

12 Softwareentwicklung

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 Softwareprodukt 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 Softwareprojekten 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	über 1 Mio.	über 200

Abb. 12.1: Klassifikation von Softwareprojekten

Eines der Hauptprobleme der Softwareentwicklung 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 Softwareprojekt ist, desto unwahrscheinlicher ist es, dass sein Ergebnis jemals *fehlerfrei* wird. Man muss sogar davon ausgehen, dass es unmöglich ist, umfangreiche Softwareprodukte vollständig fehlerfrei zu entwickeln.

Ein weiteres Hauptproblem der Softwareentwicklung besteht darin, die Software so zu entwickeln, dass sie später problemlos *geändert* werden kann. Für den Entwickler ist es vorteilhaft, wenn die Entwicklung eines Softwaresystems nach seiner Fertigstellung beendet ist. Eine weitergehende Betrachtung zeigt jedoch, dass die Entwicklung eines Softwaresystems 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 Softwaresysteme 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 geänderten Hardware- oder Softwareumgebung eingesetzt werden
- die Software muss zusätzliche Hardware- oder Softwarekomponenten nutzen.

12.1 Herausforderungen an die Softwareentwicklung

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,
- leichte Erlernbarkeit und Benutzerfreundlichkeit
- Effizienz,
- Kosten.

Die Ergebnisse einer *idealen* Softwareentwicklung wären völlig fehlerfreie, kostengünstige Softwaresysteme, 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:

- *Top-Down-Entwurf* von Softwaresystemen,
- *Modularisierung* von Software nach dem *Datenabstraktionsprinzip*,
- *Objektorientierte Analyse, Entwurf und Implementierung von Softwaresystemen*

Während einer Informatikausbildung erwirbt der Lernende in der Regel die Fähigkeit zur Entwicklung von *sehr kleinen* Softwareprodukten. In einem Softwarepraktikum soll eventuell ein etwas größeres Softwaresystem entwickelt werden, das aber immer noch *klein* im Sinne der obigen Klassifikation ist.

Die Hauptprobleme der Softwareentwicklung tauchen erst bei mittleren und großen Projekten auf, die in einem größeren Entwicklerteam bearbeitet werden. Die Entwicklung großer Systeme kann allein schon aus Zeitgründen in keiner Informatikausbildung geübt werden. Die genannten Probleme und Techniken werden den Lernenden zwar theoretisch erläutert und an überschaubaren Beispielen demonstriert, wirklich begreifen werden sie diese aber erst in der Praxis, wenn sie zum ersten Mal 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 Softwareprojekte ist eine formale *Projektorganisation* nicht unbedingt erforderlich, aber häufig schon nützlich. Sind an einem Softwareprojekt jedoch viele, zum Teil wechselnde Entwickler längere Zeit tätig, kann man ohne die methodische Organisation des Projektes nicht mehr auskommen. Je größer ein Softwaresystem 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. Immer wieder hört man von Fehlfunktionen, wie z.B. bei Weltraumflügen oder Satellitensystemen, die letzten Endes von Softwarefehlern verursacht werden und die zu hohen Risiken und Verlusten führen.

Für die *Produktivität* von Softwareentwicklern sind viele Faktoren maßgeblich. Dazu gehören z.B. die Ausstattung mit leistungsfähigen Rechnern, Netzverbindungen sowie geeignete Sprachen, Bibliotheken und Werkzeuge. Diese werden unter der Bezeichnung *Programmierungsumgebung* oder *Softwareentwicklungsumgebung* zusammengefasst.

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 *Peopeware* (dt. Titel: *Wien wartet auf Dich*) von Tom de Marco und Timothy R. Lister 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 Softwareentwicklung (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 *Softwarekrise* 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 Programmsys-

teme für neue, hochkomplexe Aufgabenstellungen auf immer neuen, einem stetigen Wandel unterliegenden technischen Plattformen entwickelt werden.

