich sogar Modelle mit Rose erstellen, in denen sich Klassen für mehrere Programmiersprachen verwalten lassen.

Möglichkeiten der Codegenerierung

Die mit Rose lieferbaren Codegeneratoren erlauben das Erzeugen von Klassen- und Dateigerüsten. Sie sehen nicht vor, aus weiteren Modellteilen wie Interaktions- oder Zustandsdiagrammen Anweisungen zu erstellen, die bereits Instruktionen enthalten. Dies mag auf den ersten Blick leicht möglich sein. Bei näherer Betrachtung werden Sie aber feststellen, dass Sie, falls Sie einen solchen Codegenerator erstellen wollten (was prinzipiell natürlich möglich ist), verschiedene Erweiterungen an der UML selbst machen müssten. Daneben wäre der Aufwand, solche Diagramme als Vorgabe für Code zu erstellen, sicher höher, als den Code selbst dort zu erstellen, wo er sowieso hingehört: in die IDE des Programmiersystems.

Papierdokumentation

Dadurch, dass Rose keinen internen Editor für formatierte Dokumentationstexte hat, gehört die Ausgabe von zusammenhängenden Dokumentationen auf Papier sicherlich auch nicht zu den Stärken von Rose. Es lassen sich zwar ohne weiteres Ausdrucke von Diagrammen erzeugen, aber die Informationen, die z. B. in Word-Dokumenten zu den Use-Cases enthalten sind, können damit nicht ohne weiteres verbunden werden. Dieser Nachteil relativiert sich jedoch recht bald, wenn Sie sich die Natur von Papierdokumenten als statischem Abbild eines temporären Modellzustands vor Augen führen. Dessen Nutzwert wird mit zunehmendem zeitlichen Abstand von der Ausgabe auf dem Drucker beständig und rapide geringer. Daneben hat auch druckfrisches Papier nur begrenzte Aussagekraft. Es ist nämlich praktisch unmöglich, darauf alle Querverweise auf dem Stand zu halten bzw. die zugehörigen Informationen über diese Verweise zu »erblättern«. Denken Sie z. B. daran, dass von einer Klasse Objekte in Szenarios verwendet werden, dass die zugehörige Superklasse in einer anderen Komponente implementiert ist, und dass zu diesen jeweils eigene Ausdrucke erzeugt wurden, auf deren Blattnummer Sie referenzieren müssen.

Viel Wichtiger und auch besser nutzbar erscheint daher die Möglichkeit, eine HTML-Dokumentation zu erzeugen (s. a. *5.4 Dokumentationsausgabe*).

1.3.6 Angrenzende Tools

Als Ergänzung zu Rose benötigen Sie in der Regel weitere Werkzeuge, um zusätzliche Aufgaben in Ihrem Projekt zu erledigen. Neben dem schon genannten MS Word für Use-Cases und einem Programmiersystem für die Programmierung sind dies zum Beispiel Programme zur Versionsverwaltung, zur Ausgabe von integrierten Dokumenten, zum Testen und zur Verwaltung von Anforderungen.

Rational bietet neben der Einzellizenz für Rose eine ganze Reihe von Programmsammlungen an. Diese sind zugeschnitten auf verschiedene Schwerpunkte bei der Arbeit an Softwareprojekten, nämlich Analyse, Design, Implementation und Test. Zu den in diesen Paketen enthaltenen Programmen liefert Rational jeweils Schnittstellen zu Rose mit. Damit können Sie Ihren Arbeitsprozess nahtlos durch entsprechende Werkzeuge unterstützen.

Die Art des Zusammenwirkens mit Rose kann dabei unterschiedlich sein: von der transparenten Anbindung z. B. an Versionierungstools wie ClearCase von Rational oder Visual SourceSafe von Microsoft, über die Datenübertragung per spezieller Schnittstellenprogramme an und von der Anforderungsverwaltung von z. B. Rational Requisite Pro bis zur einfachen selbstentwickelten COM-Anbindung von Dokumentations- und Präsentationswerkzeugen wie MS Word oder Powerpoint.

