7 Objektorientierte Softwareentwicklung und die UML

Nach dem Überblick über die verschiedenen klassischen Aspekte des Software Engineering leitet nun das letzte Kapitel dieses einführenden Teils zum Kern des Buches über, der methodischen objektorientierten Softwareentwicklung.

7.1 Objektorientierte Softwareentwicklung

Das objektorientierte Paradigma, ursprünglich im Rahmen von Programmiersprachen wie z. B. Simula, Smalltalk oder C++ eingeführt, hat zusammen mit entsprechenden Modellierungstechniken große Hoffnungen hinsichtlich der produktiven Entwicklung qualitativ hochwertiger Software geweckt.

Tatsächlich trägt die Objektorientierung zu einer systematischeren und produktiveren Softwareentwicklung bei. Die *Kapselung* von Modell- und Softwarebausteinen in Klassen mit sauberen Schnittstellen und das Verbergen von Realisierungsdetails (*Geheimnisprinzip*) ermöglichen eine wohldefinierte Zerlegung komplexer Modelle in präzise spezifizierte Bausteine mit klar definierten Kommunikationsspielregeln. Die *Vererbung* als technischer Aspekt der *Generalisierung* erweitert bisherige Modellierungsansätze um ein wichtiges Konzept, das im Entwurf und bei der Implementierung eine höhere Flexibilität ermöglicht und damit das Wiederverwendungspotenzial erhöht und die Produktivität steigert.

Allerdings haben sich die hohen Erwartungen bezüglich der *Wiederverwendung* bisher nur teilweise erfüllt. Einerseits bieten Klassenbibliotheken, Rahmenwerke und Entwurfsmuster deutlich bessere Wiederverwendungsmöglichkeiten als konventionelle Entwicklungsmethoden und -techniken. Jenseits dieser implementierungsnahen Ebene ist die Wiederverwendung jedoch ein Stiefkind geblieben. Es zeigt sich, dass Wiederverwendung vor allem ein Managementproblem ist. Solange das Management nicht willens ist, zunächst Aufwand zu investieren, um dann die Früchte der Wiederverwendung genießen zu können, wird sich daran wenig ändern.

Ein großer Vorteil der Objektorientierung besteht in der *Durchgängigkeit* von Methoden und Techniken durch alle Aktivitäten der Softwareentwicklung. Insbesondere die Brüche zwischen Anforderungsermittlung und Entwurf, wie sie andere Methoden aufweisen, können hier deutlich verringert werden. Zentrale Modellkonstrukte der Anforderungsspezifikation finden sich im Entwurf und sogar in der Implementierung

wieder, natürlich auf jeweils niedrigerem Abstraktionsniveau. So ist man dem alten Traum der *Softwareentwicklung durch Verfeinerung* ein Stück näher gekommen: Beginnend mit abstrakten, noch unscharfen Modellen werden diese Stück für Stück verfeinert und präzisiert, bis sich schließlich der Programmcode fast kanonisch ergibt. Natürlich ist das gezeichnete Bild überoptimistisch. Dennoch ist die Durchgängigkeit bei objektorientierten Ansätzen deutlich höher als bei konventionellen Vorgehensweisen.

Sollte es also tatsächlich etwas auf der Welt geben, das nur Vorteile und keine Nachteile hat? Natürlich nicht, und schon gar nicht in der Softwareentwicklung. Saubere objektorientierte Softwareentwicklung ist schwer und stellt erhebliche Anforderungen an die intellektuellen Fähigkeiten und technischen Fertigkeiten der Entwickler. Nur weil jemand Programme in C++ zum Laufen bringt, ist er noch lange kein guter objektorientierter Entwickler. Ein Porsche oder Ferrari macht aus einem mäßigen Autofahrer noch lange keinen Fahrer, der die Möglichkeiten des Fahrzeugs adäquat umsetzt. Schlimmer noch, er wird oft mehr Unheil im Straßenverkehr anrichten als mit einem Golf, der mit einem bescheidenen Motor ausgerüstet ist (so kommen überproportional viele Sportwagen in Kurven von der Straße ab, obwohl ihre Straßenlage viel besser als die gewöhnlicher Autos ist).