Diese Herausforderungen an die Softwareentwickler und an die verantwortlichen Manager und Projektleiter sind bis heute kaum geringer geworden. Zwar hat sich die *Softwaretechnik* als ein Fachgebiet der Informatik etabliert und eine Vielzahl nützlicher, heute unentbehrlicher Methoden, Techniken und Werkzeugen hervorgebracht. Auch sorgt eine solide Ausbildung in Informatik und speziell in der Softwaretechnik für einen Stamm von gut qualifizierten Softwareentwicklern. 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. Die Beherrschung der Komplexität bleibt die zentrale Herausforderung in der Softwareentwicklung.

12.2 Softwareentwicklungsprozesse

Die Entwicklung eines großen Softwaresystems sollte systematisch durchgeführt werden. Grundlegende Arbeitsschritte wie Analyse und Entwurf eines Systems sowie Implementierung und Testen werden in einem Vorgehensmodell für Softwareentwicklung sinnvoll strukturiert. Während man kleine Softwaresysteme ohne weitere Planung und Vorüberlegungen direkt implementieren kann, führt diese Vorgehensweise meist zu code von fragwürdiger Qualität und zu einem langwierigen Austesten der Software. Dies hat man schon frühzeitig erkannt und mit der Erprobung von Vorgehensmodellen für eine systematische Softwareentwicklung begonnen. Im Folgenden geben wir einen knappen Überblick über die wichtigsten Vorgehensmodelle.

12.2.1 Wasserfallmodelle

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.

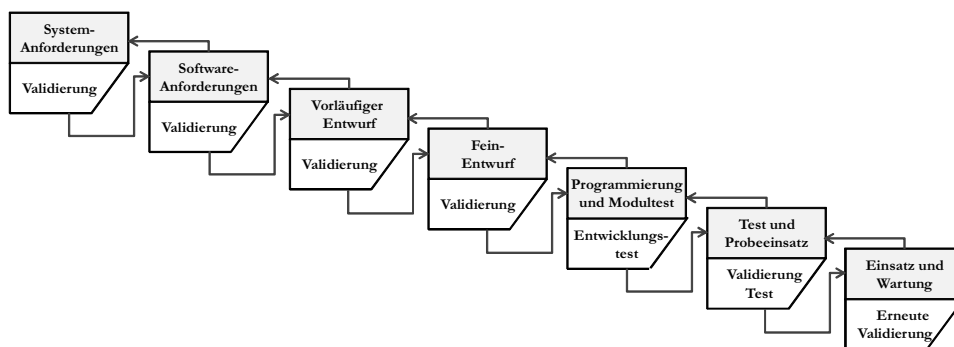


Abb. 12.2: Wasserfallmodell von Royce/Boehm

Für das *Projektmanagement* 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 Softwareprojekten 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 Entwicklungsmodellen (in mehr oder weniger modifizierter Form) die folgenden Projektarbeiten:

1. Problemanalyse und Anforderungsdefinition
2. Modellierung und fachlicher Entwurf
3. Software-technischer Entwurf
4. Programmierung und Modultest
5. Systemintegration 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 Softwaresystem (die sich in der Regel auch auf die davon betroffene organisatorische Umgebung beziehen) in überprüfbarer (d.h. schriftlicher) Form niedergelegt.

Früher wurde diese Startphase des Projekts 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 *Systemanalyse* oder *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*, *Infor-*

mationsmodell, *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*.

Softwaretechnischer Entwurf

Im Gegensatz zum fachlichen bezieht sich der softwaretechnische Entwurf ganz auf die Struktur oder auch Architektur der zu entwickelnden Software. Im technischen Entwurf wird das System in Teilsysteme (oft *Komponenten* oder *Pakete* – engl. *packages* – genannt) gegliedert, die wiederum aus selbständigen, meist getrennt übersetzbaren Bausteinen, den so genannten *Modulen*, bestehen. Diese haben miteinander *Schnittstellen*, über die sie z.B. untereinander Operationen aufrufen oder auf gemeinsame Daten zugreifen. Zum Entwurf gehört eine saubere *Spezifikation* aller Schnittstellen.

