HANSER

Leseprobe

zu

Objektorientierte Softwareentwicklung mit UML

von Peter Forbrig und Anke Dittmar

Print-ISBN: 978-3-446-47951-7
E-Book-ISBN: 978-3-446-48053-7
E-Pub-ISBN: 978-3-446-48144-2

Weitere Informationen und Bestellungen unter
https://www.hanser-kundencenter.de/fachbuch/artikel/9783446479517
sowie im Buchhandel

Inhalt

Vorwort

Sowohl die objektorientierte Programmierung als auch die objektorientierte Modellierung haben in den letzten Jahren beträchtlich an Bedeutung gewonnen. Zunächst sind die Vorteile des objektorientierten Ansatzes bei der Programmierung sichtbar geworden. Das zeigt sich auch an der Vielzahl von Programmiersprachen, die die entsprechenden Konzepte unterstützen. Mehr und mehr hat sich der Einfluss aber auch auf die frühen Phasen der Softwareentwicklung ausgedehnt. Darin ist eine Parallele zur Entwicklung des strukturierten Ansatzes zu sehen. Auch dort ging die Entwicklung von der Durchsetzung der Konstrukte zur strukturierten Programmierung in der Algorithmierung und in den Programmiersprachen aus. Später führte dies zum strukturierten Entwurf und zur strukturierten Analyse.

Mit der Unified Modeling Language (UML) hat sich eine Modellierungssprache für die objektorientierte Spezifikation herausgebildet, die große Akzeptanz in der Industrie findet. Damit ist die UML auf dem besten Weg zum Standard: Die Sprache ist nicht nur standardisiert, sondern wird in vielen Bereichen angewendet und ist Gegenstand einer großen Zahl von Werkzeugen. Viele Hersteller von CASE-Tools in diesem Bereich sind bemüht, den vollständigen Sprachumfang von UML zu unterstützen.

Aus diesen Gründen müssen auch Ausbildungseinrichtungen wie Universitäten und Hochschulen die Anwendung der Sprachelemente der UML in ihren Lehrprogrammen berücksichtigen. Dabei geht es aber nicht nur um die richtige Notation der Spezifikationen, sondern auch um die Vermittlung der zugrunde liegenden Konzepte und deren korrekte Anwendung für bestimmte Problemstellungen.

Der Inhalt des Buches basiert auf Erfahrungen von Lehrveranstaltungen zur Softwaretechnik, in denen objektorientierte Konzepte vermittelt wurden. Das Buch versucht an Hand von Beispielen, einen Einstieg in die objektorientierte Spezifikation mit UML zu ermöglichen. Es ist nicht darauf angelegt, alle Einzelheiten darzustellen, die mit der Definition von UML zusammenhängen, sondern hier sollen die wichtigsten Informationen geliefert werden, die einen Einstieg in die Projektarbeit erleichtern. Die Beispiele sind in der aktuellen Version von UML 2.5.1 formuliert.

Besonders viel Aufmerksamkeit erfährt die Spezifikation dynamischer Zusammenhänge. Hierfür wurden Videos erarbeitet, die das Verständnis der Spezifikation mit endlichen Automaten in Form von Zustandsdiagrammen erleichtern sollen. Daneben werden auch die Möglichkeiten von Aktivitätsdiagrammen aufgezeigt und der Zusammenhang zu den besonders bei Banken und Versicherungen sehr beliebten Ereignis-Prozess-Ketten hergestellt.

Entwurfsmuster sind ein weiteres wichtiges Konzept. Durch die Wiederverwendung von Software auf einem völlig neuen Niveau sind sie ein Schlüssel für eine erfolgreiche Softwareentwicklung. Die Idee der Entwurfsmuster (engl. Design Patterns) wird vorgestellt und die Notationsmöglichkeiten in UML werden diskutiert. Auch Konsequenzen für die Programmierung in Java, Python, C# und Eiffel werden aufgezeigt. Die Quelltexte für jede Sprache sind auf der Webseite des Verlages verfügbar, sofern die Sprache dies ohne größeren Aufwand ermöglichte.

Auch der textuellen Spezifikation in OCL (Object Consraint Language) widmet sich ein Abschnitt. Bei OCL handelt es sich um eine textuelle Teilsprache von UML, die notwendig ist, wenn die grafischen Ausdrucksmittel nicht genügend Aussagekraft besitzen. Für die praktische Anwendung von UML in größeren Projekten hilft OCL, Mehrdeutigkeiten zu vermeiden.

Benutzerorientierte Entwicklungsansätze wurden im neuen Kapitel 4 aufgegriffen. Sie müssen in bestehende Vorgehensweisen des Softwareengineering, wie agile Softwareentwicklung, integriert werden, um die Perspektive der Benutzer stärker zu beachten.

Modelle der UML lassen sich gut mit Skizzen, Stories oder dem aus der Softwareergonomie bekannten Konzept der Personas verbinden. Es wird beispielsweise gezeigt, wie durch die Nutzung der Ideen von Personas die Spezifikation von Anwendungsfalldiagrammen ausdrucksstärker gestaltet werden kann.

Von den Fähigkeiten der Beteiligten und den konkreten Projektbedingungen hängt es ab, in welchem Maße beispielsweise Anwendungsfallmodelle, Klassendiagramme, Zustandsübergangsautomaten, Aufgabenmodelle, Szenarien, Personas, Skizzen oder Prototypen sinnvoll genutzt werden können.

Für die in der agilen Softwareentwicklung gebräuchlichen User Stories wird die Idee der Story-Splitting-Patterns diskutiert, die ein Pendant zu den Design Patterns der „Gang of Four“ darstellen.

Das Buch ist folgendermaßen aufgebaut:

- **Kapitel 1** gibt eine Einführung in die wichtigsten Grundbegriffe der Objektorientierung.
- **Kapitel 2** stellt die Sprache UML vor und ergänzt die verschiedenen Diagramme stets mit einer Reihe von Anwendungsbeispielen.
- **Kapitel 3** beschäftigt sich mit den Problemen der Softwarespezifikation bezogen auf den gesamten Lebenszyklus und stellt unterstützende Techniken zur Ermittlung von Anforderungen vor. Außerdem werden Entwurfsmuster und Modelltransformationen sowie deren Werkzeugunterstützung etwas genauer betrachtet.
- **Kapitel 4** wurde zusammen mit Frau Dr. Anke Dittmar geschrieben. Es widmet sich der kollaborativen Analyse und dem kollaborativen Design. Heutzutage müssen komplexe interaktive Systeme in interdisziplinären Teams gestaltet werden. Dazu ist die Einbeziehung der Expertise aller Beteiligten notwendig. Speziell die späteren Benutzer des Systems sollte man frühzeitig mit in die Entwicklung einbeziehen.

Mit den diskutierten Beispielen und Modellen hoffen wir, den Lesern Möglichkeiten aufgezeigt zu haben, wie Modelle der UML genutzt und mit weiteren Sichten kombiniert werden können.

Dieses Buch ermöglicht in der aktualisierten Fassung einen optimalen Einstieg in die Softwareentwicklung auf der Basis von UML. Und der Autor hofft, dass das Buch nicht nur Anfängern beim Einstieg in das Thema begleitet, sondern auch Fortgeschrittene neue Erkenntnisse gewinnen. Konstruktive Hinweise zur Verbesserung der Darstellung sind herzlich willkommen (*peter.forbrig@uni-rostock.de*).

Beim Hanser Fachbuchverlag möchte ich mich ganz herzlich für die Unterstützung bedanken. Mein besonderer Dank geht an Brigitte Bauer-Schiewek, Kristin Rothe, Irene Weilhart und Jürgen Dubau.

