Prof. Dr. Hans-Werner Six, Dr. Mario Winter

Kurs 01793

Software Engineering

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Kapitel 2

Allgemeine Aspekte

2.1 Was ist Software Engineering?

Eine Verbesserung der Situation bei der Softwareentwicklung kann nur durch den breiten Einsatz gesicherter Software Engineering-Methoden und kompetentes Personal erzielt werden. Was verstehen wir unter *Software Engineering*? Knapp gesprochen befasst sich Software Engineering mit der systematischen Entwicklung großer Softwaresysteme, wobei die Anwendungsentwicklung im Vordergrund steht. Detaillierter formuliert beschäftigt sich Software Engineering mit Methoden, Techniken und Werkzeugen zur Entwicklung großer Software mit dem Ziel,

- Software hoher Qualität,
- kostengünstig innerhalb eines festen Budgetrahmens und
- zum geplanten Zeitpunkt

zu produzieren.

Diese Formulierung macht deutlich, dass Software Engineering nicht nur die Softwareentwicklung im engeren Sinn behandelt, sondern auch Fragen der Qualitätssicherung (es geht um die Entwicklung von Software hoher Qualität) und des Managements zum Gegenstand hat (es geht um die kostengünstige Entwicklung innerhalb einer fest vorgegebenen Zeit). Ein weiterer wichtiger Aspekt besteht darin, dass große Softwaresysteme von heterogen zusammengesetzten Teams entwickelt werden (die beteiligten Personen setzen sich aus Auftraggebern, Anwendungsexperten, Benutzern, Entwicklern und Managern zusammen), so dass interdisziplinäres Wissen und soziale Kompetenz der Beteiligten für den Projekterfolg ebenfalls eine zentrale Rolle spielen.

Als *Softwareentwicklung* bezeichnen wir die Gesamtheit aller Aktivitäten, die zu einem lauffähigen Softwaresystem führen. Software Engineering befasst sich damit, einzelne Aktivitäten zu identifizieren und zu beschreiben, ihre Ergebnisse festzulegen und in einem Vorgehensmodell anzuordnen. Dabei wird häufig nach *produktbezogenen (*eher technischen*)*

und nach *prozessbezogenen (*eher nicht-technischen*) Aktivitäten* unterschieden. Produktbezogene Aktivitäten finden z. B. bei der Anforderungsermittlung und der Implementation statt; ihre Ergebnisse gehen direkt in das Produkt ein — z. B. als definierende Dokumente oder kompilierbarer Code. Prozessbezogene Aktivitäten stellen den Organisations- und Managementrahmen für die produktbezogenen Aktivitäten in den Vordergrund und umfassen z. B. die Projektplanung, die Leitung und Kontrolle während der Projektdurchführung, die Produktverwaltung und das Qualitätsmanagement.

Software Engineering ist heute ein umfangreiches und ausdifferenziertes Fachgebiet der Informatik, das vielfältige Bezüge zu anderen Fachgebieten der Informatik hat (wobei die Grenzen unscharf sind) und im Austausch mit anderen Disziplinen wie Organisationstheorie, Arbeitspsychologie, Betriebswirtschaft und den Ingenieurwissenschaften steht. Hervorzuheben ist die zunehmende Auffächerung des Fachgebietes. Ursprünglich wurde Software Engineering anwendungsneutral gesehen. Doch werden je nach Anwendungsbereich inzwischen wichtige Unterschiede deutlich. So muss bei Systemen zur Steuerung technischer Prozesse der Schwerpunkt auf Zuverlässigkeit, systeminterne Fehlerbehandlung und Erfüllung von Echtzeitanforderungen gelegt werden, während bei interaktiven Anwendungssystemen die Benutzbarkeit im Vordergrund steht und Fehler oft besser vom Benutzer als vom System behoben werden können. Durch verteilte Basishardware, Rechner am Arbeitsplatz, Mikroelektronik in technischen Geräten und moderne Medien zur Mensch-Rechner-Interaktion ergeben sich zudem neue Gesichtspunkte, die in eigenen Teildisziplinen behandelt werden.

