Ausgewählte Kapitel der Software-technik

Skriptum zur Vorlesung

Ausgewählte Kapitel der Softwaretechnik

Dr. Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
76128 Karlsruhe

Sommersemester 1999

Stand: 13. April 1999

Kurzübersicht

Wir werden in dieser Vorlesung eine Reihe von Themen behandeln, die in der „Softwaretechnik“ gar nicht oder zu knapp behandelt werden, aber dennoch einen erheblichen Stellenwert für die Planung, Erzeugung und Einführung von Softwaresystemen und den Betrieb einer softwareerzeugenden Organisation haben. Voraussetzung zu dieser Vorlesung ist ein solides Grundwissen in Informatik sowie die Kenntnis der Vorlesung „Softwaretechnik“.

Die Vorlesung umfaßt drei große Kapitel: Grundlagen der Softwaretechnik, der Software-Prozeß, Techniken. Jedes dieser Kapitel ist in mehrere Abschnitte unterteilt, von denen jeder ein eigenes Thema behandelt.

Das **Kapitel „Grundlagen“** hat vier solche Abschnitte: In Abschnitt 1 sprechen wir über die Basis einer wissenschaftlichen Softwaretechnik-Disziplin, nämlich das **Messen und Quantifizieren** dessen, worüber man etwas wissen möchte. Zu messen gibt es in der Softwaretechnik zahlreiche Eigenschaften der erstellten Produkte und der Prozesse, die diese Produkte herstellen. Die Messungen haben das Ziel, Voraussagen zu machen oder Beurteilungen zu objektivieren. Der Abschnitt 2 bildet sozusagen einen Gegenpol: Er stellt einige grundlegende **Ergebnisse der psychologischen Forschung** dar, die für die Softwaretechnik relevant sind. Diese Ergebnisse erlauben ebenso wie die in Abschnitt 1 besprochenen Messungen die Beurteilung von Organisationen und Prozessen und die Vorhersage von Ereignissen, jedoch nur in einer „weichen“, nicht quantitativen Weise. Dennoch sind die entsprechenden psychologischen Erkenntnisse Fakten, und es ist sinnvoll und nützlich für Softwareingenieure, sie zu kennen. Abschnitt 3 bespricht eine Reihe von Prinzipien und Themen der **Softwaretechnik-Forschung** und zeigt, unter welchen Umständen die Beurteilung der Softwaretechnik als „Laberfach“ unangebracht ist. Der Abschnitt 4 bespricht Fragen der **Systemsicherheit** und verdeutlicht, daß man in Fragen der Sicherheit über die Softwareteile eines Systems hinausblicken und Systeme im ganzen betrachten muß. In diesem Abschnitt werden auch Vorgehensgrundsätze für Sicherheitsmaßnahmen und eine Reihe von Fallbeispielen spektakulärer Sicherheitsprobleme beschrieben.

Das **Kapitel „Software-Prozeß“** hat drei Abschnitte. Im ersten (Abschnitt 5) geht es um allgemeine Überlegungen und Modelle zur **Softwareprozeß-Verbesserung**, insbesondere um CMM und ISO 9000. Diese Überlegungen beziehen sich auf ganze Organisationen und größere Projekte. Im Abschnitt 6 wird ein konkreter Ansatz zur kontinuierlichen Verbesserung der Softwareprozesses auf der Ebene des einzelnen Softwareingenineurs diskutiert (**„Der persönliche Softwareprozeß“**, PSP). Abschnitt 7 bespricht die **Cleanroom-Methodik**, einen Ansatz, der von seiner Ausrichtung her zwischen diesen Extremen liegt und die effiziente Produktion von Software hoher Zuverlässigkeit zum Ziel hat.

Das **Kapitel „Techniken“** umfaßt bislang lediglich einen Abschnitt, und zwar Abschnitt 8 zum Thema **Anforderungsanalyse**, die oft die entscheidende Phase einer Softwareentwicklung darstellt, weil von ihr die größten Probleme ausgehen.

Ein **Anhang** zu diesem Skriptum enthält eine Anleitung zum Schreiben technischer Dokumente, ein Thema, das zwar in der Softwaretechnik eine sehr wichtige Stellung einnimmt, aber leider im Rahmen einer Vorlesung kaum sinnvoll gelehrt werden kann.

In jedem Abschnitt wird zunächst eine globale Einordnung des Themas vorgenommen, dann werden die Ziele und die spezifischen Ansätze, Probleme und Hindernisse diskutiert, schließlich werden konkrete Techniken vorgestellt.

Ich bin für konkrete (und am besten konstruktive) Hinweise auf Ungereimtheiten oder zur Verbesserung des Skripts jederzeit dankbar. Eine aktuelle Fassung des Skripts ist jeweils im WWW unter http://wwwipd.ira.uka.de/~prechelt/swt2 verfügbar.

Inhaltsverzeichnis

Abschnitt 1

Messen und Maße

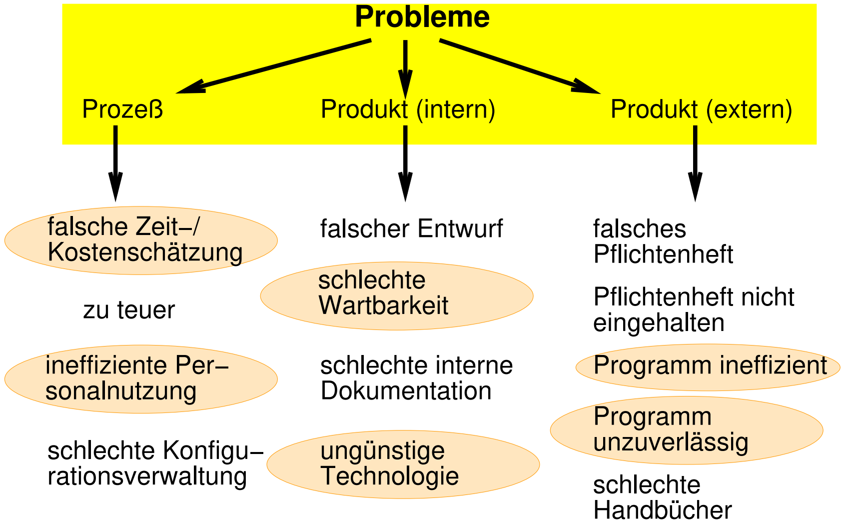

Abbildung 1.1: Vom Messen angesprochene Problembereiche.

1.1 Einordnung und Ziele

Messen ist ein Prozeß, nämlich die Anwendung eines Maßes auf ein Objekt. Ein **Maß** ist eine Abbildung: die objektive Zuweisung einer Zahl oder eines anderen Symbols an ein Objekt zur Charakterisierung eines bestimmten Attributs des Objekts.

Im Bereich der Softwaretechnik ist der Begriff der „**Metrik**" sehr gebräuchlich. Dieser Begriff stammt ursprünglich aus der Mathematik und bezeichnet symmetrische Abbildungen von Paaren von Objekten auf nichtnegative reelle Zahlen mit dem Ergebnis Null nur für identische Objekte. Das, was in der Softwaretechnik im allgemeinen unter Metrik verstanden wird, ist jedoch eher das, was die Mathematik als eine Norm bezeichnet. Der Begriff Metrik wird aber in der Softwaretechnik oft so lax verwendet, daß er überhaupt nicht mehr eine wohldefinierte Sache bezeichnet, sondern nur ein vages Konzept. In seinen „ordentlichen" Verwendungen ist das Wort Metrik synonym zu dem Wort Maß (measure), das wir in diesem Kapitel zugrundelegen.

Das unmittelbare Ziel des Messens in der Softwaretechnik ist es, die Beschreibung und Beurteilung von Softwareprodukten und ihren Teilen sowie von Softwareproduktionsprozessen und ihren Teilen, Techniken und Werkzeugen zu objektivieren. Das mittelbare Ziel ist in fast allen Fällen eine Vorhersage. Der hinter diesem Ziel stehende Zweck liegt in der Verbesserung dieser Produkte und Prozesse, die auf den durch Messungen gewonnenen Informationen in vielfältiger Weise aufbauen kann. Vor Einsatz eines Maßes muß der jeweilige Zweck genau bekannt sein.

Grundsätzliche **Anwendungen des Messens** (und ihre Zwecke) sind:

1. **Beschreibung**. Wir wollen kompakte und objektive Beschreibungen einzelner Aspekte von Produkten und Prozessen als Grundlage für Beurteilungen oder Planungen bekommen.

2. **Beurteilung**. Manche Messungen eignen sich direkt zur Beurteilung der Qualität, Effektivität oder Effizienz von Produkten oder Prozessen. Solche Beurteilungen sind direkt Ansatzpunkt für Verbesserungen.

3. **Vorhersage**. Objektivieren und Quantifizieren sind auch Voraussetzungen dafür, eine geordnete Planung vorzunehmen. Meßergebnisse lassen sich wesentlich zuverlässiger auf die Zukunft übertragen als Ahnungen, Eindrücke, Erfahrungen, Gefühle und Vermutungen. Das liegt zum einen am objektiven Charakter von Messungen, zum anderen daran, daß sie erheblich vom jeweiligen Einzelgegenstand abstrahieren.

1.2 Ansätze

1.2.1 Skalen

Eine sehr wichtige Unterscheidung zwischen verschiedenen Maßen betrifft die Art der für den Meßwert verwendeten Skala:

- **Nominalskala**. Eine Nominalskala ist qualitativ. Als Meßwert wird eines aus einer endlichen Menge von Symbolen verwendet. Zwischen diesen Symbolen ist kein Zusammenhang definiert. Z.B. läßt sich die Programmiersprache, in der ein Programm geschrieben ist, nur auf einer Nominalskala angeben. Bei Nominalskalen spricht man meist nicht von Messen, sondern von Kategorisieren.

- **Ordinalskala**. Eine Ordinalskala ist ebenfalls qualitativ, jedoch ist auf den Werten eine Totalordnung definiert. Die Antwortmöglichkeiten in Fragebögen haben oftmals diese Art, z.B. „zufrieden“, „teilweise zufrieden“, „unzufrieden“.

- **Intervallskala (Differenzskala)**. Eine Intervallskala ist die schwächste Art einer quantitativen Skala. Sie läßt als einzige Operation auf Meßwerten die Differenzbildung zu. Die Temperatur wird z.B. üblicherweise auf den Intervallskalen Celsius oder Fahrenheit gemessen.

- **Verhältnisskala**. Auf einer Verhältnisskala hat zusätzlich der Nullpunkt eine ausgezeichnete Bedeutung; man kann dann auch Verhältnisse bilden, z.B.: dieses Programm ist doppelt so lang wie jenes.

- **Absolutskala**. Der Meßwert gibt eine Anzahl an. Dies ist einerseits ein Spezialfall der Verhältnisskala (denn dort sind evtl. auch gebrochene und negative Meßwerte zulässig), andererseits hat auf einer Absolutskala neben der Null auch die Eins eine ausgezeichnete Bedeutung.

Es ist zu beachten, daß die Skala eine Eigenschaft eines Maßes ist, nicht eine Eigenschaft der zu messenden Größe. So ließe sich die Zufriedenheit auch auf einer Verhältnisskala messen, indem man z.B. den Geldbetrag mißt, den jemand bereit wäre, für den Erhalt eines Zustandes zu bezahlen, den man vermessen möchte. Auch die Temperatur läßt sich auf einer Verhältnisskala angeben: den Kelvingraden.

Die Skala bestimmt, wie oben angedeutet, welche Operationen man auf den Meßwerten sinnhaltigerweise ausführen kann und welche nicht. Demnach sind gewisse Aussagen sinnleer, z.B.:

- Man kann nicht sinnvoll von „20 Prozent Verbesserung der Qualität“ sprechen, wenn man nicht festlegen kann, was Qualität Null sein sollte, denn relative Verbesserungen setzen eine Verhältnisskala voraus.
- Wenn jemand sagt, „diese Kunden sind doppelt so zufrieden wie jene“, dann hat er wahrscheinlich die 5 Zufriedenheitsstufen eines Fragebogens in die Zahlen 1 bis 5 umkodiert und munter damit herumgerechnet. Der Unsinn tritt zutage, wenn man fragt „*wie* zufrieden sind denn jene“, denn die Antwort würde dann beispielsweise lauten: „zwei komma eins“. Das ist reiner Unsinn.

Die meisten Fehler beruhen darauf, daß wir gerne, wenn wir Zahlen sehen, sofort eine Verhältnisskala als gegeben annehmen, etwa bei der Aussage „heute ist es doppelt so warm wie gestern“. Das mag bei heute 20 Grad Celsius und gestern 10 Grad Celsius ja gut klingen. Jemand, der die Temperatur in Fahrenheit zu messen gewöhnt ist, würde aber wohl kaum zustimmen, daß 68 Grad Fahrenheit doppelt so warm seien wie 50 Grad Fahrenheit — das ist aber das Gleiche. Und ein Physiker könnte beide darüber aufklären, daß die Temperatur in Wirklichkeit nur um dreieinhalb Prozent zugenommen hat: von 283 Kelvin auf 293 Kelvin; denn der Nullpunkt der Kelvinskala ist tatsächlich ein Nullpunkt (keine thermische Energie mehr vorhanden).

1.2.2 Anforderungen an Maße

Es lassen sich eine Reihe von Anforderungen an Maße formulieren, die erfüllt sein sollten, damit ein Maß nützlich ist. Wie wir unten noch sehen werden, sind leider bisher die Versuche, solche Anforderungen zu formalisieren, weitgehend gescheitert. Deshalb lassen sich nur sehr allgemeine und ungenaue Anforderungen angeben.

- **Wohldefiniertheit**. Ein Maß sollte eine vollständige und genaue Definition besitzen. Diese Definition muß fehlerfrei zwischen Benutzern des Maßes kommunizierbar sein.
- **Aussagekraft**. Ein Maß sollte einen Wert ergeben, der Information über eine Größe liefert, welche tatsächlich von Belang für die Softwareentwicklung ist. Das erscheint selbstverständlich, wird aber verblüffend oft mißachtet.
- **Sinnhaltigkeit**. Diese Werte sollten in ihrer Größe dem Sinn der Messung entsprechen. Dies ist der Kern eines Maßes überhaupt: Wenn die Meßwerte nicht die gemessene Eigenschaft qualitativ gleichartig abbilden, ist das Maß wertlos. Zum Beispiel ist ein Größenmaß sinnlos, das offensichtlich dramatisch unterschiedlich großen Objekten dieselbe Größe zuweist und erst recht ist es sinnlos, wenn es gar dem größeren Objekt in manchen Fällen eine kleinere Größe zuweist. Diese Anforderung enthält insbesondere die Forderung aus der Meßtheorie, die dort **Repräsentationsbedingung** genannt wird: Die Abbildung von Objekten auf Meßwerte muß die Relationen zwischen den Objekten (im Bezug auf das gemessene Attribut) in dem Sinne erhalten, daß es ihnen entsprechende Relationen zwischen den Meßwerten gibt, die stets gleich ausfallen.

- **Genauigkeit**. Das Maß sollte hinreichend genau bestimmbar sein, damit Messungen zuverlässig genug sind, um sich auf sie zu verlassen.

- **Berechenbarkeit**. Die Berechnungsvorschrift für das Maß muß algorithmisch und berechenbar sein.

- **Praktische Durchführbarkeit**. Die zur Berechnung eines Maßes nötigen Eingabedaten sollten verfügbar sein und zwar zu der Zeit, zu der wir am Meßergebnis interessiert sind.

Dabei müssen wir uns eventuell mit dem Besten zufriedengeben, was wir bekommen können. So sind wir unter Umständen insbesondere bereit, Abstriche bei der Wohldefiniertheit, der Aussagekraft, der Genauigkeit und der Berechenbarkeit zu machen, weil wir entsprechende Maße einfach nicht kennen. Ein Beispiel ist die Zuverlässigkeit von Software. Zu ihrer Bestimmung sind wir meist auf Beobachtungen beim Einsatz der Software angewiesen, die ungenau sind und eigentlich zu spät Ergebnisse liefern. Ein anderes Beispiel ist die Komplexität von Software: Wir kennen kein Maß, das diese zufriedenstellend mißt; also benutzen wir Ersatzmaße, (z.B. das McCabe-Maß, das im wesentlichen die Zahl von binären Abfragen im Programm angibt), deren Aussagekraft eigentlich gering ist.

Zusammenfassend könnte man sagen, die wichtigste Anforderung an ein Maß ist seine **Zweckmäßigkeit**: In dem Spannungsfeld zwischen wünschenswerten Eigenschaften und durchführbaren Messungen ist die Lösung angebracht, die mir am besten die gewünschte Information liefern kann, also den besten Kompromiß zwischen Kosten und Qualität der Daten darstellt. Wie man solche Maße findet, werden wir kurz in einem getrennten Abschnitt über die Goal/Question/Metric–Methode besprechen (Abschnitt 1.2.4).

1.2.3 Versuch einer Axiomatisierung

Für Maße der Softwarekomplexität gibt es einen Versuch, die wünschenswerten Eigenschaften von Maßen in ein Axiomensystem zu fassen. Dieses System wurde 1988 von Elaine Weyuker vorgestellt [Wey88]. Seien P, Q und R Programmstücke und $|P|$ die Komplexität von P bezüglich eines angenommenen Maßes. Sind zwei Programme syntaktisch identisch, so schreiben wir $P = Q$, sind sie semantisch identisch, so schreiben wir $P \equiv Q$. Die textuelle Verkettung zweier Programme notieren wir als $P; Q$. Die Komplexität sei eine nichtnegative Zahl, so daß sich Komplexitäten totalordnen lassen (im Gegensatz zu Komplexitäten, die durch Vektoren ausgedrückt werden):

1. $\exists P, Q : |P| \neq |Q|$
   „Nicht konstant“: Ein Maß darf nicht allen Programmen den gleichen Meßwert zuordnen.

2. $\forall c \geq 0 : |\{P \mid |P| = c\}| < \infty$
   „Fein genug“: Ein Maß muß die Programme in genügend viele Komplexitätsklassen unterteilen, so daß keine dieser Klassen unendlich viele Elemente hat.

3. $\exists P, Q : P \neq Q \wedge |P| = |Q|$
   „Nicht zu fein“: Es gibt Programme, die verschieden sind und dennoch die gleiche Komplexität haben.

4. $\exists P, Q : P \equiv Q \wedge |P| \neq |Q|$
   „Implementationsabhängig“: Zwei Programme können unterschiedliche Komplexität haben, auch wenn sie dieselbe Funktionalität besitzen.

5. $\forall P, Q : |P| \leq |P;Q| \wedge |Q| \leq |P;Q|$
„Monotonität“: Das Zufügen von Teilen am Anfang oder Ende eines Programms kann seine Komplexität nicht verringern.

6. a.) $\exists P, Q, R : |P| = |Q| \wedge |P;R| \neq |Q;R|$
b.) $\exists P, Q, R : |P| = |Q| \wedge |R;P| \neq |R;Q|$
„Interaktion“: Programmstücke gleicher Komplexität können, wenn man ein identisches Stück anhängt (Fall a), Gesamtprogramme unterschiedlicher Komplexität ergeben. Das gleiche gilt für das Voranstellen (Fall b). Gleich komplexe Programmstücke interagieren also evtl. verschieden mit ihrem Kontext.

7. $\exists Q, \mathit{Permutation} : P = \mathit{Permutation}(Q) \wedge |P| \neq |Q|$
„Anordnungsabhängigkeit“: Die Komplexität eines Programmstücks kann sich verändern, wenn man die Reihenfolge seiner Anweisungen verändert.

8. $\forall Q, \mathit{Renaming} : P = \mathit{Renaming}(Q) \Rightarrow |P| = |Q|$
„Namensunabhängigkeit“: Die Komplexität eines Programmstücks kann sich nicht ändern, wenn man die Programmobjekte umbenennt.

9. $\exists P, Q : |P| + |Q| < |P;Q|$
„Mehr als Summe der Teile“: Diese quasi umgedrehte Dreiecksungleichung besagt, daß ein Programm komplexer sein kann als die Summe seiner Teile.

Zu diesem System von Axiomen sind eine Reihe von Anmerkungen zu machen:

- Die Bedingungen sind nicht hinreichend dafür, daß ein Maß ein gutes Komplexitätsmaß ist. So zeigt [CS91] an einem Beispiel, daß sich willkürliche, sinnlose Maße definieren lassen, die alle Bedingungen erfüllen.

- Die Bedingungen sind aber auch nicht notwendig. Als Beispiel betrachten wir das Maß „Datenflußkomplexität“, das für ein Programmstück die Gesamtanzahl von Datenflüssen zu Variablennutzungen angibt, die in irgendeinen Block hineinführen. Dieses Maß ist durchaus nützlich zur Beschreibung der Programmkomplexität, erfüllt aber die Monotonitätsbedingung nicht, weil ein Davorhängen von Anweisungen hereinführende Datenflüsse zu blockinternen Referenzen umwandeln kann.

- Zu beachten ist, daß manche Maße nützlich sein können, obwohl sie keine nützlichen *Komplexitäts*maße sind. Beispielsweise verletzt das Lines-of-Code-Maß mehrere der obigen Bedingungen und ist kein gutes Komplexitätsmaß, kann aber dennoch als nützliches Maß betrachtet werden — nur eben für einen anderen Zweck, nämlich die Beschreibung der Größe.

- Manche der Bedingungen sind skeptisch zu betrachten. So ist z.B. die Forderung nach Namensunabhängigkeit aus psychologischer Sicht sehr dubios. Die Feinheitsforderung (2) ist für viele Zwecke vermutlich übertrieben. Wenn man nur bestimmte Aspekte von Softwarekomplexität betrachtet, können auch noch andere Forderungen unnötig werden; z.B. ist für die Schnittstellenkomplexität der Funktionen in einem Modul die Reihenfolge der Funktionen unerheblich, also Forderung 7 überflüssig.

- Dennoch liefert diese Liste von Anforderungen einen guten Maßstab für den Entwurf oder die Beurteilung von Komplexitätsmaßen. Jede Verletzung einer Anforderung muß gut begründet sein, andernfalls kann man das Maß als unbefriedigend ansehen.

1.2.4 Goal, Question, Metric (GQM)

Im vorletzten Abschnitt 1.2.2 haben wir festgestellt, daß die zusammenfassende Anforderung an ein Maß die Zweckmäßigkeit ist. Wie findet man zweckmäßige Maße?

Basili und Weiss haben 1984 eine Methode vorgestellt, mit der man dieses Problem systematisch anpacken kann [BW84]. Sie nennen diese Methode „Goal-question-metric paradigm“ oder kurz GQM. Sie besteht aus folgenden Schritten:

1. **Definiere Ziele**. Ausgangspunkt jedes Meßvorhabens muß die Frage sein, was man eigentlich erreichen will. Messen ist nicht ein Ziel an sich, sondern dient der Verfolgung gewisser Ziele. Beispiele hierfür könnten sein, die Zahl von ins Programm eingefügter Fehler zu minimieren, die Wartbarkeit der Programme zu verbessern oder die Projektdurchlaufzeiten zu verkürzen. Die Ziele können mehr oder weniger allgemein sein, müssen sich aber jedenfalls direkt an den Geschäftszielen, also an Zwecken, orientieren.

2. **Finde Fragen**. Anhand des Messungsziels kann man nun Fragen finden, deren Beantwortung zum Erreichen des Ziels hilfreich wäre. Man konkretisiert das Ziel also in einen Informationswunsch. Zu den obigen Beispielen könnten die Fragen etwa lauten: „Was für Fehler machen wir eigentlich? Und wie viele? Wann machen wir sie? Warum machen wir sie?“ oder „Welche Programme sind schlecht wartbar? Wie kann man diese frühzeitig erkennen?“ oder „Auf welche Arbeiten verwenden wir wieviel Zeit? Welche davon könnte man einsparen? Welche Arbeiten könnte man parallel ausführen?“.

3. **Definiere Maße**. Erst anhand dieser konkreten Informationswünsche sollten nun Maße definiert werden. Das ist in vielen Fällen nach wie vor schwierig, aber zumindest versteht man die Stärken und Schwächen eines Maßes jetzt viel leichter, weil die Fragen einen Maßstab zu ihrer Beurteilung vorgeben. Zu den obigen Beispielen könnten folgende Arten von Messungen sinnvoll sein: Zur Fehleranalyse muß man Fehlerprotokolle führen, die für jeden Fehler eine Fehlerklasse angeben und möglichst den Zeitpunkt, den Arbeitsschritt und die Ursache des Entstehens beschreiben. Zur Wartbarkeitsuntersuchung muß man Hypothesen aufstellen, welche meßbaren Eigenschaften von Programmen (Entwürfe oder Quellcode) die Wartbarkeit beeinflussen und dann dazu Maße aufstellen und ihre Nützlichkeit validieren; wir werden weiter unten einige Beispiele für solche Maße sehen. Solche Maße findet man intuitiv oder mittels induktiver Analyse einiger Programme und deren Wartungshistorie oder mittels schrittweiser Verbesserung vorhandener Maße; die drei Methoden schließen sich natürlich keineswegs aus. Zur Projektbeschleunigung braucht man ein quantitatives Modell des Softwareherstellungsprozesses, das wiederum auf Maßen basiert, die gefunden und validiert werden müssen.

Die Methode von Basili und Weiss beschreibt weiter, wie man das eigentliche Datensammeln auf Basis der nun definierten Maße plant, vorbereitet, durchführt und analysiert, um seine Fragen zu beantworten.

Die wesentliche Nachricht von GQM ist jedoch diese: Bevor Du anfängst irgendetwas zu messen, überleg Dir genau, was Du eigentlich erreichen willst, welche Information Du dazu benötigst und welche Maße Dir diese Information liefern können (oder verwandte Information, die eine genügend gute Annäherung darstellt). Andernfalls sind die Ergebnisse von Messungen keine Information (jedenfalls keine nützliche), sondern nur ein Berg nutzloser Daten. Leider wird dieser elementare Ratschlag in der Praxis oft mißachtet; das Messen und Datensammeln verkommt zum Selbstzweck und richtet aufgrund seiner Kosten und Belästigung mehr Schaden an als es Nutzen bringt.

1.2.5 Auflösung, Genauigkeit und Fehler

Es ist generell beim Messen wichtig, zwischen der Meßauflösung (precision) und der Meßgenauigkeit (accuracy) zu unterscheiden. Die **Auflösung** beschreibt die Auflösungsstärke des Maßes, also die Anzahl von Maßstufen (für nominale und ordinale Skalen), bzw. den geringsten Unterschied zwischen zwei Meßwerten (für Intervall- und Verhältnisskalen). Zum Beispiel hat eine Messung von Zeitaufwand in Minuten eine höhere Auflösung als eine in Stunden (wenn nur ganze Stunden und ganze Minuten angegeben werden).

Die **Genauigkeit** hingegen beschreibt, wie weit der gemessene Wert voraussichtlich oder höchstens vom tatsächlichen Wert abweicht, wenn man eine ganze Reihe von Messungen betrachtet. Beispiel: Eine Berechnung des tatsächlichen Gesamtzeitaufwandes in Stunden für ein Projekt könnte man folgendermaßen vornehmen: Für jeden Arbeitstag der Projektdauer multipliziere man die Anzahl der an diesem Tag am Projekt beteiligten Mitarbeiter mit 8. Eine solche Berechnung wäre nicht genauer als etwa 90 Prozent, weil aufgrund diverser Gründe weniger als 8 Stunden pro Tag pro Person für das Projekt gearbeitet werden: manche Mitarbeiter haben auch noch andere Aufgaben, gelegentlich werden Mitarbeiter krank etc. Abhängig von den Umständen könnte es auch eine Abweichung nach oben geben, weil Mitarbeiter viele Überstunden gemacht haben. Eine gute Beschreibung der Genauigkeit erfolgt durch Angabe einer Fehlerverteilung, also einer Wahrscheinlichkeitsverteilung, die die Erwartung über die Größe des Fehlers beschreibt. So könnte in obigem Beispiel idealerweise die Angabe lauten „die Ungenauigkeit dieses Maßes ist normalverteilt mit Mittelwert -5 Prozent und Standardabweichung 3 Prozent", was dann bedeutet, daß in etwa 2.5 Prozent der Fälle eine Abweichung nach unten von mehr als 11 Prozent erwartet wird (denn Abweichungen von mehr als zwei Standardabweichungen haben bei normalverteilten Daten eine Wahrscheinlichkeit von ungefähr 5 Prozent, je zur Hälfte nach oben und nach unten). Häufiger als ganze Wahrscheinlichkeitsverteilungen wird man Angaben von Konfidenzintervallen benutzen, also zum Beispiel „in mindestens 95 Prozent der Fälle liegt die Abweichung unter 6 Prozent".

Der **Fehler** ist die Ursache von Ungenauigkeit, also die Herkunft der Abweichungen; allerdings werden die Begriffe Fehler und Ungenauigkeit auch oft synonym benutzt. Der Fehler setzt sich zusammen aus dem systematischen Fehler und dem statistischen Fehler. Der **systematische Fehler** ist eine deterministische (oft sogar konstante) Komponente von Ungenauigkeit, wogegen der **statistische Fehler** eine Zufallsgröße ist, die jede einzelne Messung gemäß einer gewissen Verteilung stochastisch beeinflußt. Beispiel: Die Messung einer kurzen Zeit mit einer Handstoppuhr hat als systematischen Fehler den regelmäßigen Falschgang der Uhr, beispielsweise plus 0.1 Promille, also ein Vorgehen von 0.36 Sekunden pro Stunde und als statistischen Fehler die zufälligen Schwankungen in der Reaktionszeit des Bedieners, beispielsweise im Mittel plus oder minus 0.05 Sekunden. Im Beispiel überwiegt also bei kurzen Messungen der statistische Fehler, bei langen der systematische Fehler; der Übergang liegt nach etwas mehr als vier Minuten. Ähnliche Betrachtungen sollte man bei jeder gründlichen Genauigkeitsanalyse anstellen.

Die Genauigkeit wird durch die Auflösung beschränkt, jedoch gaukeln hochauflösende Maße oft eine Genauigkeit vor, die in Wirklichkeit gar nicht gegeben ist. Es ist wichtig, die Genauigkeit eines Maßes oder eines einzelnen Meßvorgangs zu kennen oder wenigstens abschätzen zu können und die Genauigkeit bei der Interpretation von Meßergebnissen, insbesondere beim Vergleich mehrerer, immer im Auge zu behalten, weil sonst leicht ungerechtfertigte Schlüsse zustandekommen. Eine gute Abschätzung der Genauigkeit (erst recht in Form einer Wahrscheinlichkeitsverteilung) ist aber häufig sehr schwierig.

1.2.6 Arten von Softwaremaßen

Maße in der Softwaretechnik lassen sich entlang verschiedener Dimensionen unterscheiden, zum Beispiel nach dem Meßgegenstand, nach der Unterscheidung quantitativ/qualitativ, nach der Unterscheidung automatisierbar/teilautomatisierbar/nichtautomatisierbar oder nach der Unterscheidung direkt/indirekt.

Die Unterscheidung nach dem Meßgegenstand ergibt folgende Arten von Maßen:

- **Externe Produktmaße**. Diese beschreiben Merkmale des Produkts, die von außen sichtbar sind, wenngleich eventuell nur indirekt. Dazu gehören insbesondere Maße für Qualitätsattribute wie die Wartbarkeit, die Zuverlässigkeit, die Benutzbarkeit, die Effizienz etc.

- **Interne Produktmaße**. Basis für die Charakterisierung externer Produktattribute sind interne Attribute, die sich zumindest zum Teil besser messen lassen. Hierzu gehören z.B. die Größe, die Modularität und die Komplexität.

- **Prozeßmaße**. Diese charakterisieren Eigenschaften des Produktionsprozesses, z.B. die typische Produktivität, Kostenaufteilung auf bestimmte Arbeiten, Dauer von Arbeitsschritten etc.

- **Ressourcenmaße**. Diese charakterisieren Eigenschaften der im Prozeß verwendeten Ressourcen, z.B. die Auslastung von Rechnern, die typische Produktivität und Fehlerrate eines bestimmten Ingenieurs etc.

Wir werden diese Kategorien in nachfolgenden Abschnitten noch genauer besprechen.

Quantitative Maße sind solche, deren Meßwerte Zahlen sind und das Rechnen mit ihnen erlauben (also Maße mit Intervallskala oder höheren); alle übrigen Maße sind **qualitativ**. Ein Beispiel für ein qualitatives Maß ist das „Strukturierungsniveau“ eines Programms, das auf einer Ordinalskala mit den Stufen mitGoto, strukturiert, modularisiert, objekt-orientiert gemessen werden könnte. Es ist wichtig, sich darüber klar zu sein, daß aus dieser Ordinalskala auch dann keine Differenz- oder gar Kardinalskala wird, wenn man diesen Stufen jetzt z.B. die Werte 0, 1, 2, 3 zuweist. Rechenoperationen mit diesen Werten wären irreführend und weitgehend sinnlos.

Maße sind **automatisierbar**, wenn ich sie in einem Computerprogramm formulieren kann und alle Eingabedaten zur Verfügung stehen. Sie sind nicht automatisierbar, wenn Entscheidungen von Menschen getroffen werden müssen z.B. Zuweisung einer Fehlerart.

Wir nennen ein Maß **direkt**, wenn es unmittelbar durch einen einzelnen Meßvorgang ermittelt wird, wie z.B. Lines of Code oder Gesamtanzahl Fehler. Die meisten Maße sind jedoch **indirekt**, d.h. sie basieren auf mehreren Meßvorgängen, deren Ergebnisse durch Rechnung verbunden werden, z.B. Fehlerdichte als Gesamtanzahl Fehler geteilt durch Lines of Code.

1.3 Ergebnisse

Es gibt eine Riesenmenge an Berichten darüber, was man mit Messen (angeblich) alles hinbekommen kann oder irgendwann einmal nicht hinbekommen hat. Wir greifen hier nur drei Beispiele heraus.

1.3.1 PSP

Als positives Beispiel für den Einsatz von Maßen eignet sich die Methode des „persönlichen Software-Prozesses“ (PSP), in der zahlreiche recht simple Maße sehr erfolgreich eingesetzt werden, um die Planbarkeit der Softwareerstellung und die Qualität der erzeugten Produkte auf der Ebene eines einzelnen Softwareingenieurs zu verbessern. So wird die Größe von Softwareprodukten in Lines of Code gemessen, was ausreichend ist, wenn man Daten nur auf eine Person und eine Programmiersprache bezieht. Fehler werden nach einem einfachen Schema in 10 Klassen eingeteilt und der Herstellungsprozeß in 6 Phasen, deren Beginn und Ende zeitlich festgehalten wird — und schon ergibt sich eine Vielzahl von Kennzahlen, die beschreiben, wie hoch Fehlerdichten und Fehlerentfernungsquoten in einzelnen Phasen sind, wie effektiv man Inspektionen einsetzt, wie hoch die Produktivität ist etc. Diese Maße sind nicht nur wohldefiniert und aussagekräftig, sie sind auch tatsächlich Grundlage von Verbesserungen. Der PSP wird später (Abschnitt 6) noch ausführlicher besprochen.

1.3.2 o-o

Als ein Beispiel dafür, was alles schiefgehen kann und wie vorsichtig man bei der Interpretation von Meßergebnissen sein muß, vor allem, wenn man versucht, sie von einem Bereich auf einen anderen zu übertragen, soll folgender Ausschnitt aus einer Nachricht dienen, die Steve Walters (swalters@infinet.com) in die Usenet-Newsgruppe comp.software-eng gesendet hat:

> „We just finished a software development project and discovered some curious metrics. This was a project in which we had good domain experience and about six years of metrics, both team productivity and other analogous software of similar scope and functionality. The difference with this project was that we switched from a functional design methodology to o-o.
> First the good news: the overall team productivity (SLOC/person-month) was almost three times our previous rate.
> Now for the bad news: the delivered SLOC was almost three times greater than estimated, based on the metrics from our previous projects.
> We completed the project on time and on budget, but the radical departure from what we have been used to is unnerving.“

1.3.3 GQM

Bei den Arbeiten, die zu der oben bereits besprochenen Goal-question-metric Methode geführt haben, machten Basili und Weiss einige Erfahrungen mit Meßvorhaben, die sie ebenfalls in ihrem Artikel [BW84] beschrieben haben. Diese lassen sich folgendermaßen zusammenfassen:

1. **Verstehe** Deinen Anwendungsbereich (Arbeitsumfeld und Projektgegenstand). Ohne ein solches Verständnis kann man die Gründe für die Phänomene, die man mißt, nicht verstehen und aus den Messungen wenig Nutzen ziehen.

2. **Validiere** die Daten rechtzeitig. Bei allen Messungen treten Widersprüche und Inkonsistenzen auf, insbesondere dann, wenn Daten „von Hand“ erfaßt werden müssen. Solche Widersprüche können oft noch bereinigt werden, wenn man das kurz nach der Messung versucht, weil sich die betreffende Person dann noch gut an den Sachverhalt erinnern kann. Eine Bereinigung mit großen zeitlichen Abstand ist jedoch meist zum Scheitern verurteilt.

3. Minimiere den **Arbeitsaufwand** für die Datenerfassung. Dies spart nicht nur unmittelbar Kosten, es verringert auch die Zahl der Fehler und Auslassungen in den Daten. Am günstigsten ist es, die Datenerfassung mit anderen Prozeßschritten zu integrieren. Eine gute Gelegenheit für das automatische Vermessen von Dokumenten ist beispielsweise das Einchecken der Dokumente in das Konfigurationsverwaltungssystem.

4. Berücksichtige bei der Analyse der Daten die **Umgebung**, in der sie entstanden sind. In der Regel müssen die Umgebungsfaktoren bei der Interpretation der Daten berücksichtigt werden, um nicht zu falschen Schlüssen zu gelangen.

5. **Unterschätze nicht** den Aufwand zur Datensammlung, -validation und -analyse. Wenn die Daten eine hohe Qualität haben sollen, ist ein erheblicher Arbeitsaufwand nötig.

6. **Überzeuge** die Softwareingenieure davon, daß die Daten nicht gegen sie verwendet werden. Andernfalls wird die Datensammlung unterlaufen und ist ziemlich sicher völlig wertlos.

7. Berücksichtige, daß eventuell der **Hawthorne-Effekt** am Werk ist. Dies ist ein psychologischer Effekt, der darin besteht, daß Menschen sich anders verhalten, wenn sie wissen, daß sie beobachtet werden. Im Falle der Softwareherstellung werden sie in der Regel eine bessere Leistung zeigen als normalerweise — wenn der Effekt überhaupt auftritt, was nicht sicher ist. Falls der Hawthorne-Effekt ausgeprägt vorhanden ist, können die anfänglichen Meßergebnisse irreführend sein, denn es ist unwahrscheinlich, daß er jahrelang bestehen bleibt.

1.4 Probleme und Hindernisse

Beim Einsatz von Maßen gibt es eine Reihe von Schwierigkeiten, von denen einige fundamental sind, andere jedoch nur durch schlampiges Vorgehen zustandekommen. Zu letzterer Klasse gehören die Verwendung schlecht definierter Maße und die unzulässige Übertragung oder falsche Interpretation von Ergebnissen. Fundamentale Probleme sind hingegen die Nichtverfügbarkeit geeigneter Maße oder die Nichtverfügbarkeit der Eingabedaten für geeignete Maße. Diese Probleme diskutieren wir jetzt im einzelnen.

1.4.1 Mangel an Wohldefiniertheit

Eine Methode, sich sehr schnell in Schwierigkeiten zu bringen, besteht darin, Maße zu verwenden, ohne sie exakt zu definieren. Das beste Beispiel hierfür ist zweifellos Lines of Code. Die Zahlen, die bei Maßen mit Intervall- und höheren Skalen als Meßwerte entstehen, drängen ein Vergleichen verschiedener Messungen ja förmlich auf. Dabei muß man aber beachten, daß die Vergleichbarkeit im Hinblick auf das, was man aus der Messung an Information gewinnen möchte, eben oftmals nicht gegeben ist. So ist es weitgehend sinnlos, Programmgrößen verschiedener Programmierer zu vergleichen, solange nicht geklärt ist, ob und wie weit die Zählung der Programmzeilen vom Layout des Programms abhängt, wie Leerzeilen, Fastleerzeilen und Kommentarzeilen gezählt werden etc.

Ohne eine genaue Definition geht also alles schief, mit einer Definition muß man eventuell die Interpretation der Meßwerte ändern. So messen Lines of Code mit Ignorieren von Leerzeilen, Fastleerzeilen und Kommentaren so etwas wie die Länge des Programmtextes, beim Mitzählen der Kommentare so etwas wie die Länge Programmtextes plus seiner Erläuterung (wobei Trennzeilen aus lauter Sternchen oder Strichen Schwierigkeiten machen), und beim Mitzählen auch der Leerzeilen und Fastleerzeilen ist das Ergebnis kaum noch sinnvoll interpretierbar.

1.4.2 Mangel an Übertragbarkeit

Ebenso sinnlos wie das Messen ohne genaue Definition des Maßes ist das Vergleichen von Meßwerten, die aus nicht direkt vergleichbaren Verhältnissen stammen. Zum Beispiel nützen mir auch noch so wunderbar wohldefinierte Produktivitätsmessungen aus der Entwicklung eines WWW-Browsers rein gar nichts, wenn meine Aufgabe lautet, die Produktivität bei der Herstellung einer Fabriksteuerung vorherzusagen oder zu beurteilen. Dennoch werden genau solche Übertragungen (nur meist nicht ganz so offensichtlich) ständig gemacht — am liebsten in Kombination mit der Verwendung ungenau definierter Maße.

1.4.3 Fehlinterpretation

Der dritte Fehler aus der Kategorie „Schlamperei" bei Softwaremaßen besteht in einer unzulässigen Interpretation von Meßergebnissen. Das dramatischste Beispiel in dieser Hinsicht ist der Begriff der Produktivität. Drückt man Produktivität zum Beispiel, wie es sehr oft geschieht, in erstellten Programmzeilen pro Zeit aus, so werden die Programmierer zu einer Reihe von ganz fürchterlichen Verhaltensweisen animiert (insofern an ihre Produktivität irgendwelche Anreize oder Bestrafungen gekoppelt sind). Produktivität wird aber korrekterweise am erzeugten Nutzen (oder Wert) pro Zeit bemessen. Ein richtiges Maß wäre also beispielsweise „Vom Endkunden erwünschte Funktionalität pro Zeit". Die Identifikation von Programmzeilen pro Zeit mit Produktivität ist eine Fehlinterpretation des Maßes. Siehe auch die Diskussion in Abschnitt 6.5.2.

1.4.4 Mangel an aussagekräftigen Maßen

Dies ist das erste fundamentale Problem beim Einsatz von Maßen: Für viele Größen, von deren Relevanz man überzeugt ist, gibt es auch nach jahrzehntelanger Suche noch keine Maße, die sich prozedural formulieren ließen. So wird beispielsweise seit langem nach einem Maß gesucht, das die Komplexität eines Programmstücks oder eines Moduls beschreibt. Von allen bisher vorgeschlagenen Maßen für diesen Zweck ist jedoch bekannt, daß sie eine solche Auskunft allenfalls für eine sehr eingeschränkte Klasse von Programmen geben, deren Einhaltung man wiederum meist nicht automatisch prüfen kann. Ein anderes Beispiel ist die Funktionalität, deren quantitative Beschreibung außerordentlich schwierig ist, aber eine Voraussetzung dafür wäre, ein sinnvolles Produktivitätsmaß zu erhalten. Funktionspunkte liefern hierfür eine gute Teillösung, die aber leider nur für betriebswirtschaftliche und ähnliche Anwendungen zu gebrauchen ist. Die Suche nach aussagekräftigen Maßen ist nach wie vor ein Forschungsthema.

1.4.5 Mangel an Eingabemeßwerten

Das zweite fundamentale Problem besteht darin, daß man viele sinnvolle indirekte Maße zwar bequem formulieren kann, jedoch die Zwischendaten fehlen, um sie zu berechnen. Diese Schwierigkeit tritt in drei Formen auf: Zum einen gibt es Maße, für die es einfach einen sehr großen Aufwand verursacht, sie auszuwerten, z.B die Häufigkeit und Art mündlicher Anfragen eines Softwareingenieurs bei Kollegen.

Zum anderen, und dies ist der häufigere Fall, gibt es oftmals Maße, die nützlich wären, deren Eingabedaten aber erst zu spät verfügbar werden. Beispielsweise ist die Zahl von bei Auslieferung noch in einem Programm verbliebener und später in Erscheinung getretener Fehler grundsätzlich erst am Ende der Einsatzdauer der Software, also nach mehreren Jahren, zu ermitteln, wenn das

Interesse an dieser Information bereits gering geworden ist, weil die betroffenen Programmierer inzwischen dazugelernt haben, die betroffenen Prozesse schon verbessert worden sind und die betroffenen Werkzeuge und Methoden teilweise gar nicht mehr eingesetzt werden.

Drittens gibt es Maße, die man zwar konzeptionell klar definieren kann, die sich aber grundsätzlich überhaupt nicht berechnen lassen, z.B. die Anzahl von Fehlern in einem Programm, die bei Auslieferung noch vorhandenen sind.

1.5 Die Klassiker: Lines of Code, Fehleranzahl, Personentage

1.5.1 Beschreibung und Zweck

- Lines of Code: Dient zur Beschreibung der **Größe** von Programmen und wird in der Regel auf Quellcode angewendet. Meßwert ist die Anzahl von Programmzeilen, die eventuell durch das Weglassen von leeren Zeilen und Kommentarzeilen, eventuell auch Deklarationszeilen modifiziert werden kann. Die Programmgröße ist eine wesentliche Grundlage für viele Vorhersagen, weil sie einerseits Ausgabe von Schätzungen ist und andererseits als Eingabe anderer Schätzungen benötigt wird (Zeit, Personal, Kosten, Speicherplatz etc). Sie dient auch zur Berechnung zahlreicher indirekter Maße wie Fehlerdichten, Strukturiertheitsgrad etc.

- Fehleranzahl: Dient zur Beschreibung der **Korrektheit** von Programmen und wird entweder auf den Quellcode oder die Programmausführung bezogen. Meßwert ist entweder die Zahl dynamisch auftretender Programmversagen oder die Zahl statisch im Programmtext vorhandener Defekte. Ebenso wie die Größe ist die Fehleranzahl die Basisgröße für sehr viele Maße, die die Qualität von Software beschreiben (Fehlerdichte, Zuverlässigkeit, Debuggingproduktivität etc.).

- Personentage: Dient zur Beschreibung des **Aufwand**s an Arbeitszeit für eine Softwareaufgabe. Meßwert ist z.B. die Anzahl von Projektmitarbeitern für jeden Projekttag, summiert über alle Projekttage; oft werden für die Erfassung Wochen oder Monate anstatt Tage zugrundegelegt. Ebenfalls eine wichtige Basisgröße für zahlreiche indirekte Maße, die sich mit Kosten, Zeitbedarf oder Produktivität befassen.

Eine exakte Definition wird hier für keine dieser Größen angegeben, weil es Dutzende unterschiedliche Arten gibt, diese Größen sinnvoll zu definieren.

Es gibt noch ein viertes klassisches Basismaß, nämlich die Echtzeit (gemessen meist in Kalendertagen). Dies ist jedoch so elementar, daß wir im folgenden nicht näher darauf eingehen.

1.5.2 Probleme und Grenzen

Die Definitionen aller drei Maße sind im praktischen Einsatz meist ungenau. Das liegt daran, daß die Maße so verführerisch einfach und intuitiv einleuchtend aussehen. Bei Lines of Code hat sich die Notwendigkeit einer genauen Definition inzwischen herumgesprochen, aber schon bei der Fehleranzahl hapert es erheblich: Was genau ist eigentlich ein Fehler (im Gegensatz z.B. zu einer „freiwilligen“ Verbesserung)? Von einer Einteilung der Fehler in disjunkte Klassen, die die Grundlage für viele weitere nützliche Analysen des Fehlerverhaltens ist, ganz zu schweigen. Bei den Personenmonaten stellt sich die Frage, ob die Basis ihrer Berechnung ein Pauschalverfahren ist, wie oben angegeben, oder eine Zeiterfassung im Einzelfall (d.h. für jede Person) auf Basis von Wochen oder Tagen oder Stunden oder gar Minuten.

Selbst mit genauen Definitionen ist es für alle drei Maße schwer, übertragbare Messungen zu machen, weil die Meßwerte von vielen Umgebungsfaktoren abhängen, die nur selten gleich (oder auch nur überhaupt alle bekannt) sind.

- Lines of Code: Größen sind nur für den gleichen Programmierer im gleichen Problembereich und mit der gleichen Programmiersprache direkt vergleichbar. In allen anderen Fällen ist die Größe nur ein grobes Maß für die „Menge" an Software, unter der man intuitiv zumeist die Menge an realisierter Funktionalität versteht. Beim Vergleich sehr verschiedener Anwendungsgebiete oder sehr verschiedener Programmiersprachen kann das Maß sogar äußerst irreführend werden. Siehe auch die Diskussion in Abschnitt 6.5.2.
- Fehleranzahl: Die Reproduzierbarkeit (und damit Objektivität) ist für das Maß Fehleranzahl nicht gewährleistet, weil zur Abgrenzung zwischen Fehler/Nichtfehler oder zwischen ein Fehler/mehrere Fehler oft komplexe menschliche Entscheidungen nötig sind, die sich nicht formalisieren lassen. Es ist sehr schwierig, Fehlerklassifizierungen aufzustellen, die eine zumindest meistens eindeutige Zuordnung erlauben und dennoch viel nützliche Information liefern.
- Personentage: Die Übertragung zwischen unterschiedlich strukturierten Softwareorganisationen ist schwierig, spätenstens sobald die Tätigkeit von unterstützendem Personal wie Sekretärinnen oder Systembetreuern mit eingerechnet werden soll. Selbst die Vergleichbarkeit über Projekte derselben Organisation hinweg ist gering, sobald einzelne Projektbeteiligte an mehreren verwandten Projekten zugleich arbeiten oder sobald Wiederverwendung der erzeugten Softwarekomponenten angestrebt wird oder bei ihrer Erstellung frühere Software wiederverwendet wird. Zwar kann man den Aufwand für die gesamte Organisation recht gut berechnen, eine sinnvolle Zuweisung an die einzelnen Projekte ist aber sehr schwierig.

1.5.3 Wichtigkeit

Die beschriebenen Maße sind die Basis für fast alle höheren Maße und sind oft nicht durch bessere Maße zu ersetzen: sie sind faktisch unverzichtbar.

- Lines of Code: Da der Ausstoß an Programmzeilen pro Tag im Mittel für einen Programmierer recht konstant ist, sind LOC eine wichtige Grundlage für viele Maße, die mit Arbeitsgeschwindigkeit oder Arbeitsmenge zusammenhängen. Es ist beispielsweise zwar ein Fehler, LOC/Tag mit der Produktivität zu verwechseln, jedoch liefert das Maß sehr wohl Hinweise darauf, ob z.B. ein Programmierer plötzlich zu viel von der Arbeit abgehalten wird (Wert sinkt) oder vielleicht zwischendurch schlampig arbeitet (Wert steigt).
- Fehleranzahl: Fehler nehmen eine sehr zentrale Rolle im Softwareprozeß ein, weil sie zum einen die Produktivität beherrschen und zum anderen zentrale Qualitätsmerkmale von Software direkt beeinflussen. Große Teile der Zeit im Softwareprozeß werden auf die Fehlervermeidung, -findung und -behebung verwendet und die Zahl von Fehlern, die schließlich im Produkt ausgeliefert wird, wirkt sich stark auf die Zuverlässigkeit, Benutzbarkeit und Nützlichkeit der Software aus. Deshalb ist die Fehleranzahl Grundlage für sehr wichtige Maße wie Fehlerdichten pro Modul (um kritische Module zu entdecken), Fehlerdichten pro Arbeitsphase (um den Prozeß zu verbessern), Fehlerartenhäufigkeiten (ebenfalls um den Prozeß zu verbessern).

- Personentage: Da sich die meisten Anstrengungen der Softwaretechnik letztlich darum ranken, die Kosten zu senken, die fast vollständig von den Personalkosten beherrscht werden, sind Personentage ein grundlegendes Maß zur Bestimmung fast aller Arten von Kostenfaktoren im Softwareprozeß. Es gibt noch einen zweiten zentralen Aspekt bei der Softwareproduktion: die Zeit bis zum Markteintritt. Diese ist heute oft sogar noch wichtiger als die Kosten, weil der Markt fast völlig vom ersten Anbieter einer neuen Funktionalität beherrscht wird (siehe z.B. Netscape). Da sich Softwareproduktion nicht beliebig auf Arbeitskräfte aufteilen läßt, sind auch für dieses Maß und damit zusammenhängende Hilfsmaße die Personentage eine wichtige Basis.

1.5.4 Fazit

Die klassischen Maße verlangen zu ihrer sinnvollen Anwendung nach einer genauen Definition und disziplinierten Meßverfahren. Dann jedoch kann mit ihnen trotz ihrer Schwächen einiges geleistet werden:

- LOC: Ist bei konstanten Umgebungsbedingungen ein konsistenter Normierungsfaktor für viele Prozeß-, Produkt- und Ressourcenmaße. Nützlich, weil automatisierbar und im voraus schätzbar.

- Fehleranzahl: Hat begrenzte Genauigkeit, aber extrem hohen Nutzwert zur Steuerung des Prozesses und seiner Verbesserung, wenn die Daten in Form einer Fehlerklassifizierung vorliegen. Die Fehleranzahl ist vielleicht die wichtigste direkte Messung überhaupt, weil Fehler so stark sowohl auf Qualität als auch Produktivität einwirken.

- Personentage: Haben begrenzte Genauigkeit, sind aber unersetzlich zur Beschreibung der Kosten. Sind auf Umweg über andere Maße schätzbar, und bilden deshalb eine wichtige Grundlage für die Planung.

1.5.5 Beispiel: Definition von LOC

Hier zur Veranschaulichung eine halbwegs genaue realistische Definition von Lines of Code:

- Richtlinien:
  - Jede Anweisung wird auf mindestens eine Zeile für sich geschrieben.
  - Anweisungen, die dafür zu lang sind, werden unter Beachtung der korrekten Einrückung und einer sinnvollen Zerlegung in möglichst wenige Stücke zerlegt.
  - Jede Deklaration einer Variablen erfolgt auf einer getrennten Zeile. Das gilt auch für Elemente von Verbunden und Parameter von Prozeduren.
  - Jede Deklaration eines Verbundtyps mit n Elementen oder einer Prozedur mit n Parametern umfaßt also $n + 1$ Zeilen.
  - Eine For-, While-, Repeat- etc. Anweisung ist um zwei Zeilen länger als ihr Rumpf (Kopfzeile und END-Zeile).
  - Eine If-Then-Else-Anweisung ist um vier Zeilen länger als ihr Rumpf, bei fehlendem Else-Teil um drei Zeilen.

- Zählweise: Bei Einhaltung dieser Richtlinien werden Lines of Code dann folgendermaßen gezählt: Jede physische Zeile jeder zum Programm gehörigen Datei zählt, ausgenommen

- Leere Zeilen.
- Zeilen, die ausschließlich Kommentar oder...
- ...Blocktrenner enthalten (BEGIN, END).

Je nach Programmiersprache müßte man noch Richtlinien zum Layout von Case-Anweisungen und von Elseif-Ketten geben, sowie für diverse Spezialkonstruktionen, die die Sprache zuläßt (beispielsweise Import- und Exportlisten, Schabloneninstanziierungen etc.).

1.6 Vorhersage und Validation

Wie schon erwähnt ist das Ziel des Messens meist eine Vorhersage mit dem Zweck, die Qualität oder die Kosten eines Produkts oder Prozesses zu verbessern. Wie kommt man zu einer solchen Vorhersage?

Vorhersagen werden hergestellt durch ein **Vorhersagesystem**. Ein solches besteht aus einem mathematischen Modell und einer Menge von Vorhersageprozeduren zur Bestimmung unbekannter Parameter des Modells und zur Interpretation der Resultate. Ein Modell ist eine abstrakte Repräsentation eines Objekts (hier: eines Produkts oder Prozesses).

1.6.1 Beispiel: Vorhersagesystem

Hier ein einfaches Beispiel für ein Vorhersagesystem: Gesucht sei ein Vorhersagesystem zur Vorhersage des Personenzeitaufwands zur Erstellung einer Softwarekomponente k bekannter Größe. Wir modellieren den Personenzeitaufwand $T(k)$ durch die Beziehung $T = b + G(k)/p$, also einen konstanten Basiszeitaufwand b plus die Komponentengröße $G(k)$ geteilt durch eine Programmiergeschwindigkeit p (gemessen in Größe pro Zeit).

Dieses Modell macht also erstens die Annahme, daß es eine Menge von Verwaltungs- und sonstigen Arbeiten gibt, deren Dauer pro Komponente von der Größe und der Art dieser Komponente unabhängig ist. Das könnten zum Beispiel die Zeit zum Registrieren der Komponente in der Konfigurationsverwaltung, zum Planen des Herstellungsprozesses und zum Sammeln von Daten über das fertige Produkt sein.

Zweitens macht das Modell die Annahme, daß die eigentliche Herstellung mit einer Geschwindigkeit (Größe pro Zeit) abläuft, die sowohl von der Größe der Komponente als auch von der Art der Komponente unabhängig ist — zumindest innerhalb des separat festzulegenden Gültigkeitsbereichs des Modells.

Um aus diesem Modell ein Vorhersagesystem zu gewinnen, brauchen wir Prozeduren zur Bestimmung der Parameter $b, G(k)$ und p. Die Bestimmung von $G(k)$ geschieht, so könnten wir zum Beispiel festlegen, mittels eines Lines-of-Code-Maßes. Die Bestimmung von b und p könnte beispielsweise durch lineare Regression anhand historischer Daten erfolgen. Das geht so: Wir sammeln für eine Reihe von Softwarekomponenten die tatsächlichen Daten für $T(k)$, sagen wir in Minuten, und $G(k)$, beispielsweise

$G(k)$	$T(k)$	
78	33	
565	207	
330	191	
187	93	
w 1160	524	Summe
290	131	Durchschnitt

nun berechnen wir die Regressionsgerade nach den Regressionsformeln

$$\beta_1 = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2} \qquad \beta_0 = \bar{y} - \beta_1 \bar{x}$$

wobei $\bar{x}$ und $\bar{y}$ die jeweiligen Mittelwerte der x_i und der y_i sind. In unserem Fall setzen wir als x_i die Werte $G(k)$ ein und als y_i die $T(k)$, der Ergebnisparameter β_1 entspricht bei uns $1/p$ und β_0 entspricht b. Wir erhalten

$$p = \frac{199950 - 4 \cdot 290 \cdot 131}{469178 - 4 \cdot 290} = \frac{47990}{132778} = 0.361 \qquad b = 131 - 0.361 \cdot 290 = 26.3$$

Damit können wir Vorhersagen treffen in der Form $T(k) = 26.3 + 0.361 \cdot G(k)$.

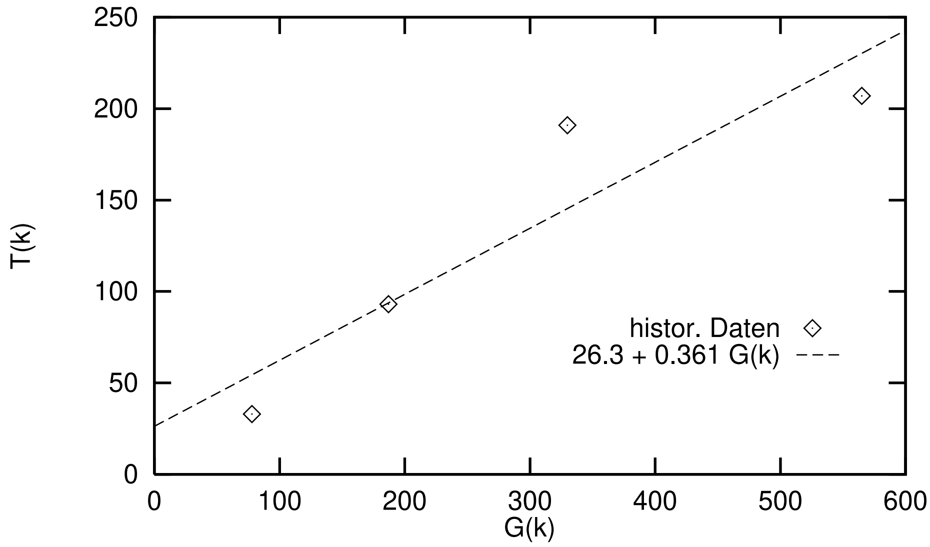

Abbildung 1.2: Datenpunkte und Regressionsgerade für Vorhersagemodell

Die vier oben angegebenen Datenpunkte und die Regressionsgerade sind zur Veranschaulichung in Abbildung 1.2 dargestellt.

Bei diesem ganzen Prozeß ist übrigens ein fundamentaler Fehler passiert, ist er Ihnen aufgefallen?
Ein Programmierer, der fast drei Zeilen pro Minute erzeugen kann ist kaum vorstellbar. In obiger Rechnung wurden von den wirklichen Daten die Zeilenanzahlen mit den Minutenanzahlen genau vertauscht. Dieses Problem kann durch Mitdenken und Plausibilitätsprüfungen mittels Überschlagsrechnung leicht entdeckt werden, führt andernfalls aber zu verwirrenden Auswirkungen. Dies verdeutlicht, daß der gesunde Menschenverstand immer der Hauptmaßstab bleiben sollte.

1.6.2 Validation

Bevor man nun ein Vorhersagesystem benutzt, sollte man sicherstellen, daß es auch wirklich für den angestrebten Zweck brauchbar ist. Diesen Vorgang bezeichnet man als Validation. Die

Validation eines Vorhersagesystems bedeutet letztlich nichts anderes, als seine Genauigkeit und Zuverlässigkeit zu bestimmen und zu entscheiden, ob sie ausreichend ist. Eine solche Anforderung könnte sein, daß die Vorhersagen in höchstens 10 Prozent der Anwendungsfälle um mehr als 20 Prozent vom tatsächlichen Wert abweichen dürfen.

Dabei ist zu beachten, daß die Validation im Hinblick auf den Anwendungszweck vorgenommen wird. Es wird also gegebenenfalls nicht nur die Güte des mathematischen Modells samt seiner Parameter betrachtet, sondern auch die Interpretation der Ergebnisse aus dem Modell. Beispiel: Bei der Validation eines Vorhersagesystems für die Zuverlässigkeit beschreibe das mathematische Modell die Häufigkeit des Versagens (also des Auftretens von Fehlern). Selbst wenn diese Häufigkeit gut vorhergesagt wird, wäre noch zu klären, ob der zugrundegelegte Fehlerbegriff mit dem Zuverlässigkeitsempfinden harmoniert, oder ob vielleicht die Benutzer manche Fehler kaum als Beeinträchtigung der Zuverlässigkeit empfinden aber andere Fehler als erhebliche Beeinträchtigung.

Oft jedoch ist der Kern der Validation eines Vorhersagesystems direkt an den verwendeten Maßen aufgehängt: für ein Maß ist stets zu prüfen, ob es die angeblich von ihm charakterisierte Eigenschaft auch der Intutition entsprechend widerspiegelt. So ist beispielsweise das Maß Lines of Code nur dann ein „valides“ (gültiges) Maß für die Programmlänge entsprechend unserer Intuition, wenn wir das Zählen von leeren Zeilen ausschließen. Denn nach unserem Empfinden wird ein Programm nicht länger, wenn wir beispielsweise 5000 Leerzeilen anhängen. Ein Lines-of-Code-Maß, das einfach die Anzahl von Zeilenwechseln in einer Datei zugrundelegt, wäre deshalb eine unsichere Grundlage für Vorhersagesysteme und würde eine Validation als Maß für Programmlänge nicht bestehen.

1.6.3 Beispiel: Validation

Verfolgen wir das obige Beispiel der Vorhersage von $T(k)$ weiter: Wir haben ein Lines-of-Code-Maß verwendet, das die Länge gut charakterisiert, von dort droht also keine Gefahr. Eine Interpretation der Ergebnisse findet in unserem Beispiel ebenfalls nicht statt, kann also auch die Validation nicht verderben. Bliebe noch sicherzustellen, daß die mit den berechneten Parametern aus dem mathematischen Modell entstehenden Vorhersagewerte genau genug sind.

Wir müssen also zunächst eine gewünschte Genauigkeit festlegen und dann prüfen, ob sie eingehalten wird. Welche Genauigkeit zu fordern ist, hängt von der Anwendung des Vorhersagesystems ab. Werden aufgrund der Vorhersagen sehr kritische Entscheidungen getroffen, wird eine recht hohe Zuverlässigkeit und eventuell auch eine hohe Genauigkeit benötigt. In vielen Fällen ist man aber auch bereit, sich mit weniger genauen und zuverlässigen Werten zufriedenzugeben, weil das immer noch besser ist, als überhaupt keine systematisch erzeugten Vorhersagen zur Verfügung zu haben. Nehmen wir an, in unserem Fall wünschen wir eine Genauigkeit von plus/minus 20 Prozent mit einer Zuverlässigkeit von 90 Prozent.

Dann benötigen wir eine ausreichend große Menge von Datenpunkten, die repräsentativ für die geplante spätere Anwendung des Vorhersagesystems sein muß, um seine Genauigkeit zu messen. Für das Beispiel ist dies in Abbildung 1.3 dargestellt: Wir verwenden 50 Paare von Größe G und Zeit T, die wir bereits gemessen haben. Aus der Ungenauigkeitsschranke von 20 Prozent ergeben sich die beiden im Bild mit ‘min’ und ‘max’ bezeichneten Geraden, die den erlaubten Vorhersagebereich abgrenzen. Für eine Zuverlässigkeit von 90 Prozent im Rahmen dieser Schranke dürften von 50 Datenpunkten also nur 5 außerhalb dieses Bereichs liegen. Tatsächlich finden wir aber 14 Punkte außerhalb vor. Das Vorhersagesystem genügt in diesem Beispiel unseren Ansprüchen also eindeutig nicht.

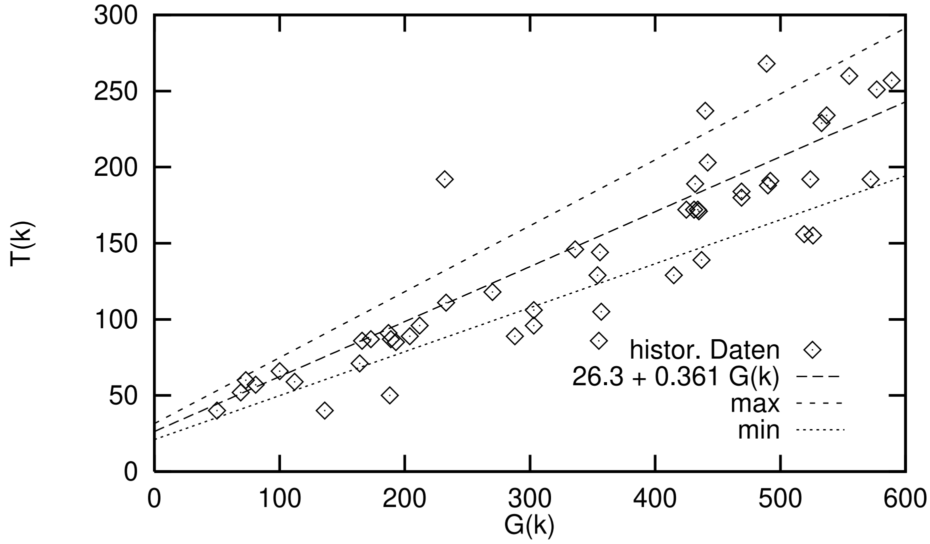

Abbildung 1.3: Datenpunkte und erlaubter Ungenauigkeitsbereich zur Validierung des Vorhersagesystems

In einer solchen Validierung gewinnt man oftmals neue Einsichten in das zu vermessende System, die anschließend die Definition besserer Vorhersagesysteme erlauben. Im Beispiel etwa erkennen wir, daß das Vorhersagesystem zwar grundsätzlich brauchbare, aber systematisch etwas zu hoch liegende Vorhersagen macht. Die Parameter der Regressionsgerade sollten also mit mehr Daten noch einmal neu berechnet werden, dann wird sich die Qualität der Vorhersagen verbessern.

1.6.4 Häufige Fehler bei Validation

Drei Fehler werden bei der Validation von Maßen und Vorhersagesystemen häufig gemacht: Erstens wird häufig nicht die Übereinstimmung der Meßwerte mit dem tatsächlich zu beurteilenden Sachverhalt überprüft, sondern stattdessen nachgewiesen, daß eine **Korrelation** mit anderen, bereits bekannten Maßen vorliegt. Eine solche Aussage ist jedoch weitgehend nutzlos: Ist die Korrelation sehr hoch, ist das neue Maß eigentlich überflüssig, denn man kann fast ebensogut das alte verwenden. Ist die Korrelation nicht sehr hoch, so stellt sich die Frage, ob das neue Maß eine bessere oder eine schlechtere Übereinstimmung mit der zu messenden Eigenschaft hat als das alte. Diese Frage kann durch eine Korrelationsmessung jedoch prinzipiell nicht beantwortet werden. Ein solches Vorgehen ist nur dann nützlich, wenn mehrere Bedingungen erfüllt sind:

1. Das neue Maß korreliert sehr gut mit einem alten.
2. Das alte Maß ist anerkanntermaßen nützlich.
3. Das neue Maß ist einfacher zu berechnen als das alte oder steht zu einer früheren Zeit zur Verfügung.

Zweitens wird oftmals keine klare und einleuchtende **Hypothese** an den Anfang einer Validation gestellt, sondern es wird gegen irgendetwas validiert, worüber gerade Daten zur Hand sind. Dazu müssen dann implizit Annahmen gemacht werden, die eventuell jeder Grundlage entbehren. Beispielsweise wurde ein Maß für die Modularität aufgrund seiner Vorhersagekraft für die Entwicklungskosten validiert, obwohl keine Aussagen über einen Zusammenhang zwischen Modularität und Entwicklungskosten vorliegen.

Aus ähnlichen Erwägungen wird auch manchmal gleich bei der Definition eines Maßes eine Konstruktion gemacht, die gar nichts mit dem Ziel des Maßes zu tun hat; ich zitiere als Beispiel Chidamber und Kemerer mit einer Aussage über ihr RFC-Maß [CK94]:

> It should be noted that membership to the response set is defined only up to the first level of nesting of method calls due to the practical considerations involved in collection of the metric.

Drittens wird manchmal ein Vorhersagesystem aufgrund einer gewissen Menge von Datenpunkten entwickelt und optimiert und dann aufgrund guter Vorhersagen für **dieselben Datenpunkte** als validiert betrachtet. Das ist im Prinzip Unsinn, weil man unter dieser Randbedingung stets auf triviale Weise ein scheinbar perfektes Vorhersagesystem konstruieren kann. Das Vorgehen ist allenfalls zulässig, wenn sichergestellt werden kann, daß die einzelnen Werte der Datenpunkte die grundsätzliche Arbeitsweise des Vorhersagesystems nicht beeinflussen.

1.6.5 Wahl von Vorhersagesystemen

Was für Vorhersagesysteme sollte man angesichts dieser Schwierigkeiten denn nun anstreben? Es gibt mehrere Klassen von Vorhersagesystemen, die sowohl nützlich als auch validierbar sind:

- Messung interner Produktattribute aus frühen Phasen des Software-Lebenszyklus zur Vorhersage von internen Attributen für Produkte aus späteren Phasen. Beispiel: Verwendung von Größe und Modularität einer Spezifikation zur Vorhersage von Größe und Strukturiertheit einer Implementation.
- Messung früher Prozeßattribute zur Vorhersage von späteren Prozeßattributen. Beispiel: Anzahl von Fehlern, die in der Grobentwurfsinspektion gefunden wurden zur Vorhersage der Häufigkeit von Entwurfskorrekturen während der Implementation.
- Messung interner Produktattribute zur Vorhersage von Prozeßattributen. Beispiel: Strukturiertheit des Codes zur Vorhersage des Zeitbedarfs für bestimmte Wartungsarbeiten.
- Messung von internen Produktattributen und Prozeßattributen zur Vorhersage externer Produktattribute. Beispiel: Zahl von bei Codedurchsichten gefundener Fehler zur Vorhersage der Zuverlässigkeit.

Die Vorhersage externer Attribute aus internen Attributen wird immer problematisch bleiben, weil äußerlich sich gleich verhaltende Software intern dramatisch unterschiedlich aufgebaut sein kann. Die Vorhersage kann aber dennoch in einem nützlichen Rahmen gelingen, wenn Einschränkungen an das Universum der internen Attribute ohnehin festliegen. Ferner kann die zusätzliche Nutzung von Prozeßinformation die Vorhersage verbessern.

1.7 Beispiel: Objektorientierte Entwurfsmaße

Chidamber und Kemerer (CK) haben 1991 einen Satz von Maßen zur Beurteilung objektorientierter Entwürfe vorgeschlagen. In [CK94] werden diese Maße ausführlich beschrieben und anhand empirischer Daten diskutiert. Die Daten entstammen zwei großen industriellen Klassensammlungen (eine in C++ und eine in Smalltalk) von zwei verschiedenen Firmen und umfassen hunderte von Klassen.

Basili, Briand und Melo (BBM) haben 1995 die gleichen Maße im Hinblick auf ihre Eignung zur Vorhersage der Fehleranfälligkeit von Klassenimplementationen validiert. Datenquelle hier waren 8 Implementationen eines Softwaresystems für eine hypothetische Videothek, die von Studenten in einem Praktikum über 4 Monate entwickelt wurden. Die Systeme wurden dabei mit OMT modelliert und mit C++ und verschiedenen Bibliotheken implementiert.

Die Analyse zur Validation bestimmte die Signifikanz α, mit der jedes Maß zur Vorhersage verwendet werden kann, ob mindestens ein Fehler in der Testphase gefunden werden wird oder nicht (binäre Antwort). Diese Vorhersage wird mit einem logistischen Regressionsmodell gemacht, das die Form

$$\log\left(\frac{p}{1-p}\right) = C_0 + C_1 M$$

hat. Dabei ist M der jeweilige Meßwert und p die Wahrscheinlichkeit, daß ein Fehler auftritt. C_0, C_1 sind die freien Parameter des Modells, die mittels der logistischen Regression aus den Daten berechnet werden. Die Signifikanz gibt die Wahrscheinlichkeit dafür an, daß ein aus den Daten berechneter Zusammenhang nur zufälliger Natur ist, also das wahre C_1 gleich Null ist; eine „hohe Signifikanz" liegt vor, wenn α klein ist, dann ist ein Zufall unwahrscheinlich. Ferner kann berechnet werden, wie groß der Anteil der Varianz in der Variablen „Fehler tritt auf" ist, der sich mit Hilfe des Modells erklären läßt. Dieser Wert wird als R^2 bezeichnet; ist er 0, so hat das Modell überhaupt keine Erklärungskraft, ist er 1, so beschreibt das Modell einen funktionalen Zusammenhang, es ist also perfekt.

BBM unterschieden drei Teilsysteme ihrer Anwendung: GUI (graphische Benutzerschnittstelle), TUI (textuelle Benutzerschnittstelle) und DB (Datenbankoperationen). Ferner wurde bei Klassen, die auf Basis einer existierenden Klasse der Bibliothek entwickelt wurden, zwischen unverändert, leicht verändert (bis 25% Änderung) und stark verändert unterschieden.

Alle folgenden Maße werden auf jeweils eine Klasse angewendet.

1.7.1 Gewichtete Methoden pro Klasse (WMC)

„Weighted Methods per Class": WMC $= \sum_{Methods\, i} c_i$ wobei c_i die Komplexität der Methode i ist, die aufgrund irgendeines beliebigen Komplexitätsmaßes berechnet werden kann. Sowohl bei CK als auch bei BBM wird hierfür immer die Konstante 1 eingesetzt.

Bedeutungen und Beobachtungen laut CK:

1. Vorhersage des Aufwands zur Entwicklung und Wartung der Klasse.
2. Hohe Zahl an Methoden bedeutet großen Einfluß auf Unterklassen (wegen Vererbung).
3. Klassen mit vielen Methoden sind meist anwendungsspezifischer und deshalb schlechter wiederverwendbar.

Beobachtungen: CK fanden, daß die Anzahl der Klassen mit steigendem WMC in beiden Firmen exponentiell schnell abnimmt, trotz unterschiedlicher Anwendungsgebiete, Projektmannschaften und Programmiersprachen.

BBM fanden eine Signifikanz von $\alpha = 0.06$, was auf eine mäßige Brauchbarkeit von WMC zur Vorhersage von Fehlern hindeutet, 0.7% der Varianz des Fehlerauftretens können mittels WMC erklärt werden; hoher WMC bedeutet höhere Wahrscheinlichkeit eines Fehlers. Für bestimmte Klassenarten waren die Ergebnisse besser: Bei neuen und stark veränderten Klassen ist die Signifikanz 0.0003 und 2.4% der Varianz werden erklärt. Bei GUI- und TUI-Klassen ist die Signifikanz 0.001 und 4.7% der Varianz werden erklärt.

1.7.2 Tiefe im Vererbungsbaum (DIT)

„Depth of Inheritance Tree“: DIT = Länge des längsten Pfades von der Klasse zur Wurzel ihres Vererbungsbaums. Bei Einfachvererbung gibt es nur einen Pfad, bei Klassen ohne Oberklasse ist das Ergebnis 0.

Bedeutungen laut CK:

1. Je tiefer eine Klasse in der Hierarchie, desto schwieriger wird es, ihr Verhalten vorherzusagen, weil sie mehr und mehr Code von Oberklassen erbt.
2. Tiefere Bäume haben höhere Entwurfskomplexität, weil mehr Methoden und Klassen involviert sind.
3. Je tiefer eine Klasse in der Hierarchie ist, desto höher ist ihr Potential für die Wiederverwendung von Methoden.

Beobachtungen: CK fanden Mediane für DIT von 1 bzw. 3. Sie behaupten, daß die DIT-Verteilung es erlaubt zu entscheiden, ob ein Entwurf „kopflastig“ sei, also zu viele Klassen zu weit oben in der Vererbungshierarchie hat, oder „fußlastig“, also zu viele Klassen zu tief in der Hierarchie, wobei Kopflastigkeit auf zu schlechte Ausnutzung der Wiederverwendungsmöglichkeiten hindeute. CK vermuten, daß diese Möglichkeiten zugunsten leichterer Verständlichkeit des Entwurfs geopfert werden. Sie stufen beide betrachteten Klassensammlungen als kopflastig ein.

BBM fanden mit einer Signifikanz von $\alpha = 0.0000$, daß hohe DIT Fehler vorhersagt, DIT erklärt 6.5% der Varianz. Dieses Ergebnis bedeutet, daß zumindest im Hinblick auf Fehler, kopflastige Entwürfe klar besser sind — jedenfalls im Experiment von BBM.

1.7.3 Zahl von Unterklassen (NOC)

„Number of Children“: NOC = Zahl direkter Unterklassen der Klasse.

Bedeutungen laut CK:

1. Je größer die Zahl der Unterklassen, umso höher die Wiederverwendung, da Vererbung eine Form der Wiederverwendung ist.
2. Je größer die Zahl der Unterklassen, desto höher die Wahrscheinlichkeit von falscher Abstraktion in der Oberklasse und die Wahrscheinlichkeit, daß die Vererbung mißbraucht wurde.
3. Je größer die Zahl der Unterklassen, desto mehr Einfluß hat die Klasse auf den Gesamtentwurf. Sie sollte deshalb z.B. sorgfältiger getestet werden.

Beobachtungen: CK fanden wiederum sehr ähnliche Verteilungen in beiden Firmen. Etwa 70% der Klassen hat gar keine Kinder. NOC kann benutzt werden, um zu prüfen, inwieweit ggf. die jeweilige vorgegebene Entwurfsphilosophie (entweder möglichst tiefe Vererbungshierarchien für starke Wiederverwendung oder möglichst flache für bessere Effizienz und leichtere Verständlichkeit) eingehalten wird.

BBM fanden mit Signifikanz von $\alpha = 0.0000$, daß Klassen mit hoher NOC eine geringere(!) Fehlerwahrscheinlichkeit aufweisen, wobei 14% der Varianz erklärt werden. Sie erklären das damit, daß die meisten Klassen höchstens eine Unterklasse haben und eine hohe Zahl von Unterklassen bei unverändert wiederverwendeten Klassen auftritt. Allerdings ist die Signifikanz für nur neue oder geändert wiederverwendete Klassen immer noch 0.0011, so daß diese Erklärung nicht recht überzeugt. Vielleicht ist der relevante Effekt der, daß eine Klasse, von der weitere Klassen abgeleitet werden, sorgfältiger entworfen und implementiert und besser verstanden wird. Inwieweit dieser Effekt auf andere Projekte und Teams zu übertragen ist, läßt sich nicht sagen.

1.7.4 Kopplung zwischen Objektklassen (CBO)

„Coupling Between Object Classes“: CBO = Anzahl von anderen Klassen, aus denen Methoden dieser Klasse Methoden oder Instanzvariablen verwenden.

Bedeutungen laut CK:

1. Hohe Kopplung vermindert die Möglichkeit zur Wiederverwendung der Klasse in einer anderen Anwendung.
2. Hohe Kopplung vermindert die Modularität und Kapselung von Klassen und erhöht ihre Empfindlichkeit gegen Änderungen an anderen Stellen des Entwurfs.
3. Das Kopplungsmaß erlaubt vorherzusagen, wie schwierig das Testen der Klasse wird bzw. wie sorgfältig dabei vorgegangen werden sollte.

Beobachtungen: CK fanden, daß der Median von CBO bei C++ gleich Null war, bei Smalltalk hingegen gleich 9 und zwar wohl deshalb, weil in Smalltalk auch elementare Datentypen Klassen sind, ebenso die Kontrollflußkonstrukte wie if, while und repeat. Die mittlere CBO wächst mit der Größe der Klassenmenge.

BBM fanden eine Signifikanz von $\alpha = 0.0000$ für hohe Fehlerwahrscheinlichkeit bei hoher CBO, welche im logistischen Regressionsmodell 6.8% der Varianz erklären kann.

1.7.5 Antwortmenge einer Klasse (RFC)

„Response for a Class“: RFC = Gesamtanzahl verschiedener Methoden, die von irgendeiner Methode der Klasse direkt aufgerufen werden.

Bedeutungen laut CK:

1. Große Antwortmengen machen das Testen und Debuggen schwieriger, weil mehr Code verstanden werden muß.
2. Eine große Antwortmenge bedeutet eine hohe Komplexität der Klasse.

Beobachtungen: CK fanden, daß die Anzahl von Klassen wieder exponentiell mit steigendem RFC fällt und bei Smalltalk erheblich höher liegt als bei C++. Es gibt jeweils eine kleine Zahl von Klassen mit sehr hoher RFC; dies sind meist Schnittstellenklassen, die sich laut CK gut als strukturelle Treiber zum Systemtesten eignen. Auch RFC erlaubt eine Untersuchung des Entwurfs von Klassenhierarchien.

BBM fanden eine Signifikanz von $\alpha = 0.0000$ für hohe Fehlerwahrscheinlichkeit bei hoher RFC, die 6.5% der Varianz erklärt. Die Vorhersageeignung war am höchsten für die neuen oder stark geänderten Klassen (25% der Varianz erklärt) und für die UI-Klassen (36% der Varianz erklärt).

1.7.6 Mangel an Zusammenhang zwischen Methoden (LCOM)

„Lack of Cohesion in Methods“: LCOM $= max(|N| - |S|, 0)$, wobei N die Menge von Methodenpaaren ist, bei der die zwei Methoden eines Paares keine Instanzvariable gemeinsam benutzen (no sharing) und S die Menge von Methodenpaaren mit gemeinsamer Nutzung von mindestens einer Instanzvariablen (sharing). Eine solche gemeinsame Benutzung erzeugt einen Zusammenhang zwischen zwei Methoden. Das Maß gibt also an, wieviel mehr unzusammenhängende Paare von Methoden es gibt als zusammenhängende, wobei ein Überwiegen von zusammenhängenden abgeschnitten wird (also negative Werte nicht zugelassen sind, sondern auf Null abgebildet werden).

Bedeutungen laut CK:

1. Zusammenhang ist wünschenswert, weil er Kapselung unterstützt.
2. Mangel an Zusammenhang zeigt an, daß die Klasse wahrscheinlich in zwei Klassen aufgespalten werden sollte.
3. Ein Maß für Verschiedenheit von Methoden hilft beim Entdecken von Entwurfsfehlern.
4. Geringer Zusammenhang erhöht die Komplexität und die Wahrscheinlichkeit von Fehlern beim Entwicklungsprozeß.

Beobachtungen: CK fanden für ihre zwei Klassenmengen einen Median von LCOM von 0 bzw. 2. Das Maximum war bei den C++-Klassen 200, bei den Smalltalk-Klassen 17. LCOM eignet sich gut, um zu beurteilen, ob das Zusammenhangsprinzip ausreichend eingehalten wird.

BBM fanden für keine ihrer Teilmengen, daß LCOM signifikant die Vorhersage von Fehlern erlaubt. Sie führen das darauf zurück, daß das künstliche Abschneiden negativer Werte die meisten ihrer Klassen mit dem Wert 0 versieht und deshalb Klassen mit sehr unterschiedlich gutem Zusammenhang gleich bewertet werden. Diese künstliche Modifikation erscheint in diesem Licht also als glatter Entwurfsfehler bei der Definition des Maßes.

1.7.7 Diskussion

Es handelt sich bei den vorgelegten Maßen um insgesamt durchaus nützliche Grundlagen für Vorhersagesysteme, wobei die Nützlichkeit und Aussagekraft zwischen den einzelnen Maßen doch erheblich schwankt.

Bemerkenswert ist die unterschiedliche Sichtweise der beiden angeführten Arbeiten: CK benutzen den Satz von Maßen als Ideenlieferant für die Bewertung von Entwürfen. Zu jedem Maß gibt es mehrere Überlegungen zu möglichen Bedeutungen gewisser Meßwerte, wobei sich diese Überlegungen zum Teil sogar widersprechen. Die Betrachtung des Histogramms der Meßwertverteilung wird als Mittel zur qualitativen Beurteilung gewisser Entwurfseigenschaften von Klassenmengen angesehen. Der ganze Ansatz dokumentiert eine Einstellung der Art „wir haben da so einen Satz netter Maße, jetzt laßt uns doch mal schauen, was man damit vielleicht anfangen kann“. Zwar treffen CK mehrere Aussagen der Art „mit dem Maß XY läßt sich dieses oder jenes beurteilen“, aber nachgewiesen sind diese Behauptungen noch nicht, es handelt sich lediglich um Vermutungen.