Rostock, im Januar 2024

Zusatzmaterial zum Buch

Zu diesem Buch stehen Ihnen weitere Inhalte digital zur Verfügung:

- Neben Korrekturen zum Manuskript befinden sich dort Interessante Links zu UML Systemunterlagen, zu im Buch diskutierten Werkzeugen wie ArgoUML, USE, CTTE und zur Unterstützung von Patterns, zu Lehrmaterialien zu Entwurfsmustern und zu Videos mit animierten Zustandsautomaten.
- Als Downloads finden Sie alle Abbildungen des Buches, die Quelltexte der Programme aus dem Buch sowie die entsprechenden Programme in Java, C#, Python und Eiffel, sowie Lösungen zu den im Buch formulierten Aufgaben. Außerdem gibt es dort Links zum Herunterladen von Programmierumgebungen, eigenen Werkzeugen und einem Word-Template zur textuellen Beschreibung von Anwendungsfällen.

Gehen Sie dazu einfach auf

plus.hanser-fachbuch.de

und geben Sie dort diesen Code ein:

1 Grundbegriffe der objektorientierten Softwareentwicklung

1.1 Einführung

Mit der Objektorientierung ist in den letzten Jahren ein Paradigmenwechsel in der Softwareentwicklung eingetreten, der sich von der Implementation über den Entwurf bis zur Analyse in die sehr frühen Phasen durchgesetzt hat. Die zunächst gültige Trennung von Daten und Funktionen wurde überwunden. David Parnas [1.6] propagierte Anfang der 70er Jahre die Nutzung von Datenkapseln, bei denen der Zugriff auf die Daten nur über eine Menge bereitgestellter Funktionen, die sogenannte Schnittstelle, ermöglicht wurde. Es hat eine ganze Weile gedauert, bis sich diese Idee in der Praxis der Softwareentwicklung durchgesetzt hat. Um eine Vielzahl von derartigen Datenkapseln schnell erzeugen zu können, folgte später die Idee der Programmierung von abstrakten Datentypen. Auch hier trat der Erfolg nicht sofort ein. Erst die Einordnung dieser Datentypen in eine Hierarchie, die über Vererbungsmechanismen verfügt, führte zu einem durchgreifenden Erfolg. Dieser Ansatz wurde als *objektorientiert* charakterisiert. Er ist eng mit den Begriffen von *Klasse* und *Objekt* verbunden.

Ein Objekt wird durch Eigenschaften und Fähigkeiten charakterisiert. Die Eigenschaften beschreiben den aktuellen Zustand des Objektes, und die Fähigkeiten stellen Tätigkeiten dar, die auf das Objekt angewendet werden können, um seine Eigenschaften zu verändern. So kann ein Füllfederhalter durch die Farbe der Tinte, mit der er gefüllt wurde, beschrieben werden. Diese Eigenschaft ist durch das *Entleeren* und nachfolgendes *Füllen* änderbar. Damit sind auch schon zwei Fähigkeiten genannt, die mithilfe des Füllfederhalters ausgeführt werden können. Die wichtigste Fähigkeit ist natürlich das *Schreiben*. Sie ist der Grund, warum man sich den Federhalter zulegt. Eine weitere Eigenschaft des Schreibgerätes beschreibt den Bereitschaftszustand. Es kann durch eine Schutzkappe *verschlossen* oder *unverschlossen* sein. Eine Veränderung des Bereitschaftszustandes erfolgt durch die Tätigkeiten *Öffnen* und *Schließen*, die weitere Fähigkeiten darstellen.

Bei der objektorientierten Analyse wird von den Objekten ausgegangen, die in der realen Welt existieren. Durch geeignete Abstraktion wird aus einem realen Objekt ein Objekt eines Modells. Dabei wird besonderes Augenmerk auf Charakteristika gelegt, die im Zusammenhang mit einer bestimmten Aufgabe von Interesse sind. Eine Modellierung ohne ein bestimmtes Ziel ist nicht möglich, weil die Anzahl der Charakteristika fast ins Unendliche steigt. Eigenschaften und Fähigkeiten werden durch *Attribute* und *Methoden* beschrieben.

Die Problematik der Modellierung der Charakteristika wird sicher am Beispiel deutlich. Gegenstand des Interesses sei eine Person. Da keine konkrete Zielvorgabe existiert, ist deren Modellierung praktisch nicht möglich. Ist die Haarfarbe von Interesse? Sind die Kinderkrankheiten wichtig? Welche Bedeutung haben Hobbys? Niemand kann das ohne präzisere Zusatzinformationen genau wissen. Für ein Programm zur Beratung von Farbvorschlägen für einen Friseur ist die momentane Haarfarbe der zu betreuenden Kundin von großer Bedeutung. Für die Diagnose von Erkrankungen sind die bereits durchlaufenen Kinderkrankheiten sicher wichtig. Ein System, das Literatur für einen Kunden empfehlen soll, möchte sicher auf die Hobbys der betreffenden Person zurückgreifen.

Hier sei eine Person im Kontext einer Universität exemplarisch modelliert. Zunächst wird sie als Felix identifiziert, der am 17.11.2007 geboren wurde und als Einkommen über ein Stipendium verfügt. Als besondere Fähigkeit fällt nur auf, dass er lernen und feiern kann. Diese Informationen können wie in Bild 1.1 dargestellt grafisch repräsentiert werden.

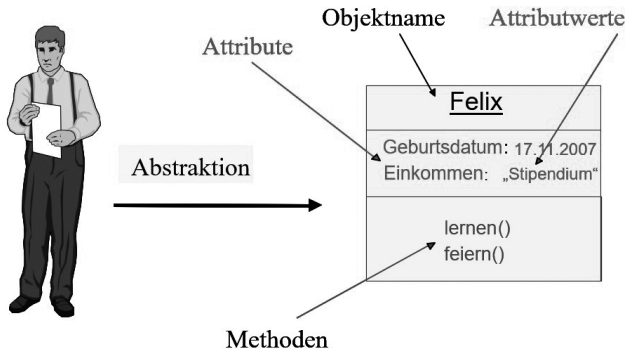


Bild 1.1 Modellierung eines Objektes

In der grafischen Repräsentation eines Objektes werden die Eigenschaften (`Geburtsdatum`, `Einkommen`) und die Fähigkeiten (`lernen`, `feiern`) getrennt dargestellt.

Fallen weitere Personen auf, die ähnliche Eigenschaften und Fähigkeiten besitzen, so sollten diese auch als Gruppe modellierbar sein. Dafür hat sich der schon in der Schule bekannte Begriff einer *Klasse*, der eine Reihe von Schülern beschreibt, die in etwa den gleichen Ausbildungsstand besitzen, eingebürgert. In diesem Zusammenhang betitelt der Begriff *Klasse* eine Sammlung von Objekten mit gleichen Charakteristika. Dabei handelt es sich um Objekte, die die gleichen Eigenschaften (Attribute) und Fähigkeiten (Methoden) besitzen.

Für obiges Beispiel weisen die Charakteristika von Felix darauf hin, dass die Klasse als Student bezeichnet werden kann. Dabei ist das Attribut entscheidend, welches das Einkommen charakterisiert. Studenten verfügen über ein Stipendium. Auch die beobachteten Fähigkeiten wie `lernen` und `feiern` stehen zu der Einschätzung nicht im Widerspruch.

Von dem Objekt Felix kann daher auf die Klasse Student abstrahiert werden. Das grafische Symbol für Objekte und Klassen ist gleich. Auch die Methoden werden identisch dargestellt. Nur an der Stelle des unterstrichenen Namens eines Objektes steht der Name einer Klasse (ohne Unterstrich). Die Darstellung der Namen von Attributen ist bei Objekten und

Klassen auch gleich. Nur haben Attribute von Objekten einen Wert, während das bei Klassen noch nicht vorliegt. Bei ihnen ist nur der Typ der Attribute bekannt (bei Programmiersprachen findet man Erweiterungen, die nicht dieser klassischen Sicht entsprechen).