Seit der Entstehung des Software Engineering gibt es eine Vielzahl nebeneinander existierender Schulen, die jeweils ihre Sicht vertreten, eigene Methoden lehren, weiterentwickeln und den sich ändernden Anforderungen anpassen. Dazu stellen sie passende Werkzeuge zur Unterstützung dieser Methoden bereit. Heute existiert nicht *das* Software Engineering, sondern es gibt verschiedene sinnvolle Ausprägungen für unterschiedliche anwendungs- und technologiebezogene Schwerpunktsetzungen.

2.2 Eigenschaften von Software

Zu einem adäquaten Verständnis der Softwareentwicklung muss man sich zunächst die Eigenschaften von Software klarmachen. Zu den grundsätzlichen Merkmalen gehören:

- *Software ist nicht sinnlich wahrnehmbar.* Zugang zum Verständnis bieten nur die definierenden Texte oder die Erprobung des bereits existierenden Produktes. Softwarevisualisierungsverfahren befinden sich noch in den Anfängen.
- *Software ist Text.* Da Texte fast beliebig strukturiert und verändert werden können, erwartet man, dass Software an veränderliche Bedürfnisse angepaßt werden kann.
- *Software ist digital.* Daher greifen die Verfahren der traditionellen kontinuierlichen Ingenieurmathematik nicht. Die formale Behandlung beruht vielmehr auf der diskre-

ten Mathematik und der Logik. Wegen der ungeheuren Zahl möglicher Systemzustände und Interaktionen zwischen Softwaremodulen sind die entsprechenden Verfahren allerdings nur eingeschränkt praktikabel.

Wenn wir von Softwareentwicklung und Software Engineering sprechen, meinen wir die entwicklung großer Softwaresysteme. Groß ist für uns z. B. ein Softwaresystem, an dem ein Team von mehr als fünf Personen mehr als ein Jahr entwickelt. Im Gegensatz dazu kann ein "kleines" Programm von weniger als fünf Personen in wenigen Monaten erstellt werden.

Große Software weist folgende wesentliche Eigenschaften auf:

- *Große Software ist komplex.* Sie verknüpft mehrere Realitätsbereiche mit ihrem Fachwissen und ihrer Fachsprache. Sie verarbeitet kompliziert strukturierte und miteinander interagierende Anwendungsfälle, die über einen langen Zeitraum oder räumlich weit verteilt eingesetzt werden können. Dies schlägt sich in einer Software nieder, die ein System von miteinander verknüpften Modulen bildet, wobei sowohl die Verständlichkeit des Gesamtsystems als auch die der einzelnen Module anzustreben ist. Ein zentrales Anliegen des Software Engineering besteht in der Beherrschung der Komplexität.
- *Große Software besteht aus umfangreichen Artefakten, z. B.* Spezifikationen, Programmen und unterschiedlichen Formen von Dokumentation. Die Artefakte sind formalisiert und formatiert und liegen in einer grafischen Notation oder textuell vor. Ihre Konsistenz und Vollständigkeit kann insgesamt nicht überprüft werden. Auch fehlen gemeinsame Grundlagen einer verbindlichen Semantik.
- *Große Software hat eine lange Lebensdauer.* Langlebig bedeutet, dass die Software über mehrere Jahre (oft länger als 10 Jahre) eingesetzt wird, um den hohen Entwicklungs- und Einarbeitungsaufwand für die Benutzer zu amortisieren. Dabei muss sie kontinuierlich an sich ändernde Anforderungen (neue Benutzerwünsche, erweiterte Funktionalität, andere Plattformen) angepasst werden. Dies führt zur Bildung von Versionen.
- *Große Software verfestigt Sichtweisen.* Verständnis und Interessen der Beteiligten schlagen sich im computerverarbeitbaren Modell des Anwendungsbereichs nieder. Umgekehrt prägt der Einsatz von Software Rahmenbedingungen für Arbeitsabläufe.