Demgegenüber haben BBM ein klares Ziel: Sie wollen vorhersagen, welche Klassen voraussichtlich Fehler haben werden. Mit einer solchen Vorhersage kann man den Entwicklungsprozeß so steuern, daß kritische Klassen besonders sorgfältig implementiert und getestet werden, um die Qualität der Software insgesamt zu optimieren. Mit diesem klaren Ziel können BBM eine erheblich strukturiertere Vorgehensweise in der empirischen Untersuchung verwenden, die tatsächlich den Namen Validierung im oben eingeführten Sinne verdient. Sie haben eine Hypothese, die klar genug ist, um mit statistischen Methoden getestet werden zu können, nämlich daß ein Maß die Vorhersage von Fehlern besser erlaubt als eine optimale Zufallsentscheidung. Durch die Erklärung der Varianz kann die Nützlichkeit jedes Maßes zur Vorhersage bei Zutreffen der Hypothese auch quantitativ beschrieben werden.

Bei BBM ergibt sich als klare Aussage, daß RFC, DIT und eventuell auch NOC brauchbare Hinweisgeber auf fehleranfällige Klassen darstellen und das LCOM anders definiert werden sollte. Bei CK bleiben hingegen die Erkenntnisse vage und allgemein. Dies verdeutlicht, wie wichtig ein erkenntnisorientiertes Vorgehen bei der Validierung von Maßen ist. Optimal wäre es, den zielorientierten Ansatz von BBM mit den industriellen Daten von CK auszuführen. Solche Vorhaben scheitern leider meistens daran, daß im industriellen Kontext nicht alle benötigten Daten (hier: wo sind denn nun Fehler aufgetreten?) mit der nötigen Gleichförmigkeit gesammelt werden.

1.8 Weitergehende Information

Ein gutes allgemeines Buch über Messen und Maße ist [Fen91]. Es deckt viele Aspekte ab und geht erheblich weiter ins Detail, als das in diesem Skript möglich ist.

Ein Artikel, der zwar nichts mit Softwaretechnik zu tun hat, aber sehr wohl mit Messen ist „12 ways to fool the masses“, der unter http://www.nas.nasa.gov/NAS/TechReports/RNRreports/dbailey/RNR-91-020/RNR-91-020.html verfügbar ist.

Abschnitt 2

Psychologie und Softwaretechnik

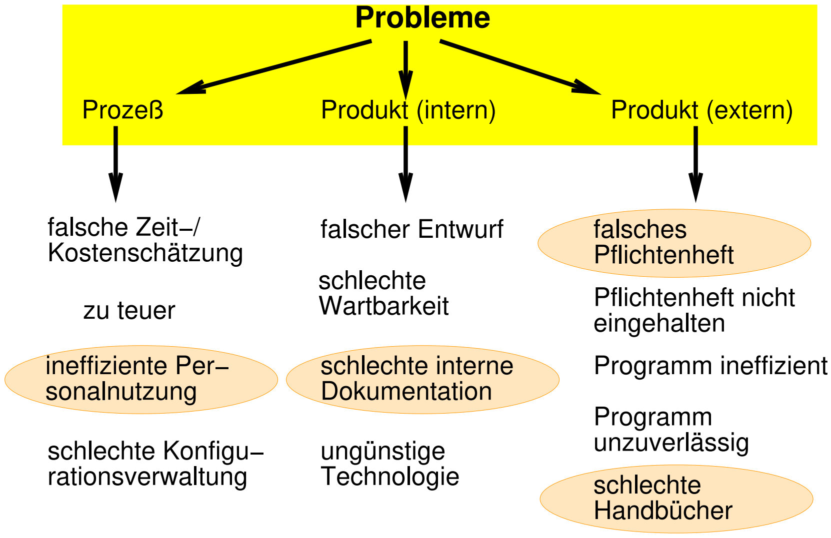

Abbildung 2.1: Von der Psychologie angesprochene Problembereiche.

2.1 Einordnung und Ziele

Psychologische Effekte sind an vielen Stellen von erheblicher Bedeutung für die Herstellung von Software — sowohl bei einer einzelnen Person als auch vor allem bei Teams. Eine ausreichende Kenntnis dieser Effekte ist Grundlage dafür, Schwierigkeiten mit psychologischen Ursachen möglichst gut zu vermeiden und sogar psychologische Effekte gezielt zur Verbesserung der Softwareherstellung einzusetzen.

In diesem Kapitel werden sich Abschnitte über Lernen, Gedächtnis, Sprache, Denken und Problemlösen mit den Fähigkeiten einzelner und deren Grenzen befassen, und weitere Abschnitte über Bedürfnisse und Motivation, Streß und Sozialpsychologie mit dem menschlichen Umfeld dieser Fähigkeiten, ihrer Pflege und Nutzung.

2.2 Probleme und Hindernisse

Der überwiegende Teil der Ergebnisse psychologischer Forschung ist mit dem berühmten „gesunden Menschenverstand“ sofort einzusehen und läßt sich aus der persönlichen Lebenserfahrung sofort nachvollziehen und bestätigen.

Das bedeutet jedoch durchaus nicht, daß diese Erkenntnisse deshalb trivial wären. Vielmehr ist ihr wissenschaftlicher Nachweis Grundlage dafür, mit ihnen als Fakten gezielt umzugehen und sie nutzbringend einzusetzen. Der erste Schritt eines solchen Umgangs besteht darin, sich ein aktives Bewußtsein für die Effekte zu schaffen, so daß man sie in der Handlungsplanung auch aktiv berücksichtigt.

Die Schwierigkeit bei der Anwendung psychologischen Wissens in der Softwaretechnik besteht meist darin, die Effekte quantitativ richtig abzuschätzen, aus den möglichen Auswirkungen die wesentlichen herauszufiltern und diese dann durch Maßnahmen zu unterstützen oder zu bekämpfen, die nicht wieder negative Nebenwirkungen aufgrund anderer psychologischer Effekte haben. Die Intuition ist dabei zwar meist ein recht guter Ratgeber, täuscht aber an manchen Stellen systematisch, wie wir im folgenden sehen werden.

2.3 Lernen

Lernen wird in der Psychologie definiert als eine dauerhafte (im Gegensatz zu einer vorübergehenden) Änderung im Verhalten, die durch Übung (im Gegensatz zu automatischer „Reifung“) erfolgt. Es werden vier Arten des Lernens unterschieden, und zwar in Reihenfolge aufsteigender Komplexität:

1. Habituation (Gewöhnung): Lernen, einen Reiz zu ignorieren, der keine nützliche Information enthält, z.B. das Ticken einer Uhr. Sinn ist die Vermeidung von Reizüberflutung.
2. Klassische Konditionierung: Lernen, daß einem Reiz ein anderer folgen wird, z.B. dem Piepen des Rechners folgt eine Fehlermeldung. Zweck der Konditionierung ist es, frühzeitig Vorbereitungen für den zweiten Reiz treffen zu können.
3. Operative Konditionierung: Lernen, daß einer Aktion eine Konsequenz folgt, z.B. dem Aufdrehen des Wasserhahns folgt das Fließen des Wassers. Dieses Lernen ist die Basis gezielter nichtinstinktiver Handlungen.
4. Komplexes Lernen: Lernen, das über das Bilden von Assoziationen hinausgeht, z.B. die Anwendung einer Strategie zur Problemlösung oder die Ausbildung einer geistigen Karte einer Umgebung.

Habituation läuft ständig und unbewußt ab und ist kaum zu vermeiden, wenn die entsprechenden Randbedingungen vorliegen. **Klassische Konditionierung** läuft ebenfalls automatisch ab; ihr Ergebnis kann aber bei Bedarf leichter als eine Habituation von bewußten Überlegungen “überstimmt” werden.

Konsequenzen für die Softwaretechnik lauten, in Softwaresystemen und Softwareprozessen unbedingt Reize zu vermeiden, die meistens nutzlos sind und nur selten Bedeutung bekommen. Wird zum Beispiel bei einer bestimmten Aktion routinemäßig eine Dialogbox angezeigt, so wird eine Habituation auf deren Inhalt eintreten (d.h. der Inhalt wird ignoriert) und die Reaktion des Benutzers wird durch Konditionierung automatisiert. Kommt nun an eben dieser Stelle in seltenen Fällen auch eine andere Dialogbox vor, so muß die Habituation durchbrochen werden, damit der Inhalt der Dialogbox wahrgenommen wird und nicht die übliche automatisierte Benutzerreaktion erfolgt. Für dieses Durchbrechen wiederum kann man eine Konditionierung auf einen anderen Reiz gezielt einsetzen (z.B. den üblichen Piepston, vorausgesetzt, daß dieser bei der normalen Dialogbox nicht auftritt.)

Die **operative Konditionierung** ist im Rahmen der Softwaretechnik vor allem in Zusammenhang mit Motivationsfragen von Bedeutung. Ein Aktions-Konsequenz-Paar, dessen Konsequenz eine positive Motivationswirkung hat, kann man gezielt zum Antrainieren der Aktion verwenden, entsprechend eines mit negativer Motivationswirkung zum Abgewöhnen der Aktion.

Komplexes Lernen setzt die Herstellung mentaler Abbilder der Welt voraus und die Manipulation dieser Abbilder anstelle der Manipulation der Welt selbst. Es handelt sich also vor allem um einen Abstraktionsprozeß. Der Lernende muß die richtigen Abstraktionen finden sowie die richtigen Operationen zu ihrer Manipulation.

Als Konsequenz in der Softwaretechnik sollte man versuchen, bei allen komplexen Lernvorgängen dem Lernenden die Konzepte hinter den zu lernenden Zusammenhängen zugänglich zu machen. Diese Konzepte geben die Gemeinsamkeiten hinter den zu lernenden Zusammenhängen (also deren Essenz) wieder sowie idealerweise auch die Motivation hinter deren Entwurf (also die Begründung), denn es handelt sich ja um bewußt und gezielt für einen Zweck entworfene Artefakte. Somit liegen diese Konzepte schon nah an den Abstraktionen, die zum Lernen ausgebildet werden müssen, und ihre Kenntnis erleichtert deshalb das Lernen. Die Begründungen für die Konzepte liefern zugleich meist Hinweise auf nützliche Operationen auf den Abstraktionen und erleichtern somit das Lernen zusätzlich. Häufig ist es dabei nützlich, sich mittels Metaphern, Vergleichen etc. auf andere Konzepte zu stützen, die den Lernenden bereits geläufig sind.

Ein Beispiel wäre die Erläuterung des Konzepts „Verzeichnis“ als eine Datei, die Dateien speichert (anstatt beliebige Daten), um zu erklären, was hinter Einzelteilen wie dem Erzeugen eines Verzeichnisses, dem Betreten eines Verzeichnisses und der Angabe von Verzeichnisnamen als Teil eines Dateinamen steckt. Eine nützliche Ergänzung ist die Entwurfsbegründung, daß Verzeichnisse zum Klassifizieren und Ordnen von Dateien dienen und die Übersicht verbessern. Eine nützliche Metapher ist die des Ordners oder des Aktenschranks, wie er aus dem Büroalltag bekannt ist.

Müssen Zusammenhänge aus Daten erlernt werden, die diese Zusammenhänge nicht vollständig sichtbar machen, so wenden Menschen sogenannte **Vor-Überzeugungen** an, um sich die Zusammenhänge verständlich zu machen, d.h. ihre Abstraktionen zu vervollständigen. Dies kann zur Ausbildung falscher Abstraktionen führen, indem Zusammenhänge postuliert werden, die gar nicht wirklich gegeben sind.

Ein wichtiger Schluß aus diesem Lernmechanismus lautet, beim Lehren nach Möglichkeit Zusammenhänge vollständig sichtbar zu machen, um ein unkontrolliertes Benutzen von Vor-Überzeugungen zu vermeiden. Falls das nicht möglich ist, sollte eine Vor-Überzeugung mitgeliefert werden, die zwar nicht den korrekten Zusammenhang wiedergibt, aber eine Annäherung darstellt, die zu möglichst wenigen, möglichst unschädlichen Fehlschlüssen führt. Falls das auch nicht möglich ist, sollten wenigstens die häufigsten Vor-Überzeugungen der Lernenden identifiziert und ausdrücklich als falsch gekennzeichnet werden. In diesem Fall können die Lernenden aber eine Abstraktion überhaupt nicht vervollständigen, was einen Lernerfolg weitgehend ausschließt; irgendetwas muß die Lücke füllen.

Es gibt grundsätzlich zwei Arten, wie eine Abstraktion beim komplexen Lernen erworben werden kann: induktiv oder deduktiv. Beim **deduktiven Lernen** (Lernen des Speziellen aus dem Allgemeinen) wird eine Beschreibung des zu lernenden Zusammenhangs vorgegeben, die in Begriffen abgefaßt ist, die bereits zuvor gelernt wurden. Der Lernende analysiert diese Beschreibung und entwickelt daraus geistig die neue Abstraktion. Deduktives Lernen setzt Sprache oder sprachähnliche Systeme voraus.

Beispiel: “Ein Stringdenotierer ist eine Kette aus einem Anführungszeichen, einer Folge von beliebigen Zeichen, die nicht selber Anführungszeichen sind und einem weiteren Anführungszeichen”.

Beim **induktiven Lernen** hingegen (Lernen des Allgemeinen aus dem Speziellen) werden eine Reihe von Beispielen und Gegenbeispielen für die zu lernende Abstraktion vorgegeben. Überwiegend unbewußt wendet der Lernende eine große Zahl von früher gelernten Abstraktionen auf die Beispiele an, um deren relevante Gemeinsamkeiten und Unterschiede herauszufiltern. Die Gemeinsamkeiten werden vor dem Hintergrund der Unterschiede verallgemeinert (was ein unzuverlässiger Schritt ist!) und bilden den Inhalt der neu gelernten Abstraktion.

Beispiel: Stringdenotierer sind zum Beispiel `"ekr"`

```
"kdieiw Fduw dWIEi   "
" %(#&@"
"1858745897"
""
" "
"a"
```

und keine Stringdenotierer sind zum Beispiel

```
ekr
"alsdk OWERLKJsldk
lskdf"
"lsk"df"
"
```

Menschen sind enorm leistungsfähig beim induktiven Lernen, zumindest, wenn alle notwendigen Basisabstraktionen bekannt sind. Aus diesem Grund ist es stets vorteilhaft, das Lehren möglichst stark auf induktives Lernen zu stützen. Allerdings erlaubt das induktive Lernen nicht, mit vertretbarem Aufwand eine genaue Grenzziehung einer Abstraktion zu lernen. Zweifelsfälle können also nach einem rein induktiven Lernprozeß oft noch nicht korrekt beurteilt werden. Im obigen Beispiel sind schon eine ganze Reihe von Beispielen nötig, um die meisten Zweifelsfälle auszuschließen, obwohl sonst sicherlich drei positive und drei negative Beispiele ausgereicht hätten. Deshalb sollte eine induktive Lernlektion möglichst mit einer entsprechenden deduktiven vervollständigt werden, die nach der Ausbildung eines Verständnisses für den überwiegenden Teil des Gehalts der zu lernenden Abstraktion dann auch recht schnell aufgenommen werden kann. Zum Beispiel haben Sie die obige Beschreibung dessen, was ein Stringdenotierer ist, nur deshalb so schnell verstanden, weil sie es vorher schon wußten; andernfalls wäre es mit vorheriger Betrachtung der Beispiele erheblich schneller und vor allem sicherer gegangen.

2.4 Gedächtnis

Bei der Diskussion des Gedächtnisses ist zu unterscheiden zwischen dem sensorischen Gedächtnis (mit Speicherdauer im Subsekundenbereich), das als Teil des Wahrnehmungsapparats verstanden werden kann und deshalb hier nicht diskutiert wird, dem Kurzzeitgedächtnis und dem Langzeitgedächtnis. Gedächtnisbenutzung läuft grundsätzlich in drei Phasen ab, nämlich Kodierung, Speicherung und Abruf.

Eine weitere Unterscheidung betrifft die Art der gespeicherten Information: Fakten speichern wir im expliziten Gedächtnis, motorische Fertigkeiten im impliziten Gedächtnis. Letzteres soll uns hier ebenfalls nicht interessieren.

2.4.1 Kurzzeitgedächtnis

Die **Kapazität** des Kurzzeitgedächtnisses beträgt etwa sieben Objekte. Der Wert schwankt von einem Individuum zum anderen nicht mehr als um etwa plus oder minus zwei. Das bedeutet, daß wir eine willkürliche Folge von Ziffern der Länge 5 noch im Kurzzeitgedächtnis behalten können, beispielsweise 2 7 6 4 9, von einer Folge der Länge 15, beispielsweise 2 7 6 5 8 3 7 5 8 4 3 6 6 7 5, jedoch nur Bruchstücke. Versuchen Sie es. Jede Folge darf nur einmal durchgelesen werden und ist dann wiederzugeben.

Die **Speicherdauer** im Kurzzeitgedächtnis ist bei nur einmaliger Einspeicherung nur sehr kurz, einige Sekunden. Wollen wir den Inhalt länger behalten, so müssen wir ihn im Geiste wiederholen. Das gelingt am einfachsten bei sprachlicher Information, etwas schwieriger bei anderen Informationsarten. Wenn wir keine besonderen Vorkehrungen treffen, wird das Kurzzeitgedächtnis nach Art einer **Schlange** verwaltet. Das erste Objekt, das die Kapazität des Kurzzeitgedächtnisses übersteigt, wird also dasjenige Objekt aus dem Kurzzeitgedächtnis verdrängen, welches schon am längsten dort ist. Dem kann man begegnen, indem man die Inhalte selektiv wiederholt, wobei sie jedesmal wie neu eingespeichert behandelt werden, so daß sich wählen läßt, welches Objekt als erstes verdrängt werden soll.

Die gespeicherten Objekte müssen nicht unbedingt so elementar sein wie die Ziffern im vorigen Beispiel, denn die Information wird in sogenannten **Bündeln (chunks)** gespeichert, deren Beschaffenheit sich danach richtet, was für Inhalte das Langzeitgedächtnis zur Verfügung stellt. Ein Bündel ist jedenfalls eine semantische Einheit, ein Konzept. Dafür kommen beispielsweise Wörter in Frage. Eine Folge von fünf willkürlich gewählten bekannten Wörtern ist ebenso leicht oder schwer zu merken, wie eine Folge von fünf Ziffern oder fünf Buchstaben, obwohl die fünf Wörter zusammen wesentlich mehr als sieben Buchstaben umfassen. Bei einer langen Wortliste von mehr als neun Wörtern wird der Versuch ebenso unmöglich wie bei einer langen Buchstabenliste.

Hier ein Beispiel (wieder ist nur einmaliges Durchlesen erlaubt):

> divers ein Experiment hat Kapazität Kurzzeitgedächtnis minus Objekt plus sieben unser von zufolge zwei.

Vermutlich ist ihnen hierbei schon aufgefallen, daß Sie bemüht waren, einen Zusammenhang zwischen diesen Wörtern herzustellen, und Ihnen das auch zum Teil gelungen ist. Ohne einen solchen Zusammenhang haben sie keine Chance, alle Wörter korrekt wiederzugeben, mit Hilfe des Zusammenhangs können sie vermutlich bis auf kleine Fehler alle behalten. Der korrekte Satz lautet:

> Diversen Experimenten zufolge hat unser Kurzzeitgedächtnis eine Kapazität von sieben Objekten, plus oder minus zwei.

und Sie haben wenig Schwierigkeiten, den ganzen Satz im Kurzzeitgedächtnis zu behalten, obwohl er erheblich länger ist als 9 Wörter (nämlich 15). Dies zeigt, daß es noch höhere Ebenen von Bündelbildung geben kann als Wörter und mit Hilfe dieser Ebenen die effektive Kapazität des Kurzzeitgedächtnisses noch erheblich weiter gesteigert werden kann.

Die Folgerung aus diesen Tatsachen für die Softwaretechnik lautet, daß Sie als Softwareingenieure, Programmanwender oder Trainer versuchen sollten, möglichst **leistungsfähige Abstraktionen** im Langzeitgedächtnis auszubilden, damit ihr Kurzzeitgedächtnis eine höhere Leistungsfähigkeit zur Durchführung von Problemlösungsaufgaben erhält.

Ein beeindruckendes Beispiel für die Leistungsfähigkeit solcher Abstraktionen liefert das folgende berühmte psychologische Experiment: DeGroot ließ 1965 seine Versuchspersonen Schachstellungen auf Schachbrettern kurz betrachten und dann aus dem Kurzzeitgedächtnis wiedergeben. Wie erwartet konnten sehr gute Schachspieler dabei oftmals alle Figuren, immer aber ziemlich viele, korrekt wiedergeben. Schachanfänger hingegen konnten sich immer nur an sehr wenige Figuren korrekt erinnern. Stellte man die Figuren nun aber willkürlich auf das Brett, anstatt eine echte Stellung aus einem Schachspiel vorzugeben, so sank die Leistung der guten Spieler auf das Niveau der Anfänger ab; die Anfänger waren in beiden Fällen gleich gut. Dies zeigt, daß problemspezifische Abstraktionen und Muster im Langzeitgedächtnis die effektiv für einen bestimmten Zweck verfügbare Kapazität des Kurzzeitgedächtnisses vervielfachen können. Im Falle des Schachspiels sind diese Abstraktionen häufig auftretende Kombinationen von Positionen mehrerer Figuren, die ein konzeptuelles Bündel bilden.

Im Falle der Softwaretechnik können solche höheren Abstraktionen auf diversen Ebenen auftreten: typische Anweisungsmuster aus mehreren Teilen, z.B. eine Schleife oder eine Rekursion, typische Verfahrensmuster in Algorithmen, z.B. Teilen-und-Herrschen, typische Entwurfsmuster, z.B. Datentypen mit Konstrukturen, Selektoren und Iteratoren, typische Namen für solche Operationen, typische Aktionsfolgen zur Durchführung einer Aufgabe und so fort. Ein Erkennen solcher Muster ermöglicht die Speicherung einer wesentlich höheren Gesamtzahl von Fakten im Kurzzeitgedächtnis und damit die Möglichkeit, viele Aufgaben sehr viel schneller und sicherer zu erledigen, weil keine externen Speicher wie Zettel oder Dateien mehr zu Hilfe genommen werden müssen bzw. keine Fehler auftreten, die sonst die Erledigung der Aufgabe gefährden.

Eine weitere Möglichkeit zur Verbesserung ergibt sich aus der Beobachtung, daß offenbar jedes sensorische System über ein eigenes Kurzzeitgedächtnis verfügt. Bislang sind nur das Sprach-/Hörsystem und das visuelle System eingehend erforscht und der Beweis zweier **getrennter Kurzzeitgedächtnisse** für verbale und visuelle Informationen erbracht, aber vermutlich gibt es weitere Kurzzeitgedächtnisse für Geräusche, für Gerüche/Geschmäcke und für Tastempfindungen.

Sprachliche Informationen werden im sogenannten phonologischen Kurzzeitgedächtnis gespeichert und zwar unabhängig davon, ob sie visuell oder akustisch angeliefert worden sind. Demgegenüber werden nichtsprachliche visuelle Informationen im visuellen Kurzzeitgedächtnis gespeichert.

Das bedeutet, daß eine gleichzeitige Nutzung der beiden Kurzzeitgedächtnisse und eine damit einhergehende Erhöhung der Speichermöglichkeit erreicht werden kann, wenn für die Darstellung der Information kombinierte sprachliche und graphische Darstellungsweisen gewählt werden, vorausgesetzt, die graphischen Teile werden nicht sprachlich interpretiert sondern bildlich. Diese Erkenntnis ist von Belang für den Entwurf von Benutzerschnittstellen aller Art, Programmiersprachen, Entwurfsmethoden etc. Dabei ist allerdings zu beachten, daß visuell gespeicherte Informationen weniger genau sind als sprachlich gespeicherte. Es sollten daher in hybriden Methoden möglichst Vorkehrungen gegen Fehler in der visuellen Speicherung getroffen werden, z.B. indem man kritische visuelle Information zusätzlich noch einmal sprachlich einbringt.

2.4.2 Langzeitgedächtnis

Während die Inhalte des Kurzzeitgedächtnisses als Aktivierungen von Neuronen gespeichert werden (also als Hirnaktivität), sind die Inhalte des Langzeitgedächtnisses in Form von Verbindungen zwischen Neuronen gespeichert (also als Hirnstruktur) — die genaue Wirklichkeit ist leider noch um einiges komplizierter und deshalb noch nicht voll bekannt, aber dies ist der grundsätzliche Mechanismus.

Dieser Mechanismus begründet auch die zwei wichtigsten Eigenschaften des Langzeitgedächtnisses. Es hat eine unbegrenzte Speicherdauer und eine fast unbegrenzte Kapazität. Eine dritte Eigenschaft ist von Bedeutung, kann aber nicht ganz so leicht erklärt werden. Dies ist die Tatsache, daß für die Übertragung von Informationen ins Langzeitgedächtnis offenbar nur eine sehr **geringe Bandbreite** zur Verfügung steht. Innerhalb einer gegebenen Zeit kann man sich also nur relativ wenige Dinge neu langfristig merken. Auch hier ist die Wirklichkeit wieder recht kompliziert, weil die Bandbreite von der Speicherdauer abhängt. Wir können uns relativ viel innerhalb kurzer Zeit für einige Stunden oder Tage einprägen, aber nur wenig, was wir das ganze Leben lang behalten. Dies liegt daran, daß die Umsetzung von Informationen in Hirnstrukturen ein Prozeß ist, der mehrere Wochen dauert, während derer die Informationen auf andere, noch flüchtige Weise zwischengespeichert werden.

Die **Funktionalität** des Langzeitgedächtnisses besteht aus zwei Teilen. Erstens können wir Dinge wiedererkennen und zweitens sind zu diesen Dingen eine Vielzahl von Beziehungen abgespeichert, vor allem semantische wie zahlreiche Varianten von "Teil von" oder von "ist ein", sowie zeitliche und örtliche wie die Abfolge von Ereignissen in einer Geschichte oder Ziffern in einer Telefonnummer.

Die Speicherung erfolgt immer in möglichst abstrakter Form, weil damit auf sehr viele Details verzichtet werden kann und weniger Platz benötigt wird. So könnte zum Beispiel nach dem Lesen eines Handbuchs die Information, daß man zum Verlassen eines bestimmten Programms Ctrl-Q drücken muß, im Langzeitgedächtnis haftenbleiben, aber es ist vielleicht nicht mehr zu rekonstruieren, ob dort gestanden hatte "Zum Verlassen geben Sie Ctrl-Q ein." oder aber "Wenn Sie das Programm verlassen möchten, so drücken sie die Taste 'Ctrl', halten Sie sie fest und drücken sie dann gleichzeitig die Taste Q.". Obwohl die beiden Sätze sehr unterschiedlich sind, werden sie im Langzeitgedächtnis auf dieselbe Repräsentation abgebildet, weil die Abstraktion des Betätigens einer Control-Taste bereits verfügbar war und die Details der Formulierung ohnehin nicht von Bedeutung sind.

Dieser Mechanismus bedeutet umgekehrt, daß es für die Leichtigkeit, mit der etwas behalten werden kann, einen großen Unterschied macht, was für **Abstraktionen zur Repräsentation** des zu Erinnernden zur Verfügung stehen — und dann auch verwendet werden, denn das geschieht nur zum Teil automatisch; zu einem anderen Teil ist es eine bewußte Anstrengung, die als Elaboration bezeichnet wird. Je gründlicher die Elaboration, desto besser die Erinnerung.

Auf die Softwaretechnik bezogen heißt das, daß nicht nur, wie oben bereits erwähnt das Lernen, sondern auch das Behalten komplexer Konzepte erheblich erleichtert wird, wenn geeignete Vorbildung vorhanden ist. Deshalb sollte man auf die Vermittlung von Basiskonzepten, die hinter Produkten oder Prozessen stehen, immer großen Wert legen und den Zusammenhang eines Faktums mit diesen Basisabstraktionen jeweils verdeutlichen.

Experimente haben außerdem ergeben, daß Fakten, zu denen eine Erläuterung oder Begründung mitgeliefert wurde, besser wieder korrekt abgerufen wurden als andere, die ohne Begründung gelernt wurden. Das beruht vermutlich darauf, daß die Begründung einen weiteren **Abrufpfad** zum gewünschten Faktum eröffnet: Das Faktum konnte direkt erinnert werden oder aber die Begründung wurde gefunden und von dort aus das Faktum.

Dasselbe Prinzip liegt fast allen **mnemonischen Systemen** zugrunde. Es werden zusätzliche Pfade geschaffen, über die die gewünschte Information abgerufen werden kann. Besonders hilfreich in dieser Hinsicht sind Taxonomien, also hierarchische Ordnungen von Begriffen.

Andere Methoden bestehen zum Beispiel in der Erzeugung von Geschichten, die mehrere zu merkende Einzelteile miteinander verbinden, in der Erzeugung bildlicher Darstellungen von zwei oder mehr zu verbindenden Einzelteilen oder der Erzeugung von Merkreimen. Alle diese Methoden werden jedoch mit zunehmender Komplexität der zu merkenden Information immer schwieriger anzuwenden.

Eine Methode, die sich auch auf komplizierte Informationen anwenden läßt, ist die der **Schemata**. Ein Schema ist eine Art Schablone, in die Informationen einer gewissen Art eingetragen werden können. Die Schablone selbst repräsentiert mit einigen Lücken den Normalfall eines solchen Informationsblocks. Die Lücken müssen mit konkreten, aber wenig umfangreichen Informationen gefüllt werden. Um uns den gesamten Informationsblock zu merken, müssen wir uns dann nur noch die Abweichungen vom Normalfall merken, der ganze Rest steckt im Schema selbst, das nur einmal für alle seine zahlreichen Anwendungen im Gedächtnis sein muß.

Ein Beispiel wäre etwa ein Schema, das den Aufbau eines Kapitels in diesem Skript beschreibt. Das Schema besagt, daß es üblicherweise die Abschnitte "Einordnung und Ziele", "Ansätze", "Ergebnisse" und "Probleme" gibt, gefolgt von einer Reihe von Abschnitten über Techniken. Das Schema trifft nicht auf alle Kapitel zu, zum Beispiel nicht auf das vorliegende, macht es aber dennoch um einiges leichter, die Struktur jedes Kapitels ins Gedächtnis zu speichern, verglichen mit der Situation, in der die Gemeinsamkeiten in den Kapitelstrukturen nicht bemerkt werden und kein Schema gebildet wird. Um sich die Struktur eines Kapitels zu merken, muß man sich nun nur die Namen der Techniken merken, sowie ob irgendwelche Abschnitte ausnahmsweise entfallen oder umgestellt sind.

Wir alle verwenden Schemata andauernd und ohne uns dessen auch nur bewußt zu sein. Sie lassen sich jedoch wie gerade verdeutlicht auch gezielt einsetzen, um komplizierte Informationen stark zu komprimieren und leichter merken zu können.

Eine letzte wichtige Erkenntnis aus der Psychologie zum Thema Langzeitgedächtnis betrifft die Bedeutung des inhaltlichen **Kontextes** bzw. auch der räumlichen Umgebung für das erfolgreiche Abrufen von Erinnerungen. Godden und Baddeley prüften 1975 die Auswirkungen eines Umgebungswechsels auf die Erinnerung, indem sie Taucher eine Liste von Wörtern am Strand lernen ließen und andere Taucher im Wasser. Beide Gruppen konnten am jeweils gleichen Ort (Strand oder Wasser) die Wörter etwa gleich gut wieder abrufen. Sollten jedoch die Im-Wasser-Lerner am Strand abrufen oder umgekehrt, so sank die Abrufleistung um 40 Prozent!

Dieser Effekt kommt daher, daß unbewußt Elemente der Umgebung verwendet werden, um daran Assoziationen zu den zu merkenden Informationen zu befestigen, was das Abrufen der Informationen nur dann erleichtert, wenn diese Elemente zum Abrufzeitpunkt als Anregungen verfügbar sind. Der Effekt tritt nicht nur im Bezug auf örtliche Umgebungen, sondern auch für situative Kontexte auf. Letztere schließen auch Aspekte der internen Situation ein, z.B. die Gefühlslage oder kurz zuvor gemachte Erlebnisse. Als Folgerung sollte das Lernen von softwaretechnischem Wissen und Fertigkeiten nach Möglichkeit in anwendungsnahen Situationen und Räumen erfolgen, um den späteren Abruf zu optimieren. (Das für die Universität typische Lernen im Hörsaal ist insofern also grundsätzlich erschwert, es sei denn im Hinblick auf eine Klausur, die im gleichen Hörsaal geschrieben wird.)

2.5 Sprache, Denken, Problemlösen

In diesem Abschnitt werden viele der in den Abschnitten Lernen und Gedächtnis besprochenen Aspekte wieder auftreten. Dennoch ist eine separate Betrachtung von Sprache in unserem Zusammenhang sinnvoll, weil die Informatik und damit auch die Softwaretechnik sich zu einem großen Teil um die künstliche Erschaffung und die Benutzung von Sprachen dreht. Deshalb ist es sehr nützlich, etwas über die psychologische Funktionsweise natürlicher Sprache und den Zusammenhang von Sprache und Denken zu wissen.

2.5.1 Arten des Denkens

Zum Zwecke einer klareren Begriffsbildung kann man drei Arten des Denkens unterscheiden: Das Aussagendenken korrespondiert zu dem Strom sprachlicher Sätze, den wir im Geiste zu hören scheinen. Das Vorstellungsdenken korrespondiert zum geistigen „Sehen“, und schließlich korrespondiert das motorische Denken zu Folgen im Geiste vorgenommener Bewegungen. Diese drei Arten gehen nahtlos ineinander über und lassen sich nicht immer klar trennen. Wir wollen uns in diesem Kapitel jedoch hauptsächlich mit dem Aussagendenken befassen.

2.5.2 Konzepte als Basis von Sprache und Denken

Eine Aussage postuliert eine Beziehung zwischen Objekten oder Klassen von Objekten, z.B. „Der Befehl `rm` löscht eine Datei.“ Im Mittelpunkt einer Aussage stehen also die Konzepte, welche die betroffenen Objekte beschreiben (im Beispiel Befehl und Datei). Die Beziehungen sind ebenfalls Konzepte (im Beispiel das Löschen).

Konzepte ermöglichen eine Art von kognitiver Sparsamkeit: Dadurch, daß wir Klassen von Objekten oder Beziehungen bilden, müssen wir nicht mehr jedes einzelne Objekt oder jede einzelne Beziehung bezeichnen, was verhindert, daß das Begriffsuniversum zu nicht mehr handhabbarer Größe explodiert. Die Welt wird durch Konzepte in Teile von handhabbarer Größe zerlegt.

Ein Konzept umfaßt die Menge aller Eigenschaften, die wir mit der Objektklasse, Beziehungsklasse oder Eigenschaftsklasse verbinden, und ist insofern auch ein mächtiges Mittel zur Abkürzung von Beschreibungen. Jedes Objekt, das zu einem Konzept gehört, nennen wir ein Exemplar des Konzepts oder auch eine *Instanz* des Konzepts.

Sprachlich finden wir, vereinfacht ausgedrückt, Objektkonzepte (Gegenstände, Abstraktionen) in Form von Substantiven, Beziehungskonzepte (Zustände, Vorgänge) in Form von Verben und Eigenschaftskonzepte in Form von Adjektiven und Adverben. Konzepte bilden also die inhaltliche Basis der Sprache; abgesehen von einigen grammatikalischen Regeln, die unmittelbar für die Sinnbildung wichtig sind, ist der Rest nur noch Syntax.

Müssen wir ein einzelnes Objekt bezeichnen, anstatt einer Objektklasse, so können wir dies mit Hilfe eines Namens tun (im Beispiel `rm`). Die Übereinstimmung (Identität) von Objekten, ob benannt oder nicht, wird dann mit Hilfe der bestimmten Artikel oder von Pronomen ausgedrückt.

Die Zuweisung eines Objekts zu einem Konzept bezeichnet man als **Kategorisierung**. Diese ist eine der wichtigsten kognitiven Grundoperationen. Wenn wir ein Objekt aufgrund seiner bisher wahrgenommenen Eigenschaften kategorisieren, weisen wir ihm damit implizit auch alle anderen Eigenschaften zu, die dem Konzept zukommen, obwohl wir diese am Objekt noch gar nicht wahrgenommen haben und oft auch niemals wahrnehmen werden. Insofern ist mit jeder Kategorisierung ein Informationsgewinn verbunden. Aus diesem Grund liegt im Vorgang der

Kategorisierung aber auch ein erhöhtes Fehlerrisiko: Nach einer falschen Kategorisierung werden wir durch unser Wissen über das Konzept zu Fehlannahmen verleitet, die ganz ohne eine Kategorisierung nicht auftreten würden.

Dieser Effekt macht klar, daß im Rahmen der Softwaretechnik einer sorgfältigen und klaren Konzeptbildung und Vorsicht beim Kategorisieren allergrößte Wichtigkeit zukommt. Ein sehr großer Teil der Fehler in fast allen Phasen der Softwareentwicklung, besonders aber im Entwurf, lässt sich auf unklare Konzeptbildung oder falsche Kategorisierung zurückführen.

2.5.3 Struktur von Konzepten: Prototypen und Kerne

Die mit einem Konzept verbundenen Eigenschaften fallen in zwei nicht disjunkte Mengen. Zum einen die Menge der Eigenschaften, die den **Prototyp** des Konzepts bildet. Darin befinden sich alle Eigenschaften, die für Exemplare des Konzepts als typisch zu betrachten sind. Der Prototyp ist das, was uns in den Sinn kommt, wenn wir allgemein an das Konzept denken. Für das Konzept „Computer" könnten dazu zum Beispiel diverse Einzelteile wie Hauptplatine, Bildschirm, Tastatur, Gehäuse, Platte gehören, sowie die Eigenschaften „hat Betriebssystem" und „kann Programme ausführen" etc. Nicht alle Eigenschaften des Prototyps treffen jedoch auf jedes Exemplar des Konzepts zu.

Eine andere Menge von Eigenschaften bildet die Essenz des Konzepts. Diese Menge nennen wir seinen **Kern**. Alle Eigenschaften des Kerns treffen auf jedes Exemplar des Konzepts zu, sind also notwendig. Sie sind außerdem hinreichend, um eine Mitgliedschaft im Konzept festzustellen. Sehr oft, aber nicht immer, ist der Kern ein Teil des Prototyps. Für das Konzept „Vogel" gehören z.B. zum Prototyp nur Äußerlichkeiten und Verhaltensweisen, der Kern wird jedoch über die Genetik definiert. Im Falle des Computers gehören zum Kern wohl nur das Vorhandensein von Speicher und von Prozessor oder Prozessoräquivalent. Genauer sollte man jedoch von einem digitalen Computer sprechen, und dann handelt es sich um digitalen Speicher und einen digitalen, deterministischen Prozessor. Mit einer anderen Definition hat man nämlich wahrscheinlich ein unscharfes und damit fehlerprovozierendes Konzept gebildet.

Wann sind Konzepte **unscharf**? Um zu prüfen, ob ein Objekt zu einem Konzept gehört, müssen wir im Prinzip prüfen, ob alle Eigenschaften des Konzeptkerns vom Objekt erfüllt werden. Ist dies bequem möglich, so sprechen wir von einem klassischen Konzept. Bei vielen Konzepten macht eine solche Prüfung jedoch erhebliche Schwierigkeiten. Wir ersetzen dann die Prüfung des Kerns durch eine Prüfung auf "genügende" Ähnlichkeit mit dem Prototyp. Solche Konzepte nennen wir unscharfe Konzepte.

Nach dieser Definition sind übrigens die Konzepte, die durch die sogenannte unscharfe Logik (Fuzzy Logic) beschrieben werden, stets klassische Konzepte, denn Exemplare werden durch eine mathematisch wohldefinierte Methode eindeutig gegen eine Kernanforderung auf Mitgliedschaft im Konzept geprüft. In der realen Welt sind dagegen fast alle Konzepte unscharf, meist weil überhaupt keine präzise Definition ihres Kerns vorliegt.

Tatsächlich unscharfe Konzepte sollten in der Softwaretechnik so weit es geht vermieden werden; stattdessen ist immer eine exakte Definition anzustreben. Wo das nicht geht, sollten unscharfe Konzepte, z.B. in einem Pflichtenheft oder einem vorläufigen Entwurf, stets deutlich als solche gekennzeichnet werden, um die mit der Unschärfe verbundenen begrifflichen Fallen (voreilige Schlüsse aus einer Kategorisierung) besser vermeiden zu können.

2.5.4 Erwerb von Konzepten

Beim Erlernen von Konzepten kann man unterscheiden zwischen dem Erwerb eines Prototyps und dem Erwerb eines Kerns. Kerne sind üblicherweise das, was beim expliziten Lehren übermittelt wird. Demgegenüber lernen wir beim Lernen aus Erfahrung auch und vor allem einen Prototyp.

In beiden Fällen muß der jeweils fehlende Teil irgendwann ergänzt werden. Ein explizit gelernter Kern wird nach und nach durch Erfahrung um einen zugehörigen Prototypen erweitert und beim Lernen aus Erfahrung kann irgendwann aus den Erfahrungen der Kern des Konzepts herausdestilliert werden — zumindest bei klassischen Konzepten.

Für die Softwaretechnik ist es wichtig, in der Lernphase zu wissen, wieviel von welchen Teilen der Konzeptbeschreibung man vermutlich schon erworben hat. Für den Kern muß man sich fragen, wie vollständig man ihn bereits erfaßt hat und ob diese Erfassung korrekt ist; für den Prototyp muß man sich fragen, wie genau er bisher ist.

Für das Lernen von Konzepten aus Erfahrung gibt es drei unterschiedliche Strategien:

1. Die **Exemplarstrategie** speichert alle bislang aufgetretenen Exemplare des Konzepts ab und bestimmt das Konzept eines neuen Objekts durch eine Suche nach der Exemplarmenge, deren Exemplare im Mittel die größte Ähnlichkeit zum neuen Objekt aufweisen. Diese Strategie hat den Vorzug, nur sehr wenig Weltwissen zu benötigen: Es ist lediglich zu klären, welches die geeignetsten Ähnlichkeitsmaße sind.

2. Beim **Hypothesentesten** wird ebenfalls eine Sammlung bisher bekannter Exemplare angelegt, diese dann jedoch in anderer Weise verwendet: Es werden einige Exemplare nach systematischen Gemeinsamkeiten und Unterschieden untersucht. Aus diesen werden Hypothesen über die Eigenschaften hergeleitet, die das Konzept ausmachen. Diese Hypothesen werden dann anhand der übrigen Exemplare geprüft und entweder akzeptiert oder verworfen. Beim Verwerfen kann die Abweichung vom Erwarteten oft zu einer Modifikation der Hypothese benutzt werden.

3. Beim **Top-Down-Lernen** werden die beiden obigen Strategien ebenfalls eingesetzt, jedoch erweitert um die Anwendung von Vorwissen. Dieses Vorwissen beschreibt zum Beispiel, welche Arten von Ähnlichkeitsmaßen erfolgversprechend sind oder nicht (z.B. „Farbe ist meist nicht aussagekräftig, wenn es sich um künstliche Gegenstände handelt.“) oder welche Arten von Hypothesen untersucht werden sollten (z.B. „Wenn es um Tiere geht, achte auf die Art der Oberfläche (Federn, Haut, Haare etc.)“).

Das Top-Down-Lernen ist weitaus am mächtigsten, da es durch Wissenstransfer von bereits bekannten Konzepten erlaubt, schon aus einer sehr kleinen Zahl von Beispielen eine sehr genaue Konzeptbeschreibung herzuleiten. Es birgt jedoch aufgrund eben dieser Mächtigkeit auch die Möglichkeit zu besonders krassen Fehleinschätzungen, wenn nämlich eine der eingebrachten Annahmen und damit ein großer Teil des eingebrachten Wissens falsch sind.

In der Softwaretechnik sollte deshalb in kritischen Fällen möglichst eine bewußte Auswahl zwischen den Konzepterwerbsmethoden getroffen und ggf. das Top-Down-Lernen nur mit Vorsicht eingesetzt werden. Zum Beispiel handelt es sich beim Einkreisen eines Programmfehlers mittels Debugging je nach Vorgehensweise anfänglich um einen Konzepterwerb, nämlich dann, wenn die Frage lautet „Unter welchen Umständen tritt der Fehler auf?“. In diesem Fall kann eine verfrühte Anwendung der Top-Down-Methode völlig in die Irre führen, indem Ergebnisse einzelner Testläufe überinterpretiert werden und daraufhin die Suche zu früh auf einen (dann falschen) Programmbereich eingeengt wird. Stattdessen sollten die Gefahren der Überinterpretation bewußt wahrgenommen und eine ausreichend lange Zeit ausschließlich mit der Methode des Hypothesentestens gearbeitet werden. Ähnliche Verfahrensregeln sollte man beim Erlernen von sehr neuartigen Konzepten benutzen, z.B. wenn es um das Konzept „günstige Anwendungsfälle für Vererbung“ geht und noch wenig Erfahrung mit dem Konzept der Vererbung vorliegt. Die Nichtbeachtung der Regeln in genau diesem Fall beschert derzeit der softwaretechnischen Praxis eine wundervolle neue Ladung zukünftiger unwartbarer Altlasten.

2.5.5 Schließen

Konzepte bilden auch die Basis für logisches Schließen, denn sie liefern zu Objekten die Eigenschaften, aufgrund derer Schlüsse gemacht werden können.

Taxonomien

Den einfachsten Fall des Schließens liefert die Anwendung von Taxonomien. Die wichtigste Taxonomie ist die Ordnung nach Ober- und Unterkonzepten, wobei ein Unterkonzept einfach eine Teilmenge der Objektklasse des Oberkonzepts ist. Eine solche Taxonomie ermöglicht sehr einfache deduktive Schlüsse aufgrund der Tatsache, daß alle Eigenschaften eines Konzepts wegen der Teilmengenbeziehung für Unterkonzepte ebenfalls gelten.

Eine immer noch nicht ausgerottete Verwechslung in der Softwaretechnik ist der Glaube, die Vererbungshierarchie beim objektorientierten Programmieren beschreibe eine solche „ist ein“–Hierarchie. Das ist nicht richtig. Technisch bedeutet die Vererbung in den heute üblichen oo-Sprachen eine Typerweiterung und Verhaltensmodifikation. Dabei müssen die Eigenschaften eines Obertyps im Untertyp nicht erhalten bleiben. Ein sehr einfaches Beispiel sind die Klassen „Rechteck“ und „Quadrat“. Zwar ist Quadrat ein Unterkonzept von Rechteck, jedoch wird zur Darstellung eines Rechtecks die Länge von zwei Seiten benötigt, was zur Modellierung eines Quadrats zumindest unnatürlich ist und dem Verfahren der Typerweiterung in Unterklassen direkt zuwiderläuft. Ein Operationenpaar `setzeLänge` und `setzeBreite`, welches zusichert, daß die jeweils andere Seitenlänge konstant bleibt, darf in der Quadratklasse nicht erhalten bleiben, weil sonst Quadratinstanzen die Quadrateigenschaft verlieren können. In diesem Fall macht also die Unterklasse nicht alle Zusicherungen der Oberklasse und verletzt damit die „ist ein“–Hierarchie.

Andere Taxonomien von nicht so großem praktischen Wert für das Schließen können über andere Beziehungen gebildet werden, z.B. „ist ein Teil von“ (Teil-Ganzes-Relation) oder „zieht nach sich“ (Kausalkettenrelation).

Deduktives Schließen

Das deduktive Schließen ist Informatikern aus ihrer Ausbildung hinreichend bekannt, samt der erheblichen Schwierigkeiten, es immer korrekt zu machen. Interessant ist jedoch die psychologische Beobachtung, daß die Fehlerquote erheblich vom **Inhalt des Schlusses** beeinflußt wird, obwohl das deduktive Schließen im Prinzip eine rein formale, also inhaltsunabhängige Technik ist.

Betrachten wir als Beispiel dazu die Abbildung 2.2. Im oberen Teil sehen wir Version 1 eines Testproblems. Die Testpersonen bekamen gesagt, die eine Seite jeder Karte enthielte einen Buchstaben und die andere eine Zahl. Sie wurden gefragt, welche Karten man umdrehen müsse, um die Aussage zu prüfen „Wenn eine Karte einen Vokal auf der einen Seite hat, steht eine gerade Zahl auf der anderen.“

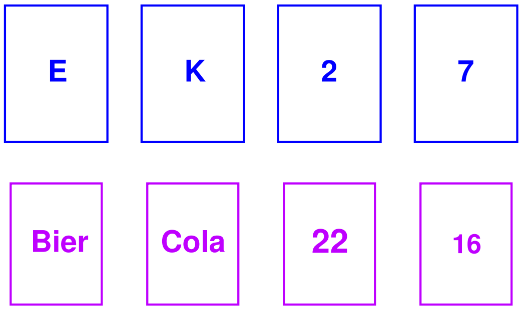

Abbildung 2.2: Versionen 1 und 2 des Testproblems zum Nachweis der Inhaltsabhängigkeit deduktiven Schließens.

Die Häufigkeit falscher Antworten für dieses Problem nahm erheblich ab, wenn man das Problem wie in Version 2 umformulierte: Eine Seite jeder Karte enthält ein Getränk, die andere Seite die Altersangabe für die zum Getränk gehörige Person. Welche Karten muß man umdrehen, um die Aussage zu prüfen „Wenn eine Person Bier trinkt, so ist sie mindestens 18 Jahre alt.“ Diese zweite Fassung ist exakt äquivalent zur ersten, dennoch treten bei der Beantwortung seltener Fehler auf. (Die richtigen Antworten lauten übrigens E und 7 beziehungsweise Bier und 16.)

Wie kommt es zu solcher Inhaltsabhängigkeit? Es gibt zwei Möglichkeiten, wie dies zustande kommen kann. Die erste ist die Anwendung **pragmatischer Regeln**, die bereits vorher bekannt sind. In diesem Fall ist das die Erkenntnis, daß es zum Prüfen ausreicht, bei Biertrinkern zu kontrollieren, ob sie volljährig sind, und wo man das Getränk nicht weiß, alle Minderjährigen zu kontrollieren. Diese Regel hätte man auch auf Version 1 anwenden können, jedoch war dort ihre Gültigkeit nicht ersichtlich.

Die zweite Möglichkeit ist das Bilden **mentaler Modelle**, die die Durchführung der Deduktion erleichtern. Im vorliegenden Fall könnte man sich vorstellen, daß alle Personen (Karten) mit einem Bier in der Hand (Kartenvorderseite) und einer Altersangabe auf dem Rücken (Kartenrückseite) dasitzen. Um die Aussage zu prüfen, muß ich also bei allen Biertrinkern den Rücken ansehen (Karte umdrehen) und bei allen Minderjährigen das Getränk (auch Karte umdrehen).

Beide Vorgehensweisen sind Tricks, die sich in der Softwaretechnik zum deduktiven Schließen ebenfalls einsetzen lassen. Hierzu müssen Software-Situationen auf bekannte andere Software-Situationen zurückgeführt werden (pragmatische Regeln) oder durch Metaphern auf andere Situationen abgebildet werden, die man leichter versteht (mentale Modelle).

Induktives Schließen

Korrekte deduktive Argumente heißen „deduktiv gültig“ und sind die einzigen sicheren logischen Schlüsse (Vollständige Induktion ist ein deduktives Prinzip!). Dennoch gibt es Argumente, die gut und recht zuverlässig sind, obwohl sie nicht notwendigerweise korrekt sein müssen — es ist nur wahrscheinlich, daß sie gelten. Solche Argumente werden **induktiv stark** genannt. Hier ein Beispiel:

1. Johanna hat Informatik studiert.

2. Johanna arbeitet in der Softwareabteilung von VW.

3. (Schluß:) Johanna arbeitet als Informatikerin.

Das Argument könnte falsch sein, denn vielleicht hat Johanna ja ihr Studium unterwegs abgebrochen oder arbeitet nicht in ihrem erlernten Beruf, sondern in einem anderen. Das ist jedoch nicht wahrscheinlich. Die Stärke solcher Argumente basiert auf den Regeln der Wahrscheinlichkeitsrechnung sowie auf unseren Erfahrungen, die die entsprechenden Wahrscheinlichkeiten bestimmen. Im obigen Beispiel etwa hätten Sie vermutlich das Argument als noch stärker empfunden, wenn statt Johanna dort Johann gestanden hätte.

Wir machen solche induktiven Argumente ständig und ohne uns dessen überhaupt bewußt zu sein. Zu den dabei relevanten mathematischen Regeln gehören insbesondere die Basisratenregel und die Konjunktionsregel. Die **Basisratenregel** besagt, daß die Wahrscheinlichkeit, daß ein Objekt zu einer bestimmten Klasse gehört, umso größer ist, je größer der Anteil dieser Klasse am ganzen Universum ist. Im Beispiel also wird das Argument umso stärker, je mehr Berufstätige als Informatiker arbeiten. Das Argument würde noch besser, wenn wir als eine Prämisse noch wüßten, daß Johanna auch Mitglied der Gesellschaft für Informatik ist, denn dort ist der Anteil von Leuten, die als Informatiker arbeiten, besonders hoch.

Die **Konjunktionsregel** besagt, daß die Wahrscheinlichkeit einer beliebigen Aussage nicht zunehmen kann, wenn ich sie konjunktiv mit einer weiteren verknüpfe. Zum Beispiel kann obiges Argument nicht stärker werden, wenn ich die Aussage 3 erweitere zu „Johanna arbeitet als Informatikerin und verdient 85.000 DM im Jahr.“ Interessanterweise halten wir beim induktiven Schließen diese beiden Regeln oftmals nicht ein, wie die folgenden Beispiele zeigen.

Tversky und Kahneman fanden 1973, daß Information über Basisraten manchmal schlichtweg ignoriert wird: Sie gaben Testpersonen Personenbeschreibungen von Ingenieuren und von Anwälten zu lesen und baten sie, bei jeder zu entscheiden, ob sie zu einem Anwalt oder einem Ingenieur gehört. Zwei Gruppen von Testpersonen gaben etwa gleiche Urteile ab, obwohl der einen Gruppe gesagt wurde, unter den 100 Beschreibungen seien 70 Anwälte und 30 Ingenieure und der anderen, es sei gerade andersherum. Solche Fehler werden wahrscheinlich, wenn die Basisraten nur abstrakt über Zahlen bekannt sind und weniger wahrscheinlich, wenn sie mit eigenen Erfahrungen direkt wahrgenommen werden. Aus diesem Grund sind Urteile von Praktikern oft erheblich verläßlicher als die von Theoretikern, sogar wenn die Theoretiker ein eigentlich umfangreicheres Wissen haben.

Die Konjuktionsregel wird ebenfalls manchmal verletzt, siehe das folgende Beispiel: Gegeben sei die Beschreibung

> Das Pflichtenheft war vollständig in Textform abgefaßt, völlig ohne Tabellen oder Abbildungen. Es war unstrukturiert und unübersichtlich und wies zahlreiche inhaltliche Mängel und Fehler auf.

und zu beurteilen sei die Wahrscheinlichkeit der folgenden beiden Aussagen:

1. Im Pflichtenheft waren 12 wesentliche Anforderungen nicht erfaßt.
2. Im Pflichtenheft waren 12 wesentliche Anforderungen nicht erfaßt und weitere 9 waren unklar oder mehrdeutig formuliert.

Testpersonen schätzen die Wahrscheinlichkeit von Aussage 2 oft höher ein als die von Aussage 1, obwohl das logisch unmöglich ist. Ursache dieses Fehlers ist eine Verwechselung von Ähnlichkeit mit Wahrscheinlichkeit. Aussage 2 erscheint als eine Beschreibung, die unserem Konzept von einem schlechten Pflichtenheft näher kommt als Aussage 1. Deshalb wurde sie auch als wahrscheinlicher eingestuft, obwohl sie unwahrscheinlicher sein muss, weil sie eine zusätzliche Bedingung enthält (die bei Aussage 1 erfüllt sein kann oder auch nicht). Der gleiche Fehler kann statt mit Ähnlichkeiten mit zusätzlichen Begründungen passieren. So wird von Testpersonen die Aussage

> Im Jahr 2007 wird es eine massive Überflutung in Nordamerika geben, der mehr als tausend Menschen zum Opfer fallen.

als weniger wahrscheinlich eingestuft als die Aussage

> Im Jahr 2007 wird es ein Erdbeben in Kalifornien geben, das eine massive Überflutung hervorruft, der mehr als tausend Menschen zum Opfer fallen.

Letzteres ist zwar logisch weniger wahrscheinlich, erscheint uns aber subjektiv wahrscheinlicher, weil es innerhalb der Aussage durch eine Kausalbegründung plausibel gemacht wird. Wohlgemerkt: Wenn die Wahrscheinlichkeit absolut einzuschätzen ist, kann es durchaus sein und ist sogar zu erwarten, daß die Schätzung für Aussage 2 zwar unlogischerweise höher ausfällt als die für Aussage 1, jedoch absolut gesehen erheblich besser ist. Das liegt daran, daß unser Augenmerk innerhalb der Aussage 2 auf einen wichtigen Faktor zur Einschätzung der Wahrscheinlichkeit gelenkt wird, den wir bei der Abschätzung von Aussage 1 eventuell übersehen und dann die Wahrscheinlichkeit unterschätzen.

Ein weiterer wichtiger psychologischer Effekt beim induktiven Schließen ist die sogenannte **Bestätigungsneigung**: Wenn wir eine Hypothese aufgestellt haben und sammeln nun Informationen, um diese Hypothese zu überprüfen, neigen wir dazu, die Informationen in einer Weise falsch zu interpretieren, die zur Bestätigung der Hypothese führt. Häufiger noch wirkt sich der Effekt dahingehend aus, daß Prüfungen, deren Ergebnis zur Ablehnung der Hypothese führen könnte, unbewußt gar nicht erst gemacht werden. Dieser Effekt kommt zum Beispiel häufig beim Überprüfen von deduktiven Schlüssen zum Tragen, wo wir dazu neigen, beispielsweise beim Prüfen einer allquantifizierten Konklusion ein positives Beispiel zu suchen („Ja, könnte stimmen.“), was nichts über die Richtigkeit der Aussage besagt, anstatt nach einem Gegenbeispiel zu suchen („Ich sehe nichts, weshalb das nicht stimmen sollte.“), was gegebenenfalls, also wenn wir tatsächlich eines finden, die Falschheit der Aussage beweist.

Für die Softwaretechnik sollte man sich diese häufigen Arten von Fehlschlüssen gut klarmachen und an entsprechenden Stellen genau prüfen, ob man vielleicht gerade dabei ist, ihnen aufzusitzen. [TLR93] bestätigt beispielsweise eine beträchtliche Bestätigungsneigung für den Fall des Programmtestens.

2.5.6 Probleme lösen

Die Psychologie identifiziert drei grundsätzlich verschiedene universelle Arten des Problemlösens:

1. **Abstandsverminderung** ist eine Methode, die dem Problemlösen eine räumliche Metapher aufprägt. Der momentane Zustand und der erwünschte Zielzustand werden als Orte aufgefaßt und der Problemlösungsprozeß als Weg zwischen diesen Orten. Die Abstandsverminderungsstrategie bedeutet nun, im nächsten Schritt immer die Operation auszuführen, die den Abstand zum Zielpunkt möglichst stark verkleinert. Dies führt gelegentlich in Sackgassen, in denen dann das Problem erweitert oder umformuliert werden muß, um wieder hinauszugelangen.

2. **Zweck/Mittel-Analyse** funktioniert ähnlich, jedoch wird nicht eine möglichst große Abstandsverringerung angestrebt, sondern in jedem Schritt wird der *wichtigste* Unterschied zwischen Zustand und Ziel identifiziert, die Beseitigung dieses Unterschieds zum Unterproblem erklärt und dann ein Schritt gewählt, der die Lösung des Unterproblems anstrebt. Ab einer gewissen Verschachtelung von Unterproblemen werden diese Schritte dann einfache Abstandsverminderungsschritte sein.

3. **Rückwärtsarbeiten** beginnt hingegen die Problemlösung nicht am Ausgangspunkt, sondern am Ziel. Es fragt quasi immer „Was müßte ich denn als letztes tun, um ans Ziel zu kommen?" und setzt als neues Ziel dann den Ausgangspunkt dieses letzten Schrittes. Eine solche Vorgehensweise eignet sich zum Beispiel oft für mathematische oder anderweitig sehr abstrakte Probleme, bei denen man den Lösungsweg erst versteht, wenn man ihn im Ganzen sieht.

Im konkreten Einsatz von Problemlösungsmethoden gibt es erhebliche Unterschiede zwischen **Experten** und Laien. Der wohl wichtigste besteht im Vorhandensein reicher Repräsentationen: Experten verfügen über problemspezifische Darstellungsweisen für Zustände und komplexe Operationen, die eine erheblich einfacheren geistigen Umgang mit den möglichen Lösungsschritten vor ihrer Durchführung erlauben. Solche Repräsentationen ermöglichen eine so starke Kompression der zu handhabenden Problemgegenstände, daß Denkoperationen durchgeführt werden können, die ohne Kompression zum Beispiel aufgrund der begrenzten Kapazität des Kurzzeitgedächtnisses unmöglich wären. Der Weg, auf dem ein werdender Experte zu solchen Repräsentationen gelangt, ist die Ausbildung einer großen Zahl nützlicher Konzepte, die überwiegend das typische Zusammenwirken anderer Konzepte beschreiben.

Zwei weitere Unterschiede im Vorgehen zwischen Experten und Laien können eher als Symptome für bei den Laien fehlendes Problemlösungswissen gedeutet werden: So haben oder machen Experten fast immer einen Plan, bevor sie mit einer Problemlösung beginnen, gewissermaßen eine Handskizze, die den Weg zum Ziel beschreibt. Dieser Plan erleichtert dann unterwegs die Orientierung und vermeidet ein Abgleiten in irrelevante Bereiche. Ferner wenden Laien wesentlich häufiger als Experten das Rückwärtsarbeiten an, ein Zeichen für ihre Unfähigkeit, wichtige Unterprobleme zu identifizieren.

Auch hier wieder ist der Nutzen dieser Erkenntnisse für die Softwaretechnik, daß die bewußte Kenntnis der Methoden eine angemessenere Auswahl in einer gegebenen Situation erlaubt und die bewußte Kenntnis von nützlichen geistigen Werkzeugen ein gezielteres Hinarbeiten auf deren Erwerb.

2.6 Bedürfnisse und Motivation

Die Aktivitäten eines Menschen werden wesentlich von seinen Bedürfnissen und Motivationen geprägt. Neben den Grundbedürfnissen, die durch Triebe realisiert werden, gibt es eine Reihe von sozialen Bedürfnissen, die darüber liegen und für uns von Interesse sind. Der Antrieb zur Befriedigung dieser Bedürfnisse wird durch Motivationen hergestellt.

2.6.1 Arten und Stärke von Motivationen

Motivation im psychologischen Sinne ist also nicht das, was dieses Wort zumeist im Zusammenhang mit Berufsausübung bedeutet. Üblicherweise wird unter Motivation eine hohe Bereitschaft zur Leistungserbringung im Sinne eines betriebswirtschaftlichen Nutzenbegriffs verstanden. Demgegenüber steht der psychologische Motivationsbegriff für einen Drang zu Aktivität, ob sie nun betriebswirtschaftlich nützlich ist oder nicht. Der psychologische Motivationsbegriff hat also betriebswirtschaftlich gesehen nicht an sich eine positive Nebenbedeutung, sondern ist neutral: hohe Motivation kann gut sein, aber auch schlecht; je nachdem, wie sie sich konkret auswirkt.

Motivationen entstammen verschiedenen Quellen:

1. Ehrgeiz. Ehrgeiz kann als die menschliche Neigung definiert werden, Hindernisse zu überwinden, und zwar so schnell und so gut wie möglich.
2. Machtstreben. Jeder Mensch hat grundsätzlich den Wunsch, Kontrolle über seine Umgebung auszuüben, einschließlich des Verhaltens seiner Mitmenschen.
3. Soziale Bedürfnisse. Menschen haben außerdem grundsätzlich ein Bedürfnis nach sozialen Beziehungen zu anderen Menschen.
4. Neugier. Menschen neigen auch abgesehen von anderen Motivationen dazu, ihre Umgebung zu manipulieren und zu erforschen: Sie sind neugierig.

Alle diese Motivationsarten sind grundsätzlich in jedem Menschen vorhanden. Die Stärke einzelner Arten ist aber von Person zu Person unterschiedlich und hat die Eigenschaften eines Charakterzugs, kann also als weitgehend konstant angesehen werden. Die Stärke einer Motivation in einem konkreten Einzelfall setzt sich aber über diese Grundmotivation hinaus aus zwei **weiteren Faktoren** zusammen: Den Erfolgsaussichten und dem subjektiven Wert eines Ziels.

So mag beispielsweise, gegeben eine erhebliche Ehrgeiz-Grundmotivation, der **subjektive Wert** eines Nobelpreises für mich sehr hoch sein; dennoch ist meine Motivation, nach ihm zu streben, gering, wenn ich meine **Erfolgsaussichten** als verschwindend klein einstufe. Umgekehrt kann ich die Erfolgsaussichten des Unterfangens, die Seiten 45 bis 50 des Karlsruher Telefonbuchs komplett auswendig zu lernen, durchaus als hoch einstufen und bin dennoch kaum motiviert, das zu versuchen, weil der subjektive Wert einer solchen Tat zu nahe bei Null liegt. Schließlich könnten sowohl die Erfolgsaussichten als auch der subjektive Wert des Ziels, eine Firmenabteilung mit über hundert Mitarbeitern zu leiten, hoch sein und dennoch die Motivation gering, es anzustreben, weil die Grundmotivation zur Machtausübung klein ist.

Für die Softwaretechnik ist die Beobachtung von Belang, daß sich sowohl die Einschätzung der Erfolgsaussichten eines Menschen für ein Ziel als auch der subjektive Wert dieses Ziels für diesen Menschen von außen beeinflussen lassen: Ersteres durch entsprechenden Zuspruch („Ich traue Ihnen das ohne weiteres zu.“), durch Bereitstellung von Hilfe oder durch Ausbildung, letzteres durch Aussicht auf Belohnung. Die Stärke von Motivation ist also einem gezielten Management in gewissem Rahmen durchaus zugänglich. Kern solcher Maßnahmen sollte aber in jedem Fall die möglichst genaue Einschätzung der schwer veränderlichen Grundmotivation sein — insbesondere der eigenen.

2.6.2 Motivation und Belohnung

Der am stärksten variable der drei Motivationsfaktoren ist meist der subjektive Wert. Dieser kann nämlich durch das Eintreten oder Ausbleiben einer **Belohnung** (externen Bestärkung) in früheren ähnlichen Situationen schneller und stärker geändert werden als die Erfolgsaussichten oder gar die Grundmotivation.

Es gibt eine Reihe von psychologischen Erkenntnissen über Effekte in diesem Zusammenhang:

1. Ein mehrfaches Ausbleiben einer Belohnung bei wiederholter Ausführung einer Tat führt tendenziell zum Abnehmen der Motivation — bis hin zum völligen Ausbleiben der Handlung, zum Beispiel dem Abliefern eines Wochenberichts, den sowieso niemand liest.

2. Viele Handlungen sind kaum auf Belohnungen angewiesen, um die Motivation zu erhalten, weil sie eine sogenannte **intrinsische Motivation** liefern, d.h. eine Motivation, die der Handlung selbst innewohnt. Das Überwinden von Hindernissen und die Ausübung von Kontrolle scheint für die meisten Menschen eine hohe intrinsische Motivation zu haben; man könnte das zusammenfassen unter „Dinge geschehen machen." Deshalb ist die Softwaretechnik ein sehr motivierender Beruf.

3. Belohnung erhöht im allgemeinen die intrinsische Motivation einer Handlung; die motivierende Wirkung der Belohnung wird auf die Handlung an sich übertragen.

4. Externe Bestärkung kann die intrinsische Motivation hinter einer Handlung verringern. Belohungen können also, falsch angewandt, die Motivation senken statt sie zu erhöhen. Dies tritt vor allem dann ein, wenn die Belohnung nicht von der Leistung abhängt, die bei der Durchführung der Handlung erbracht wurde.

5. Mehrfaches Versagen bei einer Handlung kann zu „erlernter Hilflosigkeit" führen: Es wird gar nicht mehr der Versuch unternommen, die Handlung erfolgreich auszuführen, sondern von vornherein aufgegeben. Dieses völlige Schwinden von Motivation kann sogar dann anhalten, wenn sich die Situation so ändert, daß die Handlung sehr wohl erfolgreich durchgeführt werden könnte. Um das zu vermeiden, muß diese Situationsänderung der betreffenden Person deutlich angezeigt werden.

2.6.3 Anwendung in der Softwaretechnik

Um wieder einmal die softwaretechnische Brille aufzusetzen: Aufgrund der individuellen Motivationsunterschiede eignen sich vermutlich verschiedene Personen unterschiedlich gut für verschiedene Aufgaben innerhalb der Softwaretechnik. Es empfiehlt sich, diese Eignung möglichst gezielt zu berücksichtigen. Auch einige andere Auswirkungen der verschiedenen Motivationsarten gilt es zu bedenken.

Neugier. So sind Personen mit hoher Neugier gut geeignet zur Entwicklung von Prototypen: in kurzer Zeit schnell viel Neues erschaffen. Auch das freudige Ausloten unterschiedlicher Entwurfsmöglichkeiten gehört zu ihren Stärken. Demgegenüber sind weniger neugierige Personen gut geeignet für Aufgaben, die eine hohe Sorgfalt und Ruhe erfordern. Beispiele wären das Testen oder die Erstellung von Dokumentation.

Für die meisten Aufgaben dürfte eine gute Mischung von Personen dieser beiden Typen die besten Ergebnisse bringen, weil sich dann die spezifischen Nachteile beider Typen am ehesten kompensieren lassen. Dies gilt vermutlich besonders für die Anforderungsanalyse, bei der sowohl Neugier als auch Sorgfalt in besonders hohem Maße benötigt wird.

Soziale Bedürfnisse. In der Anforderungsanalyse kommen Menschen mit hohen sozialen Bedürfnissen besonders effektiv zum Zuge. Sie verfügen meist auch über besser ausgebildete soziale Kompetenzen und sind deshalb besonders geeignet, gefährliche Fehler wegen aufgrund mangelnder Akzeptanz unzureichender Kommunikation vermeiden zu helfen, die in der Anforderungsanalyse besonders leicht vorkommen.

Ehrgeiz. Hoher Ehrgeiz ist natürlich in der Softwaretechnik grundsätzlich hilfreich, jedoch müssen die Belohnungen sorgfältig darauf abgestimmt werden, daß der Ehrgeiz nicht dem Gesamtunterfangen zuwiderläuft. Extrem gefährlich sind zum Beispiel Belohnungssysteme, die auf formalen Produktivitätsbegriffen basieren (wie z.B. abgelieferte Zeilen Code pro Tag) oder solche, die die Teamhaftigkeit von Softwareproduktion unterlaufen, indem sie z.B. das Hin- und Herschieben von Schuld an irgendwelchen Mängeln herausfordern.

Machtstreben. Wegen des Teamcharakters von Softwareproduktion ist das Machtstreben eine grundsätzlich hinderliche Motivationsart. Es gibt jedoch Möglichkeiten, das Machtstreben in ungefährliche, eventuell sogar nützliche Bahnen zu lenken. Die wichtigste dieser Möglichkeiten ist die Erhöhung der Kontrolle jedes einzelnen über seine unmittelbare Arbeitsumgebung (*ausgenommen* die Menschen). Eine hohe Kontrolle impliziert nämlich zugleich eine hohe Verantwortung und kann bei entsprechender Führung bewirken, daß weniger oft etwas Richtiges oder Notwendiges nur deshalb nicht getan wird, weil sich niemand zuständig fühlt.

Diese Fragen sind, wie auch sehr viele andere, im Rahmen der Softwaretechnik bislang praktisch überhaupt nicht erforscht worden, so daß die obigen Überlegungen lediglich eine plausible Spekulation darstellen. Forschungsfelder für psychologisch interessierte Softwaretechniker oder softwaretechnisch interessierte Psychologen gibt es in Hülle und Fülle.

2.7 Streß

2.7.1 Ursache und Wirkung

Entwicklungsgeschichtlich ist Streß eine körperliche Reaktion, die dazu dient, gefährliche Situationen besser zu bewältigen. Durch komplexe physiologische Maßnahmen wird der Körper in einen Kampf-oder-Flucht-Zustand gebracht, in dem die körperliche Leistungsfähigkeit stark erhöht ist. Die physiologischen Abläufe im Streßzustand sind zwar für den Körper schädlich, aber das macht nichts, solange Streßsituationen nur kurz andauern und nicht sehr häufig auftreten.

Der moderne Mensch ist jedoch mittlerweile oftmals Streßsituationen ausgesetzt, die lange anhalten und sehr häufig auftreten. Zu den daraus resultierenden langfristigen Gesundheitsschädigungen gehören hoher Blutdruck, erhöhte Neigung zu Diabetes, Herzgefäßverengung und Magengeschwüren und vermutlich sogar stärkere Gedächtnisschwäche im Alter.

2.7.2 Streßvermeidung

Das Maß, in dem eine averse Situation (ein „Stressor“) Streß auslöst, hängt von mehreren Faktoren ab: Der Situationswahrnehmung durch den Betroffenen, die emotionale Reaktionsbereitschaft, dem Maß, in dem Kontrolle über die Situation ausgeübt werden kann, und der Vorhersehbarkeit der Situation. Außerdem kann man natürlich versuchen, das Auftreten von Stressoren überhaupt zu reduzieren.

Alle fünf Faktoren lassen sich im Prinzip nutzen, um gezielt Streß zu vermindern. Die Wahrnehmung und Reaktionsbereitschaft sind Persönlichkeitseigenschaften, die durch geeignete Schulung entschärft werden können (wobei noch niemand genau weiß, wie man das am besten anfängt). Dagegen sind das Auftreten von Stressoren und das Maß verfügbarer Kontrolle organisatorische Randbedingungen, die sich in einer Softwareorganisation optimieren lassen. So entsteht zum Beispiel viel Streß, wenn eine Person ein Problem lösen soll, aber nicht über die Kompetenzen verfügt, alle dazu nötigen Maßnahmen auszuführen oder einzuleiten, z.B. wegen einer zu rigiden

Dokumentenverwaltung. Eine solche Organisation wird auf die Dauer je nach der Persönlichkeitsstruktur der Betroffenen entweder zu Streßkrankheiten führen oder zum Absinken der Motivation und zur Gleichgültigkeit — also Streßvermeidung durch Änderung der Reaktionsbereitschaft.

2.7.3 Umgang mit Streß

Es gibt zwei grundsätzlich verschiedene Arten des Umgangs mit Streß: problemorientiert oder emotionsorientiert.

Der problemorientierte Umgang mit Streß erfolgt durch Anwendung zumindest einiger der folgenden Schritte: Definition des Problems, Entwicklung mehrerer alternativer Lösungsideen, Abwägung zwischen diesen in Bezug auf Kosten und Nutzen, Auswahl einer Idee und Umsetzung einer Idee. Diese Strategie kann auch nach innen gerichtet sein, also das zu lösende Problem in einer erwünschten Persönlichkeits- oder Verhaltensänderung sehen, z.B. die Umlenkung des Ehrgeizes auf andere Ziele oder die Anhebung oder Absenkung gewisser Ansprüche an sich selbst. In Längsschnittstudien (d.h. Beobachtung über längere Zeit statt nur punktuell) wurde nachgewiesen, daß problemorientierter Umgang mit Streß zu kürzeren Phasen von Bedrückung führt.

Emotionsorientierter Umgang mit Streß versucht zu verhindern, von den Gefühlen überwältigt zu werden, die der Streß auslöst. Dafür gibt es eine ganze Reihe von Strategien, z.B.

1. Verhaltensstrategien sind alle, die Aktivitäten bewirken, z.B. Sport treiben, Drogenmißbrauch oder Hilfesuche bei Freunden.

2. Kognitive Strategien zeigen demgegenüber kaum äußerliche Aktivität. Hierzu gehören das bewußte Wegschieben des Problems („Es ist nicht wert, daß ich darüber nachdenke.“) oder eine Uminterpretation der Bedeutung („Eigentlich ist es ja gar nicht so schlimm.“)

3. Grüblerische Strategien bedeuten ein volles Eintauchen in das Problem mit intensivem Nachdenken oder Sprechen darüber, wie schlecht man sich fühlt und was noch alles weiterhin schlecht sein kann und wird.

4. Ablenkungsstrategien sind positive Verhaltensstrategien wie Sport treiben oder ins Kino gehen, die vom Problem wegführen.

5. Negative Vermeidungsstrategien sind hingegen vom Problem wegführende Strategien, die neue negative Folgen nach sich ziehen, z.B. Drogenmißbrauch, gefährliche Unternehmungen oder aggressives Verhalten.

Die Wahl zwischen problemorientiertem und emotionsorientiertem Umgang mit Streß ist überwiegend eine Frage der Persönlichkeit, kann aber auch von der jeweiligen Situation abhängen. So ist emotionsorientierter Umgang die einzige Lösung, wenn das Problem unbeeinflußbar ist, z.B. schon komplett abgeschlossen, und wird auch immer dann bevorzugt, wenn die Erfahrungen lehren, daß ein Versuch zur konkreten Problemlösung wahrscheinlich nicht erfolgreich sein wird.

Nicht nur, aber auch für den Bereich der Softwaretechnik gilt, daß problemorientierter Umgang mit Streß eine Fertigkeit ist, die man erlernen und üben kann und sollte — zumal die Vorgehensweise dabei dem Softwarelebenszyklus wirklich frappierend ähnlich sieht.

2.8 Sozialpsychologie

Die Sozialpsychologie befaßt sich mit den psychologischen Aspekten des Zusammenlebens von Menschen, im Gegensatz zur isolierten Betrachtung des Individuums.

2.8.1 Attribution

Der Prozeß, durch den wir uns die Ursachen des Verhaltens anderer Menschen erklären, heißt Attribution. Der Attribution liegt ein Vorgang zugrunde, der implizite Psychologie genannt wird: Unbewußt sammeln wir ständig Daten über das Verhalten anderer, interpretieren diese und konstruieren daraus Theorien. Die Methoden wie auch die Theorien sind privat und implizit und können oftmals selbst vom Betroffenen nicht erklärt werden. Tatsächlich werden die Theorien auch oftmals wie Fakten behandelt. Logischerweise tendieren Menschen dann dazu, die Theorien nicht umzuwerfen, selbst wenn sie mit Daten konfrontiert werden, die ihnen widersprechen.

Die wichtigste Unterscheidung zwischen verschiedenen Begründungen für das Verhalten einer Person A ist jene zwischen dispositiven und situativen Faktoren. **Situative Faktoren** sind Reize aus der jeweiligen aktuellen Umgebung von A, während **dispositive Faktoren** Persönlichkeitsmerkmale von A sind. Unsere Lebenserfahrung lehrt uns, welchen Einfluß welche situativen Faktoren üblicherweise auf das Handeln haben oder haben können und was für Arten von dispositiven Faktoren vorkommen. Direkte Bekanntschaft mit einer bestimmten Person läßt uns auch deren individuelle dispositive Faktoren einschätzen. Oftmals schließen wir dabei aus einer einzigen Beobachtung, die z.B. auf Geiz oder auf Aggressivität hindeutet.

Ob wir ein Verhalten eher mit situativen oder eher mit dispositiven Faktoren erklären, hängt an drei Eigenschaften dieses Verhaltens: Konsens, Konsistenz und Eigentümlichkeit.

1. Konsens: Ein Verhalten, das von sehr vielen Personen gezeigt wird, wird tendenziell situativen Faktoren zugeschrieben.

2. Konsistenz: Ein Verhalten, das bei einer Person verläßlich immer wieder auftritt (konsistent), wird eher dispositiven Faktoren zugeschrieben. Ändert es sich von Mal zu Mal (inkonsistent), wird es eher situativen Faktoren zugeschrieben.

3. Eigentümlichkeit: Ein Verhalten, das stets unter bestimmten Umständen auftritt, wird situativen Faktoren zugeschrieben.

Menschen machen bei der Attribution eine Reihe von systematischen Fehlern:

1. Fundamentaler Attributionsfehler: Wir schreiben beim Verhalten anderer fast stets einen zu großen Teil davon dispositiven Faktoren zu. Zum Beispiel schätzt nach einem Quizspiel, bei dem der Frager sich die Fragen beliebig ausdenken darf (und folglich der Befragte nur wenige Antworten weiß), der Befragte das Wissen des Fragers meist sehr viel höher ein als sein eigenes; es wird also ein dispositiver Faktor herangezogen. Der Einschätzungsunterschied verschwindet, wenn der Frager die Einschätzung vornimmt; der situative Faktor wird korrekt erkannt. Die Erklärung dieses Fehlers könnte sein, daß wir über die Gründe unseres eigenen Handelns so viel besser informiert sind als bei anderen, so daß wir leichter erkennen, wo überall situative Faktoren einen Einfluß haben, während sich das bei anderen Personen unserer Kenntnis entzieht.

2. Falscher Konsens: Wir neigen dazu, unsere Meinung oder Reaktion als typisch für eine große Mehrheit zu betrachten, auch wenn das gar nicht stimmt. Eine Erklärung hierfür lautet, daß wir nicht gern anders sein wollen als andere, eine weitere Erklärung ist, daß sich Menschen gern in der Gesellschaft anderer bewegen, die ihnen tatsächlich ähnlich in ihren Meinungen und Reaktionen sind.

3. Eigenerfolg und Fremdversagen: Wir neigen dazu, Erfolge unseren eigenen Handlungen zuzuschreiben, aber Mißerfolge auf äußere Faktoren zurückzuführen.

4. Relevanzvermutung: Wenn wir die Motivationsstruktur einer Person A so einschätzen, daß eine bestimmte Handlung für A eine hohe Relevanz hat, neigen wir eher dazu, A's Verhalten auf situative Faktoren zurückzuführen. So gilt eine höherrangige Person, wenn sie einem einen Gefallen tut, eher als freundlich (dispositiv), während eine niederrangige Person als eher gezwungen zu dem Gefallen angesehen wird (situativ).

5. Verursachungsillusion: Wir neigen dazu anzunehmen, daß unsere willentlichen Handlungen einen Einfluß auf Ereignisse der Zukunft haben, auch wo das gar nicht der Fall sein kann. So wurden Besitzer eines 50-Cents-Lotterieloses danach gefragt, für wieviel Geld sie ihr Los verkaufen würden. Personen, denen das Los vom Verkäufer in die Hand gedrückt worden war, wollten etwa 2 Dollar; Personen, die es selbst gezogen hatten, wollten etwa 9! Offensichtlich erzeugt das Auswählen des Loses die Illusion, ein Gewinn werde dadurch wahrscheinlicher. Selbst schon der Willensakt, das Los überhaupt zu kaufen, erzeugt diese Illusion, denn der Erwartungswert für den wirklichen Wert des Loses liegt natürlich unter dem Kaufpreis. Die gleiche Art von Fehlschluß tritt auch in sozialen Situationen auf. Ursache dieses Fehlers ist vermutlich eine falsche Interpretation unserer Lebenserfahrung, daß wir tatsächlich mit unseren Handlungen zukünftige Ereignisse beeinflussen können.

Die Folgerung aus all diesen Beobachtungen für die Softwaretechnik lautet, daß innerhalb eines Softwareteams für ausreichende Kommunikation gesorgt werden muß, um Konflikte und Mißverständnisse aufgrund von Attributionsfehlern zu vermeiden. Insbesondere die falsche Konsensvermutung kann zu Einbußen in der Effektivität der Arbeit einer Softwaregruppe führen und der Effekt von Eigenerfolg und Fremdversagen kann das Klima in einem Team vergiften.

2.8.2 Heuristiken im sozialen Schließen

Wir benutzen zum Ziehen von Schlüssen bei sozialen Interaktionen Heuristiken, die gelegentlich zu Schlußfehlern führen.

1. Die Repräsentativitätsheuristik. Wenn wir etwas Neues kennenlernen, vor allem aber bei Personen, suchen wir anhand unserer ersten Beobachtungen einen Stereotyp aus unseren früheren Erfahrungen, auf den die neue Person zu passen scheint. Wir schliessen dann (oft fälschlich), daß auch andere Eigenschaften des Stereotyps auf die Person zutreffen. Bestandteil solcher Fehlschlüsse ist oftmals der Basisratenfehler, der schon im Abschnitt über das induktive Schließen beschrieben wurde.

2. Die Verfügbarkeitsheuristik. Die relative Wichtigkeit oder Häufigkeit zweier Ereignisse wird vor allem dann falsch eingeschätzt, wenn es bei dem weniger häufigen besonders leicht gelingt, uns Exemplare davon ins Bewußtsein zu rufen. So überschätzen die meisten Menschen die Wahrscheinlichkeit eines Mordes oder eines Flugzeugabsturzes, weil sie in den Medien so überproportional gut mit Informationen darüber versorgt werden. Testpersonen schätzten z.B. die Anzahl englischer Wörter, die mit k beginnen, höher ein als die

Anzahl englischer Wörter, die ein k als dritten Buchstaben haben. Letztere fallen einem nicht so leicht ein, sind aber doppelt so häufig. Das Versagen der normalerweise nützlichen Verfügbarkeitsheuristik passiert nicht nur aufgrund eines allgemeinen Erfahrungshintergrundes, sondern oft auch situationsabhängig: Wenn wir mit einer Sache gerade vor kurzem Kontakt hatten, ist sie uns geistig leicht zugänglich und wird als wichtiger oder häufiger eingestuft als ihr zukäme. Aus diesem Grund haben persönliche Begegnungen mit Menschen, die ihre Meinung äußern, so einen großen Einfluß auf unsere Entscheidungen: Das Erlebnis der persönlichen Begegnung ist intensiv und ihr Inhalt deshalb besonders leicht verfügbar.