Bild 1.2 zeigt die Darstellung einer Klasse, die mehrere Objekte repräsentiert.

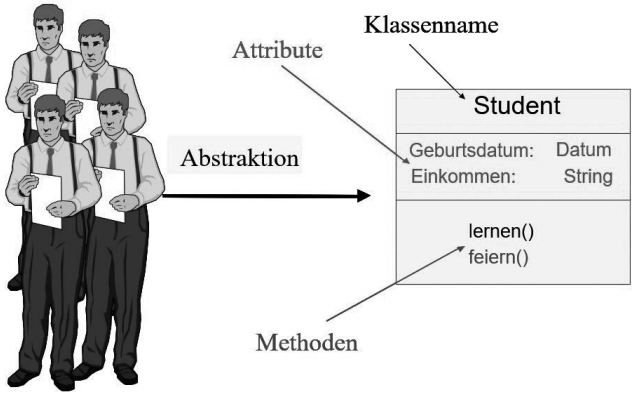


Bild 1.2 Modellierung einer Klasse

Eine recht anschauliche Abstraktion des Prozesses der Modellierung wurde von Heeg [1.4] gegeben, die hier aufgegriffen und etwas modifiziert in Bild 1.3 dargestellt wird.

Zunächst wird versucht, ein Problem der realen Welt aus einem subjektiven Blickwinkel zu modellieren. Der subjektive Blick des Modellierers wird durch das Auge symbolisiert. Er ist durch das Ziel der Modellierung beeinflusst. Dabei werden subjektiv Personen, Phänomene oder reale Dinge erfasst, die für das Modellierungsziel relevant sind. Deren Bezeichnungen werden in einem Glossar verwaltet. Dabei ist bereits auf Synonyme (unterschiedliche Bezeichnungen für gleiche Sachverhalte – z. B. Behälter, Ablage und Container) und Homonyme (gleiche Bezeichnungen für unterschiedliche Sachverhalte – z. B. Schloss) zu achten. In unbekannten Anwendungsbereichen ist das nicht immer ganz einfach.

Ist das Glossar ausreichend mit Informationen gefüllt, kann mit der Modellierung von Klassen begonnen werden. Zu den zunächst nur durch Begriffe charakterisierten Objekten werden Eigenschaften und Fähigkeiten ermittelt, die mit dem Modellierungsziel in Zusammenhang stehen.

Auf der Basis der modellierten Klassen findet die Entwicklung von Programmen statt, in denen Instanzen der Klassen erzeugt werden. Während der Laufzeit der Programme erfolgt die Erzeugung der entsprechenden Objekte einer Klasse, die durch Variablen repräsentiert werden. Diese Variablen ermöglichen die Beschreibung des Zugriffs auf die Objekte bereits während der Programmierung.

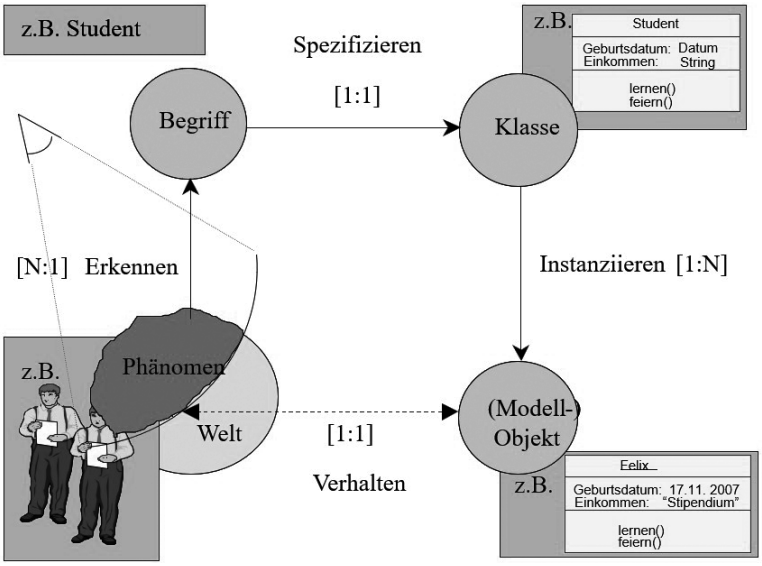


Bild 1.3 Objektorientierte Sichtweise der Softwareentwicklung

Im Allgemeinen entsprechen mehrere Objekte oder Phänomene der realen Welt einem Begriff, zu dem dann eineindeutig eine Klasse gleichen Namens existiert. Während der Laufzeit eines Programms kann aus einer Klasse eine Vielzahl von Objekten erzeugt werden. Ist die Modellierung korrekt gelungen, dann sollte das beobachtbare Verhalten der Objekte im Modellbereich dem beobachtbaren Verhalten in der realen Welt unter Einschränkung auf das Modellierungsziel exakt entsprechen. Es sollte jedem Modellobjekt eineindeutig ein Objekt aus der realen Welt zugeordnet werden können.

Genauso wie in der realen Welt kommunizieren Objekte in der Modellwelt über Botschaften.

An der Hochschule wird eventuell eine Lehrkraft einem Studenten die Botschaft `lernen` schicken, wenn sie der Meinung ist, dass diese Aufforderung notwendig ist. Bild 1.4 stellt diesen Sachverhalt grafisch dar. Ein Student wird entsprechend auf die Botschaft reagieren und seine Methode `lernen` aktivieren.

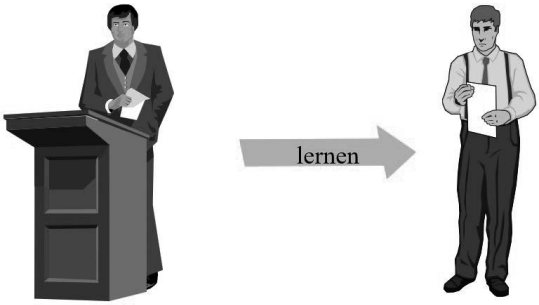


Bild 1.4 Übermittlung einer Botschaft in der Realität

Genauso sieht es auch in der Modellwelt aus (Bild 1.5). Trifft eine Botschaft bei einem Objekt ein, so wird überprüft, ob sie für das jeweilige Objekt von Bedeutung ist, ob eine Interpretation möglich ist.

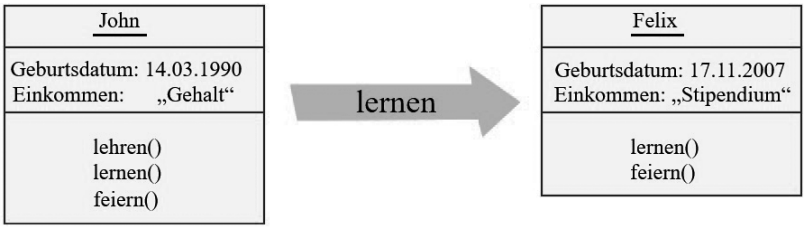


Bild 1.5 Kommunikation von Objekten in der Modellwelt

Mitunter treffen Botschaften ein, bei denen dem jeweiligen Adressaten nicht klar ist, wie er darauf reagieren soll. In der Modellwelt ist dies der Fall, wenn zu einer eintreffenden Nachricht keine gleichnamige Methode existiert.

In der Realität ist mitunter nicht so genau festzustellen, ob eine Botschaft verstanden wurde oder nicht. In der Modellwelt wird die Botschaft nicht verstanden, wenn zu einer eintreffenden Nachricht keine gleichnamige Methode existiert. In dem Falle reagiert ein Objekt nicht.

Ein Objekt reagiert nur auf eine Botschaft, wenn eine Methode gleichen Namens existiert.

