12 Software-Entwicklung

Die Entwicklung von Software ist ein außerordentlich komplexer Prozess, der umso problematischer wird, je umfangreicher die zu entwickelnde Software ist. Dabei kann man den Umfang von Software auf verschiedene Arten messen. Beispiele für solche Maße sind:

- die Zahl der Quelltextzeilen der Programme, aus denen das Software-Produkt besteht;
- die Zeit, die benötigt wird, um ein Programm zu erstellen. Diese kann z.B. in Bearbeiter-Jahren (*BJ*) gemessen werden.

Beide Maße sind offensichtlich nicht sehr genau. Es gibt mehr oder weniger kompakten Quelltext und mehr oder weniger produktive Bearbeiter. Außerdem kann ein schnell gelieferter Quelltext unbrauchbar oder fehlerhaft sein – daher ist die Erstellungsgeschwindigkeit oft ein trügerisches Maß. In grober Annäherung kann man davon ausgehen, dass ein Bearbeiter an einem Arbeitstag ca. 10 bis 100 Zeilen (*Lines of Code = LOC*) produziert. Diese Schätzung ist von vielen Einflussgrößen abhängig, insbesondere auch von der Komplexität und dem Umfang des Projektes. Als grobe Näherung wollen wir im Folgenden ca. 5000 Programmzeilen pro Jahr annehmen.

Die folgende Tabelle enthält eine grobe Klassifikation von Software-Projekten hinsichtlich ihres Umfangs:

Projektklasse	Quelltext-Zeilen (LOC)	Bearbeitungsaufwand (BJ)
sehr klein	0 – 1.000	0 - 0,2
klein	1.000 – 10.000	0,2 - 2
mittel	10.000 – 100.000	2 - 20
groß	100.000 – 1 Mio.	20 - 200
sehr groß	1 Mio. – ...	200 - ...

Abb. 12.1: *Klassifikation von Software-Projekten*

Eines der Hauptprobleme der Software-Entwicklung ist die Entwicklung *zuverlässiger Software*. Man bezeichnet ein Programm als *zuverlässig*, wenn es sich – relativ zu vorgegebenen Toleranzwerten für Abweichungen – im Betrieb so verhält, wie man es aufgrund der *Anforderungen* an das Programm erwartet. Je umfangreicher ein Software-Projekt ist, desto unwahrscheinlicher ist es, dass sein Ergebnis jemals *fehlerfrei* wird. Man muss sogar davon ausgehen, dass es unmöglich ist, umfangreiche Software-Produkte vollständig fehlerfrei zu entwickeln.

Ein weiteres Hauptproblem der Software-Entwicklung besteht darin, die Software so zu entwickeln, dass sie später problemlos *geändert* werden kann. Für den Entwickler ist es vorteilhaft, wenn die Entwicklung eines Software-Systems nach seiner Fertigstellung beendet ist. Eine weitergehende Betrachtung zeigt jedoch, dass die Entwicklung eines Software-Systems ein *evolutionärer Prozess* ist, der oft sehr lange währt, und dessen Ende womöglich nicht abzusehen ist. So wurden in den Jahren 1960 bis 1980 viele Software-Systeme entwickelt, von denen man annahm, dass sie das Jahr 2000 nicht überdauern würden. Dies hat dann zu dem bekannten „Jahr 2000 Problem“ geführt.

Änderungen eines „fertig“ entwickelten Softwareprodukts werden nötig, wenn z.B. einer der folgenden Fälle eintritt:

- es werden Fehler entdeckt;
- es werden neue Anforderungen an die Software gestellt, weil sich z.B. gesetzliche, betriebliche oder organisatorische Rahmenbedingungen geändert haben;
- die Software soll in einer anderen Hardware-Umgebung eingesetzt werden;
- die Software muss zusätzliche Hardware-Komponenten nutzen.

12.1 Methoden und Werkzeuge für Projekte

Seit 1968 beschäftigt man sich in der Informatik unter dem Schlagwort *Software Engineering* mit der Frage, wie die Entwicklung von qualitativ hochwertiger Software ablaufen sollte. Einige Kriterien für die Qualität von hochwertiger Software sind:

- Zuverlässigkeit und Korrektheit – d.h. relative Fehlerfreiheit,
- Modifizierbarkeit, Wartbarkeit, Testbarkeit und Wiederverwendbarkeit,
- Effizienz,
- Kosten.