Objektorientierte Softwareentwicklung muss intensiv gelernt und geübt werden, bis sie ihre Vorteile voll ausspielen kann. Dies fällt vor allem denjenigen Entwicklern schwer, die bislang imperativ programmieren und eine Sprache wie Cobol oder C gelernt haben (selbst wenn sie mittlerweile mit einer objektorientierten Sprache arbeiten). In der industriellen Softwareentwicklung bilden diese Entwickler immer noch eine große Gruppe. Entwickler, die bereits mit modularen Techniken vertraut sind (vgl. [PagSix94]), kennen mit der Kapselung und dem Geheimnisprinzip schon zwei fundamentale Prinzipien der Objektorientierung und müssen »nur noch« die Vererbung bzw. Generalisierung verstehen und richtig anzuwenden lernen.

Ein weiteres Problem der objektorientierten Softwareentwicklung besteht darin, dass systematisches *Testen* der Software nur in eingeschränktem Maß möglich ist. Zwar sind funktionale Tests, die ja als Black-Box-Tests vom Programmcode abstrahieren, wie üblich einsetzbar, doch systematische strukturelle Tests sind nur beschränkt möglich. Ursache ist die kombinatorische Explosion möglicher Programmzustände, die sich aus Objektzuständen, Interaktionen zwischen Objekten und insbesondere aus Vererbungsbeziehungen ergeben. Es gibt viele Fachleute, die aus diesem Grund objektorientierte Software als wenig geeignet für sicherheitskritische Anwendungen ansehen.

Schließlich erfordert die objektorientierte Softwareentwicklung angepasste *Vorgehensmodelle* und *Managementtechniken*. Aus Sicht des Managements wird die Fortschrittskontrolle durch die höhere Durchgängigkeit der Methoden nicht gerade leichter. Klassische Meilensteine etwa am Ende der Phasen »Anforderungsermittlung« und »Entwurf« lassen sich nicht ohne weiteres übertragen, da oft schwer zu entscheiden ist, ob man sich z. B. noch in der Anforderungsermittlung oder schon im Entwurf befindet. Dedizierte objektorientierte Vorgehensmodelle wie z. B. der in Abschnitt 3.3 skizzierte »Rational Unified Process« [JBR99] wirken überladen und überkompliziert. Neu eingeführte objektorientierte Entwicklungsaktivitäten in einen ebenfalls neuen, komplizierten Entwicklungsprozess einzubetten, dürfte in den allermeisten Fällen des Guten zu viel sein.

7.2 UML im Überblick

Den ersten Schwerpunkt des Buches bilden die verschiedenen Modellierungskonstrukte der *Unified Modeling Language (UML)*, die zur Spezifikation objektorientierter Modelle im gesamten Softwareentwicklungsprozess eingesetzt wird. Entstanden ist die UML aus dem Bestreben, die Vielzahl der zu Beginn der 90er Jahre entstandenen objektorientierten Notationen zusammenzufassen und ein einheitliches Metamodell für Softwareentwickler zu schaffen ([OMG97], [BRJ99], [OMG03a]).

Die UML zielt darauf ab, sämtliche Aspekte aller nur denkbaren Realweltsituationen und Softwaresysteme mit dedizierten Konstrukten modellieren zu können. Dieser Universalitätsanspruch ist zugleich ihre größte Schwäche. Die UML umfasst ein riesiges Arsenal von Modellierungselementen, die sich zum Teil nur sehr subtil unterscheiden. Selbst erfahrene Entwickler werden die UML kaum derart vollständig verinnerlichen, dass sie stets in der Lage sind, ein besonders gut geeignetes Konstrukt zur Modellierung eines bestimmten Sachverhalts auszuwählen. Vor diesem Hintergrund kann es auch nicht überraschen, dass die UML trotz etlicher Überarbeitungen immer noch einige unklar definierte und zum Teil widersprüchliche Elemente enthält.