Von den Programmiersprachen wird meistens gefordert, dass man nur Nachrichten an Objekte verschickt, auf die der Adressat auch reagieren kann. Es wird bei der Analyse von Programmen bereits überprüft, ob die Versendung von Nachrichten sinnvoll ist. Ein Compiler signalisiert meist schon einen Fehler. Das gilt für Sprachen wie Pascal, Java, C++ oder C#. Bei der Nutzung eines Interpreters kommt es erst zur Laufzeit der Programme zu Fehlermeldungen. Das kann man beispielsweise bei der Nutzung der Entwicklungsumgebung der Sprache Python beobachten. Es wird also nicht einfach ignoriert, dass die Nachricht nicht verstanden wird.

Programmiersprachen fordern, dass Objekte zu jeder eintreffenden Botschaft eine entsprechende Methode besitzen.

Es gibt damit einen Unterschied zwischen der Realität und den in Programmiersprachen spezifizierten Modellwelten.

Nicht alle Studenten haben im Detail das gleiche Verhalten. Man kann sie beispielsweise nach ihren Studiendisziplinen klassifizieren, die unterschiedliche Fähigkeiten fordern.

Die allgemeinen Eigenschaften und Fähigkeiten eines Studenten haben alle Studierenden, sie besitzen aber mitunter noch weitere Fähigkeiten wie Singen oder Tanzen. Was diese Fähigkeiten im Einzelnen sind, soll hier zunächst erst einmal nicht interessieren. Bild 1.6 stellt grafisch dar, dass es verschiedene spezielle Klassen gibt. Sie haben die Eigenschaften

2 UML – Unified Modeling Language

2.1 Entwicklung der Sprache

UML (Unified Modeling Language) ist nach Aussage ihrer Entwickler eine Sprache zur Spezifikation, Visualisierung, Konstruktion und Dokumentation von Software. Ursprünglich aus den Ideen der drei Gurus Booch, Jacobson und Rumbaugh zusammengestellt, hat sie sich durch die Beteiligung wichtiger Industriepartner mehr und mehr als wirklicher Standard für die Modellierung objektorientierter Spezifikationen etabliert.

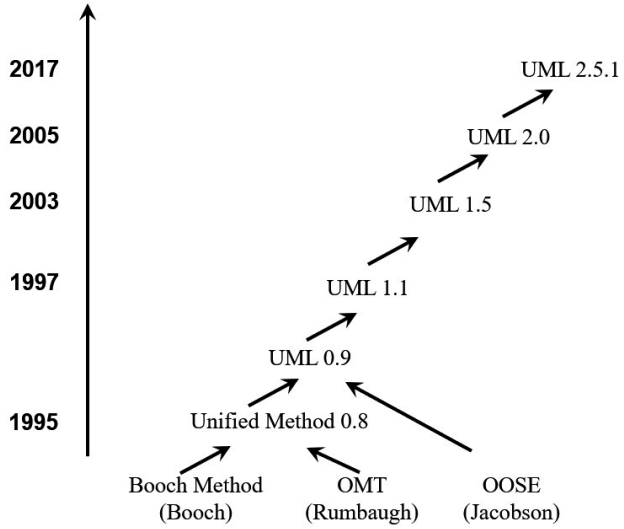


Bild 2.1 Überblick zur Geschichte von UML

Die Entwicklung von UML kann bis 1994 zurückverfolgt werden, als Booch und Rumbaugh begannen, ihre Entwicklungsmethoden zusammenzuführen. Beide Ansätze waren zu dieser Zeit weltweit schon stark verbreitet, wobei die Stärken von Booch in der Modularisierung und im Entwurf lagen, während die Stärken Rumbaughs besonders bei der objektorientierten Modellierung bestanden. Den in beiden Ansätzen gemeinsamen Konzepten galt

es, eine einheitliche Notation zu geben. Eine erste Version der „Unified Method“ wurde im Oktober 1995 veröffentlicht. Zu dieser Zeit wurde die Zweiergruppe durch Ivar Jacobson erweitert, der mit seinem OOSE (Object-Oriented Software Engineering) den Gesichtspunkt des Anwendungsfalls in die Methodendiskussion einbrachte.

Gemeinsam legten sie 1996 eine Spezifikation von UML in der Version 0.9 vor, mit der sie sich wohl auch den Namen die „drei Amigos“ verdient haben, unter dem sie heute oft zitiert werden.

Mit dem gemeinsamen Bericht von 1996 sahen auch viele Unternehmen und Organisationen in einer einheitlichen Modellierungssprache UML eine strategische Basis für ihre Arbeit. Auf einen Aufruf der Object Management Group (OMG), einer Vereinigung von Unternehmen mit Bemühungen zur Standardisierung in der Softwareentwicklung, zur Einreichung von Veränderungsvorschlägen gab es große Resonanz. Daraufhin etablierte die Firma Rational, bei der die „drei Amigos“ tätig waren, ein Konsortium zur Definition der Sprache UML.

Dieses Konsortium umfasste bedeutende Firmen wie DEC, HP, IBM, Microsoft, Oracle und Unisys. Es modifizierte den Sprachvorschlag und erarbeitete Berichte zur genauen Definition von UML. Im Januar 1997 reichten dann weitere Firmen einen Vorschlag bei der OMG ein. Diese Gruppe wurde in das Konsortium aufgenommen, und ihre Vorschläge führten zur Sprachentwicklung UML 1.1. Die weiteren Revisionen der Unterlagen erfolgten bis zur Spezifikation von UML 1.5 schrittweise. Danach wurde eine vollständige Überarbeitung der Sprache vorgenommen und UML 2.0 entwickelt. Mittlerweile gibt es bereits Version 2.5.1. Die letzten Änderungen waren hauptsächlich Präzisierungen und nicht mehr so umfangreich. Die Version 2.5.1 wurde genau 20 Jahre nach der ersten Standardisierung von UML eingereicht und bis 2023 noch nicht weiter verändert.

Das immer größere Interesse an UML hat auch seine Nachteile. Immer mehr unterschiedliche Ansichten beeinflussten die Diskussion über die Entwicklung der Sprache. So hat es mehrere Jahre gedauert, bevor der Standard durch alle Gremien bestätigt wurde. Die Firma Rational ist in der Zwischenzeit durch IBM aufgekauft worden.

UML ist nicht abgeschlossen, sondern offen für neue Konzepte. Die Sprache stellt in sich bereits Erweiterungsmöglichkeiten zur Verfügung. Zukünftige Entwicklungen sind damit integrierbar. Bei der OMG wurde auch eine „Revision Task Force“ (RTF) installiert, die Ansprechpartner für die Öffentlichkeit ist, um Fehler in den Spezifikationen zu beheben.

Welche anderen Konzepte und Methoden hauptsächlich beim Entwurf der Grundprinzipien von UML eine Rolle spielten, ist aus Bild 2.2 zu entnehmen, die aus einer Präsentation von Grady Booch selbst stammt.

Diese Abbildung stellt vereinfacht Einflussfaktoren auf die Entwicklung von UML dar. Dabei spielen die Arbeiten von Booch [2.3], Jacobson [2.17] und Rumbaugh [2.24] eine herausragende Rolle. Sie sind in der Abbildung links mit grauen Pfeilen dargestellt. Natürlich basieren die Konzepte von UML auf weiteren Arbeiten wie beispielsweise der Idee der Kapselung nach Parnas oder anderen objektorientierten Ansätzen. Diese sind aus Übersichtlichkeitsgründen nicht dargestellt.