Das Ergebnis einer *idealen* Software-Entwicklung wären völlig fehlerfreie, kostengünstige Software-Systeme, die man später leicht ändern kann. Eine Patentlösung zur Erfüllung dieses Ideals ist bisher allerdings nicht gefunden worden. Jedoch sind u.a. folgende Methoden mit unbestrittenem Erfolg eingesetzt worden:

- *Strukturierte Programmierung*,
- *Top-down-Entwurf* von Software-Systemen und *schrittweise Verfeinerung* von Programmen,
- *Modularisierung* von Software nach dem *Daten-Abstraktionsprinzip*,
- *Objektorientierte Analyse, Modellierung und Software-Konstruktion.*

Während einer Informatik-Ausbildung erwirbt der Lernende in der Regel die Fähigkeit zur Entwicklung von *sehr kleinen* Software-Produkten. In einem Praktikum stellt sich ihm die Aufgabe, eventuell ein etwas größeres Programm zu entwickeln, das aber immer noch *klein* im Sinne der obigen Klassifikation ist.

Die Hauptprobleme der Software-Entwicklung tauchen erst bei mittleren und großen Projekten auf, die in einem Team von mehreren Entwicklern bearbeitet werden. Die Entwicklung großer Software-Systeme kann allein schon aus Zeitgründen in keiner Informatik-Ausbildung geübt werden. Die genannten Probleme und Techniken werden den Lernenden zwar theoretisch erläutert und an überschaubaren Beispielen demonstriert, wirklich begreifen werden sie diese aber vermutlich erst in der Praxis, wenn sie zum ersten Mal selbst in einem umfangreichen Entwicklungsprojekt tätig werden.

Wenn die Programme einen Umfang von 1000 oder 2000 Zeilen übersteigen, versagt in der Regel eine völlig unsystematische Programmierung. Das Problem muss dann in mehrere Teilprobleme zerlegt werden. Die Teile müssen mehrfach *wiederverstanden* werden, wenn sie z.B. an anderer Stelle oder von anderen Programmierern *benutzt* werden, was eine methodische Vorgehensweise voraussetzt. Darum teilt man *mittlere* und *große* Programme in mehrere *Module* auf, die möglicherweise von mehreren Entwicklern unabhängig voneinander erstellt und separat übersetzt werden. Dabei muss der Compiler die gemeinsam verwendeten Daten- und Kontrollstrukturen auf konsistente Verwendung überprüfen.

Für kleine Software-Projekte ist eine formale *Projektorganisation* nicht unbedingt erforderlich. Sind an einem Software-Projekt jedoch viele, zum Teil wechselnde Entwickler längere Zeit tätig, kann man ohne die methodische Organisation des Projektes nicht mehr auskommen. Je größer ein Software-System wird, desto mehr erhöht sich die Wahrscheinlichkeit, dass gravierende Fehler unentdeckt bleiben oder – noch schlimmer – dass die gesamte Funktionalität des Systems nicht mehr überschaubar ist und damit keine zuverlässigen Aussagen über sein Verhalten gemacht werden können. Da aber die umfangreichsten Software-Systeme ausgerechnet in Bereichen benutzt werden, die unter Sicherheitsaspekten kritisch sind, ist die Frage zu stellen, ob *zu große* Software-Systeme überhaupt geplant und realisiert werden sollten. Immer wieder hört man von Fehlfunktionen, wie z.B. bei Weltraumflügen oder Satellitensystemen, die letzten Endes von Software-Fehlern verursacht werden und die zu hohen Risiken und Verlusten führen.

Für die *Produktivität* von Software-Entwicklern sind viele Faktoren maßgeblich. Dazu gehören z.B. die Ausstattung mit leistungsfähigen Rechnern, Netzverbindungen und geeigneten Entwicklungs-Werkzeugen. Diese werden unter der Bezeichnung *Programmierumgebung* oder *Software-Entwicklungsumgebung* zusammengefasst. (Siehe dazu auch S. 864.)

Neben den technischen Voraussetzungen sind die organisatorischen und sozialen Rahmenbedingungen für die Arbeitsproduktivität ausschlaggebend: Wer sich in seiner Arbeitsumgebung wohl fühlt, leistet mehr. Diese oft unterschätzte Erkenntnis wird u.a. in dem Buch *Peopleware* von Tom de Marco in eindrucksvoller Weise demonstriert. Wer z.B. seine Programmierer wie Sklaven in fensterlose Kabäuschen („cubicles“) sperrt und schamlos für unbezahlte Überstunden ausnutzt, kann nicht damit rechnen, dass diese lange in seiner Firma bleiben und damit wirklich produktiv für diese werden. Spitzenleistungen lassen sich in der Software-Entwicklung (wie in anderen Berufszweigen) nur von hoch motivierten, aufeinander eingespielten, „verschworenen“ Teams erreichen.