3. Der Regressionsfehlschluß: Auf ein Lob nach einer extrem guten Leistung folgt meistens eine schwächere Leistung, während auf Kritik nach einer extrem schlechten Leistung meistens eine bessere Leistung folgt. Kann man daraus schließen, daß Lob demotiviert und Kritik motiviert? Man kann, und wir tun das auch dauernd — aber es ist falsch! In Wirklichkeit sind beide Ereignisse, die Verbesserung als auch die Verschlechterung, schlicht und einfach sehr wahrscheinlich, denn eine extrem gute oder schlechte Leistung ist so weit vom Mittelwert entfernt, daß fast immer die nächste Leistung näher am Mittelwert liegen wird. Dieses „Zurückgehen“ zum Mittelwert heißt Regression, daher der Name dieser Falle. Der Regressionsfehlschluß ist eine spezielle Form des Versagens der Verfügbarkeitsheuristik.

In der Softwaretechnik machen Repräsentativitätsheuristik und Verfügbarkeitsheuristik nicht nur in sozialer Hinsicht, sondern auch in technischen Situationen (z.B. Fehlersuche bei Entwurfsprüfungen) häufig große Probleme. Ein ständiges klares Bewußtsein für diese Effekte ist ein wichtiges Werkzeug zur Leistungsverbesserung in der Softwaretechnik. Probleme können auf der technischen Ebene am besten dadurch umgangen werden, daß möglichst viel aufgrund objektiver Daten anstatt subjektiver Eindrücke und Erinnerungen gehandelt wird.

2.8.3 Haltungen

Eine Haltung zu einem Thema T setzt sich aus drei Komponenten zusammen: Der affektiven Komponente, die die Gefühle beschreibt, die in Zusammenhang mit T auftauchen, der Verhaltenskomponente, die die Handlungen beschreibt, zu denen wir im Zusammenhang mit T neigen und schließlich die kognitive Komponente, die aus den Annahmen besteht, die wir über T haben.

Die affektive Komponente bildet sich durch Konditionierung; entweder direkte Konditionierung mittels positiver oder negativer Erlebnisse in Zusammenhang mit T oder sekundäre Konditionierung durch unterschwelliges Wahrnehmen und Übernehmen der affektiven Haltung anderer Personen (insbesondere von Autoritätspersonen). Die affektive Komponente widersteht Änderungen stärker als die anderen beiden. Ein Auseinanderklaffen zwischen einer alten negativen affektiven Haltung und einer nicht mehr negativen kognitiven Haltung zu einem Thema erzeugt oft Schuldgefühle.

Die kognitive Komponente wird meist von anderen übernommen, nur selten selbst gebildet — und auch dann nur zu einfachen Themen vollständig. Im allgemeinen geschieht die Übernahme durch Nachahmung von Vorbildern oder anderen Autoritäten.

Man sollte erwarten, daß affektive und kognitive Komponente unser Verhalten weitgehend bestimmen. Dies ist aber oft nicht der Fall. Folgende Gründe stehen dagegen:

1. Allgemeinheitsgrad: Die meisten Haltungen sind recht allgemein. Handlungen dagegen sind stets äußerst spezifisch.

2. Relevanz für Motive: Eine Haltung zu haben (oder auszudrücken) kostet erheblich weniger Mühe, als eine entsprechende Handlungsweise an den Tag zu legen. Deshalb führen selbst ernsthafte Haltungen meist nicht zu den entsprechenden Handlungen, wenn diese Handlungen eine geringe persönliche Bedeutung für die Person haben.

3. Selbstattribution: Haltungen sagen Handlungen nur dann gut voraus, wenn sie aufgrund eigener Erfahrungen gebildet wurden — oft sind sie aber von anderen Menschen übernommen: wir erklären zwar unsere eigenen Absichten mit ihnen, neigen aber dazu, diesen dann im Handeln gar nicht zu folgen.

4. Verhaltensbeschränkungen: Äußere Umstände machen es häufig schwer oder unmöglich, die zu einer Haltung passende Handlung auszuführen. Besteht eine solche Situation über längere Zeit, wird sich die Haltung entweder verstärken oder abschwächen.

Es sind aber nicht nur unsere Haltungen Grund für unsere Handlungen, sondern auch umgekehrt ändern Handlungen die Haltungen. Wenn Haltungen und Handlungen auseinanderklaffen, empfindet man eine kognitive Dissonanz. Um diese zu reduzieren, gibt es drei Möglichkeiten: Senkung der Wichtigkeit eines dissonanten Elements, Zufügen konsonanter Elemente oder Beseitigung eines dissonanten Elements.

So kann ein Student, der sich für sehr intelligent hält, aber dauernd schlechte Noten bekommt, diese Dissonanz beseitigen, indem er sich sagt, Noten seien ohnehin unwichtig und hätten wenig mit Intelligenz zu tun (Senkung der Wichtigkeit), beschließen, daß äußere Umstände verantwortlich sind, z.B. ungerechte Notenvergabe oder zu große Belastung durch den Nebenjob (konsonante Elemente zufügen) oder er kann bessere Noten bekommen oder seine Meinung über seine Intelligenz revidieren (dissonantes Element beseitigen). Bis auf das Bessere-Noten-Bekommen sind dies alles Beispiele für Haltungsänderungen aufgrund von Handlungen.

Auch andere Situationen führen zu Haltungsänderungen. Allgemein passen wir unsere Haltungen und deren Gewichtung so an, daß sie wenig Diskrepanz mit der Situation ergeben, in der wir sind. So halten wir etwas für besser und mehr wert, wenn wir mehr dafür bezahlt haben oder mehr Mühe aufgewendet haben, um es zu erlangen. Oder wir legen uns künstlich ein positives Urteil über eine Sache zu, mit der wir gezwungenermaßen verbunden sind, um unsere Selbstachtung zu wahren. Oder wir erhöhen die positive Beurteilung von A, nachdem wir uns in einer Konfliktsituation zwischen A und B für A entschieden haben.

Eine besonders wichtige Klasse von Haltungen sind Vorurteile. Sie entstehen vor allem durch die zu großzügige Anwendung von Stereotypen in Verbindung mit selektiver Wahrnehmung, die durch ein bereits vorhandenes Vorurteil noch verstärkt wird: Es wird nur noch das wahrgenommen, was in das Vorurteil paßt. Anderes wird ignoriert oder uminterpretiert. Vorurteile sind kaum zu vermeiden, dürfen jedoch nie zu ausgeprägt werden, damit sie nicht negative Auswirkungen entfalten. Eine für die Softwaretechnik wichtige Klasse solcher Auswirkungen ist die selbsterfüllende Prophezeihung: Eine falsche oder zumindest voreilige Einschätzung einer Situation erzeugt ein Verhalten, daß dazu führt, daß die Einschätzung richtig wird.

Selbsterfüllende Prophezeihungen sind mächtige soziale Faktoren und können im positiven wie im negativen wirken. Hier ein eindrucksvolles Beispiel für eine positive Wirkung:

Männlichen Versuchspersonen, die mit weiblichen Versuchspersonen zu telefonieren hatten, wurde vor Beginn des Telefonats ein Photo der angeblichen Gesprächspartnerin gezeigt. Unabhängig von der tatsächlichen Partnerin bekamen manche Männer ein Bild einer attraktiven Frau, andere ein Bild einer unattraktiven Frau gezeigt. Die Gespräche wurden aufgezeichnet und die weibliche Stimme später weiteren Versuchspersonen vorgespielt. Diese Personen empfanden die Stimme als freundlicher, sympathischer und geselliger, wenn es sich um ein Gespräch handelte, vor dem ein attraktives Photo gezeigt worden war (und zwar im Blindtest, d.h. ohne zu wissen, welcher dieser Fälle gerade vorlag), als bei Gesprächen, vor denen ein unattraktives Photo gezeigt worden war. Die ungerechtfertigte Prophezeihung des männlichen Telefonierers, eine attraktive oder unattraktive Partnerin zu haben, hatte sich ohne Wissen dieser Partnerin selbst erfüllt!

Offensichtlich ist eine Vermeidung von negativen und ggf. der gezielte Einsatz von positiven selbsterfüllenden Prophezeihungen ein wichtiges Augenmerk für softwaretechnische Arbeitsgruppen.

2.8.4 Soziale Einflüsse

Unser Verhalten wird natürlich auch direkt von anderen beeinflußt, oftmals in unbewußter oder unerwünschter Weise.

Konformität ist ein Wesenszug des Menschen, der das Zusammenleben enorm erleichtert. Konformität ist der Wunsch, nicht aus dem normalen Rahmen herauszufallen. Selbst Menschen, die scheinbar sehr wenig konform sind, z.B. Punks, sind in Wirklichkeit in den allermeisten Verhaltensbereichen konform. Und selbst in den Bereichen ihrer Nonkonformität stellt sich oftmals wieder Konformität auf einer anderen Ebene ein. So bilden z.B. die Punks eine Gruppe, die wiederum ihre eigenen Konformitätsregeln hat; ein Punk mit einer netten, immer gutgekämmten Allerweltsfrisur ist nur schwer vorstellbar.

Konformität ist in der Softwaretechnik dadurch schädlich, daß sie die Wahrnehmung verschlechtert: Es ist schwer, sich einer Meinung, die von mehreren anderen geäußert wurde, nicht anzuschließen, selbst wenn man eigentlich glaubt, daß sie falsch ist.

Soziale Förderung: Wenn jemand anderes an den eigenen Taten teilnimmt, z.B. einfach zuschaut, steigt unsere Anregung. Wir geben uns dann mehr Mühe. Bei einfachen Arbeiten führt das zu höherer Leistung, bei komplexen Arbeiten jedoch eventuell zu einem effektiven Absinken der Leistung, trotz des Bemühens.

Soziales Faulenzen: Auch der gegenteilige Effekt sozialer Anteilnahme ist möglich. Als Mitglied einer Gruppe bin ich weniger motiviert, Leistung zu erbringen, als als Einzelperson, zumindest dann, wenn meine Leistung nicht als Einzelleistung aus der Gruppenleistung zu identifizieren ist.

Diese beiden Effekte der sozialen Förderung und des Faulenzens sind offenbar delikate Balancierungsaufgaben für alle, die eine softwaretechnische Arbeitsgruppe zu organisieren und zu führen haben.

Reziprozität: Menschen mit intaktem Sozialverhalten haben einen starken Wunsch, einen Gefallen, der ihnen angetan wurde, zurückzugeben. Dies ermöglicht eine weitere Form von Motivationssteigerung durch Erhöhung der Kontrolle über die Umwelt: Wenn ein Mitglied einer Arbeitsgruppe einem anderen Hilfestellung nicht etwa aufgrund einer festen Arbeitsvorschrift leistet, sondern im Einzelfall aufgrund eigenen Entschlusses, so ist das andere Mitglied besser motiviert, später eine andere Hilfe zurückzuleisten.

Vor-Engagement: Wenn man einmal einer Sache zugestimmt hat, sich für etwas engagiert hat oder einer Person entgegengekommen ist, wird es schwierig, sich bei einer nachgefragten Verstärkung dieser Zustimmung, des Engagements oder des Entgegenkommens dann zurückzuziehen. In der sozialen Interaktion ist dieser Effekt bekannt unter dem Satz „Es ist schwer, Nein zu sagen, wenn man einmal Ja gesagt hat.“. In der Softwaretechnik wirkt sich der gleiche Effekt vor allem bei der Korrektur größerer Fehler aus: Man hat große Widerstände dagegen, einen schlechten, aber schon abgeschlossenen Entwurf noch einmal umzuwerfen oder ein Modul, das sich als sehr fehlerhaft herausgestellt hat, komplett wegzuwerfen und neu zu implementieren etc., selbst wenn das das eindeutig klügere Verhalten wäre.

2.9 Nachtrag: Psychologische Schulen

Als kleiner Anhang für Interessierte hier noch ein sehr kurzer und vereinfachter Überblick über die wichtigsten psychologischen Schulen.

2.9.1 Strukturalismus

Als Begründer der modernen Psychologie gilt gemeinhin der Deutsche Wilhelm Wundt. Er gründete 1879 das erste ausdrücklich so genannte psychologische Labor und zwar an der Universität Leipzig. Sein Denkansatz stammt aus den damals schnellen Fortschritten zum Beispiel der Chemie: Wenn die Chemie Fortschritte machte, indem sie Stoffe in ihre Elemente zerlegte, zum Beispiel Wasser in Wasserstoff und Sauerstoff, vielleicht konnte man dann Fortschritte in der Analyse des Denkens machen, indem man Wahrnehmungen (Perzeptionen) in ihre Elemente (Sensationen) zerlegte, zum Beispiel den Geschmack von Limonade in die Teile süß, bitter, kalt und so fort?

Der Schwerpunkt dieser Schule lag auf der Analyse von Wahrnehmungen, mit dem Ziel, ihre Struktur zu ermitteln — daher der Name. Die Untersuchungsmethode der Wahl im Strukturalismus war die Introspektion, das heißt die Aufklärung der Tatbestände durch Nachdenken. Diese der Philosophie entlehnte Methode wurde jedoch nicht in Reinform benutzt, sondern durch einfache Experimente mit menschlichen Subjekten erweitert; die Introspektion diente dann zum Erklären der experimentellen Beobachtungen.

2.9.2 Funktionalismus

Vorreiter des Funktionalismus war gegen Ende des neunzehnten Jahrhunderts ein Psychologe von Harvard namens William James. Die Funktionalisten waren gegen das Schwergewicht des Strukturalismus auf der statischen Analyse von Bewußtsein. Sie wollten stattdessen mehr ein Verständnis für seinen fließenden Charakter entwickeln, Prozesse untersuchen. Der Forschungsschwerpunkt lag auf der Frage, wie das Bewußtsein funktioniert, damit sich Individuen an ihre Umgebung anpassen können. Über den Begriff der Gewohnheit wurde hierbei erstmals das Lernen zu einem zentralen Begriff der Psychologie. Das große Interesse an der Anpassungsfähigkeit wurde vor allem durch die Arbeiten von Charles Darwin angeregt. Die Untersuchungsmethoden wurden verbreitert auf Untersuchungen zum Verhalten von Tieren und zur Entwicklung von Verhalten. Dennoch war die Introspektion immer noch das Hauptmittel der psychologischen Forschung.

2.9.3 Behaviorismus, S-R–Psychologie

Etwa ab 1920 setzte sich rapide eine radikal neue Sicht der psychologischen Forschung durch: Die Behavioristen, also rein verhaltensorientierte Forscher, proklamierten eine Psychologie völlig ohne Introspektion. Ihr Vorreiter war John B. Watson, der 1913 ein behavioristisches Manifest veröffentlichte. Hauptmerkmal behavioristischer Psychologie ist es, keinerlei Annahmen über Struktur oder Funktionsweise des Geistes zu machen, sondern ausschließlich die Beobachtungen aus Experimenten zu sammeln und zu kategorisieren. Nur mit einer solchen Herangehensweise könne man, so die Behavioristen, einem echten wissenschaftlichen Anspruch gerecht werden, weil der Vorgang der Introspektion keiner Nachuntersuchung durch andere zugänglich sei. Wissenschaftliche Daten müssen aber eine öffentliche Inspektion zulassen.

Motor der behavioristischen Forschung waren zunächst vor allem die Arbeiten zur Konditionierung von Iwan Pawlow. Die konditionierte Reaktion wurde gewissermaßen als das Atom von Verhalten angesehen: Kompliziertere Verhaltensweisen können als aus konditionierten Reaktionen zusammengesetzt betrachtet werden und die Umwelt formt ständig unser Verhalten durch Bestärkung gewisser Gewohnheiten — nichts anderes als Konditionierung. Der vielleicht berühmteste Vertreter dieser Richtung ist Burrhus Frederick Skinner.

Diese Mechanismen wurden in Strukturen von Reiz und Reaktion dargestellt, was zu der Bezeichnung „Reiz-Reaktions-Psychologie“ (stimulus response psychology, S-R psychology) geführt hat. Die späteren Reiz-Reaktions-Psychologen gingen über die frühen Behavioristen insofern hinaus, daß sie bereit waren, auch Annahmen über Prozesse zu machen, die zwischen Reiz und Reaktion ablaufen. Damit entwickelte sich aus der Reiz-Reaktions-Psychologie eine Sprache, die geeignet ist, unabhängig von einer bestimmten Verhaltenstheorie psychologische Aussagen explizit zu formulieren und kommunizierbar zu machen. Diese Sprache ist bis heute vorherrschend in der Psychologie der meisten Schulen.

2.9.4 Gestaltpsychologie

Etwa zeitgleich mit der behavioristischen Revolution der Psychologie, die in Amerika ihren Anfang nahm, gab es eine weitere, die ebenfalls radikal dem Strukturalismus und Funktionalismus eine Absage erteilte, allerdings auf eine völlig andere Weise: 1912 proklamierte der Deutsche Max Wertheimer die Gestaltpsychologie. Die zentrale Annahme dieser Schule lautet, daß einem aus Teilen zusammengesetzen Bewußtseinsprozeß eine Qualität zukommt, die die Summe der Einzelteile nicht hat — eben die Gestalt.

Ursprünglich befaßte sich die Gestaltpsychologie vor allem mit Wahrnehmungseffekten, vor allem dem Phi-Phänomen: Zwei Lichter, die nahe beieinander angeordnet sind und kurz hintereinander kurz aufblinken, werden als ein Licht wahrgenommen, das sich vom Ort des ersten zum Ort des zweiten bewegt. Die Gestaltpsychologie schließt aus diesem Effekt, daß unsere Erfahrungen von den Mustern abhängen, die von Reizen gebildet werden und davon, wie unsere Erfahrungen *organisiert* sind. Das Ganze besteht nicht nur aus Teilen, sondern aus Teilen und deren Beziehung zueinander.

Die Gestaltpsychologen lehnten die Introspektion als Hauptmethode ebenso ab wie die Behavioristen, bekämpften die Behavioristen jedoch trotzdem entschieden: Die Gestaltpsychologie verlangte eine als „Phänomenologie“ bezeichnete Form der Erkundung, die es unternahm, Menschen nach ihren subjektiven Wahrnehmungen und ihren Urteilen zu befragen und diese Antworten als Daten der Forschung zu behandeln.

Wegen dieser Vorgehensweise ist gestaltpsychologische Forschung überwiegend qualitativ, im Gegensatz zur quantitativen des Behaviorismus. Aus diesem Grund ist die Gestaltpsychologie oft als vage und unwissenschaftlich abgelehnt worden. Dennoch hat die Gestaltpsychologie durch ihre Betrachtungsweise wichtige Anstöße für viele heute aktuelle Erkenntnisse und Forschungsthemen geliefert.

2.9.5 Psychoanalyse

Aus der von Sigmund Freud zu Beginn des Jahrhunderts begründeten Psychoanalyse kann man hier am besten eine Hauptbetrachtungsweise herausgreifen, nämlich den Begriff des *Unbewußten*. Laut Freuds Theorie des Unbewußten werden in der Kindheit alle unerlaubten oder unakzeptablen Wünsche aus dem Bewußtsein verdrängt und werden Teil des Unbewußten. Das Unbewußte versucht dann zeitlebens, sich zu äußern, was sich in verschiedener Weise niederschlägt, zum Beispiel in Form von Träumen, Versprechern oder kleinen, kaum bewußten Angewohnheiten. Freuds ursprünglich fast ausschließliche Anwendung seiner Theorien auf sexuelle Bereiche hat die Annahme der richtigen Anteile seiner Theorien sehr erschwert. Heute bilden diese Anteile jedoch einen festen Bestandteil des psychologischen Wissens.

2.9.6 Informationsverarbeitungssysteme, Psycholinguistik

Mit dem Aufkommen der Computer in den 1950er Jahren hielt eine neue Sichtweise Einzug in die Psychologie: Viele Aspekte des Bewußtseins lassen sich als algorithmisch informationsverarbeitendes System interpretieren. Unter einer solchen Betrachtungsweise kann man den Computer einsetzen, um dieses System zu simulieren, so daß sich Theorien über die Struktur von Denkprozessen leichter überprüfen lassen. Entsprechende Arbeiten fanden in den 50er und 60er Jahren im interdisziplinären Forschungsbereich der künstlichen Intelligenz statt. In diesen Kontext lassen sich zum Teil auch Arbeiten an Schachprogrammen oder automatischen Theorembeweisern einordnen. Es ist mittlerweile offensichtlich, daß die Situation leider doch nicht ganz so einfach ist, wie viele damals annahmen, denn alle bislang gebauten computergestützten informationsverarbeitenden Systeme sind dermaßen viel primitiver als fast jeder einzelne Aspekt irgendwelcher Denkprozesse im Gehirn, daß sich kaum beurteilen läßt, ob sie zumindest einen essentiellen Teil davon korrekt modellieren. Zumindest hat aber die Informationsverarbeitungsperspektive dafür gesorgt, daß die Psychologie heute mit viel reicheren Beschreibungsmitteln für die inneren Prozesse des Bewußtseins operiert, als es nach der Einführung der S-R–Psychologie der Fall war.

Ein besonders wichtiger Teil dieser Forschung waren die Arbeiten von Chomsky und Nachfolgern. Chomskys 1957 erschienenes Buch *Syntactic Structures* lieferte die wesentliche Basis für eine aktive Zusammenarbeit von Psychologen und Linguisten für die ersten signifikanten psychologischen Analysen von Sprache, die Psycholinguistik.

2.9.7 Neuropsychologie

Eine Gegenbewegung zum Ansatz der Psychologie informationsverarbeitender Systeme entstand gleichzeitig aus Forschungen auf dem Gebiet der Neurophysiologie, die klar den Zusammenhang gewisser physiologischer Prozesse auf der Ebene einzelner Nervenzellen im Gehirn mit Denkvorgängen belegen. Somit setzt die an Symbolen und ihrer Manipulation orientierte Betrachtungsweise zu hoch an, denn einzelne Nervenzellen können nur erheblich primitivere Signale verarbeiten.

Aus der Möglichkeit physiologischer Erforschung ergibt sich eine Sichtweise der Psychologie, die besagt, daß für ein vollständiges Verständnis des Denkens auch die inneren Prozesse im Gehirn geklärt werden müssen und geklärt werden können und die behavioristische Reiz-Reaktions-Betrachtung nicht ausreicht. Entsprechende Forschungen sind seit einigen Jahren in vollem Gange.

In der ingenieurmäßig orientierten, also nützliche Anwendungen anstrebenden Informatik hat dieser Gegensatz zu einem Streit zwischen der sogenannten symbolischen KI, die den informationsverarbeitenden Ansatz vertritt, und dem sogenannten Konnektionismus, der die neuropsychologische Richtung vertritt, geführt. Dieser Streit hielt viele Jahre an und wird erst in den letzten Jahren allmählich durch die Erkenntnis beigelegt, daß die erfolgreichsten Anwendungen meist Elemente beider Ansätze verbinden; beispielsweise regelbasierte Systeme *und* Neuronale Netze.

2.9.8 Kognitive Psychologie

Seit Ende der 1970er Jahre ist eine Arbeitsrichtung entstanden, die sich *kognitive Psychologie* nennt und eine Abteilung der interdisziplinären *Kognitionswissenschaften* ist. Hierin werden alle psychologischen Teilschulen zusammengeführt, die nützliche Beiträge zum Verständnis zu liefern haben: Einerseits von „oben" her behavioristische Experimentalbeobachtungen ebenso wie strukturalistisch, funktionalistisch, gestaltpsychologisch oder informationsverarbeitungstheoretisch gebildete Hypothesen über die Struktur, die Organisation und den Ablauf von Wahrnehmungs- oder Denkprozessen. Andererseits von „unten" her Experimentaldaten aus der Neurophysiologie, die unmittelbar die Arbeitsweise des Gehirns (als Organ) beschreiben. Die kognitive Psychologie versucht nun, alle diese Daten in Modellen zu bestimmten Aspekten der Psychologie zu vereinigen. Diese Modelle werden in der Art von informationsverarbeitenden Systemen auf verschiedenen Abstraktionsebenen formuliert und können zum Teil durch Simulation auf Computern überprüft und verfeinert werden. Umgekehrt können solche Simulationen auch Anstöße liefern für gezielte Forschungen in den empirischen Bereichen.

Durch dieses Aufsaugen und Vereinigen aller Forschungsrichtungen ist den kognitiven Wissenschaften die beste Chance einzuräumen, uns langfristig ein korrektes und umfassendes Bild von der Funktionsweise des Gehirns im Sinne der Psychologie zu verschaffen.

2.10 Weitergehende Information

Eine sehr schöne Einführung in die Psychologie, in der alle obigen und auch noch eine Reihe weiterer Themen abgedeckt werden, ist [Car93]. Recht ähnlich und ähnlich gut ist [AASB93]. Beide Bücher sind sehr schön aufbereitet, gut lesbar und enthalten viele ausführliche Beispiele. In [SS92] findet man einen ausführlichen Abriß der Geschichte der Psychologie.

http://matia.stanford.edu/cogsci.html ist ein guter WWW-Einstiegspunkt für Informationen über psychologische Themen.

Abschnitt 3

Softwaretechnik-Forschung

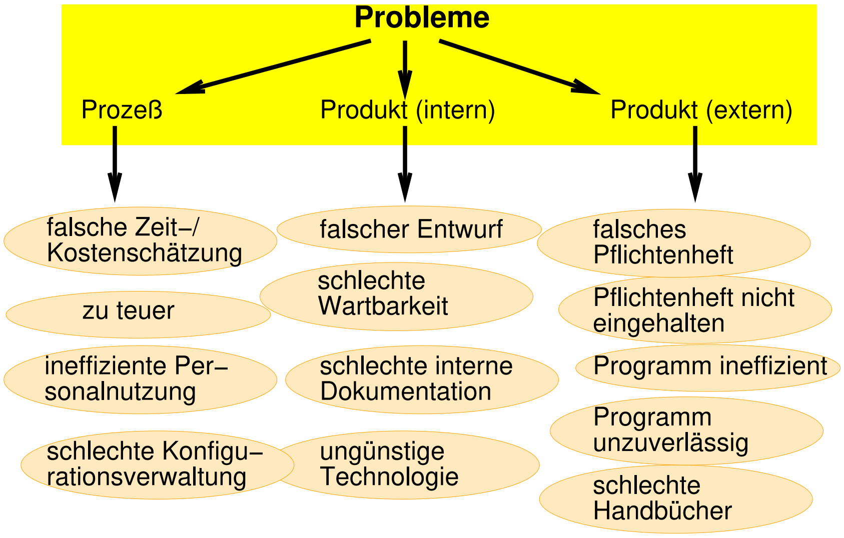

Abbildung 3.1: Von der Softwaretechnik-Forschung angesprochene Problembereiche.

3.1 Einordnung und Ziele

Die Softwaretechnik, wie auch die meisten anderen Teile der Informatik, kann man als eine Ingenieurdisziplin betrachten: Sie ist damit befaßt, wie man kostengünstig nützliche Artefakte (in diesem Fall Softwaresysteme oder öfter noch die Softwareteile von größeren Systemen) produziert, deren Funktionstüchtigkeit sich während der Konstruktion gezielt sicherstellen läßt. Es stellt sich die Frage, welchen Platz Forschung (also Wissenschaft) in diesem Zusammenhang einnimmt.

3.1.1 Ingenieurwesen oder Wissenschaft?

Fred Brooks [Bro96] sieht die Informatik (engl.: Computer Science) nicht als Wissenschaft, denn sie ist nicht mit der Produktion von Wissen als ihrem Haupt*zweck* befaßt, sondern mit der Herstellung von Sachen:

> The scientist *builds in order to study*, the engineer *studies in order to build*. I submit that by any reasonable criterion the discipline we call „computer science“ is in fact not a science, but a *synthetic*, an engineering, discipline. We are concerned with making things, be they computers, algorithms, or software systems.

Aus diesem Blickwinkel bezeichnet er die Informatiker als „Werkzeugmacher“:

> In contrast with many engineers who make houses, cars, medicines, and clothing for human need and enjoyment, we make things that do not themselves directly satisfy human needs, but which others use in making things that enrich human living. In a word, the computer scientist is a toolsmith — no more, but no less. It is an honorable calling.
> If we perceive our role alright, we then see more clearly the proper criterion for success: a toolmaker succeeds as, and only as, the *users* of his tools succeed with his aid.

Brooks sieht mehrere Gefahren darin, daß sich die Informatiker als Wissenschaftler betrachten:

1. Wissenschaft sieht die Entdeckung von Fakten und Gesetzen als Selbstzweck an. Das führt bei der Informatik zu einer Flut von neuen Computern, Algorithmen und Sprachen, deren Neuigheit (weil sie Entwürfe sind) jedoch keinen Wert an sich hat; sie sollten an ihren Kosten und ihrer Nützlichkeit gemessen werden.

2. Wir neigen dazu, unsere Benutzer und ihre wirklichen Probleme zu vergessen, um uns in den Elfenbeinturm zurückzuziehen und uns mit handhabbaren Abstraktionen von echten Problemen zu beschäftigen — Abstraktionen, die das Wesen des ursprünglichen Problems oftmals verloren haben.

3. Die Hochschätzung der Wissenschaft (im Gegensatz zum Ingenieurwesen) führt dazu, daß brilliante Geister von der Lösung herausfordernder praktischer Probleme abgezogen werden, so daß diese Probleme nicht mit der Stärke angepackt werden können, die ihrer Bedeutung zukäme.

Dieser Analyse ist zum Teil zuzustimmen. Es ist richtig, daß die Informatik Ingenieurcharakter hat und den *Bau* von Dingen anstrebt. Auch der Betrachtung von Informatikprodukten als Werkzeugen kann man überwiegend zustimmen. Es ist ebenfalls richtig, wie wir unten noch sehen werden, daß die Informatik, vor allem die Softwaretechnik, recht ausgeprägt die Probleme hat, die Brooks als Folgen ihrer Betrachtung als Wissenschaft aufführt.

Dennoch muß man Brooks' Sicht in einem entscheidenden Punkt widersprechen: Ingenieurcharakter und Wissenschaftscharakter schließen einander für ein und dasselbe Gesamtfach durchaus nicht aus. Die Beobachtung “The scientist builds in order to study, the engineer studies in order to build” ist zwar brilliant und treffend, gilt aber (sogar ihrer sprachlichen Form nach!) eben nicht für ein Fach als ganzes, sondern nur für einzelne Personen, die es betreiben — und auch da nicht unbedingt lebenslang.

Natürlich ist die Informatik ein Ingenieurfach und die meisten, die es ausüben, sind Ingenieure. Das heißt aber nicht, daß die Informatik nicht auch eine Wissenschaft wäre. Weil der überwiegende Teil der Informatik zweckgerichtet ist und somit Ingenieurcharakter hat, spricht Brooks dem ganzen Fach den Wissenschaftscharakter ab.

Ingenieurwesen (engl. engineering) ist laut Webster's New Encyclopedic Dictionary von 1993

> The application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to human beings.[1]

so daß sich für den Ingenieuranteil der Informatik sofort die Frage stellt, welche Wissenschaft denn das sein könnte, die da zum Wohle der Menschen angewendet wird. Die Mathematik allein ist sicherlich nicht ausreichend, selbst wenn sie nicht ohnehin als Zusatz in jedem Fall gefordert wäre. Die einzig logische Anwort auf diese Frage lautet, daß die Informatik selbst einen wissenschaftlichen Teil hat, der das Wissen produziert, welches die Informatikingenieure dann anwenden.

Der Grund für diese Zweiteilung ist in den Randbedingungen praktischer Ingenieurarbeiten zu suchen: Im Mittelpunkt der Arbeit steht im Ingenieurwesen das konkrete Lösen eines praktischen Problems zu möglichst geringen Kosten. Deshalb bleibt meist keine Zeit, dabei zugleich die Grundlagen dieser Arbeit, nämlich das wissenschaftliche Wissen über das Fach, grundsätzlich zu verbessern. Eine Wissenschaft hat hingegen nur die Produktion und Organisation von Wissen zum Ziel und kann, ja muß, sich genau mit diesen Grundlagen befassen, anstatt nur ein einzelnes Projekt im Auge zu haben.

Für eine Wissenschaft ist es unerheblich, ob die Produktion von Wissen überwiegend Selbstzweck ist, wie beispielsweise die Klassifikation von Pflanzen in der Biologie, oder stark zweckgerichtet wie in den meisten Teilen der Informatik. Bei Betrachtung der Wissenschaftsgeschichte stellt man ohnehin fest, daß fast sämtliche Wissenschaften in weiten Teilen zweckgerichtet waren und sind. Deshalb spricht überhaupt nichts dagegen, die Informatik als Wissenschaft anzusehen, auch wenn sie zugleich als Ingenieurfach ausgeübt wird, im Gegenteil: Zu jedem Ingenieurfach muß es eine zugehörige Wissenschaft geben, damit das Ingenieurwesen größere Fortschritte machen kann.

Alle obigen Überlegungen gelten nicht nur für die Informatik im ganzen, sondern auch insbesondere für die Softwaretechnik. Bei der Softwaretechnik tritt allerdings besonders der Aspekt der Interdisziplinarität mit in den Vordergrund: Da überall der Mensch so eng eingebunden ist, bekommen Überschneidungen mit der Psychologie, Soziologie und anderen Fächern, die sich mit dem Menschen befassen, eine verglichen mit anderen Teilen der Informatik besonders hohe Bedeutung.

Jedenfalls hat die Softwaretechnik-Forschung die Aufgabe, wissenschaftliche Ergebnisse zu liefern, die den Wissensschatz vergrößern, auf den Softwareingenieure zur Lösung ihrer Probleme zurückgreifen. Da die Softwaretechnik eine Ingenieurwissenschaft ist, werden die Ergebnisse an ihrer Nützlichkeit gemessen. Im Einzelfall kann die Abgrenzung zwischen einer wissenschaftlichen Studie und einem Teilstück eines ingenieurmäßigen Problemlösungsvorgangs durchaus schwierig sein.

3.1.2 Methoden zum Erkenntnisgewinn

Hier geht es darum, wie man Wissen über seine Umwelt (hier: Softwaretechnik) erwirbt. Die Menschheit verwendet zum Wissenserwerb eine Reihe verschiedener Ansätze und Techniken, die wir im folgenden kurz vorstellen wollen.

Festhalten: Die einfachste Methode zum Wissenserwerb besteht darin, einfach *irgendetwas* festzuhalten und als Wissen zu betrachten — unabhängig davon, ob es korrekt ist oder nicht. Die

[1] Offensichtlich hat die Informatik, die ja hauptsächlich mit Abstraktem umgeht, noch keinen Eingang in diese Definition gefunden. Eine modernere Definition sollte sich vielleicht lieber zurückziehen auf „The application of science and mathematics for the creation of artifacts that are useful for human beings."

wichtigsten Beispiele für dieses Vorgehen sind Aberglauben und Gewohnheiten. Bei Aberglauben werden Annahmen über Sachverhalte ohne Begründung als richtig angesehen und an dieser Meinung unabhängig von Bestätigung festgehalten; eventuell wird die Wahrnehmung verändert, um Bestätigungen zu finden. Bei Gewohnheiten wird an einer Verhaltensweise festgehalten, wobei auch dies zumindest teilweise unabhängig von der Zweckmäßigkeit geschieht. Die Herkunft der Gewohnheit kann begründet oder zufällig sein. Festhalten hat zwei Nachteile: Sehr oft ist das erworbene Wissen falsch und es gibt keine Mechanismen, um es zu korrigieren.

Intuition: Erwerb durch Intuition ist verwandt mit dem Festhalten, jedoch kommt das Wissen hier nicht aus völlig beliebigen Quellen, sondern durch spontane Eingebung zustande — allerdings ohne eine Herleitung oder logisches Schließen. Die Nachteile sind die gleichen wie beim Festhalten.

Autorität: Hier wird eine Aussage deshalb als Wissen angenommen, weil sie aus einer geachteten Quelle stammt. Dies ist nicht zu verwechseln mit der Akzeptanz von Aussagen geachteter Quellen aus inhaltlichen Gründen. Auch diese Methode hat die oben erwähnten Nachteile, ist aber im Gegensatz zu Festhalten und Intuition auch heute noch selbst in eigentlich gebildeten Kreisen sehr häufig.

Rationalismus: Der rationalistische Standpunkt besagt, daß Wissen nur durch reines Schließen erlangt werden soll. Er nimmt an, daß das Wissen korrekt sei, wenn die Schlußweisen korrekt verwendet werden. Problematisch an diesem Standpunkt ist, daß er den Ausgangspunkten des Schließens zu wenig Aufmerksamkeit widmet. Wenn diese Ausgangspunkte nicht ausreichend auf Beobachtungen gegründet sind, können verschiedene Rationalisten trotz korrekter Schlußmethoden zu verschiedenen Ergebnissen gelangen. Dieses Problem ist in der Geschichte der Theologie und der Philosophie in vielfacher Ausführung zu besichtigen. Reiner Rationalismus ist nur für solche Wissensbereiche akzeptabel, die nicht in der Realität gründen. Dies trifft im wesentlichen nur für die reine Mathematik zu.

Empirizismus: Dies ist gewissermaßen die Gegenposition zum Rationalismus. Der empirizistische Standpunkt besagt, daß ich als gültiges Wissen alles das und nur das akzeptiere, was ich selbst erfahren kann. Diese Haltung hat zwei Nachteile: Erstens kann man damit Widersprüche in den Beobachtungen verschiedener Personen über den gleichen Sachverhalt nicht ausräumen, und bekanntlich ist Wahrnehmung etwas subjektives und unzuverlässiges. Zweitens beschränkt der empirizistische Standpunkt den Bereich des Wissens auf das unmittelbar erfahrbare. Theorien über den Ursprung des Universums oder die Farbwahrnehmung von Fröschen zählten demnach nicht zum Wissen.

Wissenschaftliche Methode: Die wissenschaftliche Methode ist der beste bekannte Prozeß zum Wissenserwerb insofern, daß sie den Wissensbereich wenig beschränkt, den Erwerb falschen Wissens weitestgehend vermeidet und die Korrektur solcher Fehler direkt unterstützt. Die wissenschaftliche Methode besteht darin, daß zunächst eine klare, eindeutig beantwortbare Frage formuliert wird und dann eine Hypothese über die Antwort aufgestellt (auf Basis früherer Erfahrungen, insbesondere wissenschaftlicher Untersuchungen). Anschließend wird ein Experiment entworfen und durchgeführt, das dazu dient, die Korrektheit der Hypothese zu prüfen. Die Auswertung der Beobachtungen aus dem Experiment ergibt ein Stück Wissen, das abschließend über eine wissenschaftliche Veröffentlichung dem Rest der Welt bekanntgemacht wird. Der letztere Schritt bewirkt, daß die wissenschaftliche Methode einen kontinuierlichen Fortschritt des Wissensstandes ermöglicht, weil spätere Hypothesen und Experimente auf den Erkenntnissen früherer aufbauen können.

3.1.3 Wissenschaft in der Softwaretechnik

Die Softwaretechnik als praktische technische Disziplin hat heute wenig wissenschaftlichen Charakter. Das vorhandene Wissen stammt überwiegend aus einer Mischung von Intuition und Empirizismus und ist nur in wenigen Fällen wissenschaftlich abgesichert, denn die meisten Beiträge sind Entwürfe (von Modellen, Methoden oder Werkzeugen) und stellen insofern nur Hypothesen dar — und selbst diese werden meist nicht klar ausgesprochen. Geprüft werden diese Hypothesen bislang nur selten; nicht zuletzt deshalb, weil eine solche Prüfung in der Softwaretechnik meist enorm aufwendig ist.

So wissen wir beispielsweise herzlich wenig darüber, welche Eigenschaften von z.B. Entwurfsmethoden, Programmiersprachen oder Entwicklungswerkzeugen welche Qualitätsattribute der Produkte in welcher Weise beeinflussen, obwohl diese Fragen den Kern unseres Faches betreffen. Wir wissen auch fast nichts darüber, welche Fehler in welchen Situationen gemacht werden und welche Maßnahmen das verhindern könnten, oder wie man die Fehler zumindest schnell wieder findet und behebt.

Als Folge dieses Zustands arbeiten vermutlich die allermeisten softwareerzeugenden Organisationen weit unterhalb ihrer Möglichkeiten, was ihre Produktivität und die Qualität ihrer Produkte angeht.

Eine entscheidende Beobachtung in diesem Zusammenhang lautet, daß die grundsätzliche Arbeitsweise der wissenschaftlichen Methode in abgeschwächter Form auch für sehr praktische Problemstellungen in der Anwendung der Softwaretechnik verwendet werden könnte und sollte. Zum Beispiel kann die Auswahl des besseren von zwei Werkzeugen in der Regel durch ein entsprechend gestaltetes Experiment erledigt werden, wenn zuvor eine klare Fragestellung vorliegt (Was heißt „besser“? In welchem Zusammenhang?) und etwas Aufwand investiert wird. Dieser Aufwand könnte sich in vielen Fällen leicht amortisieren, und dennoch werden solche Untersuchungen bisher selten gemacht.

3.2 Ansätze

3.2.1 Wissensarten: Modell, Methode, Werkzeug

Es gibt drei Wissensarten, in denen wissenschaftliche Beiträge zur Softwaretechnik zu erbringen sind:

1. **Modelle**. Ein Modell ist eine vereinfachte abstrakte Repräsentation eines Teils der wirklichen Welt. Der Zweck von Modellen besteht darin, das Verständnis des modellierten Sachverhalts zu verbessern und insbesondere Vorhersagen über sein Verhalten zu treffen. Ferner sind Modelle das Hauptmittel zur Organisation einzelner Wissensteile in ein verständliches Ganzes. Modelle sind deshalb in allen Wissenschaften Bindeglied oder sogar Grundlage aller weiteren Beiträge.

2. **Methoden**. Eine Methode ist eine allgemeine Anleitung zum Lösen einer Klasse von Problemen. Im Gegensatz zu einem Algorithmus muß eine Methode das Vorgehen nicht unbedingt prozedural und auch nicht vollständig beschreiben. Methoden bilden gewissermaßen das Herz der Softwaretechnik, denn ihre zentrale Frage „Wie erstelle ich am besten Software?“ ist eine Methodenfrage.

3. **Werkzeuge**. Die praktische Nützlichkeit von Methoden hängt ganz erheblich vom Aufwand ihrer Anwendung ab. Werkzeuge können diesen Aufwand erheblich reduzieren. Sie stellen dann eigene Beiträge zum Wissen dar, wenn entweder bekannt ist, wie oder warum ein bestimmtes Werkzeug nützlich ist (weil dann vom einzelnen Werkzeug abstrahiert werden kann, was zur Modellbildung beiträgt) oder wenn damit erstmalig die praktische Anwendbarkeit einer annerkannterweise im Prinzip nützlichen Methode hergestellt wird.

Das ist alles. Mehr gibt's nicht.

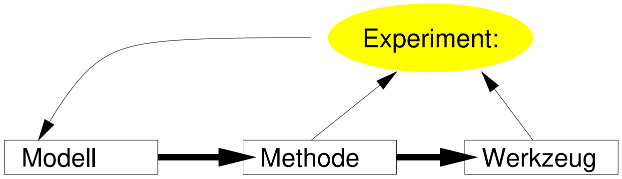

Abbildung 3.2: Die drei Wissensarten in der Softwaretechnik und ihre Beziehungen.

In der Softwaretechnik ist beispielsweise das Phasenmodell und seine Abarten die Kommunikations- und Verständnisgrundlage für zahlreiche Effekte bei der Entwicklung von Softwaresystemen. Es prägt eine einheitliche Grundterminologie (z.B für die Bezeichnungen und Aufgaben der Phasen), ordnet Wissenselemente an (anhand der Disjunktheit der Phasen) und ist die Grundlage für Vorhersagen, z.B. des Zeitbedarfs. Sprachen sind häufig ebenfalls Modelle, nämlich von bevorzugten Problembeschreibungsmethoden, z.B. imperativ oder deklarativ.

Aus einem anderen Blickwinkel ist das Phasenmodell eine Methode. Es ist eine Anleitung, die beschreibt, was wann zu tun ist. Sprachen (samt ihrer Übersetzer), die im Prinzip Modelle sind, kann man in ähnlicher Betrachtung als Werkzeuge ansehen; maschinelle Hilfen zur Bearbeitung und Lösung von Problemen. Eine solche Doppelsicht kommt oft vor, weil sich hinter jedem Werkzeug und jeder Methode irgendwo ein Modell versteckt.

3.2.2 Arten von Beiträgen

Ein wissenschaftlicher Beitrag in der Softwaretechnik hat immer einen der folgenden Typen (oder eine Mischung von mehreren):

1. **Existenzbeweis**. Oftmals ist für ein bestimmtes Problem überhaupt noch keine Lösung bekannt. Dann ist eine praktikable Lösung, die gefunden wird, in jedem Fall ein wissenschaftlicher Beitrag, weil sie beweist, daß das Problem sinnvoll lösbar ist. Dies ist zunächst weitgehend unabhängig von der Nützlichkeit dieser Lösung.
   Beispiel: Wenn das erste Mal eine komplette mathematisch-formale Spezifikation einer Programmiersprache vorgelegt wird, beweist das, daß Programmiersprachen nicht unbedingt Eigenschaften haben müssen, die eine vollständige formale Definition ausschließen. Dies ist ein wissenschaftlicher Beitrag auch dann, wenn diese Spezifikation weniger nützlich sein sollte als eine „normale" oder die Nützlichkeit ungeklärt bleibt.

2. **Offensichtliche Verbesserung**. Manchmal kann ein alternativer Ansatz zur Lösung desselben Problems so offensichtlich besser sein als ein bisheriger, daß sein reines Vorstellen, ähnlich wie beim Existenzbeweis, ein wissenschaftlicher Beitrag ist.
   Beispiel: Die Erfindung des Volltexteditors als Verbesserung des Zeileneditors.

3. **Theorem**. Eine nützliche theoretische Aussage ist ein wissenschaftlicher Beitrag, wenn diese Aussage mathematisch bewiesen werden kann.
   Beispiel: Berechenbarkeitstheorie. „Versuche nicht, ein Werkzeug zu bauen, das automatisch *alle* Endlosschleifen identifizieren kann, denn so etwas ist unmöglich.“

4. **Validierter Entwurf**. In der überwiegenden Zahl der Fälle haben Beiträge in der Informatik die Form eines Entwurfs: ein Algorithmus, eine Programmiersprache, ein Werkzeug, eine Entwurfsmethode, ein Modell etc. Solche Entwurfsbeiträge sind in beliebiger Zahl mit geringem Aufwand herzustellen. Sie können deshalb nur dann als wissenschaftliche Beiträge (also Beiträge zum Wissen) gewertet werden, wenn ihre Nützlichkeit nachgewiesen wird: eine Programmiersprache muß ausdrucksstark sein, Programmiersprachen, Werkzeuge und Methoden müssen Produktivität oder Qualität erhöhen, Modelle oder Theorien müssen realistische Aussagen machen. Häufig wird dieser Nachweis die Form haben, daß eine Eigenschaft für den Entwurf behauptet und ihr Vorhandensein dann nachgeprüft wird (Hypothesentest). Den höchsten wissenschaftlichen Wert haben dabei Modelle, weil sie am meisten Wissen liefern bzw. repräsentieren, gefolgt vom Methoden, gefolgt von Werkzeugen. Beachte, daß meist mehrere Komponenten gemischt einen Beitrag bilden. Hypothesentests können auch losgelöst von einem Neuentwurf stattfinden (also pure Validation).

5. **Eigenschaftsbeweis**. Oft können nicht alle behaupteten Eigenschaften eines Entwurfs in der ersten Arbeit nachgewiesen werden oder es werden erst später zusätzliche Eigenschaften entdeckt. Eine Art von Forschungsbeiträgen besteht deshalb darin, eine Validation einer Eigenschaftsbehauptung zu einem schon vorher bekannten Entwurf zu liefern.
   Beispiel: Bei Verwendung von Methode X entdecken gut geschulte Programmierer meist mehr als die Hälfte aller ihrer Fehler schon vor dem Test.

6. **Vergleich**. Wenn alle Beiträge eine der obigen Formen hätten, stünden viele Ideen zur Lösung der gleichen Probleme nebeneinander, ohne daß man ihre relative Nützlichkeit beurteilen könnte. Vergleiche zwischen Modellen, Methoden oder Werkzeugen schließen diese Lücke. Solche Vergleiche können auf empirischer oder (seltener) auf theoretischer Basis erfolgen.
   Beispiel: In welchem Programmierparadigma machen Programmierer die wenigsten Fehler (rein prozedural, prozedural objektorientiert, funktional) und was für Fehler sind das jeweils?
   Vergleiche sind neben dem Hypothesentest bei validierten Entwürfen die wichtigste Grundlage zur Formulierung neuer Modelle und Theorien und nehmen insofern eine Schlüsselstellung im Wissenschaftsprozeß ein.

Beiträge, die zu keinem dieser Typen gehören, sind keine wissenschaftlichen Beiträge in dem Sinne, daß sie unmittelbar gesichertes Wissen liefern. Sie können eventuell dennoch nützliche Beiträge sein, indem sie den Wissenschaftsprozeß anregen. Häufiger jedoch findet man leider Forschungsarbeiten, die weder einen nennenswerten direkten noch einen indirekten Beitrag zum Wissen liefern.

3.2.3 Forschungsmethoden

Zur Erbringung eines wissenschaftlichen Beitrags in der Softwaretechnik gibt es nur wenige Klassen von Forschungsmethoden, aus denen geschöpft werden kann:

1. **Entwurf und Bau**. Entwurf und Bau eines Softwaresystems oder der Entwurf eines Modells oder einer Methode stellen gelegentlich einen eigenen wissenschaftlichen Beitrag

dar, im Sinne eines Existenzbeweises oder einer offensichtlichen Verbesserung. In der großen Mehrzahl der Fälle jedoch sind diese beiden Forschungsmethoden nur Teilmethoden, da der hergestellte oder implementierte Entwurf noch validiert werden muß.

2. **Mathematischer Beweis**. Mathematische Beweise sind die Hauptmethode für die Beitragsart Theorem, sind aber auch oftmals als Hilfsmethode an anderen Stellen nützlich.

3. **Kontrolliertes Experiment**. In einem kontrollierten Experiment wird eine klar formulierte und sehr enge Forschungsfrage untersucht und, sofern das Experiment richtig entworfen und durchgeführt wird, auch recht klar beantwortet. Die Bezeichnung „kontrolliert“ stammt daher, daß hier versucht wird, für alle relevanten Faktoren wohldefinierte Verhältnisse herzustellen, was die Voraussetzung dafür ist, eine zuverlässige Aussage zu bekommen. Dies wird entweder durch Festhalten des Faktors auf einem bestimmten Niveau erreicht oder durch Randomisierung (Zufallsauswahl) über eine Menge (Grundgesamtheit) von Individuen, die betrachtet werden sollen. Letztere Methode ist fast immer nötig, wenn Menschen im Spiel sind, weil deren Eigenschaften nicht genau genug gemessen, geschweige denn fixiert werden können. Über kontrollierte Experimente folgt weiter unten ausführliche Information.

4. **Fallstudie**. Eine komplette Validation eines Entwurfs ist in sehr vielen Fällen außerordentlich schwierig zu bewerkstelligen. Denken sie beispielsweise an die Aussage „Das Spiralmodell halbiert die Gefahr von Fehlentwürfen oder über 30prozentigen Terminüberschreitungen gegenüber dem Phasenmodell (Wasserfallmodell) bei Projekten von mehr als 10 Personenjahren Umfang“. Um eine solche Aussage komplett zu validieren müßte man also eine größere Zahl von solchen großen Projekten (vielleicht etwa 10) zweimal durchführen, je einmal mit dem Spiralmodell und mit dem Phasenmodell und zwar jeweils mit Teams vergleichbarer Qualifikation für das Projekt. Dies ist offensichtlich unbezahlbar. In derartigen Situation nimmt man Zuflucht zu Fallstudien als Validationsersatz. In Fallstudien wird eine kleine Zahl von Versuchen mit dem Entwurf ausgewertet. Um die geringe Zahl auszugleichen, werden im Gegensatz zu anderen empirischen Methoden auch die nichtobjektiven Beobachtungen in die Ergebnisse einbezogen und vorsichtig in Bezug auf die zu untersuchenden Aussagen hin interpretiert.

5. **Feldbeobachtung**. Eine Zwischenposition zwischen Experimenten und Fallstudien nehmen Beobachtungen im Feld ein, also Beobachtungen an realen Entwicklungsprojekten. Anders als bei Fallstudien liegt hier oft eine genügend große Zahl von Beobachtungen vor, so wie es bei Experimenten ebenfalls nötig ist, um die Effekte der Randomisierung auszugleichen. Im Unterschied zu Experimenten ist aber bei Feldbeobachtungen keine Kontrolle über die Umgebungsfaktoren gegeben, so daß die Beobachtungen von Effekten anderer Variablen als der zu beobachtenden „verschmutzt“ sind. Wenn man abschätzen kann, welche Variablen dies sind, stellen aber auch Feldbeobachtungen wertvolle Forschungsbeiträge dar.

3.2.4 Forschungsfelder

Die derzeit bearbeiteten Forschungsfragen lassen sich ungefähr wie folgt in Felder innerhalb der Softwaretechnik einordnen. Die Felder sind natürlich nicht disjunkt und anders gestaltete Einteilungen könnten ebenso sinnvoll sein. Die Auflistung soll nur eine ungefähre Übersicht über die Themen vermitteln, die derzeit in der Softwaretechnik-Forschung von Bedeutung sind. Für die meisten der Fragen existiert ein gewisses Teilwissen, aber bei kaum einer gibt es Antworten, die auch nur entfernt als endgültig angesehen werden sollten.

- **Anforderungsanalyse.** Wie kommuniziert man mit dem Anwender? Wie setzt man vage Anforderungen (Benutzerfreundlichkeit, Sicherheit) in konkrete um? Wie versteht man Prioritäten und Zufriedenheitskriterien des Anwenders? Wie stellt man Eindeutigkeit und Vollständigkeit sicher? Wie verteilt man Aufgaben zwischen System und Anwender?
  Querbezüge zu Spezifikation, Entwurf, Dokumentenverwaltung.

- **Spezifikation.** Wie integriert man verschiedene Sichten und Darstellungen? Welche Darstellungen sind effizient und fehlerarm handhabbar? Warum? Welche erlauben die Verfolgung von Anforderungen? Welche sind anwendertauglich? Warum? Wie prüft man Eindeutigkeit und Vollständigkeit?
  Querbezüge zu Anforderungsanalyse, Entwurf, Dokumentenverwaltung.

- **Entwurf.** Wie evaluiert man verschiedene Entwurfsansätze? Wie findet man einen guten Entwurf in Bezug auf Fehler, Implementationskosten, Implementationsgeschwindigkeit, Produkttestbarkeit, Produktwartbarkeit? Wie prüft man diese Qualität? Welche wiederkehrenden Muster in Entwürfen lassen sich entdecken? Warum? Welche Formalismen sind zur Darstellung von Entwürfen günstig? Warum? Wie dokumentiert man Anforderungen und Entwurfsentscheidungen?
  Querbezüge zu allen anderen Feldern.

- **Implementation.** Welche Sprachen haben die höchste Produktivität, die geringste Fehlerhäufigkeit? Warum? Welche Programmierstile optimieren Effizienz, Verständlichkeit, Erweiterbarkeit, Fehlerhäufigkeit? Warum? Wie unterstützt man Verfolgung von Entwurfsentscheidungen, Anforderungen?
  Querbezüge zu Entwurf, Test, Reengineering.

- **Test.** Welche Testmethoden sind am effektivsten? Am effizientesten? Warum? Wie kann man das Testen automatisieren? Wie kann man Testaufbauten wiederverwenden? Wie kann man die SW-Zuverlässigkeit schätzen? Wie kann man das Fehlerlokalisieren beschleunigen?
  Querbezüge zu Entwurf, Implementation, Wiederverwendung.

- **Reengineering.** Wie deckt man Anforderungen, Entwurfsentscheidungen, Entwurfsstrukturen in Programmen auf, deren Anforderungs- und Entwurfsdokumente verloren oder veraltet sind? Wie stellt man die möglichen Auswirkungen einer Änderung an einer Stelle fest? Wie findet man die Stelle(n), wo man ein großes Programm ändern muß, um einen gewissen Zweck zu erreichen?
  Querbezüge zu Dokumentenverwaltung, Entwurf, Implementation.

- **Dokumentenverwaltung.** Wie bewahrt man Bezüge zwischen Dokumenten gleicher oder verschiedener Phasen? Wie verwaltet man Konfigurationen (Revisionen, Varianten, Familien)? Wie geht man mit unvollständiger Kontrolle über Dokumente um? Wie unterstützt man örtlich verteilte Softwareentwicklung?
  Querbezüge zu allen anderen Feldern.

- **Wiederverwendung.** Wie entwirft, validiert, katalogisiert man wiederverwendbare Software? Wie findet man eine Komponente? Wie entwirft, produziert, testet man mit wiederverwendbaren Komponenten? Wie wirkt sich Wiederverwendung auf Qualität, Kosten, Produktivität aus? Warum? Wie verwendet man Artefakte früher Phasen wieder (Anforderungen, Entwürfe)?
  Querbezüge zu allen anderen Feldern.

- **Softwareprozeß.** Wie sollte eine SW-Organisation arbeiten? Wieviel Vorschriften braucht man? Welche? Warum? Wie kontrolliert man ihre Einhaltung? Wie optimiert man das

Vorgehen allmählich aufgrund von Beobachtung? Wie wirkt sich das auf die Arbeit des einzelnen aus? Wie verhindert man unerwünschte Nebenwirkungen?
Querbezüge zu allen anderen Feldern.

3.3 Probleme und Hindernisse

Die experimentelle Forschung in der Softwaretechnik leidet an einer Reihe von Problemen, die nicht auf die Unfähigkeit der Forscher und Forscherinnen oder auf eine falsche Ausrichtung der Forschungslandschaft zurückzuführen, sondern grundsätzlicher Natur sind. Alle diese Probleme lassen sich auf zwei Grundprobleme zurückführen, nämlich hohen Aufwand für Experimente und eine große Zahl relevanter Variablen:

Hoher Aufwand: Ein gutes und wissenschaftlich aussagekräftiges Experiment in der Softwaretechnik ist in den meisten Fällen enorm teuer. Das hat mehrere Gründe.

- Menschen: Fast alle Forschungsfragen betreffen das Verhalten von Menschen und verlangen nach der Mitwirkung von Versuchspersonen am Experiment. Das ist stets teuer, insbesondere, weil die Schwankungen im menschlichen Verhalten verlangen, daß der Experimentaufbau nicht nur einmal durchlaufen wird, sondern viele Male, um dann durch statistische Analyse die Schwankungen herausrechnen zu können.

- Gruppen: Die meisten wichtigen Fragestellungen betreffen nicht das Verhalten einer einzelnen Person, sondern von Personengruppen. Das erhöht die Anzahl nötiger Versuchspersonen und die Zwänge bei der Versuchsdurchführung nochmals. Schlimmer noch, die Gruppen sollten idealerweise meistens eingespielte Teams sein und nicht beliebig zusammengestellt.

- Qualifikation: Von den Versuchspersonen wird ferner eine sehr hohe Qualifikation verlangt, da sich die Untersuchungen auf anspruchsvolle, professionelle Arbeitsgänge beziehen. Das führt häufig zu Schwierigkeiten bei der Interpretation der Resultate, wenn als Versuchspersonen nur Studierende zur Verfügung stehen, denn es ist nicht klar, in welcher Weise ihre Leistungen von denen im relevanten professionellen Kontext abweichen. Experimente mit Profis sind jedoch noch um ein Vielfaches teurer.

- Dauer: Die betrachteten Arbeitsgänge sind fast immer sehr komplex und dauern eine lange Zeit. Das reduziert die Möglichkeiten zur Definition schlanker Versuche, die in geringer Zeit durchgeführt werden können. Häufig müßten im Prinzip Arbeiten verrichtet werden, die mehrere Arbeitswochen in Anspruch nehmen. Auch dies ist selten finanzierbar.

Viele Variablen: Im Vergleich zu Experimenten in anderen Wissenschaften oder anderen Gebieten der Informatik müssen in der Softwaretechnik erheblich mehr zu kontrollierende Variablen beachtet werden, um ein echtes kontrolliertes Experiment zu erhalten. Diese Variablen liegen in folgenden Sektoren:

- Versuchspersonen: Ausbildung, Fähigkeiten, Erfahrungshintergrund, alles jeweils sowohl allgemein-softwaretechnisch als auch speziell in Bezug auf die vorliegende Aufgabe.

- Aufgabenstellung: Größe, Anwendungsgebiet, Komplexitätsgrad, besondere Schwierigkeiten.

- Randbedingungen: Terminanforderungen, Qualitätsanforderungen, Wahl von Methoden, soweit sie nichts mit der Forschungsfrage zu tun haben.
- Arbeitsumfeld: Art und Umfang der maschinellen Unterstützung, Art und Umfang der organisatorischen Unterstützung.

Das erschwert zwar nicht die Versuchsdurchführung, schränkt aber die Verallgemeinerbarkeit der Resultate (externe Gültigkeit) sehr ein.

Diese beiden Grundprobleme zusammengenommen führen dazu, daß schon ein einzelner wissenschaftlicher „Datenpunkt" teuer ist, aber viele davon zum Aufbau einer sinnvollen Theorie über einen Teilbereich benötigt werden. Jedes Experiment liefert nur einen solchen Datenpunkt, also eine Beobachtung über das Verhalten eines Softwaretechnik-Systems. Für jede der relevanten Variablen brauchen wir jedoch mehrere Beobachtungen, um ihren Einfluß abschätzen zu können. Da wir auch die Wechselwirkungen zwischen den Variablen nicht kennen, aber davon ausgehen müssen, daß sie relevant sind, gibt es eine kombinatorische Explosion in der Zahl der benötigten Experimente zum Abtasten eines Raums von mehreren Variablen.

Machen wir mal eine kurze Überschlagsrechnung anhand der obigen Auflistung relevanter Variablen: Nehmen wir an, wir bräuchten für eine bestimmte Fragestellung (sagen wir mal, den Vergleich zweier Entwurfsmethoden) zum ausreichenden Abtasten des Einflusses aller versuchspersonenbezogenen Variablen nur 3 Klassen von Versuchspersonen (mäßig kompetent, kompentent, sehr kompetent). Nehmen wir ferner an wir brauchen 3 Schwierigkeitsgrade der Aufgabenstellung (einfach, mittel, schwierig) und 8 Anwendungsgebiete, 2 Stufen von Qualitätsanforderungen (normal oder hoch), 2 Arten von Terminplänen (normal oder schnell) und 8 Kombinationen sonstiger Methoden; schließlich nur 3 verschiedene Arbeitsumfelder. Dann kommen wir durch Bildung des Kreuzproduktes aller dieser Variablenräume auf $3 \cdot 3 \cdot 8 \cdot 2 \cdot 2 \cdot 8 \cdot 3 = 6912$ Versuche nur zur kompletten Klärung der Frage, unter welchen Umständen eine Entwurfsmethode A besser ist als eine andere Entwurfsmethode B!

Der einzige Ausweg aus diesem Problem ist das Beschneiden dieses Suchraums während der Suche aufgrund der Zwischenergebnisse, denn da wir ingenieurmäßige Anwendungen unseres Wissens im Sinn haben, interessiert uns ja nicht der gesamte Raum, sondern nur seine effizienten Teilbereiche. Wenn wir also im Beispiel nach einer kleinen Zahl von Experimenten in weit auseinanderliegenden Teilbereichen des Variablenraumes erkennen, daß Methode A wohl nur für sehr simple Probleme besser ist als B, dann können wir die ganze restliche Versuchsreiche einsparen. Für ein solches Vorgehen dürfen dann nicht nur die nackten Experimentergebnisse herangezogen werden, sondern wir müssen auch die detaillierteren (teilweise subjektiven) Beobachtungen bei der Experimentdurchführung berücksichtigen. Auch in diesem Punkt läuft alles mal wieder darauf hinaus, daß die Anwendung des gesunden Menschenverstandes durch nichts zu ersetzen ist.

3.4 Experimentelles Arbeiten

3.4.1 Formulierung der Forschungsfrage

Am Anfang jeder Forschung steht die Festlegung des Themas, der Forschungsfrage. Dies ist ein entscheidender Schritt, denn Fehler, die hier gemacht werden, lassen sich in späteren Phasen nicht mehr ausbügeln (ganz ähnlich wie Fehler bei der Anforderungsbestimmung für Softwaresysteme). Folgendes sind die Teilschritte.

Identifikation eines Problems: Zuerst einmal muß festgelegt werden, welches Problem oder Problemfeld überhaupt bearbeitet werden soll. Die Auswahl richtet sich einerseits nach Nützlichkeits- und Machbarkeitsüberlegungen, andererseits nach den Interessen des jeweiligen Forschers oder der Forscherin. Man sollte nur an Themen arbeiten, die einen auch interessieren, weil sonst die nötige Begeisterung fehlt.

Literaturstudie: Liegt das Thema im groben fest, so sollte in der Literatur nachgeforscht werden, was darüber schon bekannt ist. Eine solche Literaturstudie ist heutzutage recht aufwendig, weil der Umfang der möglicherweise relevanten Literatur enorm ist. Andererseits wird sie allmählich durch umfangreiche Datenbanken wieder erleichtert. Die Literaturstudie sollte ergeben, an welchen Teilaspekten des Themas bereits gearbeitet wurde, welche Ergebnisse dabei erzielt wurden und was diese für die Ausrichtung der eigenen Arbeit bedeuten, wo die Schwierigkeiten liegen und welche Aspekte am lohnendsten sein dürften. In der Softwaretechnik gibt es zu den meisten Fragen bereits Fallstudien, aus denen sich wichtige Hinweise auf sinnvollen Experimentaufbau ermitteln lassen, immer aber zumindest anekdotenhafte Berichte (anectodal evidence) über Phänomene. Der Nachteil von letzteren besteht darin, daß sie meist recht schwierig zu finden sind.
Einen zusätzlichen Weg zur Informationsbeschaffung über den Stand des Wissens eröffnen die modernen Medien: Mailinglisten und eventuell Newsgroups können eine hervorragende Quelle von Kommentaren und Anregungen zu halbfertigen Forschungsideen sein.

Machbarkeitsprüfung: Auf Basis der Literaturstudie sollte das Thema nun soweit eingeengt werden, daß es eine Frage betrifft, die einerseits von erheblichem Interesse für die Wissenschaft ist und andererseits eine effektive experimentelle Untersuchung unter den gegebenen Rahmenbedingungen gestattet. Im Bereich der Softwaretechnik ergibt dieser Schritt leider häufig, daß keine der interessanten Fragen praktisch experimentell untersuchbar scheint. In diesem Fall ist oft einfach eine weitere Einengung angezeigt — oft geht es aber auch wirklich nicht.
An dieser Stelle ist auch die grundsätzliche Frage zu entscheiden, ob die Untersuchung qualitativ sein soll, also die Prüfung einer booleschen Hypothese, oder quantitativ, also die Bestimmung des Werts eines Parameters. Letzteres ist nur dann möglich, wenn das grundsätzliche Verhalten dieses Parameters bereits bekannt ist. Ein Beispiel für eine qualitative Forschungsfrage wäre „Verbessern zusätzliche Inspektionen die Zuverlässigkeit von SW bei gleichem Aufwand gegenüber reinem Test oder nicht?“. Die zugehörige quantitative Frage lautet vielleicht „Um wieviel verbessern zusätzliche Inspektionen die Zuverlässigkeit von SW bei gleichem Aufwand gegenüber reinem Test?“. Die Klärung letzterer Frage erfordert einen erheblich aufwendigeren Versuch und sollte deshalb nur angegangen werden, wenn man ziemlich sicher ist, daß das Ergebnis den Aufwand lohnt.

Formulierung: Nun muß die Forschungsfrage forschungsreif formuliert werden. Dazu ist sie einerseits zu präzisieren und andererseits zu operationalisieren. Präzisierung bedeutet etwa im obigen Beispiel die Festlegung, was man unter Inspektionen genau verstehen will, welches Testverfahren als Maßstab dient und woran die Zuverlässigkeit gemessen werden soll. Operationalisieren bedeutet festzulegen, welche Manipulationen es sind, mit denen man die betrachteten Faktoren ändern bzw. messen will. Im Beispiel stellt sich etwa die Frage, wie die Messung der Zuverlässigkeit praktisch zu bewerkstelligen ist. Ergebnis der Operationalisierung ist eine genaue Beschreibung der unabhängigen Variablen (also was manipuliert werden soll) und der abhängigen Variablen (also was beobachtet werden soll). In den meisten Fällen sollte nur eine unabhängige Variable verwendet werden, weil das Experiment sonst zu umfangreich wird. Die Zahl der abhängigen Variablen ist eventuell etwas größer, weil es mehrere Maße gibt, mit denen der zu beobachtende Effekt charakterisiert werden kann.

In der Realität laufen die 4 Schritte der Problemidentifikation, Literaturstudie, Machbarkeitsprüfung und Formulierung nicht hintereinander ab, sondern sind in einem wüsten Knäuel auf eine Weise miteinander verquickt, die selbst die beteiligten Forscher meist nicht verstehen. Das macht aber nichts, solange am Ende eine sinnvolle und konkrete Forschungsaufgabe steht.

3.4.2 Kontrolle von Variablen

Das Ziel eines Experiments besteht in einer gesicherten Aussage der folgenden Form:

> Wenn man in der Situation S die Änderung X an der Variablen x vornimmt, dann kann man als Wirkung die Änderung Y an der Variablen y beobachten.

Unter welchen Umständen kann man eine solche Aussage als gesichert betrachten? Nun, dazu muß sichergestellt sein, daß die Beobachtung Y tatsächlich kausal auf die Änderung X zurückzuführen ist.
Unter welchen Umständen weiß man das? **Nur dann, wenn X das einzige ist, was sich ändert.** Denn dann und nur dann ist jede Änderung in den Beobachtungen garantiert auf X zurückzuführen — irgendeinen Grund muß es schließlich geben und X ist der einzige, der in Frage kommt.

Dies ist natürlich nie ganz zu erreichen; es werden sich immer auch noch andere Dinge ändern als X, da X (genannt die „Behandlung") eine gewisse Zeit braucht: zum Beispiel schreitet die Zeit fort, der Bewußtseinszustand des Experimentators ändert sich, es schlagen unterschiedliche Zahlen von Gammaquanten in den Versuchsaufbau ein usw. Aber man kann die obige Forderung sinnvoll abschwächen: X muß das einzige sein, was sich ändert unter all denjenigen Faktoren, von denen man sinnvollerweise erwarten kann, daß sie vielleicht einen Einfluß auf y haben. Das bedeutet, S muß eine vollständige Beschreibung der Situation sein. S muß alle Faktoren umfassen, die einen Einfluß ausüben könnten. Die Faktoren werden zusätzliche Variablen genannt, weil sie über die im Experimentaufbau eigentlich zu betrachteten unabhängigen Variablen einen Einfluß auf die abhängigen Variablen ausüben.

Alle zusätzlichen Variablen müssen konstantgehalten werden. Dieses Konstanthalten nennt man die Kontrolle von Variablen — eine Kerntechnik experimentellen Arbeitens.

Es gibt eine Reihe von zusätzlichen Variablen, die in vielen Experimenten in der einen oder anderen Form auftreten und die deshalb in jedem Experimententwurf zu prüfen und gegebenenfalls zu berücksichtigen sind.

- **Individuelle Unterschiede**: Die mit Abstand wichtigste zusätzliche Variable sind die Unterschiede zwischen einzelnen Menschen, seien sie nun angeboren oder durch Ausbildung und Erfahrung bedingt. In der Softwaretechnik ist dieser Faktor sogar wichtiger als die meisten technischen Faktoren und muß deshalb besonders sorgfältig kontrolliert werden.

- **Historie**: Speziell bei Experimenten, die über eine längere Zeit gehen (mehrere Tage), können irgendwelche Einflüsse, die außerhalb des Versuchsaufbaus liegen, das Verhalten beeinflussen. Beispiel: Eine Versuchsperson liest während eines Experiments einen sehr lehrreichen Artikel oder hat eine andere lehrreiche Erfahrung außerhalb des Experiments, die ihre Leistung verbessert.

- **Reifung (maturation)**: Ebenso können bei längeren Experimenten Effekte Einfluß haben, die sich im Innern einer Versuchsperson einfach dadurch abspielen, daß Zeit vergeht. Beispiel: alle Studien, die Lehr- und Lernmethoden untersuchen, müssen sicherstellen, daß die beobachteten Effekte sich nicht „von alleine“, sondern wirklich aufgrund der Lehrmethode einstellen. Reifung und Historie sind eng verwandt und können (und müssen) meist nicht klar getrennt werden.

- **Instrumentation**: Eventuell ergeben sich über die Zeit Änderungen im Beobachtungs- und Meßaufbau — durch dessen Benutzung. Beispiel: Wenn Programmierer Fehlerlogbücher führen, in denen eine Fehlerklasse eingetragen wird, so ändern sich die verwendeten Klassifikationskriterien wahrscheinlich im Laufe von Wochen allmählich, weil die Fehlerklassifikation und ihr Nutzen allmählich immer besser verstanden werden. Dadurch scheinen die Meßdaten Änderungen zu dokumentieren (im Typ der aufgetretenen Fehler nämlich), die in Wirklichkeit vielleicht überhaupt nicht existieren.

- **Sterblichkeit (mortality)**: Bei fast jedem Experiment fallen Versuchspersonen während der Experimentdurchführung aus; diesen Effekt nennt man Sterblichkeit. Wenn es einen Zusammenhang zwischen der Häufigkeit des Herausfallens und der Behandlung gibt, so ist dieser eine zusätzliche Variable. Dieser Effekt ist eines der schwierigsten Probleme bei medizinischen Experimenten. Sterblichkeit bedeutet durchaus nicht, daß eine Versuchsperson zu Tode kommt, sondern lediglich, daß sie am weiteren Experiment nicht mehr teilnimmt; oft ohne Angabe von Gründen. Beispiel: Gerade eine positive Entwicklung kann ein Grund für Sterblichkeit sein. In der Medizin könnte es etwa bei einem Vergleich von Medikamenten vielen Versuchspersonen mit Medikament A schon nach der halben Behandlungsdauer so gut gehen, daß sie nicht mehr auftauchen, um die Ergebnisse erfassen zu lassen. In der Softwaretechnik könnte es beim Vergleich von Techniken passieren, daß die Personen, die eine schlechtere Technik verwenden sollen, vermehrt das Experiment vorzeitig abbrechen.

- **Statistische Regression**: Wenn ein Experiment aus einem Pretest (vor der Behandlung) und einem Posttest (nach der Behandlung) besteht, so ist zu berücksichtigen, daß im Rahmen normaler statistischer Schwankungen alle extrem hohen und extrem niedrigen Ergebnisse (Ausreißer) aus dem Pretest die Neigung haben, sich im Posttest zu „normaleren“ Ergebnissen hin zu verändern und zwar unabhängig von der Wirkung der Behandlung selbst. Dies ist einfach eine Folge der statistischen Seltenheit solcher Ausreißer und wird dann zum Problem, wenn man gerade solche Testergebnisse (aus einem Vortest) zur Auswahl der Versuchspersonen oder zu ihrer Einteilung in Gruppen heranzieht. Siehe auch im Abschnitt 2 über Psychologie unter Regressionsfehlschluß (3).

- **Anforderungscharakteristik**: Die Art, wie die Anforderungen und Aufgaben des Experiments einer Versuchsperson übermittelt werden, kann erheblichen Einfluß auf deren Verhalten haben. Derartige Beeinflussung kann sehr leicht passieren. Zum Beispiel macht es sicherlich einen Unterschied, ob ich einer Versuchsperson sage: „Schreiben Sie bitte mal dieses Programm und achten Sie darauf, möglichst wenige Fehler drinzuhaben.“ oder ob ich die gleiche Anforderung formuliere als „Schreiben Sie bitte dieses Programm und geben Sie sich alle erdenkliche Mühe, keinesfalls mehr Fehler dabei zu machen oder vor allem zu hinterlassen, als absolut und unbedingt unvermeidbar für Sie ist!“ Dieser Schwierigkeit geht man beispielsweise dadurch aus dem Weg, indem man die Versuchsanleitung in schriftlicher Form oder in Form von Tonbändern oder Videos verfügbar macht und verbalen Kontakt vermeidet.

- **Erwartung (expectancy)**: Die Erwartungen des Experimentators im Hinblick auf das Ergebnis des Experiments können ebenfalls dessen Verlauf unerwünscht beeinflussen. Ein Beispiel dafür wäre die obenbeschriebene unterschiedliche Motivation der Versuchspersonen, wenn unbewußt die Mitglieder einer Gruppe besser motiviert werden als die der anderen. Andere Auswirkungen können unbewußte Verfälschungen beim Erfassen oder Interpretieren der Daten sein. Da Versuchspersonen normalerweise versuchen, sich möglichst positiv darzubieten und Erwartungen gerecht zu werden, können solche Erwartungen auch vom Experimentator auf das Verhalten einer Versuchsperson überspringen.

- **Sequenzeffekte**: Werden die gleichen Versuchspersonen nacheinander verschiedenen Behandlungen unterworfen, deren Effekte miteinander verglichen werden sollen, so ist die Reihenfolge der Behandlungen fast immer von Bedeutung, wobei die Art der Auswirkung sich oft sehr schlecht abschätzen läßt. Zur Kontrolle dieser Variablen müssen alle Reihenfolgen gleichmäßig miteinander abwechseln. Im Falle der Softwaretechnik treten Sequenzeffekte meist in Form von Lerneffekten auf. Beispiel: Personen, die ein Programm mit Methode A und dann ein ähnliches Programm mit Methode B entwerfen sollen, werden tendenziell mit der jeweils zweiten Methode besser abschneiden, ob sie nun besser ist oder nicht, einfach weil die Personen während des Experiments Übung im Entwerfen dieser Art von Programme bekommen.

- **Erfahrenheit (sophistication)**: Wissen der Versuchspersonen über die Methoden und Ziele experimenteller Forschung kann das Verhalten beeinflussen. Dieses Problem ist in der Psychologie allgegenwärtig; in der Softwaretechnik tritt es meist nur an den Stellen auf, wo man nur einen sehr schmalen Aspekt des Verhaltens untersuchen möchte und vermeiden muß, daß die Versuchspersonen sich unnatürlich stark auf diesen Aspekt konzentrieren. In diesem Fall ist das Verschweigen und Verschleiern des eigentlichen Forschungsziels bei forschungserfahrenenen Versuchspersonen erschwert. Beispiel: Ein Experiment zur Messung der Fehlerreduktion durch automatische Speicherbereinigung (garbage collection) im Vergleich zu manueller Speicherfreigabe wird verfälscht, wenn sich die Versuchspersonen ungewöhnlich stark auf diese Art von Fehlern konzentrieren.

In der Psychologie sind noch eine ganze Reihe weiterer, subtiler zusätzlicher Variablen von Interesse [Chr94, S.211ff], die aber in der Softwaretechnik glücklicherweise meist kaum eine Rolle spielen.

3.4.3 Zufallsstichproben und Vergleichsgruppen

Wie kontrolliert man denn nun zusätzliche Variablen? Die wichtigste Technik zur Kontrolle von Variablen im Bereich der Softwaretechnik ist die Benutzung von Vergleichsgruppen auf Basis von Zufallsstichproben.

Die Überlegungen hierzu sind folgende:

1. Wenn man einen menschlichen Faktor vermessen will, muß man mehrere Menschen betrachten und den Mittelwert oder die ganze Verteilung berücksichtigen, denn Menschen verhalten sich sehr verschieden.

2. Auch eine beträchtliche Anzahl der oben in 3.4.2 besprochenen zusätzlichen Variablen ist an die einzelne Versuchsperson gebunden.

3. Diese beiden Probleme kann man dadurch in den Griff bekommen, daß man im Experiment eine Gruppe von Menschen benutzt, die zufällig aus der Grundmenge von Menschen, die im Experiment betrachtet werden soll (also über die eine Aussage gemacht werden soll), ausgewählt werden: Mehrere Personen zu nehmen kann die Schwankungen in den Menschen und den zusätzlichen Variablen ausgleichen. Sie zufällig auszuwählen bedeutet, daß

diese Möglichkeit mit hoher Wahrscheinlichkeit auch tatsächlich eintritt und sich nicht versehentlich durch bestimmte Merkmale der Auswahlprozedur systematische Ähnlichkeiten zwischen ihnen einstellen können, die das Ergebnis verfälschen.

4. Zufallsauswahl liefert also eine Stichprobe, deren Eigenschaften wir zwar vielleicht nicht genau beschreiben können, von der wir aber wissen, daß sie repräsentativ ist und die relevante Situation, über die das Experiment eine Aussage machen soll, deshalb gut charakterisiert. Allerdings darf die Stichprobe nicht zu klein sein, sonst können zufällige Auswahleffekte die Repräsentativität leicht zunichte machen.

5. Wenn mein Experiment nur eine einzige Behandlung auf ihre Wirkung prüfen soll, so ist nun häufig noch die Frage, ob die beobachteten Wirkungen tatsächlich von der Behandlung herrühren oder von zusätzlichen Variablen, die ich nicht explizit kontrolliert habe. Zufallsauswahl gleicht zwar Schwankungen in den zusätzlichen Variablen zwischen verschiedenen Menschen aus, kontrolliert aber diese zusätzlichen Variablen nicht.

6. Also muß ich zu meiner eigentlichen Behandlungsgruppe noch eine Vergleichsgruppe verwenden, die das Experiment in gleicher Weise durchläuft, nur eben die Behandlung nicht erhält (sondern eine neutrale Scheinbehandlung, ein Placebo). Alle zusätzlichen Variablen, die meinen Versuch beeinflussen, werden nun genauso auf die Vergleichsgruppe wie auf die Behandlungsgruppe wirken und fallen beim Vergleich der beiden deshalb aus der Betrachtung heraus: die zusätzlichen Variablen sind kontrolliert.

Um das noch klarer zu machen, betrachten wir den umgekehrten Fall: Angenommen, wir machen **keine Zufallsauswahl**. Dann kann es uns passieren, daß wir bei der Auswahl eine gewisse Vorliebe für eine bestimmte Teilmenge der zu betrachtenden Grundmenge entwickeln. Angenommen, wir vergleichen die Entwurfsmethoden A und B. Dann ist es naheliegend, als Versuchspersonen für A Leute zu nehmen, die A bereits kennen. Falls es solche gibt, jedoch keine Leute gibt, die B schon kennen, ist es kein Wunder, wenn A besser abschneidet. In diesem Fall macht unser Experiment also eine Aussage über eine andere Situation, als wir eigentlich betrachten wollen (z.B. Informatikstudierende, die in A geschult sind, in B aber nicht; anstatt einfach alle Informatikstudierenden). Diese Aussage ist erheblich spezieller und dementsprechend wenig wertvoll.

Hier sehen wir auch das Hauptproblem bei der Zufallsauswahl: Wenn wir nicht unsere Grundmenge extrem eng definieren, ist eine echte Zufallsauswahl fast stets unmöglich, weil wir immer auf nur wenige Elemente der Grundmenge überhaupt für unser Experiment Zugriff haben; die meisten stehen als Versuchspersonen gar nicht zur Verfügung. Hier kann man nur mit Geschick und Urteilsvermögen versuchen, die sinnvollste mögliche Stichprobe zusammenzutragen. Beispielsweise kann man versuchen, unterschiedliche Vorbildung explizit durch Schulung vor dem Experiment auszugleichen.

Zweiter Teil des Problems: Angenommen, wir machen ein Experiment **ohne Vergleichsgruppe**. Dann kann es uns passieren, daß die scheinbare Wirkung unserer Behandlung, die wir beobachten, gar nicht auf die Behandlung zurückzuführen ist, sondern auf andere Faktoren, wie die Reifung oder die Historie der Versuchspersonen — oder irgendeine Mischung von Behandlung und zusätzlichen Variablen, die wir nicht aufdröseln können. In diesem Fall ist das Experiment vollkommen wertlos.

Ein Problem selbst bei der Verwendung von Vergleichsgruppen kann der Einfluß von Erwartungen sein. Erwartungen auf der Seite der Versuchspersonen können mit **Scheinbehandlungen** (Placebos) vermieden werden, die allerdings in der Softwaretechnik oft schwer zu konstruieren sind. Ein solcher Versuch, bei dem die Versuchperson nicht weiß, welcher Behandlung sie unterzogen wird, heißt **Blindversuch**. Sind zusätzlich auch Einflüsse durch Erwartungen auf der Seite des Experimentators zu befürchten, so wird üblicherweise z.B. in der Medizin die sogenannte **Doppel-Blind-Technik** verwendet. Der Name stammt daher, daß weder Experimentator noch Versuchsperson wissen, ob gerade die echte Behandlung oder die Scheinbehandlung angewendet wird. Was bei Pillen leicht zu erreichen ist, ist aber in Softwaretechnik-Experimenten meist unmöglich; der einzige Ausweg besteht darin, daß sich der Experimentator so weit wie möglich aus dem Experiment heraushält und die Einweisung der Versuchspersonen nur durch Computeranweisungen, gedrucktes Material, Videos etc. vorgenommen wird.

3.4.4 Varianzreduktion

Selbst wenn wir einen Versuchsaufbau gefunden haben, der alle zusätzlichen Variablen ordentlich kontrolliert und deshalb im Prinzip saubere Schlüsse aus dem Experiment gestattet, kann es leicht passieren, daß wir kein schlüssiges Resultat erhalten.

Wenn der zu beobachtende Behandlungseffekt, also der Unterschied zwischen den Vergleichsgruppen, nicht groß ist, kann er von den Schwankungen (Varianz) innerhalb der beiden Zufallsstichproben völlig verdeckt werden. Siehe als Beispiel die Abbildung 3.3.

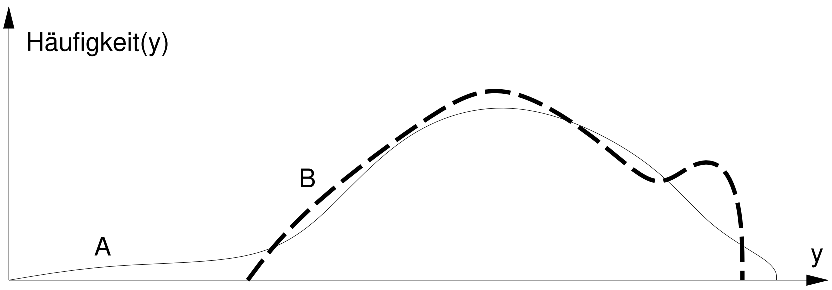

Abbildung 3.3: Resultatverteilung zweier hypothetischer Versuchsgruppen

Hier sind die Verteilungen einer beobachteten Variablen y für die zwei Vergleichsgruppen A und B angegeben. Die Experimentfrage lautet, ob der Mittelwert bei A kleiner ist als bei B. Sieht ja ein bißchen so aus, nicht wahr? Aber sicher kann man sich nicht sein, weil die Zufallsschwankungen innerhalb der Verteilungen so viel größer sind als der Unterschied — falls es ihn gibt.