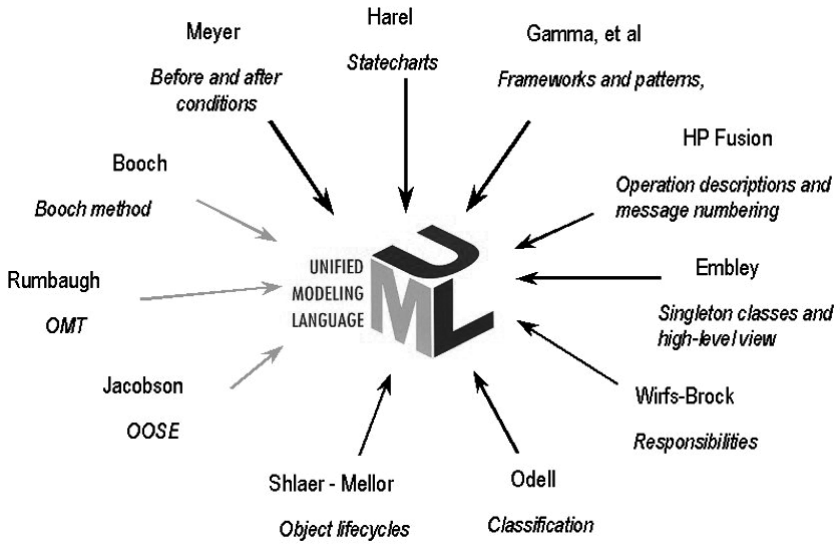


Bild 2.2 Einflüsse auf die Entstehung von UML (nach Booch)

Einige Konzepte wie die *Statecharts* (Zustandsdiagramme) nach Harel [2.14] sind nur leicht modifiziert in UML übernommen worden. Sie dienen zur Spezifikation dynamischer Vorgänge und sind daher unter anderem für die Spezifikation der Lebenszyklen von Objekten geeignet. In diesem Buch werden sie deshalb in zwei Abschnitten ausführlich behandelt. Über die Grundlagen finden sich einige Ausführungen bereits in Abschnitt 1.2.3. Mit den entsprechenden Sprachelementen der UML befasst sich Abschnitt 2.5.1.

Um die Semantik der Zustandsdiagramme zu verdeutlichen, wurden auch einige Animationen erstellt, die auf der Webseite zu diesem Buch (im Zusatzmaterial auf www.plus.hanser-fachbuch.de) bereitgestellt werden. Sie können herangezogen werden, um das Verständnis für die Spezifikationen zu erhöhen.

Eigene Sprachelemente wurden auch für Design Patterns (Entwurfsmuster) in UML vorgesehen. Sie werden in Abschnitt 2.3.9 erläutert. Die Anwendung von Patterns wird dann später noch in Kapitel 3 diskutiert. Entwurfsmuster spielen bei der Softwareentwicklung eine immer wichtigere Rolle. Sie ermöglichen eine neue Art der Wiederverwendung. Dem Leser und der Leserin wird das Studium der entsprechenden Literatur (z. B. [2.13]) unabhängig von der Nutzung von UML ans Herz gelegt. In Bünnig et al. [2.4] wird eine Programmiersprache auf der Basis von Design Patterns vorgeschlagen. Dazu findet man mehr auf der Webseite zu diesem Buch.

Vor- und Nachbedingungen hat Meyer sehr erfolgreich in den Entwurf seiner Programmiersprache Eiffel [2.20] integriert. Damit hat er einen wichtigen Beitrag zur Etablierung einer sicheren Softwareentwicklung geleistet. Das Konzept basiert auf den Arbeiten von Hoare [2.15]. Dieser hat sich sehr intensiv mit dem Nachweis der Korrektheit von Programmen beschäftigt. Zusicherungen spielen dabei eine wichtige Rolle. Der Nachweis der Korrektheit

der Nachbedingungen aus der Korrektheit der Vorbedingungen wird zum Beweis der Korrektheit von Programmen genutzt. Die Möglichkeit der Notation von Zusicherungen ist in UML integriert. Ein Nachweis ihrer Korrektheit ist allerdings nicht möglich, da die einzelnen Sprachelemente nicht ausreichend formalisiert sind.

Auf weitere Einflussfaktoren soll hier nicht weiter explizit eingegangen werden. Der Leser und die Leserin seien auf die entsprechende Literatur verwiesen, um den Grad des Einflusses auf die Sprachgestaltung von UML selbst zu beurteilen (z. B. [2.8], [2.19], [2.20]).

Die eine kompakte Darstellung der UML findet man in [2.1], [2.2] und [2.9].

Der folgende Abschnitt widmet sich zunächst dem anwendungsfallorientierten Ansatz und den Modellen, die dafür bei der Analyse eines Problems zur Verfügung stehen. Danach werden Diagramme vorgestellt, die zur Beschreibung statischer Zusammenhänge geeignet sind, bevor sich ein weiterer Abschnitt mit der Spezifikation dynamischer Eigenschaften beschäftigt.

■ 2.2 Anwendungsfallmodelle

Softwareentwicklung beginnt bei der Anforderungsanalyse mit dem Ziel, die wirklichen Bedürfnisse der Anwender und Auftraggeber zu ermitteln. Basierend auf den intuitiven Hauptzielen eines Projektes sind präzise Spezifikationen zu entwickeln. Dieser Prozess kann nur unter Einbeziehung der Anwender und in einer gut strukturierten Form erfolgreich gemeistert werden. Dabei hat die *Analyse der Anwendungsfälle* eine entscheidende Bedeutung.

Diese Analyse wird nicht nur bei der objektorientierten Softwareentwicklung hervorgehoben. Bei der strukturierten Analyse ist das Kontextdiagramm Ausgangspunkt der Betrachtungen. Dabei wird die Kommunikation zwischen Umgebung und System in Form von Datenflüssen zu Datenquellen und Datensenken modelliert.

Bei der Analyse betrieblicher Zusammenhänge greift man auf die Geschäftsprozesse zurück, die Ziele eines Unternehmens unterstützen.

Mit dem Aufkommen der objektorientierten Analysemethoden [2.3] wurde der funktionale Aspekt der Anwendungen zunächst etwas in den Hintergrund gerückt. Es war ein Verdienst von Jacobson [2.17], den Anwendungsfallaspekt in die Welt der objektorientierten Modellierung zu integrieren. Der Versuch der Integration von Datenflüssen in die Spezifikationsmöglichkeiten von OMT durch Rumbaugh [2.24] war zunächst nicht von diesem anhaltenden Erfolg geprägt. Das Buch über die Spezifikationssprache OMT war zwar ein weltweiter Bestseller, die Datenflüsse fanden aber nicht den Weg in die UML.

Definition 2.1: Szenario

Ein Szenario ist eine spezifische Folge von Aktionen, die zur Verdeutlichung des Verhaltens eines Systems dient. ■

Szenarien sind Ausgangspunkt jeglicher Spezifikation, die menschliche Handlungen beschreiben. Sie sind die Beispiele aus dem Anwendungsbereich, die als Ausgangspunkt für eine Abstraktion dienen. Szenarien können konkret oder abstrakt sein.

Definition 2.2: Konkretes Szenario

Ein konkretes Szenario ist eine spezifische Folge von Aktionen, die von konkreten Akteuren (Personen oder Systemen) unter konkreten Randbedingungen durchgeführt werden.

So ist beispielsweise die Buchung einer Reise für einen konkreten Touristen Felix mit dem konkreten Preis von 1.000 Euro an den Zielort Berlin mit dem Anreisedatum 3. Mai 2023 und weiteren konkreten Randbedingungen ein konkretes Szenario. Ein solches Szenario ist die Basis für spätere Testfälle des entwickelten Softwaresystems. Es entspricht genau einem Durchlauf durch dieses System. Erfolgt die Beschreibung der Folge von Aktionen etwas weniger konkret durch allgemeine Akteure wie beispielsweise „ein Tourist“, dann spricht man von abstrakten Szenarien.