Die Entwicklung großer Software wirft somit grundlegend andere Probleme auf und braucht andere methodisch-technische Unterstützung als die kleiner Programme. Von zentraler Bedeutung ist die Beherrschung der Komplexität.

2.3 Anwendungssysteme und Workflow Management Systeme

Anwendungssysteme (oft kurz Anwendungen oder Applikationen genannt) sind Softwareprodukte, die in einem bestimmten Anwendungsbereich eingesetzt werden. Sie unterstützen Arbeitsprozesse (Auftragsbearbeitung, Lagerhaltung) oder steuern technische Prozesse (Flugüberwachung, Kraftwerksteuerung).

Ein Anwendungssystem kann eine *Individualsoftware* sein, die für einen speziellen Auftraggeber in einem bestimmten Anwendungsbereich erstellt wird. Hierunter fallen z. B. die Steuerungssoftware für eine Fertigungsstraße oder die Website bzw. das Portal einer Firma. Ein Anwendungssystem kann aber auch eine *Standardsoftware* sein, die einen bestimmten Anwendungsbereich abdeckt und durch Konfiguration an die unterschiedlichen Bedürfnisse verschiedener Anwender angepasst werden kann. Hierzu gehören z. B. Finanzbuchhaltungs- und Materialwirtschaftssysteme, Branchenlösungen für Handwerksbetriebe und Office-Programme für Textverarbeitung und Tabellenkalkulation. Nicht unter den Begriff Anwendungssystem fallen z. B. als *Systemsoftware* Betriebssysteme und Compiler, da sie sich keinem bestimmten Anwendungsbereich zuordnen lassen.

Anwendungssysteme sind in Unternehmenskontexte eingebunden und unterstützen dort zusammenhängende Abläufe, die man als *Geschäftsprozesse* bezeichnet. Scheer definiert den Begriff Geschäftsprozess wie folgt [Scheer98]:

> Allgemein ist ein *Geschäftsprozess (business process)* eine zusammengehörige Abfolge von Unternehmensverrichtungen zum Zweck einer Leistungserstellung. Ausgang und Ergebnis des Geschäftsprozesses ist eine Leistung, die von einem internen oder externen "Kunden" angefordert und abgenommen wird.

Betrachten wir eine Kundenauftragsbearbeitung als Beispiel für einen Geschäftsprozess [Scheer98]. Ein Kunde bestellt bei der Vertriebsabteilung eines Unternehmens einige Artikel, die gefertigt werden müssen. Die Bestellung wird anhand von Informationen über Kunden und Artikel auf ihre Ausführbarkeit geprüft. Bei Auftragsannahme werden die benötigten Materialien von entsprechenden Lieferanten beschafft. Nach Eintreffen der Materialien und Einplanung des Auftrags werden die Artikel anhand eines Arbeitsplans gefertigt und an den Kunden versandt. Zu jedem Artikel wird auch eine Produktinformation erstellt und versandt.

Geschäftsprozesse sind Gegenstand betriebswirtschaftlicher Betrachtungen, für die Ziele wie z. B. "Optimierung des zeitlichen Aufwands" definiert werden und die der Kostenrechnung unterliegen. Wichtig ist das tiefe Verständnis der Anwendungssituation, was insbesondere eine große Anschaulichkeit der Modelle für Geschäftsprozesse bedingt.