Was kann man tun, um solche Schwankungen oder ihre Auswirkung zu minimieren? Zur Verringerung der Schwankungen dient die Vorsortierung. Hierbei schränke ich meine Versuchspersonen ein auf solche, die sich in den relevanten Eigenschaften, die die Schwankung verursachen, möglichst ähneln. Wie das konkret zu erreichen ist, muß man im Einzelfall untersuchen; es kann jedenfalls sehr schwierig sein.

Zur Verringerung der Auswirkungen der Schwankung dienen sogenannte Paarungstechniken (matching techniques). Der Grundgedanke hierbei ist, daß ich eine oder mehrere zusätzliche Variable ausdrücklich in meinem Experimentaufbau erfasse und dann dafür sorge, daß sie in beiden Versuchsgruppen die gleiche Verteilung haben. Beispielsweise könnte man seine Versuchspersonen vorsortieren in sehr gute, gute und weniger gute Programmierer und dann dafür sorgen, daß jede Versuchsgruppe gleich viele von jeder Sorte erhält. Somit würden sich die Varianzen der Gruppen ähneln, wenn beide dieselbe Behandlung erhielten. Wenn im Experiment trotzdem unterschiedliche Varianzen auftreten, so nur, weil die Behandlung das nach sich zieht.

Eine besonders hohe Aussagekraft können wir dann erzielen, wenn es uns gelingt, Versuchspersonen jeweils so zu Paaren zusammenzustellen, daß wir annehmen dürfen, die beiden Mitglieder jedes Paares seien äquivalent. Die dann zulässigen statistischen Auswertungsmethoden führen zu einer erheblich höheren Empfindlichkeit des Experiments auf die erstrebte Aussage hin. Ob solche Paarung möglich ist und welche Maßnahmen nötig sind, um die Paare zu finden, hängt vom Einzelfall ab und muß mit dem gesunden Menschenverstand anhand des verfügbaren Wissens entschieden werden. Die einfachste Methode zum Paaren besteht darin, *dieselbe* Versuchsperson beide Behandlungen durchlaufen zu lassen. Das ist aber natürlich nur möglich, wenn wir Lerneffekte ausschließen oder kontrollieren können. Oft ist das Kontrollieren von Lerneffekten dadurch möglich, daß beide Reihenfolgen von Behandlungen gleich häufig in jeder Versuchsgruppe vorkommen; diese Technik wird Gegenbalancierung (counter-balancing) genannt.

3.4.5 Versuchspläne

So nach und nach haben wir jetzt schon die meisten wichtigen Überlegungen zur Aufstellung von Versuchsplänen (experiment designs) besprochen. Der Versuchsplan beschreibt die Regeln, nach denen wann für welche Subjekte welche Behandlung angewendet und welcher Test zur Beobachtung durchgeführt wird.

Gehen wir mal von der Bildung zweier Versuchsgruppen mittels reiner oder durch Paarung modifizierter Zufallsauswahl aus. Die Gruppen dienen zum Vergleich zweier Behandlungen oder dem Vergleich einer Behandlung mit einer Scheinbehandlung. Dann gibt es folgende Hauptentscheidungen bei der Aufstellung eines Versuchsplans:

Erstens muß ich wählen zwischen einem **Intra-Subjekt**-Plan und einem **Inter-Subjekt**-Plan. Bei ersterem durchläuft jede Versuchsperson beide Behandlungen, im anderen Fall nicht. Zweitens gibt es **Vortest-Nachtest**-Pläne und **Nur-Nachtest**-Pläne. Im ersteren Fall beobachtet man den Unterschied in einer Variable von vor zu nach der Behandlung, im zweiten Fall nur den Wert nach der Behandlung. Vortests sind z.B. in der Regel sinnvoll bei Experimenten, die mit Lernen zu tun haben. In der Softwaretechnik kommt man oft ohne Vortests aus.

In vielen Fällen ist nur jeweils eine der beiden Möglichkeiten machbar, manchmal kann man jedoch auch auswählen. In der Regel wird man dann Intra-Subjekt-Pläne bevorzugen, weil sie sparsamer und aussagekräftiger sind und einen Vortest nur dann durchführen, wenn man befürchtet, auf die dort gewonnene Information zur Interpretation der Ergebnisse angewiesen zu sein.

Alle bisherigen Überlegungen lassen sich natürlich auf den Fall von mehr als zwei Behandlungen ausdehnen. Da aber der Fall von nur zweien bereits alle wesentlichen Probleme enthält, habe ich auf diese zusätzliche Komplikation verzichtet.

Versuchspläne, in denen mehrere einzelne Behandlungseinzelteile auf alle möglichen Arten miteinander zu Behandlungen kombiniert werden, nennt man **faktorielle Versuchspläne (factorial designs)**; werden einige der dabei entstehenden Behandlungsarten aus dem Experiment herausgelassen, spricht man von **partiellen faktoriellen Versuchsplänen (partial factorial designs)**. Möchte ich beispielsweise den Einfluß zweier Entwurfsmethoden A und B untersuchen, wenn mit je einer von drei verschiedenen Programmiersprachen, sagen wir C, C++ und Eiffel implementiert wird, so erhalte ich 6 Behandlungen (A/C, A/C++, A/Eiffel, B/C, B/C++, B/Eiffel) für einen faktoriellen Versuchsplan. Wähle ich aus diesen nur einige aus, weil beispielsweise die Kombination von B mit C von vornherein keinen Sinn ergibt, so erhalte ich einen partiellen faktoriellen Versuchsplan. So beeindruckend können Ausdrücke für ganz einfache Ideen sein.

Nicht alles, was auf Anhieb wie ein Versuchsplan für ein kontrolliertes Experiment aussieht, ist wirklich ein solcher; es gibt auch **fehlerhafte Versuchspläne**. Das wichtigste Beispiel dafür ist der **Eingruppen-Vortest-Nachtest**-Plan. Dieser kontrolliert mangels Vergleichsgruppe diverse zusätzliche Variablen nicht, nämlich alle, die zwischen Vortest und Nachtest außerhalb der eigentlichen Behandlung einen Effekt haben könnten, insbesondere Reifung und Historie.

Daneben gibt es noch **quasi-experimentelle Versuchspläne**, die eine brauchbare, aber unvollständige Annäherung an echte kontrollierte Experimente darstellen. Hier die beiden wichtigsten Beispiele:

Nichtequivalente Kontrollgruppe: Hier wird die Zugehörigkeit der Versuchspersonen zu den Vergleichsgruppen nicht per Zufall festgelegt, sondern die Personen melden sich entweder nach persönlicher Vorliebe selbst oder sie werden aufgrund eines äußeren Kriteriums zugewiesen, von dem man annehmen muß, daß es von Belang für die Beobachtungen ist. In beiden Fällen fließen zusätzliche Variablen in die Beobachtungen ein, die nicht kontrolliert werden.

Unterbrochene Zeitreihe: In Fällen, in denen für einen Vortest-Nachtest-Versuch keine Kontrollgruppe zum Vergleich mit der Behandlungsgruppe zur Verfügung steht, kann man sich mit einem Zeitreihenversuchsplan behelfen. Das Problem ist ja, daß Effekte im Nachtest vielleicht auf Reifung oder Historie zurückgeführt werden müssen anstatt auf die Behandlung. Also wendet man den Vortest mehrfach in gleichen Abständen an und ebenso den Nachtest. Ergibt sich nun über die Vortests und über die Nachtests jeweils eine gleichmäßige Entwicklung, so kann man annehmen, daß Reifung und Historie einen gleichmäßigen Einfluß über die Zeit ausüben. Beobachtet man nun zwischen letztem Vortest und erstem Nachtest einen Sprung, der im Vergleich dazu unerwartet groß ist, kann man ihn auf die Behandlung zurückführen. Siehe das Beispiel in Abbildung 3.4.

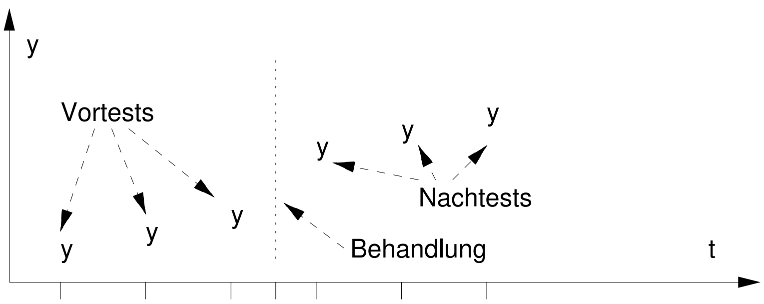

Abbildung 3.4: Beobachtungen in einem Quasi-Experiment

Können nur echte kontrollierte Experimente überzeugende wissenschaftliche Beweise abgeben? Nein, ein quasi-experimenteller Versuch kann völlig ausreichend sein. Betrachten wir folgendes Beispiel: Angenommen, ein 30jähriger, kerngesunder Mensch tritt auf die Straße, wird von einem Auto überfahren und hinterher für tot befunden. Dann gehen wir normalerweise davon aus, daß der Autounfall, also der Zusammenstoß, die direkte Todesursache war. Das ist nicht gesichert, denn eine Reihe anderer Möglichkeiten (zusätzlicher Variablen) wie ein plötzlicher Herzanfall oder ein Gehirnschlag sind nicht kontrolliert worden und können deshalb eigentlich nicht als Grund ausgeschlossen werden. Aber die Annahme, daß der Aufprall der Grund war, ist so plausibel, daß wir sie vernünftigerweise als richtig akzeptieren.

3.4.6 Statistische Auswertung

Durch bloßes Draufgucken auf die beobachteten Werte läßt sich in vielen Fällen die Forschungsfrage nicht klar entscheiden. Siehe nochmals die Abbildung 3.3: Ist denn nun A im Mittel kleiner oder nicht?

Glücklicherweise hält die Statistik Methoden bereit, wie man solche Fragen quantitativ beantworten kann — allerdings können die Aussagen immer nur Wahrscheinlichkeitsaussagen sein, denn die betrachteten Phänomene sind ja stochastischer Natur. Experimentatoren in der Softwaretechnik müssen solche Techniken kennen und korrekt anwenden können oder zumindest verstehen, was man damit machen kann und zur Anwendung Fachleute aus der Statistik um Rat fragen. Man beachte die Formulierung „korrekt anwenden", denn leider gibt es eine Reihe von Fehlern, die bei der Anwendung statistischer Methoden immer wieder gemacht werden. Die Ergebnisse werden in einem solchen Fall wertlos.

Für die Softwaretechnik-Forschung sind vor allem zwei Arten von statistischen Methoden relevant: Tests und Regression. Wir können beide nur sehr grob anhand eines Beispiels besprechen, weil eine ausführliche Einführung und Diskussion ein ganzes Statistikbuch füllt.

Ein Test (genauer: **statistischer Signifikanztest**) ist eine Methode, um die Gültigkeit einer quantitativen Hypothese zu prüfen. Hierzu wird die Wahrscheinlichkeit berechnet, mit der das beobachtete Ergebnis eintreten würde, wenn man annimmt, daß die Hypothese nicht gilt (diese Annahme nennt man die Nullhypothese). Ist diese Wahrscheinlichkeit klein, so können wir annehmen, daß unsere Hypothese gilt und die richtige Erklärung für die Beobachtung ist. Ist die Wahrscheinlichkeit groß, so wissen wir nichts: vielleicht ist unsere Hypothese falsch, vielleicht stimmt sie aber auch und wir hatten nur keine Versuchsbedingungen, die ein zufälliges Eintreten der gleichen Beobachtungen gut genug verhindert haben (zum Beispiel zu wenige Beobachtungen; da kann vieles noch zufällig vorkommen).

Hier ein Beispiel: Wir wollen die Behauptung prüfen, daß unsere neue Entwurfsmethode E zu einer geringeren Zahl von Entwurfsfehlern führt als die altbekannte Methode A. Also beschaffen wir uns, sagen wir 40 Versuchspersonen, lassen jede davon einen Entwurf des gleichen Programms machen, eine Gruppe von 20 mit Methode A und eine andere Gruppe von 20 mit Methode E. Wir erhalten also 20 Entwürfe A_i und 20 Entwürfe E_i. Zuvor haben wir jede Gruppe in „ihrer" Methode gleich gut ausgebildet. Dann stellen wir in jedem Entwurf die Zahl F der Entwurfsfehler fest, indem wir zum Beispiel den Entwurf implementieren und testen. Wir nehmen an, die gemessenen Fehleranzahlen jeder Gruppe seien im Mittel über jede Gruppe korrekt. Sei für alle i $F(A_i) = a_i$ und $F(E_i) = e_i$. Dann lautet unsere Forschungshypothese H:

H: Im Mittel ist die Fehlerzahl der E_i kleiner als die der A_i, also $\bar{e_i} < \bar{a_i}$.

Nehmen wir an, wir haben folgende Fehlerzahlen bei E gemessen:
(0 4 0 2 0 0 2 1 1 0 0 0 1 3 6 0 3 0 1 0)
und folgende bei A:
(0 4 1 2 2 3 5 1 0 0 6 6 1 3 3 1 2 1 4 0).

Dann können wir uns zunächst die Verteilungen der Fehlerhäufigkeit bei Methode A und Methode E anhand dieser beiden Stichproben ansehen. Dazu stellen wir jeweils das Histogramm auf, wie in Abbildung 3.5 gezeigt.

Das linke Histogramm besagt also beispielsweise, daß es in der Stichprobe zu Methode A insgesamt 5 Elemente gibt, die eine Fehlerzahl von 1 haben usw. Diese Histogramme scheinen die recht klare Nachricht zu enthalten, daß E tatsächlich, wie erwartet, kleinere Fehlerzahlen produziert. Die **Mittelwerte** sind mit 2.25 für A und 1.2 für E auch weit verschieden — es scheint also alles klar zu sein. Aber können wir diese Aussage auch statistisch bestätigen? Dazu müssen wir die Hypothese H einem statistischen Signifikanztest unterziehen. Ein häufig benutzter Test ist der t-Test; er prüft die Annahme, daß die Mittelwerte zweier Verteilungen verschieden sind. Bevor wir ihn einsetzen, müssen wir zunächst prüfen, ob seine **Anwendungsvoraussetzungen** erfüllt

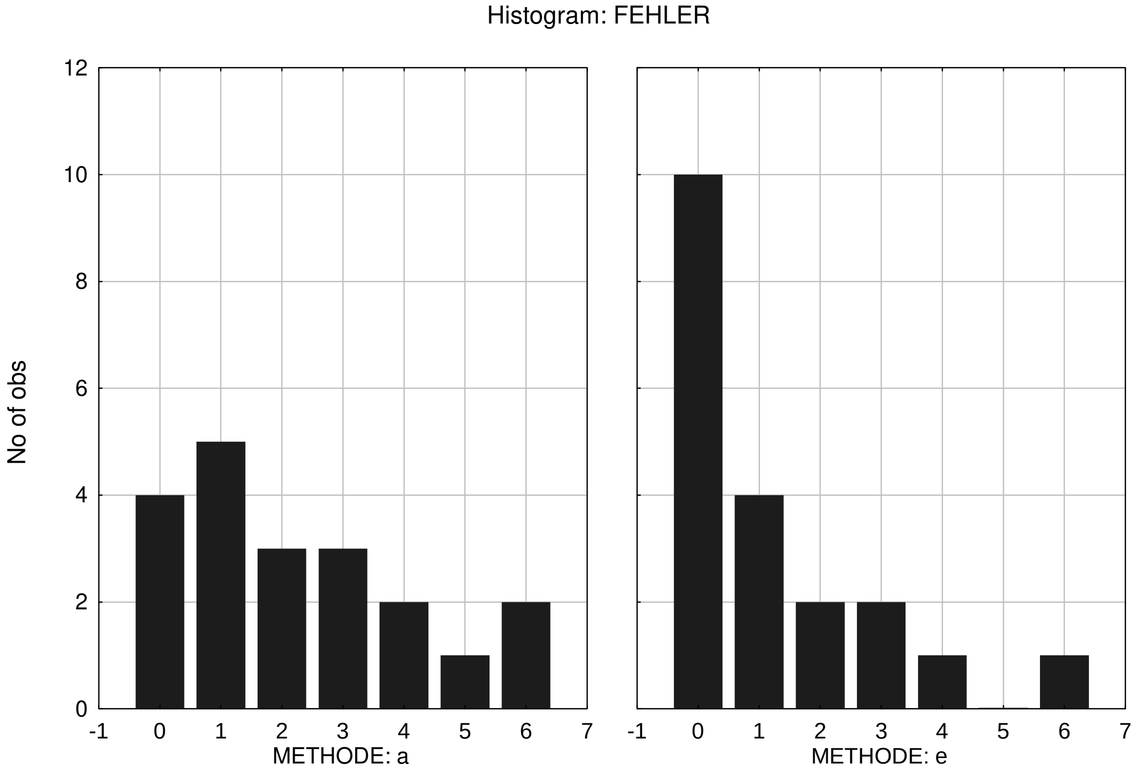

Abbildung 3.5: Histogramm der Fehlerverteilungen von A und E

sind. Der 2-Stichproben-t-Test, den wir hier einsetzen könnten, macht zwei Annahmen: Erstens, daß beide Stichproben einer normalverteilten Population entstammen und zweitens, daß die Varianzen dieser beiden Normalverteilungen gleich sind. Diese Voraussetzungen sind natürlich niemals exakt erfüllt, aber wir müssen zumindest für eine „hinreichend gute“ Erfüllung sorgen.

Für die Varianz verwendet man beim t-Test normalerweise die Faustregel, daß die Standardabweichung um maximal Faktor 2 verschieden sein darf. Das ist im vorliegenden Fall erfüllt: Stichprobe A hat eine Standardabweichung von 1.94, B von 1.67. Schwieriger sieht es bei der Form der Verteilung aus. Schon Probe A hat nur wenig Ähnlichkeit mit einer Normalverteilung, aber bei B kann man ganz sicher nicht von normalverteilten Daten sprechen. Eine andere Darstellungsart der Daten zur Prüfung der Normalverteilungsannahme ist der sogenannte Normalquantilplot. Hierbei werden die tatsächlichen Quantile der Daten so aufgetragen, daß normalverteilte Daten auf einer Geraden liegen würden. Diese Gerade wird dann als Referenz eingetragen und die Stärke der Abweichung der Datenpunkte von dieser Geraden gibt die Abweichung von einer Normalverteilung an. Dies ist in Abbildung 3.6 zu sehen.

Bei E sind die Abweichungen so massiv, daß wir diese Stichprobe nicht mehr als normalverteilt annehmen sollten. A kann notfalls noch als normalverteilt durchgehen.

Den t-Test sollten wir uns auf diesen Daten also lieber verkneifen, weil seine Anwendungsvoraussetzungen nicht erfüllt sind. Was würde geschehen, wenn wir ihn dennoch anwenden? Der t-Test hat die Eigenschaft, daß jede Abweichung der Daten von der Normalverteilung eventuell vorhandene Unterschiede in den Mittelwerten tendenziell verwischt. Wenn also ein t-Test behauptet, ein Unterschied sei vorhanden, dann stimmt das auch; es kann nur leicht passieren, daß wir einen vorhandenen Unterschied nicht nachweisen können. Also können wir den t-Test doch anwenden? Ja, wir können. Wir müssen nur darauf gefaßt sein, daß die Ergebnisse einen schwächeren Unterschied anzeigen als in Wirklichkeit vorhanden ist.

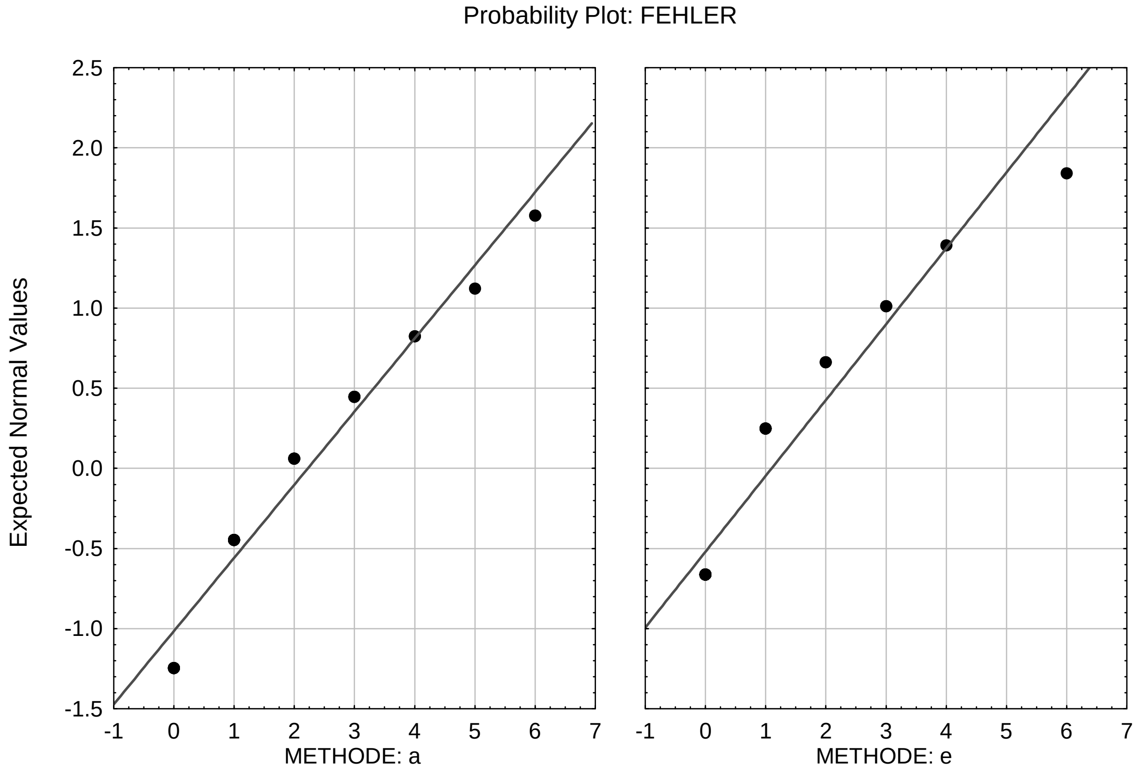

Abbildung 3.6: Normalquantilplot der Fehlerverteilungen von A und E

Die Anwendung des t-Tests liefert in unserem Beispiel ein Resultat für p von 0.075. Was bedeutet das? Es heißt, der t-Test ergibt, daß die Wahrscheinlichkeit p dafür, daß die Nullhypothese „Mittelwerte bei A und E gleich“ gültig ist und die in den Daten beobachteten Abweichungen der Mittelwerte zufällig sind, bei 7.5% liegt. Mit anderen Worten: wahrscheinlich ist E wirklich besser als A, aber so ganz sicher können wir nicht sein. Man nennt p auch das Signifikanzniveau des Tests. Üblicherweise legt man eine Schranke für p im Vorhinein fest und entscheidet an ihr, ob man die Hypothese für erfüllt hält oder nicht. Ein üblicher Wert für diese Schranke liegt bei 0.05, also Chancen von eins zu zwanzig für ein zufälliges Ablehnen der Nullhypothese. In kritischen Fällen wählt man den Wert niedrigerer, z.B. bei lebenskritischen medizinischen Untersuchungen zu 0.01 oder 0.001 und in wenig kritischen Fällen, wo nur nach ungefähren Hinweisen gesucht wird, auch höher, vielleicht 0.1 oder sogar 0.2.

Wenn nun der t-Test keine so gute Idee ist, weil die Daten nicht normalverteilt sind, wie können wir denn dann unsere Hypothese prüfen? Für solche Fälle gibt es die sogenannten nichtparametrischen Methoden, so genannt, weil sie nicht einen Parameter (beim t-Test: den Mittelwert) einer festgelegten Verteilung prüfen, sondern nur schwache Annahmen über die Form der Verteilung machen. Zum Vergleich von Mittelwerten ist am besten geeignet der Mann-Whitney U-Test (auch oft als Wilcoxon Rangsummentest bezeichnet). Dieser Test macht nur die Annahme, daß die Daten auf einer Ordinalskala liegen, so daß man sie ordnen kann (der t-Test verlangt eine Intervallskala, weil sich sonst keine Mittelwerte berechnen lassen!). Aufgrund dieser schwachen Annahme kann der U-Test keine Aussage über die Mittelwerte machen, sondern er gibt, vereinfacht gesagt, an, ob die eine Verteilung einen höheren Median (50%-Quantil) hat als die andere oder nicht. Er macht also eine Aussage anhand der Quantile, statt über Mittelwert und Standardabweichung, was den zusätzlichen Vorteil hat, daß der Test sehr robust gegen „Ausreißer“ ist, d.h. wenige extreme Werte können das Testergebnis nur wenig beeinflussen, während die Stärke ihres Einflusses beim t-Test mit ihrer Größe wächst. Eine Darstellung der Daten, die noch kompakter ist als das Histogramm und die gut zu dieser nichtparametrischen Betrachtungsweise paßt, ist der Boxplot (Abbildung 3.7).

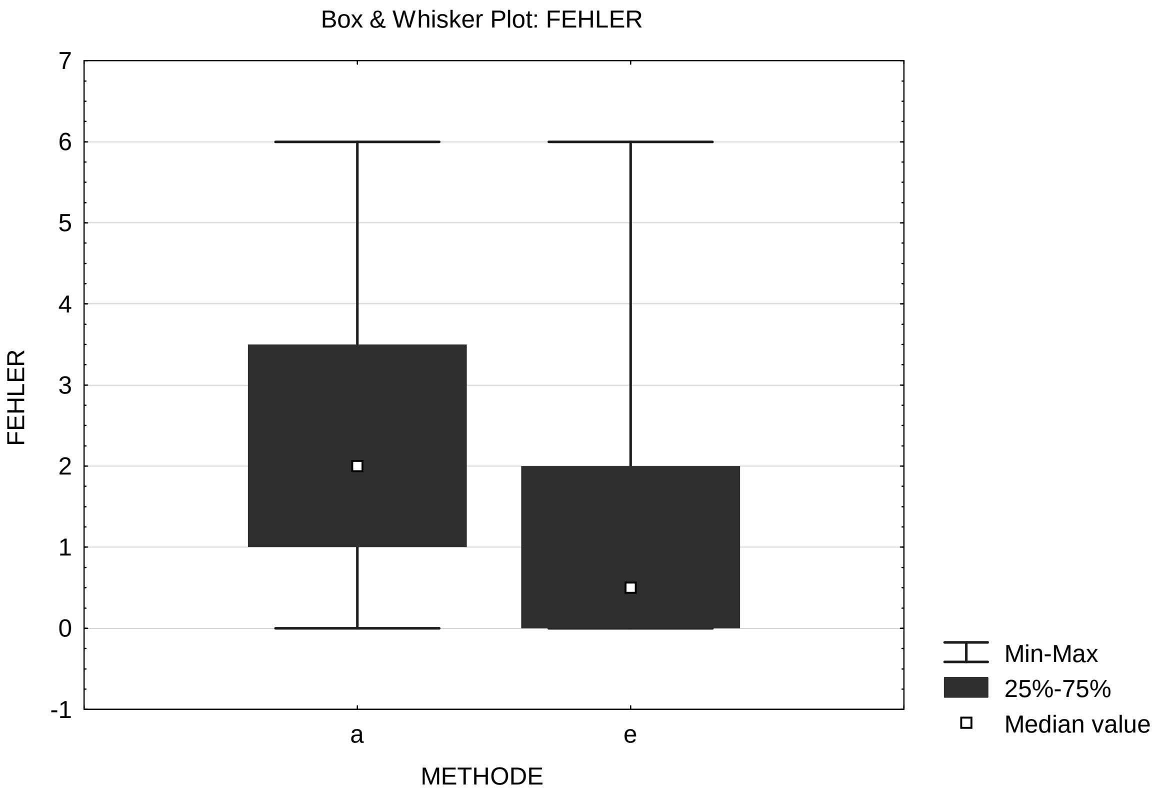

Abbildung 3.7: Boxplot der Fehlerverteilungen von A und E

Hierbei wird eine Verteilung über nur 5 Werte charakterisiert: Ihren Median (Punkt in der Mitte), ihr 25%- und 75%-Quantil (unterer und oberer Rand der Kiste) und zwei andere Punkte, die verschieden gewählt werden können (unterer und oberer Schwanz). Die letzteren sind üblicherweise die äußersten Punkte im nachfolgenden Interquartilbereich, d.h. die letzten Punkte innerhalb der Stichprobe, die höchstens einen Interquartilabstand vom ersten bzw. dritten Quartil entfernt sind. Wie bitte? Beispiel: der obere Schwanz im obigen Bild liegt bei demjenigen der 20 Stichprobenwerte, der gerade noch etwas weniger weit oberhalb des dritten Quartils (also 75%-Quantils) liegt, als das erste Quartil unterhalb des dritten liegt. Noch anders gesagt: Klapp die Kiste um ihren oberen Rand nach oben um und nimm den Schwanz bis dorthin, wo der letzte Datenpunkt ist, der dann noch innerhalb der Kiste wäre. Man kann die Schwänze auch anders berechnen, zum Beispiel setzt man sie einfach auf die 10%/90%-Quantile oder die 5%/95%-Quantile. In unserem Boxplot sind das Minimum und Maximum, also 0%/100%-Quantil benutzt worden, weil die Stichproben recht gutartig sind, in dem Sinne, daß die Minima und Maxima sinnvolle, interessante Werte sind und nicht irgendwelche Ausreißer. Gelegentlich sieht man auch Boxplots, die Mittelwerte und Standardabweichungen darstellen, aber das ist eher ungewöhnlich und auch weniger nützlich.

Ein solcher Boxplot ermöglich eine sehr einfache Abschätzung der wichtigsten Eigenschaften zweier Verteilungen; wesentlich besser, als es ein Histogramm vermag. In unserem Beispiel erkennen wir die erheblich verschiedenen Mediane von 0.5 bzw. 2.0 und sehen unsere Hypothese H eher noch klarer bestätigt als in den Histogrammen oben.

Der U-Test liefert auf unseren Daten einen p-Wert von 0.055. Die Signifikanz ist also höher als beim t-Test, aber immer noch knapp schlechter als die übliche Schranke für die Annahme von Hypothesen. Wie wir sehen, ist die Aussage unseres Experiments also gar nicht so glasklar, wie wir das als Erfinder von Methode E gern gesehen hätten. Wahrscheinlich ließe sich diesem Umstand jedoch mit ein paar zusätzlichen Versuchspersonen abhelfen, die die p-Werte erhöhen, falls die Verteilungen von ihrer Form her gleich bleiben: Je öfter die Unterschiede beobachtet werden, desto geringer wird die Chance, daß sie bloßer Zufall sind.

Boxplots und U-Test sind fast immer gute Hilfsmittel zum Vergleich der Lage von Verteilungen und sind häufig dem t-Test vorzuziehen. Sogar wenn die Anwendungsbedingungen des t-Tests erfüllt sind, hat der U-Test meist eine ähnliche, manchmal sogar höhere Trennschärfe als der t-Test. Soviel zu statistischen Tests.

Regression ist eine Methode, wie man zu einer Menge von Daten ein quantitatives Modell erzeugt, das diese Daten erklären kann. Wir haben ein Beispiel für die Anwendung der Regression bereits anhand der Validation eines Vorhersagesystems (in 1.6.3) gesehen. Hier sei ergänzend nur noch folgendes zugefügt:

- Regression kann man nicht nur zur Bestimmung der Parameter einer linearen Funktion einer Variablen machen. Im Prinzip lassen sich beliebige Funktionen benutzen, wobei die Berechnung der Parameter schwierig wird, wenn die Funktion nicht linear in diesen Parametern ist, wie zum Beispiel bei den populärsten neuronalen Netzen, den mehrschichtigen Perzeptrons. In diesem Fall muß man langsame Iterationsverfahren zur Bestimmung der Parameter verwenden, anstatt eine geschlossene und exakte Lösung angeben zu können. Merke: Die Funktion darf sehr wohl nichtlinear in ihren *Variablen* sein! So kann man beispielsweise ohne Schwierigkeiten Regressionen für quadratische oder logarithmische Modelle berechnen, z.B. $y = a + bx + cx^2$. Ebenso sind Regressionen mehrerer Variablen möglich (mehrfache Regression), z.B. $y = a + bx_1 + cx_2$. Sehr oft ist eine mehrfache lineare Regression von geeignet berechneten Eingangsgrößen ein besseres Modell als das, was beispielsweise ein neuronales Netz aus den Daten „lernen“ würde.

- Vor der Anwendung einer Regression muß sichergestellt sein, daß die benutzen Daten auch tatsächlich einen Zusammenhang der modellierten Art beschreiben. Andernfalls ist das erzeugte Modell sinnlos. Im Falle einer linearen Regression prüft man dies, indem man die Korrelation der Eingangsvariablen mit der Ausgangsvariablen betrachtet. Als Faustregel sollte diese über 0.7 liegen, damit man einen hinreichend starken Zusammenhang annehmen darf.

- Der Zusammenhang, der von einer Regression modelliert wird, muß durchaus nicht kausaler Natur sein, ein Regressionsmodell beschreibt einen Effekt, aber nicht den Mechanismus, aus dem er entsteht. Aus der Existenz eines guten Regressionsmodells auf eine Ursache/Wirkung-Beziehung zu schließen, ist also unzulässig.

Die Statistik hält zur Auswertung von Experimenten noch einige andere Techniken bereit, die wir hier nicht besprechen können. Auch zu den Themen Test und Regression gäbe es noch sehr viel mehr zu sagen, das zu einem sicheren Grundwissen benötigt wird. Bitte lesen Sie deshalb in einem Statistikbuch noch über einseitige und zweiseitige Tests nach, über Fehler erster und zweiter Art und über Trennschärfe (power). Ein gutes Buch ist beispielsweise [MM93]; es gibt zahlreiche andere gute Bücher auf diesem Sektor. Es ist allgemein ein guter Tip, sich im „Ernstfall“ einen Statistiker oder eine Statistikerin zu suchen und sich bei der Datenauswertung beraten zu lassen, falls diese Möglichkeit besteht, denn man kann in der Statistik viel falsch machen oder Möglichkeiten ungenutzt lassen, wenn man sich nicht gut auskennt.

3.4.7 Interne und externe Gültigkeit

Die korrekte Kontrolle zusätzlicher Variablen stellt sicher, daß das Experimentergebnis interne Gültigkeit hat. Das bedeutet, daß die gemachte Beobachtung tatsächlich nur auf die Manipulation der unabhängigen Variablen zurückzuführen ist, das Experiment also einen Kausalzusammenhang beweist und beschreibt (oder auf dessen Fehlen hinweist). Die interne Gültigkeit ist umso höher, je genauer die Versuchsbedingungen beschrieben (also die Werte aller relevanten Variablen) sind und je genauer diese Beschreibungen eingehalten werden.

Eine ganz andere Frage ist die nach der externen Gültigkeit: Läßt sich das Resultat des Experiments auf andere Situationen übertragen als die, die im Experimentaufbau gegeben waren? Auf welche?

Unglücklicherweise widersprechen sich interne und externe Gültigkeit in einem gewissen Maß. Je genauer ich Versuchsbedingungen beschreibe und einhalte, desto enger müssen sie sein. Enge Versuchsbedingungen bedeuten aber, daß das Experiment eine sehr spezielle Aussage liefert, deren Verallgemeinerbarkeit auf andere Umstände zweifelhaft oder zumindest schwer abzuschätzen ist.

Drei Klassen von Gründen stehen möglicherweise einer Gültigkeit der Experimentaussage in anderen Umständen im Wege, nämlich Populationsgültigkeit, ökologische Gültigkeit und zeitliche Gültigkeit. In jeder dieser Klassen gibt es eine Reihe von möglichen Effekten, die die Verallgemeinerung der Beobachtungen des Experiments unzulässig machen.

Populationsgültigkeit: Dies betrifft den Personenkreis, für den die Aussage gültig ist. Im Experiment konnten wir nur über die sogenannte experimentell zugängliche Population verfügen. Die Schlüsse gelten also, interne Gültigkeit vorausgesetzt, für diese. Eigentlich möchte ich jedoch meist eine Aussage über eine viel größere Gruppe von Menschen, die sogenannte Zielpopulation machen. Dies wird fast stets dadurch verhindert, daß die experimentell zugängliche Population erkennbar *nicht* repräsentativ für die Zielpopulation ist. Beispielsweise werden in der Softwaretechnik die meisten Experimente aus Kostengründen mit Studierenden gemacht, die Zielpopulation sind aber professionelle Softwareingenieure. Die Frage, inwieweit eine Übertragung erlaubt ist, kann schwer beantwortet werden, da entsprechende Modelle fehlen — diese zu bilden wäre eine wichtige Forschungsaufgabe.

Ökologische Gültigkeit: Dies betrifft die Umstände des Experiments. Diverse Effekte können eine unbeabsichtigte Abweichung der Umstände im Experiment von den üblicherweise vorhandenen Umständen zur Folge haben. Erstens können mehrere Behandlungen, die derselben Versuchsperson verpaßt werden, miteinander eine Wechselwirkung haben (Mehrfachbehandlungseffekt). Zweitens sind Versuchspersonen eventuell im Experiment besonders motiviert, mehr als im normalen Leben, weil sie wissen, daß sie beobachtet werden (Hawthorne-Effekt). Schließlich können Unterschiede auch dadurch entstehen, daß die erstmalige Konfrontation mit einer gewissen Situation, z.B. praktische Anwendung einer bestimmten Entwurfsmethode, andere Effekte hervorbringt, als man sie bei regelmäßiger Anwendung zu erwarten hätte (Neuigkeitseffekt). Umgekehrt gibt es natürlich auch Effekte in der „wirklichen Welt", die im Experiment nicht vorhanden sind und deshalb die ökologische Gültigkeit beeinträchtigen, z.B. der Erfolgsdruck, unter dem ein angestellter oder selbständiger Softwareingenieur arbeitet, diverse andere Aspekte der Professionalität der Arbeitsweise oder eine Reihe von Umgebungseinflüssen aus dem Arbeitsumfeld, das im Experiment fehlt.

Zeitliche Gültigkeit: Auch die Zeit, zu der das Experiment stattfand, kann ein Grund sein, weshalb sich die Ergebnisse nicht verallgemeinern lassen. Hier gibt es zwei Arten von häufigen

Effekten: lineare und zyklische. Zyklische Effekte haben die Form von saisonalen Schwankungen (z.B. verringerte Konzentrationsfähigkeit durch Frühjahrsmüdigkeit oder am Montagmorgen und Freitagnachmittag) oder von subjektinternen Zyklen (z.B. Wach-Schlaf-Rhythmus). Lineare Effekte sind grundsätzlich eine Folge von Reifung und Historie, lassen sich aber eben nicht über die Dauer des Experiments *hinaus* verfolgen oder kontrollieren.

Welche Folgen können diese Störungen für die Korrektheit der Experimentaussage haben? Es gibt drei Möglichkeiten: Es kann sein, daß alle Beobachtungen gültig bleiben, weil sich verschiedene Effekte gegenseitig neutralisieren. Die zweite Möglichkeit wäre eine moderate oder sogar starke Änderung in den Beobachtungen, die aber die Schlüsse aus dem Experiment gleich läßt (beispielsweise ist immer noch B besser als A, obwohl beide erheblich schlechter abschneiden als im Experiment beobachtet). Die dritte und unangenehmste mögliche Folge besteht darin, daß sich Änderungen ergeben, die die gezogenen Schlüsse ungültig machen (beispielsweise ist B viel schlechter geworden, A jedoch nicht, so daß B nicht mehr besser ist als A).

Es ist fast immer außerordentlich schwierig zu entscheiden, welche dieser drei Möglichkeiten vorliegt. Diese Feststellung sollte uns dazu veranlassen, Wiederholungen von Experimenten mit fast gleichem oder mit geändertem Aufbau als eine wichtige Routine in der Softwaretechnik-Forschung zu fordern und einzuführen. Außerdem sollten wir möglichst versuchen, experimentelle Ergebnisse in Feldstudien zu bestätigen, wo viele der Probleme externer Gültigkeit von allein verschwinden.

3.4.8 Veröffentlichung

Selbst wenn ein Experiment hervorragend geplant, durchgeführt und ausgewertet wurde, ist damit noch längst nicht alles in Butter. Die Wissenschaft lebt nämlich vor allem davon, daß das Wissen sich allmählich *ansammelt*. Dazu ist es aber nötig, daß Forschungsergebnisse dem Rest der Wissenschaftsgemeinde bekanntgemacht werden und dauerhaft verfügbar bleiben. Die klassische Methode hierzu ist die schriftliche Veröffentlichung, zumeist in Form eines Artikels in einer wissenschaftlichen Zeitschrift oder dem Berichtsband einer wissenschaftlichen Konferenz. Die Qualität dieser Veröffentlichung hat maßgeblichen Einfluß darauf, ob und inwieweit das Ergebnis von der Wissenschaftsgemeinde wahrgenommen wird, und ob tatsächlich jemand darauf aufbaut.

Im Abschnitt 9 im Anhang finden sich einige Hinweise darauf, wie zum Schreiben eines guten Artikels vorzugehen ist. Auf einer globaleren Ebene und speziell für experimentelle Arbeiten gibt es eine zusätzliche Richtschnur, die unbedingt einzuhalten ist: **Reproduzierbarkeit**. Es ist absolut entscheidend, daß alle diejenige Information im Artikel vorhanden ist, die andere Forscher bräuchten, um das Experiment zumindest so genau nachzumachen, daß die gleichen grundsätzlichen Ergebnisse herauskommen. Auch wenn die Wiederholung von Experimenten bislang in der Informatik eine Rarität darstellt: andere Disziplinen, zum Beispiel die Physik, sehen sie als eine Selbstverständlichkeit an. Wichtiger noch, die wissenschaftliche Information ist einfach unvollständig, wenn nicht zumindest im Prinzip eine Wiederholung möglich wäre, denn dieser Fall bedeutet, daß nicht alle relevanten Umgebungs- und Versuchsbedingungen, die zu dem vorgelegten Ergebnis geführt haben, bekannt sind, was den Wert des Ergebnisses schmälert, wenn nicht sogar zunichte macht.

3.5 Beispiel: N-Versions-Programmierung

In diesem Abschnitt besprechen wir ein Beispiel für vorbildliche experimentelle Arbeit im Bereich des Software-Engineering. Das vorliegende Experiment von John Knight und Nancy Leveson [KL86] ist hervorragend entworfen, sauber durchgeführt und ausgewertet und leistet einen wichtigen wissenschaftlichen Beitrag.

3.5.1 Forschungsfrage

Die Ausgangsfrage war, ob eine Annahme, die bei der N-Versions-Programmierung gemacht wird zutrifft oder nicht. Als N-Versions-Programmierung bezeichnet man die Methode, mehrere Programme zu erstellen (zum Beispiel drei oder fünf), die dasselbe berechnen, und im Betrieb dann die endgültige Ausgabe mit einem Abstimmungsverfahren aus den Ausgaben der Einzelprogramme zu ermitteln. Hat eines der Programme bei der vorliegenden Berechnung einen Fehler gemacht, so wird es bei der Abstimmung von den anderen überstimmt und der Fehler richtet keinen Schaden an. N-Versions-Programmierung erhöht also die Zuverlässigkeit des Gesamtsystems, *vorausgesetzt, es machen nicht alle Programme stets bei den gleichen Berechnungen Fehler*. Bei der N-Versions-Programmierung wird sogar vom entgegengesetzten Verhalten ausgegangen: Die Versagen aller Programme sind statistisch voneinander unabhängig. Für diesen Fall ergibt sich bei einer Versagenshäufigkeit jedes Einzelprogramms von p eine Häufigkeit des Versagens von zwei Programmen gleichzeitig von p^2 usw. Um dies zu erreichen, fordert die N-Versions-Programmierung eine sehr sorgfältige Problembeschreibung, die frei von Mehrdeutigkeiten und Widersprüchen ist, und eine Implementation durch voneinander unabhängige Entwicklungsteams, die möglichst verschiedene Herangehensweisen benutzen sollen. Diese Vorgehensweise, so die Annahme der N-Versions-Programmierung, führt dann dazu, daß die Versionen nur verschiedene Fehler enthalten (wenn überhaupt welche) und die Versagen dieser Versionen im Betrieb voneinander statistisch unabhängig sind. Die Annahme dieser Unabhängigkeit wurde im vorliegenden Experiment geprüft. Sind die Versagen nicht statistisch unabhängig, so steigt die Zuverlässigkeit eines N-Versions-Programms gegenüber einem einzelnen Programm weniger stark an als erwartet, und die Benutzung der Annahme würde uns in trügerische Sicherheit wiegen.

Dieser Forschungsfrage liegt die Vermutung zugrunde, daß gewisse Teile eines Problems inhärent schwieriger sind als andere und daß deshalb verschiedene Programmierer, selbst wenn sie eine unterschiedliche Ausbildung haben und verschiedene Methoden benutzen, *tendenziell* die gleichen Fehler machen.

3.5.2 Experimentidee

Die Grundidee des vorliegenden Experiments besteht darin, eine große Anzahl von Versionen eines Programms für dasselbe Problem unabhängig voneinander schreiben zu lassen und dann anhand einer großen Zahl zufällig erzeugter realistischer Testfälle zu prüfen, ob gemeinsames Versagen von zwei oder mehr Versionen häufiger vorkommt, als bei Unabhängigkeit zu erwarten wäre, oder nicht.

3.5.3 Experimentaufbau

15 „undergraduate“ Studenten, 8 „graduate“ Studenten und 4 Doktoranden schrieben 27 Versionen eines Programms in Pascal. Alle hatten eine ausreichende Ausbildung in Informatik und

Mathematik und gute Pascal-Kenntnisse. Sie investierten pro Version im Mittel 5 Stunden zum Verstehen der Aufgabe, 16 Stunden zur Implementation und 27 Stunden zum Test; die resultierenden Programme waren zwischen 327 und 1004 Zeilen lang. 18 der Studenten waren von der University of California in Irvine, 9 von der University of Virginia. Dies erlaubt zu prüfen, inwieweit gleiche Ausbildung eine Rolle spielt. Die Versuchspersonen wurden instruiert, nicht über die Aufgabe oder ihre Lösungsansätze miteinander zu diskutieren, um eine unabhängige Bearbeitung der Programme sicherzustellen.

Die Aufgabe bestand darin, aus einem Eingabevektor von Radarreflektionen anhand einer vorgegebenen Menge von Bedingungen zu ermitteln, ob diese Reflektionen von einer anfliegenden Rakete stammen oder nicht. Die Anforderungsbeschreibung zu dieser Aufgabe war sehr sorgfältig aufbereitet und auf Klarheit und Eindeutigkeit geprüft. Sie war schon in früheren Studien verwendet worden, aus denen auch ein äußerst sorgfältig getestetes Programm, das sogenannte Gold-Programm, vorlag, das als fehlerfrei angenommen werden kann und als zusätzlicher Vergleichsmaßstab in der detaillierten Fehleranalyse eingesetzt wurde. Zur Lösung der Aufgabe waren mathematische Kenntnisse erforderlich (Winkelberechnungen) und ein sorgfältiger Umgang mit Genauigkeitsproblemen in den (Fließkomma)berechnungen.

Außer der boolschen Ausgabe hatten die Programme gemäß Anforderungsbeschreibung eine 15×15 boolsche Matrix und einen 15elementigen boolschen Vektor als Zwischenergebnisse abzuliefern, die die einzelnen Bedingungen repräsentieren, insgesamt also 241 Bits. Als Versagen wurde bereits gewertet, wenn zwei Programme in *irgendeinem* dieser Bits nicht übereinstimmten.

Jede Versuchsperson erhielt zum Testen 15 Sätze von Eingabe- und zugehörigen Ausgabewerten. Bevor ein Programm als eine der 27 Versionen akzeptiert wurde, wurde es einem Abnahmetest mit 200 zufällig ausgewählten Fällen unterworfen und zurückgewiesen, wenn dabei irgendein Fehler auftrat. Die Zufälligkeit der Testfälle sollte hier dafür sorgen, daß nicht Programme mit gleichen Fehlern (die bei einer bestimmten festen Menge von Testfällen gerade nicht zum Vorschein kommen) selektiv den Abnahmetest passieren konnten. Diese Prozedur entspricht bei wirklicher Anwendung von N-Versions-Programmierung einem gründlichen Test jeder unabhängig entwickelten Version vor ihrer Auslieferung.

Die akzeptierten Versionen wurden schließlich einem extensiven Test unterworfen, der eine Million Fälle umfaßte. Diese Anzahl repräsentiert die gesamte Lebensdauer des N-Versions-Programms. Die Eingaben wurden (wie auch die übrigen in diesem Experiment) von einem speziell für dieses Problem entwickelten Testdatengenerator erzeugt.

Folgendes mathematische Modell wurde zur Prüfung der Unabhängigkeit verwendet. Sei für die N Versionen 1 bis N die Versagenswahrscheinlichkeit der Version i gleich p_i und seien diese Versagen unabhängig voneinander. Dann ist die Wahrscheinlichkeit P_0 dafür, daß auf einem zufällig gewählten Testfall keine einzige der Versionen versagt

$$P_0 = \prod_{i=1}^{N} 1 - p_i$$

und die Wahrscheinlichkeit P_1, daß genau eine versagt ist

$$P_1 = \sum_{i=1}^{N} p_i \frac{P_0}{1 - p_i}$$

damit ergibt sich für jeden Testfall die Wahrscheinlichkeit P_m, daß zwei oder mehr Versionen zugleich versagen als

$$P_m = 1 - P_0 - P_1$$

Betrachten wir nun die Gesamtheit aller n Testfälle. Sei K die Zahl von Testfällen, auf denen zwei oder mehr Versionen versagen. Diese Größe ist binomialverteilt. Bei unabhängigem Versagen ist dann die Wahrscheinlichkeit dafür, daß K den Wert x hat, gleich

$$P(K = x) = \binom{n}{x} P_m^x \cdot (1 - P_m)^{n-x}$$

Dies ist also die diskrete Dichtefunktion für K. Für große Werte von n nähert sich eine solche Binomialverteilung einer Normalverteilung. Deren Mittelwert ist dann $n\,P_m$ und die Varianz beträgt $n\,P_m(1 - P_m)$. Im vorliegenden Fall ist n sehr groß, deshalb benutzen wir anstelle von K die entsprechende standardnormalverteilte Größe z.

$$z = \frac{K - n\,P_m}{\sqrt{n\,P_m(1 - P_m)}}$$

Die Nullhypothese des Experiments lautet mit diesem Ansatz, daß z ein korrektes Modell für K ist, wenn P_m aus den Beobachtungen geschätzt wird.

3.5.4 Ergebnis

Alle 27 Versionen waren zuverlässiger als 99 Prozent, 23 sogar zuverlässiger als 99.9 Prozent, 6 funktionierten ganz fehlerfrei. Die Versionen waren also nicht weniger zuverlässig, als man das von professionell entwickelten Programmen erwarten kann.

In 1255 der 1000000 Testfälle trat ein Versagen in mehr als einer Version auf. Zusammen mit dem aus den Versagenshäufigkeiten der einzelnen Versionen berechneten Wert für P_m ergibt sich daraus ein Wert für z von 100.51. Die Wahrscheinlichkeit, einen solchen Wert für eine standardnormalverteilte Größe zu erhalten, ist kleiner als ein Milliardstel. Die Hypothese, daß z die Daten modelliert, wird deshalb zurückgewiesen und somit ebenso die Annahme, daß die Versagen der Versionen voneinander unabhängig seien.

Knight und Leveson ziehen die folgenden Schlüsse:

1. Die Annahme der Unabhängigkeit von Versagen in N Versionen eines Programms wird zurückgewiesen mit einer Konfidenz von 99 Prozent. (Tatsächlich sogar wesentlich höher.)
2. Dieses Ergebnis gilt für die im Experiment betrachtete Anwendung. Es ist unbekannt, ob sich das Ergebnis auf andere Anwendungen übertragen läßt.
3. Jedenfalls darf aber die Annahme von Unabhängigkeit nicht mehr ohne weiteres verwendet werden.
4. Das bedeutet nicht, daß N-Versions-Programmierung nicht nützlich ist. Jedoch muß die Modellierung der Zuverlässigkeit verbessert werden.
5. Weitere Forschung ist nötig, um die Ursachen zu bestimmen, aus denen gemeinsame Fehler in Programmen hervorgehen oder verwandte Fehler, die gemeinsame Versagen produzieren.

3.5.5 Diskussion

Diese Arbeit ist wie gesagt ein vorbildliches Beispiel für experimentelles Vorgehen aus folgenden Gründen:

- Gute Auswahl der Forschungsfrage: Die untersuchte Fragestellung ist einerseits so bedeutsam, daß das Ergebnis sehr relevant ist, und zugleich so eng, daß eine sorgfältige und erfolgreiche Bearbeitung möglich bleibt.
- Guter Experimententwurf: Die Autoren haben sorgfältig darauf geachtet, einen Aufbau und eine Durchführung zu erhalten, die Zweifel an der internen Gültigkeit des Experiments möglichst ausräumt. Von der Auswahl der Programmieraufgabe, über die Wahl der Versuchspersonen, bis hin zur Durchführung der Implementationen werden die für N-Versions-Programmierung geforderten Rahmenbedingungen eingehalten. Die Auswertung ist so gründlich, daß die Richtigkeit der Beobachtungen außer Frage steht, der Umfang der Tests so groß, daß eine Zufälligkeit ausgeschlossen werden kann.
- Gute Argumentation: Knight und Leveson argumentieren auch geschickt, um eine hohe externe Gültigkeit ihrer Beobachtungen zu bezeugen. Kritik an der Verwendung studentischer anstatt professioneller Programmierer wird dadurch widerlegt, daß die Zuverlässigkeit der einzelnen Versionen auf professionellem Niveau liegt. Die relativ geringe Größe der Arbeit wird deshalb als unschädlich eingestuft, weil auch in größeren Systemen die N-Versions-Programmierung aufgrund ihrer hohen Kosten meist nur für recht kleine Einzelteile eingesetzt werden kann.
- Gute Aufbereitung: Die Forschungsarbeit wird verständlich dargestellt, sowohl von ihrer Idee, als auch von ihrem Aufbau und ihren Ergebnissen. Es gibt eine gründliche Einführung und übersichtliche Information auch über Details. Mögliche Kritikpunkte werden ausdrücklich angesprochen und diskutiert.
- Vorsichtige Auslegung der Ergebnisse: Knight und Leveson ziehen nur äußerst vorsichtige Schlüsse aus ihren Ergebnissen und sind auch beim Aufzeigen der Konsequenzen sehr bedächtig. So nennen sie nur eine Konfidenz von 99 Prozent, obwohl die Daten wesentlich mehr hergeben, erheben nicht den Anspruch, das Ergebnis zwangsläufig auf andere Probleme übertragen zu können und sprechen der N-Versions-Programmierung auch nicht die Daseinsberechtigung ab.

Es sei hier angemerkt, daß die Arbeit aber auch durchaus nicht perfekt ist. Zum Beispiel ist nirgends erwähnt, wie viele Versionen im Abnahmetest versagt haben und wie oft, was daraufhin jeweils geschah und wie wir das in Bezug auf die Ergebnisse vielleicht zu interpretieren haben.

Aber dennoch: Angesichts dieser Qualitäten stimmt es nachdenklich, wie sehr Knight und Leveson von einigen Befürwortern der N-Versions-Programmierung für ihre Arbeit angegriffen wurden. Diese Angriffe waren so heftig, daß Knight und Leveson sich mehr als 4 Jahre nach dem Erscheinen des Artikels genötigt sahen, eine öffentliche Antwort auf die Kritik zu schreiben, die 1990 in den Software Engineering Notes erschien [KL90]. Diese Antwort ist eine furiose Entkräftung der Angriffe, die zum Teil an der Sache vorbeigingen und zum Teil sogar unwahre Behauptungen über den Aufbau des Experiments oder seine Ergebnisse aufstellten. Ich empfehle jedem, sich diesen Artikel zu Gemüte zu führen; es ist ein Genuß, ihn zu lesen.

3.6 Beispiel: Pseudokode versus Flußdiagramm

Zum Abschluß wollen wir auch noch ein Beispiel für mißlungene experimentelle Forschung vorstellen. Und zwar nicht mißlungen in dem Sinne, daß die erwünschten oder erwarteten Resultate beim Experiment ausgeblieben wären, sondern in dem Sinne, daß der Entwurf des Experiments diese Resultate weitgehend irrelevant macht. Es handelt sich um eine Arbeit von Scanlan [Sca89].

3.6.1 Forschungsfrage

Scanlan untersucht die Verständlichkeit von Flußdiagrammen im Vergleich zu Pseudokode. Im einzelnen stellt er die folgenden Hypothesen auf:

1. Flußdiagramme erzeugen weniger Verständnisfehler.
2. Flußdiagramme geben Studenten mehr Zutrauen in ihr Verständnis eines Algorithmus.
3. Flußdiagramme verringern die Zeit, die zur Beantwortung von Fragen über einen Algorithmus benötigt wird.
4. Flußdiagramme verringern die Anzahl von Malen, die auf den Algorithmus geblickt werden muß.

Alle diese Hypothesen werden für Algorithmen kleiner, mittlerer und hoher Komplexität getestet, um die zusätzliche Hypothese zu prüfen, daß die Nützlichkeit von Flußdiagrammen mit steigender Komplexität zunimmt.

Der vorliegenden Arbeit vorangegangen war eine Serie von Experimenten von Shneiderman, Mayer, McKay und Heller [SMMH77], die die Nützlichkeit von Flußdiagrammen als Hilfe bei verschiedenen Arbeitsvorgängen untersuchten: Programmentwicklung, -verstehen, -modifikation und Fehlersuche. Die Experimente fanden keine statistisch signifikanten Unterschiede in der Leistung zwischen Gruppen, die mit Flußdiagrammen arbeiteten und solchen, die das nicht taten, sondern nur Fortran-Quellcode benutzten.

Scanlan führt dies auf Schwächen im Experimentaufbau zurück: zu kleine Gruppen (je 10 bis 30 Versuchspersonen), zu einfache Algorithmen in den Aufgaben, Fragen nach exakten Ausgabeformaten, welche im Flußdiagramm nicht ersichtlich waren, und vor allem fehlende Zeitmessung, sondern Beurteilung nur nach Korrektheit.

3.6.2 Experimentidee

Gebe einen einfachen, einen mittleren und einen komplexen Algorithmus in Pseudokode und äquivalent als Flußdiagramm an. Stelle für jeden Algorithmus einen Katalog von Verständnisfragen auf. Stelle eine Experimentiervorrichtung her, die es erlaubt, die Anzahl und Dauer der Blicke auf die Algorithmen zu messen und die Antworten auf die Fragen sowie das Zutrauen der Versuchspersonen in diese Antworten festzuhalten.

Wende dann diesen Versuchsaufbau auf viele Versuchspersonen an und prüfe anhand der Beobachtungen die Hypothesen statistisch.

3.6.3 Experimentaufbau

Am Experiment nehmen 82 Versuchspersonen teil, deren Herkunft, Ausbildung, Alter etc. Scanlan genau dokumentiert.

Scanlan tut alles nur denkbare, um keine unkontrollierten Variablen in seinem Experimentaufbau übrigzulassen. Die Algorithmen werden nur ihrem Kontrollfluß nach festgelegt (ELSE-IF-Ketten) und dann im Zufallsverfahren mit Bedingungen aus einer festen Bedingungsmenge und Anweisungen aus einer festen Anweisungsmenge gefüllt, um sicherzustellen, daß keine der Versuchspersonen bereits mit einem der Algorithmen einmal in Kontakt gewesen ist.

Von jedem Algorithmus werden zwei Versionen A und B erzeugt und der Versuchsperson entweder in der Reihenfolge A als Flußdiagramm, dann B als Pseudokode oder umgekehrt B als Flußdiagramm, dann A als Pseudokode vorgelegt, um Lerneffekte zu vermeiden. Die Auswahl zwischen den Reihenfolgen wird ebenfalls zufällig vorgenommen.

Die Flußdiagramme verwenden in ihren Beschriftungen dieselbe Schriftart und -größe wie der Pseudokode, um Unterschiede in der Lesbarkeit auszuschließen.

Die Experimentdurchführung wurde komplett von einem Computer gesteuert, der mittels einer Spezialvorrichtung die auf Papier gedruckten Flußdiagramme oder Pseudokodes sichtbar und unsichtbar machen konnte, um die Messung der Betrachtungszeit und -häufigkeit zu erlauben. Die Fragen wurden auf dem Bildschirm angezeigt, Antworten wurden über die Tastatur eingegeben. Die Versuchsperson konnte mit einem Knopf steuern, ob sie die nächste Frage sehen und die Antwort eingeben wollte, oder den Pseudokode bzw. das Flußdiagramm betrachten; beides zugleich war nicht möglich. Dadurch sollte der Zeitaufwand für diese beiden zeitlichen Teile der Bearbeitung der Aufgaben klar erfaßbar werden.

Um ungleiche Behandlung der Versuchspersonen zu vermeiden, wurden alle Anweisungen in den Übungsphasen vor Experimentbeginn mittels eines Sprachausgabegeräts ebenfalls direkt vom Computer gegeben. Das Experiment fand ferner in einem tageslichtlosen, schallgedämmten Raum statt.

Um die Varianz der Versuchspersonen im Hinblick auf das Verständnis der Experimentalprozedur zu reduzieren, wurde jeder Versuchsperson einige Tage zuvor eine sechsseitige Beschreibung ausgehändigt und zum Experiment nur solche Personen zugelassen, die im ersten oder zweiten Versuch eines Tests mit 10 Fragen über das Experiment mindestens 9 richtig hatten.

Hier ist zur Veranschaulichung der Pseudokode des Algorithmus mittlerer Komplexität in derselben Form, die im Experiment benutzt wurde. Der Gegenstand des Algorithmus sind Behandlungsvorschriften für Gemüse:

```
PROC
  IF GREEN
    THEN
      IF CRISPY
        THEN
          STEAM
        ELSE
          CHOP
      ENDIF
    ELSE
      FRY
      IF LEAFY
        THEN
          IF HARD
            THEN
              GRILL
            ELSE
              BOIL
          ENDIF
        ELSE
          BAKE
      ENDIF
  ENDIF
END PROC
```

dieses Programm enthält 4 Tests (Adjektive) und 6 Anweisungen (Verben). Das einfache Programm ist genau analog aufgebaut und enthält 2 Tests und 4 Anweisungen, das „komplexe“ ist ebenfalls analog und enthält 6 Tests und 8 Anweisungen. Keines der Programme enthält eine Schleife, einen Test, dessen Bedingung aus mehr als einem Wort besteht, oder eine Anweisung, die aus mehr als einem Wort besteht. Das Layout des Pseudokodes war jeweils wie oben.

Zu diesen Programmen hatten die Versuchspersonen Fragen zu beantworten und zwar 12 über das einfache Programm, 36 über das mittlere und 60 über das komplexe. Die Fragen hatten zum Beispiel die folgende Form: „What are the values (true/false/unknown) at all decisions in the algorithm when the vegetable is boiled?“

3.6.4 Ergebnis

Alle Hypothesen konnten bestätigt werden:

1. Die mittlere Zahl richtig beantworteter Fragen war bei Flußdiagrammen stets signifikant höher als bei Pseudokode, z.B. beim komplexen Algorithmus 98.7% gegenüber 91.7%. Die Konfidenz liegt in allen drei Fällen über 99.6%.

2. Das mittlere subjektive Zutrauen der Versuchspersonen in ihre Antworten liegt ebenfalls bei Flußdiagrammen stets signifikant höher als bei Pseudokode, z.B. beim komplexen Algorithmus bei 3.94 Punkte gegenüber 3.71 Punkten. Die Konfidenz für diese Unterschiede liegt in allen drei Fällen über 99.9%.

3. Die mittlere Zeit zur Beantwortung der Fragen liegt bei Flußdiagrammen signifikant niedriger. Beim mittleren Algorithmus mit 6.8 gegen 7.5 Sekunden mit Konfidenz 97% und beim komplexen mit 7.3 gegen 8.7 Sekunden mit Konfidenz 99.98%. Beim einfachen Algorithmus ist der Unterschied nicht signifikant.

4. Die mittlere Anzahl von Betrachtungen des Algorithmus pro Frage ist bei Flußdiagrammen ebenfalls signifikant niedriger, z.B. beim komplexen Algorithmus 0.72 mal gegenüber 0.82 mal. Die Konfidenz liegt in allen drei Fällen über 99.5%.

5. Die Konfidenz in die Überlegenheit der Flußdiagramme ist für den komplexen Algorithmus stets höher als für den einfachen. Also nimmt die relative Nützlichkeit von Flußdiagrammen mit steigender Komplexität zu.

3.6.5 Diskussion

Trotz seines handwerklich so sauberen Aufbaus ist das Experiment ein furchtbarer Mist. Das Problem besteht darin, daß die konkreten Aufgabenstellungen so grotesk unrealistisch sind, daß die Ergebnisse einfach gar nichts aussagen. Hier die Probleme im einzelnen:

1. Die Algorithmen sind allesamt sehr kurz; selbst der „komplexe“ hat nur 14 elementare Einzelteile.

2. Ein Programmstück, das keine Schleife aufweist, stellt allenfalls ein sehr simples Beispiel für einen Algorithmus dar, ist aber jedenfalls atypisch.

3. Der Sinn von Pseudokode besteht darin, ein intuitives Verständnis (mit dem „gesunden Menschenverstand“) für die Arbeitsweise des Algorithmus zu ermöglichen. Bei zufälliger und dementsprechend sinnloser Herstellung der Algorithmen wie in diesem Experiment ist das gar nicht möglich.

4. Welchen Sinn hat überhaupt das Programmverstehen, wenn das Programm selbst keinen Sinn hat?

5. Die gestellten Fragen scheinen alle nur ein mechanistisches Verständnis für die Arbeitsweise des Algorithmus zu verlangen. Die eigentliche Herausforderung beim Programmverstehen besteht aber im Erkennen *tieferer* Zusammenhänge, die den *Zweck* einer Anweisung betreffen anstatt nur den rohen Effekt. Entsprechende Fragen fehlten im Experiment — was kein Wunder ist, denn die benutzten Algorithmen lassen solche Fragen überhaupt nicht zu.

6. Die Fragenkataloge waren viel zu umfangreich, z.B. 60 Fragen für ein Programm mit nur 14 Einzelteilen. Am Ende der Bearbeitung kannten die Versuchspersonen das Programm vermutlich auswendig. Ein Indiz dafür ist die Tatsache, daß zur Beantwortung jeder Frage im Mittel weniger als einmal auf das Programm geschaut wurde! Dies ist vollkommen unrealistisch.

Bitte sowas nicht nachmachen.

3.7 Weitergehende Information

Eine Vielzahl von Zeitschriften und Konferenzen beschäftigen sich mit dem Thema Softwaretechnik-Forschung. Die wichtigsten davon sind wohl die IEEE Transactions on Software Engineering (TSE) und die jährliche International Conference on Software Engineering (ICSE). Wer auf der Suche nach Forschungsfragen oder anekdotischen Aussagen ist, wird beispielsweise in den zahlreichen sehr interessanten und leicht lesbaren Kolumnen der Zeitschrift IEEE Software fündig.

Ein äußerst empfehlenswertes Buch zum Thema Experimentieren stammt von Christensen [Chr94]. Es richtet sich hauptsächlich an Psychologen und ist deshalb nicht unmittelbar auf die Bedürfnisse von Softwaretechnik-Forschern ausgerichtet, ist aber sehr ausführlich, verständlich und interessant geschrieben und übersichtlich aufbereitet. Einige Abschnitte über Experimentation enthält auch das ebenfalls sehr empfehlenswerte Buch von Conte, Dunsmore und Shen [CDS85]. Es hat gegenüber Christensen den Vorteil, daß es speziell für die Softwaretechnik geschrieben ist.

Diejenige Einrichtung, die bisher am meisten im Gebiet der experimentellen Softwaretechnik-Forschung getan hat ist die Zusammenarbeit zwischen dem Software Engineering Laboratory am NASA Goddard Space Flight Center in Maryland (http://fdd.gsfc.nasa.gov/seltext.html) und der Experimental Software Engineering Group an der University of Maryland, College Park (http://www.cs.umd.edu/projects/SoftEng/ESEG).

Auch in Deutschland gibt es seit einiger Zeit eine spezielle Einrichtung für experimentelle Softwaretechnik-Forschung, nämlich das Fraunhofer-Institut für experimentelles Software Engineering (IESE, http://www.iese.fhg.de).

Abschnitt 4

Software- und Systemsicherheit

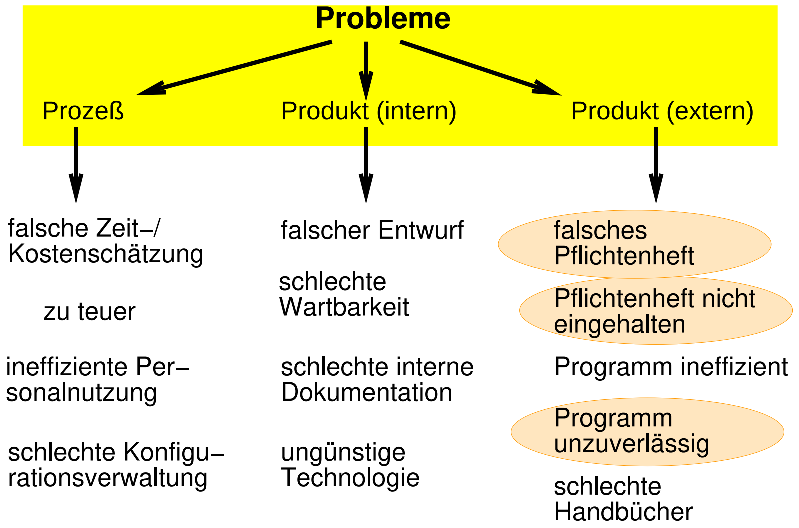

Abbildung 4.1: Von Software- und Systemsicherheit angesprochene Problembereiche

4.1 Einordnung und Ziele

Software ist fast immer Teil eines größeren Systems. Das gilt gleichermaßen für Software, die ausschließlich direkt von Menschen benutzt wird (Büroanwendungen u.ä., hier ist die jeweilige Organisation samt der Menschen das System), und für Software, die direkt in technische Systeme eingebettet ist, um diese zu steuern, seien es nun ein Radiowecker, das ABS eines Autos, eine chemische Fabrikanlage oder die Elektrizitätsversorgung eines ganzen Landes.

Aufgrund der zentralen Rolle, die Software in allen solchen Systemen mittlerweile spielt, liefert sie wesentliche Beiträge zur Sicherheit dieser Systeme — oder den Mängeln solcher Sicherheit. Da viele solche Systeme inzwischen kritische Aufgaben haben und erhebliche Gefahren darstellen können, wenn sie falsch funktionieren, nehmen Fragen der Sicherheit einen hohen Stellenwert ein, auch wenn das mancherorts bisher noch kaum gesehen wird.

Eine wichtige Beobachtung in diesem Zusammenhang lautet, daß eine getrennte Betrachtung von Softwaresicherheit und Systemsicherheit nicht sinnvoll ist. Erstens kann eine klare Zuweisung eines auftretenden Sicherheitsproblems nur auf das Konto der Software oder nur auf das

Konto des Restsystems ohnehin oft nicht vorgenommen werden und zweitens wäre das auch gar nicht wünschenswert, denn Software und Restsystem sollten nach Möglichkeit stets versuchen, die Schwächen des jeweils anderen Teils möglichst weitgehend auszugleichen. Beispiel: Betrachten wir ein System aus Bestellsoftware und den Menschen die sie benutzen (Besteller und Lieferant). Angenommen, ein Bäcker bestellt wöchentlich per Computer sein Mehl, normalerweise eine bis eineinhalb Tonnen und vertut sich einmal bei der Eingabe mit den Nullen und bestellt 14 Tonnen. Die Software sollte eine Plausibilitätsprüfung haben, die diesen Fall als verdächtig einstuft und den Bäcker rückfragt, ob er das ernst meint. Der Softwaresystemteil sollte also gegen Fehler des menschlichen Systemteils vorsorgen. Kommt eine solche Bestellung, etwa wegen eines Softwarefehlers, beim Lieferanten an, sollten sich beispielsweise die Leute im Lager dafür zuständig fühlen, den Wert verdächtig zu finden und nachzuforschen, ob er korrekt ist. Der menschliche Systemteil sollte also gegen Fehler im Softwaresystemteil vorsorgen. Ein gut entworfenes System hat immer diese Eigenschaft in möglichst vielen seiner Teile mit möglichst vielen möglichst verschiedenen und möglichst einfachen Maßnahmen gegen möglichst viele Arten von Fehlern vorzubeugen.

Der einzige Aspekt von Softwaresicherheit, den separat zu betrachten sich lohnt, ist die Frage nach der Zuverlässigkeit, also: Wie oft treten im Softwarebetrieb Abweichungen des Softwareverhaltens von der Spezifikation auf und wie schwerwiegend sind diese? Wir werden uns in diesem Abschnitt jedoch auf die Sicht von Gesamtsystemen konzentrieren.

Aus den obigen Betrachtungen ergeben sich zwei Forderungen:

- Systeme (und Software) müssen sicher sein!
- Denke in Systemen!

4.2 Begriffe

Im Zusammenhang mit Systemsicherheit gibt es eine Reihe von Begriffen, die im Interesse einer genaueren Sprechweise hier definiert werden sollen. Bei der Verwendung der Begriffe ist zu beachten, daß die Terminologie recht uneinheitlich ist und im Einzelfall nochmals genau vereinbart werden muß. Die Begriffe stammen wie auch der überwiegende Teil des restlichen Materials aus den Büchern [Lev95] und [Neu95].

Sicherheit (safety): Zustand der Abwesenheit von Unfällen (evtl. stärker: Abwesenheit von Risiken unakzeptabler Höhe).

Unfall (accident): Unerwünschtes und ungeplantes (aber nicht unbedingt unerwartetes) Ereignis, das zu einem gewissen Verlust führt.

Risiko (risk): Als Quantität die Wahrscheinlichkeit eines Unfalls mal dessen Schadenshöhe. Die Wahrscheinlichkeit kann man evtl. zerlegen in die Wahrscheinlichkeit und Dauer einer Gefahr und der Wahrscheinlichkeit des Eintretens der restlichen Unfallbedingungen während dieser Dauer. Häufig ist eine quantitative Betrachtung von Risiken jedoch nicht angemessen, weil sich insbesondere die Wahrscheinlichkeit nicht genau genug berechnen läßt.

Versagen (failure): Ein untolerierbares, eventuell gefährliches Fehlverhalten eines Systems. Hierunter fallen nur Abweichungen von der Spezifikation.

Zuverlässigkeit (reliability): Als Qualität das Funktionieren eines Systems gemäß der Anforderungen unter allen vorgesehenen(!) Bedingungen. Als Quantiät die Wahrscheinlichkeit des Nichtversagens in einem bestimmten Zeitraum.

Robustheit (robustness): Funktionieren ohne Unfall auch unter unvorhergesehenen Bedingungen.

Fehler (error): Ereignis bei der Herstellung eines Systems, das zum Entstehen eines Mangels führt. Manchmal auch: Zustand der Abweichung von einem gewünschten Zustand.

Gefahr (hazard): Eine Menge von Bedingungen für ein System, die gemeinsam mit anderen Bedingungen in der Umgebung unweigerlich zu einem Unfall führen.

Vorkommnis (incident, near miss): Ein Ereignis ohne Verlust, das aber ein Potential für einen Verlust unter anderen Umständen hat.

Mangel, Defekt (fault, defect): Struktureller Fehler in einem System. Defekte sind eine Grundlage für Fehlverhalten.

Verletzlichkeit (vulnerability): Schwachstelle im Entwurf eines Systems, an der Mängel besonders leicht zu Versagen führen können.

Schutz (security): Widerstandsfähigkeit des Systems gegen absichtliche Angriffe. Dies ist meist ein Teilaspekt von Sicherheit.

Schwaches Glied (weak link): Schwache Glieder in einem System sind Ereignisse, deren Eintreten die Funktionsfähigkeit des Systems beeinträchtigt oder zerstört. Solche schwachen Glieder lassen sich grundsätzlich nicht vermeiden. Das Ziel des Systementwurfs muß es stattdessen sein, dafür zu sorgen, daß immer möglichst viele, möglichst unwahrscheinliche Einzelereignisse zu jedem schwachen Glied gehören. Besonders gefährlich sind schwache Glieder, die nur aus einem Ereignis bestehen. Problematisch bei der Analyse sind Korrelationen zwischen dem Eintritt der verschiedenen Ereignisse zusammengesetzter schwacher Glieder, die oft schwer abzuschätzen sind und manchmal schlicht übersehen werden (siehe das Beispiel der vermeintlich sieben getrennten Datenleitungen auf dem gleichen Kabel weiter unten).

Die Begriffe Fehler, Defekt und Versagen werden ähnlich auch im Softwarebereich benutzt: Ein Fehler (Ereignis) führt zu einem Defekt (Softwarestruktur) in der Software. Wird der Defekt ausgeführt, kann er, muß aber nicht, zu einem Versagen (Ereignis) führen.

4.3 Ansätze

4.3.1 Grundregeln

Es gibt eine Reihe von zentralen Erkenntnissen und Regeln für Systemsicherheit, die wir nun besprechen wollen:

1. Baue Sicherheit von vornherein ein. Versuche sie nicht erst nachzurüsten. Systeme, die nicht von Anfang an für Sicherheit entworfen werden, erreichen nur selten eine hohe Sicherheitsstufe, weil zu viele ihrer strukturellen Eigenschaften der Sicherheit entgegenstehen.

2. Betrachte das System als Ganzes, nicht seine Teile. Viele Sicherheitsprobleme ergeben sich erst durch das Zusammenwirken der Teile in unvorhergesehenen Weisen. Außerdem erzeugt eine separate Betrachtung der Teile die Tendenz, daß die jeweiligen Verantwortlichen der Teile die Verantwortung für die Sicherheitsprobleme hin- und herschieben.

3. Nicht nur Systemversagen ist eine Gefahr. Unfälle können auch dann geschehen, wenn das System sich genau gemäß seiner Spezifikation verhält. Zuverlässigkeit ist nicht gleich Sicherheit. Dies ist eine besonders wichtige Lehre für Softwarebauer.

4. Verlaß Dich nicht auf Erfahrungen und Standards. Für Systemsicherheit gibt es zwar viele vorgefertigte Regeln und Maßnahmen, diese können die Sicherheit aber nicht garantieren. Analysiere das System!

5. Verwende qualitative statt quantitative Methoden. Der Versuch, die Sicherheit zu quantifizieren führt dazu, die Kräfte auf das falsche Ziel zu konzentrieren. Sicherheit läßt sich nicht quantifizieren, aber sehr wohl läßt sich mit Augenmaß abschätzen, ob eine gewisse Maßnahme ein günstiges Verhältnis von Aufwand zu (möglicher) Wirkung hat oder nicht.

6. Erkenne, daß Abwägungen nötig sind und Konflikte auftreten. Sicherheit ist niemals total. Bei der Herstellung sicherer Systeme treten Beschränkungen aufgrund der Kosten oder der nötigen Eingriffe in die Umgebung auf, die gegen den möglichen Nutzen der Maßnahmen abgewogen werden müssen.

7. Beachte die Vorrangfolge beim Umgang mit Gefahren:

- Gefahr eliminieren („intrinsische Sicherheit"),
- Gefahr reduzieren,
- Gefahr beherrschen (passiv, aktiv),
- Schaden verringern.

Alles in allem kann man sagen, daß die Herstellung von Systemsicherheit gleichermaßen Augenmaß, Sorgfalt und Kreativität erfordert und eine sehr schwierige und herausfordernde Aufgabe ist. Eine Aufgabe zudem, die aufgrund sozialer Effekte ziemlich schwierig gut zu erledigen ist, wie im nachfolgenden Abschnitt zu sehen sein wird.

4.3.2 Hierarchische Sicht von Unfällen

Zur Analyse von Unfällen empfiehlt sich eine dreischichtige, hierarchische Sichtweise:

1. **Mechanismen:** konkreter Hergang beim Unfall, Ereignisse und ihr Zusammenwirken. Diese Ebene dient dazu, den Unfall zu beschreiben.
2. **Bedingungen:** unfallrelevanter Zustand des Systems und seiner Umgebung bei Unfallbeginn. Diese Ebene dient dazu, den Unfall zu verstehen.
3. **Urgründe (root causes):** allgemeine Umfeldbedingungen, die zu (2) geführt haben. Diese Ebene dient dazu, die Unfallklasse zukünftig vermeiden zu können.

Die Urgründe sind fast immer menschlicher Natur, meist eine Form von Nachlässigkeit oder, seltener, von Unwissen. Viele schwere Unfälle sind auf den Urgrund eines mangelnden Risikobewußtseins zurückzuführen. Das drastischste Beispiel hierfür ist wohl der Untergang der Titanic [Lev95, S.56f].

Hier ist die Geschichte dieses Untergangs; bitte Satz für Satz in Ruhe würdigen: Die Titanic war das größte und sicherste Passagierschiff, das je gebaut wurde. Es war so konstruiert, daß der Rumpf aus 16 Kompartments bestand, von denen bis zu 4 geflutet werden konnten, ohne daß das Schiff sinken würde. Ein gleichzeitiges Vollaufen von 4 unabhängigen Rumpfabteilungen hatte es in der Geschichte der Seefahrt noch niemals gegeben. Lloyds London stellte aus diesem Grund der Titanic ein Unsinkbarkeitszertifikat aus. Der Kapitän sagte einer ängstlichen

Passagierin zu Beginn der Reise: „Nicht einmal Gott könnte dieses Schiff versenken.“ Die Jungfernfahrt des Schiffes war ein Versuch, den Geschwindigkeitsrekord für Atlantiküberquerungen zu brechen und das dafür ausgelobte „blaue Band“ zu erringen. Deshalb fuhr das Schiff trotz tiefer Nacht mit hoher Geschwindigkeit, obwohl mehrere unabhängige Nachrichten eingegangen waren, die vor Eisbergen um durchquerten Gebiet warnten. Bei der Kollision mit einem Eisberg wurde aufgrund der hohen Geschwindigkeit und eines spät eingeleiteten Ausweichmanövers der Rumpf auf einer Länge von fast 100 Metern aufgeschlitzt. Dabei wurden 5 der Rumpfabteilungen leckgeschlagen und liefen schnell voll. Nach der Kollision gab die Schiffsbesatzung zunächst Entwarnung und empfahl den Passagieren, in ihren Kabinen zu bleiben. Als etwas später die Evakuierung des Schiffes eingeleitet wurde, stellten sich zwei Probleme heraus. Erstens waren die normalerweise auf einem Schiff üblichen Sicherheitsübungen mit der Besatzung unterlassen worden, so daß die Besatzungmitglieder nicht wußten, wo ihr Platz und was ihre Aufgabe gewesen wäre, um das Schiff schnell zu evakuieren. Zweitens waren nicht genügend Rettungsboote vorhanden, um alle an Bord befindlichen Menschen aufzunehmen, weil die zu jener Zeit gültigen Sicherheitsvorschriften (die durchaus eingehalten worden waren) kein Schiff dieser Größe vorgesehen hatte und deshalb für die Titanic nicht angemessen waren. Bei der Evakuierung brach ein völliges Chaos aus und das Schiff sank schnell. Mehr als 1500 Menschen starben.

Ähnliche, wenn auch weniger dramatische Beispiele ließen sich zuhauf anführen. Wie wir daran lernen können, ist Risikobewußtsein manchmal paradox: Sein Vorhandensein führt quasi zu seiner eigenen Abschaffung; die Vorsicht und Aufmerksamkeit schläft auf dem Ruhekissen guter Sicherheitsvorkehrungen ein oder macht sogar entgegengesetztem Leichtsinn Platz.

Maßnahmen der Systemsicherheit sollten eigentlich immer auf den Ebenen 2 *und* 3 ergriffen werden. Ebene 2 umfaßt im weiteren Sinne technische Lösungen, Ebene 3 betrifft organisatorische und psychosoziale Vorkehrungen. Leider ist denjenigen, die die Risikoanalyse zur Steigerung der Systemsicherheit durchführen, der Zugriff auf die Ebene 3 oftmals weitgehend versperrt. Dies deutet auf eine ungünstige Organisationsform hin und sollte geändert werden.

4.3.3 Allgemeines Vorgehen

Leider kann man kein algorithmisches Verfahren angeben, wie es gelingt, ein System sicher zu konstruieren. Jedoch lassen sich einige grundlegende Arbeitsschritte identifizieren, die dazu betrieben werden sollten. Wir gehen davon aus, daß eine halbwegs konkrete Vorstellung davon vorliegt, was das zu konstruierende System tun soll.

Gefahrenbestimmung: Als erstes müssen mögliche Sicherheitsprobleme erkannt werden. Darüber gibt es unten einen eigenen Abschnitt.

Risikoanalyse: Nach Bestimmung der Gefahren muß zu jeder das entsprechende Risiko abgeschätzt werden. Zu jedem Risiko ist abzuwägen, ob überhaupt und in welchem Umfang Maßnahmen ergriffen werden sollen, um es zu vermindern oder zu beseitigen.

Entwurf: Dann müssen diese Gegenmaßnahmen erfunden und umgesetzt werden. Dabei gilt die oben bereits erwähnte Vorrangfolge von Beseitigung, Reduktion, Beherrschung und Schadensverminderung.

Entwurfssicherheitsprüfung: Das fertig entworfene Gesamtsystem sollte dann nochmals einer Sicherheitsanalyse unterzogen werden, die ähnlich vorgeht wie die ursprüngliche Gefahrenbestimmung. Diesmal jedoch können viele Überlegungen auf wesentlich konkreterem Grund vorgenommen werden. Werden dabei noch unzureichend beherrschte Risiken entdeckt, muß eine weitere Iteration von Entwurf und Prüfung vorgenommen werden.

Bau: Nun erst wird das System realisiert.