Etwa gleichzeitig mit dem Aufkommen des Begriffs Software Engineering wurde das Wort von der *Software-Krise* geprägt. Damit war im weitesten Sinne die Summe aller Probleme angesprochen, die sich einerseits aus schnell wachsenden Anforderungen und immer komplexeren Aufgabenstellungen und andererseits aus dem schnellen Wandel der Hardware- und Kommunikationstechnik ergaben. D.h. in kürzester Zeit sollten immer größere Programmsysteme für neue, hochkomplexe Aufgabenstellungen auf immer neuen, einem stetigen Wandel unterliegenden technischen Plattformen entwickelt werden.

Diese Herausforderungen an die Software-Entwickler und an die dafür verantwortlichen Manager und Projektleiter sind bis heute kaum geringer geworden. Zwar hat sich die *Softwaretechnik* als Informatik-Fachgebiet etabliert und eine Vielzahl von nützlichen, heute unentbehrlichen Methoden, Techniken und Werkzeugen hervorgebracht. Auch sorgt heute eine solide Ausbildung in Informatik und speziell Softwaretechnik für einen Stamm von gut qualifizierten Software-Entwicklern. Aber gleichzeitig schaffen immer kürzere Innovationszyklen bei Rechnerausstattung und Vernetzung sowie ein ungebremster Drang zur Automatisierung mit immer größeren und komplexeren Anwendungen ständig neue Herausforderungen an die Softwaretechnik. Ein Ende dieses Wettlaufs ist zurzeit noch nicht abzusehen.

12.2 Vorgehensmodelle

Eine zentrale Frage der Softwaretechnik betrifft das Vorgehen im Projekt. B. Boehm hat 1988 vier Klassen von *Vorgehensmodellen* unterschieden:

- *Code and fix*-Modelle,
- *Phasen*- und *Wasserfall*-Modelle,
- *Transformations*-Modelle,
- *Evolutionäre Entwicklungsmodelle.*

Wir erweitern diese Klassifizierung um

- Prototyping- und Spiralmodelle,
- Modelle zur *inkrementellen* Systementwicklung,
- Modelle zur *objektorientierten* Systementwicklung.

Im Folgenden wollen wir diese sieben Modellklassen (in einer leicht veränderten Reihenfolge) kurz durch ihre wesentlichen Eigenschaften charakterisieren. Dabei unterscheiden wir *sequentielle* Modelle, d.h. solche, die den Software-Entwicklungsprozess als eine Folge sequentiell ablaufender Phasen betrachten, von *nicht-sequentiellen* Modellen.

12.2.1 Code and fix-Verfahren

Unter *code and fix*-Verfahren versteht B. Boehm die unsystematische Vorgehensweise in der Frühzeit des Programmierens: Man beginnt ohne weitere Planungen und Vorüberlegungen mit dem Schreiben von *Code* (oft in einer niederen, wenig Übersicht bietenden Programmiersprache) und endet mit dem langwierigen und mühseligen Austesten und Zusammenfügen der Programmbausteine (*fix*).

12.2.2 Wasserfall-Modelle

Der Begriff *Wasserfall* steht für ein phasenweises Vorgehen, das auch heute noch, in verschiedenen Variationen, in der industriellen Praxis weit verbreitet ist. Kennzeichnend für diese Vorgehensweise ist die Einteilung des Entwicklungsprozesses in sequentiell aufeinander folgende Phasen. Für jede Phase sind Ausgangspunkt und Vorgaben, durchzuführende Tätigkeiten und Ergebnisse genau festgelegt. Jedes Phasenergebnis bildet gleichzeitig die Vorgabe für die weitere Entwicklung in der Folgephase. Deren Ergebnis ist dann am Ende des Phasendurchlaufs gegenüber der Vorgabe zu überprüfen.

Das bekannteste Beispiel für ein Wasserfall-Modell zeigt die folgende Abbildung:

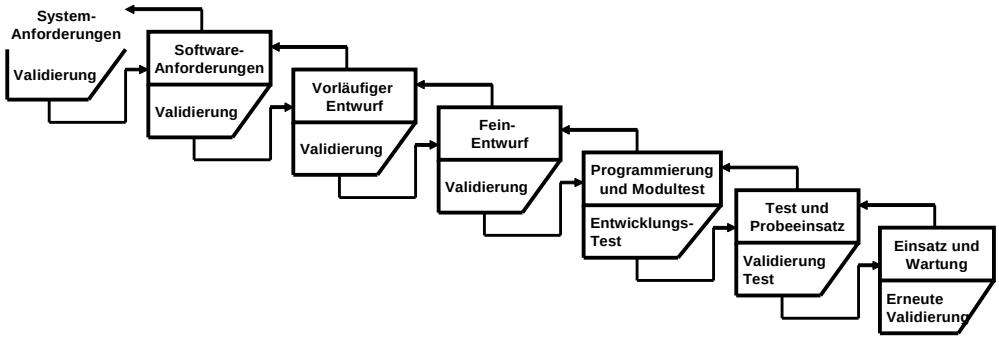

Abb. 12.2: *Wasserfall-Modell von Royce/Boehm*

Die wasserfallartigen Vorgehensmodelle haben vor allem zusammen mit der in den 1970er Jahren entwickelten und danach zunehmend auch industriell eingesetzten Daten-Abstraktionstechnik (vgl. S. 840) eine große Verbreitung gefunden. Diese Technik ermöglicht die systematische Dekomposition von großen Software-Systemen in klar definierte Module mit sauber spezifizierten und überprüfbaren Schnittstellen. Damit war eine notwendige Voraussetzung für eine weitgehende Arbeitsteilung und die Synchronisation parallel verlaufender Entwicklungstätigkeiten gegeben.

