Kapitel 6

Softwaretechnik: Entwicklung von Softwaresystemen

6.1 Phasenmodell

Im Gegensatz zur Entwicklung relativ kleiner Programme bzw. Softwaresysteme ergeben sich bei der Entwicklung großer Systeme neue, unerwartete Aspekte, Hürden und Aufgaben.

klein : Problem überschaubar, ein Entwickler, kurze Entwicklungszeiten (Wochen), Software nicht sehr umfangreich, Lebensdauer kurz, Anforderungen so gut wie fest, so gut wie kein sozialer, gesellschaftlicher Kontext.

groß : Problem komplex, mehrere Entwickler, Software sehr umfangreich, umfangreiche Software, lange Entwicklungszeiten (PersonenJahre), lange Lebensdauer, sich ändernde Anforderungen, mehrere beteiligte Personengruppen mit unterschiedlichen Interessen, usw.

Die Erfahrungen, die man beim Entwickeln kleiner Softwaresystem macht, lassen sich oft nicht auf große übertragen. (kein "Scaling up"). Dies betrifft fast alle Bereiche: sowohl die Entwicklung selbst als auch die Eigenschaften des fertigen Systems und dessen Verwendung.

Einige Aspekte dabei sind (Liste kann erweitert werden):

- Mehrere Anwender und Entwickler sind beteiligt: Kommunikation und Organisation (Management ist notwendig).
- Die Anforderungen an das System sind nur bis zu einer bestimmten Detaillierung klar bekannt.
- Die Menge der zu verarbeitenden und zu speichernden Daten muß analysiert werden (Mengengerüst).
- Während der langen Entwicklungszeit kann es u.a. neue verfügbare Hardware geben. Ebenso könnten sich die Anforderungen ändern: Z.B.
  - ECU-Umstellung

  - neue Gesetze
  - System mehrsprachig
  - andere technische Anforderungen: Client /Server statt Mainframe
  - ...

- Lange Lebensdauer von Software kann unerwartete Probleme mit sich bringen (Jahr 2000-Problem)
- Große Softwaresystem sind nie fehlerfrei.
- Software unterliegt einem Alterungsprozeß (relativ zur Umgebung).

Man unterscheidet die folgenden Phasen beim Erstellen kommerzieller Software-Systeme (nach H. Balzert, Softwareentwicklung)

- Planung (Anforderungsermittlung, Analyse)
- Definition
- Entwurf
- Implementierung
- Validierung
- Einführung
- Wartung und Pflege

Diese Phasen kann man als zeitlich nacheinander ansehen (sog. Wasserfallmodell). I.a. ist dies auch in anderer zeitlichen Abfolge möglich: es werden Phasen mehrfach durchlaufen. bzw. parallel mit verschiedenem Dataillierungsgrad.

Einige Bemerkungen zu den Phasen:

Planung (Anforderungsermittlung, Analyse) Anforderungen aus der Anwendung werden informell ermittelt. Verschiedene Kriterien werden in dieser Stufe verwendet: Kosten / Nutzen. Technische Machbarkeit. Soziale Aspekte

Definition Technische, planerische und organisatorische Entscheidungen werden getroffen. Fachlicher Funktionsumfang; die zu modellierenden Aspekte , Mensch-Rechner Interaktion, Benutzerfreundlchkeit.

Entwurf Die Architektur des Softwaresystems wird entworfen: Einzelkomponenten, Schnittstellen zwischen den Komponenten, Ein- /Ausgaben.

Implementierung Dies schließt neben der Implementierung auch den Entwurf kleinerer Einheiten ein.

Meist sind Standards und Konventionen einzuhalten, die firmenspezifisch sein können. Die Implementierung schließt die Tests auf technische Korrektheit mit ein.