Workflow-Modelle dienen als Grundlage für die Automatisierung von Geschäftsprozessen. Ein *Workflow* ist die Unterstützung oder Automatisierung (von Teilen) eines Geschäftsprozesses durch ein Computersystem [WFMC97]. Workflow-Modelle besitzen somit eine größere Realisierungsnähe als Geschäftsprozesse. Sie präzisieren alle Details der

Geschäftsprozesse und liefern eine konsistente Gesamtbeschreibung. Oft wird der Begriff *Business Process Reengineering* synonym für die Analyse und Spezifikation von Workflows gebraucht. Folgende Aspekte werden in einem Workflow-Modell spezifiziert:

- der *organisatorische Aspekt* zeigt, *wer* etwas ausführt bzw. ausführen kann (und darf);
- der *funktionale Aspekt* spezifiziert, *was* ausgeführt wird;
- der *operationale Aspekt* gibt an, *wie* etwas ausgeführt wird (z. B. computergestützt durch spezielle Anwendungssysteme oder manuell);
- der *ablaufbezogene Aspekt* legt fest, *wann* oder besser *in welcher Reihenfolge* etwas ausgeführt wird; es wird also der "Kontrollfluss" spezifiziert;
- der *informationsbezogene Aspekt* beschreibt das *Womit*, also die Daten,
- der *kausale Aspekt* letztendlich gibt an, *warum* bzw. aufgrund welcher Intra-Workflow-Abhängigkeiten etwas ausgeführt wird.

Zur Unterstützung betriebswirtschaftlich weitgehend standardisierter Workflows gibt es kommerzielle Systeme z. B. für Enterprise Resource Planning (ERP), Customer Relationship Management (CRM) und Supply Chain Management (SCM). Systeme, mit denen man Geschäftsprozesse zunächst als Workflow-Modell spezifizieren und dann auch ausführen kann, nennt man *Workflow Management Systeme* [WFMC97].

Die Geschäftsprozesse in einem Unternehmen können also mit einem Workflow-Modell in einem Workflow Management System spezifiziert werden, welches diese dann durch die koordinierte Ausführung verschiedener Anwendungssysteme automatisiert. Diesen Zusammenhang zeigt Abb. 2.1.

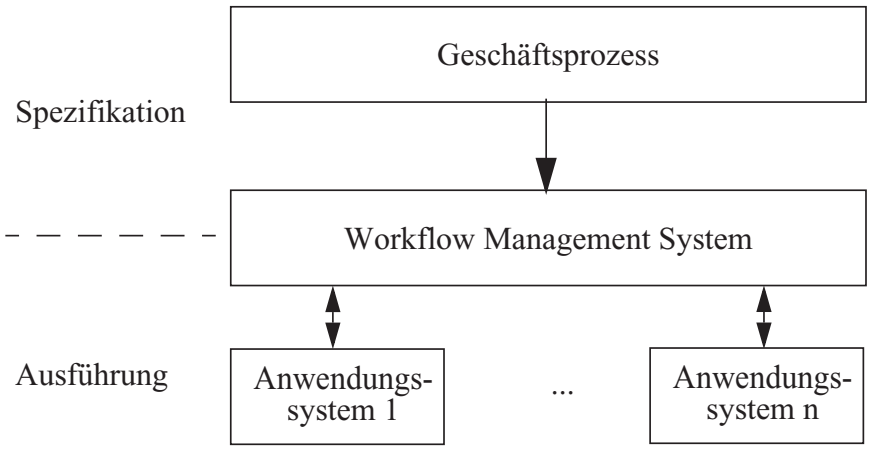

Abb. 2.1 Geschäftsprozess, WFMS und Anwendungssysteme

Bezogen auf das obige Geschäftsprozessbeispiel der Kundenauftragsbearbeitung könnten die verschiedenen Anwendungssysteme zur Bearbeitung eines Kundenauftrags z. B. dedizierte Systeme für den Vertrieb, die Beschaffung, die Fertigung und den Versand darstellen.

Kapitel 5

Management

Für das erfolgreiche Gelingen eines großen Softwareprojekts sind nicht nur exzellente Informatikkenntnisse, sondern insbesondere ein kompetentes Management nötig. Einige typische Probleme, mit denen das Management von Softwareprojekten zu kämpfen hat, beleuchten die folgenden Beispiele [Sneed87].