Abnahmesicherheitsprüfung: Am konkret realisierten System sollten abschließend noch alle jene Aspekte auf ihre Sicherheit geprüft werden, die im Entwurf aufgrund fehlender Information noch nicht vollständig analysiert werden konnten. Dies muß je nach System möglicherweise kontinuierlich oder periodisch im laufenden Systembetrieb wiederholt werden, weil sich die relevanten Bedingungen im Betrieb ändern können (z.B. durch Alterung oder durch Evolution des Betriebsumfelds).

4.3.4 Gefahrenbestimmung

Der kritischste Schritt bei der Verbesserung der Sicherheit ist die Bestimmung der Gefahren. Eine Gefahr die nicht erkannt wurde, kann auch nicht gezielt bekämpft werden. Es ist deshalb von außerordentlicher Bedeutung, möglichst viele gefährliche Situationen und Einflüsse auf ein geplantes System schon in der Konstruktionsphase zu erkennen und zu berücksichtigen.

Hier ist eine allgemeine Vorgehens- und Prüfliste zur Gefahrenidentifikation, siehe auch [Lev95, S. 296]:

1. Nutze historische Daten von Beteiligten oder aus externen Quellen: Sicherheitserfahrungen, gelernte Lektionen, Problemberichte, Vorkommnisberichte, Unfallberichte.
2. Nutze veröffentlichte Gefahrenchecklisten, Standards, Vorgehensbeschreibungen.
3. Untersuche Energiequellen, Energieflüsse im System, gefährliche Materialien (brennbar, unter Druck, explosiv, giftig, Materialunverträglichkeiten) und die Maßnahmen zu ihrer Kontrolle.
4. Untersuche die Mensch-Maschine-Schnittstelle.
5. Untersuche die Systementwicklung (insbes. SW-System):
   - Systemkonzept, Annahmen und Modelle korrekt?
   - Anforderungsanalyse vollständig und korrekt?
   - Spezifikation und Entwurf gut?
   - Implementation und Test sorgfältig?
   - Infrastruktur (Betriebssysteme, Übersetzer, etc.) geeignet und zuverlässig?
   - Weiterentwicklung bedrohlich (Fehlereinfügung, Komplexitätszunahme, Probleme bei Versions- oder Technologiewechsel)?
6. Untersuche alle normalen und besonderen vorgesehenen Betriebszustände, insbesondere Übergangszustände wie Systemstart, -neustart, -stop, -test, -zusammenbruch, -wartung, -reparatur.
7. Untersuche denkbare ungewöhnliche Umstände des Systembetriebs
   - Unerwartete Umwelteinflüsse: Extremes Wetter, Tiere, Erdbeben, Fluten, kosmische Strahlung etc.
   - Infrastrukturversagen: Stromausfall, Ausfall von Klimaanlagen etc.
   - Hardwareversagen: endgültig oder vorübergehend.

- Mißbenutzung: Fehlinstallation, Fehlbedienung, Benutzung zu falschem Zweck, Langsamkeit, Mißachtung von Meldungen, vorsätzlicher Mißbrauch.

8. Beachte eine langfristige Perspektive: Änderungen am System, Weiterentwicklungen des technischen Umfelds, Änderungen im organisatorischen und im sozialen Umfeld.
9. Gehe Schritt für Schritt durch den Gesamtprozeß des Systems und überlege, was schiefgehen könnte, wie man das grundsätzlich vermeiden kann und was zu tun ist, wenn der schlimmste Fall eintritt.

4.4 Probleme und Hindernisse

Aus der Perspektive der Hauptschwierigkeiten und genügend großer Distanz betrachtet, hat die Systemsicherheit große Ähnlichkeit mit der Softwaretechnik als Ganzes: die Hauptprobleme sind erstens, es gut zu machen und zweitens, den Versuch dazu überhaupt durchzusetzen. Allerdings liegt die Schwierigkeit bei der Sicherheit im Gegensatz zur Softwaretechnik doch eher auf dem zweiten Punkt. Woran liegt das? Dafür gibt es eine Reihe von miteinander verwandten Gründen:

1. Sicherheit kostet viel Geld, von dem man nie genau weiß, ob man es wirklich investieren muß.
2. Risikoanalyse und Sicherheitsstreben tragen den Geruch des Negativen und sind deshalb oft schwer vermittelbar.
3. Das Vorhandensein irgendwelcher Sicherheitsmaßnahmen wird oft tendenziell mit vollständiger Sicherheit gleichgesetzt.
4. Auch die gegenteilige Haltung kommt vor: „Egal was wir machen, wir kriegen es ja doch nie sicher.“
5. Sicherheitsmaßnahmen sind nicht sexy. Sie wirken nicht so imponierend wie funktionelle Leistungen des Systems und sind deshalb schwerer zu verkaufen.
6. Bei gewissen Arten von Risiken trifft man manchmal auf die Haltung „Sicherheit ist was für Spießer“.

4.4.1 Risiken der Risikoanalyse

In einem gewissen Sinne kann man eine Risikoanalyse eigentlich nur falsch machen, denn es gibt nur folgende drei Möglichkeiten, und alle drei sind ungünstig:

Risiken werden überschätzt: Vermeidbare Vermeidungskosten treten ein, Mittel und Aufmerksamkeit werden von anderen wichtigen Bereichen abgezogen, evtl. werden sinnvolle Möglichkeiten ungenutzt gelassen. Dies ist die unsichtbare Niederlage.

Risiken werden unterschätzt: Vermeidbarer Schaden tritt ein. Dies ist die klassische Niederlage.

Risiken werden richtig eingeschätzt: Oft kann später der Wert der Risikoanalyse nicht bewiesen werden, weil das Eintreten des Unglücksfalls aufgrund der Vorsorge ja gerade vermieden wurde. Das kann die *nächste* Risikoanalyse negativ beeinflussen. Dies ist die indirekte Niederlage.

Aufgrund dieser Effekte ist es ungeheuer schwer, das richtige Maß von Sicherheitsanstrengungen zu finden und gegen Angriffe von Kritikern zu verteidigen.

4.4.2 Software-Mythen für sichere Systeme

Hinter den Schwierigkeiten beim Durchsetzen von Sicherheitsanstrengungen stecken, soweit es die Software angeht, einige falsche Vorstellungen über die Eigenschaften von Software und Systemen, die Software verwenden. Dies sind Varianten bekannter Software-Mythen in der speziellen Fassung für sichere Systeme.

1. Computer sind billiger als die analogen oder elektromechanischen Systeme, die sie ersetzen.
   Das stimmt nicht unbedingt, wegen hoher Kosten für sichere Software.

2. Software ist leicht zu ändern.
   Aber schwer korrekt zu ändern.

3. Computer sind zuverlässiger als die Geräte, die sie ersetzen.
   Sie haben zwar weniger Ausfälle durch Alterung oder Verschleiß, aber häufigere Versagen durch Softwaredefekte. Zum Beispiel wurden in der Software des Space Shuttle im Zeitraum von 1980 bis 1995 immerhin 16 lebensgefährdende SW-Defekte entdeckt, obwohl die Software mit den höchsten nur denkbaren Qualitätsansprüchen entwickelt wurde.

4. Eine Verbesserung der SW-Zuverlässigkeit steigert die Sicherheit.
   Korrollar: formale Verifikation kann alle Fehler entfernen.
   Beides ist leider nicht notwendigerweise richtig, denn viele Sicherheitsprobleme stammen aus falscher Anforderungsdefinition, falscher Spezifikation oder falschem Systementwurf.

5. SW-Wiederverwendung steigert die Sicherheit.
   Kann sein. Sie kann sie aber auch senken, weil die Komplexität der Interaktionen nicht überschaut wird. Diese Problematik wird seit einigen Jahren intensiv im Zusammenhang mit objektorientierter Software diskutiert. Diese ist zwar durch die Vererbungstechnik vielversprechend im Hinblick auf Wiederverwendung, erzeugt aber ebenfalls durch die Vererbung zugleich äußerst komplexe Wechselwirkungen zwischen Systemteilen, die sehr schwer zu kontrollieren sind.

4.5 Beispiele für Sicherheitsprobleme

In diesem Abschnitt wollen wir einige konkrete Beispiele für Sicherheitsprobleme in Systemen besprechen, an denen Software beteiligt war. Diese Liste hat mehr als einen Zweck:

1. Sie gibt einen Überblick über konkrete Schwierigkeiten oder Unfälle, die als Beispiele zur Veranschaulichung oder bei der Argumentation für Systemsicherheit dienen können.

2. Sie vermittelt in Ansätzen ein Gefühl dafür, was alles schiefgehen kann und schult damit das Mißtrauen, das jeder gute Sicherheitsingenieur braucht.

3. Sie ist recht unterhaltsam und liefert Stoff dafür, abends in der Kneipe Nichtinformatiker zu beeindrucken.

4. Sie gibt einen (nicht repräsentativen) Überblick darüber, welche Arten von Fehlern es sind, die zu Problemen führen. Diese sind nämlich nur relativ selten auf Kodierfehler zurückzuführen, was viele Informatiker überraschend finden.

4.5.1 Fehlertaxonomie

Für den letzteren Zweck führen wir eine sehr einfache Taxonomie von Fehlern ein, die in Abbildung 4.2 dargestellt ist. Diese Taxonomie beschreibt die Herkunft von Fehlern. Sie unterteilt

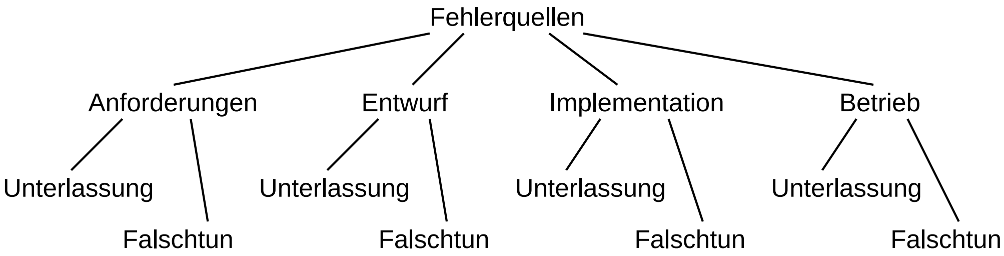

Abbildung 4.2: Herkunftsarten von Fehlern, die zu Sicherheitsproblemen führen

zunächst in die Bereiche der Systementwicklung, nämlich die Bestimmung und Beschreibung der Anforderungen an das System, den Systementwurf (einschließlich Entwurf der äußeren Softwareschnittstellen), die Herstellung (einschließlich Entwurf der internen Softwareschnittstellen, also dem klassischen Softwareentwurf) und den Systembetrieb. In jeder dieser vier Abteilungen unterscheiden wir ferner zwei Fehlerarten, nämlich Fehler durch Unterlassung (*omission*) und Fehler durch Falschtun (*commission*). Leider lassen sich Fehler oft nicht eindeutig zwischen Anforderungen und Entwurf oder auch zwischen Entwurf und Implementation zuordnen, weil nicht ganz klar ist, wo eine gewisse Entscheidung hätte getroffen werden sollen; dennoch ist die Betrachtung recht hilfreich.

4.5.2 Beispiele: Entwurfsprobleme

Dezember 1986, ARPAnet-Ausfall: Es gab 7 Verbindungen im Arpanet (heutiges Internet) zwischen Neu-England und dem Rest des ARPAnet. Die Anbindung von Neu-England wurde deshalb für fehlertolerant gehalten. Alle 7 Verbindungen fielen gleichzeitig aus, als ein Glasfaserkabel bei Bauarbeiten beschädigt wurde, weil alle 7 über das gleiche Kabel geführt wurden [Neu95, S. 14]. Entwurfsfehler, Unterlassung.

Osprey Schwenkrotorflugzeug abgestürzt: Einer von fünf Prototypen des Bell Osprey stürzte beim Erstflug kurz nach dem Start ab. Zwei der drei Rollraten-Gyrosensoren (dreifache Redundanz) waren falschherum verkabelt und überstimmten den dritten. Zwei weitere Prototypen flogen erfolgreich mit einem genauso falsch verkabelten Sensor, ohne daß die Piloten oder das Wartungspersonal eine Warnung bekommen hätte [Neu95, S. 37]. Entwurfsunterlassung und Implementationsfalschtun.
Diese Geschichte ist ein eindrückliches Beispiel dafür, wie gute Sicherheitsmaßnahmen (hier: die dreifache Redundanz der Sensoren) die Vorsicht in gefährlicher Weise untergraben können (hier: keine Warnvorrichtung vorgesehen für den Fall, daß die Sensoren nicht übereinstimmen).

April 1990, U-Bahn ohne Fahrer: Eine U-Bahn-Tür schloß nicht richtig. Der Fahrer stieg aus, um das zu beheben. Als die Tür schließlich schloß, fuhr der Zug ohne den Fahrer los, weil jener den Losfahr-Knopf mit Klebeband festgeklebt hatte. Das Losfahren wurde immer automatisch bis zum Türenschließen verhindert, das Anhalten war ebenfalls automatisch. Das Verhalten des Fahrers verletzte die Betriebsregel, niemals die Führerkabine eines automatischen Zuges zu verlassen. Ein ähnliches Ereignis gab es im Dezember 1993. [Neu95, S. 53f]. Entwurfsunterlassung (das Festkleben hätte der Entwerfer vorhersehen sollen) und Betriebsfalschtun.

1985–1987, Therac-25: Der Therac-25 war ein computergesteuerter Elektronenbeschleuniger zur Strahlentherapie, Nachfolger des nicht computergesteuerten Therac-20. Das System hatte drei schwere Mängel:

- Sicherheitsprüfungen im Programm wurden durch einen Softwarefehler bei jeder 64. Benutzung ausgelassen (wenn ein 6-bit Zähler Null wurde).
- Behandlungsanweisungen konnten mittels Editieren am Bildschirm so teilgeändert werden, daß die Maschine für die nächste Behandlung nicht den gewünschten Zustand einnahm (nämlich Niederintensität).
- Mehrere Sicherheitsverriegelungen, die beim Vorgängermodell Therac-20 in Hardware realisiert waren, wurden nicht übernommen, sondern durch Software ersetzt.

Anstatt 80 bis 200 rad erhielten mehrere Patienten Strahlungsdosen von 8000 bis 25000 rad; es gab mehrere Tote und Schwerverletzte. Als erste Korrektur wurden die Cursortasten zum Editieren mechanisch unbrauchbar gemacht. Das Gerät wurde danach vom Markt genommen [Neu95, S. 68ff]. Entwurf und Implementation, Falschtun.

Tödliche Herzschrittmacher: Mehrere Menschen starben, weil ihre Herzschrittmacher ungewollt von außen beeinflußt wurden. Ein 62jähriger wurde mit Mikrowellen gegen Arthritis behandelt. Die Strahlung setzte seinen Herzschrittmacher auf 214 Schläge pro Minute. Ein ähnlicher Fall trat durch die Signale einer Diebstahlsicherung in einem Geschäft ein. In einem dritten Fall wurde der Herzschrittmacher versehentlich vom Arzt ausgeschaltet, als er ihn umprogrammieren wollte [Neu95, S. 70f]. Anforderungen/Entwurf/Implementation?, Unterlassung.

März 1979, 5 Kernkraftwerke abgeschaltet wegen Simulationsfehler: Lange nach ihrem Bau wurde bei 5 amerikanischen Kernkraftwerken entdeckt, daß das Simulationsprogramm Shock II, mit dem die Erdbebenfestigkeit der Konstruktion geprüft worden war, einen Fehler hatte. Alle Kraftwerke wurden abgeschaltet. Im Simulationsprogramm war an einer Stelle eine einfache Summe anstatt einer Summe von Absolutwerten berechnet und weiterverwendet worden [Neu95, S. 79f]. Entwurf, Falschtun.
Es erhebt sich außerdem die Frage, inwieweit die Anforderungen hier überhaupt akzeptabel waren, nämlich daß die Kraftwerksbauten den stärksten jemals in der Baugegend aufgetretenen Erdbeben *gemäß einer Simulation* widerstehen müssen. Da ein Experiment nicht durchführbar ist und die Simulation nicht validiert werden kann, stellt diese Anforderung in sich bereits ein Risiko dar.

1992, Hundertjährige: Mary Bandar bekam 1992 eine Einladung zum Besuch des Kindergartens, gemeinsam mit anderen, den Kindern des Jahrgangs 1988. Allerdings war sie schon 104. Der Bediener hatte zwar korrekt 1988 für die Datenbankanfrage eingetippt, das Programm davon jedoch nur die letzten zwei Stellen benutzt. G.C. Blodgetts Autoversicherung wurde dreimal so teuer wie zuvor, als er 101 Jahre alt wurde. Das Programm hatte sein Alter als 1 interpretiert und das paßte auf die Definition eines Teenagers als "jemand unter 20". Wir sind alle sehr gespannt, was nach Sylvester 1999 alles passieren wird... [Neu95, S. 87]. Anforderungen/Entwurf/Implementation, Unterlassung.

April 1993, Kühlunfall im Stahlwerk: Die Computersteuerung eines deutschen Stahlwerks benutzte die Braunschweiger Normalzeituhr als ständigen Zeitgeber und verkürzte das Kühlen einer Ladung Stahl um eine Stunde, nachdem die Zeit von 1:59 auf 3:00 umgesprungen war, weil die Sommerzeit begonnen hatte. Die unbeabsichtigte Weiterverarbeitung von noch geschmolzenem Stahl richtete erheblichen Schaden im Werk an [Neu95, S. 89]. Entwurf, Unterlassung.
Hier treffen zwei Fehler aufeinander: Nicht nur wurde es unterlassen, die Zeitumstellung zu

berücksichtigen, sondern es wird auch nicht das geprüft, auf das es eigentlich ankommt, nämlich die Temperatur des Stahls, sondern ein Surrogat herangezogen, nämlich die Kühlzeit. Es hätte eine Temperaturprüfung stattfinden sollen, die den Unfall verhindert hätte.

Fortran auf Deutsch: Manche Fortran-Compiler erlauben die Ausgabe an *units* und an *devices*. Unit 6 ist die Standardausgabe, device 6 ist die Platte. Diverse deutsche Fortran-Benutzer überschrieben versehentlich ihre Platte, weil in Deutsch sowohl *unit* als auch *device* mit *Einheit* übersetzt werden und sie deshalb die beiden Konzepte verwechselten [Neu95, S. 93]. Tja, welcher Klasse soll man das nun zuordnen?

TENEX Paßwortprüfung: Bei der *connect* Prozedur des TENEX Betriebssystems wurde das eingegebene Paßwort zeichenweise sequentiell mit dem korrekten Paßwort verglichen und bei Auftreten eines falschen Zeichens abgebrochen. Das ermöglicht ein zeichenweises Knacken des Paßworts, indem die Prozedur aus einem Programm heraus aufgerufen wird, wobei das zu testende Paßwort gezielt so im Speicher liegt, daß das nächste unbekannte Zeichen direkt vor einer Seitengrenze steht. Bei hoher Systemlast tritt dann ein Seitenwechselvorgang genau dann auf, wenn dieses Zeichen korrekt ist. Der Seitenwechsel ist beobachtbar, so daß das richtige Zeichen dadurch erkennbar wird [Neu95, S. 109]. Entwurf, Unterlassung.
Die Sicherung gegen *willentliche* Angriffe wie diesen ist ein Thema für sich, in das wir hier nicht weiter einsteigen wollen. Es ließen sich aber Hunderte weitere Beispiele angeben, die nach Wirkung in folgende Kategorien eingeordnet werden können: unbefugter Zugang, unbefugte Dienstbenutzung, Bereicherung, Spionage, Dienstverhinderung, Zerstörung.

McDonalds Toaster: Für das neue Produkt McMuffin installierte McDonalds in seinen Filialen tausende von speziellen Toastern. Die Zeitgeber in diesen Toastern induzierten Spannungsspitzen im Stromnetz, die bewirkten, daß die Uhren, die die Arbeitszeit der Beschäftigten aufzeichnen, 2 bis 4 Stunden pro Tag vorgingen. Die Gehälter wurden entsprechend zu hoch ausbezahlt. Es wurden 5000 solche Toaster installiert, bevor das Problem gefunden wurde [Neu95, S. 160]. Entwurf, Unterlassung.
Dieses Beispiel ist insofern interessant, als es zeigt, wie ein System, das jahrelang zuverlässig funktioniert hat, aufgrund neuer Betriebsumstände plötzlich versagen kann. Der Fehler liegt hier in unzureichender Robustheit der Uhren gegen rauhe Betriebsbedingungen.

Hohe Verbrauchsrechnungen: Ein Ehepaar in Tampa, Florida, erhielt eine Stromrechnung für einen Monat in Höhe von 4.062.599 Dollar und 57 Cents, nebst einem Angebot "in monatlichen Raten" zu zahlen. Der korrekte Betrag war 146,76 Dollar. Der Grund war offenbar ein Bedienungsfehler des Computerpersonals. Malcolm Graham erhielt eine Wasserrechnung für einen Monat über etwa 35 Millionen Liter. Eine neue Wasseruhr war installiert und nicht korrekt eingestellt worden und der Computer hatte die Differenz zum alten Anzeigestand widerspruchslos als Verbrauch akzeptiert. In beiden Fällen hätten Plausibilitätsprüfungen dafür sorgen sollen, daß sich ein Mensch die Rechnungen nochmal vornimmt [Neu95, S. 191]. Entwurf, Unterlassung.

BRAVO editor: Der vermutlich erste WYSIWIG Editor überhaupt, BRAVO von Xerox PARC, wurde durch das Kommando `edit` in den Änderungsmodus geschaltet. War dieser jedoch schon eingeschaltet, so bedeutete `edit` stattdessen `everything delete insert t` und ersetzte den gesamten Dateiinhalt durch das einzelne Zeichen t. Es gab aber immerhin ein Tastaturlogbuch, aus dem man den vorherigen Zustand ausgehend von der ursprünglichen Version der Datei wieder herstellen konnte. Hier wurde offensichtlich keine Analyse typischer zu erwartender Bedienerfehler vorgenommen [Neu95, S. 206]. Entwurf, Unterlassung.

Space Shuttle Missionsabbruch-Problem: Bei Simulatortests für die zweite Space Shuttle Mission tauchte folgendes Problem auf: Die Mannschaft entschied sich zunächst zum Missionsabbruch und leitete ihn ein, brach den Abbruch dann aber ab und „flog" weiter. Als sie etwas später dann doch abbrechen mußten, geriet das Programm in eine enge Endlosschleife. Niemand hatte vorhergesehen, daß eine Mission quasi zweimal würde abgebrochen werden können [Neu95, S. 219]. Entwurf, Unterlassung.

4.5.3 Beispiele: Herstellungsprobleme

April 1981, Space Shuttle Startverzögerung: Vor dem Start des ersten Space Shuttles trat ein sehr subtiler Softwarefehler beim Initialisieren des fünfach redundanten Bordcomputersystems auf, der eine Chance von 1 zu 67 hatte. Der Fehler bewirkte, daß der fünfte Rechner, das Reservesystem, einen Zeitschritt hinter den vier Hauptrechnern lief, deshalb die Datenwörter, die aus seiner Sicht um einen Zeittakt zu früh in den Hauptrechnern auftauchen, als fehlerhaft zurückwies und sich nicht ordentlich in den Betrieb hineinstarten ließ. Der Reserverechner wurde erst kurz vor dem Start in Betrieb genommen, die vier Hauptrechner hatten während des 30stündigen Countdowns korrekt gearbeitet. Der Start verzögerte sich um zwei Tage [Neu95, S. 20f]. Implementationsfehler, Falschtun.

April 1993, Space Shuttle Startverzögerung: 11 Sekunden vor dem Start, unmittelbar vor dem Zünden der Triebwerke, wurde der Countdown abgebrochen, weil angeblich ein Ventil nicht geschlossen hatte. Später stellte sich heraus, daß nur die Software die Sensordaten des Ventils falsch interpretiert hatte. Der nächste Startversuch fand erst dreieinhalb Monate später statt; wegen mehrerer weiterer Probleme startete die Mission letztendlich erst mit mehr als fünf Monaten Verzögerung [Neu95, S. 24f]. Implementationsfehler, Falschtun.

Londoner Rettungsdienst: Einsatzplanungssoftware für Krankenwagen wurde nach Schwierigkeiten bei der Entwicklung trotz mehrerer Ausfälle bei Tests in Betrieb genommen. Resultat: Totalzusammenbruch der Einsatzplanung mit Verzögerungen bis zu 11 Stunden, etwa 20 Tote. Ursachen: System zu komplex, ungenügende Ausbildung der Leit- und Wagenmannschaften [Neu95, S. 72]. Entwurfsfalschtun und Betriebsunterlassung.

Versehentliche Bankbuchung: Die Bank of New York (BoNY) buchte bei einer Zahlung 32 Milliarden Dollar zuviel, als ein 16-bit Zähler unbemerkt überlief. Die meisten Zähler waren 32 bit groß. Die Bank mußte sich 24 Milliarden Dollar für einen Tag leihen, bis der Softwarefehler gefunden und behoben war; die Zinsen dafür betrugen 5 Millionen Dollar! [Neu95, S. 169]. Implementation, Falschtun.
Das Beispiel zeigt, daß ein einziges Auftreten eines einzigen, subtilen und unwahrscheinlichen Softwarefehlers einen großen Schaden anrichten kann — selbst wenn wie in diesem Fall die Auswirkung später korrigiert werden kann (denn das Geld wurde ja zurücküberwiesen).

4.5.4 Beispiele: Betriebsprobleme

August 1991, Rakete gesprengt: Eine Rakete wurde nach ihrem Start von Cape Canaveral absichtlich im Flug gesprengt, weil sie vom Kurs abgekommen war. Ein Techniker hatte beim Laden der Software in die Rakete einen falschen Knopf gedrückt und die Bodentestversion geladen anstatt der Flugversion [Neu95, S. 29]. Entwurfsunterlassung und Betriebsfalschtun.

April 1993, Intercity-Zusammenstoß: Am Karfreitag stieß ein aus Berlin kommender Intercity bei Wannsee mit einem nach Berlin fahrenden anderen Zug frontal zusammen. Während der vorangegangenen Arbeitswoche war die Strecke wegen Bauarbeiten im einspurigen Betrieb benutzt worden. An diesem Feiertag sollte eigentlich im normalen zweispurigen Betrieb gefahren werden, der Fahrdienstleiter hatte es jedoch versäumt, die Steueranlage entsprechend einzustellen. Soweit noch kein Problem. Der Steuercomputer stellte korrekt das Signal für den

Intercity auf Halt. Der Fahrdienstleiter hielt diese Reaktion für einen Systemfehler und benutze ein zusätzliches, für besondere Situationen bei Bauarbeiten vorgesehenes Signal, um trotzdem die Weiterfahrt zu erlauben und versuchte nicht, irgendjemanden wegen des vermeintlichen Fehlers anzurufen. Er übersah dabei, daß sich ein zusätzlicher, nicht fahrplanmäßiger Zug aus der Gegenrichtung näherte. Es gab drei Tote und mehr als 20 Verletzte, obwohl die Züge wegen des Einspurbetriebs nur mit 30 km/h fuhren! [Neu95, S. 51f]. Betrieb, Unterlassung.

Chemiefabrik explodiert: Ein Operateur in einer holländischen Chemiefabrik verwechselte beim Eintippen eines Rezeptes für einen Reaktor den Tank 632 (ein Harz) mit dem Tank 634 (Dicyclopentadien). Er vergaß das vorgeschriebene Nachprüfen. Durch einen Überhitzungsalarm beim Ablauf der Reaktion wurde die Feuerwehr verständigt, war aber mit dieser speziellen Situation überfordert; es erfolgte eine Explosion. 3 Tote, 11 Verletzte, Explosionsteile im Umkreis von 1 Kilometer, mehrere 10 Millionen Mark Sachschaden. Der Operateur war erst drei Monate auf seiner Arbeitsstelle und befand sich noch in Ausbildung [Neu95, S. 61f]. Betrieb, Falschtun (des Operateurs) und Unterlassung (der Firma).

April/Mai 1989, Fehler in Fahrstuhltür: Ein Fehler in Verriegelungsschaltkreisen erlaubte einem Fahrstuhl das Losfahren, obwohl nur eine der zwei Türen geschlossen war. Zweimal im Abstand von 8 Wochen wurde im selben Fahrstuhl ein Mensch getötet, der vom Fahrstuhl mitgeschleift wurde. Bereits 11 Jahre zuvor war der Fehler von der Fahrstuhlfirma erkannt worden, die 1978 auch eine Reparaturvorschrift herausgab (mit nur 1 zusätzlichem Draht). Das Gebäude wechselte 1980 den Eigentümer und 1988 die Fahrstuhlwartungsfirma, es wurden keine Wartungslogbücher geführt. Die Reparatur wurde erst nach den beiden tödlichen Unfällen ausgeführt [Neu95, S. 62f]. Betrieb, Unterlassung.

4.5.5 Analyse

Eine tiefschürfende Analyse der obigen Beispiele im Hinblick auf die Herkunftsarten der Fehler würde die Liste sicherlich überbewerten, denn sie ist alles andere als repräsentativ. Es fallen aber dennoch drei Dinge ins Auge:

1. Sehr viele der Probleme sind auf Unterlassungen zurückzuführen. Diese Probleme hätten zu einem großen Teil durch die Verwendung einfacher Checklisten beseitigt werden können. Solche Checklisten kann man z.B. durch Analyse früherer Probleme zusammenstellen.
2. Der weitaus überwiegende Teil der Probleme fällt nicht in den Bereich von schlichten Programmierfehlern, sondern entsteht in den frühen Phasen der Systemgestaltung oder im Betrieb.
3. Für Systemsicherheit kann man die Software nicht isoliert betrachten, sondern muß ihr Zusammenwirken mit dem restlichen System und dessen Betriebsumgebung verstehen.

Da leider in vielen Firmen ein völlig unterentwickeltes Sicherheitsbewußtsein vorherrscht, müssen **Sie**, lieber Leser und liebe Leserin, auch wenn Sie sich eigentlich nur als Softwareingenieur verstehen, einen Sinn für Sicherheitsprobleme entwickeln, diesen im Systementwurf kontinuierlich in Betrieb halten und energisch für die Lösung einmal erkannter Sicherheitsprobleme eintreten.

4.6 Allgemeine Suchtechniken zur Gefahrenbestimmung

Grundsätzlich eignen sich als konkrete Techniken viele der Techniken aus der Anforderungsbestimmung. Mit dieser Phase sollte die Gefahrenidentifikation auch stets verknüpft werden.

Außerdem sind Brainstorming-Sitzungen anhand der obigen Liste sinnvoll. Auf einer feineren Ebene der Analyse kann man Verkettungstechniken (zeitorientiert) und Zerlegungstechniken (strukturorientiert) unterscheiden:

- Bei der **Vorwärtssuche** bestimmt man, ausgehend von einem bestimmten Ereignis, die Einflüsse, die dieses und folgende Ereignisse auf den Systemzustand haben. Die folgenden Ereignisse müssen mit dem ersten nicht kausal verbunden sein; oft wird sogar ausdrücklich die Unabhängigkeit angenommen. Vorwärtssuche führt zu einer großen Zahl von Folgezuständen, von denen die meisten „uninteressant“ (nämlich nicht mit einer Gefahr verbunden) sind. Deshalb kann man sie meist nur für bestimmte, sehr kleine Bereiche eines Systems benutzen. Die wichtigste Anwendung besteht in der Untersuchung der Folgen von vorhersehbarem Versagen: „Was geschieht, wenn dieses oder jenes schiefgeht?“.
- Die **Rückwärtssuche**, die andere Verkettungstechnik, bestimmt hingegen ausgehend von einer als gefährlich erkannten Situation die vorhergehenden Zustände, aus denen sie erreicht werden könnte. „Was könnte zu einer solchen Situation führen?“. Sie ist insofern stärker gerichtet, kann aber nur schwierig mehrfache Ursachen für Probleme aufdecken. Rückwärtssuche ist die Grundlage der Fehlerbaumtechnik, die aus der Softwaretechnik-Vorlesung bekannt ist.
- Eine **Top-Down-Suche** zerlegt ein Ereignis oder einen Zustand in seine Teile gemäß der Systemstruktur und bestimmt die Ursachen dort. Z.B. kann man so versuchen, alle Arten zu bestimmen, auf die eine Datei gelöscht werden könnte. Dies ist analog zur Rückwärtssuche.
- Bei der **Bottom-Up-Suche** hingegen bestimmt analog zur Vorwärtssuche die Wirkungen eines Ereignisses/Zustands in einer Komponente auf das Gesamtsystem. Wegen Wechselwirkungen mit dem Zustand anderer Komponenten ist es aber meist sehr schwierig, die Wirkungen bis auf die Systemebene zu verfolgen.
- Häufig führt nur eine **kombinierte Suche**, die mehrere der vorigen Methoden verbindet und von Schritt zu Schritt wechselt, mit tragbarem Aufwand zum Erfolg. Zur richtigen Anwendung einer solchen Strategie ist Gespür und Erfahrung nötig.

4.7 Ursachen-Folgen-Analyse

Ein Beispiel für eine solche kombinierte Suchtechnik ist die Ursachen-Folgen-Analyse (cause consequence analysis): Hierbei wird als Ausgangspunkt ein kritisches Ereignis benutzt, zu dem erstens mit Top-Down-Suche oder Rückwärtssuche die Ursachen bestimmt werden und zweitens mittels Vorwärtssuche die Konsequenzen. Unter diesen Konsequenzen können weitere kritische Ereignisse auftreten. Damit diese eintreten können, müssen fast immer zusätzliche Vorbedingungen erfüllt sein gegenüber denen, die der Vorwärtssuche zugrundeliegen. Jede dieser zusätzlichen Vorbedingungen wird bei einer Ursachen-Folgen-Analyse mit Hilfe eines Fehlerbaums untersucht.

Die Abbildung 4.3 zeigt ein sehr einfaches Beispiel für die Anwendung dieser Technik: Das kritische Ereignis ist hier ein Überdruck in einem Druckbehälter des Systems. Als Ursache wurde mit Rückwärtssuche lediglich sehr allgemein ein unkontrollierter Druckanstieg ermittelt, da im Normalbetrieb kein Überdruck auftreten kann. Die Ursache wurde jedoch nicht weiter untersucht. Als Konsequenz sollte zunächst das Ventil 1 öffnen, ein weiteres kritisches Ereignis. Öffnet Ventil 1 nicht, sollte Ventil 2 öffnen, ebenfalls ein kritisches Ereignis.

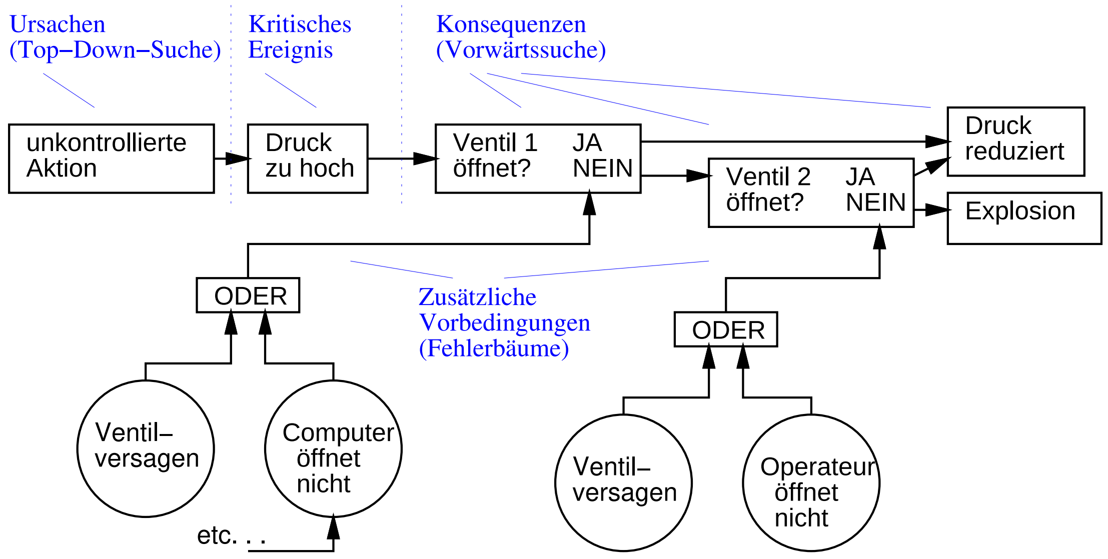

Abbildung 4.3: Beispiel einer Ursachen-Folgen-Analyse

Mögliche Ursachen für Fehlschläge bei den zusätzlichen kritischen Ereignissen, also der Fall, daß das Ventil nicht öffnet, werden mit Hilfe von Fehlerbäumen untersucht. Diese stellen die Bedingungen dar, unter denen die Fehlschläge auftreten könnten. In unserem Beispiel kann dies für das Ventil 1, welches eine automatische Steuerung durch einen Computer hat, ein Versagen dieser Steuerung sein oder ein Versagen des Ventils selbst. Ventil 2 ist handbetrieben, so daß als Grund wiederum Ventilversagen in Frage kommt oder der Fall, daß der Operateur, der das Ventil öffnen sollte, dies aus irgendeinem Grund nicht tut. Die Fehlerbäume können nun an allen 4 Enden weiterverfolgt werden, um zu bestimmen, warum die Ventile versagen oder warum die Steuerungen nicht das tun, was sie sollen.

In einem realen System ist natürlich die Zahl der Möglichkeiten an jedem Punkt einer solchen Analyse so groß, daß nur sehr selektiv einige davon verfolgt werden können. In der Auswahl dieser Möglichkeiten liegt die Herausforderung bei der Anwendung dieser Technik.

4.8 Gefahrenbestimmung mit formalen Notationen

Um die Schwierigkeiten einer korrekten und gründlichen Durchführung der Gefahrenbestimmung zu verringern, kann man versuchen, sie auf eine formale Darstellung des zu untersuchenden Systems aufzusetzen. Ein Beispiel hierfür wäre die Darstellung des Systems als endlichem Automaten und die Gefahrenbestimmung anhand dieses Automaten [Lev95, S. 346ff], die im folgenden diskutiert werden soll.

4.8.1 Darstellung des Systems als endlicher Automat

Grundsätzlich läßt sich jedes technische System als Zustandsautomat darstellen. In den meisten Fällen wäre die naturgetreue Modellierung in dieser Form jedoch nicht ein Automat mit einer endlichen Zahl von Zuständen und Zustandsübergängen, weil in den meisten Systemen kontinuierliche Variablen auftreten; im obigen Beispiel zur Ursachen-Folgen-Analyse etwa der Druck. Jedoch kann man in fast allen Fällen von der Menge aller Zustände abstrahieren, wenn es um die Gefahrenbestimmung geht und stattdessen nur die Menge der qualitativ verschiedenen

Zustände betrachten. Diese ist immer endlich; im Beispiel wäre dies für den Druck (Bestandteil des Zustands) nur die Unterscheidung, ob er akzeptabel oder zu hoch ist und für die Druckänderung (Zustandsübergänge), ob der Druck steigt, fällt oder konstant ist. Eine solche Betrachtung abstrahiert also aus einer großen Zahl von echten Systemzuständen in einen einzigen abstrakten Zustand in einer solchen Weise, daß dabei keine gefahrenrelevante Information verlorengeht. Zu beachten ist, daß nicht nur die Zustände ausreichend erfaßt werden müssen, sondern auch die Zustandsübergänge.

Es ist nicht immer klar, welche Zustände sinnvollerweise für die Gefahrenbestimmung als qualitativ verschieden betrachtet werden sollten. Erfahrung erlaubt jedoch eine Auswahl, die nur wenige Zustände unterscheidet und dennoch alle Gefahren zu identifizieren erlaubt. Es können leider keine allgemeingültigen Regeln hierfür angegeben werden.

4.8.2 Gefahrenbestimmung im endlichen Automaten

Liegt das System in Form eines endlichen Automaten vor, so müssen zunächst alle diejenigen Zustände identifiziert werden, die eine Gefahr bedeuten. Diese bilden dann den Mittelpunkt der Analyse: Im Prinzip kann man nun vom Startzustand des Systems aus mittels Vorwärtssuche alle erreichbaren Zustände untersuchen und ermitteln, welche gefährlichen Zustände darunter sind. Leider ist wegen der großen Zahl von Zuständen und Übergängen dieses Vorgehen schon für relativ einfache Systeme unrealistisch. Es gibt eine alternative Vorgehensweise, nämlich von gefährlichen Zuständen aus zu beginnen und mittels Rückwärtssuche zu prüfen, welche davon vom System erreichbar sind. Auch diese Methode scheitert jedoch an der enorm großen Zahl von Zustandspfaden.

Eine praktikable Lösung besteht oft darin, mit der Rückwärtssuche nur jeweils soweit fortzufahren, bis man einen Weg erkennen kann, wie das Modell (also auch das System) geändert werden muß, damit der gefährliche Zustand unerreichbar wird. Der Nachteil dieses Vorgehens besteht darin, daß man eventuell unnötige Änderungen am Modell vornimmt, weil der Zustand ohnehin unerreichbar war; dies kann man jedoch normalerweise gut in Kauf nehmen.

4.8.3 Probleme dieses Ansatzes

Die beschriebene Vorgehensweise hat drei Hauptschwierigkeiten, nämlich die Größe des Automaten, die Größe des Suchraums und die Frage nach der Korrektheit.

Die ersten beiden Probleme wurden oben bereits angesprochen. Der Versuch, ein System im ganzen als Automat zu erfassen, führt selbst in einfachen Fällen und unter Ausnutzung aller Abstraktionsmöglichkeiten zu so großen Automaten, daß deren Aufstellung, geschweige denn vollständige Untersuchung praktisch unmöglich wird. Deshalb läßt sich der Ansatz nur für Ausschnitte aus dem Systemverhalten durchführen oder unter bewußter Verwendung von Übervereinfachung, d.h. unter Inkaufnahme des Verlustes gefahrenrelevanter Information durch übermäßiges Zusammenfassen von Zuständen oder Übergängen.

Das dritte Problem teilt der Ansatz mit allen Verfahren, die auf einem abstrakten Modell des Systems arbeiten (z.B. Programmverifikation): Es kann grundsätzlich nicht sichergestellt werden, daß das Modell korrekt ist. Deshalb ist es möglich (und in der Realität sogar wahrscheinlich), daß auch nach einer vollständigen Gefahrenbestimmung am formalen Modell noch unerkannte Gefahren im wirklichen System vorhanden sein werden. Wie in allen anderen Fällen von formalen Methoden kann man auch hier versuchen, dem Problem durch Validation zu entgehen. Im vorliegenden Fall bedeutet das die Simulation des Systems anhand des formalen Modells und

Prüfung, ob sich die Simulation so verhält, wie man es vom System erwarten würde. Ihre Grenzen findet dieses Vorgehen dort, wo das Verhalten des zu modellierenden Systems nicht bekannt ist.

Dieser prinzipielle Mangel bedeutet jedoch keineswegs, daß der Einsatz formaler Methoden eine schlechte Idee wäre. Das Gegenteil ist der Fall, denn alle geschilderten Schwierigkeiten hat man prinzipiell auch dann, wenn man informell vorgeht, nur bemerkt man sie dann oftmals nicht — auch dies hat der vorliegende Fall mit anderen Anwendungen formaler Methoden gemein. Eine Gefahrenbestimmung mit endlichen Automaten ermöglicht eine gezieltere und bewußtere Auslassung von Bereichen in der Analyse als es ad-hoc-Methoden erlauben.

4.9 Weitergehende Information

[Lev95] ist ein sehr ausführliches Buch über Systemsicherheit, das aus der Perspektive der Informatik geschrieben ist. Es ist eine äußerst empfehlenswerte Lektüre für alle, die sich näher mit Fragen der Sicherheit auseinandersetzen möchten.

[Neu95] konzentriert sich vor allem auf eine umfangreiche Sammlung von realen Fallbeispielen für Sicherheitsprobleme aus den verschiedensten Bereichen. Die in diesem Buch zusammengetragenen Beispiele stammen überwiegend aus der regelmäßigen Kolumne des Autors Peter Neumann über Software-Risiken, die in den „Software Engineering Notes" erscheint, dem Organ der ACM Fachgruppe für Softwaretechnik. Der anfängliche Sammlungsort dieser Fälle ist meist die Usenet Newsgruppe comp.risks. Dort finden auch aktuelle und interessante Diskussionen zum Thema statt.

Auf die Perspektive der beteiligten Menschen konzentriert sich das Buch von Ivars Peterson [Pet95], das eine kleine Reihe von schweren Unfällen ausführlich untersucht und damit ein gutes Gegenstück zu den obigen beiden Büchern bildet.

Eine sehr ausführliche Beschreibung des Umfeldes eines sehr spektakulären Sicherheitsproblems stellt [Fey91] dar. Der Physiker Richard Feynmann beschreibt darin die Arbeit der Kommission, die die Explosion der Raumfähre Challenger untersucht hat. Die Geschichte verdeutlicht, daß es nicht ausreicht, wenn die Ingenieure gute Arbeit leisten, aber die Gesamtorganisation Risiken ignoriert. Ein deutlicher Aufruf an jede/n einzelne/n, sich auch auf der „politischen" Ebene von Sicherheitsfragen zu engagieren.

Ein weiteres lesenswertes Buch is „To Engineer is Human" von Henry Petroski [Pet92]. Darin geht es um die Rolle von Irrtümern und Versagen im Ingenieurwesen insgesamt, besonders im Bauwesen. Siehe insbesondere den Abschnitt 15 über den Übergang vom Rechenschieber auf den Computer und die negativen Folgen, die das für die Intuition der Ingenieure und damit für die Sicherheit hatte.

Der Artikel von Anderson [And94] schlägt die Brücke zum Bereich der Sicherheit gegen willentliche Angriffe (Schutz) und zeigt am Beispiel von Geldautomaten auf, warum dieser Bereich viel mit Sicherheit gegen Versagen zu tun hat und warum bislang die meisten Schutzsysteme nur unbefriedigend gut funktionieren.

Abschnitt 5

Softwareprozeß-Verbesserung

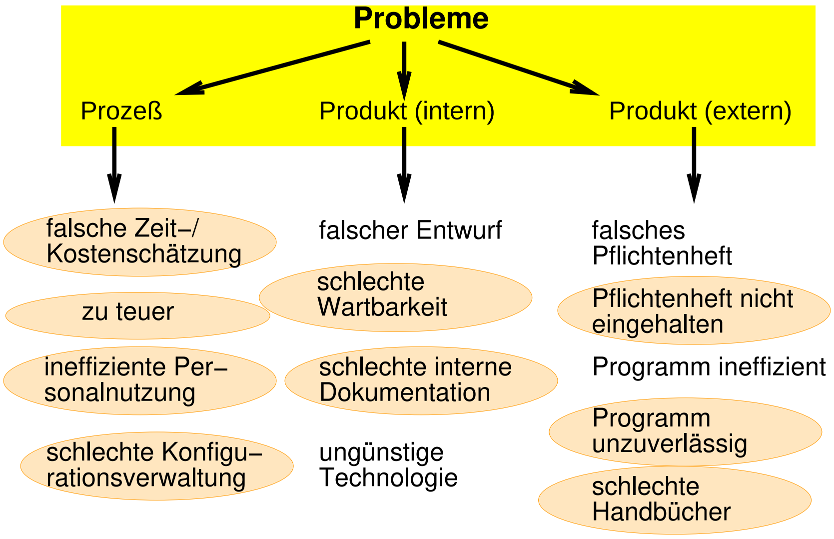

Abbildung 5.1: Von Softwareprozeß-Verbesserung angesprochene Problembereiche.

5.1 Einordnung und Ziele

5.1.1 Warum ist die Softwareproduktion so schlecht?

Verschiedene Leute in der Softwaretechnik-Forschung vertreten unterschiedliche Ansichten darüber, warum wir nicht besser Software produzieren können als wir es heute tun. Hier sind die wichtigsten ihrer Theorien:

Anforderungen unklar: Einige, beispielsweise Tom DeMarco, sehen es als das Grundübel an, daß wir meistens nicht genau genug wissen, welches System wir eigentlich bauen sollten; die Anforderungen sind zu ungenau oder (seltener) falsch festgestellt worden.
Wirkungsweise: Ungenaue Anforderungen müssen während des Projektablaufs präzisiert werden und führen zuvor meist zu Mißverständnissen und Fehlern. Falsche Anforderungen haben entweder den gleichen Effekt oder führen am Schluß zu Systemen, die geringen Nutzen haben.

Entwurf schlecht: Andere, zum Beispiel David Parnas, halten unsere mangelhafte Fähigkeit zum Finden guter Entwürfe für das größte Problem. Damit ist vor allem die Zerlegung von Sy-

stemen in Einzelteile mit möglichst geringen Wechselwirkungen beim Konstruktionsprozeß und bei Änderungen an Anforderungen oder Realisierungsentscheidungen gemeint.
Wirkungsweise: Ein schlechter Entwurf führt zu einer Systemzerlegung in Teile, die viele und komplizierte Wechselwirkungen miteinander haben. Das führt schon bei der planmäßigen Herstellung der Software, erst recht aber bei nachträglichen Änderungen an ihrem Aufbau zu einem sehr hohen Kommunikationsbedarf zwischen den Herstellern der Einzelteile, der ab einer gewissen Systemgröße den Konstruktionsprozeß allmählich ins Chaos stürzt; kleine Ursachen entfalten übermäßig große Wirkungen.

Werkzeugunterstützung gering: Wieder andere mit eher technokratischer Sichtweise sehen als Hauptproblem an, daß in der Softwareherstellung zuviel mit der Hand gearbeitet wird, was auch maschinell erledigt oder zumindest unterstützt werden könnte.
Wirkungsweise: Schwache Werkzeugunterstützung führt vor allem zu drei Effekten. Erstens kostet die Beschaffung und Verwaltung von Informationen zuviel Geld. Zweitens sind selbst mit hohem Aufwand Informationen oft nicht verfügbar, jedenfalls dann und dort, wo sie benötigt werden. Und drittens sind Informationen zu oft falsch oder ungenau, weil ihre Verwaltung „von Hand" zu unzuverlässig ist. Der Begriff „Informationen" bezeichnet hier sowohl alle Teile des Softwareprodukts selbst als auch alle Informationen, die zur Organisation seiner Herstellung verwaltet werden müssen.

Notationen ungenau: Eine weitere Gruppe von mathematisch orientierten Informatikern glauben, das zentrale Problem sei die zu geringe Verwendung von Notationen mit exakt definierter Bedeutung. Formale Notationen erleichterten, so glauben sie, die Entwicklung in einem solchen Maße, daß damit mehr Probleme gelöst werden könnten als mit anderen Ansätzen.
Wirkungsweise: Ungenau definierte Notationen erzeugen Mißverständnisse und verbieten Korrektheitsbeweise bei Transformationen.

Softwareprozeß schlecht: Eine letzte Gruppe schließlich, allen voran zahlreiche Vertreter des Software Engineering Institute der Carnegie Mellon Universität, halten ungenügende Qualität des Herstellungsprozesses von Software für die Wurzel allen Übels und für den Schlüssel zu seiner Beseitigung. Hier geht es also nicht um einzelne Methoden oder Werkzeuge, sondern um deren Zusammenspiel miteinander, mit den Menschen in der Organisation und mit dem gesamten restlichen Umfeld von Kunden, Anforderungen, Beschränkungen und so fort.
Zur Wirkungsweise gleich unten mehr.

Es sollte klar sein, daß alle diese Erklärungen einige Argumente auf ihrer Seite haben; alle diese Problemfelder sind bedeutend[1]. Vermutlich ist jedes davon bedeutend genug, daß die Frage müßig ist, welches das größte Problem darstellt; das hängt zu sehr vom Anwendungsgebiet und der jeweiligen Softwareorganisation ab.

Allerdings nimmt die Softwareprozeß-Verbesserung eine Sonderstellung ein: Zu ihren Anliegen gehört es nämlich, die relative Wichtigkeit der anderen Probleme festzustellen und Lösungen vorzuschlagen, die für die jeweilige Organisation und das jeweilige Projekt geeignet sind.

5.1.2 Ziele von Softwareprozeß-Verbesserung

Alle nachfolgenden Ausführungen beziehen sich vorrangig immer auf Softwareentwicklung in recht großen Projekten mit einigen Dutzend oder mehr Beteiligten. Für kleinere Teams gelten

[1]Es ist übrigens eine interessante Beobachtung, daß alle diese Schwierigkeiten ihre Beeinträchtigung der Qualität und/oder Produktivität der Softwareentwicklung überwiegend auf dem Weg einer Kommunikationserschwernis erzielen: Entweder sie führen zu Mißverständnissen oder zumindest zu hohen Kommunikationskosten; beides erhöht die Zahl von Fehlern und verlangsamt den Arbeitsprozeß.

im Prinzip die gleichen Überlegungen, jedoch können kleine Teams die Schwierigkeiten manchmal auf andere Weise überwinden, wenn die Teammitglieder sehr kompetent sind.

Folgende Haupteigenschaften eines Softwareprozesses lassen sich identifizieren, die als Grundlage für einen guten Softwareprozeß gesehen werden können. Die Ziele von Softwareprozeß-Verbesserungsmaßnahmen müssen also zunächst sein, den Prozeß so umzugestalten, daß er diese Eigenschaften aufweist. Erst darauf aufbauend wird dann die eigentliche Verbesserung angestrebt.

Wohldefiniertheit: Die grundlegende Forderung an einen Softwareprozeß für ein großes Team lautet, daß er wohldefiniert sein muß. Das heißt, es muß allen bekannte Regelungen geben, was unter welchen Umständen mit welchen Randbedingungen getan werden muß, soll, darf oder nicht darf. Wenn der Softwareprozeß nicht definiert ist, wird die Arbeit der einzelnen Personen zu oft aneinander vorbeilaufen, werden Arbeitsschritte nicht, unvollständig, falsch oder doppelt erledigt und werden Dokumente unverständlich sein, weil jeder sein eigenes Format verwendet. Definiertheit ist ab einer gewissen Teamgröße Grundvoraussetzung für geordneten Ablauf. Man beachte, daß Definiertheit nicht in Form von Vorschriften erfolgen muß (präskriptiv definierter Prozeß), sondern im Prinzip auch durch passive Beschreibung der Arbeitsweise erfolgen kann, die in einem weiten Rahmen freigestellt bleibt (deskriptiv definierter Prozeß).

Quantitatives Verstehen: Der nächste Anspruch lautet, beobachten und verstehen zu können, was im Softwareprozeß vor sich geht. Das dabei entwickelte Verständnis sollte nicht nur qualitativer Natur sein, sondern quantitativ zu modellieren erlauben, wo im Softwareprozeß wieviel Zeit verwendet wird (desgleichen andere Ressourcen) und wo wie viele und welche Fehler gemacht werden. Dieses Verständnis bildet die Grundlage für Prozeßverbesserung, weil andernfalls Verbesserungen nur aus intuitiven Eindrücken einzelner heraus gemacht werden können, die aber in einem großen Team zwangsläufig nur einen kleinen Ausschnitt des Gesamtprozesses berücksichtigen und meist dementsprechend wenig wirksam sind. Die meisten wichtigen Verbesserungen können ohne quantitatives Verständnis des Prozesses nicht entdeckt werden.

Kontinuierliche Änderung: Als letzte Eigenschaft braucht der Softwareprozeß die Fähigkeit, sich kontinuierlich zu ändern und zwar teilweise, um Verbesserungen zu realisieren, teilweise aber auch, um sich ändernden Anforderungen in der Umgebung gerecht zu werden (Werkzeuge, Anwendungsgebiete, Zusammenarbeit mit Zulieferern etc.). Ist diese Fähigkeit im Softwareprozeß nicht vorhanden, so gibt es zwei Möglichkeiten: Entweder der Prozeß wird (unterschwellig oder gewaltsam) doch geändert und verliert damit seine Wohldefiniertheit, oder er verliert seinen Nutzen, weil er zu ineffektiv ist oder den neuen Anforderungen nicht gerecht werden kann. Ein guter Softwareprozeß muß laufend kleine Änderungen an sich selbst vornehmen — man kann hier von einem Meta-Prozeß sprechen. Gute Softwareorganisationen haben deshalb oft eine Gruppe von Leuten, die sich nur mit der Weiterentwicklung des Softwareprozesses befaßt, die Software Engineering Process Group (SEPG).

5.2 Ansätze

Wir besprechen Ansätze zur Prozeßverbesserung nicht im Rahmen einzelner Maßnahmen oder Techniken, sondern von Rahmenwerken, die das Ziel von Prozeßverbesserungsmaßnahmen beschreiben (also die Eigenschaften eines guten Softwareprozesses) und insofern eine Entwicklungsrichtung vorgeben.

Wir besprechen einen Ansatz ausführlich, nämlich das SEI CMM, und mehrere andere kurz: ISO 9001 (5.4), Bootstrap (5.5), und Spice (5.6).

5.3 Das Capability Maturity Model (CMM)

Das Software Engineering Institute (SEI) ist eine mit großen Mengen staatlicher Gelder geförderte Einrichtung an der Carnegie Mellon Universität in Pittsburgh und hat die Aufgabe, die Softwarepraktiken in der amerikanischen Industrie zu verbessern. Kernstück seiner Aktivitäten ist das sogenannte „disciplined engineering"–Programm, in dem in direkter Zusammenarbeit mit Firmen bessere Ingenieurpraktiken auf der Methodenebene entwickelt werden. Eine der Entwicklungsrichtungen hierbei ist eine Methode zur Bewertung der Fähigkeiten einer Softwareorganisation. Die Ausgangsüberlegung hierzu lautet, daß die Qualität eines Softwareprodukts weitgehend von der Qualität der Entwicklungsprozesse abhängt, mit denen es hergestellt wird. Also wurde das Capability Maturity Model (CMM) entwickelt.

Wie der Name schon sagt, ist das CMM ein Modell, also eine abstrakte Beschreibung eines Teils der natürlichen Welt. In diesem Fall eine Beschreibung mehrerer Klassen von Softwareprozessen. Der Zweck des Modells ist, einen Vergleichsmaßstab für wirkliche Softwareprozesse zu bekommen, um gezielt und reproduzierbar deren Stärken und Schwächen bestimmen zu können. Dies kann man sowohl als Basis zur Verbesserung der eigenen Organisation einsetzen als auch zur Auswahl zwischen fremden Organisationen, wenn man Aufträge zu vergeben hat. Das CMM soll einen Entwicklungspfad aufzeigen, der zur allmählichen Verbesserung des Softwareprozesses begangen werden kann.

Dieser Entwicklungspfad besteht beim CMM aus dem Durchlaufen von fünf Stufen zunehmender Prozeßreife, die wir unten etwas ausführlicher besprechen werden. Zu jeder dieser Stufen gibt es eine kleine Zahl von Schlüsselbereichen (key process areas), in denen der Prozeß eine gute Qualität erreichen muß, um auf die jeweilige Stufe zu gelangen. Diese gute Qualität ist mit Hilfe einer ganzen Reihe von Schlüsselpraktiken (key practices) zu jedem Bereich näher beschrieben. Die Schlüsselpraktiken sind gruppiert nach denjenigen Implementationsaspekten der Verbesserung im Schlüsselbereich, die sie betreffen, den sogenannten relevanten Aspekten (common features). Siehe auch Abbildung 5.2.

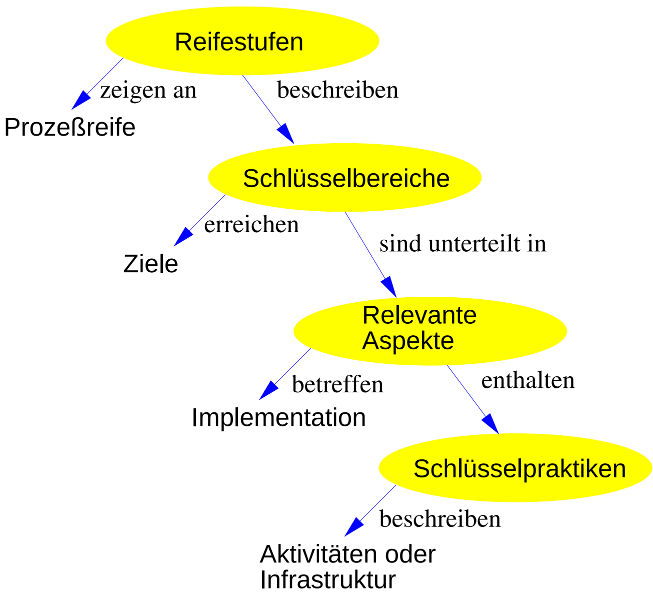

Abbildung 5.2: Organisation des CMM

Die Bezeichnung Schlüsselbereich deutet an, daß diese Bereiche diejenigen sind, an denen besonders oft Mängel die Qualität des Softwareprozesses erheblich beeinträchtigen. Verbesserungen in diesen Bereichen sind also besonders effektiv zur Erhöhung der Prozeßreife. Das CMM beschreibt nur diese Schlüsselbereiche, nicht auch alle anderen Aktivitäten bei der Softwareentwicklung. Jeder Schlüsselbereich muß im Hinblick auf fünf verschiedene Aspekte verbessert werden, um eine Prozeßverbesserung damit zu erzielen. Diese werden die relevanten Aspekte genannt und sind

- Selbstverpflichtung. Dies sind die Aktivitäten, die zur Vorbereitung und Aufrechterhaltung der Maßnahmen im Schlüsselbereich nötig sind.
- Fähigkeit. Dies sind die Vorbedingungen, die zur kompetenten Durchführung der Maßnahmen gebraucht werden: Ressourcen, Organisationsstrukturen und Ausbildung.
- Aktivitäten. Dies sind die Aktivitäten zur Einführung der Maßnahmen.
- Messung und Analyse. Technische Aktivitäten zur Untersuchung von Status und Effektivität der Maßnahmen.
- Implementationsüberprüfung. Managementmaßnahmen, die sicherstellen, daß die eingeführten Maßnahmen auch tatsächlich mit dem vorgesehenen Prozeß übereinstimmen.

Aus einem anderen Blickwinkel betrachtet gehen die Ziele der Schlüsselbereiche in die folgenden Richtungen:

- **Planung**: Ein Ziel ist immer, daß alle Aktivitäten geplant ablaufen, damit nicht unerwartete Ressourcenanforderungen auftreten, unsinnige Ausrichtungen der Aktivitäten zustandekommen oder schlicht Durcheinander auftritt.
- **Technische Aktivitäten**: Weitere Ziele betreffen natürlich die eigentlichen technischen Aktivitäten bei der Softwareentwicklung.
- **Management**: Andere Ziele besagen, daß im Fall besonderer Schwierigkeiten jemand zuständig sein muß, der ihre Beseitigung in die Hand nimmt und dann auch bis zu Ende verfolgt.
- **Zustimmung**: Schließlich gibt es an vielen Stellen Ziele, die darauf hinauslaufen, die Zustimmung der beteiligten Personen und Gruppen zu den Entscheidungen und Aktivitäten sicherzustellen. Diese Ziele tragen der Tatsache Rechnung, daß Softwareentwicklung nicht funktionieren kann, wenn die Beteiligten unzufrieden oder frustriert sind oder sich übergangen fühlen. In der praktischen Umsetzung bedeuten diese Ziele, daß Softwareprozeßverbesserung kein überwiegend technokratischer Vorgang ist, sondern vor allem viel Fingerspitzengefühl in der Einschätzung und dem Umgang mit Menschen verlangt.

5.3.1 Stufe 1: ad-hoc

Auf der Eingangsstufe gibt es keine ordentlichen Vorkehrungen für die Verwaltung von Softwareprojekten und keine stabile Umgebung für die Entwicklung. Im Fall von Krisen werden geplante Vorgehensweisen typischerweise über Bord geworfen und durch bloßes Kodieren und Testen ersetzt. Der gesamte Softwareprozeß wird ständig geändert und modifiziert, wie es gerade passend zu sein scheint; er ist ad-hoc.

Wenn eine solche Softwareorganisation trotzdem gute Produkte erzeugt, so liegt das an gutem Personal und meist heroischem Einsatz desselben. Die Funktionalität und Qualität und der

Fertigstellungszeitpunkt eines Softwareprodukts sind aber weitgehend unvorhersehbar. Stabilisierende Einflüsse auf den Prozeß durch ausnehmend gute Manager kommen in Stufe 1 zwar vor, hängen aber vollkommen von der Person ab und verschwinden gegebenenfalls mit dieser.

Die weitaus überwiegende Zahl von Softwareorganisationen befindet sich auf Stufe 1 und ein sehr großer Teil aller schweren Fehlschläge bei Projekten läßt sich auf die mangelnde Prozeßreife dieser Organisationen zurückführen. Es ist zu beachten, daß Stufe 1 weder eine notwendige noch eine hinreichende Bedingung für ein fehlschlagendes Projekt ist — lediglich die Wahrscheinlichkeit wird enorm erhöht.

5.3.2 Stufe 2: wiederholbar

Auf Stufe 2 werden Maßnahmen für eine geordnete Verwaltung und Verfolgung von Softwareprojekten eingeführt. Die Planung und Durchführung neuer Projekte nutzt explizit und geordnet die Erfahrungen aus früheren Projekten. Dies führt zu einer gewissen Vorhersagbarkeit des Prozesses und zu der Fähigkeit, Erfolge in früheren Projekten auf neuen, ähnlichen Projekten zu wiederholen. Man kann die Prozeßfähigkeiten von Stufe 2 mit dem Wort „disziplinierter Prozeß" zusammenfassen.

Auf Stufe 2 und allen höheren Stufen gibt es eine Reihe von Schlüsselbereichen des Prozesses (key process areas), in denen bestimmte Ziele erreicht werden müssen, bevor man davon sprechen kann, daß die Organisation diese Stufe erreicht hat.

Hier die Beschreibung der Schlüsselbereiche und Ziele für Stufe 2:

- **Anforderungsverwaltung**:
  Ziel 1: Systemanforderungen, die der Software zugewiesen werden, werden so verwaltet, daß sie eine Arbeitsgrundlage für das Software Engineering und das Management bilden.
  Ziel 2: Pläne, Produkte und Aktivitäten werden mit diesen Anforderungen konsistent gehalten.

- **Software-Projektplanung**:
  Ziel 1: Softwareschätzungen werden dokumentiert, um sie in der Planung und Verfolgung des Projekts einsetzen zu können.
  Ziel 2: Aktivitäten und Selbstverpflichtungen werden geplant und dokumentiert.
  Ziel 3: Die betroffenen Gruppen und Personen stimmen ihren Verpflichtungen zu.

- **Software-Projektverfolgung und -überblick**:
  Ziel 1: Tatsächliche Resultate und Leistungen werden gegenüber den Plänen verfolgt.
  Ziel 2: Gegenmaßnahmen werden eingeleitet und bis zuende verfolgt, wenn sich erhebliche Abweichungen zwischen Plänen und tatsächlichen Resultaten und Leistungen ergeben.
  Ziel 3: Die betroffenen Gruppen und Personen stimmen Änderungen ihrer Verpflichtungen zu.

- **Software-Unterauftragsverwaltung**:
  Ziel 1: Der Erstauftragsnehmer wählt qualifizierte Unterauftragsnehmer.
  Ziel 2: Der Erstauftragsnehmer und die Unterauftragsnehmer einigen sich über ihre gegenseitigen Verpflichtungen.
  Ziel 3: Erstauftragsnehmer und Unterauftragsnehmer sorgen für ständige Kommunikation miteinander.
  Ziel 4: Der Erstauftragsnehmer verfolgt die Resultate und Leistungen des Unterauftragsnehmers gegen dessen Verpflichtungen.

- **Software-Qualitätssicherung**:
  Ziel 1: Qualitätssicherungsmaßnahmen werden geplant.
  Ziel 2: Die Übereinstimmung von Softwareprodukten und -aktivitäten mit den relevanten Standards, Verfahren und Anforderungen wird objektiv überprüft.
  Ziel 3: Die betroffenen Gruppen und Personen werden über die Qualitätssicherungsmaßnahmen und ihre Resultate informiert.
  Ziel 4: Nichtübereinstimmungen, die nicht innerhalb des Projekts aufgelöst werden können, werden vom höheren Management in die Hand genommen.

- **Software-Konfigurationsverwaltung**:
  Ziel 1: Software-Konfigurationsverwaltungsaktivitäten (schönes Wort, gell?) werden geplant.
  Ziel 2: Bestimmte Software-Arbeitsprodukte werden identifiziert, verwaltet und verfügbar gemacht.
  Ziel 3: Änderungen an bestimmten Produkten werden überwacht.
  Ziel 4: Die betroffenen Gruppen und Personen werden über den Status und Inhalt der Software-Arbeitsgrundlagen informiert.

5.3.3 Stufe 3: definiert

Auf Stufe 3 werden alle Prozesse der Organisation, sowohl die technischen Prozesse als auch die zur Verwaltung (Management), definiert und dokumentiert. Sie formen nun erstmals ein zusammenhängendes Ganzes, das Standard-Softwareprozeß der Organisation genannt wird. Die Herausforderung der Stufe 3 lautet, Prozesse zu definieren, die die einzelnen Softwareingenieure in die Lage versetzen, effizient nützliche Arbeit zu tun, ohne daß die Prozesse dabei einengend oder unflexibel werden.

Die einzelnen Prozeßteile streben an, günstige Techniken zur Durchführung und Verwaltung der Arbeit einheitlich vorzuschreiben und anzuwenden. Dabei gibt es Eintritts- und Austrittskriterien für jede Arbeit, eine wohldefinierte Vorgehensweise, Qualitätssicherungsmaßnahmen (Begutachtungen) und eine Arbeitsgruppe, die diese Teile angemessen festlegt. Für jedes Projekt wird aus dem allgemeinen Rahmen eine spezielle Fassung erzeugt, die an die Anforderungen des Projektes gut angepaßt ist, der sogenannte definierte Prozeß des Projekts. Alle Beteiligten erhalten die nötige Ausbildung, um diesen Prozeß kompetent durchzuführen. Die Prozeßfähigkeiten auf Stufe 3 lassen sich in dem Wort „konsistent“ zusammenfassen, erstmalig gibt es *den* Softwareprozeß in einer Form, die sich tatsächlich angeben läßt. Eine Organisation auf Stufe 3 weiß, was sie tut. Sie weiß aber nicht unbedingt, wie gut sie es tut.

Hier die Schlüsselbereiche und Ziele:

- **Schwerpunkt auf organisationsweiten Prozeß**:
  Ziel 1: Aktivitäten zur Verbesserung des Softwareprozesses werden über die gesamte Organisation hinweg koordiniert.
  Ziel 2: Die Stärken und Schwächen der verwendeten Prozesse werden an einem Standard gemessen.
  Ziel 3: Organisationsweite Prozeßentwicklungs- und Prozeßverbesserungsaktivitäten werden geplant.

- **Prozeßdefinition**:
  Ziel 1: Ein Standard-Softwareprozeß für die Organisation wird entwickelt und gewartet.
  Ziel 2: Es wird Information über die Verwendung des Standardprozesses durch die Projekte gesammelt, ausgewertet und verfügbar gemacht.

- **Ausbildungsprogramm**:
  Ziel 1: Ausbildungsaktitäten werden geplant.
  Ziel 2: Es wird für Ausbildung gesorgt, die die Fertigkeiten und das Wissen zum Ausfüllen von technischen Rollen und Managementrollen in Softwareprojekten liefert.
  Ziel 3: Einzelpersonen in den Softwaregruppen erhalten die Ausbildung, die sie zur Erfüllung ihrer Rollen brauchen.

- **Integriertes Softwaremanagement**:
  Ziel 1: Der definierte Softwareprozeß des Projekts ist eine angepaßte Version des Standard-Softwareprozesses der Organisation.
  Ziel 2: Das Projekt wird entsprechend seinem definierten Softwareprozeß geplant und durchgeführt.

- **Software-Produktherstellung**:
  Ziel 1: Die Softwaretechnikaufgaben sind definiert, integriert und werden konsistent durchgeführt, um die Software herzustellen.
  Ziel 2: Die Software-Arbeitsprodukte werden miteinander konsistent gehalten.

- **Koordination zwischen Gruppen**:
  Ziel 1: Alle betroffenen Gruppen stimmen den Kundenanforderungen zu.
  Ziel 2: Alle betroffenen Gruppen stimmen jeweils Verpflichtungen zwischen Gruppen zu.
  Ziel 3: Die Ingenieursgruppen identifizieren, verfolgen und lösen Intergruppen-Fragen.

- **Peer Reviews**:
  Ziel 1: Begutachtungsaktivitäten werden geplant.
  Ziel 2: Defekte in Software-Arbeitsprodukten werden identifiziert und entfernt.

5.3.4 Stufe 4: verwaltet

Auf Stufe 4 werden quantitative Ziele zur Prozeßgüte formuliert und überwacht. Wohldefinierte Messungen werden eingeführt, um zahlreiche Aspekte des Prozesses quantitativ zu erfassen, die Daten werden in einer organisationsweiten Datensammlung abgelegt, um die Grundlage für Prozeßanalysen und -verbesserungen zu schaffen. Damit lassen sich Leistungsvariationen vermindern. Die Prozeßfähigkeiten auf Stufe 4 lassen sich in dem Wort „vorhersagbar“ zusammenfassen. Eine Organisation auf Stufe 4 weiß, wie gut ihr Prozeß ist. Sie weiß aber nicht unbedingt, wie sie ihn verbessern kann.

Die Schlüsselbereiche und Ziele für Stufe 4 lauten:

- **Quantitative Prozeßsteuerung**:
  Ziel 1: Die Aktivitäten zur quantitativen Prozeßsteuerung werden geplant.
  Ziel 2: Die Prozeßleistung des definierten Softwareprozesses des Projekts wird quantitativ überwacht.
  Ziel 3: Die Fähigkeiten des Standardprozesses der Organisation sind quantitativ bekannt.

- **Software-Qualitätsmanagement**:
  Ziel 1: Die Software-Qualitätssicherungsmaßnahmen des Projekts werden geplant.
  Ziel 2: Meßbare Ziele für Prozeßqualität und ihre Prioritäten werden definiert.
  Ziel 3: Der tatsächliche Fortschritt auf das Erreichen dieser Ziele für die Softwareprodukte wird quantifiziert und verwaltet.

5.3.5 Stufe 5: optimierend

Auf Stufe 5 richtet sich die Organisation auf laufende Verbesserung ihres Softwareprozesses aus. Sie kann ihre Schwächen identifizieren und ihren Prozeß vorbeugend verbessern. Die Meßdaten werden nun für Kosten-Nutzen-Analysen verwendet, auf deren Basis neue Technologien oder vorgeschlagene Änderungen im Softwareprozeß untersucht und beurteilt werden. Werden dabei Verbesserungsmöglichkeiten identifiziert, werden diese organisationsweit eingeführt. Defektvermeidung nimmt dabei eine wichtige Stellung ein: Begangene Fehler werden analysiert, gemeinsame Ursachen entdeckt und die Vorgehensweisen so verändert, daß diese Ursachen möglichst ausgeschlossen werden. Die Prozeßfähigkeiten von Stufe 5 lassen sich als „selbstverbessernd“ zusammenfassen.

Dies sind die Schlüsselbereiche und Ziele von Stufe 5:

- **Defektvermeidung**:
  Ziel 1: Defektvermeidungsaktivitäten werden geplant.
  Ziel 2: Wiederkehrende Gründe von Defekten werden gesucht und identifiziert.
  Ziel 3: Wiederkehrende Gründe von Defekten werden in eine Prioritätsfolge gebracht und systematisch beseitigt.

- **Verwaltung von Technologieänderung**:
  Ziel 1: Technologieänderungen im Prozeß werden geplant.
  Ziel 2: Neue Technologien werden bewertet, um ihre Wirkungen auf Qualität und Produktivität zu bestimmen.
  Ziel 3: Geeignete neue Technologien werden in der Organisation in die normale Arbeit eingeführt.

- **Verwaltung von Prozeßänderung**:
  Ziel 1: Kontinuierliche Prozeßverbesserung wird geplant.
  Ziel 2: Die Beteiligung an den Prozeßverbesserungsaktivitäten ist organisationsweit.
  Ziel 3: Der Standardsoftwareprozeß der Organisation und die definierten Softwareprozesse der Projekte werden kontinuierlich verbessert.

5.3.6 Managementsicht auf die Reifestufen

Eine andere Art, die Reifestufen zu betrachten, ist der Blickwinkel des Managements, das nicht selbst im technischen Prozeß der Softwareherstellung steckt. Stellen wir uns einmal die Frage, welche Information über den Ablauf der Softwareentwicklung und die Zwischenzustände in Bezug auf Qualität und Arbeitsfortschritt den Managern auf den verschiedenen Reifestufen zur Verfügung stehen.

Auf Stufe 1 ist der gesamte Softwareprozeß für das Management völlig undurchsichtig. Er beginnt irgendwann und nach einer gewissen Zeit kommt ein gewisses Ergebnis heraus. Weder kann vorher in irgendeiner Weise die Qualität des Ergebnisses abgeschätzt werden, noch läßt sich vor dem plötzlichen „Auftauchen“ des Ergebnisses vorhersehen, wann das geschehen wird. Software machen ist schwarze Magie. Das Projekt hat meist die berüchtigte 90-90-Form: Über 90% seiner Laufzeit ist es angeblich bereits zu 90% fertig.

Auf Stufe 2 ergibt sich zumindest ein gelegentlicher Einblick in die Softwareentwicklung. An vordefinierten Meilensteinen kann eine Abschätzung der Zeitplanung und eventuell auch der

Produktqualität vorgenommen werden. Softwareentwicklung mutiert zum Durchlaufen einer Abfolge von schwarzen Kästen. Zwischen je zwei solchen Kästen wird kurz eine Einsicht möglich, das Management kann auf Probleme reagieren, wenn sie aufgetreten sind.

Auf Stufe 3 wird auch die interne Struktur der Arbeitsstufen zwischen den Meilensteinen sichtbar, da der darin ablaufende Prozeß definiert ist; die schwarzen Kästen werden durchsichtig. Manager und Ingenieure verstehen ihre Rollen genauer und ein vorzeitiges Abschätzen von Risiken samt entsprechender Vorsorge wird möglich.

Auf Stufe 4 wird die Einsicht in die Prozesse von der qualitativen auf die quantitative Stufe gehoben. Das ermöglicht erheblich genauere Vorhersagen der Entwicklung, insbesondere von Problemen. Der Entwicklungsprozeß wird beherrscht.

Auf Stufe 5 erstreckt sich diese quantitative Einsicht auch auf Änderungen des Prozesses, so daß auch für Prozeßänderungen allmählich Vorhersagen ihrer Wirkungen möglich werden. Nicht nur der Entwicklungsprozeß selbst, sondern auch seine Fortschreibung und Verbesserung wird beherrscht.

5.4 ISO 9001

Ziel von Softwareprozeß-Verbesserung ist ja neben einer Verbesserung der Produktivität vor allem die Verbesserung und Garantie der Qualität, sowohl des Vorgehens als auch der resultierenden Produkte. Dieses Thema wird aber ja auch von der Normenserie ISO 9000 behandelt; also stellt sich die Frage nach einem Vergleich dieser Normen mit dem CMM.

Die ISO-9000-Standards beschreiben die Anforderungen an Qualitätssicherungssysteme, die zum Vertragsgegenstand gemacht werden können, wenn ein Vertragspartner dem anderen seine Fähigkeit zusichern will, ein gewisses Produkt herzustellen und zu liefern. Aus der ISO-9000-Serie ist vor allem ISO 9001 relevant für die Herstellung und Wartung von Software. ISO 9000-3 enthält Richtlinien für den Einsatz von 9001. Die Perspektive von ISO 9001 ist eher die einer Anforderungsbeschreibung, wohingegen CMM eher auf einen Verbesserungsprozeß abzielt. Extreme Unterschiede der beiden Ansätze ergeben sich vor allem im Detaillierungsgrad: CMM ist mehr als zehnmal so umfangreich wie ISO 9001 und 9000-3.

Die Form von ISO 9001 ist eine Liste von 20 Klauseln, die die Anforderungen beschreiben. Ein Stufensystem ähnlich der CMM-Reifestufen gibt es nicht. Von den zwanzig Klauseln haben 17 eine recht klare Entsprechung in einer oder mehreren Schlüsselpraktiken des CMM; zu den restlichen drei gibt es nur ungefähre oder gar keine Entsprechungen. So fordert Klausel 4.7 „control of customer-supplied product“, das im CMM allenfalls im „software subcontract management“ in Ansätzen wiederzufinden ist, Klausel 4.15 beschreibt die Anforderungen an „handling, storage, packaging, preservation, and delivery“, was im CMM nur teilweise abgehandelt wird und zwar bei „software configuration management“ und „software product engineering“. Schließlich gibt es zur Klausel 4.19 „servicing“ überhaupt kein Pendant im CMM.

Eine Organisation, die ISO 9001 genügt, erfüllt die meisten Anforderungen von CMM Stufe 2 und viele von Stufe 3. Sie kann jedoch im Prinzip auf Stufe 1 stehen, weil ISO 9001 nicht alle Schlüsselbereiche von CMM Stufe 2 anspricht. Viele der heute auf ISO 9001 zertifizierten Software-Organisationen sind tatsächlich nur auf Stufe 1; einige davon haben aber vermutlich ihre Zertifizierung nur deshalb bekommen, weil die Auditoren, die sie vornehmen, oft ungenügendes Verständnis für die Softwareproduktion mitbringen.

Umgekehrt kann eine Organisation auf CMM Stufe 3 die ISO 9001 Zertifizierung erhalten, wenn sie zusätzlich die Klausel 4.19 erfüllt und zugelieferte Softwareprodukte, die in ihren Softwareprodukten enthalten sind, richtig verwendet werden. Auch Organisationen von Stufe 2 müßten normalerweise mit nur wenigen zusätzlichen Maßnahmen die ISO 9001 Zertifizierung schaffen.

Zu der Frage, ob als Maßstab bei der Softwareprozeß-Verbesserung lieber CMM oder ISO 9001 benutzt werden sollte, lautet die wichtigste Antwort, daß diese Möglichkeiten keinen Widerspruch bilden. Da beide Rahmen sich erheblich überlappen, ist es nicht allzu problematisch, beide zu verwenden. Der Markt fordert wohl eine ISO-9001-Zertifikation in jedem Fall, aber für die wirkliche Leistungsverbesserung ist CMM vermutlich der bessere Weg, weil es erheblich detaillierter ist (über 500 Seiten gegenüber 11+15 für ISO 9001 und ISO 9000-3) und besser auf die Bedürfnisse von Softwareproduktion zugeschnitten.

5.5 Bootstrap

Bootstrap wurde im Rahmen von Esprit entwickelt und ist eine europäische Methode zur Softwareprozeß-Verbesserung, die weitgehend kompatibel mit dem CMM ist. Bootstrap ist etwas mehr auf die Bedürfnisse der europäischen Softwareindustrie abgestimmt, die mehr eingebettete Software macht und weniger Softwareprodukte als die amerikanische. Außerdem berücksichtigt Bootstrap alle Anforderungen von ISO 9001 und ISO 9000-3, was im CMM bislang (Version 1.1) nicht der Fall ist.

Bootstrap bestimmt nicht wie CMM nur einen Reifegrad, sondern erzeugt ein Fähigkeitsprofil mit über dreißig Attibuten für verschiedene Bereiche der Prozeßreife, indem verschiedene Bereiche auf verschiedene Reifestufen fallen können. Dieses Profil beschreibt Stärken und Schwächen detaillierter und erlaubt gezieltere Aufstellung von Aktionsplänen. Insbesondere kann ein Profil verwendet werden, um zu berechnen, wie wahrscheinlich es ist, daß eine ISO 9001-Zertifizierung bestanden werden wird. Eine nochmalig feinere Beurteilung wird dadurch erreicht, daß Antworten auf Fragen nach Reifemerkmalen nicht nur Ja oder Nein lauten können, sondern auf einer vierstufigen Skala abgegeben werden. Die Berechnung der Gesamtreifestufe ist in Bootstrap recht kompliziert und ist mehr eine Mittelung über die Stärken und Schwächen mit nur schwacher Schwellwertvoraussetzung zum Erreichen einer Reifestufe — im Gegensatz zum CMM, das fast ausschließlich eine Schwellwertprüfung macht, indem es verlangt, daß von n Merkmalen für Stufe i mindestens $n-1$ voll erfüllt werden.

5.6 Spice

Spice steht für **S**oftware **P**rocess **I**mprovement and **C**apability d**e**termination (au weia!) und ist ein geplanter Standard der ISO JTC1/SC7/WG10 (in lang: International Standardization Organization joint technical committee 1, subcommittee 7, working group 10).

Es handelt sich dabei um eine Sammlung von Dokumenten, die einen Rahmen für die Bewertung von Softwareprozessen von Lieferanten oder der eigenen Organisation liefern. Spice ist zur Selbstbewertung geeignet, berücksichtigt das Umfeld des Softwareprozesses und liefert als Ergebnis ein Bewertungsprofil für ein konkretes Exemplar eines Softwareprozesses. Es ist geeignet für Organisationen aller Größen und in allen Anwendungsfeldern. Die Bewertung beurteilt für jede Anforderung aus dem Maßnahmenkatalog die Angemessenheit der Maßnahmen, die im zu bewertenden Prozeß zur Erfüllung dieser Anforderung ergriffen werden, sowie das Maß der Übereinstimmung mit dem im Katalog vorgeschlagenen Standard.

Der grundsätzliche Unterschied zwischen Spice und CMM besteht darin, daß Spice kein Stufenmodell für die Prozeßreife verwendet, sondern von einer kontinuierlichen Reifeskala ausgeht, zu der zahlreiche stufenlose Reifefaktoren graduell beitragen.

5.7 Weiterentwicklung

Alle erwähnten Ansätze, (CMM, Bootstrap, Spice, ISO 9001) befinden sich in ständigem Fluß. Sie werden kontinuierlich an neue Anforderungen durch Änderungen im technischen oder betrieblichen Umfeld angepaßt.

Nehmen wir als Beispiel das CMM her: Sein Entwicklungsbeginn war 1986, die erste Ausprägung gab es 1987 in Form des ersten Reifestufen-Fragebogens, 1989 folgte die klassische Referenz auf das CMM, das Buch [Hum89] von Humphrey. Es folgten 1990 das CMM Version 0.2, 1991 das CMM Version 0.6 und Version 1.0 und schließlich 1993 das CMM 1.1. Gegenwärtig ist eine weitere Revisionsrunde im Gange, zu der Änderungsvorschläge eingereicht werden können. Diese Vorschläge werden vom SEI gesichtet, verglichen, zusammendestilliert und dann in die nächste Version des CMM eingearbeitet, wenn sie angemessen erscheinen. Eine Übersicht über diese Entwicklung gibt der Artikel von Paulk [Pau95a].