Definition 2.3: Abstraktes Szenario

Ein abstraktes Szenario ist eine spezifische Folge von Aktionen, die von Akteuren (im Sinne von Rolle) unter abstrakten Randbedingungen durchgeführt werden.

Abstrakte Szenarien können Zusicherungen (Constraints) enthalten, die für konkrete Szenarien exakte Werte annehmen. Um eine Reise zu buchen, muss ein Tourist sein Reiseziel und sein Reisedatum angeben. Danach erhält er ein Angebot, das er innerhalb von drei Tagen bezahlen muss, um die Reise zu buchen.

Aus einer Reihe solcher abstrakten oder konkreten Szenarien kann eine allgemeinere (noch abstraktere) Beschreibung abgeleitet werden. Dabei sind alle Eventualitäten und Besonderheiten zu berücksichtigen. Für das dargestellte Beispiel bedeutet dies, dass beschrieben wird, wie sich das Reisebüro verhält, wenn dem Touristen der Preis des Angebotes zu hoch ist.

Szenarien können dazu genutzt werden, um das Anwendungsgebiet zu charakterisieren und damit für alle Beteiligten erschließbar zu machen. Umgekehrt können sie auch zur Verdeutlichung der abstrakten Beschreibung dienen. Sie stellen in diesem Fall ein konkretes Beispiel dafür dar, was mit einer abstrakten Spezifikation gemeint ist. Eine allgemeine Beschreibung wird durch Szenarien erläutert und damit verständlicher. So kann zur Verdeutlichung der allgemeinen Beschreibung, was bei der Buchung einer Reise alles zu beachten ist, das konkrete Beispiel der Buchung einer speziellen Reise beitragen.

Es ist offensichtlich, dass zwischen allgemeiner Beschreibung und den abstrakten und konkreten Szenarien keine Widersprüche existieren dürfen.

Im Folgenden ist mit Szenario immer ein abstraktes Szenario gemeint. Nur bei einem konkreten Szenario wird dies explizit hervorgehoben.

3 Von der Analyse zur Implementierung

3.1 Überblick

Dieses Kapitel widmet sich dem Entwicklungsprozess eines Softwareproduktes von der Analyse bis zur Implementierung. Dabei werden Grundprinzipien, Methoden und Techniken vorgestellt, die diesen Prozess unterstützen können. Aus der Vielfalt der vorhandenen Erfahrungen muss sich ein Entwicklungsteam diejenigen heraussuchen, die für das entsprechende Projektvorhaben Erfolg versprechend sind. Einige Methoden und Techniken sind als alternativ zu betrachten, andere können sich aber auch ergänzen.

Zunächst steht die Problematik der Analyse im Mittelpunkt, bevor auf den Entwurf und die Implementation etwas genauer eingegangen wird. Die Anforderungsanalyse ist für den Erfolg oder Misserfolg eines Projektes von ganz entscheidender Bedeutung. Hier werden die Weichen dafür gestellt, was in den folgenden Projektphasen realisiert wird. Anforderungen, die übersehen oder falsch aufgenommen werden, verursachen im späteren Verlauf der Projektentwicklung sehr große Aufwendungen.

Im Verlauf des gesamten Kapitels erfolgt stets ein Seitenblick auf eine mögliche Werkzeugunterstützung der vorgestellten Methoden, die für die Akzeptanz für den praktischen Einsatz mehr und mehr an Bedeutung gewinnen. Zunächst aber noch einige einführende Bemerkungen zum gesamten Entwicklungsprozess.

Um den Prozess der Softwareentwicklung zu beschreiben, wurden Modelle erarbeitet, die sogenannten Lebenszyklusmodelle.

Definition 3.1: Softwarelebenszyklus

Der Softwarelebenszyklus ist die Menge der einzelnen Tätigkeiten, die während der Prozesse der Entwicklung und Anwendung von Software in einer vorgegebenen Reihenfolge ablaufen und sich technologisch bedingt oder bei veränderten Ausgangsbedingungen zyklisch wiederholen. Er ist in abgegrenzte Teilprozesse, die sogenannten Phasen unterteilt, um den Arbeitsablauf einer Aufgabenstellung arbeitsteilig inhaltlich, technologisch, leitungsmäßig und organisatorisch effektiv zu beherrschen.

Softwareentwicklung ist damit eine Folge von abgegrenzten Phasen. Jede Phase liefert ein abgeschlossenes Ergebnis. Der Übergang von einer Phase zur nächsten beginnt erst nach der Qualitätskontrolle der abzuschließenden Phase.

Der Grundzyklus der Softwareentwicklung lässt sich in folgende Phasen unterteilen:

- **Analysieren**
  Analyse des Basisprozesses

  Ergebnis: Aufgabenstellung zur Softwareentwicklung
- **Spezifizieren**
  Dokumentation der Funktionen des Softwareproduktes

  Ergebnis: funktionelle Spezifikation
- **Entwerfen**
  Dokumentation der Problemlösung

  Ergebnis: logische Gliederung der Funktionen und Daten (fachlicher Entwurf) und Ablaufstruktur (programmtechnischer Entwurf)

  Ergebnis: Entwurfsspezifikation
- **Implementieren**
  Codierung der Problemlösung in einer Programmiersprache, Übersetzen und Verbinden

  Ergebnis: lauffähiges Softwareprodukt
- **Testen**
  Verwendung von vorbereiteten Testmitteln, Testverfahren und Testdaten zum Vergleich von Soll- und Ist-Eigenschaften des Programms – Fehlerfindung

  Ergebnis: Fehlerprotokoll
- **Nutzen**
  Anwendung des Softwareproduktes, Nachweis des stabilen Dauerbetriebs, Nutzereinweisung.

  Ergebnis: laufendes Softwareprodukt
- **Warten**
  Änderung des fertigen Softwareproduktes zur weiteren Nutzbarkeit (Anpassung an neue Bedingungen, Mängelbeseitigung)

  Ergebnis: aktualisiertes Softwareprodukt (samt Spezifikationen)

Als man sich in den 1960er Jahren intensiver mit den Problemen der Entwicklung anwendungsfähiger Software beschäftigte, war man noch der Meinung, dass ein Problem nur lange genug analysiert werden muss, um zu einer vollständigen Anforderungsanalyse zu gelangen. Die Hauptprobleme wurden damals in einer nicht korrekt durchgeführten Analyse gesehen. Man legte das Augenmerk auf verbesserte Spezifikationen in diesem Bereich, ließ aber das Problem der Kommunikation zwischen Entwicklern und Anwendern außer Acht. Erst später erkannte man, dass Formulierungen von Anwendern das eine und ihre wirklichen Vorstellungen etwas anderes sein können. Formal zu beweisen, dass eine Implementation den Anforderungsdokumenten entspricht, reicht für eine erfolgreiche Zusammenarbeit nicht aus. Es bedarf auch einer ständigen Kommunikation mit dem Anwender und einer gleichzeitigen Revision der erarbeiteten Dokumente. Auch die Probleme der Wartung wurden damals noch völlig unterschätzt. Daher war das erste Lebenszyklusmodell zur Beschreibung der Softwareentwicklung noch linear.