- Der Entwicklungsfortschritt ist oft kaum zu ermitteln, da Produkt und Teilprodukte immateriell sind. Das Projektmanagement kann selbst nicht überprüfen, ob z. B. eine Anforderungsspezifikation tatsächlich fertiggestellt ist, und ist auf die Aussagen der Entwickler oder Tester angewiesen. Erstere können fast jederzeit behaupten, sie seien mit einem Teilprodukt fertig. Zur Prüfung solcher Aussagen fehlen in der Regel einfache, aber aussagekräftige und messbare Größen.

  Hinzu kommt, dass sogar gute Entwickler oft glauben, fertig zu sein, obwohl sie es nicht sind, wenn eine Aktivität nicht eindeutig definiert wurde. Natürlich gibt es auch Entwickler, die den wahren Stand ihrer Arbeit verdecken, um Zeit zu gewinnen. Bei Großprojekten wird die Verschleierung des Projektzustandes oft von den Managern der mittleren Ebene geschickt fortgesetzt, so dass es noch schwieriger wird, den wahren Zustand zu ermitteln.

- Aufgrund des hohen Spezialisierungsgrades können die Aktivitäten bei der Entwicklung von Software nicht jedem beliebigen Entwickler zugeteilt werden. Fällt ein Entwickler aus, kann man ihn nur durch einen Entwickler vergleichbarer Qualifikation ersetzen. Trotzdem geht die Personalplanung oft fälschlicherweise von der Austauschbarkeit der Entwickler aus.

- Die Entwicklung großer Software tendiert dazu, sich grundsätzlich von früheren Entwicklungen zu unterscheiden, so dass Erfahrungen nur von begrenztem Wert sind.

- Die Entwicklung großer Software erfordert ein iteratives Vorgehen. Neue Erkenntnisse während der Entwicklung haben Auswirkungen auf die bisherigen Ergebnisse. Deshalb ist ein bestimmtes Teilprodukt immer nur bedingt fertig. Oft wird die Hälfte der

Arbeit zur Überarbeitung bereits erstellter Teilprodukte benötigt. Die Projektplanung muss solche entwicklungsinternen Zyklen berücksichtigen.

Die spezifischen Probleme des Managements von Softwareprojekten sind für Aussenstehende nur schwer zu erkennen und erfordern mehr und anders gelagerte Managementaktivitäten als bei konventionellen technischen Produktentwicklungen.

Die Aufgaben des Managements von Softwareprojekten lassen sich grob wie folgt verdeutlichen.

- ❑ Ist der Bedarf für ein neues Produkt etwa durch die Vertriebs- oder Marketingabteilung oder in Form einer Ausschreibung durch Dritte formuliert, findet eine Voruntersuchung statt, die eine Aufwandsschätzung sowie eine darauf aufbauende Machbarkeitsstudie, Zeitbedarfsschätzung und Kosten/Nutzen-Analyse beinhaltet. Abhängig von deren Ergebnissen wird entschieden, ob die Entwicklung überhaupt gestartet bzw. ob und mit welchem Angebot auf die Ausschreibung reagiert werden soll.
- ❑ Ist die Entscheidung positiv bzw. der Auftrag erteilt, findet eine (detailliertere) Projektplanung statt, welche die Ablauf- und Terminplanung, die Ressourcenplanung und die Kostenplanung umfasst.
- ❑ Während der eigentlichen Projektdurchführung gehören die Leitung des/der Entwicklungsteams sowie das Projektcontrolling zu den Hauptaufgaben des Managements.
- ❑ Gutes Management führt nach Projektabschluss eine Abschlussanalyse (wrap-up) durch, bei der die guten und schlechten Vorkommnisse während des Projekts analysiert und dokumentiert werden. Andernfalls kann man davon ausgehen, dass sich beim nächsten Projekt die schlechten Dinge zwangsläufig, die guten aber nur vielleicht wiederholen.