Trotz der genannten Kritikpunkte hat sich die UML als De-facto-Standard für objektorientierte Modellierung durchgesetzt. Um mit der UML zurechtzukommen, sollte man sie sich zunächst auf seinen Anwendungsbereich zuschneiden, d.h. eine kleine Teilmenge klar definierter Konstrukte auswählen, die den zu modellierenden Aspekten möglichst gerecht wird. Auch dieses Buch wird nur einen Ausschnitt der UML behandeln können und wollen.

Wichtig ist, dass die UML nur eine *Notation*, nicht aber eine Entwicklungsmethodik festlegt. Wie ihr Name sagt, ist sie lediglich eine Sprache. Die UML gibt auch nicht vor, in welcher Reihenfolge die Modelle zu erstellen sind und wie sie untereinander zusammenhängen. Um zu einer systematischen objektorientierten Softwareentwicklung zu gelangen, wird daher zusätzlich eine geeignete *Vorgehensmethodik* benötigt. Dabei wird in neuerer Zeit zwischen modellbasierten und modellgetriebenen Entwicklungsmethoden unterschieden.

7.3 Modellbasierte Softwareentwicklung

Bei der modellbasierten Softwareentwicklung werden Modelle vornehmlich zur Spezifikation und Dokumentation erstellt und bilden somit eine Art »Blaupause« für die Implementierung. Hierbei übersteigt der Aufwand für die Modellierung oft den für die reine Programmierung. Mit Hilfe von Modellen, die von irrelevanten und überdetaillierten Aspekten des Realweltproblems und der Software abstrahieren, versucht man, die bei der Entwicklung größerer Softwaresysteme auftretende Komplexität konzeptuell zu beherrschen ([Ludewig03], [Luft84]). Dabei werden verschiedene Modelle eingesetzt, die unterschiedliche abstrakte Sichten auf den jeweiligen Sachverhalt darstellen. Zentrale Sichten sind die strukturelle, die funktionale und die verhaltensorientierte Sicht.

Strukturelle Modelle, wie z.B. das Klassenmodell, beschreiben die für die Anwendungsdomäne oder die zu entwickelnde Software relevanten Objekte sowie deren Beziehungen untereinander. *Funktionale Modelle*, wie z.B. Anwendungsfälle (use

cases, vgl. Kapitel 12), charakterisieren die globalen Funktionalitäten der Anwendung, liefern also gewissermaßen die Funktionen, für die die in strukturellen Modellen festgehaltenen Objekte verantwortlich sind. *Verhaltensorientierte Modelle* kann man unterteilen in Modelle zur Beschreibung des *Ablaufverhaltens* von Funktionen (z.B. Interaktionsdiagramme, vgl. Kapitel 14) und des *Objektverhaltens* (z.B. Zustandsdiagramme, vgl. Kapitel 15). Man kann verhaltensorientierte Modelle als Verfeinerungen struktureller und funktionaler Modelle betrachten: Zustandsübergangsdiagramme präzisieren das Verhalten von Objekten, während Interaktionsdiagramme die Abläufe von Funktionen genauer spezifizieren.

Die Einschränkung auf die jeweilige Sicht reduziert die Komplexität des einzelnen Modells, bringt aber das Multimodellproblem mit sich: Der Blick für das Ganze ist erschwert, und vor allem treten Konsistenzprobleme zwischen den Modellen auf. Insgesamt sind jedoch mehrere Modelle die bessere Alternative zu einem monolithischen Modell, das bei gleicher Ausdrucksmächtigkeit nicht mehr handhabbar ist.

Bei der oben skizzierten Vorgehensweise der modellbasierten Softwareentwicklung treten zwei Probleme auf. Erstens spiegeln sich Änderungen, die z.B. zur schnellen Reaktion auf geänderte oder neue Anforderungen direkt am Quellcode vorgenommen wurden, nicht im Modell wider, und zweitens trägt das Modell selbst nur mittelbar zum eigentlichen »Produkt« bei, da es von den Entwicklern ja bis hin zur Implementierung zu verfeinern ist.