Folgende Entwicklungen lassen sich durch die verschiedenen Versionen des CMM erkennen:

- **Vereinfachung**: Die Zahl der Konzepte wurde verringert. Damit wird das CMM übersichtlicher und einfacher verständlich. Die Vereinfachung betrifft auch die *Darstellung* des CMM: In Version 1.1 wurden beispielsweise Formulierungsschablonen beim Schreiben der CMM-Beschreibung verwendet, um die Konsistenz der Formulierungen zu erhöhen, damit Leser leichter herausfinden, wo gleichartige Konzepte besprochen werden.
- **Orthogonalisierung**: Mit dieser Entwicklung einher geht der Versuch, die Konzepte möglichst unabhängig voneinander zu machen, weil dies der beste Weg ist, mit einer kleinen Zahl von Konzepten auszukommen.
- **Detaillierung**: Zugleich wurden die einzelnen Beschreibungen der Schlüsselbereiche und Schlüsselpraktiken im Laufe der Entwicklung detaillierter gefaßt.
- **Konkretisierung/Objektivierung**: Damit einher geht die Konkretisierung abstrakter Anweisungen und die Vermeidung subjektiver Formulierungen.

Außer *dem* CMM (voller Name: *CMM for Software*) gibt es noch andere Capability Maturity Models: for engineering, for software acquisition, for people management, for human interaction, for systems engineering [Pau95a]. Auf diese können wir hier nicht weiter eingehen. Eine Entwicklung, die noch aussteht, ist die Integration des CMM für Software mit den anderen CMMs, um einen gemeinsamen Rahmen für die erfolgreiche Entwicklung großer Systeme zu erhalten. Ebenso wird das SEI versuchen, die verwandten Aktivitäten der ISO (SPICE, 9001) im CMM zu berücksichtigen.

5.8 Ergebnisse

Das SEI bietet als Dienstleistung an, Einstufungen von Organisationen oder Projekten gemäß CMM vorzunehmen („SEI-assisted assessments"). Dazu gibt es eine genau definierte Methode, die auf einem umfangreichen Fragebogen basiert, an dessen Beantwortung eine ganze Reihe von Personen der Organisation teilnehmen müssen, und zwar sowohl technisches Personal als auch Management. Solche Einstufungen können alternativ auch ohne direkte Mitwirkung des SEI von der Organisation selbst vorgenommen werden („self-assessments“). Eine direkte Methode ist ein stark verkürztes Verfahren mit einem vereinfachten Fragebogen, das im Rahmen eintägiger CMM-Einführungskurse durchgeführt werden kann („workshop assessments“). Die frühesten Ergebnisse solcher Untersuchungen aus den Anfangsjahren des CMM hat das SEI in [HKK89] veröffentlicht. Dabei erreichten bei 55 vollen und 113 verkürzten Bewertungen unter 5% der Organisationen die Stufe 3 und keine einzige Stufe 4 oder 5! Seither hat sich die Lage etwas gebessert. Ergebnisse aus den Jahren 1992 bis 1997, entnommen aus [Zub97] sind in Abbildung 5.3 dargestellt.

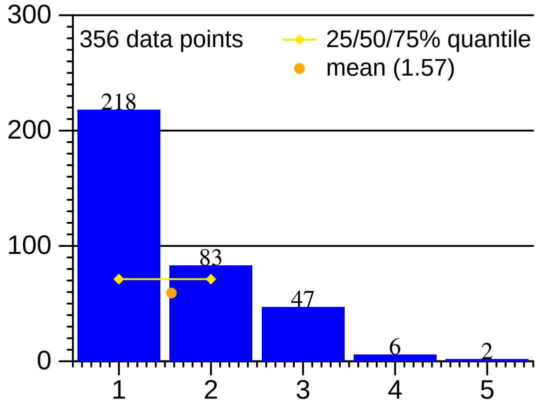

Abbildung 5.3: Ergebnisse der ersten CMM-Bewertungen von Organisationen

Wie wir sehen, ist der weit überwiegende Teil aller Organisationen und Projekte auf CMM-Reifestufe 1, nur jede siebte Organisation erreicht die Stufe 3 und die Stufen 4 oder 5 treten nur als Ausnahmen auf.

1995 machte das SEI eine größere Auswertung von **Anwendungserfahrungen** mit dem CMM und zwar in Form einer Umfrage [HG96]. Dazu schickte sie Fragebogen an 167 Personen aus 61 Institutionen, die mindestens ein Jahr und höchstens drei Jahre zuvor eine CMM-Selbstbewertung vorgenommen hatten. Die Zeitbeschränkungen sollten bewirken, daß einerseits genügend Zeit gewesen war, um Veränderungen anzustoßen, andererseits aber nicht soviel, daß die Leute, die die Bewertung mitgemacht hatten, nicht mehr erreichbar waren. In jeder Institution wurde soweit möglich ein Manager, ein erfahrener Softwareingenieur und ein Mitglied der Softwareprozeßgruppe (soweit vorhanden) befragt. Der Rücklauf betrug 83% oder 138 Fragebögen (47 von SW-Ingenieuren, 47 von Managern, 44 von SEPG-Mitgliedern). Die Antworten zeigten keine systematischen Unterschiede zwischen den drei Gruppen von Personen.

Die eigene Institution wurde zum Zeitpunkt der Umfrage von den Antwortenden in folgende **Reifestufen** eingeordnet: 87 mal Stufe 1, 36 mal Stufe 2 und 15 mal Stufe 3 oder höher. Bei der vorangegangenen früheren Selbstbewertung hatten erst 14 die Stufe 2 gehabt und 9 die Stufe 3 oder höher.

	Stufe 1	Stufe 2	Stufe 3
alte Bewertung	115	14	9
Eigeneinschätzung	87	36	15

Es wurden eine Reihe von Fragen gestellt, die mit einer aus den vier Möglichkeiten „exzellent“, „gut“, „mäßig“, „schlecht“ beantwortet werden konnten. Die Antwortenden sollten aus ihrer eigenen Perspektive eine möglichst gute subjektive Einschätzung abgeben. Dabei ergaben sich folgende Unterschiede zwischen den verschiedenen Reifestufen, wenn man über alle Antworten summiert.

Bei der Frage „**Wie gut ist die Fähigkeit ihrer Organisation**, den Projektzeitplan einzuhalten“ antworteten 40% der Antwortenden aus Organisationen auf Stufe 1 mit „exzellent“ oder „gut“, hingegen 80% der Antwortenden aus Organisationen auf Stufe 3. Ähnliche Änderungen ergeben sich beim Durchlaufen der Reifestufen von 1 nach 3 für die anderen Fragen:

- Budgetpläne gut einhalten: von 40% nach 68%.
- gute Produktqualität: von 75% nach 100%.
- hohe Produktivität: von 53% nach 88%.
- hohe Kundenzufriedenheit: von 77% nach 100%
- Zufriedenheit des Personals: von 23% nach 61%.

Ein interessantes Detail ergibt sich bei der Kundenzufriedenheit: Hier lag der Wert von Stufe 2 nicht, wie sonst überall, zwischen denen von Stufe 1 und 3, sondern bei nur 69%. Möglicherweise ist das darauf zurückzuführen, daß nicht die Kundenzufriedenheit bei Organisationen auf Stufe 2 schlechter ist als bei Stufe 1, sondern nur das Bewußtsein für deren Fehlen besser ausgeprägt.

Die Antwortenden wurden gebeten, im nachhinein abzuschätzen, wie zutreffend die Aussagen in der CMM-Bewertung waren im Hinblick auf die **Identifikation von Stärken und Schwächen** der Organisation. Bei den Schwächen fanden 63% die Aussagen sehr zutreffend, 35% zutreffend, bei den Stärken 38% sehr zutreffend, 54% zutreffend. 74% stimmten der Aussage zu, daß die CMM-Bewertung die Mühe wert war und einen erheblichen positiven Einfluß auf die Organisation hatte. 90% berichteten, daß die meisten ihrer Prozeßverbesserungsbemühungen auf den Resultaten der CMM-Bewertung basieren. 86% hielten das CMM für eine gute Leitschnur bei der Festlegung der Reihenfolge von Verbesserungsschritten.

Manche gelegentlich geäußerte **Kritik** am CMM und den CMM-Bewertungen erhielt in der Umfrage einige Zustimmung, allerdings immer nur mäßig viel: Die Aussage, die Empfehlungen seien zu ehrgeizig, erhielt 29% Zustimmung, die Bewertungsresultate hingen zu sehr vom Bewertungsteam ab, erhielt 18%. Alle anderen Kritikpunkte hatten weniger als 10% Zustimmung.

Wieviel und welche **Verbesserungen** haben die Organisationen denn nun erreicht? 44% fanden, ihre Verbesserungen seien beschränkt oder fast gar nicht vorhanden, 26% fanden sie mäßig und 31% fanden sie substantiell oder sehr groß. Fast alle Organisationen (96%) hatten einen Aktionsplan aufgestellt, aber nur 53% hatten Änderungen in Pilotprojekten vorgenommen, und ebensoviele hatten Änderungen organisationsweit eingeführt. Letztere Werte sind insofern überraschend, als 89% nicht nur einen Aktionsplan hatten, sondern auch ein Aktionsteam, um ihn umzusetzen.

Eine andere Untersuchung [BJ94] konzentrierte sich auf die Erfahrungen, die kleinere amerikanische Softwareunternehmen oder Softwareabteilungen (unter 500 Personen, im Mittel 40 Personen) mit dem CMM haben. Die Ergebnisse stammen aus 190 Fragebögen (bei 545 verschickten) und 46 Interviews mit hochrangigen Firmenvertretern (Geschäftsführer, Softwaremanager, Leiter der Softwareprozeßgruppe). Kleine Softwareabteilungen in größeren Unternehmen haben häufiger Prozeßverbesserungsprogramme (z.B. alle, die mehr als 20 Mitarbeiter haben) als kleine Unternehmungen (nur ca. die Hälfte derer mit 20-40 Mitarbeitern).

Die am häufigsten geäußerten Schwierigkeiten (mit Überlappungen) waren folgende:

- **Unerreichbare CMM-Anforderungen.** Einige Anforderungen, die aus dem CMM hervorgehen, sind für kleine Organisationen nicht erfüllbar. Dazu gehören insbesondere all jene, die die Abtrennung organisatorischer Gruppen und Rollen verlangen, wie Prozeßgruppe, Qualitätsmanagementgruppe und eine Prozeßbürokratie.

- **CMM-Schwächen.** Das CMM ist zu wenig detailliert und zu wenig anschaulich verfaßt und deshalb schwierig zu verstehen und anzuwenden.

- **Implementationskosten.** Prozeßverbesserungsprogramme sind generell teuer. Für kleine Organisationen sind die Kostenabwägungen in Bezug auf Amortisation und Konkurrenzfähigkeit besonders schwierig. Insbesondere die Kosten für die Ausbildung des Personals sind schwierig zu tragen.

- **Ressourcen.** Gerade kleine Organisationen haben erhebliche Schwierigkeiten, neben dem Geld auch Zeit und Personal für Prozeßverbesserungsaktivitäten bereitzustellen.

- **Kultur.** Viele Organisationen haben Schwierigkeiten damit, daß die Haltung ihrer Mitarbeiter oder die hergebrachte Arbeitsweise der Prozeßverbesserung entgegensteht, oder daß die Kunden ihr gegenüber nicht aufgeschlossen sind.

- **Kunden.** Kleine Organisationen arbeiten oft als Unterauftragsnehmer und haben sich dabei oft auch in Prozeßhinsicht auf die Anforderungen der Auftraggeber einzustellen. Das erschwert die Ausformung eines eigenen standardisierten Prozesses und den Aufbau einer in sich geschlossenen Sammlung von Prozeßdaten.

Auf der anderen Seite haben kleine Organisationen auch Vorteile gegenüber größeren: Die geringere Trägheit und Bürokratie erleichtert generell die Umsetzung organisatorischer Änderungen. Kürzere Dauer von Projekten beschleunigt die Einführung technischer und organisatorischer Änderungen, weil diese leichter am Projektanfang vorzunehmen sind als mitten im Verlauf eines Projekts. Die größere Nähe der Manager zum technischen Personal verbessert die Kommunikation und erlaubt eine leichtere Überwachung und Steuerung der Prozeßverbesserung.

Trotz allem überwiegen die Nachteile: Neben den Einschränkungen als Unterauftragnehmer, denen kleine Organisationen besonders oft ausgesetzt sind, und der Unmöglichkeit, manche Anforderungen des CMM überhaupt umzusetzen, sind es vor allem die Kosten, die kleinen Organisationen eine ungünstige Position bescheren. Die Kosten für Prozeßverbesserung sind für kleine Organisationen relativ höher als für große, weil sich der Aufwand nicht proportional mit der Organisationsgröße herunterskaliert. Das liegt daran, daß es einen gewissen Basisaufwand gibt, der nicht vermieden werden kann, zum Beispiel bei der Dokumentation eines standardisierten Prozesses oder bei der Beschaffung und Installation technischer Infrastruktur.

Nichtsdestoweniger haben viele kleine Organisationen erfolgreiche Prozeßverbesserungsprogramme. Sie haben den Übergang hin zu einer Betonung von Qualität als Arbeitsziel vollzogen, haben eine organisationsweite Unterstützung entsprechender Aktivitäten eingeführt und haben die Vorgaben des CMM oder anderer Prozeßverbesserungsrahmen kreativ auf ihre Bedürfnisse hin angepaßt und umgesetzt. Viele dieser Organisationen erfüllen rein formal betrachtet nicht einmal die Anforderungen der CMM-Stufe 2, haben aber trotzdem schon einen überwiegenden Teil aller Eigenschaften der Stufen 4 oder gar 5, weil sie angemessen und im Geiste des CMM ihren Prozeß verbessern.

5.9 Probleme und Hindernisse

Prozeßverbesserung nach dem CMM oder einem verwandten Modell hat zwei Hauptschwierigkeiten, die wir hier besprechen wollen. Erstens ist es schwierig und zweitens kann es unerwünschte Nebenwirkungen haben.

Die Schwierigkeit von Prozeßverbesserung ist recht offensichtlich: Eine ganze Organisation mit sehr komplexen Arbeitsabläufen muß unter genaue Kontrolle gebracht werden, wobei die beteiligten Individuen oftmals sehr eigenwillig reagieren und sich nur ungern irgendwelchen Regelungen unterwerfen. Kein Wunder, daß da oft die Erwartungen die Wirklichkeit übertreffen.

In der obenerwähnten Studie [HG96] finden sich folgende Angaben über die *Wirkungen* der CMM-Bewertung: 26% der Befragten stimmten der Aussage zu, es habe sich durch die Bewertung wenig geändert, 49% fanden, es gebe bei ihnen eine Menge **Desillusionierung** wegen der Resultate; allerdings widersprachen 84% der Aussage, die Softwareprozesse seien durch CMM zu bürokratisch oder unflexibel geworden. Bei 42% waren die Verbesserungsbemühungen durch besondere Ereignisse oder Krisen abgewürgt worden, und bei 72% waren sie zumindest beeinträchtigt aufgrund von Zeit- und Ressourcenbeschränkungen.

77% sagten, die Verbesserungen dauerten länger und 68% sagten, sie kosteten mehr als zunächst erwartet. 67% wünschten sich mehr Hilfe in der Frage, *wie* Verbesserungen erreicht werden können. Die ganze Unternehmung ist also alles andere als einfach. Nach den Erfahrungen des SEI dauert es normalerweise bei einem richtig eingefädelten Prozeßverbesserungsprogramm etwa 1 bis 2 Jahre, bis eine Organisation die nächsthöhere Reifestufe erreicht. Dies sind allerdings alles Einstiegsschwierigkeiten, die irgendwann überwunden werden.

Kritischer noch sind die Gefahren, die in der **Verselbständigung** der Bemühungen zur Prozeßverbesserung liegen. Hohe Prozeßreife ist eigentlich ein Mittel zum Zweck, nämlich gut Software herzustellen, kann aber leicht als **Selbstzweck** mißverstanden werden. Das führt dann unmittelbar dazu, daß die einzelnen Maßnahmen dazu neigen, sich etwas von den Zielen der Produktivitäts- und Qualitätsverbesserung abzulösen und nur noch auf bessere Kontrolle an sich, auf genauere Vorhersage an sich und auf präziseres Messen an sich abheben. Mittelbar ergibt sich aus dieser Verselbständigung noch eine gefährlichere Konsequenz, nämlich **Risikoscheu**.

Eine hohe Reifestufe nach dem CMM setzt voraus, daß der Entwicklungsprozeß gut verstanden und beherrscht wird. Das ist natürlich insbesondere nur dann möglich, wenn man genau weiß, was alles in welcher Reihenfolge und unter welchen Bedingungen zu tun sein wird. Dieses Wissen wiederum ist aber nur gegeben, wenn die Organisation bereits **Erfahrung** auf dem Gebiet hat, in dem die Software entwickelt werden soll, oder wenigstens auf einem verwandten. So ist eine Organisation, die hervorragend interaktive Einzelplatzanwendungen wie Textverarbeitungen, Tabellenkalkulationen etc. herstellen kann und dabei eine hohe Reifestufe in ihrem Prozeß hat, noch lange nicht in der Lage, die Entwicklung einer Echtzeitanwendung, eines Betriebssystems oder einer verteilten Datenbank mit hoher Prozeßreife zu betreiben. Will diese Organisation auf einem solchen neuen Gebiet Software entwickeln, muß sie damit rechnen, von ihrer Prozeßreifestufe, welche auch immer das ist, auf Stufe 2 oder gar 1 abzufallen, weil einfach kein definierter Prozeß vorliegt, der für diese unbekannten Anwendungsgebiete brauchbar wäre.

Auch schon die Integration neuer technischer Ansätze in die Produkte des vertrauten Bereichs kann die Prozeßreife zum Absturz bringen. Beispiele hierfür wären vor einigen Jahren die Einführung graphischer Benutzerschnittstellen gewesen, vor etwas kürzerer Zeit die Integration von neuronalen Netzen oder Fuzzy Logic, oder jüngst die Anbindung an HTML und das WWW; viele andere Beispiele ließen sich finden.

Die Gefahr von Prozeßverbesserungsanstrengungen liegt nun darin: Eine Organisation mit hoher Prozeßreife hat wahrscheinlich sehr wenig Neigung, diese Prozeßreife einem risikoreichen neuartigen technischen Unterfangen zu opfern. Tut sie es aber nicht, kann diese Entscheidung oftmals hohe Gewinnmöglichkeiten vereiteln. Es ist nämlich zu beobachten, daß der Profit, der sich aus gutbeherrschten Anwendungen ziehen läßt, immer mehr sinkt, weil alle sehr nutzbringenden Anwendungen in diesem Bereich längst vorhanden sind und das, was wir heute noch dazubauen, nur einen geringen Zusatznutzen aufweist. Andererseits können Organisationen, die mit hohem Risiko einen ganz neuen Bereich erschließen, dort oft noch hohe Marktanteile und dementsprechend hohe Gewinne erzielen. Deshalb ist Risikobereitschaft gerade im Softwarebereich eine wichtige Eigenschaft und Prozeßverbesserung steht ihr tendenzell entgegen.

5.10 Weitergehende Information

Das Software Engineering Institute (http://www.sei.cmu.edu) an der Carnegie Mellon Universität in Pittsburgh ist gewissermaßen der Regierungssitz der Prozeßverbesserer und deshalb eine gute Quelle von Information.

Eine andere relevante Adresse ist das European Software Engineering Institute (ESI, http://www.esi.es). Diese Einrichtung befaßt sich ähnlich wie das SEI mit Verbesserung der Softwarepraktiken (Bootstrap, Spice, ESPITI) ist jedoch weniger als das SEI aus öffentlichen Mitteln finanziert, sondern vor allem über Mitgliedsfirmen. In Italien residiert ferner das Bootstrap Institute (http://www.etnoteam.it/bootstrap/institut.html), das sich speziell mit der Fortentwicklung von Bootstrap befasst. Schließlich gibt es noch die European Software Process Improvement Foundation (ESPI, http://www.espi.co.uk/) in Großbritannien.

Die Entwicklungsgeschichte des CMM ist in [Pau95a] beschrieben. Ein Vergleich von CMM und ISO 9001 findet sich in [Pau95b]. Eine schöne breite Übersicht liefert [KS95], in dem auch noch andere Standards wie das V-Modell des Bundes oder diverse Dokumenten-Standards von IEEE angesprochen werden.

Das CMM selbst ist neben vielen anderen Dokumenten in seiner jeweils aktuellen Form auf dem FTP-Server (ftp://ftp.sei.cmu.edu/pub) des SEI zu finden. Die oben besprochenen empirischen Resultate finden sich in [HKK89] und [HG96].

Es gibt auch WWW-Seiten für Spice (http://www-sqi.cit.gu.edu.au/spice/welcome.html) und über die ISO (http://www.iso.ch/), aber erwarten Sie nicht, dort sehr umfangreiche nützliche Texte online zu finden. Beim ESI existiert eine Seite mit Verweisen auf andere Server (http://www.esi.es/sites.html).

Abschnitt 6

Der persönliche Softwareprozeß

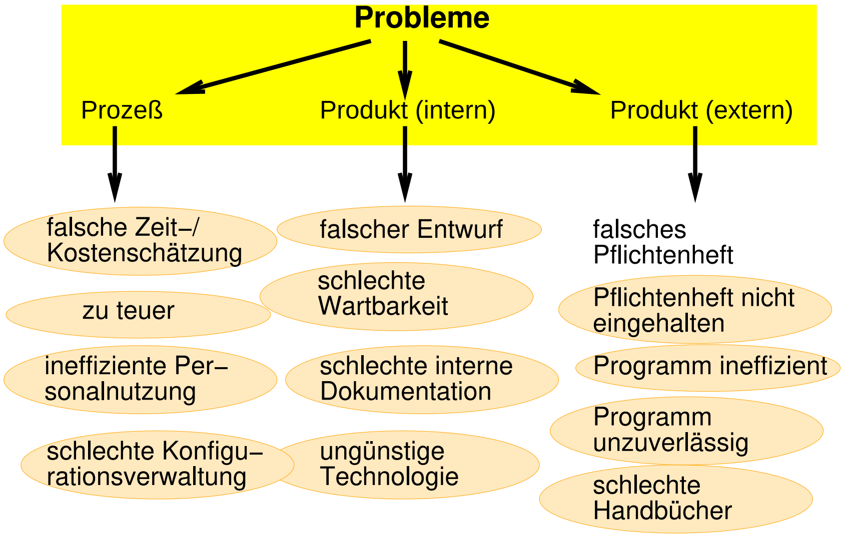

Abbildung 6.1: Vom PSP angesprochene Problembereiche.

6.1 Einordnung und Ziele

Der persönliche Softwareprozeß (PSP) bedeutet die Anwendung der Prinzipien des CMM auf einen einzelnen Softwareingenieur anstatt eine ganze Organisation. Der hier vorgestellte Ansatz wurde vom Miterfinder des CMM, Watts Humphrey, am SEI entwickelt und in einem Buch vorgestellt [Hum95]. Die nachfolgende Darstellung basiert auf den Ideen aus diesem Buch.

Ziel des PSP ist ein disziplinierter Softwareprozeß eines Individuums, der selbstoptimierend ist, also im CMM auf Stufe 5 anzuordnen wäre.

Der Vorzug des PSP-Ansatzes gegenüber CMM liegt darin, daß die Arbeit einer Einzelperson sehr viel leichter zu verändern ist als eine ganze Organisation. Zudem lassen sich leichter konkrete Techniken angeben, wie eine Verbesserung bewirkt werden kann. Drittens sind außerdem die Verbesserungen leichter als Erfolgserlebnisse für den einzelnen wahrzunehmen, was die Durchsetzung erleichtert. Der PSP soll CMM nicht ersetzen, sondern nur einen konkreten Zugang zur Prozeßverbesserung bieten, der die Erhöhung der CMM-Prozeßreifestufe erleichtert und auch schon ohne eine Erhöhung der Gesamtreifestufe der Organisation eine Verbesserung der Softwarequalität ermöglicht.

Psychologisch gesehen liegt ein wesentlicher Vorzug des PSP darin, daß er dem einzelnen Softwareingenieur hohe Kontrolle über seine Arbeitsweise beläßt bzw. überhaupt erst ermöglicht, denn bei undisziplinierter Arbeitsweise wird der jeweils nächste Schritt ja typischerweise mehr von unerwünschten Ereignissen diktiert als vom Programmierer selbst festgelegt. Wie wir in Abschnitt 2.6.3 bereits gesehen haben, ist hohe Kontrolle über die eigene Arbeitsweise und -umgebung motivationsfördernd.

6.2 Ansätze

6.2.1 Grundideen

Dem PSP liegen einige Überlegungen zugrunde, die auf den Unterschieden zwischen dem Softwareprozeß einer Organisation (umfassender Softwareprozeß, USP) und dem einer Einzelperson beruhen:

- **Wenig Kommunikation**: Innerhalb der Arbeit einer Einzelperson ist keine Koordination und Synchronisation mit anderen erforderlich. Dadurch beschränkt sich die Kommunikation innerhalb des Softwareprozesses auf Kommunikation der Person mit sich selbst (in Form von Datensammlung, Prozeßdefinition und Produktdokumentation). Diese verringerte Kommunikation läßt einige Probleme im organisationsweiten Prozeß wegfallen und erleichtert die Lösung vieler übriger.

- **Hohe Flexibilität**: Zur Änderung des Prozesses einer Einzelperson ist keine langwierige Abstimmung innerhalb einer Gruppe notwendig. Bei der Einführung der Änderung muß auch nicht auf ein schwächstes Glied gewartet werden. Außerdem sind die Prozeßelemente eines persönlichen Prozesses einfacher als die eines Prozesses für ein großes Team. Diese Unterschiede machen den PSP sehr viel flexibler als einen USP, was das Erreichen von Reifestufe 5 sehr erleichtert.

- **Selbstähnlichkeit**: Die Übertragung von Daten aus einem Projekt auf ein anderes ist im PSP sehr erleichtert, weil nur Daten des gleichen Softwareingenieurs benutzt werden. Dieser hat erstens durch seinen persönlichen Stil eine erhebliche Selbstähnlichkeit in seiner Arbeitsweise und kann zweitens gegebenenfalls recht gut abschätzen, wie alte Daten umzuinterpretieren sein dürften, falls sich am Prozeß oder am Anwendungsgebiet etwas geändert hat, denn er hat die alten Daten ja selbst produziert und kennt ihre Bedeutung dementsprechend gut. Diese Uniformität in den Daten erlaubt insbesondere auch die zuverlässige Anwendung einiger Planungs- und Beurteilungstechniken, die andernfalls nur sehr unzuverlässig oder ungenau funktionieren würden.

- **Individualität**: Ein großer Vorzug des PSP besteht im völlig geänderten Sinn des Begriffes „Standard" im Softwareprozeß. In einem USP ist ein Standard fast immer eine Regel, die einen Kompromiß zwischen verschiedenen ähnlich sinnvollen Vorlieben, Methoden, Gewohnheiten, Techniken etc. darstellt. Insofern sind USP-Standards oft zumindest für einige Mitglieder der Organisation ein unerwünschter Zwang. Demgegenüber setzt ein PSP-Standard lediglich Konsistenz voraus, nicht aber Kompromiß. Ich kann also meine PSP-Standards genau so machen, wie sie für mich selber erstens am sympathischsten und motivierendsten und zweitens am zweckmäßigsten sind. Da sich Arbeitsweisen, Problemlösungsmethoden und kognitive Prozesse von Person zu Person erheblich unterscheiden können, gibt es keine einzig optimale Lösung für die meisten Prozeßprobleme, sondern nur individuell optimale. Deshalb kann ein PSP-Standard leistungsfähiger (im Sinne von Produktivität und Softwarequalität) sein als ein USP-Standard, weil letzterer notgedrungen für einige seiner Anwender suboptimal sein wird.

6.2.2 Die Schlüsselbereiche im PSP

Der PSP zielt auf einen selbstoptimierenden persönlichen Prozeß. Dementsprechend müssen die meisten der Schlüsselbereiche des CMM auch im PSP angesprochen werden. Einige davon entfallen jedoch, weil die ihnen zugrundeliegenden Probleme nicht in den Arbeitsbereich des Individuums fallen, sondern der Koordination innerhalb einer Projektgruppe zuzuordnen sind.

Die nachfolgenden Abschnitte besprechen in Kurzform die Maßnahmen, die im PSP für die einzelnen Schlüsselbereiche vorgeschlagen werden. Wo Schlüsselbereiche grundsätzlich nicht relevant für persönliche Prozesse sind, steht „entfällt“. Eine kleine Anzahl von Bereichen könnte prinzipiell im PSP angesprochen werden, wird aber in der derzeitigen Version nicht behandelt. Dies ist ebenfalls bemerkt.

6.2.3 Stufe 2: wiederholbar

- Anforderungsverwaltung.
  Wird derzeit im PSP nicht angesprochen
- **Software-Projektplanung.**
  Es wird eine Zeitplanung auf dem Weg über eine Größenschätzung, Produktivitätsdaten und Daten über die Zeitverteilung auf Arbeitsphasen gemacht.
- **Software-Projektverfolgung und -überblick.**
  Die Einhaltung des Zeitplans wird anhand wohldefinierter Meilensteine etwa zweimal wöchentlich überprüft. Im Falle erheblicher Abweichungen vom Plan wird der Plan entsprechend korrigiert, indem entweder der Zeitrahmen neu gesetzt oder die Aufgabe verkleinert wird.
- Software-Unterauftragsverwaltung.
  Entfällt.
- **Software-Qualitätssicherung.**
  Durchsichten werden verwendet, um Fehler möglichst früh zu finden. Fehlerdaten werden gesammelt und analysiert, um Fehler zu vermeiden und die Durchsichten zu optimieren. Tests werden sorgfältig geplant und dokumentiert.
- Software-Konfigurationsverwaltung.
  Entfällt.

6.2.4 Stufe 3: definiert

- Schwerpunkt auf organisationsweiten Prozeß.
  Enfällt.

- **Prozeßdefinition.**
  Auch im PSP werden die Prozeßeinzelteile definiert und schriftlich niedergelegt, um eine konsistente und nach außen hin durchsichtige Arbeitsweise zu ermöglichen. Im Gegensatz zum USP ist normalerweise keine Unterscheidung zwischen einem Standardprozeß und definierten angepaßten Prozessen für einzelne Projekte nötig.

- **Ausbildungsprogramm.**
  Auch der PSP enthält Maßnahmen zum geregelten Erlernen der notwendigen Fertigkeiten, für das Zeit ausdrücklich eingeplant werden soll.

- Integriertes Softwaremanagement.
  Enfällt.

- Koordination zwischen Gruppen.
  Entfällt.

- **Peer Reviews.**
  Im PSP werden Durchsichten vom einzelnen durchgeführt. Das ist zwar strenggenommen kein Peer Review, dient aber demselben Zweck, nämlich Qualitätsüberwachung und -verbesserung.

6.2.5 Stufe 4: verwaltet

Auf Stufe 4 werden die Stärken eines PSP erst richtig sichtbar: Was im USP eine späte Phase der Prozeßentwicklung ist und zu seiner effektiven Benutzung sehr viel voraussetzt, ist im PSP sehr konkret und relativ einfach innerhalb weniger Wochen einzuführen.

- **Quantitative Prozeßsteuerung.**
  Mit Hilfe einer ganzen Reihe von Meßgrößen wird die Produktivität und die Produktqualität des PSP laufend überwacht. Durch Zusammenschau dieser Meßdaten sind auch die Mittelwerte und zeitlichen Entwicklungen der Prozeßgrößen bekannt.

- **Software-Qualitätsmanagement.**
  Anhand der gleichen Meßgrößen werden Vorgaben an die Prozeßqualität definiert und der Fortschritt überwacht.

6.2.6 Stufe 5: optimierend

- **Defektvermeidung.**
  Durch regelmäßige Analyse der Defektdaten werden wiederkehrende Gründe für Defekte identifiziert. Durch Änderung der Aufmerksamkeit bei der Herstellung und Änderung der Prüflisten für die Durchsichten werden diese Gründe systematisch bekämpft.

- **Verwaltung von Technologieänderung.**
  Der PSP sieht vor, Gelegenheiten zu schaffen und zu benutzen, um neue Technologien auszuprobieren und zu bewerten. Erfolgversprechende Ansätze können direkt in die Arbeit übernommen werden. Hier wirkt sich die hohe Flexibilität eines PSP besonders positiv aus.

- **Verwaltung von Prozeßänderung.**
  Durch Analyse der Prozeßdaten in gewissen Abständen können Schwächen im Prozeß identifiziert und gezielt bekämpft werden. Die Analysen selbst werden in einem geplanten und geordneten Rahmen durchgeführt.

6.2.7 Der PSP-Kurs

Für den PSP stellt sich natürlich die gleiche Frage, wie für die höheren CMM-Reifestufen: Wie komm ich hin? Humphrey hat für den PSP einen Kurs ausgearbeitet, der einen mit einem Aufwand von etwa einem ganzen Arbeitstag pro Woche innerhalb von 15 Wochen zu einem PSP bringt, der CMM-Stufe 4 entspricht. Sein Buch [Hum95] ist in seinem Hauptteil passend zu diesem Kurs strukturiert, so daß man den Kurs ohne weiteres im Selbststudium machen kann.

Der Kurs verläuft vom Anfangsprozeß PSP0 über fünf PSP-Zwischenstufen bis zum vollen Prozeß PSP3. Der Basisprozeß **PSP0** ist dabei das individuelle bisherige Vorgehen, ergänzt um die Erfassung von Arbeitszeiten und Fehlern als Grundlage aller weiteren Prozeßverbesserungsschritte.

Im **PSP1** wird das Größenmessen und -schätzen und die Zeitplanung eingeführt. Die Zeitplanung wird mit zunehmenden Datenmengen verfeinert, basiert aber von Anfang an auf gemessenen Daten anstatt auf reiner Intution; das gleiche gilt für die Größenschätzung. Es wird nicht nur eine Schätzung der Gesamtzeit vorgenommen, sondern auch eine Festlegung zahlreicher feiner Meilensteine und eine Terminplanung, wann diese jeweils erreicht werden sollen. Damit kann eine genaue Planverfolgung und ggf. frühzeitige Korrektur gewährleistet werden.

Der **PSP2** führt das Qualitätsmanagement ein: Entwurfsdurchsichten, Codedurchsichten, Vermessung der Begutachtungsgeschwindigkeit, Defektentdeckungsrate und Defektentdeckungsausbeute und die systematische Auswertung von Fehlerdaten zur Identifikation und Beseitigung wiederkehrender Fehlerursachen. Das Qualitätsmanagement enthält auch eine Kostenbetrachtung, um den Prozeß je nach Entwicklungszielen gezielt in einem günstigen Kosten-Nutzenbereich betreiben zu können. Für die Entwurfsprüfung werden einige allgemein benutzbare Kriterien und Formalismen zur Prüfung von Entwurfsvollständigkeit und -korrektheit eingeführt.

Als letzter Schritt erweitert **PSP3** das Vorgehen auf inkrementelle Entwicklung, so daß die PSP-Methoden auch für größere Programme brauchbar bleiben.

Dieses Vorgehen des PSP-Kurses über Zwischenstufen ermöglicht ein schrittweises Erlernen und Einüben, wobei jeder Schritt mit eigenen Erfolgserlebnissen verbunden ist. Auf diese Weise wird die Motivation über den Kurs hinweg aufrechterhalten.

Auf jeder PSP-Stufe werden eine oder zwei Programmierübungen gemacht, die zwei Ziele haben: Erstens dienen sie als Übungsbeispiele für die Anwendung der jeweiligen PSP-Variante; zweitens sind die darin erzeugten Programme Werkzeuge für die Messung und Datenanalyse im PSP. Die einzelnen Übungen dauern etwa 0.5 bis 4 Stunden für Übung 1 bis hin zu 5 bis 30 Stunden für Übung 10, je nach Geschick und Gründlichkeit.

Die Programmierübungen werden ergänzt durch Prozeßverbesserungsübungen, die folgende Zwecke haben:

- Definition des Prozesses (Standards, Vorgehensskripte, etc.).
- Analyse der Daten zur Bestimmung von Prozeßgrößen und zur Erhöhung der Motivation.
- Analyse der Daten zur Identifikation von Schwachstellen.
- Verbesserung des Prozesses (Prüflisten, etc.).

Ergebnis des Kurses ist ein wohldefinierter Prozeß, der zunächst zum überwiegenden Teil aus vordefinierten Teilen besteht, aber auch schon selbstdefinierte Komponenten enthält (zumindest durch Auswahl aus mehreren Möglichkeiten). Dieser Prozeß wird aufgrund von Messungen quantitativ verstanden und kontrolliert und wäre insofern auf CMM-Stufe 4 einzuordnen. Das PSP-Buch enthält auch Anleitungen dafür, wie man kontinuierliche Prozeßverbesserung und Verwaltung von Technologiewechsel in den PSP einführen kann, um Stufe 5 zu erreichen. Dies kann jedoch nicht ausdrücklich im Kurs geübt werden.

6.3 Ergebnisse

Der PSP hat zwei Schwerpunkte, nämlich Planungsgenauigkeit und Qualitätsverbesserung. Allgemein liegt außerdem noch das Prinzip der Disziplin zugrunde. Interessanterweise lassen sich die meisten Annahmen, die den PSP-Methoden zugrundeliegen, und auch die meisten Behauptungen, die über ihre Wirkung aufgestellt werden, schon im Rahmen der PSP-Kurse mit realen Daten belegen. Im folgenden werden wir das mit Beispielen aus den ersten zwei PSP-Kursen tun, die Humphrey quasi als Prototypen während der letzten Entwicklungsphase des PSP abgehalten hat.

6.3.1 Planung: Grundlagen und Effekte

Grundlage der Planung im PSP ist die Tatsache, daß die Größe eines Programms, das von einer einzelnen Person entwickelt wird, sich erstens recht genau schätzen läßt und zweitens gut mit der Entwicklungszeit korreliert. Letzteres ist in Abbildung 6.2 am Beispiel von C++ Code dargestellt.

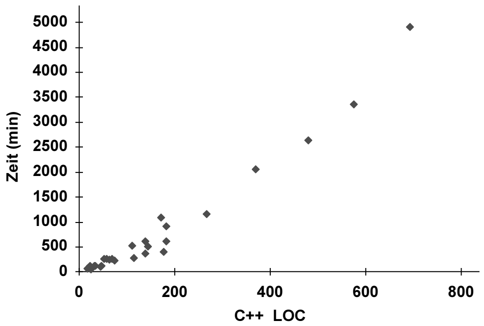

Abbildung 6.2: Korrelation von Anzahl LOC mit Entwicklungszeit (C++ Code)

Die Korrelation zwischen LOC und Entwicklungszeit ist nicht bei allen Programmierern und in allen Fällen so gut wie hier gezeigt; hohe Abweichungen werden aber meist von „fiesen" Fehlern hervorgerufen, deren Beseitigung sehr viel Zeit braucht und die in einem disziplinierten Prozeß sehr viel seltener sind als bei den meisten normalen Programmierern.

Interessanterweise läßt sich der gleiche Ansatz zur Planung auch auf ganz anderen Gebieten einsetzen. Zum Beispiel kann man auch die Zeit zum Entwurf einer größeren Anzahl von Bildschirmmasken gut aus dieser Anzahl vorhersagen oder die Zeit zum Schreiben eines Buches aus der Anzahl zu schreibender Seiten, wenn für einen Teil des Buches schon historische Daten vorliegen. Im letzteren Fall ist allerdings die Korrelation nicht so hoch, wie bei LOC, siehe Abbildung 6.3.

Eine interessante Beobachtung ist die, daß die Schätzabweichungen der einzelnen Kursteilnehmer nach oben und nach unten sich weitgehend aufheben, wie in Abbildung 6.4 zu sehen ist, so daß man die Teilnehmer zusammengenommen als ein gutes Schätzorakel betrachten kann. Dies zeigt,

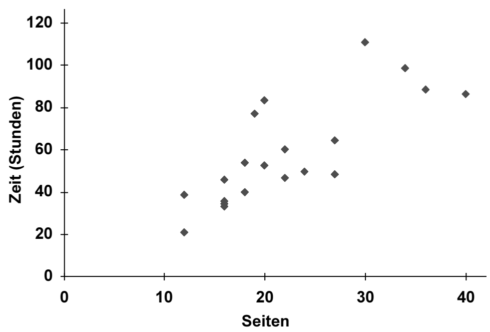

Abbildung 6.3: Korrelation von Anzahl Textseiten mit Schreibzeit beim Schreiben eines Buches

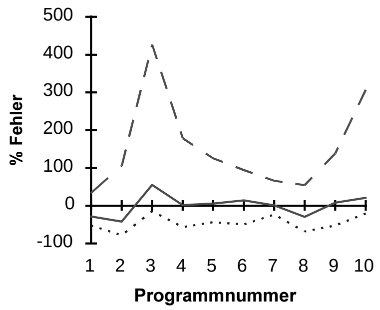

Abbildung 6.4: Schätzfehler für LOC gemittelt über Kursteilnehmer (obere Linie: Maximum, mittlere Linie: Mittelwert, untere Linie: Minimum)

daß die Schätzfehler der einzelnen keine systematischen Fehler sind, sondern nur (erhebliche) Schwankungen. Diese zu verringern, läßt sich lernen — und das ist ein Ziel des PSP.

6.3.2 Qualität: Grundlagen

Den Qualitätsmaßnahmen im PSP liegen einige Annahmen zugrunde, deren Zutreffen nicht selbstverständlich ist. Eine Annahme lautet zum Beispiel, daß jede Stufe der Fehlerentfernung nur einen gewissen Prozentsatz der Fehler findet, der sich auch mit stark erhöhter Mühe nur mäßig steigern läßt. Deshalb ist es erstens wichtig, möglichst wenig Fehler einzufügen (denn einen gewissen Prozentsatz der eingefügten Fehler werde ich immer am Schluß noch mit ausliefern). Zweitens ist es wichtig, möglichst viele Fehlerentfernungsstufen zu haben, und drittens sollten vor allem die frühen Stufen hinsichtlich der Ausbeute bei der Fehlerentdeckung optimiert werden, weil dort noch mehr Fehler vorhanden sind, die von einer prozentual gleichen Verbesserung erreicht werden. Das zentrale Maß des PSP in dieser Hinsicht ist die Ausbeute (yield), also der Prozentsatz von in Entwurf und Kodieren eingefügten Fehlern, die vor dem Übersetzen bereits wieder gefunden werden.

Abbildung 6.5 belegt, daß eine höhere Ausbeute auch zu höherer Gesamtproduktivität führen kann und Abbildung 6.6 belegt, daß sich die Ausbeute steigern läßt, indem mehr Zeit auf Durchsichten verwendet wird. Die Effekte sind nicht immer so deutlich, wie hier an zwei Kursteilnehmern gezeigt, sind aber stets vorhanden, wenn eine disziplinierte Arbeitsweise vorliegt und genug Daten vorhanden sind.

Eine zweite Annahme des PSP besagt, daß jeder Fehler schädlich ist, selbst ein trivialer. Dies läßt sich mit einer Graphik belegen, die eine positive Korrelation zeigt zwischen der Zahl der

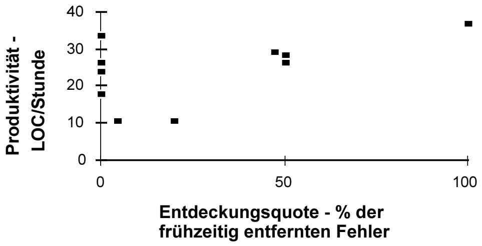

Abbildung 6.5: Zusammenhang von Fehlerausbeute und Produktivität bei Student 20

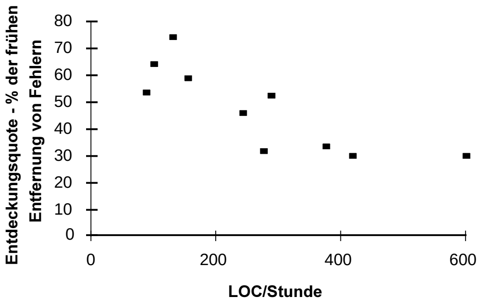

Abbildung 6.6: Fehlerausbeute gegen Begutachtungsgeschwindigkeit bei Student 12

Fehler, die vom Übersetzer gefunden werden, und der Zahl der Fehler, die in der Testphase gefunden werden; siehe das Beispiel zweier Studenten in den Abbildungen 6.7 und 6.8.

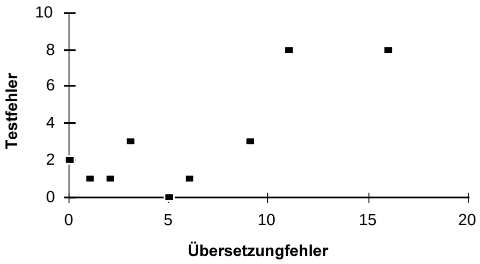

Abbildung 6.7: Zusammenhang der Anzahl gefundener Fehler bei Übersetzung und Test bei Student 1

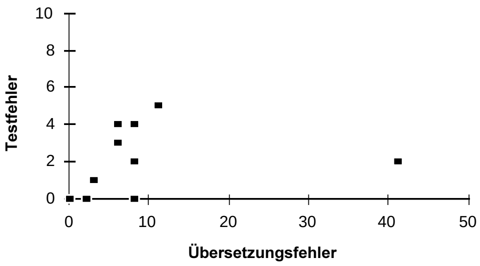

Abbildung 6.8: Zusammenhang der Anzahl gefundener Fehler bei Übersetzung und Test bei Student 20

Eine dritte Annahme besagt, daß Durchsichten zum Fehlerfinden nicht nur effektiv sind, sondern auch effizient. Dies kann durch Auftragen der Fehlerfindungsgeschwindigkeiten über die 10 Übungsprogramme belegt werden, wobei man die Fehlerfindungsmethoden Übersetzen, Durchsicht und Test unterscheidet. Siehe Abbildung 6.9.

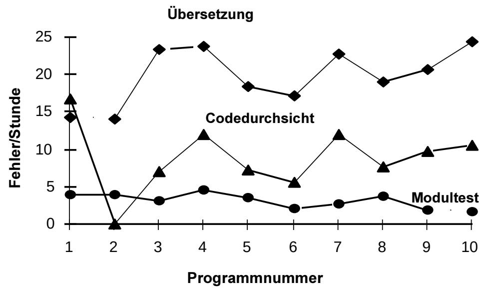

Abbildung 6.9: Vergleich der Fehlerentfernungsgeschwindigkeiten bei Übersetzung, Codedurchsicht und Modultest

6.3.3 Qualität: Effekte

Die positiven Effekte hinsichtlich der Programmqualität, die sich im Verlauf des PSP-Kurses einstellen, bedürfen kaum einer längeren Kommentierung: Das disziplinierte Vorgehen und die Durchsichten führen zu einem dramatischen Absinken des Zeitanteils, der auf die Kompilation des Programms verwendet werden muß (Abbildung 6.10).

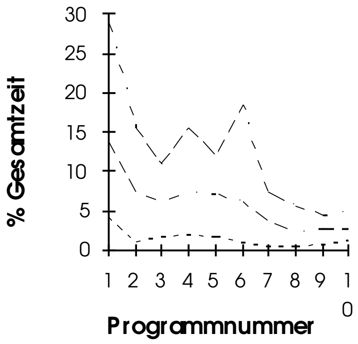

Abbildung 6.10: Entwicklung des Zeitanteils für die Übersetzung über die 10 Übungsprogramme des PSP-Kurses hinweg

Die Durchsichten führen tatsächlich mit wachsender Übung zu einer steigenden Fehlerausbeute (Abbildung 6.11)

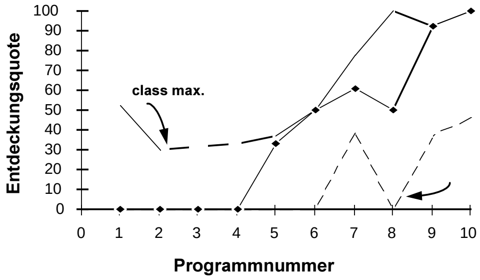

Abbildung 6.11: Entwicklung der Fehlerausbeute bei Student 3

Die im Laufe des Kurses eingeführten Qualitätsmanagementmaßnahmen reduzieren tatsächlich die Anzahl von Fehlern, die überhaupt in das Programm eingefügt werden (Abbildungen 6.12 und 6.13).

Als Ergebnis dieser Effekte sinkt die Zahl der Fehler, die im Test gefunden werden, obwohl die

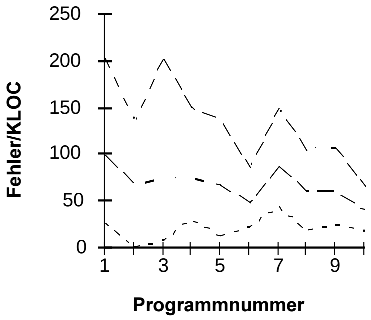

Abbildung 6.12: Entwicklung der Zahl beim Kodieren eingefügter Fehler im PSP-Kurs

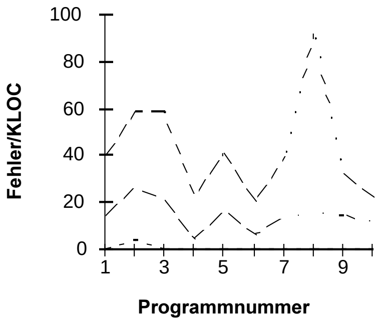

Abbildung 6.13: Entwicklung der Zahl beim Entwurf eingefügter Fehler im PSP-Kurs

Gründlichkeit der Tests über den Kurs hinweg ansteigt (Abbildung 6.14).

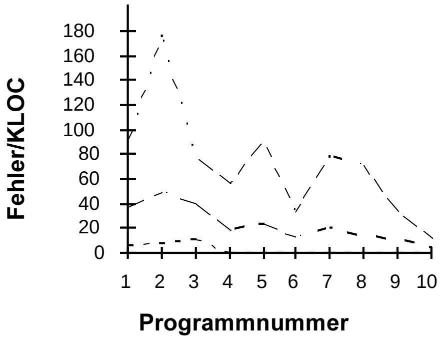

Abbildung 6.14: Entwicklung der Zahl der Testfehler im PSP-Kurs

Und schließlich: Trotz all des Verwaltungsaufwands und der zusätzlichen Arbeitsschritte, die durch den PSP bedingt werden, nimmt die Produktivität (gemessen in LOC pro Stunde) nicht ab. Das belegt die Behauptung, daß hohe Prozeßqualität auch produktivitätsfördernd ist (Abbildung 6.15). Vermutlich ist bei umfangreicheren Programmen, bei denen die Verwaltungskosten relativ weniger ins Gewicht fallen, der Effekt noch ausgeprägter; erst recht, wenn die PSP-Techniken erst einmal richtig eingeübt sind, denn während des Kurses sind die Teilnehmer darin ja alle noch Anfänger.

Wie schon weiter oben gesagt, sehen die Effekte nicht immer genau so aus, wie hier dargestellt, aber typischerweise stellen sie sich etwa in der beschriebenen Weise ein, wenn der Kurs von den Teilnehmenden wirklich ernstgenommen wird.

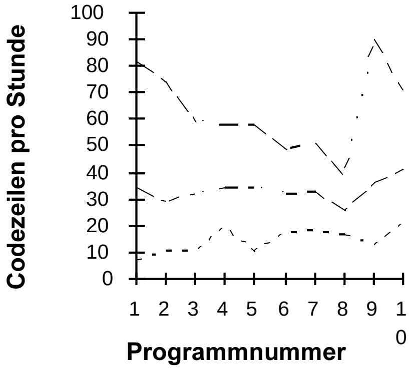

Abbildung 6.15: Entwicklung der Produktivität in LOC/Std.

6.4 Probleme und Hindernisse

Bei der Benutzung des PSP gibt es eigentlich nur ein größeres Problem: Durchhalten.

Da der PSP mit beträchtlichem Buchführungs- und Verwaltungsaufwand verbunden ist, muß der „innere Schweinehund“, der gerne Abkürzungen nehmen möchte und auf zahlreiche Aktivitäten lieber verzichtet, ständig überwunden werden. Man muß sich immer wieder den Nutzen all dieser Maßnahmen vor Augen halten, damit man auch dann den PSP durchhalten kann, wenn man nicht über eine eiserne Disziplin verfügt.

Ein weiteres, aber weniger schwerwiegendes Problem liegt in der Neigung zur Betriebsblindheit. Der Benutzer eines Prozesses kann leicht dessen Schwächen übersehen. Der vermutlich günstigste Weg, dieser Schwierigkeit zu entgehen, sind wechselseitige Prozeßbegutachtungen. Ingenieur A sichtet, analysiert und kommentiert gelegentlich den Prozeß von Ingenieur B und umgekehrt, am besten mit wechselnden Partnern, weil zusätzliche Sichtweisen zu wertvollen neuen Anregungen führen können. Da ein solches Vorgehen auch dem jeweiligen Gutachter wertvolle Ideen für seinen eigenen Prozeß vermitteln kann, ist die Methode sehr zweckmäßig und die Betriebsblindheit stellt kein schwerüberwindbares Problem dar.

6.5 Maße im PSP

Ein besonderes Merkmal des PSP ist, daß er eine Menge von recht einfachen Produkt- und Prozeßmaßen benutzt, die es trotz ihrer Einfachheit recht gut gestatten, den Prozeß zu charakterisieren und zu kontrollieren. Dieser erfreuliche Effekt rührt daher, daß durch die wohldefinierte und varianzarme Arbeitsweise (es ist ja immer die gleiche Person!) viele Daten aussagekräftig werden, die man bei projektweiter oder gar projektübergreifender Benutzung bestenfalls als dubios bezeichnen könnte.

6.5.1 Produktgröße

Die Produktgröße von Software wird im PSP durch Lines of Code gemessen. Das ist sinnvoll, weil sich diese Größe recht gut schätzen läßt. Siehe auch die Diskussion im Abschnitt 6.5.2.

Der PSP beschränkt sich allerdings nicht einfach auf die Erfassung der Gesamtproduktgröße, sondern unterscheidet die Zeilen nach ihrer Historie, in der Annahme, daß unterschiedliche Entstehungsgeschichte auch unterschiedlichen Aufwand bedeutet.

Neugeschriebene Zeilen werden als „zugefügt“ (added, A) bezeichnet, also zum Beispiel alle Zeilen eines komplett neu entwickelten Produkts. Natürlich können auch Änderungen an bestehenden Produkten vorkommen. In diesem Fall beginnt die Entwicklung mit einer gewissen Zahl von vorhandenen Zeilen B, der „Basis“ (base). Dann gibt es außer den zugefügten Zeilen auch „gelöschte“ Zeilen (deleted, D) und „geänderte“ Zeilen (modified, M). Außerdem können Programmteile unverändert wiederverwendet werden (im Gegesatz zur Basis, die verändert wird); diese werden als „wiederverwendet“ (reused, R) ebenfalls separat gezählt. Aus diesen Werten ergibt sich die Anzahl „total“ (total, T) von Zeilen zu T = B + A - D + R. Außerdem wird noch die Anzahl derjenigen Zeilen gezählt, die frisch zur späteren Wiederverwendung erzeugt wurden.

Zur Beschreibung des Arbeitsumfangs wird im Normalfall die Summe der zugefügten und geänderten Zeilen (A+M) herangezogen. Für genauere Analysen, insbesondere bei der Aufwandsschätzung werden aber auch B, D und R zusätzlich berücksichtigt.

6.5.2 Produktivität

Die Messung der Produktivität ist eine der wichtigsten Aufgaben bei der Steuerung eines Prozesses. Die Produktivität ist eine wichtige Leitgröße für die Planung und die Vorhersage der Planeinhaltung; ein Absinken der Produktivität zeigt Schwierigkeiten an, die entdeckt und behoben werden müssen, und ein Ansteigen ist das Signal für eine erfolgreiche Prozeßverbesserung.

Leider ist aus Geschäftssicht das einzige wirklich aussagekräftige Produktivitätsmaß die Menge erzeugten **Anwendernutzens** pro eingesetzter Ressourcenmenge (und zwar genaugenommen des *subjektiven* Anwendernutzens, weil dieser die Marktchancen bestimmt). Diese Größe zu messen, ist dummerweise so gut wie unmöglich. Deshalb behilft man sich üblicherweise mit **volumenorientierten** Produktivitätsmaßen, also der Menge hergestellten Programms pro Zeit. Diese Menge kann man wiederum auf eine günstige und eine ungünstige Weise bestimmen: Günstig ist, die Menge an der Zahl erfüllter Kundenanforderungen zu messen. Ungünstig ist, die Menge an der Menge erzeugten Programms zu messen, ohne direkten Bezug dazu, was dieses Programm tut. Produktivitätsmessung anhand von Kundenanforderungen sind beispielsweise mit der Funktionspunkt-Methode [AJEG83, Beh83] möglich. Ein derartiges Vorgehen hat leider den Nachteil, daß die Messung nicht automatisiert werden kann. Deshalb ist immer noch die Methode der Mengenmessung am Programmcode trotz aller ihrer Nachteile die günstigste.

Kann man dabei trotzdem von einer sinnvollen Produktivitätsmessung sprechen? Unter gewissen Voraussetzungen ja. Was Produktivität ist, ist nämlich letztlich eine Frage des Betrachtungswinkels. Subjektiver Anwendernutzen pro Ressourcenmenge ist natürlich das für die Unternehmensführung letztendlich relevante Produktivitätsmaß; im Innern eines Softwareproduktionsprozesses sind aber gewisse Randbedingungen stets schon vorgegeben. Insbesondere kann auf der Ebene, auf der ein einzelner Softwareingenieur eine Softwarekomponente entwickelt, bei einem gut strukturierten Prozeß davon ausgegangen werden, daß der Ingenieur nicht absichtlich versucht, seine Produktivität durch Herstellung großer Mengen sinnlosen Programms künstlich in die Höhe zu treiben. Tut er das aber nicht, dann ist ein Produktivitätsmaß wie Lines of Code pro Stunde (LOC/h), wie es im PSP verwendet wird, durchaus sinnvoll. Das hat zwei Gründe:

1. Das Maß verwendet zwei Maße, auf deren Basis man gut planen kann. Die Programmgröße in LOC läßt sich einigermaßen gut schätzen, wie wir in Abschnitt 6.6 sehen werden, und die Anzahl Arbeitsstunden ist die wichtigste Grundlage für die Budgetplanung. Zur Steuerung des Prozesses interessiert mich aber nicht so sehr, ob die Nutzenproduktivität ein optimales Niveau hat oder vielleicht ein wenig überflüssiger Programmcode erzeugt wird, sondern ob

die Planung eingehalten werden kann und ob sich im Rahmen der Planungsgrößen die Produktivität noch optimieren läßt, denn auf Grundlage der Planung wurden und werden ja die Geschäftsentscheidungen gefällt, die zu dem Projekt geführt haben.

2. Der zweite Zweck der Produktivitätsmessung neben der Planüberwachung besteht im Aufspüren von Problemen im Prozeß. Ein plötzliches Absinken der Produktivität an irgendeiner Stelle im Prozeß deutet normalerweise darauf hin, daß dort irgendetwas schiefläuft. Hierfür ist LOC/h ein gutes Maß, denn es charakterisiert recht gut die effektive Arbeitsgeschwindigkeit (mit nur moderater Abhängigkeit vom Arbeitsinhalt), die uns zur Prozeßbeurteilung wesentlich mehr interessiert als die sehr abstrakte Produktionsgeschwindigkeit von Anwendernutzen. Aus dem gleichen Grund eignet sich das Maß auch zur Beurteilung von Prozeßverbesserungen. Sind diese erfolgreich, so müßte LOC/h ansteigen (oder die Qualität). Diese beiden Funktionen kann LOC/h im PSP vor allem deshalb gut erfüllen, weil hier nur Daten von ein und demselben Programmierer betrachtet werden, die normalerweise recht konsistent sein sollten.

Geschwindigkeitsmessung auf Basis von LOC/h läßt sich an verschiedenen Stellen einsetzen: Für die Gesamtentwicklung, für das Kodieren, für die Codedurchsicht und für den Test. Auf gewissen Umwegen kann damit auch der Entwurfsprozeß (im Nachhinein) beurteilt werden, falls das sinnvoll erscheint.

6.5.3 Produktqualität

Von der Qualität eines Produktes läßt sich als einziger Aspekt die Zuverlässigkeit bequem messen. Der PSP verwendet hierzu eine Messung der statischen Defektanzahl; siehe den Abschnitt 7 über die Cleanroom-Technik für eine Diskussion des alternativen Ansatzes, nämlich die Messung der dynamischen Versagenshäufigkeit mittels statistischem Testen. Die **statische Defektanzahl** kann anhand der gefundenen Fehler gemessen werden. Diese Angabe ist zwar ungenau, denn es muß stets mit weiteren, noch nicht gefundenen Fehlern gerechnet werden, aber in einem hochwertigen Prozeß sollte diese Ungenauigkeit gering sein. Zwei Größen werden im PSP zur Charakterisierung der Produktqualität benutzt: Die Dichte der Fehler über alle Entwicklungsphasen (in Defekte pro 1000 Zeilen Code, defects/KLOC) und die Dichte der Fehler, die in der Testphase gefunden wurden (testdefects/KLOC). Ersteres Maß enthält zusätzlich zum zweiten auch alle Fehler, die schon während Entwurf, Kodieren und den zugehörigen Durchsichten gefunden wurden.

Warum kann man diese Werte zur Charakterisierung der Qualität des *ausgelieferten* Produkts heranziehen? Grundlage dafür ist die Beobachtung, daß jeder Defektentfernungsschritt ungefähr einen festen Prozentsatz der Fehler entfernt, beispielsweise die Hälfte, wenn ich einen vorher festgelegten Aufwand in diesen Schritt investiere. Gehe ich also mit 20 Fehlern in eine Testphase mit vordefiniertem Testprogramm, hat das ausgelieferte Programm ungefähr noch 10 Fehler; waren nur 4 vorhanden, werden ungefähr 2 ausgeliefert. Ähnliche Überlegungen gelten auch für die Defektentfernung in Durchsichten, so daß auch die Gesamtanzahl von Fehlern sinnvolle Information über die vermutliche Restfehlerzahl enthält.

Wie schon bei der Produktivität so wird auch hier die Nützlichkeit der verwendeten Maße zur Prozeßsteuerung dadurch erhöht, daß wir es mit einem PSP und also mit immer vergleichbaren Situationen zu tun haben, weil immer der gleiche Softwareingenieur zugrundeliegt.

Von konstanten Defektentfernungsquoten pro Phase kann man dann nicht ausgehen, wenn die Phasen auf die gefundene Fehlerzahl reagieren und bei hohen Zahlen ausführlichere Prüfungen einsetzen. Solches Vorgehen benutzt **Zuverlässigkeitsmodelle**, um die vermutliche Restfehlerzahl zu berechnen und könnte alternativ im PSP eingesetzt werden. Siehe Abschnitt 7.6. Eine andere Verbesserung wäre die Unterscheidung von Fehlern in z.B. drei verschiedene Schweregrade.

6.5.4 Prozeßqualität

Die Prozeßqualität spiegelt sich hauptsächlich in vier Größen wider: Produktivität und Produktqualität, wie oben gesehen, sowie Planungsgenauigkeit und Erfolg des Qualitätsmanagements.

Die Planungsgenauigkeit läßt sich pauschal in einem PSP dadurch beurteilen, daß man den Unterschied zwischen Planung und Wirklichkeit betrachtet. Der PSP schlägt dafür den *cost/performance index* (CPI) vor, der der Quotient aus geplanter und tatsächlich benötigter Zeit ist und für jede Phase separat berechnet werden kann, da jede Phase eine separate Zeitplanung hat. Der CPI sollte möglichst nah an 1 liegen; wesentlich größere oder kleinere Werte in manchen Phasen deuten auf Bereiche hin, in denen der Planungsprozeß verbessert werden sollte oder der Entwicklungsprozeß instabil ist.

Für das Qualitätsmanagement gibt es im PSP eine zentrale Meßgröße namens Ausbeute (*yield*). Die Ausbeute ist das Maß für den Erfolg der Bemühungen, Fehler bereits vor der ersten Programmübersetzung zu finden und gibt den Anteil der Programmfehler an, die vor der ersten Übersetzung gefunden wurden, von allen, die bis dahin bereits eingefügt worden sind.

Bei Auslieferung des Programms kann die Ausbeute nur geschätzt werden, weil die Zahl ausgelieferter Fehler unbekannt ist, aber eigentlich in die Berechnung eingehen müßte. Dennoch ist die Ausbeute auch in der vereinfachten Form, die nur bis zur Testphase reicht, eine extrem nützliche Größe für die Prozeßsteuerung und -beurteilung.

6.5.5 Prozeßsteuerung

Zur Beurteilung, ob der Prozeß in sinnvoller Weise abläuft, lassen sich noch weitere Größen heranziehen. Der PSP benutzt die Begutachtungsgeschwindigkeit in KLOC/h, um abzuschätzen, ob die Codebegutachtungen möglicherweise zu ausführlich oder zu flüchtig vorgenommen wurden. Ein anderes Maß für den gleichen Zweck ist das Begutachtungs-Versagens-Verhältnis (*appraisal to failure ratio*, A/FR). Dieses gibt den Quotienten aus der Zeit an, die für die Begutachtung von Entwurf und Code aufgewendet wurde, und der Zeit, die zum Übersetzen und Testen aufgewendet wurde. Letztere wird dabei als Versagenszeit betrachtet, in der Annahme, daß ihr überwiegender Teil nur deshalb aufgewendet werden muß, weil das Programm noch nicht so funktioniert, wie es nach perfekten Durchsichten eigentlich sollte. Es gibt einen optimalen Bereich für den A/FR, unterhalb dessen die Durchsichten zu flüchtig sind, so daß sich die Versagenszeit unnötig und überproportional verlängert, und oberhalb dessen der Mehraufwand in den Begutachtungen sich kaum mehr auszahlt. Dieser optimale Bereich kann identifiziert und dann im Prozeß durch Änderung der Begutachtungsgeschwindigkeit gezielt angesteuert werden.

6.5.6 Qualitätsproduktivität

Da das Einfügen und Entfernen von Fehlern einen ganz erheblichen Einfluß auf die Produktivität hat, ist es interessant, die Fehlerprozesse soweit möglich separat zu betrachten. Im Falle der Defektentfernung benutzt der PSP dafür zwei Maße.

Für die verschiedenen Phasen, die der Entfernung von Fehlern dienen, also Entwurfsdurchsicht, Codedurchsicht, Übersetzung und Test wird jeweils die Defektentfernungsgeschwindigkeit in entfernten Defekten pro Stunde (*defects removed per hour*) gemessen. Abweichungen dieser Maße von ihren üblichen Werten signalisieren einen besonders guten oder schlechten Prozeß oder ungewöhnlich gute oder schlechte Produkte und sind deshalb nützliche Prozeßkontrollgrößen.

Zur Motivation und zur Kontrolle, ob der Umfang der Durchsichten angemessen ist, dient eine weitere Größe, nämlich die Defektentfernungs-Hebelwirkung (*defect removal leverage*, DRL(A/B)), die als Quotient aus den Defektentfernungsgeschwindigkeiten zweier Phasen A und B berechnet wird, wobei normalerweise A eine Durchsichtphase ist und B der Test. Bei guten Durchsichten ist dort üblicherweise die Entfernungsgeschwindigkeit deutlich größer als im Test, was dann durch ein DRL von beispielsweise 2 oder 3 bestätigt wird.

6.5.7 Diskussion

Die Messung von Arbeitsvolumen durch LOC und Aufwand durch Zeit ist nützlich und praktikabel. Beim Arbeitsvolumen besteht die Schwierigkeit, daß das LOC-Maß nicht direkt auf Entwürfe angewendet werden kann, so daß viele Meßwerte erst recht spät zur Verfügung stehen. Allerdings kann man sich je nach Entwurfsmethode andere Maße denken, die diese Lücke schließen können, z.B. Anzahl von Methoden, Anzahl von Zuständen, etc.

Bei allen Maßen, die mit Qualität zu tun haben, ergibt sich hingegen ein Problem: Die gleichartige Gewichtung von 1 für Fehler aller Art ist irreführend. Erstens gibt es Fehler, die erheblich leichter zu finden sind als andere, zum Beispiel Fehler, die vom Übersetzer entdeckt werden oder Fehler, die fast immer zu schwerwiegendem Programmversagen führen. Es wäre also angemessen, auch unterschiedliche Entfernungskosten für solche Fehler zu erwarten. Zweitens gibt es Fehler, deren Auswirkung auf die Qualität erheblich geringer als bei anderen ist. Zum Beispiel ist ein gelegentlicher Datenverlust schwerwiegender als ein kleiner Formatierungsfehler in der Ausgabe oder die unvollständige Behandlung eines extrem seltenen Fehlerfalls.

Eine sinnvolle und zuverlässige Einteilung von Fehlern in Schwereklassen gemäß dieser Kriterien wäre also eine wichtige Erweiterung des PSP, ist aber ein ungelöstes Problem.

6.6 Größenschätzung

Sorgfältige und genaue Planung ist eines der Hauptmerkmale des PSP. Dazu gehört insbesondere eine gute Schätzung des (Zeit)Aufwandes für ein Projekt. Diese Schätzung wird im PSP auf dem Umweg über die Größe des entstehenden Programms vorgenommen, weil davon ausgegangen wird, daß erstens die Anforderungen bereits bekannt sind und zweitens die Entwicklungszeit dann recht gut mit der Größe korreliert. Diese Annahmen sind innerhalb des üblichen Anwendungsbereichs des PSP oftmals gut erfüllt.

Das große Problem beim Schätzen ist ganz allgemein, daß einerseits eine Schätzung möglichst **früh** gemacht werden soll, andererseits eine genaue Schätzung Informationen über **Details** benötigt, die so früh noch nicht vorhanden sind. Es gilt also, eine Möglichkeit zu finden, wie man das nötige Maß von Detailinformationen durch anderes ersetzen kann, um auch ohne sie zu einer genauen frühen Schätzung zu kommen. Um das zu erreichen, muß man auf direktes Schätzen verzichten und stattdessen die Schätzung über einen Vergleich mit Bekanntem vornehmen. Auf diese Weise lassen sich historische Daten über frühere Entwicklungsarbeiten nutzbringend in der Schätzung einsetzen.

6.6.1 Gesamtvorgehen

Gehen wir einmal davon aus, daß wir eine Entwicklung in einer objektorientierten Programmiersprache schätzen sollen (andernfalls läßt sich das Verfahren anpassen, indem wir Module statt Objektklassen benutzen etc.). Dann funktioniert das PSP-Größenschätzverfahren folgendermaßen: Ausgangspunkt der Schätzung ist ein sogenannter **konzeptueller Entwurf**. Ein solcher identifiziert die **Hauptkomponenten**, die zur Lösung des Problems vermutlich benötigt werden, z.B. auf der Ebene von **Objektklassen** und beschreibt grob deren Funktionalität. Der Zweck des konzeptuellen Entwurfs ist lediglich die Größenschätzung; er stellt keine Festlegung des tatsächlichen Entwurfs dar, obwohl er in der Regel natürlich eine gute Grundlage dafür bildet. Bei größeren Aufgaben kann die Herstellung eines konzeptuellen Entwurfs eine beträchtliche Arbeit sein, die mehrere Tage dauert, weil man bis zu einer Ebene von Komponenten herunterbrechen muß, die man gut schätzen kann; aber in jedem Fall steht der Aufwand in einem sinnvollen Verhältnis zum Gesamtaufwand. Im konzeptuellen Entwurf läßt sich auch etwa abschätzen, welche Teile eines bestehenden Programms als **Basisprogramm** übernommen werden können und wie umfangreich die **Änderungen** daran sind, wenn das Problem die Erweiterung eines bestehenden Programms ist, und wie viele andere bestehende Komponenten in welchem Gesamtumfang unverändert **wiederverwendet** werden können.

Die Schätzungen von Basis, Löschungen, Modifikationen, Zufügungen und neuen Objekten werden zusammengerechnet und ergeben die initiale Schätzung. Diese wird schließlich mit Hilfe historischer Daten über die Genauigkeit früherer Schätzung **korrigiert**, indem man eine **lineare Regression** berechnet, die die initiale Schätzung auf die wirkliche Schätzung abbildet. Diesen Schätzprozeß werden wir nun etwas genauer besprechen, wobei wir uns auf folgendes Beispiel beziehen:

```
Base Program                                          LOC
  Base size (B)                                       295
  LOC deleted (D)                                      25
  LOC modified (M)                                     30
Projected LOC (P)
  Base_additions (BA)      (Type) (#meth) (size)      LOC
    readFromNetwork                                    40
    openAgain                                          15
  Total_base_additions ___________________________     55
  New_objects (NO)          Type   #meth   size       LOC
     Networkcomm             I/O     10      M        160
     BasicNetwork            I/O      8      S         96
     Crypt                   Comp     3      L         75
  Total_new_objects (NO)__________________________    331
Reused objects                                        LOC
     Longint                                          780
  Reused_total (R)                                    780
Estimation:
  Projected LOC (P)   BA+NO                           386
  Estimated New&changed LOC (N)  beta0+beta1(P+M)     488
  Estimated total LOC (T)  B+N-D-M+R                 1508
  Prediction interval                           448 - 528
  Prediction interval level (%)                        70
```

Hierbei handelt es sich um ein Schätzformular, das im Laufe der Schätzprozedur ausgefüllt wird. Nun also das Vorgehen noch einmal in Schritten:

1. Erstelle einen konzeptuellen Entwurf.

2. Identifiziere ggf. das Basisprogramm, daß sich als Grundlage der neuen Entwicklung verwenden läßt und trage seine Größe unter B ein (hier: 295 LOC).

3. Schätze, wie viele der Zeilen des Basisprogramms gelöscht oder geändert werden müssen und trage diese Werte unter D bzw. M ein (hier: 25 bzw. 30 LOC).

4. Schätze, wie viele Zeilen am Basisprogramm vermutlich zugefügt werden, um die Aufgabe zu lösen. Diese Schätzung geht normalerweise Prozedur für Prozedur vor und schätzt die Länge ganz neuer Prozeduren oder die Länge von Zufügungen bei Prozeduränderungen. Ggf. kann man auch ähnlich vorgehen, wie bei der Schätzung der Objektklassengröße unten. Trage diese Zufügungen unter BA (base additions) ein. Im Beispiel gibt es zwei neue Prozeduren im Umfang von 40 und 15 LOC, insgesamt ist also BA gleich 55.

5. Schätze nun den Umfang der noch ganz neu zuzufügenden Klassen, falls nötig. Dies ist bei umfangreicheren Erweiterungen eines Basisprogramms meist nötig und enthält sogar den gesamten Programmumfang, wenn es gar kein Basisprogramm gibt. Für jede Klasse gebe einen Namen an, eine Klassenart, die voraussichtliche Anzahl von Methoden und die Größenklasse:

   - Die Klassenart unterscheidet eine kleine Zahl verschiedener Sorten von Klassen, die unterschiedlich geschätzt werden sollten. Zum Beispiel kann man unterscheiden zwischen Ein/Ausgabeklassen, Berechnungsklassen, Kontrollflußklassen, Datenklassen, Initialisierungsklassen, etc. Die genaue Wahl dieser Klassenarten muß man für die individuellen Bedürfnisse und insbesondere den Anwendungsbereich, in dem man üblicherweise arbeitet, passend selbst wählen. Insbesondere kann man anfangs, wenn noch keine ausreichenden historischen Daten vorliegen, ganz auf eine solche Unterscheidung verzichten.
   - Die voraussichtliche Zahl von Methoden fügt eine weitere Zerlegungsebene in die Schätzung ein, indem nicht die Größe der ganzen Klasse in einem Schritt geschätzt werden muß, sondern nur die mittlere Größe der Methoden und eben deren Anzahl.
   - Die Größenklasse charakterisiert genau diese mittlere Größe der Methoden. Wir benutzen nur die fünf Größenklassen sehr klein (very small, VS), klein (S), mittel (M), groß (L) und sehr groß (VL) und benutzen historische Daten, um zu bestimmen, wie groß eine Methode in jeder dieser Klassen typischerweise ist. Das Verfahren hierzu wird unten im Abschnitt 6.6.2 beschrieben und liefert einen LOC-Wert.
   - Das Produkt aus LOC-Wert der Größenklasse (für die gegebene Klassenart) und Anzahl der Methoden ergibt die geschätzte Größe der Klasse, die im Formular eingetragen wird.

   Im Beispiel haben wir drei neue Klassen: eine Ein/Ausgabeklasse `Networkcomm` mit mittlerer Größe der Methoden und voraussichtlich 10 Methoden, die insgesamt ungefähr 160 LOC lang werden müßte; eine Ein/Ausgabeklasse `BasicNetwork` mit 8 kleinen Methoden (insgesamt 96 LOC) und eine Berechnungsklasse `Crypt` mit 3 großen Methoden (insgesamt 75 LOC). Zusammen ergeben diese drei Klassen eine Zufügung von „new objects“ (NO) von 331 LOC.

6. Schließlich notieren wir noch, daß wir im Beispiel voraussichtlich eine existierende Klasse wiederverwenden können, nämlich die Klasse `Longint` im Umfang von 780 LOC.

7. Nun berechnen wir durch Addition den Schätzwert für die Zahl zugefügter Zeilen zu 55+331=386 LOC. Addieren wir hierzu die Zahl zu verändernder Zeilen, so erhalten wir die initiale Schätzung von 416.

8. Diese Schätzung wird nun mit der linearen Regression korrigiert, um systematische Fehler auszugleichen, die vielleicht in der Schätzung stecken. Das genaue Verfahren steht unten im Abschnitt 6.6.3, jedenfalls werden historische Daten verwendet, um die Korrekturparameter zu erhalten und die korrigierte Schätzung zu ermitteln. Angenommen, unsere historischen Daten zeigen in einer linearen Regression an, daß die Schätzungen im Mittel um 10 Prozent und zusätzlich 30 Zeilen zu niedrig liegen, dann ergibt sich die korrigierte Schätzung zu 30+1.1*416=487.6 LOC. Zugleich berechnen wir dabei das Vorhersageintervall (prediction interval), das den Bereich angibt, in den der tatsächliche Wert von N nach den bisherigen Erfahrungen mit einer Wahrscheinlichkeit von im Beispiel 70% fallen wird. Dieses Vorhersageintervall wird unter Benutzung der t-Verteilung aus den gleichen historischen Daten berechnet, mit denen auch die lineare Regression durchgeführt wurde. Angenommen, die historischen Daten zeigen recht gleichmäßige Schätzungen an, also geringe Streuungen, dann könnte das 70%–Vorhersageintervall, wie im Beispiel angenommen eine Breite von 80 LOC haben, so daß wir ein Vorhersageintervall von 488-40 bis 488+40 LOC erhalten.

Voila!

Mit Hilfe einer zweiten linearen Regression kann man jetzt eine Umrechnung der geschätzen Größe in die zu erwartende Entwicklungszeit vornehmen, so daß man auch gleich eine Aufwandsschätzung erhält; siehe den Abschnitt 6.6.4.

Der Trick bei dieser Schätzmethode ist die geschickte Auswahl von Stellvertretern (Ersatzgrößen), um auf das in der Schätzphase noch fehlende Detailwissen verzichten zu können ohne viel Schätzgenauigkeit einzubüßen. Anstatt direkt den Umfang zu schätzen, benutze ich eine Zerlegung in Teile, die ich mir noch recht gut vorstellen kann. Ich schätze deren Größe dann zum Teil intuitiv, indem ich nur fünf grobe Größenklassen verwende, und benutze historische Daten, die den Inhalt meiner Intuition widerspiegeln, um zu einer guten Schätzung zu gelangen.

6.6.2 Bestimmung der Objektgrößenbereiche

Wie kommt man nun zu sinnvollen LOC-Werten für die einzelnen Größenklassen? Ausgangspunkt ist die Überlegung, daß die Werte möglichst meine intuitive Vorstellung davon, was „klein“ oder „mittel“ etc. ist, wiedergeben sollten. Diese Vorstellung hängt von Häufigkeiten ab: Das, was oft vorkommt, ist mittel, etwas größeres, was nur mäßig oft vorkommt, ist groß, und nur etwas recht seltenes kann sehr groß genannt werden; für klein und sehr klein geht es analog.

Für eine Normalverteilung lassen sich recht einfach einleuchtende Bereiche für eine solche Einteilung angeben: Etwa 38.3% aller Fälle liegen im Bereich von plus oder minus einer halben Standardabweichung um den Mittelwert, also in einem Bereich der Breite einer Standardabweichung; diese nenne ich „mittel“. Als „groß“ bzw. „klein“ bezeichne ich die Bereiche, die mehr als eine halbe, aber weniger als eineinhalb Standardabweichungen vom Mittelwert entfernt liegen; diese umfassen je 24.2% aller Fälle. Die verbleibenden zweimal 6.7% aller Fälle verteilen sich auf die Bereiche, die mehr als eineinhalb Standardabweichungen vom Mittelwert entfernt sind; diese nenne ich „sehr groß“ und „sehr klein“.

Also rechnen wir einfach Mittelwert m und Standardabweichung s unserer Methodengröße aus und erhalten die Grenzen der Bereiche wie oben angegeben? Als Schätzwerte für die Bereiche könnten wir dann deren Mitte verwenden, also m-2s, m-s, m, m+s, m+2s.

Leider nein, so geht es nicht, denn Methodengrößen sind üblicherweise nicht normalverteilt. Aber sie sind meistens ungefähr log-normal verteilt, so daß wir uns mit einem Trick behelfen können. Log-normale Verteilung bedeutet, daß die Logarithmen der Werte eine Normalverteilung haben. Dieser Effekt tritt sehr oft bei Größen auf, die zwar eine natürliche untere Schranke haben (zum Beispiel Null), aber keine obere und die dieser unteren Schranke häufig nahekommen. Somit können wir die ganze obige Prozedur mit den Logarithmen der Methodengrößen vornehmen und am Schluß delogarithmieren, um die Schätzgrößen jeder Größenklasse zu erhalten.

Beispiel: Gegeben sei eine Reihe von Klassen, die unsere historischen Daten für eine bestimmte Klassenart repräsentieren. Die Werte für die LOC pro Methode lauten für diese 13 Klassen 10.33, 6, 21.75, 29, 8.33, 6, 22.25, 28.33 12.33, 55.8, 20.5, 16.4, 23. Das ergibt einen Mittelwert von 20 und eine Standardabweichung von 12.84. Die zugehörigen Logarithmen (wir verwenden hier den natürlichen Logarithmus, man könnte aber auch einen anderen benutzen) lauten 2.335, 1.792, 3.08, 3.367, 2.12, 1.792, 3.102, 3.344, 2.512, 4.022, 3.02, 2.797, 3.135. Hier sind der Mittelwert 2.802 und die Standardabweichung 0.6346.

Wir erhalten also als Bereichsgrößen im logarithmischen Bereich (also in der Maßeinheit logLOC) die Werte
VS = 2.802 - 2*0.6346 = 1.533
S = 2.802 - 0.6346 = 2.167
M = 2.802
L = 2.802 + 0.6346 = 3.437
VL = 2.802 + 2*0.6346 = 4.071
und durch berechnen des inversen Logarithmus erhalten wir daraus für den nichtlogarithmischen Bereich (also in der Maßeinheit LOC) die Werte
VS = exp(1.533) = 4.63
S = exp(2.167) = 8.73
M = exp(2.802) = 16.48
L = exp(3.437) = 31.09
VL = exp(4.071) = 58.62

Wie wir sehen, sind die Werte von VS und VL jenseits des Bereichs, der von den Daten überhaupt erreicht wird (6.0 bis 55.8). Das ist nicht verblüffend, denn bei 13 Datenwerten repräsentiert ein Wert immerhin 7.7% und der ganze VS- bzw. VL-Bereich deckt ja nur 6.7% aller Werte ab.

Vor der Anwendung der beschriebenen Methode sollte man sicherstellen, daß die benutzten Daten auch tatsächlich eine ungefähr lognormale Verteilung haben. Dazu ist insbesondere zu fordern, daß genügend Daten vorhanden sind; mit viel weniger als den obigen 13 Punkten sollte man nicht anfangen. Wenn nicht genügend Werte da sind, kann man entweder die Größenbereiche per Intuition schätzen (oder die berechneten per Intuition korrigieren) oder gleich die Schätzung nicht über einen Größenbereich ausrechnen, sondern direkt vornehmen, also „diese Klasse hat vermutlich 10 Funktionen zu je, sagen wir mal 20 Zeilen".

Warum soll man die Einteilung der Objektgrößenklassen getrennt für verschiedene Arten von Objektklassen vornehmen? Weil diese recht unterschiedliche Eigenschaften haben können im Hinblick nicht nur auf die absoluten Größen, sondern auch auf die Streuung zwischen den Größenklassen. Hier als Beispiel die Werte, die Humphrey für seine eigenen Größenklassen für C++ angibt:

Art	VS	S	M	L	VL
Comp	2.3	5.1	11	25	54
I/O	9	12	16	22	29

Die Berechnungsklassen haben also zwar geringere „klein"- und „mittel"-Größen, aber dennoch größere „groß"-Größen, weil ihre Streuung erheblich breiter ist als bei den Ein-/Ausgabeklassen.

6.6.3 Schätzkorrektur mit Regression

Der Verwendung der linearen Regression liegen zwei Überlegungen zugrunde.

1. Es gibt im obigen Meßverfahren einen kleinen eingebauten systematischen Fehler: Wenn man die Größe von Objektklassen nur über die Größe der Methoden berechnet, vernachlässigt man den Datentyp selbst, der mit der Klasse definiert wird. Dadurch fallen die Schätzungen immer ein wenig zu niedrig aus.