Zwischen Vorgehensmodell und Managementaktivitäten bestehen starke Abhängigkeiten. Je nach Vorgehensmodell können die einzelnen Managementaktivitäten zu unterschiedlichen Zeitpunkten und Anlässen und mit modifizierten Zielrichtungen stattfinden. So unterscheiden sich beispielsweise Planung und Controlling eines auf dem Wasserfallmodell basierenden Softwareprojekts merklich von denen eines Projekts, in dem nach dem RUP vorgegangen wird. Andererseits muss das Vorgehensmodell auf die jeweiligen Managementbedürfnisse abgestimmt werden.

Sind Entwicklung, Management und Qualitätssicherung (vgl. Kapitel 6) vollständig in das Vorgehensmodell integriert, sprechen wir von einem *integrierten Entwicklungsprozess*. Das von Boehm vorgestellte *Spiralmodell* beschreibt einen (generischen) Rahmen für integrierte Entwicklungsprozesse [Boehm88]. In Deutschland spielt das V-Modell 97 [V-Modell97] eine wichtige Rolle. Dieses Modell regelt die Softwareentwicklung bei Aufträgen der öffentlichen Hand und wird zunehmend auch im industriellen Umfeld eingesetzt.

Werkzeugen, die von dem Hersteller der SEU bereitgestellt werden, noch weitere Werkzeuge in einer SEU zur Verfügung stehen.

Zu den Vorteilen einer SEU zählen:

- Die integrierten Werkzeuge setzen auf das in dem OMS bzw. DBMS definierte Datenmodell der SEU auf. Jedes dieser Werkzeuge kann daher die Ausgabe eines anderen integrierten Werkzeugs als Eingabe nutzen. Somit werden die Auswirkungen von Änderungen oder Erweiterungen in einer Entwicklungsstufe automatisch in den vorausgegangenen bzw. nachfolgenden Entwicklungsstufen sichtbar.
- OMS bzw. DBMS erlauben es, die Beziehungen zwischen den einzelnen Dokumenten des Entwicklungsprozesses zu verwalten (von der Produktskizze bis zum Fehlerbericht), so dass der Weg einer Anforderung von der Anforderungsspezifikation über die Entwurfsdokumente bis zur Implementierung verfolgt werden kann und bei notwendigen Änderungen die betroffenen Stellen automatisch aufgezeigt werden können.
- Das Projektmanagement kann stets auf aktuelle Projektinformationen zugreifen, wobei Projektmanagement-Werkzeuge zur Dokumentation von Projektstatus und -verlauf eingesetzt werden können.

Kapitel 8

Objektorientierte Softwareentwicklung und die UML

Nach dem knappen Überblick über die verschiedenen Aspekte des Software Engineering wenden wir uns jetzt dem Kern des Kurses zu, der objektorientierten Softwareentwicklung. Das objektorientierte Paradigma, zunächst im Rahmen von Programmiersprachen vorgestellt, hat große Hoffnungen hinsichtlich der produktiven Entwicklung qualitativ hochstehender Software geweckt. Tatsächlich trägt die Objektorientierung zu einer systematischeren und produktiveren Softwareentwickung 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* bzw. die speziellere *Generalisierung* erweitert bisherige Modellierungsansätze um ein wichtiges Konzept, das im Entwurf und bei der Implementation eine höhere Flexibilität ermöglicht und damit das Wiederverwendungspotential erhöht und die Produktivität steigert.