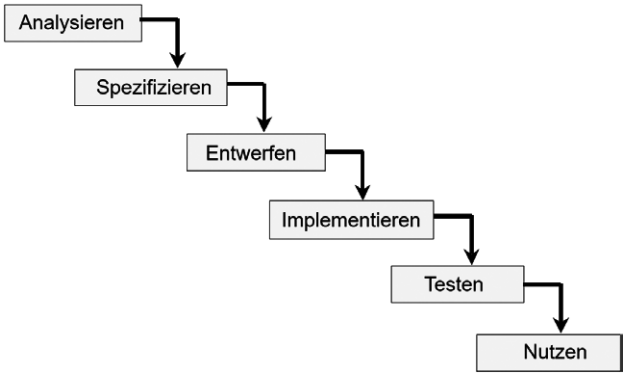


Bild 3.1 Klassisches Wasserfallmodell

Das Wasserfallmodell basiert auf den Vorstellungen eines geradlinigen Entwicklungsprozesses von der Analyse bis zur Nutzung. Die Geschichte zeigte jedoch, dass auch Projekte mit einer sehr gründlichen Analyse Probleme bei der Nutzung bekamen. Im Verlaufe der Projektentwicklung ergeben sich einerseits neue Erkenntnisse im Anwendungsgebiet, die bei der Analyse noch nicht berücksichtigt werden konnten. Andererseits entstehen bei der Analyse Kommunikationsprobleme zwischen Anwendern und Entwicklern, die erst zutage treten, wenn lauffähige Softwarebausteine vorhanden sind.

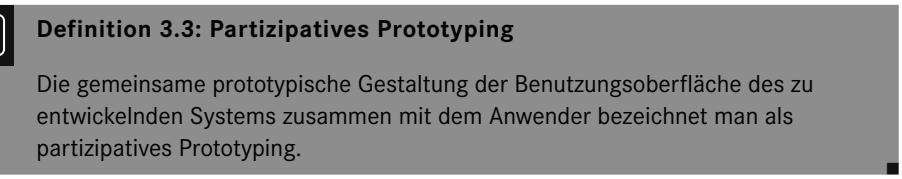
Aus diesem Grunde ist eine zyklische, evolutionäre Softwareentwicklung mit Präsentation von Prototypen notwendig ist. Diese Art der Vorgehensweise gestattet die schnelle Einbeziehung neuer Erkenntnisse und unterstützt ganz wesentlich die Kommunikation zwischen Auftraggebern, Anwendern und Entwicklern.

Die Zustimmung zu lauffähigen Prototypen sichert ein gemeinsames Verständnis des Anwendungsgebietes stärker als statische Spezifikationen in Form von Dokumenten. Das Rapid Prototyping hat sich daher als Erfolg versprechender Ansatz bei der Entwicklung von Software weitgehend durchgesetzt.

Definition 3.2: Rapid Prototyping

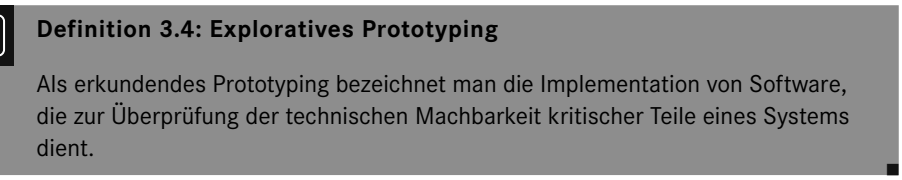
Unter dem Rapid Prototyping versteht man das schnelle (rapid) Erstellen eines lauffähigen Systems, das wesentliche Eigenschaften des endgültigen Softwaresystems besitzt.

Oft wird dieser Begriff mit dem partizipativen Prototyping gleichgesetzt, was nicht ganz korrekt ist. Unter partizipativem Prototyping versteht man die Einbeziehung des späteren Anwenders in die Systementwicklung, insbesondere bei der Gestaltung der Benutzungsschnittstelle. Daneben gibt es aber noch das explorative Prototyping, bei dem kritische Teilprobleme erkundet werden.



Definition 3.3: Partizipatives Prototyping

Die gemeinsame prototypische Gestaltung der Benutzungsoberfläche des zu entwickelnden Systems zusammen mit dem Anwender bezeichnet man als partizipatives Prototyping.



Definition 3.4: Exploratives Prototyping

Als erkundendes Prototyping bezeichnet man die Implementation von Software, die zur Überprüfung der technischen Machbarkeit kritischer Teile eines Systems dient.

Vor der Entwicklung eines Softwaresystems zum kooperativen Arbeiten wird man beispielsweise erst einmal einen Prototyp erstellen, der ein Byte zwischen zwei Orten überträgt. Von diesen Erfahrungen hängt dann die weitere Gestaltung des Softwaresystems ab. Wenn die Übertragung des einen Bytes bereits sehr zeitaufwendig ist, dann müssen prinzipiell neue Überlegungen angestellt werden.

Das Spiralmodell greift die Idee des Prototyping und des evolutionären Ansatzes auf. Es geht auf Barry W. Boehm [3.3] zurück und wiederholt zyklisch die Projektabschnitte Planung, Risikoanalyse, Realisierung, Bewertung. Dabei sind auch diese Abschnitte nicht streng getrennt, sondern überlappend.

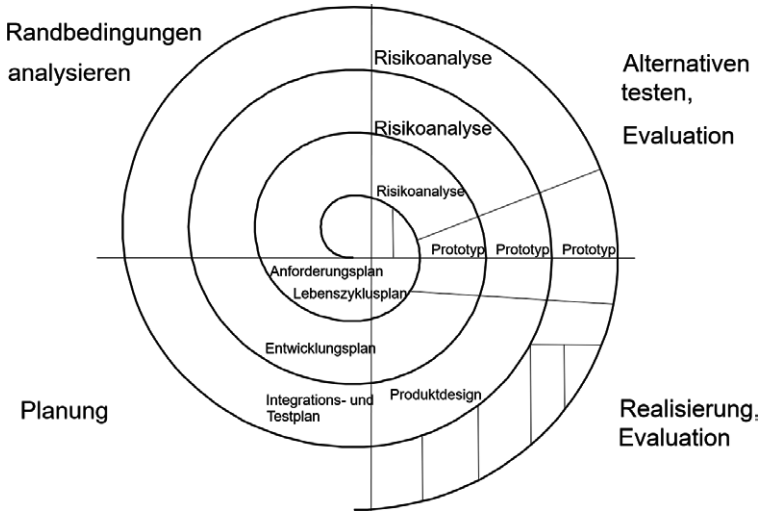


Bild 3.2 Spiralmodell nach Boehm

4 Kollaborative Analyse und Design

4.1 Einführung

Brooks [4.9] schreibt in seinem bekannten Essay zur Softwareentwicklung: „Software products are among the most complex of man-made systems, and software by its very nature has intrinsic, essential properties (e. g., complexity, invisibility, and changeability) that are not easily addressed." Die Gestaltung von interaktiven Softwaresystemen erfordert eine Betrachtung aus verschiedenen Perspektiven, damit eine hohe Produktqualität erreicht werden kann. Dazu sind in der Regel interdisziplinäre Zusammenarbeit und die aktive Einbeziehung von Benutzern und anderen Beteiligten in den Entwicklungsprozess notwendig, um ein gemeinsames Verständnis des Gestaltungsproblems und möglicher Lösungsansätze entwickeln zu können.

Dieses Kapitel macht Sie mit speziellen Methoden und Techniken vertraut, die kollaborative Analyse- und Designprozesse unterstützen. Es werden Beschreibungsmittel vorgestellt, die allgemein verständlich sind und die Darstellung und Diskussion des Warums, Was und Wie des zu entwickelnden Systems ermöglichen. In Kombination mit UML unterstützen sie eine qualitativ hochwertige Softwareentwicklung.

4.1.1 Besonderheiten und Qualität von Software

Software ist die Menge von Programmen, Verfahren, zugehörige Dokumentation und Daten, die mit dem Betrieb eines Computersystems zu tun haben [4.26]. Es wird dabei zwischen System- und Anwendungssoftware unterschieden. Unter Systemsoftware versteht man die Programme, die die Verbindung zu einer spezifischen Hardware herstellen. Dazu gehören etwa Betriebssysteme, Datenbankverwaltungssysteme oder Webserver. Anwendungssoftware ermöglicht die Abarbeitung von Anwendungsprogrammen, die die Erledigung von Aufgaben in bestimmten Anwendungsgebieten unterstützen sollen. In diesem Kapitel geht es um Anwendungssoftware, insbesondere um interaktive Systeme.