2. Außerdem, und das ist der bei weitem wichtigere Teil, machen Schätzer meist auch systematische Fehler. Viele Schätzer übersehen zum Beispiel gern Komponenten, so daß ihre Schätzungen zu niedrig werden oder sie veranschlagen die Größe der Komponenten immer etwas zu gering (andere machen sie zu hoch). In der Summe dieser Effekte ergibt sich jedenfalls neben allen Schwankungen in den Schätzungen, die von Fall zu Fall verschieden sind (also statistischer Fehler) ein gewisser Teil systematischer Fehler. Dieser läßt sich durch Vergleich mehrerer Schätzungen mit der Wirklichkeit berechnen und dann in späteren Schätzungen korrigieren.

Die Korrektur dieser zwei Teile des systematischen Fehlers leistet die lineare Regression. In der Terminologie des Abschnitts über Vorhersagesysteme (1.6.1) suchen wir also ein Vorhersagesystem zur Vorhersage der tatsächlichen Größe aus der geschätzten Größe. Wir verwenden dafür als Modell die Vorstellung „wenn Du als Größe s geschätzt hast, dann beträgt die tatsächliche Größe $g = a^*s + b$". Die Berechnungsprozedur zur Bestimmung der günstigsten Werte für a und b ist die lineare Regression auf Basis vergangener tatsächlicher Paare von Schätzung und wirklicher Größe. Die Berechnung geht genauso vor sich, wie im obenerwähnten Abschnitt gezeigt.

Solange die historischen Daten noch knapp sind, kann sich durch jeden neuen Datenpunkt eine spürbare Veränderung im Vorhersagesystem ergeben. In diesem Fall sollte das Vorhersagesystem also für jede Schätzung neu berechnet werden und zwar unter Verwendung aller bisher vorhandenen Datenpaare. Im Laufe der Zeit werden dann Änderungen immer seltener nötig, nehmen jedoch noch einen zusätzlichen Zug an: Nach einer Weile wird sich die Schätzpraxis verändern, weil man mehr und mehr Erfahrung im Schätzen sammelt. Dann spiegeln alte Datenpaare nicht mehr die gegenwärtige Schätzpraxis wider und sollten lieber nicht mehr bei der Berechnung des Vorhersagesystems benutzt werden. Es werden also im Laufe der Zeit nicht nur neue Datenpaare bei der Berechnung mit eingeschlossen, sondern irgendwann auch alte ausrangiert. Welche das sind und wann man sie ausschließt, bleibt dem Urteilsvermögen überlassen, wobei man sich durch eine graphische Betrachtung der Punktwolken der jüngsten und ältesten Datenpaare (in zwei verschiedenen Farben) helfen kann: Liegen die Punktwolken deutlich getrennt, so hat sich tatsächlich etwas geändert und die alten Punkte sollten nicht mehr benutzt werden.

6.6.4 Schätzung von Zeit aus Größe

Aus der Größenschätzung kann man ebenfalls mit einer linearen Regression auch eine Zeitschätzung erhalten, wenn man genügend historische Daten über diesen Zusammenhang hat und diese Daten eine ausreichend hohe Korrelation zwischen den beiden Größen zeigen, was üblicherweise der Fall ist, wenn die Größe anständig geschätzt wird.

Diese Berechnung ist genau der Fall, der im Abschnitt 1.6.1 besprochen wurde, nur daß nicht tatsächliche Größendaten als Eingabe dienen, sondern Größenschätzungen.

6.7 Entwurfsprüfung

For every problem there is a solution
that is simple, neat, and wrong.

Zu den wesentlichen Bestandteilen des PSP gehört das Fehlermanagement, das versucht, Fehler zuallererst zu vermeiden, und falls sie doch auftreten, sie so früh wie möglich wieder zu entdecken und zu beseitigen. Die wichtigsten Mittel zum Fehlerfinden sind dabei die Durchsichten (reviews). Üblicherweise wird im PSP eine Entwurfsdurchsicht und eine Codedurchsicht durchgeführt. Unter Entwurf wird im PSP nicht nur der Grobentwurf verstanden, also die Definition von Modul- oder Klassenschnittstellen, sondern auch der **Feinentwurf**, in dem Datenstrukturen entworfen werden und die inneren Funktionsweisen der Prozeduren oder Methoden, in schwierigen Fällen bis hin zu Pseudocode.

Der PSP schlägt keine Entwurfsmethode vor; diese soll jeder Programmierer passend zu seinen eigenen Vorlieben, Bedürfnissen und seinem Anwendungsgebiet aussuchen. Der PSP empfiehlt aber eine Reihe von Methoden, um Entwürfe auf Korrektheit zu prüfen und zwar unabhängig von der Entwurfsmethode. Dazu gehören zum Teil auch **Notationen**, mit denen bestimmte Aspekte von Entwürfen dargestellt werden können, soweit die jeweilige Entwurfsmethode nicht eine alternative Darstellung für den gleichen Zweck schon bereithält.

Die empfohlenen Notationen sind

- Mengenausdrücke.
- Boolesche Ausdrücke.
- Karnaugh-Veitch Symmetriediagramme (auch bekannt als KV-Diagramme, KVS-Diagramme, Karnaugh maps) zur Darstellung von aussagenlogischen Formeln.
- Prädikatenlogik.
- Endliche Automaten.

Alle diese Notationen bis auf die letzte zielen vor allem auf die Anwendung zur Formulierung von Bedingungen, insbesondere Vor- und Nachbedingungen sowie Bedingungen für Fallunterscheidungen und Zustandsübergänge. Mengenlogische, aussagenlogische und prädikatenlogische Ausdrücke erlauben insbesondere die kompakte Darstellung von Bedingungen; KV-Diagramme sind eine gute Methode, aussagenlogische Formeln zu vereinfachen oder in eine standardisierte Form zu bringen. Alle diese Notationen und Techniken dürften hinreichend bekannt sein.

Endliche Automaten eignen sich in sehr vielen Fällen zur Darstellung des Verhaltens von zustandsbehafteten Modulen oder Datenobjekten. Der Vorteil dieser Darstellung besteht darin, daß sich dafür Methoden angeben lassen, mit denen leicht Plausibilitätsprüfungen für einen Entwurf vorgenommen werden können, die viele Entwurfsfehler aufdecken. Dazu mehr in einem separaten Abschnitt unten.

Folgende **Entwurfsprüfungsmethoden** werden im PSP vorgeschlagen:

- Zustandsmaschinenprüfung zur Prüfung der Vollständigkeit und Eindeutigkeit von Zustandsautomaten.
- Entwurfskonventionen zur Kontrolle von Sorgfalt und Zusammenhang des Entwurfs.
- Symbolische Ausführung zur Prüfung von algorithmischer Programmlogik.
- Induktionsbeweis zur Prüfung von Schleifen oder Rekursionen (Benutzung zusammen mit Verfolgungstafeln).
- Ausführungstabellen (execution tables) als letzter Ausweg zur Prüfung beliebiger Programmlogik.
- Verfolgungstabellen (trace tables) zur Prüfung von komplexer Programmlogik aller Art (Ausführungstabellen zusammen mit symbolischer Ausführung und Induktionsbeweisen).
- Schleifenverifikation zur Prüfung von Schleifenbedingungen und Schleifenfortschaltkonstruktionen (formal oder halbformal).
- Formale Verifikation zur Prüfung formal spezifizierter Routinen wo möglich.

Im folgenden werden wir nach einigen allgemeinen Überlegungen zur Entwurfsprüfung und zur Auswahl von Prüfungsmethoden alle oben angegebenen Methoden kurz in einem eigenen Abschnitt ansprechen.

6.7.1 Allgemeine Überlegungen

Bei der Auswahl und dem Einsatz von Entwurfsprüfungsmethoden sollten einige Punkte stets beachtet werden. Erstens und vor allem sind in der Praxis immer die **Kosten** einer Entwurfsprüfung von Bedeutung. Deshalb scheiden manche Methoden eventuell aus, obwohl sie im Prinzip nützlich sein könnten. Die Kostengesichtspunkte bei der Auswahl von Methoden gelten aber nicht gleichmäßig für den ganzen Entwurf, sondern jeder Entwurf hat **kritische Abschnitte**, von denen klar ist, daß sie im Vergleich zum Rest besonders gut geprüft werden sollten, entweder, weil sie besonders fehlerträchtig sind, oder weil sie besonders kritisch für das Funktionieren des Programms sind. Für diese Abschnitte sind höhere Prüfungskosten akzeptabel als für den Rest. Das richtige Identifizieren kritischer Abschnitte in einem Entwurf ist eine Fertigkeit, die ein Entwerfer lernen kann und muß. Manche Methoden könnte man auf denselben Abschnitt in unterschiedlich großer **Detailtiefe** anwenden. Auch hierbei ist ein Gespür für das Kosten-Nutzen-Verhältnis als Leitmaßstab notwendig.

Zweitens ist die **Eignung** der Methoden für einen Anwendungsfall unterschiedlich. Oft lassen sich zwar im Prinzip mehrere Methoden anwenden, aber manche sind erheblich einfacher oder ergiebiger als andere. Diese Auswahl ist nur manchmal schwierig, meist liegt es auf der Hand; insbesondere lassen sich viele der unten besprochenen Methoden nur bei recht fein spezifizierten Entwürfen einsetzen (Pseudokode).

Drittens ist es wichtig, die eigene **Beherrschung** einer jeden Methode für einen Anwendungsfall einschätzen zu können. Es hat keinen Sinn, eine Methode einzusetzen, mit der man bei der Entwurfsprüfung mehr Fehler macht als beim Entwurf. Bei der Auswahl zwischen zwei Methoden ist eventuell die eigentlich etwas weniger geeignete der beiden vorzuziehen, wenn man sie nämlich erheblich sicherer oder schneller zu benutzen versteht. Als allgemeine Regel gilt, daß ein Entwurf, zu dem man keine Prüfungsmethode gut genug beherrscht, um ihn zuverlässig prüfen zu können, zu kompliziert und deshalb schlecht ist; er sollte nach Möglichkeit komplett neu gemacht werden.

6.7.2 Zustandsmaschinen als Notation und Prüfungsmethode

Die meisten zustandsbehafteten Objekte oder Module in Programmen haben eine so große Zahl von Zuständen, daß eine Behandlung aller dieser Zustände in einer Zustandsmaschine hoffnungslos und zudem unsinnig wäre. Es gibt aber innerhalb dieser Zustandsmenge in den allermeisten Fällen eine recht kleine Zahl von disjunkten Teilmengen, deren Vereinigung die gesamte Menge ist und die jeweils für ein qualitativ unterschiedliches Verhalten des Objekts oder Moduls stehen. Eine Modellierung des Zustandsautomaten über diese qualitativ unterschiedlichen Zustände reduziert die Zustandsbetrachtung auf das wesentliche und erlaubt eine Prüfung des Entwurfs auf Vollständigkeit und Konsistenz mit moderatem Aufwand.

Welches sind diese qualitativ verschiedenen Zustände? Wie findet man sie? Leider kann man hierfür keine einfachen Regeln angeben. Als Richtschnur kann man sagen, daß die Unterscheidung der Zustände möglichst oft mit den Fallunterscheidungen zusammenfallen sollte, die in der Programmlogik notwendig sind, und daß die Gesamtzahl der Zustände klein bleiben muß (unter 10), weil sonst die vollständige Prüfung zu aufwendig wird. Welche zusammengefaßten Zustände am besten geeignet sind, ist schwer zu sagen. Manchmal ergeben sich mehrere Möglichkeiten zur Einteilung, die sich gegenseitig ausschließen. Dann kann man mehrere dieser Möglichkeiten getrennt voneinander zur Prüfung heranziehen.

Die Beschreibung des Objekts oder Moduls als Zustandsmaschine gibt zunächst einmal an, aus welchen Zuständen welche Operationen zu welchen anderen Zuständen führen; also die Zuständsübergänge.

Hier ein sehr simples Beispiel: Wir betrachten eine Stapelklasse, deren Realisierung bis zu `maxentries` Stapeleinträge zulassen soll. Sie habe die Operationen `empty` (ist der Stapel leer?), `top` (sag mir das oberste Element), `push` (packe ein Element auf den Stapel) und `pop` (entferne das oberste Element vom Stapel und gib es mir).

Dann empfiehlt es sich, drei Zustände zu unterscheiden. `none`: Der Stapel ist leer, also `entries = 0` (Fallunterscheidungen bei `empty`, `top` und `pop`). `full`: der Stapel ist ganz voll, also `entries = maxentries` (Fallunterscheidung bei `push`). `some`: der Stapel ist teilweise voll (Fallunterscheidung bei `empty`).

Die Zustandsübergänge lassen sich folgendermaßen beschreiben: Vom Zustand `none`
in den Zustand `none`: `empty, top, pop`;
in den Zustand `some`: `push`;
in den Zustand `full`: kommt nicht vor.
Vom Zustand `some`
in den Zustand `none`: `pop` wenn `entries = 1`;
in den Zustand `some`: `push` wenn `entries < maxentries - 1`, `pop` wenn `entries > 1`, `empty`, `top`;
in den Zustand `full`: `push` wenn `entries` gleich `maxentries - 1`.
Vom Zustand `full`
in den Zustand `none`: kommt nicht vor;
in den Zustand `some`: `pop`;
in den Zustand `full`: `push, empty, top`.

Was haben wir übersehen?
Die obige Tabelle stimmt nicht, wenn **maxentries** gleich 1 ist. Macht das was? Nein.

Diese Beschreibung gibt uns nun eine Anleitung für den Entwurf der Operationen. Erstens müssen wir dafür sorgen, daß für jede Operation für jeden Ausgangszustand eine Aktion vorgesehen ist. Dies ist die **Vollständigkeit** des Entwurfs bzw. zunächst die Vollständigkeit der Zustandsmaschine. Zweitens dürfen sich die Zustände nicht überlappen. Dies ist die **Orthogonalität**. Drittens muß die Aktion jeweils in den richtigen Folgezustand führen. Dies ist die **Korrektheit** der Zustandsmaschine und zusammen mit den übrigen Wirkungen der Aktion die Korrektheit der Operation. Viertens dürfen nicht mehrere Folgezustände für den gleichen Fall spezifiziert sein. Dies ist die **Konsistenz**.

Eine vollständige Zustandsmaschine mit orthogonalen Zuständen nennen wir eine ordentliche Zustandsmaschine. Eine solche ist eine brauchbare Grundlage für einen Entwurf. Der erste Schritt bei der Entwurfsprüfung besteht also darin zu prüfen, ob alle Zustandsmaschinen vollständig und orthogonal sind.

Der Entwurf selbst kann wieder in einem ähnlichen Format an den Zuständen orientiert aufgeschrieben werden Beispiel:
Die Operation `empty`
im Zustand `none`: true.
im Zustand `some`: false.
im Zustand `full`: false.
Die Operation `top`
im Zustand `none`: Fehler.
im Zustand `some`: Oberstes liefern.
im Zustand `full`: Oberstes liefern.
Die Operation `push`
im Zustand `none`: aufstapeln.
im Zustand `some`: aufstapeln.
im Zustand `full`: Fehler.
Die Operation `pop`
im Zustand `none`: Fehler.
im Zustand `some`: entstapeln.
im Zustand `full`: entstapeln.

In diesem Format läßt sich die Vollständigkeit leicht prüfen, so daß typische Unterlassungsfehler nicht so leicht unentdeckt bleiben. Zum Beispiel könnte man in einem entsprechend komplizierteren Fall leicht eine Prüfung wie die auf den Überlauf bei `push` vergessen. Die explizite Angabe des Folgezustands wurde hier ebenso wie die genauere Beschreibung der Operation selbst unterlassen. In komplizierteren Fällen wird man auch dies noch tun, um die Prüfung der Korrektheit zu ermöglichen.

Meist kann man diese Darstellung wieder vereinfachen, indem man Zustände noch weiter zusammenfaßt. Beispielsweise reicht bei `push` der `some`-Fall für alle drei Zustände aus, wenn man ihn als „Fehler falls Stapel voll, aufstapeln sonst“ formuliert.

Ein Vorgehen mit dieser Notation steigert einerseits die Disziplin bei der Erstellung des Entwurfs und vermeidet so Fehler und ermöglicht andererseits ein systematisches Vorgehen bei der Prüfung von Vollständigkeit und Korrektheit der Realisierung des Zustandsautomaten. Ein weiterer Vorteil liegt darin, daß die Anwendung dieser Methode bei zu „cleveren“ Entwürfen, die mit raffinierten Tricks zu arbeiten versuchen, recht schwierig wird, so daß die Versuchung, solche schwerer verständlichen und fehleranfälligeren Konstruktionen zu benutzen, sinkt. Es sei betont, daß das obige Beispiel nur zur Illustration des Vorgehens dient; die Vorteile der Methode werden dabei nur zu kleinen Teilen sichtbar, sie entfalten sich erst in komplizierteren Fällen. Die Synthese und Analyse von Klassenentwürfen mit geeignet ausgewählten Zustandsmaschinen ist eine der leistungsfähigsten Methoden, um Unterlassungen und Inkonsistenzen in solchen Entwürfen aufzudecken.

6.7.3 Entwurfskonventionen

Die Einhaltung von Konventionen zu prüfen, kann ein wesentlicher Beitrag zur Qualitätsverbesserung eines Entwurfs sein. Zwei Arten von Konventionen sind vor allem zu beachten, nämlich Anforderungen an das entworfene Produkt und Anforderungen an das Entwurfsdokument als solches.

Produktkonventionen dienen dazu, das äußere Erscheinungsbild des Softwareprodukts gegenüber dem Benutzer zu vereinheitlichen. Bei interaktiver Software fallen hierunter Vereinbarungen über das Aussehen, den Inhalt und die Benutzung von Menüs und Dialogboxen, bei Bibliotheken sind es Namensgebungs- und Aufrufkonventionen und so fort. Die Prüfung von Produktkonventionen verbessert den Entwurf in zweierlei Weise: erstens durch bessere Einhaltung der Konventionen selbst, die als Qualitätsmerkmale der Software gar nicht unbedeutend sind und zweitens dadurch, daß eine Verletzung von Produktkonventionen häufig anzeigt, wo insgesamt beim Entwurf weniger sorgfältig gearbeitet wurde und vermutlich noch weitere Probleme zu erwarten sind, insbesondere übersehene oder mißverstandene Anforderungen.

Standards im Entwurfsdokument dienen dazu, die Lesbarkeit des Dokuments zu verbessern, die Orientierung darin zu erleichtern und Mißverständnisse seines Inhalts zu vermeiden. Das ist zwar in der Phase der Herstellung der Software eventuell noch nicht sehr wichtig, insbesondere, wenn der Entwerfer den Entwurf selbst realisiert, bekommt aber in jedem Fall in der Wartung erhebliche Bedeutung. Auch hier deuten Abweichungen von den Konventionen auf Mängel im Entwurf hin und eignen sich deshalb neben ihrer direkten Bedeutung als äußerliches Prüfkriterium zum Aufspüren sonstiger Entwurfsprobleme.

Eine dritte Klasse von Konventionen, deren Prüfung sich lohnt, können Richtlinien über die Gestaltung wiederverwendbarer Komponenten sein. Zur Entwurfsprüfung gehört dann einerseits die Prüfung, ob die neu entworfenen Komponenten, welche zur Wiederverwendung gedacht sind, gemäß dieser Konventionen entworfen wurden und andererseits, ob die Verwendung existierender Komponenten, die im Entwurf vorgesehen ist, diesen Konventionen entspricht.

6.7.4 Symbolische Ausführung und Induktionsbeweis

Symbolische Ausführung eignet sich manchmal zur Prüfung exakt spezifizierter kurzer Anweisungsabschnitte. Die Idee hierbei lautet, die Variablen nicht durch ihre Werte zu ersetzen, sondern für sich selbst stehen zu lassen, mit dem Wert, den sie am Eingang des Programmstücks haben, und bei jeder Operation dann mit Variablenbezeichnern zu rechnen, so wie man es aus der Mathematik kennt. Beispiel: Das Programmstück

```
a := b + 1;      (* now a = b+1 *)
c := 2*a;        (* now c = 2b+2 *)
IF a > 7 THEN
  c := c + 1     (* now c = c+1 if b+1 > 7  and c = 2*b+2 otherwise *)
ELSE
  a := b         (* now a = b   if b+1 <= 7 and a = b+1 otherwise *)
FI;
```

liefert als Ergebnis, daß

```
a = IF NOT b > 6 THEN b ELSE b+1
b = b
c = IF b > 6 THEN c+1 ELSE 2*b+2
```

wobei die Variablen auf der linken Seite die Ausgabewerte des Programmstücks bezeichnen und die auf der rechten Seite des Gleichheitszeichens die Eingabewerte.

Symbolische Ausführung ist im Prinzip sehr mächtig, sprengt aber sehr schnell die Grenzen dessen, was sich als symbolischer Ausdruck noch handhaben, geschweige denn verstehen läßt. Aus diesem Grund ist es wie gesagt nur für kurze Programmstücke geeignet, kann dort aber gelegentlich sehr nützlich sein.

Gedanklich verwandt mit der symbolischen Ausführung ist der Induktionsbeweis, der sich zur Prüfung von Schleifen oder vor allem Rekursionen eignet. Hier konzentriert sich die symbolische Umformung allerdings auf einen bestimmten Aspekt des Programms, der die nachzuweisende Eigenschaft beschreibt. Die übrigen Teile werden häufig nicht symbolisch, sondern mit anderen Methoden, auch nichtformalen, gehandhabt.

6.7.5 Ausführungstabellen

Ausführungstabellen sind ein Hilfsmittel für die Ausführung eines Programmes „von Hand“. Sie beschreiben sämtliche Schritte der Programmausführung für eine bestimmte Eingabe, samt der Folgen, die jeder Schritt auf die Inhalte aller relevanten Variablen hat. Das Verfahren ist sehr simpel, universell und aufwendig.

Dazu wird eine breite Tabelle angelegt, in deren erste Spalte sukzessive immer der nächste auszuführende Programmschritt eingetragen wird. Im Normalfall liegt dieser als Pseudokode vor, wobei die Granularität eventuell recht grob sein darf, solange die Folgen der Operation auf die betrachteten Variablen klar sind. In die weiteren Spalten werden die Werte aller relevanten Zustandsgrößen eingetragen. Jede solche Größe hat eine eigene Spalte, in die ein Eintrag nur dann gemacht wird, wenn sich in diesem Schritt der Zustand oder Wert ändert. Solche Zustandsgrößen sind Variablen, Umgebungsgrößen (z.B. der Wert, den ein bestimmter Aufruf eines anderen Moduls ergeben würde), Dateiinhalte und -positionen etc. Insbesondere gibt es eine Spalte, in der die Ergebnisse von Bedingungsauswertungen notiert werden. Im Fall von Schleifen treten die entsprechenden Anweisungen mehrfach in der linken Spalte auf.

Um noch einmal das simple Beispiel von oben zu verwenden:

```
Schritt      Bedingung    a    b    c
(START)                        5
a := b + 1;               6
c := 2*a;                           12
IF a > 7 THEN   false
ELSE
  a := b                  5
FI;
```

Eine Ausführungstabelle liefert also nicht nur die Ergebnisse der Programmausführung (abzulesen aus dem jeweils letzten Eintrag jeder Spalte, also hier `a = 5, b = 5, c = 12`), sondern bietet auch Einblicke in die Arbeitsweise der Programmlogik, die sogar dann zur Entdeckung von Fehlern führen können, wenn das Ergebnis richtig ausfällt.

Für fertigen Programmcode könnte man anstatt der Ausführungstabelle auch eine schrittweise Ausführung des Programms in einem Debugger verwenden. Das hat den zusätzlichen Vorteil, Irrtümer bei der Simulation zu vermeiden, hat aber drei erhebliche Nachteile. Erstens ist bei der Handsimulation aufgrund von deren Langsamkeit die Neigung nicht so groß, das Mitdenken einzustellen. Auf diese Weise werden Fehler oft schon sehr viel früher gefunden, was die Langsamkeit kompensieren kann. Zweitens ist die Übersicht über die Variablenzustände und vor allem deren Historie in der Tabellenform oft besser, jedenfalls, solange die Menge der relevanten Variablen klein genug ist. Drittens und vor allem kann man eine Ausführungstabelle auf höheren Abstraktionsniveaus benutzen als den Debugger: nicht erst für fertigen Programmcode, sondern auch für Pseudokode oder noch grobere Methoden der Beschreibung (beispielsweise dort, wo ganze Unterprogramme, die schon benutzt werden, noch gar nicht entworfen sind).

6.7.6 Verfolgungstabellen

Verfolgungstabellen kombinieren mehrere der obigen Techniken miteinander, nämlich Ausführungstabellen, symbolische Ausführung und Induktionsbeweise. Dadurch kann man eine Überprüfung erreichen, die trotz (teilweiser) Abstützung auf die Primitivtechnik der Ausführungstabellen mehr Fälle abdeckt, als mit jenen möglich ist und dennoch im Umfang erträglich bleibt (eventuell sogar kleiner als eine einzige Ausführungstabelle es wäre).

Der Grundgedanke von Verfolgungstabellen besteht darin, ähnlich wie bei Ausführungstabellen konkret die Logik eines Programms, das in Pseudokode spezifiziert ist, abzulaufen und zu prüfen, ob es das gewünschte Verhalten zeigt. Im Gegensatz zu Ausführungstabellen wird jedoch nicht nur mit einem einzelnen Eingabebeispiel gearbeitet, sondern eine komplette Prüfung aller Eingabefälle oder einer erheblichen Untermenge davon angestrebt. Dies geschieht durch die Kombination von Ausführungstabellen mit symbolischer Ausführung und Induktionsbeweisen. Zunächst werden alle Eingabefälle identifiziert, für die die Programmlogik qualitativ unterschiedlich verläuft (ähnlich wie beim Strukturtesten). Dann wird jeder dieser Fälle einzeln untersucht, wobei er eben nicht in einer Form gegeben ist wie „Die Eingabe für `a` lautet 17“, sondern als Bedingung, also in einer Form wie „Die Eingabe für `a` ist größer als 1“, weil beispielsweise die Fälle `a = 0` und `a = 1` getrennt untersucht werden müssen, da dann die Programmlogik anders verläuft.

Entwurfsprüfung nach diesem Verfahren ist relativ aufwendig, bietet aber auch eine gute Abdeckung und kann die meisten Fehler auch subtilerer Art finden. Es ist auch nicht aufwendiger als ein vergleichbar gründlicher Test des Programms, mit dem Unterschied, daß der Test gegebenenfalls nur anzeigt, *daß* ein Fehler vorliegt, aber nicht, worin er besteht. Deshalb kann eine Prüfung mit Verfolgungstabellen erheblich an Arbeit sparen. Da ein beträchtlicher Teil der Arbeit in der Identifikation der zu untersuchenden Fälle besteht und dies auch für den Test gemacht werden muß, ist der effektive Zeitaufwand geringer, als man annehmen könnte.

Ein ausführliches Beispiel findet sich in [Hum95, Seite 400ff].

6.7.7 (Halb)formale Verifikation

Die formale Verifikation, zum Beispiel nach dem wp-Kalkül von Gries, das aus der Informatik-Grundvorlesung bekannt ist, eignet sich nur zur Prüfung von Programmlogik, deren Zweck erstens formal spezifiziert ist und die zweitens selbst auch schon recht detailliert angegeben ist. Unter diesen Voraussetzungen ist sie eine sehr leistungsfähige Methode, was das Auffinden von Fehlern angeht, aber auch eine sehr aufwendige Methode. Wegen der sehr hohen Kosten kann sie nur selten und auch dann nur für kurze Programmstücke eingesetzt werden. Die Kosten und der Aufwand verringern sich erheblich, wenn man die Verifikation gleich bei der Konstruktion des Programmstücks vornimmt. In diesem Fall wird das Programmstück erstens so konstruiert, daß die Verifikation nicht so kompliziert wird und zweitens wahrscheinlich von vornherein mit nur sehr wenigen Fehlern, die zudem sehr früh erkannt werden. Über das Vorgehen zur Konstruktion von Programmen samt Korrektheitsbeweis gibt es eigene Vorlesungen, so daß wir hier nicht weiter darauf eingehen.

Eine zweite Methode zur Absenkung des Aufwands besteht darin, nur die kritischsten Aspekte des Programmstücks zu verifizieren, nämlich die Schleifen. Hierbei wird dann nur die Schleifeninvariante aufgestellt und geprüft, ob die Schleife diese auch einhält, wenn man annimmt, daß der Schleifenrumpf die gewünschten Operationen durchführt. Es wird also nur die Schleifenbedingung getestet, so daß der korrekte Abbruch der Schleife geprüft wird, sowie die Operationen, die von einem Schleifendurchlauf zum nächsten weiterschalten sollen und mit der Abbruchbedingung richtig zusammenspielen müssen.

Drittens kann man die Verifikation vereinfachen, indem man von der kompletten formalen Verifikation zu einer halbformalen übergeht, bei der zwar die gleichen strengen Schlußweisen verwendet werden, um die Korrektheit zu untersuchen, aber mit weniger streng gefaßtem Inhalt. Die grobe Architektur der Argumentation soll also einem formalen Beweis entsprechen, aber bei den Einzelteilen darf über Details hinweggegangen werden. Eine solche Einschränkung ist ohnehin unvermeidlich, wenn gar keine komplette formale Spezifikation vorliegt, sondern Teile der Spezifikation mit sprachlichen Begriffen operieren. Im Abschnitt über die Cleanroom-Methode (7) werden wir ein Beispiel einer solchen Vorgehensweise sehen.

6.8 Weitergehende Information

Die zentrale Informationsquelle über PSP ist das Buch von Humphrey [Hum95]. Darin finden sich genaue Anleitungen zu den einzelnen Techniken, die in die Form eines Kurses wie oben vorgestellt eingebettet sind. Das Buch ist sehr sorgfältig aufbereitet und eignet sich sowohl als Lehrbuch als auch als Nachschlagewerk über den PSP. Ein guter Übersichtsartikel ist [Hum96]; hierin finden sich auch empirische Daten aus 8 PSP Kursen.

Humphrey und andere am Software Engineering Institute (SEI, http://www.sei.cmu.edu) sammeln auch Daten über die Ergebnisse von PSP-Kursen. Diese Daten sollen später einmal öffentlich verfügbar werden, sind aber bislang noch nicht allgemein zugänglich.

An der Fakultät Informatik der Universität Karlsruhe gibt es ein Praktikum zum PSP (Softwaretechnik-Praktikum), üblicherweise jeweils im Wintersemester.

Abschnitt 7

Die Cleanroom-Methode

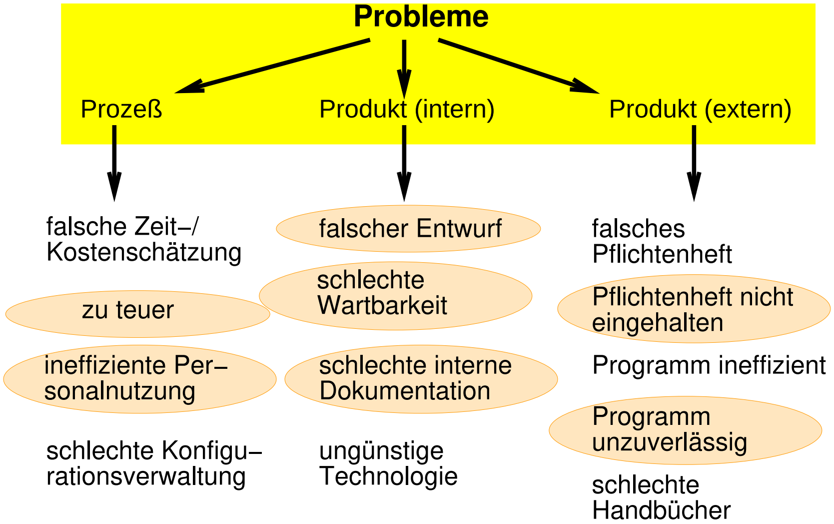

Abbildung 7.1: Von der Cleanroom-Methode angesprochene Problembereiche.

7.1 Einordnung und Ziele

Die Cleanroom-Technik wurde ab 1980 von Harlan D. Mills bei IBM entwickelt. Ziel der Technik ist die Herstellung von Software mit sehr hoher und zudem quantitativ bekannter Zuverlässigkeit zu günstigen Kosten. Der Name lehnt sich an die Reinräume bei der Chipherstellung an und soll deutlich machen, daß der Schwerpunkt der Technik bei der Fehlervermeidung (im Gegensatz zur Fehlerbeseitigung) liegt.

In Cleanroom werden sowohl bestimmte konkrete Techniken verwendet, die fast den ganzen Software-Lebenszyklus abdecken, als auch eine Reihe von Maßnahmen, die dem Bereich der Software-Prozeßverbesserung zuzuordnen sind.

Die **Grundidee** von Cleanroom besteht im Einsatz halbformaler und formaler Methoden bei der Systemherstellung und dem Ersetzen von Tests zur Fehlerbeseitigung durch Tests zur Zuverlässigkeitsmessung. Qualitätserzeugung und Qualitätsprüfung werden im Gegensatz zur üblichen Softwareentwicklung auch personell streng getrennt.

Cleanroom eignet sich für den Einsatz mit allen Programmiersprachen und harmoniert mit objekt-orientierten Methoden und CASE. Es zielt hauptsächlich auf die Neuentwicklung von Software, kann aber auch in der Wartung oder im Reengineering eingesetzt werden.

7.2 Ansätze

Die Cleanroom-Methode setzt sich aus einer ganzen Anzahl von Maßnahmen zusammen, die ineinandergreifen.

7.2.1 Kleine Teams aus Gleichen

Große Systeme werden in Teile zerlegt, die von kleinen Teams aus etwa sechs bis acht Personen hergestellt werden. Das erzwingt zum einen eine sorgfältige Spezifikation und zieht zum anderen zwei weitere Vorteile nach sich: Die Gesamtentwicklungszeit kann reduziert werden, da die einzelnen Teams unabhängig voneinander arbeiten können, und die Teams sind effizient und motiviert, weil weder eine Isolation einzelner Entwickler erfolgt, noch starke Reibungsverluste wie in großen Teams auftreten. Die Mitglieder solcher Teams begutachten ihre Arbeit gegenseitig, um ständig ein hohes Qualitätsniveau zu sichern. Ein wichtiger Aspekt von Cleanroom ist, daß die Teams sehr entschieden darauf eingeschworen werden, höchste Qualität abzuliefern. Die geringe Teamgröße erleichtert es, diesen Vorsatz auch umzusetzen, weil sich einzelne Mitglieder bei Mängeln nicht so sehr hinter einer großen Zahl anderer möglicher Schuldiger verstecken können.

7.2.2 Separation von Entwicklung und Test

Es gibt zwei völlig getrennte Arten von solchen Teams: Entwicklungsteams und Testteams.

Entwicklungsteams sind ausschließlich für die Herstellung der Software zuständig. Sie sollen von vornherein ein Produkt höchster Qualität abliefern. Zur Herstellung und Sicherung dieser Qualität verwenden sie ausschließlich Betrachtungen des Programms selbst, sowohl in seiner Quellcodeform als auch vor allem auf der Ebene seiner Entwurfsrepräsentationen. Ein Entwicklungsteam führt die Software niemals aus.

Testteams hingegen sind ausschließlich für die Feststellung der Zuverlässigkeit der Software zuständig. Dazu bedienen sie sich der Methode des statistischen Testens. Ein Testteam führt niemals eine Änderung an der Software aus und betrachtet niemals die Software von einem anderen Standpunkt als dem der Ausführung.

7.2.3 Exakte Spezifikation

Die Spezifikation von Software wird möglichst mit mathematischen Sprachmitteln beschrieben, d.h. mit solchen Sprachmitteln, die eine genau definierte Semantik haben. Dem liegt die Annahme zugrunde, daß erstens ein großer Teil von Softwarefehlern daher stammt, daß die Aufgabenstellung nicht klar genug beschrieben war und daß zweitens überhaupt nur mit einer exakten Definition der Aufgabe auch exakt definiert werden kann, was eigentlich ein Softwarefehler ist. Im Kontrast hierzu kursiert im Softwarebereich stellenweise immer noch der fliegende Spruch *"It's not a bug, it's a feature!"*. Eine exakte Spezifikation ist natürlich auch Voraussetzung dafür, mathematische Methoden bei der Korrektheitsprüfung einsetzen zu können.

Eine exakte Spezifikation wird nicht nur am Anfang der Entwicklung erzeugt, sondern ist auch das **Produkt jeder Zwischenstufe beim Entwurf**. Konkret werden dafür z.B. Spezifikationssprachen wie Z oder VDM eingesetzt oder auch die Kastenspezifikation (box structuring method). Auch problemspezifische Methoden (Spezialgrammatiken) kommen in Frage, welche sich weitgehend einer natürlichsprachlichen Spezifikation annähern können; jedoch ist immer darauf zu achten, daß alle Teile der Spezifikation wohldefiniert sind, also eine und nur eine Bedeutung haben.

Offensichtlich verlangt diese Herangehensweise, daß die Anforderungen an die Software bei Beginn der Entwicklung feststehen. Da dies in realen Situationen nur selten der Fall ist, scheint Cleanroom kaum je wirklich anwendbar zu sein. Tatsächlich bedeutet „feststehen“ aber nicht, daß nichts mehr an den Anforderungen geändert werden kann, sobald die Entwicklung begonnen hat, sondern nur, daß überhaupt erst einmal **Anforderungen festgeschrieben** werden müssen, damit eine Basis für eine zielgerichtete Entwicklung vorliegt. Sollten Änderungen in den Anforderungen nötig werden, müssen diese zu einem geeigneten Zeitpunkt eingebracht werden und werden dann quasi als eigener Entwicklungsauftrag gehandhabt.

7.2.4 Schrittweise Verfeinerung

Der Entwurf erfolgt nach der Methode der schrittweisen Verfeinerung. Dabei wird eine Spezifikation in mehreren Schritten in Richtung einer Implementation konkretisiert. Eingabe jedes Schrittes ist eine Spezifikation, Ausgabe ist eine Zerlegung der spezifizierten Funktion in mehrere Teile, deren Zusammenwirken beschrieben wird und die jeweils wiederum spezifiziert werden und dann entweder Eingabe eines weiteren Verfeinerungsschrittes sind oder als elementar betrachtet werden können.

7.2.5 Begutachtung und Verifikation

Jeder Verfeinerungsschritt wird einer Prüfung unterzogen, die in Cleanroom *Verifikation* genannt wird. Da auf jeder Verfeinerungsstufe alle Teile entweder schon realisiert oder genau spezifiziert sind, kann man für jede solche Verfeinerung einen Korrektheitsbeweis konstruieren, der zumindest annähernd der mathematischen Bedeutung dieses Wortes genügt. Ein solcher Beweis wird in Cleanroom vom Entwerfer einer Verfeinerungsstufe konstruiert und dann in einer Teaminspektion begutachtet, wobei die einzusetzenden Verfahren der Form der Spezifikation und dem jeweiligen Anwendungsfall angepaßt werden, aber in jedem Fall in mathematischer Logik gründen. Das Inspektionsteam versucht sich unter der Leitung der Erfinder der Verfeinerung von der Korrektheit der Verfeinerung zu überzeugen. Echte mathematische Verifikationen mit mathematischer Notation werden nur in einem sehr kleinen Teil der Fälle durchgeführt, zumeist für die kritischen Transaktionskerne einer Anwendung.

7.2.6 Inkrementelle Entwicklung

Damit die Entwicklung beherrschbar bleibt, werden stets nur Komponenten überschaubarer Größe entwickelt. Eine Spezifikation wird deshalb in Teile von höchstens etwa 10000 Zeilen Code zerlegt. Diese Teile werden Inkremente genannt. Ein Inkrement kann entweder auf einem früheren Inkrement aufbauen (inkrementelle Entwicklung) oder von anderen Inkrementen unabhängig sein (separate Entwicklung). In beiden Fällen müssen Inkremente jedoch so definiert werden, daß nach Abschluß der Entwicklung eines Inkrements dieses ausführbar ist und ohne spezielle Testtreiber getestet werden kann.

7.2.7 Statistisches Testen

Bei herkömmlicher Softwareentwicklung sind als Testmethoden vor allem das Funktionstesten (black box test) und das Strukturtesten (white box test) üblich.

Funktionstesten orientiert sich für die Definition der Testfälle nur an der Spezifikation der Software und strebt üblicherweise eine gute Abdeckung der diversen Äquivalenzklassen von Eingaben an, die unterschiedliche Klassen von Ausgaben erzeugen.

Das Strukturtesten legt demgegenüber die konkrete Implementation der Software bei der Auswahl der Testfälle zugrunde und versucht, eine möglichst gute Abdeckung gewisser Ausführungspfade des Programms zu erreichen. Ein Beispiel für eine Technik dieser Art ist der strukturorientierte Datenflußtest, der in der Vorlesung Softwaretechnik besprochen wird.

In Cleanroom wird ausschließlich das Funktionstesten verwendet. Dies hat den Vorteil, daß Testfälle bereits während der Programmentwicklung definiert werden können. Die Testfälle werden dabei nicht wie üblich von Hand definiert, sondern als Zufallsstichprobe aus einer Grundgesamtheit ausgewählt, die das erwartete tatsächliche Benutzungsprofil der Software widerspiegelt.

Wie erhält man diese Grundgesamtheit? Dazu ist ein Prozeß nötig, der in Cleanroom die Benutzungsmodellierung genannt wird: Es wird ein stochastisches mathematisches Modell aufgestellt, das den Prozeß modelliert, der die Programmeingaben erzeugt. Dieses Modell wird dann ausgeführt, um die Testfälle zu erhalten. Das Modell oder eine etwas weniger genaue Beschreibung davon ist Bestandteil der Spezifikation, die am Anfang der Entwicklung steht.

Zweck des Testens in Cleanroom ist nicht die Suche von Fehlern, sondern die Beurteilung der Zuverlässigkeit der Software, die bereits zu Beginn des Testens als weitgehend fehlerfrei angenommen wird.

7.2.8 Zusammenfassung

Abbildung 7.2 zeigt nochmals grob im Überblick die Abfolge der Arbeitsschritte und den Datenfluß zwischen ihnen für eine Cleanroom-Softwareentwicklung: Nach der Erstellung der Gesamtspezifikation wird jeweils ein Inkrement definiert und bearbeitet (im allgemeinen werden natürlich alle Inkremente zu Beginn vorgeplant). Innerhalb jedes Inkrements wird zunächst parallel von der Entwicklungsgruppe das Programm sowie von der Testgruppe das Benutzungsmodell und daraus die Menge der Testfälle erstellt. Dabei erfolgt die Programmentwicklung nach der Methode der schrittweisen Verfeinerung mit Begutachtung und Verifikation jedes Verfeinerungsschritts. Nach Fertigstellung des Programms wird vom Testteam der statistische Test durchgeführt, der zu einer quantitativen Aussage über die Zuverlässigkeit führt.

7.2.9 Bemerkung: Austauschbarkeit der Techniken

Es sei hier noch angemerkt, daß die obenerwähnten Techniken und Ansätze nicht unabdingbar oder unersetzlich für die Cleanroom-Methode sind. Cleanroom verlangt nur, daß Fehlervermeidung im Vordergrund steht und abschließend eine gewisse Zuverlässigkeit zugesichert werden kann. Jeder der obengenannten Ansätze kann im Prinzip durch andere Techniken ergänzt oder ersetzt werden. Die geschilderten Ansätze sind lediglich die wichtigsten derzeit für günstig befundenen Methoden, die zum angestrebten Ziel führen. Insbesondere können alle Anforderungen aufgeweicht werden und werden das auch häufig mehr oder weniger stark: Die Spezifikation ist nur teilweise mathematisch, ein Teil bleibt natürlichsprachlich; die Verifikationsschritte werden

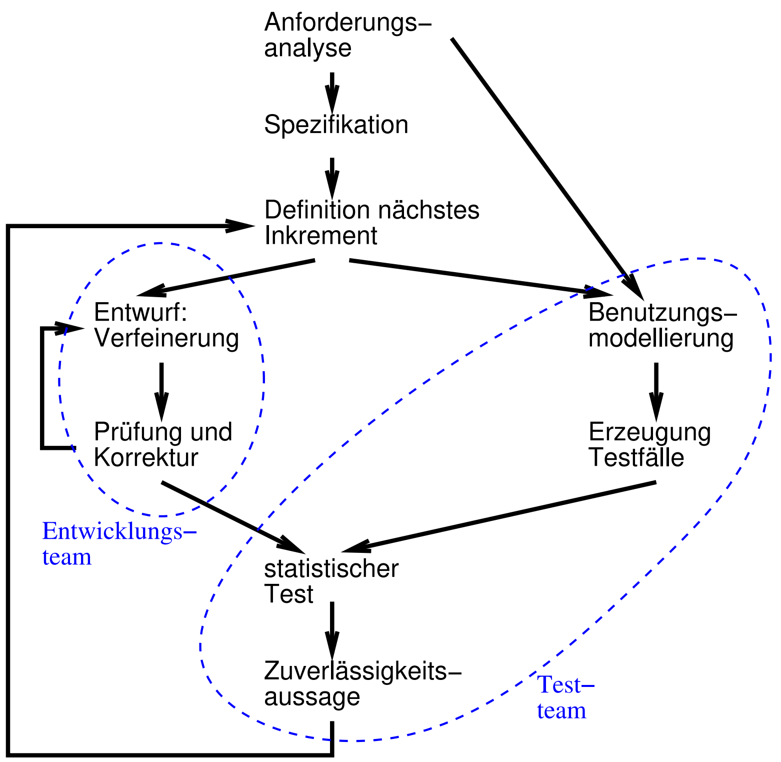

Abbildung 7.2: Überblick über Datenfluß und Prozeßschritte der Cleanroom-Entwicklung

zum Teil mit nichtmathematischen Schlußweisen durchgeführt; Fehler, die im statistischen Testen noch auftreten, werden eventuell mit Debugging-Methoden gesucht und dann entfernt. Eine klare Abgrenzung, wann man einen Entwicklungsprozeß sinnvollerweise als Cleanroom-Prozeß bezeichnen kann, gibt es nicht.

7.3 Ergebnisse

Der hohe Aufwand zur Prüfung der einzelnen Verfeinerungsschritte zahlt sich tatsächlich insofern aus, daß die Restfehlerrate in ausgelieferten Produkten bei Cleanroom typischerweise erheblich niedriger liegt als mit anderen Vorgehensmodellen. Überraschend mag man hingegen die Tatsache empfinden, daß zugleich die Produktivität der Entwicklung anwächst, obwohl man sich doch eines mächtigen Werkzeugs nicht mehr bedient, nämlich des Computers zur Ausführung der Software zwecks Test.

Mittlerweile wird Cleanroom von einer ganzen Reihe von Organisationen eingesetzt und es gibt eine beträchtliche Zahl von Erfolgsmeldungen. Hier seien einige wenige davon willkürlich herausgegriffen:

7.3.1 Ericsson OS-32

OS-32 war ein Projekt mit 70 Mitarbeitern über 18 Monate und erzeugte ein Betriebssystem für Telefonvermittlungen in einem Umfang von 350.000 Zeilen. Mit Cleanroom wurde 70% Verbesserung der Produktivität in der Entwicklung und 100% Verbesserung der Produktivität beim

Test gegenüber bisherigen Erfahrungen erzielt. Die Fehlerrate im Test betrug 1.0 Fehler pro tausend Zeilen Code (KLOC).

7.3.2 IBM Cobol SF

Die „Cobol Structuring Facility“ ist ein Programm zum automatischen Umwandeln von unstrukturiertem Cobol-Code (mit Goto) in strukturierten Cobol-Code. Es hat 85.000 Zeilen, eine Fehlerrate von 3.4 pro KLOC und ergab eine fünffach höhere Produktivität als üblich. Nach der Auslieferung traten noch sieben, allesamt geringfügige Fehler auf.

7.3.3 Ein kontrolliertes Experiment

In einem Experiment [SBB87] an der Universität von Maryland wurde 1982/1983 die gleiche Software insgesamt 15 mal von 15 dreiköpfigen Teams entwickelt. 10 Teams benutzten Cleanroom, 5 Teams herkömmliche Methoden (insbesondere Testen zur Fehlerfindung). Die Teams bestanden aus professionellen Softwareentwicklern mit im Mittel eineinhalb Jahren Berufserfahrung. Die Softwareprodukte hatten zwischen 800 und 2300 Zeilen Code, die Entwicklungszeit betrug sechs Wochen mit vier Meilensteinen (Grobentwurf, Feinentwurf, Code, Dokumentation). Hier die Ergebnisse:

1. Die Cleanroom-Teams erfüllten im Mittel signifikant mehr Programmanforderungen als die anderen. Alle 10 Cleanroom-Teams hielten alle vier Meilensteine ein, während nur 2 der 5 anderen dies schafften. Cleanroom hatte also eine höhere Produktivität.

2. Der Quellcode der Cleanroom-Teams hatte mehr Kommentare und meßbar geringere Kontrollflußkomplexität, ohne deshalb mehr Programmzeilen aufzuweisen. Er ist also besser verständlich und damit leichter wartbar.

3. Vollständigkeit der Implementation und Prozentsatz erfolgreich absolvierter Testfälle bei den Cleanroom-Programmen korrelierten positiv mit dem Anteil von Prozeduraufrufen an den Anweisungen sowie mit dem Anteil an IF-Anweisungen; sie korrelierten negativ mit dem Anteil an WHILE-Anweisungen, dem Anteil an CASE-Anweisungen und der Häufigkeit, mit der eine Variable im Mittel im Programmtext auftritt (Variablen-Wiederverwendung). Das heißt, daß erfolgreiche Anwendung von Cleanroom positiv mit einer einfacheren Anwendung der Programmiersprache korreliert.

4. Die Cleanroom-Programme absolvierten signifikant mehr Testfälle erfolgreich als die anderen.

5. 86 Prozent der Cleanroom-Entwickler sagten, sie vermißten die Befriedigung des Programmtestens. Dies hatte aber keine ersichtliche Auswirkung auf die Qualität ihrer Produkte.

7.4 Probleme und Hindernisse

7.4.1 Anwendungsbereich

Die Cleanroom-Methode ist gut geeignet für viele Fälle von besonders qualitätskritischen Systemen, vor allem hochkomplexe wie Betriebssysteme, Mikrocode, Server, Echtzeitsysteme, etc.

Kennzeichen solcher Systeme sind ein komplexer und wenig anschaulicher Zustandsraum, relativ klar definierte Ein-/Ausgabeschnittstellen und oftmals Interaktionen mehrerer nebenläufiger Aktivitäten.

Für andere Arten von Software ist Cleanroom hingegen kaum geeignet und zwar immer dann, wenn es schwierig ist, eine formale Spezifikation anzugeben. Dies gilt insbesondere für viele Arten von interaktiven Programmen. Man stelle sich beispielsweise den Versuch vor, für ein Programm wie MS-Word eine formale Spezifikation anzugeben.

Eine weitere Schwierigkeit ergibt sich im Zusammenhang mit dem statistischen Testen. Wenn der Anwendungsbereich nämlich bewirkt, daß Testdaten „teuer“ sind, kann ein statistisches Testen nicht zu vernünftigen Kosten durchgeführt werden. Das ist immer dann der Fall, wenn es schwierig ist, zu gegebenen Eingabedaten festzustellen, ob die vom Programm erzeugten Ausgaben korrekt sind oder nicht.

7.4.2 Randbedingungen

Ferner stellt Cleanroom hohe Anforderungen bezüglich gewisser Randbedingungen der Systementwicklung. So ist erstens ein hoher Ausbildungsstand der Softwareingenieure nötig, denn eine ausreichende Beherrschung formaler Beschreibungs- und Schließmethoden ist oftmals nicht gegeben und vielen heutigen Softwarepraktikern wohl auch kaum mehr vermittelbar.

Zweitens muß die Softwareorganisation einen definierten Softwareprozeß verwenden (also mindestens auf CMM-Stufe 3 sein, siehe das Kapitel über den Softwareprozeß). Andernfalls kann die Einhaltung und Überwachung der nötigen Verifikationsdisziplin nicht gewährleistet werden.

7.5 Spezifikation, Entwurf und Verifikation mit Kastenstrukturen

Eine Methode, die für Cleanroom typisch ist und die formale Spezifikation mit der schrittweisen Verfeinerung verbindet, ist die Spezifikation und der Entwurf mit Kastenstrukturen (box structures). Dabei gibt es drei Sichten auf eine Softwarekomponente: Die reine Spezifikation (black box view), die Zustandsbeschreibung (state box view) und die Prozeßbeschreibung (clear box view). Die Bezeichnung Kastenstruktur weist darauf hin, daß jede dieser Beschreibungen in sich abgeschlossen ist.

7.5.1 black box, state box, clear box

Die **black box view** beschreibt die Reaktionen der Komponente ausschließlich über Eingabehistorien. Sie ist völlig implementationsunabhängig und betrachtet die Softwarekomponente als mathematische Funktion.

Die **state box view** stellt demgegenüber zumindest einen Teil der Eingabehistorien durch innere Zustände der Komponente dar und modelliert die Ausgaben als Reaktion auf momentane Eingaben und Zustände. Dies ist der erste Schritt auf dem Weg zu einer Implementation.

Die **clear box view** komplettiert einen Verfeinerungsschritt im Entwurf, indem die Ausgaben beschrieben werden über gegenwärtige Eingaben, Zustände und die Benutzung von black box views der Teilkomponenten niedrigerer Verfeinerungsstufen.

Für verschiedene Aspekte des Softwareentwurfs haben die datenorientierte Zustandsbeschreibung oder die vorgangsorientierte Prozeßbeschreibung Vorteile gegenüber der jeweils anderen. Es ist deshalb günstig, daß in dieser Vorgehensweise beide Methoden vereint werden.

Ein Verfeinerungsschritt geht also in folgenden Teilschritten vor sich:

- **Definiere die black box:**
  (1) Definiere Eingaben,
  (2) Definiere Ausgaben anhand der Eingabehistorien.
- **Definiere die state box:**
  (3) Definiere Zustände zur Repräsentation der Eingabehistorien,
  (4) Wähle Zustandsdaten aus, die auf dieser Ebene verwaltet werden sollen,
  (5) Formuliere black box neu, so daß sie die Ausgaben anhand von Eingaben und Zustandsdaten dieser Ebene beschreibt,
  (6) Verifiziere die state box.
- **Definiere die clear box:**
  (7) Definiere Datenabstraktion für die Zustandsdaten,
  (8) Formuliere state box neu, so daß sie die Ausgaben anhand von Eingaben, Zustandsdaten dieser Ebene und Aufrufen von black boxes noch zu definierender niederer Ebenen beschreibt,
  (9) Verifiziere die clear box.

Weitere Verfeinerungsschritte beginnen dann mit den black boxes, die in der clear box eingeführt wurden, es sei denn, diese werden als elementar betrachtet und nicht weiter verfeinert.

7.5.2 Beispiel

Das folgende Beispiel soll einen groben Eindruck davon vermitteln, wie diese fortgesetzte Verfeinerung vor sich geht und wie sie mit der Verifikation zusammenspielt. Das Beispiel ist natürlich ein reines Spielzeugbeispiel. In einer wirklichen Anwendung von Cleanroom wären die einzelnen Verfeinerungschritte umfangreicher und der Anwendungsfall auch komplizierter. Das Beispiel verzichtet auch völlig auf eine state box, weil es sich nur um ein Prädikat handelt, zu dessen Realisierung kein Zustand nötig ist.

•*black box 1:* `dreieckstyp` (a, b, c)
Vorbedingung: a, b, c sind positive reelle Zahlen
Nachbedingung: Liefert GLEICHSEITIG bzw. GLEICHSCHENKLIG bzw. SONSTIGES bzw. KEINDREIECK, wenn das Tripel (a, b, c) Seitenlängen eines gleichseitigen bzw. nichtgleichseitigen gleichschenkligen bzw. eines nichtgleichschenkligen Dreiecks bezeichnet bzw. überhaupt nicht ein Tripel von Seitenlängen eines Dreiecks sein kann.

∘*clear box 1:* `dreieckstyp` (a, b, c)
IF `alle seiten erfüllen dreiecksungleichung` (a, b, c)
THEN Liefere `echter dreieckstyp` (a, b, c)
ELSE Liefere KEINDREIECK

•*black box 2:* `alle seiten erfüllen dreiecksungleichung` (a, b, c)
Vorbedingung: a, b, c sind positive reelle Zahlen.

Nachbedingung: True, falls jede Seite kürzer als die Summe der beiden anderen; False andernfalls.

•*black box 3:* `echter dreieckstyp` (a, b, c)
Vorbedingung: (a, b, c) sind die Seitenlängen eines echten Dreiecks
Nachbedingung: Liefert GLEICHSEITIG bzw. GLEICHSCHENKLIG bzw. SONSTIGES, wenn das Tripel (a, b, c) die Seitenlängen eines gleichseitigen bzw. nichtgleichseitigen gleichschenkligen bzw. eines nichtgleichschenkligen Dreiecks bezeichnet.

⋆*Verifikation clear box 1*:
Die drei Seiten können genau dann ein Dreieck bilden, wenn die beiden kürzeren, sagen wir x und y, zusammen länger sind als die längste, z. Dann nämlich können sie einen Winkel bilden, dessen Enden von einem Ende von z zum anderen reicht. $z < x + y$ (also: „Dreiecksungleichung gilt für die Seite z") ist also hinreichend, damit ein Dreieck vorliegt. Außerdem ist die Geltung der Dreiecksungleichung notwendig für jede Seite, damit ein Dreieck vorliegt. $z < x + y$ gilt insbesondere, falls die Dreiecksungleichung für jede Seite gilt.
Damit liefert `alle seiten erfüllen dreiecksungleichung` die Aussage, daß es sich wirklich um ein Dreieck handelt, die Vorbedingung von `echter dreieckstyp` wird erfüllt, und clear box 1 ist korrekt.

Ende von Verfeinerungsschritt 1.

∘*clear box 2:* `alle seiten erfüllen dreiecksungleichung` (a, b, c)
Liefere ($a < b + c$ AND $b < a + c$ AND $c < a + b$)

⋆*Verifikation clear box 2*:
3 verschiedene, korrekte Fälle geprüft, UND-verknüpft; ist offensichtlich korrekt.

Ende von Verfeinerungsschritt 2.

∘*clear box 3:* `echter dreieckstyp` (a, b, c)
IF $a = b = c$ THEN Liefere GLEICHSEITIG ELSE IF $a = b$ OR $a = c$ OR $b = c$ THEN Liefere GLEICHSCHENKLIG ELSE Liefere SONSTIGES

⋆*Verifikation clear box 3*:
SONSTIGES ist als NOT gleichschenklig korrekt geprüft laut black box 1.
gleichseitig ist ein Spezialfall von gleichschenklig und muß deshalb zuerst geprüft werden, dies geschieht hier.
Deshalb ist clear box 3 korrekt.

Ende von Verfeinerungsschritt 3.

7.6 Statistisches Testen und Zertifizierung

7.6.1 Benutzungsmodellierung

Zur Erzeugung der Testfälle für das statistische Testen muß zunächst ein Benutzungsmodell aufgestellt werden. Basis hierfür ist das sogenannte Benutzungsprofil, eine (meist natürlich-sprachliche) Beschreibung der erwarteten Benutzung des Programms, die in den Anforderungen niedergelegt ist.

Zur mathematischen Beschreibung des Benutzungsmodells werden in der Regel **Markov-Ketten** verwendet und zwar solche mit endlichem Zustandsraum und diskreten Ereignissen. Eine solche Markov-Kette ist nichts anderes als ein nichtdeterministischer endlicher Automat mit definierten Übergangswahrscheinlichkeiten.

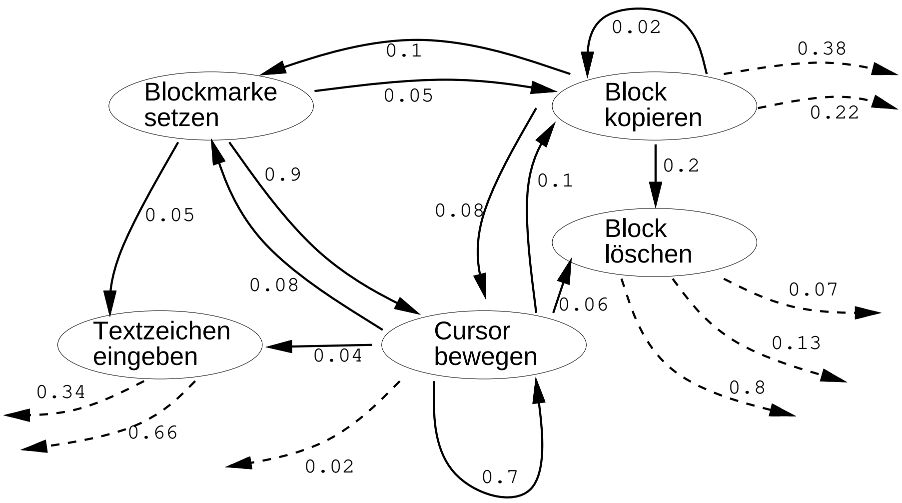

Abbildung 7.3: Ausschnitt aus einem Benutzungsmodell für einen Texteditor

In Abbildung 7.3 sehen wir als Beispiel einen kleinen Ausschnitt aus dem Benutzungsmodell für einen Texteditor. Die ovalen Kästen stehen für die Zustände, deren Erreichen zugleich stets eine Eingabeaktion des Benutzers repräsentiert. Diese Aktion ist hier verbal dargestellt. Die Pfeile zwischen den Zuständen geben die möglichen (d.h. modellierten!) Übergänge zwischen den Zuständen an, das heißt, die möglichen Abfolgen von Benutzeraktionen. Jeder Übergang ist mit seiner Wahrscheinlichkeit beschriftet. Die gestrichelten Übergänge führen zu Zuständen außerhalb des Modellausschnitts.

Die abgebildeten Übergänge beschreiben die typischen Folgen von Eingaben, wenn ein Textblock markiert wurde: Meist wird dann erst der Cursor bewegt und dann eine Blockoperation ausgelöst; andere Eingaben kommen jedoch ebenfalls vor. Das Beispiel macht auch klar, welche Abwägungen zu treffen sind: Beispielsweise steht „Cursorbewegung" natürlich nicht nur für eine einzige mögliche Eingabe, sondern dahinter verbirgt sich eine ganze Klasse von Eingaben. Der entsprechende Zustand könnte im Benutzungsmodell also je nach Bedarf entweder in eine ganze Reihe von Zuständen verfeinert werden, was das Modell enorm vergrößert, oder durch ein ad-hoc gebildetes Untermodell realisiert werden, das nicht mit einer Markov-Kette beschrieben wird, z.B. indem zufällig irgendeine der möglichen Eingaben zur Cursorbewegung ausgewählt wird.

Für realistische Programme werden die benötigten Markov-Ketten enorm groß, wenn die Modellierung auch nur einigermaßen genau sein soll. Deshalb nimmt man meistens Zuflucht zu **hierarchischen Modellen**, in denen zunächst nach Benutzertypen unterschieden wird, für jeden Benutzertyp nach Benutzungsart (z.B. Teilfunktion) und erst für jede Teilfunktion eine Markov-Kette aufgestellt wird. Zwischen Einzelpunkten in verschiedenen dieser Ketten müssen dann Verbindungen eingeführt werden, die bedeuten, daß die Ausführung eines bestimmten Übergangs in einer Kette einen bestimmten anderen Übergang in einer anderen Kette bewirkt.

7.6.2 Testen

Aus dem Benutzungsmodell kann nun eine beliebige Anzahl von **Testeingaben** automatisch erzeugt werden. Dazu wird einfach das Benutzungsmodell an seinem Eingang betreten und mit Hilfe eines Zufallsgenerators entsprechend der Übergangswahrscheinlichkeiten solange durchlaufen, bis ein Ausgang (Endzustand) erreicht wird. Die in jedem Zustand erzeugten Eingabeteile werden aufgesammelt.

Dabei ergibt sich meistens das Problem, daß **Testausgaben** zur Kontrolle der Richtigkeit des Programms nicht automatisch mit erzeugt werden können. Diese Situation ist jedoch nicht grundsätzlich anders als beim normalen Testen. Die Ausgaben müssen entweder von Hand oder mit Hilfe anderer Programme ermittelt werden, oder man kann auf sie verzichten, weil Konsistenz- und Plausibilitätsprüfungen benutzt werden, um die Richtigkeit der Programmreaktionen zu beurteilen.

Beim Testen wird jeder Testfall auf Korrektheit geprüft und die Abstände zwischen den Programmversagen werden registriert.

7.6.3 Zertifizierung

Das Testen wird solange fortgeführt, bis eine **angestrebte Zuverlässigkeitsaussage** erreicht ist. Eine solche Zuverlässigkeitsaussage hat zum Beispiel die Form „Die MTTF des Programms beträgt mindestens m Testfälle mit einer Konfidenz von k Prozent“. MTTF bedeutet *Mean Time To Failure*, also mittlere Zeit bis zum nächsten Versagen, also: Erfolgsfälle pro Versagensfall.

Eine solche Aussage kann mit Methoden der Statistik geprüft werden. Ein vereinfachtes graphisches Schema, das dieses Vorgehen verdeutlicht, ist in Abbildung 7.4 zu sehen: Für jeden erfolgreichen Testfall geht die Linie um eine Einheit nach rechts, für jeden Versagensfall um eine Einheit nach oben.

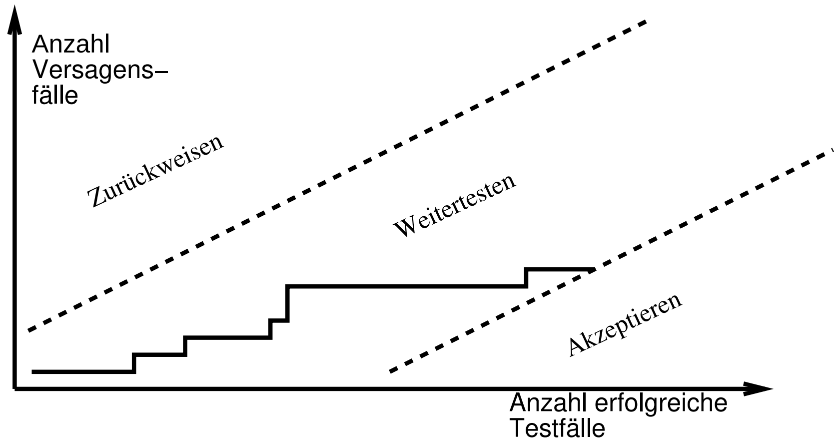

Abbildung 7.4: Schema zur Prüfung eines Programms auf gegebene Zuverlässigkeit mit bestimmter Konfidenz

Wir sehen einen Korridor, der die „Weitertesten“-Region enthält, in der der Test noch fortgeführt werden muß. Die Steigung des Korridors (Versagensfälle pro Erfolgsfall) gibt den Kehrwert der Zuverlässigkeit gemessen in MTTF an. Die Breite des Korridors beschreibt qualitativ gesehen die gewünschte Konfidenz. Wird der „Weitertesten“-Korridor nach oben verlassen, so wurde mit der gegebenen Konfidenz das Zuverlässigkeitsziel nicht erreicht, wird er nach unten verlassen, so wurde es mit der gleichen Konfidenz erreicht. Bei Betreten der „Akzeptieren“-Region wird also das Programm erfolgreich zertifiziert, bei Betreten der „Zurückweisen“-Region wird es als zu unzuverlässig abgelehnt. Es muß dann korrigiert oder neu entworfen werden und wird in seiner neuen Form dann erneut getestet.

In einer realen Anwendung können noch leistungsfähigere mathematische und statistische Methoden eingesetzt werden, um die Zuverlässigkeit und deren Konfidenz zu bestimmen. Insbesondere ist es möglich, die Ergebnisse der Testfälle von vor der Korrektur weiter in die Zuverlässigkeitsberechnung einzubeziehen. Dies erfolgt über ein Maß der Verbesserung der Zuverlässigkeit nach einer Programmkorrektur und erlaubt damit, die Genauigkeit der Aussage zu verbessern, ohne zusätzliche Testfälle durchführen zu müssen. Außerdem können verschiedene **Fehler- und Versagensmodelle** verwendet werden, um bessere Schranken für die Konfidenz zu erhalten.

7.7 Weitergehende Information

Die umfangreichste Quelle von Information in Papierform über Cleanroom ist das Buch [Dye92]. Darin werden auch die einzelnen Techniken einigermaßen ausführlich beschrieben.

Eine kurze Übersicht über den Ansatz und die Methoden von Cleanroom findet sich in [CM90]. Dieser Artikel eignet sich als allgemeine Referenz und zum Weiterempfehlen für jemand, der sich mal schnell einlesen will.

Ein Beispiel für eine erfolgreiche Anwendung von Cleanroom ist beschrieben in [TBS92]. In diesem Artikel wird auch genau auf die einzelnen Fehler eingegangen, die im geschilderten Projekt noch auftraten: Fehlerart, Fehlerursache, wann und wie gefunden etc. Ein anderer Artikel, der vor allem praktische Erfahrungen betont, ist [SKA96].

[WR94] diskutiert den Ansatz der Benutzungsmodellierung und des statistischen Testens mit Zertifizierung im Hinblick auf eine garantierte Qualität von wiederverwendbaren SW-Komponenten.

[SBB87] beschreibt die oben erwähnte empirische Laborstudie, bei der dasselbe Programm 10 mal mit Cleanroom und 5 mal „normal“ entwickelt wurde.

Im WWW gibt es Informationen zu Cleanroom zum Beispiel bei

- Software Quality Research Lab, University of Tennessee, Knoxville. Hier gibt es zum Beispiel ein Cleanroom Tutorial. (http://www.cs.utk.edu/˜trammell/CRT/cleanroom.html)
- Universität Kaiserslautern (http://wwwagse.informatik.uni-kl.de/STTI/CR/TECHNO/cleanroom.html)
- Cleanroom Software Engineering, Inc. (http://www.csn.net/˜deckm)

Abschnitt 8

Anforderungsbestimmung

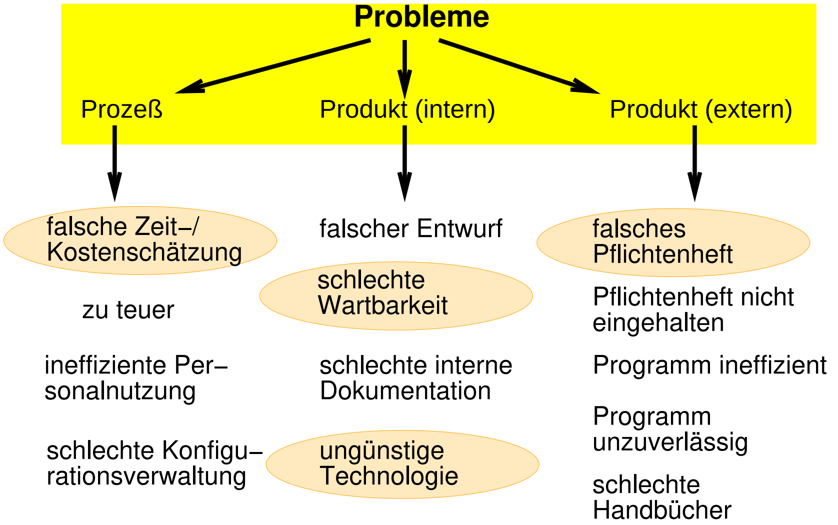

Abbildung 8.1: Von der Anforderungsbestimmung angesprochene Problembereiche.

8.1 Einordnung und Ziele

Vorbemerkung zur Begriffsdefinition: „**Kunde**“ ist bei der Anforderungsbestimmung die Person oder Institution, die die Software bezahlt. „**Anwender**“ sind die Personen(gruppen), die die Software persönlich benutzen oder von ihr unmittelbar in ihrer Arbeit beeinflußt werden. Im Falle von Fertigsoftware sind Kunde und Anwender also meist identisch, bei Auftragssoftware jedoch nicht. Auf der Entwicklerseite kann man unterscheiden zwischen Analytikern, die die Anforderungsbestimmung durchführen, und dem restlichen Softwareteam, das die Software realisiert. Diese Unterscheidung ist jedoch in der Praxis unscharf und wird hier deshalb nicht immer gemacht.

8.1.1 Übersicht

Die Anforderungsbestimmung umfaßt die gesamte erste Phase der Durchführung eines Softwareprojekts. Sie dient dazu festzustellen, *was* eigentlich gebaut werden soll (im Gegensatz zu *wie*).

Dabei gibt es drei verschiedene Arten von Anforderungen: Funktionale Anforderungen, Qualitätsanforderungen und Randbedingungen (z.B. Hardware, Betriebssystem, vorhandene Software etc.). In diesem Abschnitt werden wir uns hauptsächlich (aber nicht ausschließlich) damit beschäftigen, wie man die funktionalen Anforderungen feststellt.

Der Prozeß der Anforderungsbestimmung ist auch unter diversen anderen Bezeichnungen bekannt: auf Deutsch als Anforderungsanalyse oder Anforderungsdefinition und auf Englisch als Requirements Elicitation, Requirements Engineering, Requirements Definition oder Requirements Gathering. Gelegentlich werden mit diesen Begriffen unterschiedliche Teilaspekte des Problems bezeichnet, oft aber auch nicht. Der deutsche Begriff Anforderungsbestimmung ist zwar eigentlich am zutreffendsten, das Englische requirements engineering drückt jedoch das Prozeßhafte daran besser aus.

Ergebnisse der Anforderungsbestimmung sind einerseits ein **Pflichtenheft** (das die Ergebnisse der Anforderungsbestimmung aus der Sicht des Kunden beschreibt) und andererseits eine **Spezifikation** (die das dazu korrespondierende Dokument für das Entwicklungsteam darstellt). Ob man das Erstellen der Spezifikation als Teil der Anforderungsbestimmung betrachtet oder nicht, ist im wesentlichen eine Ansichtssache. Allerdings werden beim Erstellen der Spezifikation oftmals Lücken oder Inkonsistenzen im Pflichtenheft aufgedeckt, die mit dem Kunden geklärt werden müssen, so daß das Erstellen der Spezifikation besser nicht erst komplett nach der Anforderungsbestimmung abläuft.

In der Vorlesung „Softwaretechnik“ wurde im wesentlichen nur der reine Spezifikationsaspekt behandelt, d.h. es wurde nur die Aufgabe betrachtet, eine bekannte Anforderung so aufzuschreiben, daß ein Entwicklungsteam sie in Software umsetzen kann. Hier wollen wir stattdessen nur das Problem betrachten, wie man die Anforderung selbst überhaupt ermittelt.

Für viele Softwareprojekte ist die Anforderungsbestimmung die kritischste Phase der ganzen Entwicklung. Das hat zwei Gründe: Erstens sind falsch bestimmte Anforderungen stets sehr teure Fehler, weil ihre Auswirkungen auf das gebaute System tendenziell umfangreich sind. Zweitens kann ein Projekt mit falsch bestimmten Anforderungen leicht ein totaler Fehlschlag werden, weil die Fehler über den gesamten Verlauf des Projektes nicht entdeckt werden. Das Softwareteam orientiert sich an den Anforderungen so wie sie sind, und die Anwender bemerken die Probleme erst bei der Einführung des Systems (oder sogar erst eine Weile nach der Einführung).

Aus diesem Grund kommt der Anforderungsbestimmung ein hoher Stellenwert zu, der in der Informatikausbildung leider meist sträflich vernachlässigt wird.

Drei Hinweise zum bisher gesagten sind unbedingt zu beachten:

1. Die Sichtweise der Anforderungsbestimmung als „Ermittle das *Was*, aber nicht das *Wie*“ ist eine Idealisierung. Es ist praktisch unvermeidlich (und auch nicht unbedingt schädlich), daß die Anforderungsbestimmung bereits Entwurfsentscheidungen vorwegnimmt oder zumindest einengt.

2. Die Unterscheidung zwischen funktionalen und nichtfunktionalen Anforderungen bedeutet nicht, daß diese Klassen unabhängig voneinander sind. Vielmehr schränken nichtfunktionale Anforderungen oftmals die funktionalen ein und umgekehrt. Zum Beispiel schließen hohe Leistungsanforderungen gewisse komfortable Funktionsmerkmale aus. Das gleiche gilt für manche hohen Anforderungen an die Sicherheit, die manche heuristischen Funktionsansätze zu riskant machen und deshalb verbieten.

3. Auch die Vorstellung, ein festes Pflichtenheft erstellen zu können, das über die Projektlaufzeit unverändert bleibt, ist unrealistisch. Stattdessen sind erstens Anforderungsbestimmungen meist unvollständig und werden erst im Verlauf der Softwareerstellung allmählich

komplettiert und zweitens ändern sich fast immer manche der Anforderungen schon vor Fertigstellung der Software. Man könnte sagen, es laufen meistens Entwicklung und Wartung parallel ab.

Alle Aussagen zur Anforderungsbestimmung, die diese drei Tatsachen nicht ausdrücklich anerkennen, gelten deshalb nur mit entsprechenden Einschränkungen.

8.1.2 Fragestellungen

Im einzelnen müssen folgende Probleme gelöst werden, damit eine Anforderungsbestimmung erfolgreich ist (dies ist eine erweiterte Fassung der Aufstellung aus Abschnitt 3.2.4):

Wie kommuniziert man mit den Anwendern? Dies ist das vorderste und oftmals größte Problem der Anforderungsbestimmung: Wie führt man die völlig verschiedenen Erfahrungshintergründe und Terminologien von Anwendern und Softwareleuten so zusammen, daß eine erfolgreiche Kommunikation zustandekommt und Mißverständnisse vermieden werden?

Wie entdeckt man Anforderungen? Häufig sind die Anwender nicht nur unfähig, ihre Anforderungen so auszudrücken, daß ein Softwaremensch sie verstehen kann, sondern sie kennen sie schlichtweg gar nicht. Es ist daher ein gemeinsames Ziel von Anwendern und Softwareteam, zunächst einmal überhaupt zu entdecken, welche Anforderungen man sinnvollerweise an ein Softwaresystem haben könnte oder sollte.

Wie setzt man vage Anforderungen in konkrete um? Anforderungen haben oft einen sehr vagen, qualitativen Charakter. Musterbeispiel hierfür ist die Forderung nach Benutzerfreundlichkeit. Ein wichtiges Ziel der Anforderungsbestimmung ist die Umwandlung solcher Anforderungen in eine Form, die für beide Seiten als Vertragsgrundlage akzeptabel ist, klare Richtlinien für die Systemgestaltung vorgibt und zweifelsfrei geprüft werden kann.

Wie versteht man Prioritäten und Zufriedenheitskriterien des Anwenders? Dieses Problem ist eng mit dem vorigen gekoppelt. Bei Konflikt zwischen Anforderungen oder Auftreten von Problemen im Projekt ist die Prioritätensetzung eine kritische Frage, die früh geklärt werden muß. Hierzu, und beim Versuch, vage Anforderungen zu verstehen, ist es nützlich, ein Verständnis dafür zu haben, *warum* ein Anwender mit irgendeiner Lösung zufrieden oder eben nicht zufrieden ist.

Wie stellt man Eindeutigkeit sicher? Besagte Konflikte zwischen Anforderungen sollten natürlich möglichst früh erkannt und aus der Anforderungsbeschreibung entfernt werden. Es stellt sich also die Frage, wie man sie systematisch auffinden und auflösen kann.

Wie geht man mit Unvollständigkeit um? Idealerweise ist die Anforderungsbestimmung beendet, wenn alle Anforderungen genau bestimmt sind. Es fragt sich also, wie man das feststellt. In sehr vielen Fällen nimmt man jedoch bewußt Unvollständigkeiten in Kauf. Dabei entsteht das Problem, welche (Arten von) Unvollständigkeiten akzeptabel sind und wie man weiter mit ihnen verfährt.

Wie löst man Interessenkonflikte? Alle diese Arbeiten werden in vielen Fällen dadurch erschwert, daß es Interessenkonflikte zwischen verschiedenen Anwendern gibt oder zwischen Anwendern (z.B. Belegschaft) und Auftraggebern (z.B. Firmenleitung). Ein Beispiel wäre der Datenschutz: Viele Daten, die sich leicht erfassen ließen, wären für die Firmenleitung ein attraktives Mittel zur Überwachung der Arbeitsleistung ihrer einzelnen Mitarbeiter, führen aus deren Sicht jedoch zu einer Verletzung ihrer Persönlichkeit („big brother“). Ein Ignorieren solcher Konflikte führt fast immer zu einer schlechten Anforderungsdefinition. Deshalb müssen die Konflikte klar identifiziert und offen ausgetragen werden, um zu einem eindeutigen und sinnvollen Anforderungskatalog zu gelangen.

8.2 Ansätze

8.2.1 Arbeitsphasen

Die Anforderungsbestimmung umfaßt drei Schritte:

1. Ermittlung der Anforderungen
2. Beschreibung der Anforderungen
3. Analyse der Anforderungen

Bei der **Ermittlung** der Anforderungen geht es darum, in Zusammenarbeit zwischen Anwendern und Projektteam zu ermitteln, was für Eigenschaften des Softwaresystems die Anwender benötigen. Dieser Aspekt ist der bei weitem schwierigste und wird in diesem Kapitel im Mittelpunkt stehen. Für die Ermittlung der Anforderungen gibt es eine Reihe von teil- oder vollstrukturierten Techniken, wie Bereichsexperten (domain experts), sprich Anwender, und technische Experten (Softwareingenieure) ein gemeinsames Verständnis der Anforderungen erarbeiten können. Die Verfahren werden um so komplizierter, je mehr verschiedene Teilbereiche von verschiedenen Bereichsexperten abgedeckt werden müssen und je weiter das Anwendungsfeld vom Verständnishorizont der Softwareingenieure oder die technische Systemrealisierung vom Verständnishorizont der Anwender entfernt liegt.

Für die **Beschreibung** der Anforderungen ist einerseits die Menge der möglichen **Notationen** von Interesse, aus der einige Beispiele in der Vorlesung „Softwaretechnik“ behandelt wurden, sowie andererseits die Menge der Konventionen für die Verwendung der Notation. Auf das letztere Problem wird hier nur sehr am Rande eingegangen.

Die **Analyse** der Anforderungen, soweit sie noch *vor* der Entwurfsphase vorgenommen wird, dient zur Qualitäts- und Plausibilitätsprüfung der Arbeitsergebnisse der Anforderungsbestimmung. Schwächen des Anforderungskatalogs, die hierbei aufgedeckt werden, müssen in einer Nachbearbeitungsphase aufgeklärt werden.

Vielfach wird der Begriff des requirements engineering in Zusammenhängen benutzt, in denen die Beschreibung und die Analyse im Mittelpunkt stehen. Das Kernproblem ist aber eigentlich die Ermittlung. Es ist jedoch zutreffend, daß sich viele Schwierigkeiten bei der Ermittlung beseitigen ließen, wenn man bessere Beschreibungsmittel zur Kommunikation zwischen Anwender und Entwickler hätte. Sinnvollerweise durchläuft man alle drei Schritte ohnehin zyklisch und mit Rückkopplung, da bei der Beschreibung und Analyse Fragen auftauchen, die in die Ermittlung eingehen müssen oder bei ihr hilfreich sind.

8.2.2 Anwendungsbereiche

In unterschiedlichen Arten von Softwareprojekten stellen sich auch die Probleme der Anforderungsbestimmung verschieden dar. Generell wird die Anforderungsbestimmung um so schwieriger, je tiefer das zu gestaltende Softwaresystem in das Leben und die Arbeit der betroffenen Anwender eingreift.

Der schwierigste Fall sind deshalb normalerweise **betriebliche Informationssysteme**, die oft zentrale Faktoren des Arbeitslebens vieler Anwender darstellen. Hier wird erstens die Anforderungsbestimmung dadurch erschwert, daß das Softwaresystem nicht isoliert betrachtet werden darf, sondern im Zusammenhang des Gesamtsystems gesehen werden muß, das die gesamte Firma und ihre Wirtschaftstätigkeit umfaßt. In diesen Fällen sind zweitens auch Interessenkonflikte besonders häufig und zugleich besonders schwierig aufzulösen. Drittens tritt bei diesen Systemen die größte Verständniskluft zwischen Anwendern und Softwareteam auf.

Ähnlich problematische Fälle, wenn auch aus teilweise etwas anderen Gründen, sind alle Arten von **Benutzerschnittstellen**. Hier schwanken Erfahrungshintergründe der Benutzer und Anwendungskontext oft besonders stark. Wenn es um Massensoftware geht, wie beispielsweise ein Textverarbeitungsprogramm, ist es oft sehr schwierig, an die nötigen Informationen zu gelangen, um die Anforderungen optimal zu bestimmen, weil nicht genügend viele genügend verschiedene Anwender zugänglich sind. Außerdem spielen bei Benutzerschnittstellen kognitionspsychologische Phänomene eine große Rolle. Da die Benutzbarkeit von Software zu einem großen Teil von der Gestaltung der Benutzerschnittstelle abhängt, ist dies ein wichtiges Teilproblem der Anforderungsbestimmung in fast allen Anwendungsbereichen. Die Konstruktion von Benutzerschnittstellen ist jedoch eine eigene Disziplin, die deshalb hier in der Betrachtung ausgespart wird.

Oftmals etwas einfacher ist Software für **technische Anwendungen**, wie beispielsweise Konstruktionssysteme. Hier sind die Anwender häufig mit mehr Verständnis für Datenverarbeitung ausgestattet und können dementsprechend leichter mit einem Softwareteam kommunizieren. Auch hier gibt es aber Unterschiede, je nach konkretem Anwendungsgebiet.

Nochmals einfacher sind zumeist **Steuerungssysteme** und ähnliches, weil dort die Anforderungen zumeist durch technische Randbedingungen sehr viel enger vorgegeben sind als in den prinzipiell extrem flexiblen Umgebungen, in denen nur Menschen arbeiten. Eine Herausforderung besteht hier (wie auch auf allen anderen Anwendungsgebieten) darin, gänzlich neue Anwendungsarten von Software herauszufinden und zu beschreiben. Ein Beispiel hierfür ist die Realisierung eines Gasventils mit stufenlos wählbarer Durchflußmenge durch ein (wesentlich billigeres) binäres Gasventil plus Software.

Mit diesem Beispiel sind wir auch schon in der verwandten Kategorie „**eingebettete Software**“ (embedded systems). Hier ist die Software gar nicht mehr separat sichtbar sondern verschwindet als ein Funktionsbestandteil in einem technischen System, beispielsweise obigem Ventil oder der Steuerung einer Waschmaschine, eines Fernsehgeräts, eines Antiblockiersystems oder einer Uhr. Die Anforderungen werden hierbei noch enger von Randbedingungen bestimmt als bei Steuerungssoftware für größere Systeme und sind dementsprechend leichter zu bestimmen. Ausnahmen bilden die zahlreichen Fälle, in denen Benutzerschnittstellen gestaltet werden müssen, wie bei Waschmaschine und Fernseher.