Einen Sonderstatus nimmt hier Rational SoDA ein. Dies ist ein Werkzeug, das aus fast schon beliebigen Quellen eine integrierte Dokumentation in Form eines Word- oder Framemaker-Dokumentes erzeugt. Dieses Tool erlaubt die Einbindung von Grafiken und Texten aus Rose, kann diese zwischen manuell erstellte Textpassagen einfügen und auch andere Dateiinhalte aus anderen Rational-Tools, wie z. B. Anforderungen aus Requisite Pro einmischen. Wie bereits erwähnt, ist der Nutzwert von Papierdokumentation allerdings oft sehr begrenzt. Daher sollten Sie sich überlegen, ob sich die Anschaffung und der nicht unerhebliche Aufwand in die Einarbeitung eines solchen Werkzeugs lohnt.

1.4 Zusammenfassung

In diesem Kapitel haben wir Ihnen einen Einblick in die Welt der Objektorientierung gegeben. Diese Technologie ist sicher dazu geeignet, bessere Software zu schreiben. Allerdings *ermöglicht* Objektorientierung diese Verbesserung nur, sie *liefert* sie nicht automatisch. Die Anwendung der OO bietet zudem keine festen »Kochrezepte«, die, wenn man ihnen streng folgt, zu einem sicheren Projekterfolg führen. Die »richtige« Anwendung der Objektorientierung bedarf einer gewissen Erfahrung, denn sie ist eine Sammlung von Konstrukten und darauf anzuwendenden Prinzipien, deren Wirkungen sich teilweise widerstreben.

»Optimale Lösung?«

Es ist daher unumgänglich, dass Sie sich in der Anwendung der OO üben. Eine »optimale« Lösung erhalten Sie meist nur dadurch, dass Sie unterschiedliche Aspekte und die Auswirkungen unterschiedlicher Konstruktionen mit Kollegen diskutieren und dann entscheiden, welche Vorteile des

jeweiligen Modells ihnen so wichtig sind, dass Sie die damit verbundenen Nachteile in Kauf nehmen. Die OO steht damit in einem Gegensatz zu verschiedenen »klassischen« Modellierungen wie dem Aufbau von Petrinetzen oder der Entity-Relationship-Modellierung, zu denen es Algorithmen gibt, um die Korrektheit des Modells zu beweisen.

UML als Grundlage

Das Mittel zur Diskussion und Konstruktion von Objektmodellen ist die UML. Sie bietet die Sprache, mit der Sie Ihre Ideen mitteilen können und ermöglicht Ihnen den raschen Austausch von Modellierungsansätzen. Daneben bietet Sie die Möglichkeit, die Modellergebnisse für die Nutzung in weiteren Arbeiten zu dokumentieren. Beachten Sie aber, dass die Erstellung von Dokumentation niemals Selbstzweck sein darf. Viele Projekte leiden an der »Analysis Paralysis«, also einer Blockade vor dem Übergang zum Design. Diese resultiert daraus, dass sich die Beteiligten unsicher darin sind, ob sie alle Anforderungen erfasst und dokumentiert haben. Ziel eines Softwareentwicklungsprojektes ist es aber, lauffähige Software und nicht Dokumentation zu liefern. Sie werden eine lange Zeit mit einem Modell leben müssen, das noch nicht perfekt ist, sondern das nur den jeweils aktuellen Erkenntnisstand darstellt. Insofern kann die Erstellung von Dokumentation nur das Mittel zum Zweck, nämlich der späteren Verwendung und dem Erhalten des Verständnisses sein. Wie in der Beschreibung des gerade erschienenen »Agile Manifesto« für »Leichte Softwaremethoden« [SD 2001-08] gesagt: »The issue is *not* documentation – the issue *is* understanding«, also »Die Aufgabe ist *nicht* Dokumentation – die Aufgabe *ist* Verstehen« (s. a. Website der Agile Alliance, [AgileAlliance]).