Software ist ein immaterielles Produkt, das nur in seinen Wirkungen beim Ablauf auf einem Computer bzw. indirekt über seine Dokumentation beobachtbar ist. Daher sind Fehler in der Softwarekonstruktion oft nicht leicht zu erkennen. Balzert [4.3] und Ludewig & Lichter [4.37] charakterisieren Software weiterhin wie folgt:

- Software wird nicht gefertigt, sondern nur entwickelt. Es gibt keine Fertigungskosten, wenn man als Fertigung die Reproduktion des ersten Musters versteht. Kopie und Original sind völlig gleich.
- Software unterliegt keinem Verschleiß, altert aber trotzdem.
- Softwarefehler entstehen nicht durch Abnutzung. Für Software gibt es keine Ersatzteile.
- Software ist im Allgemeinen leichter und schneller änderbar als ein anderes technisches Produkt. Dabei können kleinste Änderungen durchaus zu massiven Änderungen im Verhalten der Software führen.
- Software ist schwer zu „vermessen". Der Nachweis des wunschgemäßen Funktionierens ist schwieriger als bei anderen technischen Produkten.
- Software ist ein immaterielles Produkt

In [4.58] wird Software als eine Form der Explikation von Wissen mithilfe von Computersprachen verstanden. Es werden folgende Besonderheiten bezüglich ihrer Entwicklung, Distribution und Konsumption beschrieben.

- *Potenzielle Unabschließbarkeit der Entwicklung:* Die Entwicklung von Software kann immer weitergeführt werden. Software kann z. B. an neue Hardware angepasst oder ihre Funktionalität erweitert werden.
- *Potenzielle Unabschließbarkeit des Produktionsprozesses:* Die wachsende Komplexität von Software und sich ändernde Bedingungen (z. B. durch neue Produktanforderungen) tragen dazu bei, dass Software fehleranfällig ist. Es können immer wieder Fehler gefunden werden. Verbesserungen oder alternative Problemlösungen sind möglich bzw. sogar nötig.
- *Parallele Entwicklung:* Software ist ein symbolisches Produkt, das einfach versendet und verteilt werden kann. Weiterhin kann Software modularisiert werden. Dadurch wird eine parallele Entwicklung der Software durch Entwickler, die auch an verschiedenen Orten arbeiten können, möglich.
- *Verschränkung von Produktion und Anwendung:* Die Anforderungen an ein Produkt sind selten vollständig im Voraus bestimmbar. Sie können sich z. B. erst während der Nutzung eines Prototyps ergeben. Software muss in der Regel iterativ entwickelt werden, und es können immer wieder Anpassungen notwendig werden. Der Aufwand an Dienstleistungen, der mit dem Einsatz von Software verbunden ist, ist daher relativ hoch.

Wann hat Software eine hohe Qualität? Was unterscheidet gute von schlechter Software?

Qualität im alltäglichen Sprachgebrauch ist ein Synonym für die Güte eines Produktes oder eines Prozesses. Etwas hat beispielsweise eine „gute Qualität", wenn es langlebig ist. Der Begriff der Qualität lässt sich, wenn überhaupt, nur schwer definieren. Garvin [4.21] schlägt fünf Perspektiven vor, aus denen Qualität betrachtet werden kann.

- *Transzendente Perspektive:* Qualität ist ein anzustrebendes Ideal. Wir können Qualität erkennen, aber nicht analysieren und definieren.
- *Benutzerbezogene Perspektive:* Qualität ist Tauglichkeit für einen Zweck. Es wird angenommen, dass Benutzer individuelle Wünsche und Bedürfnisse besitzen und dass die Produkte, die diese am besten befriedigen, eine hohe Qualität besitzen.
- *Fertigungsbezogene Perspektive:* Qualität ist die Übereinstimmung des Produktes mit der Anforderungsspezifikation. Jede Abweichung von der Spezifikation bedeutet Einbußen in der Qualität.

- *Produktbezogene Perspektive*: Qualität ist an die inhärenten Eigenschaften (Attribute) eines Produktes gebunden.
- *Wertorientierte Perspektive:* Qualität ist definiert in Bezug auf Kosten und Preis. Mit anderen Worten, die Qualität hängt von der Anzahl der Kunden ab, die bereit sind, für das Produkt zu zahlen.

Die angegebenen Perspektiven auf die Qualität eines Produktes weisen unterschiedliche Annahmen und Schwerpunkte auf. Beispielsweise wird in der benutzer- und in der fertigungsbezogenen Perspektive ein Produkt von außen betrachtet, in der produktbezogenen Perspektive dagegen von innen. Im transzendenten Ansatz nimmt man an, dass Qualität keiner Seite (Produkt oder Benutzer) zugeschrieben und auch nicht gemessen werden kann. In der wertbezogenen Perspektive gibt es eine Vermischung des Qualitätsbegriffes mit dem Wertbegriff. Es geht um „Qualität, die man sich leisten kann“. Die Betrachtung der verschiedenen, sich teilweise widersprechenden Ansätze zur Definition von Qualität mag auf den ersten Blick akademisch erscheinen. Für eine erfolgreiche Softwareentwicklung ist es jedoch wichtig, dass im Verlaufe eines Projektes verschiedene Sichten auf die Produktqualität eingenommen werden und dass der Vorgang des Perspektivwechsels bewusst gestaltet wird. Im Folgenden gehen wir näher auf die produkt-, benutzer- und fertigungsbezogene Perspektive ein.

Die Qualität von Softwareprodukten wird, aus der **Produktperspektive** gesehen, an der Erfüllung bestimmter Produktmerkmale gemessen. Dabei gibt es verschiedene Vorschläge für Attribute, die einen wesentlichen Einfluss auf die Qualität von Softwareprodukten besitzen. Beispielsweise beschreibt das standardisierte, 1991 eingeführte Qualitätsmodell ISO 9126 (seit 2005: ISO/IEC 25000) relevante Qualitätsattribute in Form einer Hierarchie und empfiehlt, die Erfüllung der Attribute der letzten Hierarchieebene zu messen. Der Standard gibt dabei nicht im Detail vor, wie diese Messungen auszusehen haben. Zu den Hauptattributen dieses Modells gehören die folgenden:

- *Funktionalität:* Menge von Attributen, die das Vorhandensein der Funktionen und deren spezifizierten Eigenschaften betreffen, die die Anforderungen an die Software erfüllen sollen.
- *Zuverlässigkeit:* Menge von Attributen, die die Fähigkeit des Softwareproduktes betreffen, seine Leistung für den definierten Zeitraum und unter den festgelegten Bedingungen aufrechtzuerhalten.
- *Benutzbarkeit:* Menge von Attributen, die sich auf den Nutzungsaufwand einer Software für eine angegebene Benutzergruppe und der subjektiven Bewertung der Benutzung durch diese Benutzer beziehen.
- *Effizienz:* Menge von Attributen, die das Verhältnis zwischen Leistung und den unter festgelegten Bedingungen eingesetzten Ressourcen betreffen.
- *Wartbarkeit:* Menge von Attributen, die den Aufwand für festgelegte Änderungen betreffen (z. B. Korrekturen, Verbesserungen, Anpassungen der Software an veränderte Umgebungsbedingungen bzw. an Änderungen in den Anforderungen)
- *Übertragbarkeit:* Menge von Attributen, die die Fähigkeit der Software betreffen, in eine andere Umgebung übertragen zu werden (z. B. andere Hardware- oder Softwareumgebung, anderes organisatorisches Umfeld).

Index