Programmierung und Modultest

Die eigentliche *Implementierung* nimmt innerhalb des gesamten Entwicklungsprozesses heute oft nur noch ca. 20 bis 30 % ein. Ein wichtiger Bestandteil dieser Phase ist der *systematische Test* der Module. Dazu werden Testfälle definiert und mit ausgesuchten Testdaten ausgeführt. Solche geplanten und jederzeit (z.B. nach Programmänderungen) wiederholbaren Tests sind nicht zu verwechseln mit der Fehlerbehebung (*debugging*), d.h. mit Programmänderungen aufgrund erkannter Fehler.

Systemintegration und Systemtest

Nach der Codierung und den erfolgreich abgeschlossenen Modultests werden einzelne Module nach einem vorher festgelegten Integrationsplan zu so genannten *Subsystemen* zusammengefügt und diese werden in ähnlicher Weise wie vorher die einzelnen Module getestet. Die Integration aller Subsysteme zum Gesamtsystem und der *Systemtest* schließen den Entwicklungsprozess (vorläufig) ab.

Installation, Betrieb und Weiterentwicklung

Ist der Systemtest in der Entwicklungsumgebung erfolgreich bestanden, wird das System in seiner Zielumgebung installiert, und unter der Aufsicht des Auftraggebers findet der *Abnahmetest* statt. Danach kann das System in den laufenden Betrieb gehen. Oft findet allerdings auch ein *schrittweiser Übergang* statt, z.B. bei der Umstellung von einem Altsystem auf ein Neusystem, bei der gemäß einer vorher festgelegten Übergangsstrategie Komponenten des alten Systems Schritt für Schritt durch neue ersetzt werden.

Ein Softwaresystem ist niemals „fertig“, d.h. es gibt immer Änderungs-, Verbesserungs- und Erweiterungswünsche. Begrenzte Modifikationen, die sich nur geringfügig auf das Gesamtverhalten des Systems auswirken, lassen sich oft im laufenden Betrieb, z.B. durch den Austausch einzelner Module, bewerkstelligen. Nehmen die Änderungswünsche dagegen einen

größeren Umfang an und sind Auswirkungen auf die Gesamtstruktur des Systems zu erwarten, so ist oft eine Weiter- bzw. Neuentwicklung des Systems in einem eigenen Projekt erforderlich. Im Zusammenhang mit der *evolutionären Systementwicklung* werden wir auf diesen Punkt zurückkommen.

12.2.2 Nichtsequentielle Vorgehensmodelle

Neben ihren unbestreitbaren Vorteilen weisen die sequentiellen Modelle auch eine Reihe von Mängeln auf, die dazu geführt haben, dass man in der Praxis häufig vom vorgegebenen Vorgehen abgewichen ist und schließlich nach anderen, nichtsequentiellen Vorgehensweisen Ausschau gehalten hat. Zu den häufig genannten Mängeln gehören:

- *Realitätsferne und mangelnde Flexibilität des Vorgehens*
Die Idealvorstellung eines sequentiellen Projektfortschritts von Phase zu Phase erweist sich in der Praxis oft als unrealistisch. Für Anforderungen, die sich noch während der Entwicklung ändern, für Tätigkeiten, die parallel über Phasengrenzen hinweg ablaufen, und für Rückkehrschleifen zu früher durchlaufenen Phasen ist in der Modellvorstellung kein Platz, in der Praxis sind sie jedoch an der Tagesordnung. Die vom Phasenmodell geforderte Festschreibung der Anforderungen und Spezifikationen geht oft an den Projektbedürfnissen vorbei. Dies gilt besonders für große, komplexe und innovative Projekte in sich schnell verändernden Anwendungsbereichen.
- *So genannte „Softwarebürokratie“*
Der Zwang, zu festgelegten Zeitpunkten ständig neue Dokumente zu erzeugen, führt zu Redundanzen und Ineffizienzen. Im Extremfall fühlen sich die Entwickler durch übertriebene Vorschriften, Richtlinien, starre Phasen- und Dokumentstrukturen gegängelt und zweifeln am Sinn ihrer Tätigkeit. Die Starrheit des Vorgehens überträgt sich häufig auch auf die Produkte und beeinträchtigt deren Anpassungsfähigkeit an geänderte Anforderungen und neue Bedürfnisse.
- *Keine langfristige Weiterentwicklungsstrategie*
Softwareprojekte sind früher häufig als isolierte, inselhafte Entwicklungsvorhaben „auf der grünen Wiese“ angesehen worden. Mittlerweile bezieht sich nur noch ein kleiner Teil der Softwareprojekte auf totale Neuentwicklungen. In der Mehrzahl der Fälle knüpft man an bereits existierende Systeme an. Dafür bieten die sequentiellen Vorgehensmodelle allerdings kaum Rezepte an.
- *Trennung von Anwender- und Entwicklerwelt*
Das Phasenschema verlangt, dass zuerst auf der Anwenderseite die Anforderungen festgelegt werden, ehe man auf der Entwicklerseite mit deren Umsetzung beginnt und dann wieder auf der Anwenderseite das entwickelte System einsetzt. In der Praxis ist man aber oft auf Zusammenarbeit und Rückkopplungen angewiesen, besonders wenn die Anforderungen noch instabil sind und nicht klar ist, ob, in welchem Umfang und mit welchem Aufwand sie sich realisieren lassen.