UML sinnvoll einsetzen

Gerade die UML macht es mit ihrer Vielfalt an Diagrammen aber leicht, sich in Arbeiten zu verlieren, die nicht unbedingt notwendig sind. Bevor Sie also einen Diagrammtyp einsetzen, überlegen Sie immer, was Sie damit erreichen wollen und ob der Aufwand lohnt.

Bei der Arbeit mit einem UML-Modell besteht also meist kein »endgültiges« Ergebnis, sondern das erstellte Modell ist über lange Zeit ein Zwischenstadium beim Aufbau des Softwareproduktes. Es dient als Diskussionsgrundlage und als Arbeitsmittel. In das Modell und die erstellten Diagramme fließen neue Erkenntnisse und Ideen ein, die Sie beim Austausch mit Ihren Kollegen erhalten haben. Modellierung und Diagramme sind also immer Arbeitsprozess und nicht Zustand.

CASE-Tools einsetzen

Sie können sich sicher vorstellen, welchen Aufwand es bedeutet, diese Diagramme ohne Computerunterstützung zu erstellen: Jede Änderung bedeutet eine aufwendige Nacharbeit und nach mehreren Änderungen

ist ein komplettes Neuzeichnen fällig. Um diese Arbeit nicht unnötig zu erschweren, brauchen Sie also ein Werkzeug wie Rational Rose, das Ihnen die flexible Einarbeitung von Änderungen erlaubt und das jeweils den Stand Ihrer Modellierung sichert.

Voraussetzung für den sinnvollen Einsatz eines solch mächtigen Werkzeugs ist aber wiederum, dass Sie sich im Klaren darüber sind, was Sie erreichen wollen. Der Einsatz von Rose ist nämlich ebenso wenig Selbstzweck wie der Einsatz der Objektorientierung. Im Extremfall kann der nicht sachgemäße Einsatz von komplexen Werkzeugen sogar ein Projekt zum Scheitern bringen: wenn nämlich der Hauptteil der täglichen Arbeit darin besteht, mit den Werkzeugen klar zu kommen und nicht darin, das Produkt zu entwickeln. Ein möglicherweise nicht für die Softwarebranche gemünzter, aber dort sehr gut passender Spruch ist »A fool with a tool is still a fool«, also »Ein Narr mit einem Werkzeug ist nach wie vor ein Narr«. Und man möchte hinzufügen: Er wird mit einem Werkzeug womöglich sogar gefährlich.

Aufbau des Buches

Damit Sie sich in eine bessere Lage bei der Anwendung des Werkzeugs Rational Rose begeben, haben wir die folgenden Kapitel als Anleitung zu dessen Einsatz erstellt. Sie sind so aufgebaut, dass Sie zunächst die wichtigsten Grundlagen erfahren und dann ein erstes eigenes Modell anhand von Übungen erstellen (Kap. *3 Der schnellste Weg zum ersten Modell*).

Aufbauend darauf stellen wir Ihnen alle Funktionen von Rose zur Erstellung der kompletten UML-Diagrammtypen vor. Diese Vorstellung wird wieder durch Übungen unterstützt, sodass Sie einen detaillierten Einblick in die Funktionen und die Zusammenhänge von Rose bekommen (Kap. *4 Die UML-Konstrukte im Detail*).

Neben der reinen Modellierung gibt es aber noch weiteren Nutzen, den Sie aus Ihrem Modell ziehen können. Wir stellen Ihnen also neben den Schritten zum Aufbau eines Modells auch Zusatzfunktionen vor, die Ihnen z.B. die gemeinsame Nutzung von Modellen mit mehreren Entwicklern ermöglichen (Kap. *5 Der Projektalltag*).

Die häufigsten Fragen zu Rose drehen sich um die Erzeugung von Quellcode. Hierzu haben wir die Kapitel zu den C++-, Java- und Visual Basic-Generatoren erstellt. Diese leiten Sie durch die verwinkelten Möglichkeiten aber auch die Grenzen der Codeerzeugung mit Rose (Kap. *7 Generatoren und Reverse-Engineering*).