Validierung Hier wird überprüft, ob das System den Spezifikationen genügt. Die Qualitätsprüfung schließt im allgemeinen mit ein, die bisher erstellten Dokumente (Dokumentation) der verschiedenen Phasen auf Konsistenz zu prüfen. Folgende Teststufen sind zu durchlaufen und auf Erfolg zu prüfen:

- Modultest
- Integrationstest
- Systemtest
- Abnahmetest (mit Produktionsdaten)

Parallel dazu sind in jeder Stufe auch Effizienzmessungen und -prüfungen notwendig, um die Folgekosten zu kennen, und evtl. Schwachstellen zu erkennen (Optimieren vs. neue Rechner kaufen, mehr Plattenplatz, ...)

Einführung Übertragung von der Entwicklungsumgebung in die Produktionsumgebung. Schulung der Benutzer. Benutzerdokumentation. Umstellung des Arbeitsprozesses der Anwender.

Wartung und Pflege Evtl. Fehlerbereinigung bei bereits eingeführtem System. Kleinere Änderungen und Verbesserungen. Zusatzfunktionalität in geringem Umfang. Anpassung an neuere Versionen der Basissoftware oder Hardware (neues BS) . Dokumentation der Änderungen (im Programm, zug. Dokumente).

Problem: Allmähliche Zerstörung der Softwarestruktur. Änderungen können die ursprünglichen Prinzipien nach und nach untergraben.

Diese Phase wird i.a. von allen Beteiligten unterschätzt, gilt als wenig attraktiv. Es zeigt sich, daß die Lebensdauer von Software länger sein kann als man vor 15 Jahren erwartet hat.

Typische Situation: Ein System soll an eine Anforderung angepaßt werden, aber keiner der ursprünglichen Entwickler ist verfügbar.

Faustregel: 80/20: 80% des Funktionsumfanges sind mit 20% des Aufwandes zu erledigen. Die restlichen 20% erfordern 80% des Aufwandes.

Dies ist eher qualitativ als quantitativ zu verstehen. Es kann noch schlimmer sein. Generell verursacht die Behandlung von (seltener auftretenden) Ausnahmen und Sonderfällen einen hohen Aufwand, während dieStandardverarbeitung meist ein einfacheres Modell hat.

Um diese Ausnahmen leicht in ein Modell zu integrieren ist es unbedingt ratsam, ein recht genaues Modell zu entwerfen und die Software / Daten diesem Modell entsprechend zu strukturieren.

Beispiel 1. Eine Datenbank von Artikeln und Preisen erfordert die Attribute Artikel-Nr., Beschreibung, Anzahl im Lager, Preis, ... Damit kann man Rechnungen schreiben. Sollte sich der Preis erhöhen, wird das entsprechende Attribut angepasst.

Diese Vorgehensweise läßt außer acht, daß auch Anfragen kommen könnten wie Reklamation, die erfordern, die Preise zu einem früheren Zeitpunkt zu rekonstruieren. Bzw. die Eingabe zukünftiger Preiserhöhungen (Rabatte, Sonderangebote).

D.h. man muß statt einem aktuellen Bild der Preise die Geschichte der Preise (bis zur Gegenwart) modellieren.

6.1.1 Vorgehensmodelle

Wasserfallmodell: Die Phasen sind zeitlich nacheinander. Dies ist i.a. zu naiv.

Spiralmodell Alle Phasen werden in einem Zyklus durchlaufen, wobei in jedem Zyklus das System entsprechend den Erfahrungen und dem Gelernten verbessert unf verfeinert wird. Dies kann teilweise Neustrukturierung mit einschließen.

Zyklische Modelle Softwaresystem leben als Folge von Versionen. Für jede Version wird ein dem Wasserfallmodell analoger Zyklus durchlaufen.

6.1.2 Prototyping

Dies ist ein Verfahren zu Softwareentwicklung, bei dem zunächst einen Prototyp entwickelt wird, der von allen Beteiligten bewertet und evtl. revidiert wird. Ein Prototyp bedeutet hierbei ein vereinfachtes aber funktionsfähiges, schnell zu implementierendes und leicht änderbares System, das die interessierenden Aspekte modelliert. Das kann sein: Dialogverhalten, technische Funktionalität, ...