Um sich in dieser Situation besser der Realität anzupassen, wurden *nichtsequentielle Vorgehensmodelle* entworfen und praktiziert. Vorrangiges Ziel der meisten nichtsequentiellen Modelle ist es, die evolutionäre Softwareentwicklung systematisch zu unterstützen, indem

Entwicklungsphasen mehrfach durchlaufen werden. Zu den nichtsequentiellen Vorgehensmodellen gehören das Spiralmodell, in dem ein Durchlauf auf direkt auf dem vorhergehenden aufsetzt, sowie inkrementelle Modelle wie der Unified Process und die agile Softwareentwicklung. Im inkrementellen Modell erfolgt die Entwicklung eines Softwaresystems stufenweise. Im Folgenden gehen wir auf die inkrementellen Modelle genauer ein.

12.2.3 Modelle zur inkrementellen Softwareentwicklung

Ausgangspunkt der inkrementellen Softwareentwicklung ist ein relativ kleiner, überschaubarer Kern. Das kann z.B. ein Steuermodul zusammen mit einer ersten Systemfunktion sein. Dieser Kern wird jedoch beibehalten und weiter ausgebaut. Weitere Funktionen werden schrittweise entwickelt und als *Inkremete* eingehängt. Das kann sequentiell nacheinander, aber auch zeitlich verschränkt und teilweise parallel zueinander erfolgen. Voraussetzung dafür ist allerdings, dass die Systemfunktionen voneinander (relativ) unabhängig sind bzw. in systematischer Weise aufeinander aufbauen.

Der Unified Process

Bei der Firma *Rational*, die heute zu IBM und zu den Marktführern für OO-Entwicklungsmethoden und -werkzeuge gehört, wurde in den 1990er Jahren die Modellierungssprache UML (vgl. S. 840) und dazu passend ein Vorgehensmodell unter dem Namen *Rational Unified Process (RUP)* entwickelt. Dieses geht auf die Ideen von G. Booch, J. Rumbaugh und vor allem auf den *Objectory Process* von I. Jacobson zurück. Methodische Aspekte wie z.B. eine anwendungsfallgetriebene (*use case driven*) Analyse oder eine inkrementelle Systementwicklung gehen ausdrücklich in die Prozessbeschreibung ein. Die Aufgaben der Entwickler sind allerdings in einer recht komplexen Projektstruktur verankert, was eine systematische Projektplanung und -verfolgung durch das Management nicht leichter macht.

Die wichtige Forderung nach Wiederverwendbarkeit der Bausteine zieht zwangsläufig deren größere Unabhängigkeit voneinander nach sich. Damit löst sich die strenge Phasenstruktur der Wasserfallmodelle vollends auf: Entwicklungszyklen einzelner Bausteine laufen „evolutionär“, d.h. zeitlich verschränkt, – und bis zu einem gewissen Grade unabhängig voneinander, aber rückgekoppelt ab – bis hin zur Verlagerung von Analysetätigkeiten in die spezifischen Bausteinzyklen hinein.