Für das *Projekt-Management* bedeutet dies die Betonung einer *ergebnisorientierten* Arbeitsweise: Entwickler können aufgrund vorgegebener oder gemeinsam abgestimmter und dann verbindlicher Spezifikationen in Parallelarbeit ihre Module entwickeln. Die Ergebnisse sind in Form und Umfang festgelegt und müssen oft in einem engen Zeitrahmen erbracht werden, um ihre Integration vorbereiten und termingerecht durchführen zu können.

Für diese „rigorose“ Vorgehensweise spricht eine ganze Reihe von Argumenten, und sie hat sich in der Praxis bei zahlreichen, erfolgreich abgewickelten Software-Projekten bewährt. Verfahren dieser Art sind nicht zuletzt deshalb so erfolgreich, weil sie sowohl ein methodisch fundiertes und technisch ausgereiftes Konzept zur Systemstrukturierung als auch wirkungsvolle Mittel für die Projektführung anbieten, um mit dem Problem der Arbeitsteilung und der Zusammenführung parallel entwickelter Bausteine in großen Projekten fertig zu werden.

Heute finden wir in den meisten bekannten Phasenmodellen (in mehr oder weniger modifizierter Form) die folgenden Projektabschnitte:

1. Problemanalyse und Anforderungsdefinition
2. Modellierung und fachlicher Entwurf
3. Software-technischer Entwurf
4. Programmierung und Modultest
5. System-Integration und Systemtest
6. Installation, Betrieb und Weiterentwicklung

Problemanalyse und Anforderungsdefinition

Zu Projektbeginn wird der *Gegenstandsbereich* des Projekts (z.B. die Kontenführung bei einer Bank oder die Einführung von Bildschirm-Terminals in den Schaltern der Postämter) analysiert und abgegrenzt gegenüber Leistungen und Vorgängen, die nicht Gegenstand des Projekts sein sollen. Sodann werden die *Anforderungen* an das zukünftige Software-System (die sich in der Regel auch auf die davon betroffene organisatorische Umgebung beziehen) in überprüfbarer (d.h. schriftlicher) Form niedergelegt.

Früher wurde diese Projekt-Startphase oft unterschätzt und vernachlässigt, Anforderungen wurden häufig gar nicht oder nur unzureichend erarbeitet und dokumentiert. Heute weiß man aus Erfahrung, wie sehr der Projekterfolg von sauber spezifizierten Anforderungen abhängt und misst dieser Phase eine so große Bedeutung zu, dass sich daraus ein eigener Zweig der Softwaretechnik, genannt *Requirements Engineering*, herausgebildet hat.

Modellierung und fachlicher Entwurf

Nachdem der Gegenstandsbereich des Projekts abgegrenzt und die Funktionalität des künftigen Systems grob festgelegt ist, werden dessen Funktionen aus fachlicher Sicht vollständig spezifiziert. Grundlage dafür ist in der Regel ein Modell, das die Objekttypen des Gegenstandsbereichs (z.B. Kunden, Banken, Konten, Überweisungen, Einlagen, Vergütungen etc.) benennt, strukturiert und miteinander in Beziehung setzt. Ein solches Modell wird *Datenmodell*, *Informationsmodell*, *Klassenmodell* oder *Objektmodell* genannt. Sind dort einmal die Bezeichnungen und Strukturen für alle relevanten Objekte festgelegt, so lassen sich die Systemfunktionen wesentlich leichter beschreiben und untereinander konsistent halten.

Das Datenmodell und die Beschreibungen der Funktionen und Abläufe werden zusammen auch als *Anwendungsmodell* bezeichnet. Dieses bildet den Kern des *fachlichen Entwurfs.*

Software-technischer Entwurf

Im Gegensatz zum fachlichen bezieht sich der (Software-)technische Entwurf ganz auf die Struktur der zu entwickelnden Software. Im Zuge der objektorientierten Methoden wurden beide Entwürfe in ihrer Struktur vereinheitlicht – man spricht hier auch von *model-driven architecture* (vgl. unten) – und die Grenze zwischen ihnen verschwimmt zunehmend. Ande-