Vorteil dieser Vorgehensweise ist einerseits eine einfachere Kommunikation mit dem Anwender, da das System vorführbar und (rudimentär) benutzbar ist. Ebenso werden bei der Implementierung schon Erfahrungen in der Umsetzung gesammelt: auftretende Probleme werden früher erkannt, man kann verschiedene Designalternativen ausprobieren. Man unterscheidet exploratives, experimentelles, evolutionäres Prototyping.

6.1.3 Beispiel: Dialogprogrammierung

Die kleinste Einheit, die man zu betrachten hat, ist ein Eingabeformular (Eingabemaske), das dem Anwender am Bildschirm (in einem Fenster) angezeigt wird, und ein zugehöriges Programm, das auf diese Eingaben reagiert.

Zu beachtende Aspekte sind:

- Ergonomisch sinnvolle Gestaltung des Formulars: Anzahl der Eingabefelder, des Erklärungstextes, Hilfefunktionen, ...

  Logischer Zusammenhang der Eingabedaten und Anordnung.

- Reaktionen auf die Eingabe. Das zugehörige Programm muß auf alle Benutzereingaben / Interaktionen / Ereignisse eingerichtet sein, und sinnvoll (in Sinne des Anwenders) reagieren: Gute Fehlermeldungen, korrekte Reihenfolge der Konsistenzprüfungen der Inhalte. Das Programm sollte keine für den Benutzer überraschenden Aktionen ausführen: z.B. nach mehrfachem Vertippen und Fehlern Löschen bzw. Ändern eines Eintrages in einer Datenbank (Löschen eines Files). Bei wichtigen Aktionen sollte der Benutzer benachrichtigt werden, ...

- Die Konventionen und Standards sollten eingehalten werden. Aussehen des Formulars, Bezeichnungen, Verhalten + Benutzung des Programms, usw.
- Die Dialogführung sollte nicht zu starr sein: Schlecht ist z.B.: Folge von erzwungenen Aktionen, aus denen es für den Anwender kein Entkommen gibt.
- Sinnvoll kann es sein, je nach Anwender: (Anfänger, Fortgeschrittener, Experte) Abkürzungen bzw. Alternativen vorzusehen.

6.2 Testen von Programmen

Literatur: Myers: Methodisches Testen von Programmen,
Oldenbourg Verlag,
Parrington, Roper, Software Test , McGrawHill

Programme und Programmsysteme sind im allgemeinen nicht fehlerfrei. D.h. das Ziel, ein absolut fehlerfreies Software-Paket zu entwickeln, ist i.a. nicht zu erreichen.

Trotzdem benutzen wir eine immer größere Menge an Software, die, wenn man ein gutes Produkt hat, nur sehr selten fehlerhaft reagiert.

Eine wichtige Methode, diese immer vorhandenen Fehler zu entdecken, ist der **Test**.

Hierunter versteht man die Überprüfung eines Softwaresystems auf korrekte Funktionsfähigkeit.

> Testen ist der Prozeß, ein Programm mit der Absicht auszuführen,
> Fehler zu finden.

Im oben angegebenen Phasenmodell der Softwareentwicklung haben wir gesehen, daß abhängig von der Phase das Testen verschiedene Aktionen erfordert, auch z.B. das Durchlesen, Verstehen und Prüfung auf Konsistenz von Dokumenten.

Wir wollen hier eher die technische Seite betrachten und meinen mit Test die Überprüfung eines gegebenen Programms oder Programmsystems mit verschiedenen Methoden auf korrekte Funktionsfähigkeit. Im folgenden sollen verschiedene wichtige Methoden und Aspekte vorgestellt werden.

Zunächst mal gibt es verschiedene Größenordnungen der zu testenden Systeme:

- Unterprogramm (oder Funktion) mit einfacher Funktionalität (Summe eine Liste, Lesen eines Satzes aus einer Datenbank, ...)
- Programm ohne Interaktion ausser einer Eingabe am Anfang und Ausgabe am Ende: (compress, Compiler, ...)