Eine Lösung der Probleme wird durch das Single-Source-Prinzip (vgl. Abschnitt 6.2) ermöglicht: Wechselweise werden dabei teils das Modell verfeinert und teils der das Modell darstellende Quellcode erweitert, bis das Anwendungssystem lauffähig ist. Änderungen des Quellcodes schlagen sich dann unmittelbar im Modell nieder. Allerdings ist der Abstraktionsgrad des resultierenden Modells so gering (man bewegt sich ja auf Quellcodeebene), dass viele Entwickler lieber nur noch mit dem Quellcode arbeiten und die Modellsicht überflüssig wird.

7.4 Modellgetriebene Softwareentwicklung

Einen ganz anderen Weg beschreitet die *modellgetriebene Softwareentwicklung* (model driven architecture, MDA [MelBal02], [WarKle03], [MSU+04]; model driven software development, MDSD [StaVöl05]).

Einfach ausgedrückt wird dabei ein Modell fachlich so präzisiert, bis dass das Anwendungssystem im Rahmen einer standardisierten Architektur generiert werden kann – ähnlich der computergestützten Fertigung, bei der ein dreidimensionales Modell eines Werkstücks in einem CAD-System erstellt, zusammen mit den notwendigen Bearbeitungshinweisen in ein Programm für die CNC-Werkzeugmaschine übersetzt, von dieser auf einem Rohling ausgeführt und so als reales Werkstück erstellt wird [StaVöl05].

Detaillierter betrachtet geht man in mehreren Schritten vor. Zunächst erstellt man ein deskriptives, von jeglicher »IT-Sicht« unabhängiges Modell des Anwendungsbereiches (computation independent model, CIM). Daraus leitet man manuell ein rein fachliches, technologieneutrales Modell für das Anwendungssystem ab (platform independent model, PIM), aus dem dann ein plattformspezifisches Modell (platform specific

model, PSM) für eine bestimmte Architektur bzw. Realisierungstechnologie generiert wird. Aus diesem wird – ggf. nach weiteren Zwischenschritten – das Implementierungsmodell (implementation specific model, ISM) bzw. direkt der Quellcode des Anwendungssystems erzeugt. Den Zusammenhang verdeutlicht Abbildung 7–1.

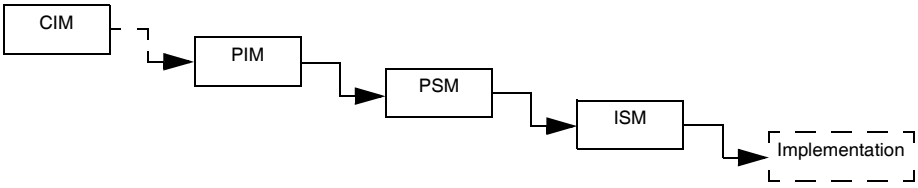

Abb. 7–1 *Grundprinzip der modellgetriebenen Entwicklung*

Natürlich ist die modellgetriebene Softwareentwicklung in der Praxis weitaus komplexer, als es obige CAD/CAM-Analogie vorspiegelt, so dass die zur Generierung notwendige fachliche Präzisierung der Modelle nur hochqualifiziertem Personal möglich ist, welches zu gleichen Teilen Analytiker-, Modellierer- und Entwicklerkompetenz hat. In vielen Darstellungen wird daher das CIM völlig vernachlässigt und das PIM als lediglich etwas gröbere Variante des PSM dargestellt. Oft werden zwar Funktionen für querschnittliche Aspekte wie Persistenz oder Transaktionsverhalten generiert, anwendungsspezifische Funktionalität ist jedoch manuell in die generierten Rahmenprogramme einzuarbeiten. Eine aus technischer Sicht umfassende Einführung in die Thematik bietet [StaVöl05].

Bei aller Eleganz des Konzepts sind einige Probleme nicht von der Hand zu weisen. So unterscheiden sich viele Anwendungsbereiche sehr stark voneinander, so dass eigene domänenspezifische Modellierungselemente oder Sprachen erforderlich werden, was zu einer erneuten Zerfaserung der Modellierungstechniken führen kann. Zudem erinnert der notwendige Präzisionsgrad der Modelle an denjenigen formaler Spezifikationstechniken, die sich in der Praxis nur in sehr speziellen Anwendungsgebieten etablieren konnten.