Der Unified Process unterscheidet die folgenden Hauptprozessphasen:

1. Anfang (engl. *inception*): Der grundlegende Umfang des Projekts ist bekannt. Diese Phase endet mit der Zusage des Auftraggebenden.
2. Vertiefung (engl. *elaboration*): Diese Phase endet mit einer Beschreibung der grundlegenden Systemarchitektur sowie der wesentlichen Datenstrukturen und Abläufe. Darüber hinaus steht der Konstruktionsplan fest und die Hauptrisiken sind klar identifiziert.
3. Konstruktion (engl. *construction*): Hier geht es ganz wesentlich um die Realisierung des Systems. Diese Phase ist sicher iterativ und endet mit einem Beta-Release.
4. Inbetriebnahme (engl. *transition*): In dieser Phase wird das Softwaresystem beim Anwender eingeführt

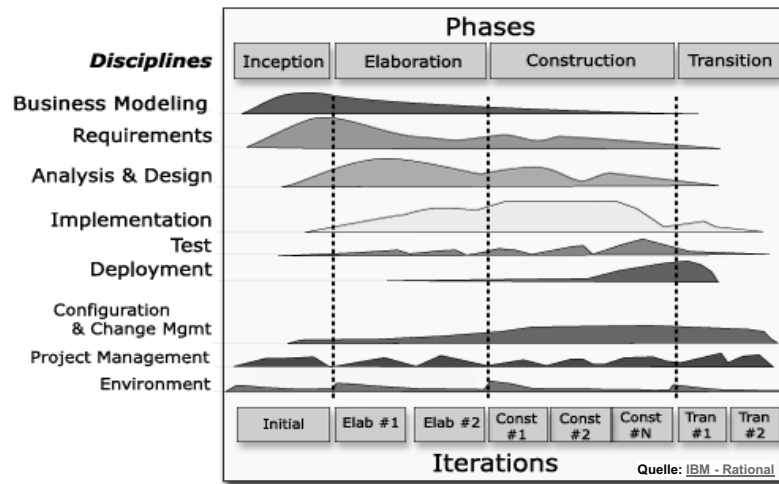


Abb. 12.3: Unified Process (Quelle: IBM-Rational)

Wie in der obigen Abbildung gezeigt, sind die Phasen zeitlich geordnet. Die Iterationen beschreiben verschiedene Abschnitte innerhalb einer Phase. Jede Iteration in der Konstruktionsphase führt zu einem ausführbaren Softwaresystem. Prinzipiell soll es möglich sein, auch neue Anforderungen nach einer Iterationsstufe zu berücksichtigen. Natürlich ist klar, dass dies meist umso schwieriger wird, je weiter das Softwaresystem entwickelt ist. Der Unified Process eignet sich insbesondere für die Entwicklung von sehr großen Softwaresystemen in großen, womöglich auch verteilten Teams.

Agile Softwareentwicklung

Als Gegenentwicklung zu einem sehr bürokratischen Vorgehen in der Softwareentwicklung, wo jeder kleine Schritt vorgeschrieben ist und dokumentiert werden muss, hat sich die agile Softwareentwicklung herausgebildet. Im "Manifest der agilen Softwareentwicklung" von Cockburn werden die agilen Werte vorgestellt. Sie bilden die Basis der agilen Softwareentwicklung und können wie folgt zusammengefasst werden:

1. Menschen und Kooperationen bzw. Interaktionen sind wichtiger als Werkzeuge und starre Prozesse.
2. Funktionsfähige Software ist wichtiger als eine umfassende Dokumentation.
3. Eine stetige Zusammenarbeit mit dem Auftraggeber bzw. Kunden ist wichtiger als bürokratische Vertragsverhandlungen.
4. Offenheit für neue Erkenntnisse und damit Bereitschaft zu dynamischen Veränderungen sind wichtiger als das strikte Festhalten an einem festgelegten statischen Plan.

Die agile Softwareentwicklung hat verschiedene Varianten. Die Bekanntesten sind das Extrem Programming (XP) und SCRUM. Wie der Name schon vermuten lässt, liegt bei XP