- interaktive Programme ohne (mit wenig) Gedächtnis (Taschenrechner,)
- Programme aus vielen Modulen mit viel Interaktion: Betriebssystem (Linux, MSWindows, ...)
- Anwendungen mit Datenbanken im Hintergrund, bestehend aus vielen Programmen, verteilt, ... (Flugreservierung, Kontoverwaltung bei Banken)

6.2.1 Was ist ein Fehler

“Das ist kein Fehler, sondern ein feature“

Oft sind Programme / System so komplex, daß es keine (formale) vollständige Definition für das korrekte Verhalten des Systems gibt. d.h. man kann auch den Begriff “Fehler“ nicht formal genau definieren. Meist ist der Begriff korrekte / nicht korrekte Funktionsweise zumindest auf Modul- oder Funktionseben genau festgelegt.

Bei größeren Einheiten kann diese Definition etwas informeller sein.

Trivial nicht korrektes Verhalten wäre: Programmabsturz (d.h. Typfehler, inkompatibles Datenformat, falsche Adressierung, fehlerhaftes Ansprechen des Betriebssystems oder von Dateien), Nichtterminierung, Daten verschwinden im System, ...

Problematisch ist:

- Ineffizienz (lange Wartezeiten bei Interaktionen)
- unlogisches/unerwartetes Verhalten bei interaktiven Programmen.
- Programmabbrüche bei knappen / fehlenden Ressourcen

6.2.2 Blackbox/Whitebox-Test

Von einem **Blackbox-Test** spricht man, wenn man den Quelltext des Programms nicht kennt bzw. nicht benutzt, aber das Programm verwenden darf, d.h. nur das Ein-Ausgabeverhalten wird berücksichtigt.

In diesem Fall sieht man, daß es i.a. nicht möglich ist, in vernünftiger Zeit alle Fälle zu testen, da z.B. ein Taschenrechnerprogramm eine verborgene Abfrage haben könnte, bei genau der Berechnung $2000 * 2000$ den Wert 1 statt 4000000 zu berechnen. Das bedeutet, das z.B. ein einfaches Berechnungsprogramm für Additionen bei vollständigem Black-Box Test und bei 10-stelligen Zahlen ca. 10^{20} Testfälle erforderlich macht: alle möglichen Additionsaufgaben der Form a+b. Wenn man unbeschränkte ganze Zahlen hat (Haskell-integer), bräuchte man sogar unendlich viele Testfälle.

Im allgemeinen geht man pragmatisch vor und wählt eine Menge von Testfällen aus (beschränkt auf nichtnegative Zahlen) :

- typische Fälle: $2 + 2, 0 + 999, 9999 + 9999$
- Grenzfälle: “maximale Zahl“ + 1, 0 + 0, 9999 + 0, (max/2) + (max/2)

- zufällige Eingaben

Whitebox-Test Hierbei kennt man das Programm und die gewünschte Funktionalität. Man kann somit anhand der Quelle systematisch Testfälle entwerfen und damit den Test durchführen. Dies ist natürlich abhängig von der verwendeten Programmiersprache.

Testen aller *Auswertungs-Pfade* durch ein Programm. Beim funktionalen Programmieren wäre das die Folge der Ausdrücke, die in normaler Reihenfolge ausgewertet werden. Bei Java die Anweisungen, die bei einem aufruf einer Methode durchlaufen werden. Ein *Auswertungspfad* soll hierbei die Folge der Anweisungen sein, die von der Eingabe zur Ausgabe durchlaufen werden. Hierbei können natürlich Anweisungen mehrfach durchlaufen werden.

Dieser Test kann i.a. nicht vollständig durchgeführt werden, da die Anzahl der verschiedenen Pfade kombinatorisch zu groß ist. Außerdem ist auch dieser Test nicht vollständig

Ein einfaches (schwaches) Kriterium ist das der Anweisungsüberdeckung: Die Testdaten sollen dazu führen, daß jede Anweisung des Programms wenigstens einmal ausgeführt wird.