Allerdings haben sich die hohen Erwartungen bez. der *Wiederverwendung* bisher nur teilweise erfüllt. Einerseits bieten Klassenbibliotheken, Rahmenwerke und Entwurfsmuster dank des Vererbungskonzepts deutlich bessere Wiederverwendungsmöglichkeiten als konventionelle Entwicklungsmethoden. Jenseits dieser implementationsnahen 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 die Aktivitäten Anforderungsermittlung, Entwurf und Implementation. Insbesondere die Brüche zwischen Anforderungsermittlung und Entwurf, wie wir sie von konventionellen Methoden kennen, können hier deutlich verringert werden. Zentrale Modellkonstrukte der Anforderungsspezifikation finden sich im Entwurf und sogar in der Im-

plementation wieder, natürlich auf jeweils niedrigerem Abstraktionsniveau. Man ist dem alten Traum der Softwareentwicklung durch Verfeinerung ein Stück nähergekommen: man beginnt mit abstrakten, noch unscharfen Modellen, die man 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 C++ Programme 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 der Fahrzeuge adäquat umsetzt. Schlimmer noch, er wird oft mehr Unheil im Straßenverkehr anrichten als mit einem Golf, der mit einem bescheidenen Motor ausgerüstet ist (soweit uns bekannt ist, kommen überproportional viele Porsche in einer Kurve 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 bisher nur strukturierte Methoden und prozedurale Programmierung kennengelernt haben. In der industriellen Softwareentwicklung bilden diese Entwickler bisher noch die Mehrheit. Entwickler, die bereits mit modularen Techniken vertraut sind (vgl. [PagSix94]), kennen mit der Verkapselung 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 4.4 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.

Objektorientierte Softwareentwicklung ist ein weites Feld. Ein Kurs zu diesem Thema muss sich daher auf wenige Kernbereiche beschränken. Zunächst einmal schränken wir deshalb den Anwendungsbereich auf kommerzielle Softwaresysteme ein. Darüber hinaus fokussieren wir auf grundlegende Konzepte, die für die *Anforderungsermittlung* und den *Entwurf* relevant sind. Dabei ziehen wir als zusätzliche konzeptuelle Maßnahme die *Analyse* als weitere Aktivität zwischen Anforderungsermittlung und Entwurf ein. Wir werden uns auch mit Fragen der *Vorgehensmethodik* bei der Anforderungsermittlung, der Analyse und beim Entwurf auseinandersetzen. Insgesamt behandeln wir die technischen Kernaktivitäten mit Ausnahme der Implementation und des Tests. Erstere wird vom Kurs 01618 “Einführung in die objektorientierte Programmierung“ abgedeckt. Auch die wichtigen Managementaspekte haben keinen adäquaten Platz gefunden. Diese werden im Kurs 01895 “Management von Softwareprojekten“ behandelt.

Einen Schwerpunkt des Kurses bilden die verschiedenen Modellierungskonstrukte. Bei systematischer Softwareentwicklung übersteigt der Aufwand für die Modellierung 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. 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 13), charakterisieren die globalen Funktionalitäten der Anwendung, liefern also gewissermaßen die Funktionen zu den in strukturellen Modellen festgehaltenen Objekten. *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.

Zur Spezifikation objektorientierter Modelle verwenden wir die *Unified Modeling Language (UML)*, die im gesamten Softwareentwicklungsprozess eingesetzt werden kann [OMG97]. Entstanden ist die UML aus dem Bestreben, die Vielzahl der in den 90er Jahren entstandenen objektorientierten Notationen zusammenzufassen und ein einheitliches Metamodell für Softwareentwickler zu schaffen.

Die UML ist bestrebt, 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 ein besonders gut geeignetes Konstrukt zur Modellierung eines bestimmten Sachverhalts auszuwählen in der Lage sind. Vor diesem Hintergrund kann es auch nicht überraschen, dass die UML etliche 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 wir werden im Kurs nur einen kleinen 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. Wie oben erwähnt, werden wir Vorschläge zum methodischen Vorgehen im Rahmen der diskutierten Aktivitäten Anforderungsermittlung, Analyse und Entwurf machen.