Letztendlich steht der Beweis aus, dass die modellgetriebene Entwicklung tatsächlich zu der versprochenen Qualität und den Einsparungen bei der Erstellung und Wartung großer Anwendungssysteme führt. Erfahrungen mit ähnlichen Ansätzen wie z. B. den in den 80er Jahren beschworenen »4th generation languages« zeigen, das eine erfolgreiche Generierung anspruchsvoller interaktiver Anwendungssysteme mit hohen Anforderungen hinsichtlich einer ergonomischen, effektiv zu verwendenden Benutzungsschnittstelle eher unwahrscheinlich ist.

7.5 MOOS: Methodische objektorientierte Softwareentwicklung

Nach dieser Einführung werden in den Teilen II und III des Buchs zunächst die wichtigsten UML-Elemente für die strukturelle, funktionale und Verhaltensmodellierung vorgestellt und an einfachen Beispielen demonstriert. Danach werden in den Teilen IV

bis VIII typische Tätigkeiten der Anforderungsermittlung, der Softwarespezifizierung, der Architekturkonzeption und des Grob- und Feinentwurfs besprochen und an einem durchgängigen Fallbeispiel demonstriert, wobei die UML als Modellierungssprache benutzt wird. Die Tätigkeiten im Einzelnen haben folgende Ziele:

- Bei der *Anforderungsermittlung* das Herausarbeiten der funktionalen und nichtfunktionalen Anforderungen, die so weit wie möglich noch benutzerverständlich dargestellt werden, sowie die Validierung der Anforderungsspezifikation durch die Anwender in Hinblick auf Vollständigkeit und fachliche Richtigkeit.
- Bei der *Softwarespezifizierung* die Integration der unterschiedlichen Modelle der Anforderungsspezifikation und die Präzisierung der Anforderungen im Spezifikationsklassenmodell sowie die Verifikation im Hinblick auf Konsistenz und modelltechnische Richtigkeit.
- Bei der *Architekturkonzeption* die globale Strukturierung des Systems gemäß der nichtfunktionalen Anforderungen und die Festlegung der globalen Realisierungsentscheidungen.
- Im *Grobentwurf* die Überführung der Softwarespezifikation in die Architektur sowie die Berücksichtigung wichtiger Aspekte der Benutzungsschnittstelle und der persistenten Datenhaltung.
- Im *Feinentwurf* dann die Verfeinerung des Grobentwurfs zur präzisen Implementierungsvorgabe unter Berücksichtigung der Zielplattform (z.B. Implementierungssprache sowie Rahmenwerke und Datenbanksysteme).

Die im Buch verfolgte »Methodische objektorientierte Softwareentwicklung« (Moos) stützt sich somit einerseits auf den im »klassischen« Software Engineering üblichen Tätigkeiten ab (vgl. Kapitel 2). Andererseits ebnet MOOS den Weg zur modellgetriebenen Entwicklung, da – mit Ausnahme der Anforderungsermittlung – die Tätigkeiten im Kern eine Folge systematischer Transformationen von Modellen darstellen. MOOS verbindet somit die Vorteile beider Welten, indem das Bewährte mit dem Vielversprechenden systematisch zu einem homogenen Ganzen verschmolzen wird.

Ein Wort noch: Obwohl die Tätigkeiten in MOOS zu einem großen Teil schematisierbar und automatisierbar sind, stelle ich sie im Buch so dar, als ob sie manuell ausgeführt werden. Sie, liebe Leserin und lieber Leser, sollen damit vom technischen Ballast hinsichtlich einer Automatisierung befreit in die Lage versetzt werden, die methodische Vorgehensweise anhand der einzelnen, wohl begründeten Schritte nachvollziehen und hinsichtlich Ihrer eigenen Bedürfnisse bewerten und anpassen zu können, um sie dann – ggf. automatisiert in einem modellgetriebenen Umfeld – erfolgreich anzuwenden.