Top-Down, Bottom-Up Testen

Diese beiden gegensätzlichen Methoden kann man leicht in Haskell veranschaulichen, mit ihren Vor- und Nachteilen.

Bottom-Up: Zuerst werden die Basisfunktionen getestet, dann die darauf aufbauenden Funktionen (Nicht ganz so einfach, wenn man verschränkt rekursive Funktionen hat).

Sinnvoll, wenn man sich auf die Basisfunktionen 100% verlassen muß.

Top-Down: Man testet die Hauptfunktion, wobei evtl. noch nicht alle Unterfunktionen voll ausprogrammiert sind, sondern statt desen Dummyfunktionen vorhanden sind.

Testfälle, Testdaten

Ein Testfall besteht aus einer Menge von Eingaben und den erwarteten Ausgaben. Die Ausgaben sollten vorher festgelegt werden, da man sonst leicht in die psychologisch Falle tappt: "es wird schon stimmen".

Wie schon erwähnt, kommt es bei der Erstellung einer Menge von Testfällen darauf an, daß diese möglichst alle Auswertungspfade erfassen, d.h. die Testdaten sind nicht zufällig gestreut, sondern sollten anhand des Programmes erstellt werden. Da man nicht alle Pfade abdecken kann, muß man eine Auswahl treffen.
Abdecken aller Fallunterscheidungen: Testfälle sind so zu strukturieren, daß jeder Zweig einer Fallunterscheidung (if-then-else) einmal erreicht wird. Auch das kann zu einer sehr großen Zahl von Fällen führen. Um diese Anzahl klein zu halten, kann man versuchen, eine Äquivalenzrelation zu finden: "findet gleiche Fehler", und dann aus jeder Äquivalenzklasse genau einen Fall zu nehmen.

Da diese Anzahl zu groß sein kann, muß man oft ein Auswahl von Hand treffen. Folgende Kriterien sollte man bei den Testfällen beachten:

Bei den Testfällen sind nicht nur die sinnvollen Eingaben zu nehmen, sondern auch eher abwegige, damit auch die falschen Eingaben getestet werden.

Man sollte neben normalen Fälle insbesondere solche Eingabe-konstellationen konstruieren, die Grenzfälle sind, z.B. 0, oder beim Test der Jahr-2000-Anpassungen: 1998, 1999, 2000, 2001.

Datentypfehler

Diesen Fehlertyp muß man in vielen Programmiersprachen testen, da die Sprachdefinition solche Situationen erlaubt und der Kompiler diese nicht vollständig zur Kompilierzeit erkennen kann. Typchecking, wie es beispielsweise in Haskell eingebaut ist, kann eine Menge Zeit und Testen ersparen, da dieser Fehler nicht auftreten kann.

Regressionstest

Ein Regressionstest ist ein Test, der nach einer Änderung / Verbesserung des Programms mit den gleichen Testdaten vor und nach der Änderung durchgeführt wird, und zu gleichen Ergebnissen führen muß. Es ist oft sinnvoll, sich die während der Entwicklungszeit erzeugten Testfälle aufzuheben und später bei Änderungen in diesem Sinne zu verwenden.

Paralleltest

ähnlich zum Regressionstest wirkt der sogenannte Paralleltest, der die Ergebnisse eines Programsystems vor der Änderung mit dem nach der änderung vergleicht. Wenn keine inhaltlichen Veränderungen vorgenommen wurden, sollten die Ergebnisse gleich sein und können automatisch geprüft werden. Ansonsten muß man noch Vergleichsroutinen schreiben, die die Ergebnisse vergleichen. Z.B. beim Jahr-2000-Problem muss man in der Ausgabe evtl. die Veränderungen der Variable des Jahres beachten.

interaktive Programme, Datenbankzugriffe

Hier ist es oft technisch schwierig (oder erfordert Werkzeuge), die Eingabefälle vorher genau spezifizieren, und diese automatisch ausführen zu lassen. Einerseits ist die Menge an Eingaben, die ein Benutzer an der Tatstatur machen kann, sehr groß. Das Verhalten des Programms kann auch von vorhergehenden Eingaben abhängen. Noch schwieriger wird es, wenn das richtige Programmverhalten noch von Daten in einer Datenbank abhängt oder von parallelen Zugriffen anderer Benutzer.

Fehlerfallen Robustes Programmieren oder auch die Verfolgung von Fehlern, die scheinbar zufällig auftreten (in interaktiven Programmen, Betriebssystemen, ...) kann man durch intelligentes Fallenstellen manchmal einfangen. Dies sind Teile eines Programms, die nur dazu dienen, in bestimmten Fällen (im laufenden Betrieb) einen Fehler der Programmlogik zu signalisieren. Ein einfaches Beispiel ist der else-Zweig von eigentlich vollständigen Fallunterscheidungen, in die man eine Anweisung schreiben sollte, die dies signalisiert.

Profiling Dies ist eine Methode, die statistische Aussagen über den Ablauf von Programmen macht, z.B. welche Anweisungen wie oft durchlaufen wurden, welche Zugriffe gemacht wurden usw. Aus diesen Daten kann man teilweise "Fehler" ablesen, die man durch übliches Testen nicht findet. Z.B.

- wenn Routinen (Funktionen, Module) aufgerufen werden, die keine Wirkung haben.

- wenn Auswertungen mehrfach gemacht werden, obwohl dies nicht notwendig ist.
- Wenn Routinen (Funktionen, Module) nicht aufgerufen werden, obwohl dies passieren sollte (etwa Konsistenzprüfungen von Daten)

Im Glasgow-Haskell-Compiler wurde erst durch Profiling herausgefunden, daß Teile der Symboltabelle nicht gespeichert wurden, sondern bei jeder Verwendung neu berechnet wurden.

Bemerkung
Studie aus den USA: Fehler der USA-Programmierer 1,69 Fehler / KLOC (kilolines of code)
Fehler der Progranmmierer in Deutschland 2,2 Fehler / KLOC,
Mittelwert 5,23 Fehler /KLOC.

6.2.3 Fehlerbehebung (Debugging)

Wenn ein Fehler oder mehrere Fehler gefunden sind, so kann man anfangen den Fehler zu beheben. Dazu muß man zunächst den Fehler verstehen. Bei einfachen Fehlern muß die fehlerhafte Stelle (die Stellen) im Programm lokalisiert werden. Diese Lokalisation kann durch Nachdenken, durch Benutzung eines Werkzeuges (Debugger), durch Einfügen von Druckausgaben gemacht werden. Letzere Methode ist als letztes Mittel gut zu verwenden, aber i.a. ineffizient.

Bei komplizierteren Fehlern ist oft eine **genauere logische Analyse** sinnvoll, damit man den Fehler genau versteht, bevor man Änderungen macht. Wichtig ist auch, bei der Änderung die Ideen aus der Designphase des Programms (-systems) sich in Erinnerung zu rufen bzw. (aus der Dokumentation und dem Programm) zu rekonstruieren.

Eine zu schnelle Änderung kann dazu führen, daß nur das Symptom, aber nicht der Fehler behoben wird. Insbesondere kann es sein, daß die Änderung selbst neue Fehler provoziert.

Eine korrekte Fehlerkorrektur kann dazu führen, daß schlafende (bisher verdeckte) Fehler geweckt werden.

6.2.4 Theoretische Grenzen

Wir wissen schon, daß das Halteproblem für Turingmaschinen unentscheidbar ist. Daraus läßt sich für das Testen von Programmen ableiten, daß es keine algorithmische Testmethode gibt, die gegebene Programme daraufhin untersucht, ob diese schleifenfrei sind. Das gleiche gilt für das Überprüfen der Ein-/Ausgabe-Funktion.

Studie aus den USA: Fehler: USA 1,69 Fehler / KLOC; Deutschland 2,2 Fehler / KLOC, Mittelwert 5,23 Fehler /KLOC.