Mit sinkender Schwierigkeit der Anforderungsbestimmung steigen üblicherweise zugleich die Anforderungen an die Zuverlässigkeit der Software. Der Grund für beide Tendenzen ist der gleiche: starke direkte Beteiligung von Menschen an der Benutzung der Software. Wo der Mensch eng ins System verflochten ist, sind die Anforderungen zwar schwer zu bestimmen, dafür können Fehler des Systems eher toleriert werden, weil der Mensch als Kontrollinstanz Probleme ausgleichen kann. Umgekehrt versteht man recht leicht, was ein Antiblockiersystem tun soll, aber wehe, wenn es nicht korrekt funktioniert.

8.3 Ergebnisse

Es gibt nur wenig konkrete Forschung über die tatsächliche Wirksamkeit verschiedener Techniken bei der Anforderungsbestimmung. Das liegt unter anderem daran, daß die Erforschung dieser Wirksamkeit mit kontrollierten Experimenten außerordentlich schwierig ist: Ein absolutes Messen von Erfolg ist unmöglich und ein direkter Vergleich von Techniken scheitert daran, daß einerseits nicht ein Anwender/Softwareteam-Paar mit zwei Techniken am selben Problem arbeiten kann und andererseits die Schwankungen zwischen Anwendern so stark sind, daß ein Vergleich mit randomisierten Vergleichsgruppen zu aufwendig würde. Deshalb gibt es an Forschung überwiegend nur Fallstudien.

[KC95] beschreiben eine Studie über die Effektivität verschiedener Vorgehensweisen und Hilfsmittel zur Anforderungsbestimmung. Im Schnitt waren die Projekte erfolgreicher, die mehr verschiedene Vorgehensweisen und Hilfsmittel benutzten, jedoch übersteigen beim Hinzunehmen irgendwann die Kosten den zusätzlichen Nutzen. Ferner verheißen direkte Kommunikationskanäle zwischen Entwicklern und Anwendern mehr Projekterfolg als indirekte.

8.4 Probleme und Hindernisse

Die Schwierigkeiten bei der Anforderungsbestimmung rühren im wesentlichen daher, daß sie ein Prozeß ist, der erstens von Menschen durchgeführt wird und sich zweitens mit einem beträchtlichen Teil der Komplexität der von Menschen gestalteten Welt herumplagen muß. Im Unterschied zum Softwareentwurf oder zur Softwareimplementation kann diese Komplexität aber hier nur wesentlich schwieriger mit Mitteln der Formalisierung, Abstraktion und Automation beherrscht werden, weil die Anforderungsbestimmung ja gerade diese Abstraktion erst ermitteln soll.

Die meisten der Schwierigkeiten sind oben bereits angesprochen worden, deshalb hier nur noch einmal kurz und in einer anderen Sichtweise: Es gibt vier grundsätzliche Bereiche, in denen Schwierigkeiten auftreten, nämlich das Verständnis des Anwendungsbereichs, die Kommunikation, das soziale Zusammenwirken der Beteiligten und das Projektmanagement.

Dies sind die **Probleme beim Verstehen des Anwendungsbereichs**:

- Die Anwender kennen zwar ihren Arbeitsbereich, können aber die darin vorhandenen Probleme, die eine DV-Lösung erlauben, nicht erkennen.
- Auch wo Anwender ihre Probleme erkennen, sind sie oft nicht in der Lage, eine vorgeschlagene DV-Lösung zu beurteilen und konstruktiv zu kritisieren.
- DV-Lösungen, die sehr anders strukturiert sind als die bisherige Arbeitsweise, oder die ganz neue Arbeiten einführen, übersteigen oft das Vorstellungsvermögen der Anwender und eventuell auch der Entwickler.
- Entwickler verstehen viele der Abstraktionen nicht, die der Anwender zur Beschreibung von Anforderungen benutzt, weil ihnen Verständnis für den Anwendungsbereich fehlt.
- Entwickler haben oft ein fast nur technisches Interesse und versagen dabei, eine nichttechnische Anforderung zu erkennen, zu verstehen und umzusetzen.

Dies sind die wichtigsten **Kommunikationsprobleme**:

- Anwender und Entwickler haben völlig verschiedene Begriffswelten und Erfahrungshintergründe, oftmals gar mit verschiedenen Begriffen hinter dem gleichen Wort.
- Sie sind verschiedene Beschreibungsmethoden und Notationen gewöhnt.
- Sie haben Schwierigkeiten, sich auf ein Abstraktionsniveau zu einigen, weil sie nicht genau verstehen, welches der andere gerade benutzt oder benutzen will.

Ferner gibt es oft auch **soziale Hindernisse**:

- Verschiedene Einzelpersonen oder Gruppen unter den Anwendern haben unterschiedliche, eventuell widersprüchliche Interessen. In manchen Fällen ist dies das größte Problem bei der Anforderungsdefinition; vor allem, wenn die Konflikte verwischt statt gelöst werden.
- Anwender sehen oftmals die Notwendigkeit der Anforderungsbestimmung nicht ein, weil sie glauben, daß die Aufgabenstellung klar sei. Sie haben meist kein Interesse an Details, die aber zur Softwareentwicklung entscheidend wichtig sind.
- Die Entwickler haben oft keine Leistungsanreize, die direkt an der Kundenzufriedenheit orientiert sind.
- Entwickler haben oft ein hinderliches Überlegenheitsgefühl gegenüber den Anwendern.

Schließlich gibt es auch **technische Probleme** bei oder aufgrund der Anforderungsanalyse, die man im weiteren Sinne einem geschickten Projektmanagement zuordnen kann:

- Bei Fertigsoftware wird der Erfolg oft nicht von der Befriedigung der tatsächlichen Anwenderbedürfnisse bestimmt, sondern von den eventuell davon recht verschiedenen Anforderungen des Marktes. Dadurch gewinnen zusätzliche Aspekte wie Zeit bis zum Markteintritt, werbewirksame Features (ob nützlich oder nicht), Anpassung an Produktprofil der Firma etc. eine erhebliche Bedeutung, die bei der Anforderungsbestimmung zu berücksichtigen ist.
- Ein Entwicklungsprozeß, der mit unvollständigen oder sich ändernden Anforderungen umgehen muß, ist kompliziert und teuer. Deshalb muß die Anforderungsanalyse zumindest an den entscheidenden Stellen diese Schwierigkeiten weitmöglichst reduzieren.
- Die stark kostensenkende Einbindung umfangreicher Fertigkomponenten in ein Softwaresystem hat oftmals Rückwirkungen auf die realisierbaren Anforderungen oder zumindest deren Form. Diese müssen bei der Anforderungsbestimmung ausdrücklich berücksichtigt werden.

8.5 Analyse der Anforderungen

Auf die Analyse zuvor ermittelter Anforderungen will ich hier nur kurz eingehen. Die Analyse der Anforderungen dient, wie schon gesagt, zur Qualitäts- und Plausibilitätsprüfung des Anforderungskatalogs. Sie ist ein wichtiges Hilfsmittel auch für die Anforderungsbestimmung, weil Lücken und Widersprüche, die bei der Analyse gefunden werden, die Aufmerksamkeit bei der Bestimmung auf diejenigen Aspekte lenken, wo sie am meisten gebraucht wird. An Anforderungen und ihre Dokumentation gibt es folgende Qualitätsanforderungen:

1. Vollständigkeit. Ist die Anforderungsliste komplett? Oder sind zumindest alle Lücken bekannt?
2. Konsistenz. Ist sie frei von inneren Widersprüchen?
3. Eindeutigkeit. Hat jede Anforderung genau eine klar verstandene Bedeutung? Wird diese auch wirklich verstanden? Sowohl vom SW-Hersteller als auch vom Kunden?
4. Nachprüfbarkeit. Kann später im Softwaresystem für jede Anforderung unstrittig entschieden werden, ob sie erfüllt ist?
5. Rückverfolgbarkeit. Kann man für jede Anforderung nachvollziehen, warum sie gestellt wurde?
6. Modifizierbarkeit. Können einzelne Anforderungen leicht geändert werden, ohne die Beschreibung anderer Anforderungen mitändern zu müssen?
7. Realisierbarkeit. Können die Anforderungen mit den vermutlich verfügbaren Mitteln implementiert werden?
8. Einschätzbarkeit. Kann für die Projektplanung geschätzt werden, wieviel Mittel für jede Anforderung nötig sein werden?

Zur Prüfung dieser Qualitätsanforderungen können hauptsächlich zwei allgemeine Methoden eingesetzt werden: Konsistenzprüfung und Vollständigkeitsprüfung. Bei der **Konsistenzprüfung** wird die erstellte Spezifikation auf innere Widersprüche untersucht, was zumindest bei Verwendung formaler Spezifikationssprachen oft relativ leicht möglich ist. Bei der **Vollständigkeitsprüfung** wird versucht festzustellen, ob alle Anforderungen auch erfaßt sind. Diese Prüfung ist natürlich nicht vollständig möglich; also wird z.B. untersucht, ob Fallunterscheidungen mit erkennbarer Fallmenge vollständig sind und ob alle Teile der Spezifikation, die an anderer Stelle benutzt werden, auch definiert sind. Beim Einsatz formaler Spezifikationsmethoden können beide Prüfungen teilweise automatisiert werden, was ihren Aufwand vermindert und ihre Zuverlässigkeit erhöht.

Die anderen Qualitätsmerkmale können nicht so einfach geprüft werden, jedoch gibt es auch für sie zum Teil Möglichkeiten, ihre Einhaltung besser sicherzustellen: Die **Rückverfolgbarkeit** kann mit Hilfe von entsprechenden Werkzeugen hergestellt werden, die die Entstehungsgeschichte einer Anforderung während der Anforderungsbestimmung aufzuzeichnen gestatten. Wenn solche Werkzeuge auch Querbezüge zwischen den Anforderungen aufzeichnen können, läßt sich auch die **Modifizierbarkeit** verbessern, weil das Werkzeug das Mitändern oder Neuüberprüfen anderer Anforderungen zwar nicht unnötig macht, aber zumindest seine Durchführung erleichtert.

Die **Eindeutigkeit** von Anforderungen wird zum Teil überprüft, wenn man sie in eine formale mathematische Notation überführt. Allerdings ist die Eindeutigkeitsprüfung von sprachlich formulierten Anforderungen zur Zeit noch in jedem Fall etwas, das nur ein Mensch machen kann und deshalb selbst dann fehleranfällig: wenn beim Überführen in formale Notation eine Mehrdeutigkeit nicht erkannt wird, sondern eine der möglichen Interpretationen ausgewählt wird, kann trotz späterer formaler Beschreibung ein Mißverständnis entstanden sein. Die Eindeutigkeit umfaßt auch, daß auch der Kunde die Anforderung, in der Form wie sie im Pflichtenheft auftaucht, wirklich verstanden hat. Bei entsprechender Sorgfalt bei der Anforderungsbestimmung kann das normalerweise immer geprüft bzw. sichergestellt werden, aber diese Sorgfalt ist extrem kritisch für den Erfolg eines Projekts: Eine Schlamperei in diesem Punkt wird nämlich oftmals erst viel zu spät, z.B. bei der Abnahme eines Systems entdeckt. Für Software, die nicht für spezifisch vorhandene Kunden, sondern für einen Markt von vielen Kunden entwickelt wird, ist dieser Aspekt nur scheinbar ohne Bedeutung. Aber auch hier wird ja die Anforderungsbestimmung aufgrund von Befragungen potentieller Kunden erstellt, und ein Fehlverständnis von deren Wünschen kann den Markterfolg des Produkts zunichte machen.

Die **Realisierbarkeit** wie auch die **Nachprüfbarkeit** und die **Einschätzbarkeit** lassen sich kaum maschinell prüfen. Hierzu sind immer noch Wissen, Erfahrung, Gefühl und Überschlagsrechnungen die Mittel der Wahl.

8.6 Grundtechniken

Dieser Abschnitt skizziert das Repertoire von Basistechniken, die zur Anforderungsbestimmung zur Verfügung stehen. Der Einsatz dieser Techniken kann in sehr verschiedenen Zusammenhängen und zu verschiedenen Zwecken erfolgen; ebenso kann die konkrete technische Ausprägung recht unterschiedlich sein. Auf alle diese Aspekte können wir hier nicht näher eingehen.

8.6.1 Befragungstechniken

Befragungstechniken sind tendenziell analytikergetrieben, d.h. die Aktivität, die die Richtung bestimmt, in der die Anforderungsanalyse fortschreitet, geht überwiegend vom Analytiker aus.

Die klassische Technik zur Systemanalyse in diesem Zusammenhang ist das Interview. **Interviews mit Benutzern** streben an, direkt von den einzelnen betroffenen Anwendern möglichst präzise und unverfälschte Auskünfte über deren Anforderungen zu bekommen. Demgegenüber wird bei **Interviews mit Kontaktpersonen** nur je ein Vertreter einer ganzen Personengruppe interviewt. Dieses Verfahren ist effizienter, führt aber auch eher zu inkorrekter und unvollständiger Information.

Umfragen in Freitextform oder mittels **Fragebogen** sind ebenfalls ein recht effizienter Weg, Anforderungsinformationen zu erhalten und erlauben die Einbeziehung einer recht großen Zahl von Anwendern. Wegen der fehlenden Interaktion sind sie jedoch nur zur Klärung von Fragen geeignet, bei denen voraussichtlich alle relevanten Alternativen bereits bekannt sind.

Gespräche im Marketing und auf Messen sind ähnlich zu beurteilen wie Interviews mit Kontaktpersonen. Sie haben den zusätzlichen Vorteil, daß der betreffende Anwender meist ein aktives und produktorientiertes Interesse an dem Gespräch hat, jedoch den Nachteil, das der Detaillierungsgrad eventuell zu gering bleibt und ein zweites Gespräch zur Klärung späterer Rückfragen oft schwer möglich ist.

8.6.2 Sammeltechniken

Bei Sammeltechniken geht die Aktivität vom Anwender aus. Diese Klasse von Techniken eignet sich deshalb vor allem zur Verfeinerung von Anforderungskatalogen bei bereits existierender Software, also zur Vorbereitung einer neuen Softwareversion.

An Quellen zum Sammeln von Anforderungen steht erstens die **Hotline** (telefonisch oder per Email) zur Verfügung. Hieraus können vor allem existierende Schwachstellen der Software im Hinblick auf die Benutzerschnittstelle und Dokumentation entdeckt werden. **Wünsche von einzelnen Benutzern** decken hingegen vor allem fehlende Funktionalität ab. Die Qualität solcher Information ist gut, weil die Anforderungen aus konkreten praktischen Problemen erwachsen.

Noch besser zu beurteilen sind oft **Wünsche von organisierten Benutzergruppen**, weil hier die Wünsche bereits durch Diskussion gefiltert und ausgereift sind. In vielen Fällen legen Benutzergruppen ganze Entwurfsvorschläge vor.

8.6.3 Gruppentechniken

Gruppentechniken, zumeist in Form **gemeinsamer Sitzungen** von Analytiker(n) und mehreren Anwendern, balancieren die Aktivität zwischen beiden Gruppen. Die Anforderungsbestimmung kann partnerschaftlich vorgenommen werden. Das ist bei Interviewtechniken meist deshalb nicht möglich, weil einzelne Anwender nicht genügend Initiative in die Diskussion einbringen, um sie sinnvoll zu steuern.

Die klassische Perspektive von Gruppensitzungen besagt, daß zwar Analytiker und Anwender ein Team bilden sollen, die Verantwortung für die Herstellung des Anforderungskatalogs aber letztendlich bei den Analytikern liegt.

Partizipatorischer Entwurf dreht dies quasi um: Die Analytiker werden eher als Zuarbeiter der Anwender betrachtet, die in einem demokratischen Prozeß „ihre“ Anforderungen definieren sollen. Die Idee des partizipatorischen Entwurfs ist vor allem im skandinavischen Raum populär.

8.6.4 Beobachtungstechniken

Beobachtungstechniken machen die Anforderungsbestimmung quasi implizit: Es werden nicht verbal Ansichten über Anforderungen zwischen Analytikern und Anwendern ausgetauscht, sondern die Anwender tun die Arbeit, die das Softwaresystem unterstützen soll und die Analytiker extrahieren die Anforderungen aus ihren Beobachtungen dieser Arbeit. Dieses Paradigma ist bei weitem nicht immer einsetzbar, hat dann aber den Vorzug, praxisrelevante Anforderungen mit hoher Wahrscheinlichkeit sichtbar zu machen.

Beispiele sind **Benutzbarkeitslabors**, in denen existierende Software und Prototypen als Grundlage für konkrete Anforderungsbestimmungen dienen. Damit verwandt ist die Technik der **Protokollauswertung**, die als Beobachtungsgrundlage Daten benutzt, die von speziell instrumentierten Programmen mit Einverständnis der Benutzer während derer normaler Arbeit gesammelt wurden. Diese Techniken eignen sich besonders zum Ausbügeln einzelner Schwächen von Software, die zu Fehlern bei der Arbeit führen, und zur Verbesserung von Benutzerschnittstellen.

Reichen diese Möglichkeiten nicht aus, so kann **Feldbeobachtung** helfen, bei der Anwender direkt in ihrer Arbeit beobachtet werden, um Anforderungen zu ermitteln. Problematisch an Feldbeobachtungen kann erstens der hohe Aufwand sein und zweitens die Tatsache, daß viele Anwender unter Beobachtung befangen sind, sich unwohl fühlen und deshalb möglicherweise nicht so arbeiten wie sonst üblich.

Eine verfeinerte Feldbeobachtungsmethode ist das **„in die Lehre gehen“**[BH95]: Ein Analytiker begleitet den Anwender bei seiner täglichen Arbeit wie ein Lehrling, der alle Arbeitsgänge erst verstehenlernen muß. Bereits das bloße Zuschauen liefert gute Informationen darüber, worauf es ankommt und zwar sowohl in großen Strukturen, als auch in Details. Zusätzlich ermöglichst das In-die-Lehre-Gehen Zugriff auf die Erfahrungen des Anwenders durch direkte Nachfragen. Ein Sich-beobachtet-Fühlen wird eher vermieden als bei normaler Feldbeobachtung, weil das In-die-Lehre-Gehen anerkennt, daß der Anwender, nicht der Analytiker, der Experte auf seinem Gebiet ist. Beyer und Holtzblatt schildern als mögliche Probleme erstens das Fallen in andere

Beziehungsmodelle als Lehrer/Lehrling, zweitens das Abgleiten ins Abstrakte anstatt der Behandlung konkreter Fälle bei Nachfragen und drittens zu allgemeine Nachfragen des Analytikers anstatt Anbieten einer konkreten Interpretation, die der Anwender dann akzeptieren, ablehnen oder modifizieren kann.

8.6.5 Hilfsmittel

Es gibt (abgesehen von konkreten Werkzeugen und Notationen für spezifische Methoden) eine Reihe von Hilfsmitteln, die sich recht vielseitig einsetzen lassen.

Reservierte Räume. Soweit eine Anforderungsbestimmung ganz oder weit überwiegend in einem einzigen Gebäude stattfindet, ist ein gutes Hilfsmittel ein ständig dafür reservierter Raum. Erstens bietet dies Gelegenheit, eine Vielzahl von Zeichnungen, Notizen, etc. an den Wänden aufzuhängen, was eine besonders überschaubare und diskussionsfördernde Aufbewahrungsart der Zwischenergebnisse darstellt. Zweitens fördert ein solcher Raum das Zusammengehörigkeitsgefühl der Beteiligten und kann deshalb helfen, Kommunikations- und Kooperationsbarrieren abzubauen.

Prototypen sind das Mittel der Wahl, wenn es darum geht, kritische, aber ungenügend verstandene Konzepte zu prüfen, die eine wichtige Rolle in der anvisierten Software spielen sollen. In schwierigen Fällen kann man mehr oder weniger vollständige Prototypen eines Systems oder Systemteils bauen, zumeist wird man jedoch Scheinprototypen erstellen, deren Funktionalität getürkt ist und sich auf die Bearbeitung einer kleinen Zahl „festverdrahteter" Anwendungsfälle beschränkt. Besonders hilfreich sind Prototypen von Benutzerschnittstellen. Prototypen lassen sehr viele Aspekte des Entwurfs bereits in die Anforderungsanalyse einfließen, was durchaus nützlich und kostensparend sein kann, obwohl es natürlich die Gefahr birgt, wichtige Implementationsentscheidungen zu früh und deshalb falsch zu treffen.

Papierprototypen sind Mengen von aus Papier ausgeschnittenen, beschrifteten und bemalten Schnipseln, die als Bausteine zur Veranschaulichung vorgeschlagener Systemfunktionen dienen. Diese Methode hat sich als sehr nützliches Hilfsmittel zur Beschleunigung und Konkretisierung der Diskussion über direkt benutzersichtbare Funktionalität erwiesen und wirkt vor allem durch Unterstützung der Vorstellungskraft und Vermeidung von Mißverständnissen.

Elektronisches Pflichtenheft: Wenn die betroffene Anwendergruppe aus eifrigen Computerbenutzern besteht, kann es hilfreich sein, ständig den aktuellen Stand der Anforderungsdiskussion elektronisch verfügbar zu halten; am besten als leicht erforschbares Hypertextdokument. Eine solche Maßnahme erlaubt, eine größere Zahl von Benutzern über einen längeren Zeitraum an der Diskussion lose zu beteiligen und kann deshalb gute zusätzliche Hinweise erbringen. Ein elektronisches Pflichtenheft ist vor allem dann wichtig, wenn der Entwurf der Software bereits nach Teilen der Anforderungsbestimmung beginnt, denn veraltete Papierdokumente sind eine fürchterliche Quelle von Fehlern.

Elektronische Diskussionsgruppen sind ebenfalls ein gutes Mittel, mehr Personen in den Prozeß einzubeziehen. Sie verringern außerdem durch ihren asynchronen Charakter eventuell den Zeitbedarf und vermeiden durch ihre Schriftform, daß Diskussionsbeiträge verlorengehen. Problematisch ist die Steuerung der Gewichtung: Zuwenig Aktivität in den Diskussionsgruppen macht sie wertlos, zuviel macht sie ineffizient.

Video kann ein unterstützendes Hilfsmittel bei vielen Techniken zur Anforderungsbestimmung sein. Anwender auf Videoband „mit nach Hause zu bringen" (also zum Softwarehersteller) ermöglicht ggf. die Einbeziehung zusätzlicher Analytiker zu schwierigen Einzelfragen und die

Klärung von im Nachhinein auftretenden Fragen (sowohl während der Anforderungsbestimmung als auch bei der Entwicklung und sogar in der Wartung) [BCW95]. Allerdings dürfen die Kosten für eine geeignete Verwaltung der Videoaufzeichnungen nicht unterschätzt werden. In Benutzbarkeitslabors erlauben Videoaufzeichnungen den Aufbau einer Bibliothek von „kritischen Anwendungsfällen“, für die im Systementwurf später geeignete Lösungen gesucht werden müssen.

8.7 Umgang mit Risiken und Konflikten

8.7.1 Anforderungsbestimmung als Konfliktlösungsprozeß

Die übliche Sicht der Anforderungsbestimmung betrachtet es als das Hauptproblem, die Anforderungen überhaupt zu finden und dann so genau zu beschreiben, daß sie korrekt implementiert werden können. Demgegenüber werden wir in diesem Abschnitt eine alternative Sicht benutzen, die die Anforderungsbestimmung hauptsächlich als einen Konfliktlösungs- und Risikomanagementprozeß betrachtet.

In dieser Sicht gibt es drei Klassen von Schwierigkeiten bei der Anforderungsbestimmung: **Unsicherheiten** (niemand weiß, wie es ist oder sein sollte), **Risiken** (man weiß, wie es sein soll, aber nicht, ob man das hinkriegt) und **Konflikte** (die Anforderungen verschiedener Parteien sind nicht unmittelbar miteinander zu vereinbaren).

Der Anforderungsbestimmungsprozeß besteht darin, diese Schwierigkeiten aufzulösen, was für alle drei Klassen mit Hilfe von Verhandlungen erfolgen kann, in denen zwischen den Beteiligten vereinbart wird, wie mit der Schwierigkeit umgegangen werden soll. Diese vereinheitlichende Sichtweise führt erstens dazu, daß wichtige Aspekte weniger leicht vergessen werden (Risiken und Unsicherheiten), die Anforderungen also vollständiger werden, zweitens dazu, daß Kommunikationsprobleme explizit (nämlich als Unsicherheiten) behandelt werden und drittens dazu, daß die in der konventionellen Sicht gar nicht ausdrücklich vorhandene Problemklasse der Konflikte besser bewältigt werden kann.

Diese beiden Sichten sind natürlich nicht äquivalent, überlappen sich aber stärker, als man zunächst meinen könnte: Die Hauptschwierigkeiten in der konventionellen Sicht sind Kommunikationsprobleme und Verständnisprobleme. Diese rühren jedoch vor allem aus der unterschiedlichen Sichtweise der Beteiligten her, die zum Teil von ihrer Herkunft erklärt wird und zum Teil von ihren verschiedenen Interessen. Sie lassen sich also als Risiko (Verständnisunsicherheit) und als Konflikt (Interessenlagen) deuten, so daß die Konfliktlösungssicht weite Teile der konventionellen Sicht mitabdeckt.

Das Interessante an der Konfliktlösungssicht ist, daß sie den Rahmen der reinen Anforderungsanalyse sprengt. Sie dient vielmehr als einheitliche Grundlage für das gesamte strategische Projektmanagement. Damit eröffnet sie insbesondere einen guten Weg, um mit sich ändernden Anforderungen während eines Projekts umzugehen, denn solche Änderungen verlangen ja gerade, die Anforderungsanalyse nicht nur als eine in sich abgeschlossene Phase vor Implementationsbeginn zu betrachten, sondern als eine ständig während der Projektdurchführung weiterzuführende Aktivität.

8.7.2 Theorie W

Der Hauptvertreter der Konfliktlösungssicht ist Barry Boehm, der die Verfahrensgrundlage dieser Sicht in Form einer einfachen allgemeinen Managementtheorie formuliert hat, genannt Theorie

W. Diese ist als Fortentwicklung dreier bisheriger Managementtheorien zu verstehen, die die Bezeichnungen Theorie X, Theorie Y und Theorie Z tragen und deshalb hier zuvor kurz vorgestellt werden sollen.

Theorie X baut auf den Ideen des „wissenschaftlichen Managements“ auf, die Frederick Taylor Anfang des 20. Jahrhunderts eingeführt hat. Sie besagt, daß die effizienteste Art, eine Arbeit zu erledigen, darin besteht, sie aufgrund von Zeit- und Bewegungsstudien so in glatt aneinanderpassende Teile zu zerlegen, daß die arbeitenden Menschen so effizient und vorhersagbar wie Maschinen arbeiten können. Management besteht in dieser Theorie darin, das System ohne Störungen am Laufen zu halten. Diese Theorie führte zur Einführung der Fließbandarbeit. Der große Nachteil dieser Theorie besteht darin, daß sie die Flexibilität, Kreativität und Selbstachtung der Menschen unterdrückt. Das führt unter anderem dazu, daß eine Organisation, die nach Theorie X arbeitet, sehr schlecht mit Wandel umgehen kann.

Theorie Y besagt deshalb, das Management solle vor allem die Kreativität und Initiative der Menschen fördern. Dies erzeugt jedoch viele Konflikte und hält keine Mittel bereit, mit ihnen umzugehen.

Theorie Z geht dieses Problem an, indem sie Vorabinvestitionen in die Ausbildung gemeinsamer Werte empfiehlt und wichtige Entscheidungen im Konsens zu fällen vorschlägt. Allerdings macht sie dabei die Annahme, daß alle diese Prozesse nur innerhalb einer Organisation stattfinden und hält wenig Rat bereit für den Fall mehrerer Organisationen und mehrerer Parteien, wie es bei Softwareentwicklung stets der Fall ist (Anwender, Kunde, Entwickler etc.).

Theorie W ist auf diesen Fall unterschiedlicher Parteien mit unterschiedlichen Interessen ausgerichtet. Bei der Softwareentwicklung haben die beteiligten Gruppen recht **unterschiedliche Interessen, z.B.**: Die Benutzer wünschen sich ein zuverlässiges, benutzerfreundliches System, das möglichst viele Funktionen hat, die sie in ihrer Arbeit unterstützen. Der Kunde möchte ein Produkt, das pünktlich und zum vereinbarten Preis geliefert wird und dann wie gewünscht funktioniert. Die Programmwarter möchten ein wohldokumentiertes, leicht änderbares System ohne Fehler. Die Entwickler wünschen sich technische Herausforderungen mit viel Entwurfs- und wenig Dokumentationsarbeit. Die Reihe ließe sich fortsetzen mit Testern, Projektmanagern und anderen.

Diese verschiedenen Interessen sind die Wurzel zahlreicher **Konflikte**. Wenn diese Konflikte nicht ausdrücklich erkannt und aufgelöst werden, sind sie die Ursache eine großen Menge von oft dramatischen Schwierigkeiten. Einen Konflikt aufzulösen bedeutet nicht nur, irgendeine Entscheidung herbeizuführen, sondern eine solche Entscheidung zu treffen, die für alle Parteien akzeptabel ist oder idealerweise sogar eine, mit der alle Parteien zufrieden sind. Das Grundprinzip von Theorie W lautet deshalb **„Mache jeden zum Gewinner (make everyone a winner)“**.

Dieses Prinzip besagt, daß bei Konflikten stets Lösungen angestrebt werden, die für alle Beteiligten vorteilhaft sind. Grundlage dieses übertrieben wirkenden Anspruchs ist die Beobachtung, daß oft Lösungen gefunden werden können, die sogenannte Gewinn-Gewinn (win-win) Situationen herstellen, wenn man nur den geeigneten Betrachtungsrahmen für die Beteiligten herstellt. Die normale Wahrnehmungsweise von Konflikten ist meist die von Nullsummenspielen: der Vorteil des einen ist der Nachteil des anderen, also eine Gewinn-Verlust (win-lose) Situation.

Beispielsweise ist eine nachlässige und deshalb billigere Softwareentwicklung ein Gewinn für den Entwickler und ein Verlust für den Kunden (win-lose). Die Situation läßt sich jedoch in eine Gewinn-Gewinn-Situation überführen, indem der Entwickler an dem Gewinn, den der Kunde aus der Software zieht, beteiligt wird. Nun wird die Softwareentwicklung sorgfältig sein und sowohl Kunde als auch Entwickler profitieren davon.

Im schlimmsten Fall treten Verlust-Verlust (lose-lose) Situationen auf: überhöhte Erwartungen, unrealistische Pläne, unverträgliche Leute in einem Projekt, oder der Versuch, Planrückstände durch Personalaufstockung aufzuholen, führen meist zu allseitigem Verlust. Leider sind diese Fehler gerade im Softwarebereich häufig zu beobachten, weil kein Risikomanagement und keine Konfliktauflösung betrieben wird.

Im Mittelpunkt von Theorie W steht also die Frage, wie man win-win Situationen erzeugt und erhält. Hierzu gibt es eine Reihe von Prinzipien:

- **Verstehe, wie die Beteiligten gewinnen wollen.** Hier kommt es erstens darauf an, die entscheidenden Beteiligten überhaupt zu identifizieren. Oft schlagen Softwareprojekte fehl, weil wichtige Beteiligte nicht in die Win-win-Überlegungen einbezogen worden sind, z.B. Unterauftragsnehmer oder Hardwarebeschaffer. Zweitens gilt es, sich in die Gewinnsituationen der anderen hineinzuversetzen. Häufige Fehler sind beispielsweise eine falsche Beurteilung der Karrierewünsche von Softwareleuten durch ihre Manager („Manager werden ist ein Karrierefortschritt") oder eine falsche Beurteilung der Anwenderwünsche durch die Softwareleute („Gut ist, was kompakt, mächtig und programmierbar ist" — das können Laien aber meist nicht bedienen).
- **Erzeuge angemessene Erwartungen.** Bringe die Beteiligten zusammen, um überzogene oder untertriebene Erwartungen zu entdecken, messe Erwartungen an Erfahrungswerten und kalibrierten Schätzmodellen. Eine alte Managementweisheit in diesem Zusammenhang besagt, daß hart-weich besser funktioniert als weich-hart, das heißt, man sollte lieber zuerst die Erwartungen entgegen dem Erwartungsdruck herunterschrauben (hart), um dann später die Erwartungen steigern oder übertreffen zu können (weich), anstatt zuerst allen Erwartungen nachzugeben (weich) und sie dann nicht einhalten zu können und zu enttäuschen (hart).
- **Passe die Aufgaben der Leute an ihre Gewinnbedingungen an.** Hier geht es darum, einerseits vorhandene win-win Situationen zu entdecken und zu realisieren und zweitens durch Erweiterung des Optionsraums neue win-win Bedingungen zu erschaffen. Dazu stehen eine Reihe von Techniken zur Verfügung:
  - Optionen entlang von Gewinnmöglichkeiten ausrichten, z.B. die Verantwortung für die Qualitätssicherung vom Entwicklungsteam, das Qualitätssicherung meist langweilig findet, auf die Programmwarter zu verlagern, die sich damit ihr späteres Leben leichter machen können.
  - Aufgaben an spätere Karrieremöglichkeiten oder an Belohnungen koppeln.
  - Einen Aufgabenbereich erweitern, z.B. die Qualitätssicherung als vorbeugende und organisationsweite Maßnahme definieren.
  - Besondere Unterstützung zusichern, z.B. anstrengende Aufgaben durch Entlastung auf anderen, langweilig empfundenen Gebieten attraktiv machen.
  - Neue Optionen auftun, z.B. einen nichterfüllbaren Plan durch die Einführung von Inkrementen in einen erfüllbaren Plan für eine akzeptable Teillösung und einen Zusatzplan für Erweiterungen umbauen.

Das Herstellen und vor allem Aufrechterhalten allseitiger win-win Bedingungen in einem Projekt erfordert die Einhaltung von zwei Hilfsprinzipien von Theorie W:

1. Plane gründlich und folge Deinem Plan („plan the flight and fly the plan").

2. Identifiziere und kontrolliere Deine Risiken.

Planen und Planeinhaltung ist wichtig, weil ein Plan die gegenseitige Zusicherung der Beteiligten ist, bestimmte win-win Bedingungen einzuhalten, und außerdem einen Rahmen für die Entdeckung von Abweichungen von diesen Bedingungen bildet. Ein Plan in Theorie W beantwortet zu diesem Zweck einheitlich immer eine Reihe von Fragen, nämlich *Warum?* (Ziele), *Was?* (Produkte), *Wann?* (Meilensteine), *Wer und Wo?* (Verantwortlichkeiten), *Wie?* (Vorgehen), *Wieviel?* (Ressourcen). Alle diese Planelemente sprechen ausdrücklich die betroffenen Gewinnbedingungen der Beteiligten an und regeln, wie diese zu win-win Situationen vereint werden. Dadurch erlaubt der Plan die Identifikation von Abweichungen von win-win Bedingungen. Das Korrigieren des Projektverlaufs im Falle solcher Abweichungen ist eine der empfindlichsten Maßnahmen für die Einhaltung der win-win Situationen. In den meisten Fällen müssen dabei nämlich neue lokale win-win Situationen erzeugt werden, während eine unterlassene oder unüberlegt vorgenommene Korrekturaktion zu win-lose oder gar lose-lose Situationen führt. Dies ist ein Hauptproblem vieler Projekte, die anfangs gut geplant waren.

Planen und der Versuch der Planeinhaltung machen nur dann alle zu Gewinnern, wenn die Pläne realistisch sind. Dies ist der Zweck des Risikomanagements. **Risikomanagement** besteht aus zwei Schritten

1. Risikobestimmung (Risikoentdeckung, -analyse, und -priorisierung).

2. Risikobehandlung (Planung, Durchführung und Überwachung).

Häufig scheitern Projekte daran, daß die Risiken zwar gut analysiert wurden, aber entweder ungünstige Prioritäten gewählt worden sind (z.B. überhaupt keine) oder die geplanten Maßnahmen zur Risikobehandlung gar nicht oder nicht bis zu Ende durchgeführt werden.

8.7.3 Das WinWin-System

Die Forschungsgruppe von Boehm hat ein CSCW-Werkzeug (computer supported cooperative work) entwickelt, das die Anforderungsbestimmung und das Projektmanagement gemäß Theorie W unterstützt. Dieses Werkzeug wird WinWin genannt [BBHL95].

Die Zusammenarbeit der Beteiligten mit Hilfe von WinWin geschieht auf Basis folgender drei Hauptkonzepte:

- **Gewinnbedingung (win-condition, WC)**. Diese beschreibt ein gewünschtes Ziel eines oder mehrerer Beteiligter. Beispiele für WCs sind allgemeine Wünsche wie Zuverlässigkeit und Benutzerfreundlichkeit und konkrete, spezifische Wünsche wie Projektkosten unter 1.2 Millionen Mark oder die Unterstützung einer ganz bestimmten Funktion.

- **Konflikt-/Risiko-/Unsicherheitsbeschreibung (conflict/risk/uncertainty specification, CRU)**. Beschreibt einen Konflikt zwischen WCs oder ein mit einer WC verbundenes Risiko oder eine Unsicherheit. Beispiele wären ein Konflikt zwischen Funktionalitätsanforderungen und Budgetanforderungen, ein Risiko bei der Fähigkeit, eine bestimmte Anforderung zu erfüllen oder eine Unsicherheit bezüglich der Bedeutung einer Anforderung.

- **Übereinkunftspunkt (point of agreement, POA).** Beschreibt eine Menge von Bedingungen, über die sich die Beteiligten geeinigt haben, und die ihre WCs einhalten. Die Menge der POAs definiert die Ziele, die bei der Systemkonstruktion verfolgt werden sollen. Beispiele wären die Vereinbarung auf eine bestimmte Funktionalität zu verzichten, um das Budget einzuhalten, oder eine genaue Definition für die verlangte Zuverlässigkeit.

Die Anforderungen werden also erst durch die WCs beschrieben und später dann wo nötig in den POAs konkretisiert und miteinander in Einklang gebracht.

Das **Vorgehensmodell** von WinWin besteht darin, daß ein **Gleichgewichtszustand (Equilibrium)** zwischen den WCs der Beteiligten hergestellt wird, der darin besteht, daß es erstens keine Konflikte zwischen WCs und POAs gibt, zweitens alle POAs miteinander konsistent sind und es drittens keine unaufgelösten CRUs oder offene POAs gibt. Dieser Zustand ist ganz zu Beginn erfüllt, wenn es überhaupt noch keine WCs gibt und kann jeweils durch die Einführung einer neuen WC gestört werden, wenn diese einen Konflikt mit einer anderen WC hat oder ein Risiko oder eine Unsicherheit aufwirft. Dieses Vorgehensmodell unterstützt ständige Änderungen der Anforderungen, weil es überhaupt keinen Unterschied zwischen einer anfänglichen Anforderungsbestimmung und späteren Anforderungsänderungen macht.

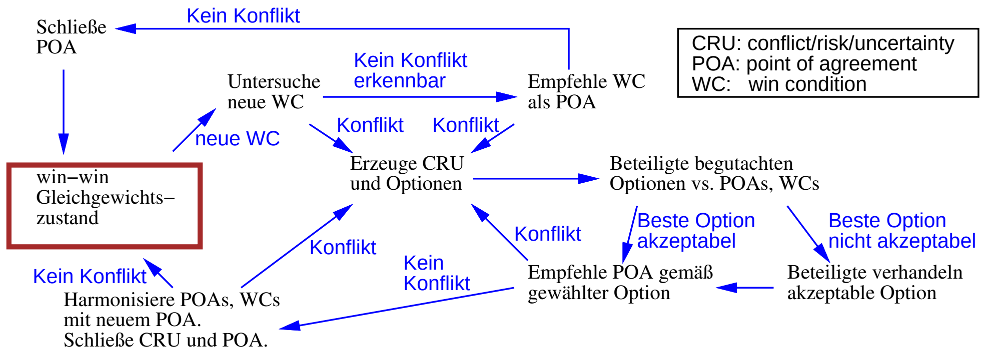

Abbildung 8.2: Zustandsübergangsdiagramm des WinWin-Systems

In Abbildung 8.2 sehen wir die verschiedenen Zustände von WinWin und die Übergänge zwischen diesen. Alle Zustände mit Ausnahme des Gleichgewichtszustands sind mit einer Aktivität verbunden, deren Ergebnis die Übergangsbedingung zum Folgezustand spezifiziert. Die Aktivitäten werden, wo nicht anders vermerkt, vom Analytiker ausgeführt. Der Gleichgewichtszustand wird verlassen, sobald von irgendwoher eine neue WC auftaucht und wird wieder erreicht, wenn diese in eine Übereinkunft (POA) umgesetzt wurde, die nicht mit den übrigen Übereinkünften kollidiert.

Das **Verfahren** bei der Benutzung von WinWin besteht also darin, daß die Beteiligten mit dem System arbeiten, um ihre WCs zu definieren und die WCs der anderen Beteiligten kennenzulernen; alle WCs werden in einem zentralen Speicher gehalten. Bei Konflikten oder Risiken werden von den Analytikern die entsprechenden CRUs erarbeitet. WinWin hilft bei der Identifikation dieser Fälle und beim Zusammentragen der nötigen Information für die CRU. Die CRU dient anschließend dazu, die Verhandlungen zwischen den Beteiligten auf das jeweilige Problem zu konzentrieren. WinWin stellt außerdem Unterstützung für die Verhandlungen zur Verfügung, zum Beispiel Schätzmodelle.

Zwei entscheidende Hilfsmittel in diesem Prozeß sind die Wahlmöglichkeitsübersichten und die Bereichstaxonomie.

Eine **Wahlmöglichkeitsübersicht (option summary)** wird vom Analytiker als Teil einer CRU aufgestellt. Sie führt alle erkennbaren Maßnahmen auf, wie der von der CRU beschriebenen Schwierigkeit begegnet werden könnte. Zu jeder Möglichkeit sind die absehbaren Vorteile und Nachteile mit aufgeführt. Diese Vor- und Nachteile werden von den Beteiligten bei der Verhandlung dazu benutzt abzuschätzen, welche Optionen für sie in Anbetracht ihrer wichtigsten WCs akzeptabel sind oder unter welchen Umständen sie es werden könnten. Dies ermöglicht die schnelle Entfernung von irrelevanten oder inakzeptablen Optionen aus der Diskussion und konzentriert die Verhandlungen schnell auf erfolgversprechende Auswahlmöglichkeiten.

Die **Bereichstaxonomie** liefert eine einheitliche Terminologie für die Formulierung von WCs, CRUs und POAs. Das hat zwei Vorteile. Zum einen ermöglichst die Taxonomie, vermutete Konflikte zwischen WCs schneller als vorhanden oder nicht vorhanden zu klassifizieren, weil sie Verständnisunsicherheit verringert. Zum anderen erlaubt sie eine schnellere Bestimmung der Kandidaten für Konflikte oder Risiken, weil sie eine Auswahl von WCs nach betroffenen Aspekten ermöglicht, indem alle WCs die von ihnen betroffenen Taxonomieelemente aufführen. Die Bereichstaxonomie in WinWin hat drei Teile: Erstens eine konzeptionelle Komponentenzerlegung des gewünschten Systems in Teile und Unterteile, zweitens eine Funktionalitätszerlegung in Funktionen und Unterfunktionen und drittens eine Hierarchie von allgemeinen Attributen des Systems (Produktattribute) oder des Entwicklungsprojekts (Prozeßattribute). Die Bereichstaxonomie von WinWin ist projektspezifisch.

8.7.4 Konfliktidentifikation für Qualitätsanforderungen

Das WinWin-System verringert eine Reihe von Problemen bei der Anforderungsbestimmung, kann aber natürlich keine Wunderlösung für den Kern der Sache sein. In der Terminologie von WinWin bleiben drei schwierige Arbeitsaufgaben bestehen:

1. Die WCs identifizieren.
2. Konflikte zwischen ihnen erkennen.
3. Handlungsoptionen finden, die in win-win Situationen führen.

Es gibt auch einen Ansatz, die Erkennung von Konflikten maschinell zu unterstützen [In96]. Dieser ist in einem wissensbasierten Werkzeug namens QARCC, einer Ergänzung zu WinWin, realisiert und soll in diesem Teilabschnitt vorgestellt werden.

QARCC beschränkt sich auf die Konflikte, die zwischen Qualitätsanforderungen (im Gegensatz zu Funktionalitätsanforderungen) auftreten. Es beruht darauf, die Menge der möglichen Konflikte samt Vorschlägen zu ihrer Lösung in einer **Wissensbasis** abzuspeichern und mit deren Hilfe die Menge der WCs auf Qualitätsanforderungskonflikte zu untersuchen.

Diese Wissensbasis besteht aus drei Teilen, die wir der Reihe nach besprechen: Erstens eine Liste der typischen Qualitätsanforderungen, aufgeschlüsselt nach Art der Beteiligten, zweitens eine Verfeinerung dieser globalen Qualitätsanforderungen in eine größere Zahl detaillierter Anforderungen und drittens eine Zuordnung von sogenannten Qualitätsattributstrategien zu diesen Anforderungen (groben oder verfeinerten).

Die Liste der typischen Qualitätsanforderungen wurde aus der Erfahrung bei der Anwendung von Theorie W hergeleitet. Sie enthält folgende Punkte: Sicherheit/Betriebsgarantien (assurance), Interoperabilität, Benutzbarkeit/Benutzungsfreundlichkeit, Geschwindigkeit, Weiterentwickelbarkeit/Portabilität, Kosten und Zeitplan, Wiederverwendbarkeit.

Die Verfeinerung dieser Anforderungen schlüsselt jede von ihnen in eine Reihe konkreterer Anforderungen auf. Beispielsweise besteht Sicherheit/Betriebsgarantien aus den Einzelpunkten Zuverlässigkeit, Genauigkeit, Korrektheit, Verfügbarkeit, Robustheit, Gefährdungsfreiheit, Datensicherheit, Datenschutz. Diese Auflistung könnte man beliebig verfeinern, sie dient aber nur dazu, Qualitätsanforderungen, die in win-conditions auftreten, erkennen und einer Anforderungsklasse zuordnen zu können.

Der dritte Teil der Wissensbasis ist der entscheidende. Er gibt zu jedem Qualitätsattribut eine Liste von Strategien an, mit denen dieses Attribut erzielt werden kann. Die Liste unterscheidet zwischen **Produktstrategien** (auch **Architekturstrategien** genannt), also anzustrebenden Eigenschaften der Software, und **Prozeßstrategien**, also anzustrebenden Eigenschaften der Vorgehensweise bei ihrer Konstruktion. Produktstrategien für gute Benutzbarkeit und Benutzungsfreundlichkeit sind zum Beispiel: Verteilbarkeit (groupware), fehlervermeidende, konsistente, flexible, benutzeranpaßbare Benutzerschnittstellen, On-line-Hilfe, äußere Modularität, Navigationshilfen, Benutzerprogrammierschnittstellen, Undo-Funktion. Zugehörige Prozeßstrategien sind: Bau von Prototypen, Benutzungsprotokollierung und -modellierung, Benutzerbeteiligung beim Entwurf, Benutzung von Benutzerschnittstellenwerkzeugen.

Die Produktstrategien werden nicht nur einfach aufgelistet, sondern weisen eine eigene Struktur in der Wissensbasis auf; jede besteht aus folgenden Teilen:

- Definition: Beschreibung der Strategie.
- Vorbedingung: Anforderungen für die Anwendbarkeit der Strategie.
- Nachbedingung: Resultate, die mit der Strategie erreicht werden (müssen).
- Effekte: Voraussichtliche Wirkungen der Strategie auf die Qualitätsattribute. Hierbei wird zwischen positiven und negativen Wirkungen unterschieden.
- Optionen: Mögliche Methoden zur Auflösung von Konflikten zwischen Qualitätsanforderungen.

Die Funktionsweise von QARCC zusammen mit WinWin ist nun folgendermaßen:

1. QARCC tritt in Aktion, wenn ein Benutzer eine win-condition eingibt, und extrahiert die im Feld „Taxonomieelemente“ angegebenen Werte.
2. Für jedes Qualitätsattribut unter diesen Werten wird die Liste aller Produktstrategien bestimmt, die für dieses Attribut in Frage kommen.
3. Für jede Strategie wird die Liste aller Qualitätsattribute bestimmt, auf die sie einen negativen Effekt haben kann.
4. In der Menge aller anderen win-conditions wird nach solchen gesucht, die dieses Qualitätsattribut betreffen.
5. Dem Benutzer, der die neue win-condition eingegeben hat, und dem, der eine damit im Konflikt stehende win-condition eingegeben hat, wird eine Nachricht zugestellt, die den Konflikt beschreibt. Diese Nachricht benennt die Liste der betroffenen win-conditions und die betroffene Strategie.

6. Die Vorbedingungs- und Nachbedingungs-Felder der Strategiebeschreibung in der Wissensbasis werden benutzt, um zu erläutern, unter welchen Umständen der Konflikt tatsächlich besteht. Das Optionen-Feld wird benutzt, um den Beteiligten Lösungsvorschläge zu machen, die sie als Grundlage ihrer Verhandlungen zur Auflösung des Anforderungskonflikts benutzen können.

In einer Fallstudie wurden mit Hilfe von QARCC sieben Konflikte zwischen Qualitätsanforderungen aufgedeckt; nur zwei von diesen waren zuvor von den menschlichen Beteiligten der Anforderungsanalyse bemerkt worden. Zusätzlich signalisierte QARCC weitere drei Konflikte, die von den Benutzern als nicht signifikant eingestuft und deshalb ignoriert wurden. Die hybrid automatisch/manuelle Vorgehensweise von QARCC ist ein sinnvoller Ansatz zur Unterstützung der Entdeckung und Behandlung von Anforderungskonflikten. Es ist noch offen, inwieweit die relativ simple Struktur der Wissensbasis auch größeren Projekten gerecht werden kann, oder ob QARCC dann dazu führt, daß die Benutzer mit Meldungen über potentielle Konflikte überschüttet werden.

8.8 Integration verschiedener Sichten

Neben der Aufdeckung von Interessenskonflikten, also Konflikten zwischen Wünschen, ist das Aufdecken von Beschreibungsinkonsistenzen von Interesse, also inhaltlichen Konflikten zwischen verschiedenen Beschreibungen des Ist-Zustands oder des Soll-Zustands. Diese können Interessenkonflikte sein, sind es oftmals aber nicht. Zur Unterstützung dieser Tätigkeit wäre es nützlich, wenn alle Beteiligten den Zustand aus der Perspektive (Sicht) schildern könnten, die ihnen am geläufigsten und die für sie am relevantesten ist. Es wird dann maschinelle Unterstützung dafür gebraucht, diese verschiedenen Sichten auf eine gemeinsame Sicht abzubilden, in der nach Widersprüchen zwischen den einzelnen Beschreibungen gesucht werden kann.

Eine Möglichkeit hierfür sind sogenannte Metamodellierungssysteme, von denen eines (und seine Anwendung) in diesem Abschnitt kurz vorgestellt werden soll.

8.8.1 Metamodellierung

Die abstrakte Beschreibung eines Zustandes ist ein **Modell** dieses Zustands. Informatiker erfinden zur Aufstellung und Manipulation solcher Beschreibungen typischerweise eine **Modellierungssprache**. Eine solche Sprache ist ihrerseits ein Modell, das aber zur Beschreibung von Modellen, nicht von Zuständen der wirklichen Welt dient, also ein **Metamodell**. Solche Sprachen erlegen ihren Benutzern normalerweise eine bestimmte Weltsicht auf, indem sie einen gewissen Beschreibungsrahmen vorgeben. Das ist einerseits praktisch, weil es die Modellierungsarbeit vereinfacht, andererseits aber auch unerwünscht, weil es die Flexibilität beim Ausdrücken der Zustände beschränkt.

Deshalb kann man eine Modellierungssprache vorsehen, mit der nicht direkt der Zustand beschrieben werden soll, sondern stattdessen die Modellierungssprache, in der dann erst der Zustand ausgedrückt wird. Eine solche Sprache stellt also ein Modell eines Metamodells dar, ein **Metametamodell (M2-Modell)**. Sie heißt deshalb **Metamodellierungssprache**.

Ist eine solche Metamodellierungssprache flexibel genug, so können mit ihr auch beliebige höhere Meta-Ebenen betreten werden. Insbesondere lassen sich dann die Beziehungen zwischen Metamodellen beschreiben, was die Grundlage für die Integration verschiedener Sichten bei der Anforderungsbestimmung bildet, denn diese Sichten (genauer: die Sichtweisen) sind Metamodelle.

8.8.2 Beispiel: Telos

Ein Beispiel für eine Metamodellierungssprache ist Telos, das auf nur zwei Entitätstypen und drei Beziehungstypen basiert:

- Entitätstyp 1 sind **Objekte**, die über einen Namen identifiziert werden.
- Entitätstyp 2 sind **Zusicherungen**, also Formeln.
- Beziehungstyp 1 sind **Attribute**. Ein Objekt kann Attribut eines anderen Objekts sein.
- Beziehungstyp 2 ist die **Klassifikation** mit der „in"–Beziehung. Ein Objekt (Instanz oder Klasse) kann Mitglied eines anderen Objekts (Klasse) sein.
- Beziehungstyp 3 ist die **Regelbeziehung**. Eine Zusicherung ist eine Regel für ein Objekt. Hier gibt es zwei Fälle, nämlich Deduktionsregeln zur Ableitung von neuen Zusicherungen und Zwänge zur Beschränkung der Eingabe (oder Herleitung) von Daten.

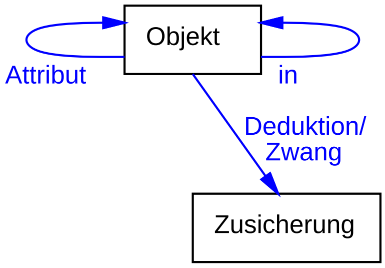

Abbildung 8.3: Entitätstypen und Beziehungstypen von Telos

Die Klassifikation von Telos ist sehr flexibel. Sie erlaubt nicht nur die Beschreibung von Klassen in einer Klassenhierarchie, sondern auch die Beschreibung von Instanzen, Metaklassen, Metametaklassen, etc. Außerdem können mehrere gleichzeitige Klassifikationen für dieselben Objekte existieren, um beispielsweise mehrere Abstraktionen derselben Welt zu beschreiben. Die Deduktion erlaubt das Ableiten von Klassenzusicherungen („Objekt o ist Mitglied der Klasse c") und von Attributwerten („Attribut a von Objekt o hat Wert x"). Damit entspricht die Ausdruckskraft von Telos der einer deduktiven relationalen Datenbank.

Beispiele für Telos-Deklarationen:

```
Department in Class with
  attribute
    head: Manager
end

Employee in Class with
  attribute
    name: String;
    salary: Integer;
    dept: Department;
    boss: Manager
end
```

```
Manager in Class isA Employee
end

Employee with
  rule
    BossRule:$
      forall e/Employee m/Manager
        (exists d/Department (e dept d) and (d head m)) ==>
        (e boss m) $
```

Dies deklariert eine Klasse `Department` mit einem Attribut `head` aus der Klasse `Manager` und analog eine Klasse `Employee` mit diversen Attributen. Drittens eine Instanz `Manager` zur Klasse `Employee`; somit ist `Manager` sowohl eine Klasse als auch eine Instanz einer Klasse und folglich `Employee` eine Metaklasse. Schließlich gibt die letzte Deklaration eine Deduktionsregel an, die beschreibt, wann ein Angestellter e den Manager m zum Chef hat: dann nämlich, wenn `e` zur Abteilung `d` gehört und `m` der Leiter von `d` ist. (Diese Beispiele haben übrigens nichts mit dem weiter unten besprochenen Projektbeispiel zu tun.)

So einfach wie Telos erscheinen mag: Aufgrund der Deduktionsregeln und der mehrfachen Klassifikation ist es sehr mächtig. In der gegenwärtigen Telos-Implementation werden die Deklarationen mittels partieller Auswertung in Prolog umgesetzt, so daß der Prolog-Interpretierer Teile der eigentlichen Deduktionen erledigt.

8.8.3 Vorgehen zur Sichtenintegration

Das grundsätzliche Vorgehen zur Integration verschiedener Systemsichten mit Hilfe eines Metamodellierungssystems sieht etwa folgendermaßen aus:

1. Definiere ein M2-Modell, das eine Terminologie bereitstellt, mit der sich alle verschiedenen Sichten beschreiben lassen.

2. Definiere alle gewünschten Sichten anhand des M2-Modells.

3. Ermittle Ist-Zustand und Soll-Zustand (Anforderungen) in Interviews mit den diversen Beteiligten. Verwende zur Beschreibung jeweils diejenige Sicht, die dem Beteiligten am angenehmsten ist. Es ist erwünscht, daß sich die Beschreibungen stark überlappen.

4. Definiere Abbildungen von einer Sicht in eine andere anhand des M2-Modells. Nun können die Beschreibungen eines Beteiligten auch in anderen Sichten betrachtet werden als derjenigen, in der sie abgegeben wurden.

Der Wechsel zwischen Sichten kann nun benutzt werden, um in einem Interview mit einem Beteiligten auf Beschreibungen eines anderen Beteiligten zurückzugreifen, der eine andere Sichtweise benutzt hat, um sie abzugeben. Außerdem bietet die Integration der Sichten eine gute Grundlage für eine Analyse der Beschreibungen (siehe den nächsten Abschnitt).

Einige Bemerkungen zum M2-Modell sind zu machen:

- Seine Aufstellung stellt in der praktischen Anwendung einen erheblichen Aufwand dar.
- Es entscheidet darüber, wie flexibel die Modellierungssprachen werden können, die sich integrieren lassen.

- M2-Modelle können in späteren Projekten unterschiedlich gut wiederverwendet werden. Ein M2-Modell kann schwerlich sowohl so präzise sein, daß es wesentliche Anforderungskonflikte zu erkennen erlaubt, als auch zugleich so allgemein, daß es in allen möglichen Anwendungen wiederbenutzt werden kann. Sehr wohl gibt es aber M2-Modelle, die in einer Reihe von ähnlich gelagerten Projekten mit nur geringen Änderungen benutzt werden können.

- Ein gutes M2-Modell muß einfach sein; es darf nur sehr wenige (z.B. unter zehn) Grundkonzepte aufweisen.

- Dann wird es bei der Anforderungsbestimmung zu einem wichtigen Kommunikationshilfsmittel, weil es die Ausdrucksweise vereinheitlicht und damit Mißverständnisse vermeiden hilft.

8.8.4 Anforderungsanalyse durch Sichtenintegration

Wie schon gesagt, bemüht man sich, viel Überlappung zwischen den Sichten zu haben und viel redundante Information von den Beteiligten zu erhalten. Solche Redundanz bietet nämlich die Gelegenheit zur Entdeckung von Mißverständnissen oder Konflikten zwischen den Beteiligten und von Fehlern beim Anforderungsbestimmungsprozeß. Die Redundanz wird ausgenutzt, indem man die miteinander integrierten Sichten auf Konsistenz prüft.

Dazu definiert man anhand des M2-Modells sogenannte **Abfrageklassen (query classes)**, von denen jede genau eine Klasse zu erwartender Inkonsistenzen beschreibt. Das Metamodellierungssystem kann die Deduktionen berechnen, die nötig sind, um alle Instanzen dieser Art von Inkonsistenz aufzuspüren. Jede Instanz muß dann einzeln untersucht und auf einen Modellierungsfehler oder einen Konflikt in den Anforderungen zurückgeführt werden.

Neben Inkonsistenzen lassen sich auf die gleiche Weise auch andere Informationen aus den zusammengeführten Sichten automatisch extrahieren. Beispiele für dieses Vorgehen werden wir in folgenden Abschnitt sehen.

8.8.5 Beispiel: PFR mit ConceptBase

Die obenerwähnte Metamodellierungssprache Telos ist Bestandteil eines Metamodellierungssystems namens ConceptBase. Dieses System ist ein deduktives Datenbanksystem mit Telos als Kern und einer Benutzerschnittstelle, die interaktiv Aufbau und Manipulation von Metamodellen und Modellen unterstützt.

In diesem Abschnitt betrachten wir eine Anwendung [NJJ+96] von ConceptBase zur Bestimmung und Analyse des Ist-Zustands in einem Unternehmen, das seine Arbeitsabläufe optimieren möchte (workflow analysis, business process reengineering). Die Analyse des Ist-Zustands, die den Ausgangspunkt dieser Optimierung bildet, wird nach der PFR-Methode durchgeführt. PFR (analysis of presence and future requirements) ist eine Analysetechnik, die wie folgt abläuft:

1. In einer zweitägigen Sitzung entwerfen die Beteiligten eine anfängliche gemeinsame Sicht: Grobe Analyse der gegenwärtigen Geschäftsprozesse, Analyse der Ziele hinter diesen Abläufen, Beschreibung neuer Ziele, grobe Skizzierung besserer Geschäftsprozesse mit Verweisen von Prozeßelementen auf die Ziele, Identifikation zentraler Prozeßelemente.

2. Die als zentral erkannten Elemente werden dann mit verschiedenen Mitteln genau untersucht: in Einzelinterviews, mit Arbeitsablaufstudien und mit Studien von Dokumentenstrukturen und -inhalten. Dabei werden mehrere verschiedene Perspektiven (die Quellsichten) benutzt.

3. Die Informationen aus den verschiedenen Sichten werden zusammengetragen und analysiert. Dabei auftretende Lücken und Widersprüche werden in zusätzlichen Interviews aufgeklärt, wobei lokale Zustimmung der Beteiligten angestrebt wird.

4. In einer zweiten Sitzung werden die Ergebnisse allen Beteiligten mit dem Ziel verständlich gemacht, globale Zustimmung aller Beteiligten zu erhalten. In diesem Prozeß werden zur Darstellung globaler Sachverhalte weitere Sichten (die Zielsichten) benutzt.

ConceptBase kann als Werkzeug zur Unterstützung von PFR benutzt werden. Es erlaubt, die Information aus den verschiedenen Quellsichten in eine einheitliche interne Form zu bringen, dann anhand dieser einheitlichen Form mit Rechnerhilfe Lücken und Widersprüche aufzudecken und schließlich automatisch Darstellungen in beliebigen Zielsichten zu erzeugen.

Ausgangspunkt aller dieser Möglichkeiten ist die einheitliche interne Darstellung, die durch ein M2-Modell beschrieben wird. Für das Beispiel wurde das M2-Modell aufgestellt, das in Abbildung 8.4 schematisch skizziert ist.

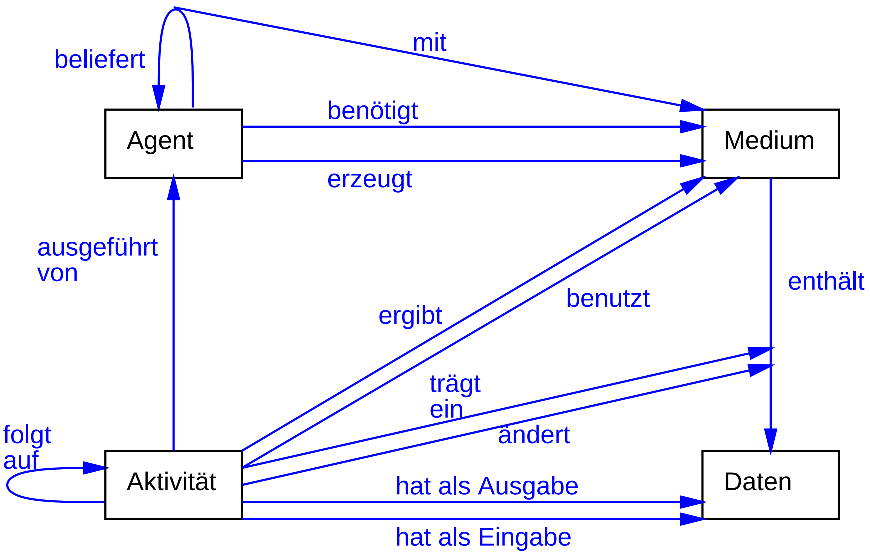

Abbildung 8.4: Beispiel für ein M2-Modell

Es enthält nur vier Entitätstypen, nämlich die Agenten, die in der Unternehmung Prozesse durchführen, die Aktivitäten, also die elementaren Prozesse selbst, die Medien, die die dabei erzeugte und verarbeitete Information tragen und viertens die Daten, also diese Informationen selbst. Hinzu kommen 12 Beziehungstypen; beispielsweise im Fall des Agenten seine Eigenschaft, der Ausführer einer Aktivität zu sein (von unten kommender Pfeil), Erzeuger oder Benutzer von Medien zu sein (nach rechts weggehende Pfeile) und schließlich einen anderen Agenten mit einem Medium zu versorgen (dreistellige Beziehung, runder Pfeil und von diesem abgehender Pfeil).

Dieses Modell ist hinreichend einfach, um als M2-Modell brauchbar zu sein. Die Terminologie ist so simpel, daß dieses M2-Modell leicht Grundlage einer gemeinsamen Sprachregelung werden kann. Die Aufstellung dieses Modells ist alles andere als einfach. Die Unterscheidung zwischen

Daten und Medien wurde beispielsweise aufgrund der Beobachtung gemacht, daß oftmals Unternehmensprozesse daran leiden, daß Informationen mehrfach von einem Datenträger auf einen anderen wechseln (z.B. aus einem Rechner auf ein Blatt Papier, dann über ein Fax auf ein anderes Blatt Papier und von dort von Hand wieder in einen Rechner), oder daß Daten überflüssigerweise mehrfach einer Aktivität zugänglich gemacht werden. Die Unterscheidung wurde als das einfachste und beste Hilfsmittel erkannt, um diese Situationen modellieren zu können.

Auf Basis dieses Metamodells können nun verschiedene Sichten von Unternehmensprozessen beschrieben werden. Im vorliegenden Fall wurden unter anderem folgende drei Sichten definiert: Die **Aktivitätsfolgensicht** aus Abbildung 8.5 stellt Handlungssequenzen in den Mittelpunkt der Betrachtung und beschreibt darüber hinaus, wer jeweils eine Handlung ausführt und welche Information dabei benutzt oder erzeugt wird.

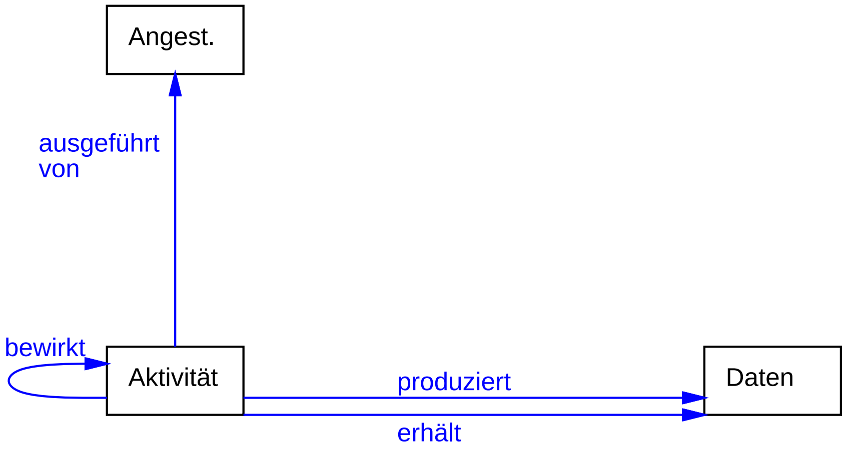

Abbildung 8.5: Modell der Aktivitätsfolgensicht

Die **Informationsaustauschsicht** aus Abbildung 8.6 beschreibt lediglich, welche Untereinheiten des Unternehmens an welche anderen Untereinheiten welche Medien (Datenträger) liefern. Diese Sicht beschreibt also isoliert den physischen Kommunikationsfluß.

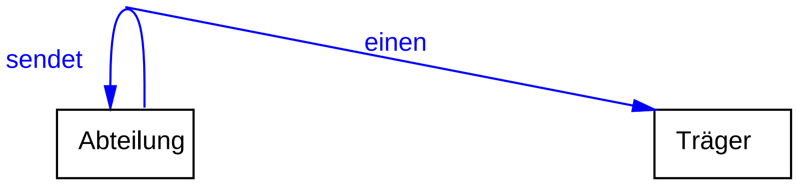

Abbildung 8.6: Modell der Informationsaustauschsicht

Die **Dokumentenstruktursicht** aus Abbildung 8.7 ist noch einfacher und beschreibt nur, welche Dokumente (Medien) welche Information enthalten. Diese Sicht legt also die Datenstrukturen im Unternehmen und ihre physische Realisierung offen.

Beachte, daß die obigen Abbildungen nur die **logische Struktur** einer Sicht wiedergeben. Die bei den Sitzungen eingesetzte **Notation** wird davon unabhängig definiert. Für alle drei Sichten steht neben einer textuellen Darstellung auch eine jeweils angemessene graphische Notation zur Verfügung.

Es werden nun von den Beteiligten Informationen über die Geschäftsprozesse eingeholt, wobei sich die Informationen von verschiedenen Beteiligten überlappen. Die bewußte Mehrfacherhebung von Informationen, oft mit Widersprüchen, ist beabsichtigt, denn sie stellt eine wichti-

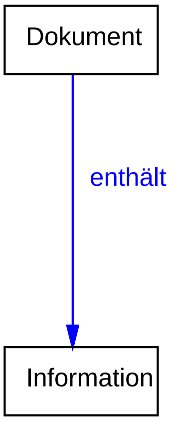

Abbildung 8.7: Modell der Dokumentenstruktursicht

ge Grundlage für die Analyse der Anforderungen dar. Die Redundanz in den Informationen ermöglicht die Aufdeckung zahlreicher Fehler und Konflikte.

Diese Analyse kann bei der Verwendung von ConceptBase wie schon erwähnt mit Hilfe von Abfrageklassen erfolgen. Eine solche Abfrageklasse sucht gezielt nach einer bestimmten Art von Widersprüchen. ConceptBase gibt über jede Instanz der Abfrageklasse, die in der vorhandenen Informationsmenge gefunden wird, genug Information aus, um das Problem identifizieren zu können, so daß man es dann korrigieren kann.

Hier ein Beispiel: Wir suchen nach allen Medien, die an einen Agenten geliefert werden, obwohl dieser keine Aktivität ausführt, die die auf dem Medium befindlichen Daten benutzt.

```
QueryClass UnbenutztesMedium isA Medium with
  attribute
    nichtbenutzer : Agent
  constraint
    c:$ exists belieferung
          (belieferung in Agent!beliefert) and
          (belieferung an nichtbenutzer) and
          (belieferung mit this) and
          not exists aktion, info
            (aktion ausgefuehrt_von nichtbenutzer) and
            (this enthaelt info) and
            ((aktion hat_als_eingabe info) or
             (aktion hat_als_ausgabe info)) $
end
```

Eine solche Anfrageklasse liefert als Ergebnis eine Menge von Paaren aus Medium `UnbenutztesMedium` und Agent `nichtbenutzer`. Jedes solche Paar kann verschiedene Bedeutungen haben:

- Fehler: Eine vorhandene Eingabe- oder Ausgabebeziehung für eine Aktivität ist bei der Analyse nicht erfaßt worden.
- Sichtenabweichung: Verschiedene Beteiligte (oftmals Management einerseits und Sachbearbeiter andererseits) sehen den Prozeß unterschiedlich.

- Vermeidbar: Die Situation ist tatsächlich so, daß der Agent das Medium überhaupt nicht benötigt. Der Prozeß kann optimiert werden.
- Korrekt: Die Aktivität, die das Medium empfängt, benötigt tatsächlich keine Information von diesem Medium, ist aber dennoch sinnvoll (z.B. Träger sammeln und gebündelt weitergeben).

Fälle von „Fehler“ und „Sichtenabweichung“ zeigen Probleme in der Anforderungsbestimmung an, die beseitigt werden müssen und entsprechen genau der Intention der Abfrage. „Vermeidbar“ zeigt quasi nebenbei eine Optimierungsmöglichkeit an. Fälle von „Korrekt“ sind unerwünschte Ausgaben, die deshalb auftreten, weil die Modellierung nicht alle Aspekte berücksichtigt.

Solche Abfragen können einen großen Teil aller Schwierigkeiten aufdecken, der in den ermittelten Informationen steckt. Dabei muß die Bereinigung solcher Schwierigkeiten natürlich nach wie vor von Hand gemacht werden, jedoch fällt nur noch wenig Handarbeit für die Entdeckung an, was die Sache erstens beschleunigt (und damit verbilligt) und zweitens sehr viel zuverlässiger macht. Die Abfragen lassen sich überdies meist wiederverwenden, wann immer das gleiche M2-Modell benutzt wird.

Abfragen können auch eingesetzt werden, um gezielt nach Verbesserungsmöglichkeiten im Prozeß zu suchen. Beispielsweise sucht die folgende Abfrageklasse nach Agenten, die ein Dokument erhalten, das nur solche Daten enthält, die sie auch noch auf anderen Medien erhalten.

```
QueryClass MehrfachVersorgter isA Agent with
  attribute
    redundant : Medium
  constraint
    c:$ exists belieferung
          (belieferung in Agent!beliefert) and
          (belieferung an this) and
          (belieferung mit redundant) and
          forall info
            (redundant enthaelt info) ==>
              exists zweitmedium
                (zweitmedium <> redundant) and
                (zweitmedium enthaelt info) $
end
```

Wiederum lassen sich die Paare aus Agent `Mehrfachversorgter` und Medium `redundant` auf mehrerlei Weise erklären:

- Unnötig: Die Versorgung wird tatsächlich als redundant angesehen und kann optimiert werden.
- Auswahl: Der Agent soll frei zwischen den Medien wählen können. Es fragt sich, ob und wann das ein sinnvoller Aufwand ist.
- Prüfung: Der Agent soll eine Gleichheitsprüfung der Daten auf den verschiedenen Medien vornehmen. Das könnte bei geeigneten Medien automatisiert werden.
- Dopplung: Der Agent soll dieselbe Tätigkeit getrennt für die gleiche Information auf verschiedenen Trägern durchführen.

Fälle von „Unnötig“ zeigen eine Verbesserungsmöglichkeit an. „Auswahl“ und „Prüfung“ zeigen an, daß Verbesserungsmöglichkeiten vorliegen, wenn gewisse Entscheidungen geeignet getroffen werden. Fälle von „Dopplung“ sind unerwünschte Ausgaben, die darauf zurückzuführen sind, daß das M2-Modell diesen Fall nicht explizit darstellen kann.

In dem hier skizzierten Beispielprojekt wurden insgesamt etwa 80 solcher Anfrageklassen benutzt, um Inkonsistenzen in den Daten, die zunächst bei der Erfassung absichtlich zugelassen wurden, konstruktiv im Analyseprozeß einzusetzen. Davon zielten 10 auf sehr spezielle Fragestellungen ab, die übrigen 70 sind so allgemein, daß sie in gleicher oder ähnlicher Form auch in anderen Projekten mit ähnlicher Fragestellung Verwendung finden können.

Zusammenfassend kann man sagen, daß ein Metamodellierungssystem von der Ausdruckskraft von ConceptBase ein leistungsfähiges Werkzeug bei der Anforderungsbestimmung darstellt. Für umfangreiche Mengen von Anforderungen bewirkt es eine enorme Arbeitsersparnis bei der Analyse der Anforderungen und erhöht die Zuverlässigkeit, mit der Probleme entdeckt werden. Allerdings muß zum Erzielen dieser Vorteile eine geeignete Metamodellierung gefunden werden, was nicht einfach ist. Schließlich kann der Deduktionsmechanismus eines solchen Systems bei größeren Datenmengen leicht zum Leistungsengpaß werden, weil nach jeder Korrektur alle oder zumindest viele der zahlreichen Suchanfragen erneut durchlaufen werden müssen.

8.9 Weitergehende Information

Ein sehr gut lesbares und außerordentlich empfehlenswertes Lehrbuch zum Thema Anforderungsbestimmung ist [RR96]. Die Autoren benutzen ein einziges, sehr umfangreiches, reales Beispiel aus ihrer eigenen Arbeit und schreiben einen sehr menschlichen und lebendigen Stil. Es enthält nebenbei auch hervorragend aufbereitete Anleitungen für Techniken wie Daten- und Datenflußmodellierung etc.

[All95] beschreibt seine Erfahrungen und Methoden beim Versuch, bessere Anforderungsbestimmung „von unten her“ in eine Organisation einzuführen. Er nennt die von ihm empfohlene Vorgehensweise die „Wasserfolter“. Einen anderen, ganz andersartigen Einblick in die (oder sagen wir besser: eine) industrielle Praxis der Anforderungsbestimmung liefert [Kol92].

IEEE Software hat im März 1994 und im März 1996 jeweils ein Heft mit dem Themenschwerpunkt Anforderungsbestimmung herausgebracht. In beiden Fällen sind vor allem die Einleitungen (nur je drei Seiten) lesenswert, da sie einen guten Überblick über aktuelle Schwerpunkte auf diesem Gebiet liefern. Communications of the ACM hatte einen ähnlichen solchen Schwerpunkt im Heft vom Mai 1995.

Lesenswert sind der Artikel über das „In-die-Lehre-Gehen“ [BH95], die Fallstudie im Artikel über Theorie W [BR89] und die Artikel über das WinWin-System [BBHL95, In96]. Information über WinWin findet sich auch bei http://sunset.usc.edu/WinWin/winwin.html.

Das ConceptBase-System zur Metamodellierung sowie weitere Information darüber ist verfügbar unter
http://www-i5.informatik.rwth-aachen.de/CBdoc/cbflyer.html.
Die obenerwähnte Anwendung ist in [NJJ+96] beschrieben. Die Seite
http://www.ifi.unizh.ch:80/groups/req/rerg_projects.html
beschäftigt sich ebenfalls mit dem Problem der Abbildung zwischen benutzerverständlichen externen Sichten und formalen, genau definierten internen Darstellungen.

Abschnitt 9

Technisches Schreiben

Schreiben ist schwierig.
[...] Es ist schwierig,
weil es eine Reihe von Aktivitäten verlangt,
die gemeinhin als Denken bezeichnet werden.
Ernst Jacobi

In diesem Anhang geht es um das Schreiben technischer Dokumente, seien es nun Software-Entwurfsdokumente, Benutzerhandbücher, wissenschaftliche Artikel oder sonst etwas derartiges.

Fertigkeiten im technischen Schreiben werden oftmals nicht als ein wesentliches Qualifikationsmerkmal von Softwaretechnikern angesehen, obwohl die Wirklichkeit dies sehr nahelegen würde: hohe Kosten entstehen bei der Wartung von Software sehr oft dadurch, daß die interne Dokumentation zu undurchsichtig organisiert oder inhaltlich zu schwer verständlich ist. Da interne Dokumentation fast stets von den Softwaretechnikern anzufertigen ist, nimmt also das technische Schreiben sogar einen sehr hohen Stellenwert ein.

Ferner ist es natürlich auch für das Schreiben von Benutzerdokumentation sehr nützlich, auch wenn diese Aufgabe zunehmend Spezialisten übertragen wird. Und drittens sind ganz ähnliche Kenntnisse und Fertigkeiten auch für das Schreiben guter Memos und das Entwerfen guter Präsentationen vonnöten. Es lohnt sich also, sich mit diesem Thema auseinanderzusetzen. Dabei dürfte klar sein, daß ein rein theoretisches Lernen hier vollkommen unmöglich ist; die Devise heißt üben, üben, üben. Das Schreiben ist ein kreativer Akt und dementsprechend befriedigend; deshalb kann solches Üben eine Menge Spaß machen. Jeder, der selber gut schreiben kann, ist als Kritiker zu gebrauchen.

In diesem Kapitel werden die Begriffe Dokument und Text weitgehend synonym gebraucht und stehen für das Ergebnis des technischen Schreibens, das außer Text auch Bilder etc. umfassen kann.

9.1 Anforderungen

Was unterscheidet denn nun das technische Schreiben von, sagen wir mal, dem Schreiben eines Romans oder eines Zeitungsartikels? Gar nicht soviel wie man meinen könnte, wie wir sehen werden.

Für die Qualität eines technischen Textes gibt es eine einzige zentrale Meßlatte: die **effektive Verständlichkeit**. Die wichtigste Frage ist also „Wieviel von der erwünschten Aussage kommt beim Leser *im Endeffekt* tatsächlich an?“. Diese Verständlichkeit entsteht aus verschiedenen Beiträgen, die in den folgenden Abschnitten kurz diskutiert werden:

9.1.1 Inhalt

Zuallererst muß natürlich der Inhalt des Dokumentes selbst verständlich sein. Die allererste Voraussetzung dafür ist, daß der Autor das Thema des Textes wirklich verstanden hat; es ist aber durchaus nicht schlimm, wenn das erst im Verlauf des Schreibens eintritt. Die zweite Voraussetzung ist, daß die Verständlichkeit nicht nur irgendwie und im Prinzip gegeben ist, sondern tatsächlich — und zwar für das Zielpublikum, das den Text lesen sollte.

Damit kommen wir zum Problem von **Abdeckung und Detaillierungsgrad**: Wo muß ich mit dem Erklären anfangen, was darf ich voraussetzen (Abdeckung) und wie genau muß eine Erklärung sein (Detaillierungsgrad)? Die wichtigste Erkenntnis lautet: Das kann man nie ganz richtig machen. Für eine gute Auslegung muß man vor allem sein Publikum und dessen Vorwissen gut kennen. Ist das nicht erreichbar, muß man versuchen, möglichst viele Zielgruppen zugleich anzusprechen, indem man im Text klar kennzeichnet, welche Teile für welche Breite und Tiefe von Informationsbedürfnis gelesen werden müssen — und natürlich diese Teile überhaupt erstmal voneinander trennt.

Ein zusätzliches Problem ist häufig der **Platzmangel**, der zu überkurzen Erklärungen und Formulierungen verleitet. Hier gilt meist die Regel „Weniger ist mehr“: wenige gut erläuterte Fakten sind allemal nützlicher als viele unverständliche. Die Wahl von Abdeckung und Detaillierungsgrad ist Teil eines allgemeineren Problems, der Adäquatheit, das unten angesprochen wird.

9.1.2 Textorganisation

Eine Schlüsselstellung für die Verständlichkeit nimmt eine geschickte Organisation des Dokuments ein. Das Ziel besteht darin, eine schnelle Erfassung des Wesentlichen auch ohne komplettes Lesen des Dokuments zu ermöglichen. Es soll also die Möglichkeit zum effektiven **„Querlesen“** geschaffen werden. Dies gilt nicht nur für das Dokument im ganzen, sondern ebenso für jeden einzelnen seiner Abschnitte.

Querlesen ist deshalb wichtig, weil ein technischer Text fast immer erheblich mehr Information enthält, als der überwiegende Teil der Leserschaft benötigt. Der Leitgedanke ist folgender: Das Dokument soll einen Service für den Leser erbringen, nämlich ihm Informationen zu übermitteln, die er benötigt. Dieser Service ist umso besser erbracht, je schneller der Leser oder die Leserin diese *und nur diese* Informationen aus dem Gesamtdokument extrahieren kann.

Das Hauptmittel einer guten Textorganisation ist eine **logische Struktur und Abfolge der Teile**. Logisch bedeutet dabei, daß es eine einfache Begründung oder Herleitung für gerade die gewählte Struktur gibt, die der Leser schnell nachvollziehen kann. Wenn der Leser ebendies auch tut, dann ist die Textorganisation erfolgreich. Das beste Mittel, den Leser zum Nachvollziehen der Struktur zu bringen, besteht darin, die Begründung oder Erläuterung der Struktur in der Einleitung des Textes oder Textabschnitts explizit in knapper Form aufzuschreiben.

Weitere **technische Mittel** zu einer guten Textorganisation sind: Treffende Überschriften, Inhaltsverzeichnisse, Schlagwortindex und andere Verzeichnisse, sowie Querverweise.

Im Inneren eines Textabschnitts lautet die wichtigste Regel für eine gute Textorganisation: **Nur ein Gedanke pro Satz!** Wenn diese Gedanken dann eine nachvollziehbar logische Abfolge bilden, ist die Verständlichkeit des Textes schon fast gesichert.

9.1.3 Interessantheit

Die Verständlichkeit eines Dokuments ist gleich Null, wenn niemand es liest. Niemand liest gern uninteressantes Material. Als Konsequenz aus diesen beiden Aussagen ergibt sich unmittelbar, daß ein technischer Text interessant sein muß — naja: sein sollte.

Und wie schafft man das? Dazu gibt es eine Fülle von Regeln, zum Beispiel aus dem Journalismus: Das Wichtigste zuerst; schreibe verständliches Deutsch (Ha, ein Ringschluß!); liefere etwas Neues; und so weiter [Sch82]. Leider reichen alle diese Regeln bei technischen Texten oftmals nicht aus; selbst wenn sie alle eingehalten werden, kann das Ergebnis steril und langweilig sein.

Den Unterschied zwischen *Gähn!* und *Staun!* machen häufig die folgenden zwei kleinen Tips:

1. Habe ein Anliegen! Erzähle eine Geschichte!
2. Lieber „Ich" als niemand.

Die erste Regel betrifft vor allem die **innere Einstellung**. Wer als Autor oder Autorin unbeteiligt ist, sich selber nicht für seinen Text interessiert, wird es kaum schaffen, ein interessantes Dokument hervorzubringen. Wer dagegen aktiv „was rüberbringen" möchte, vermeidet uninteressantes Geschwafel fast automatisch. Ein geschichtenhafter Text ist selbst bei abstrakten Sachverhalten oft zu erzielen, wenn man nicht nur abstrakt schreibt, sondern wo immer möglich konkrete Beispiele für abstrakte Aussagen angibt.

Die zweite Regel ist ein klares Votum für die **Benutzung der ersten Person**. Die meisten Leute schreiben in wissenschaftlichen Arbeiten und ebenso in technischen Dokumenten aller Art entweder ganz unpersönlich, oder sie verschanzen sich hinter der scheinbar neutralen dritten Person und schreiben „die Autoren", wo sie „wir" meinen oder „der Autor", wo sie „ich" meinen.[1] Ich plädiere dafür, wirklich „ich" oder „wir" zu schreiben, wenn man das meint. Wenn man das tut, fällt die Befolgung der ersten Regel nämlich leichter und der Leser merkt viel eher, daß da ein menschliches Wesen schreibt. Das tut der Interessantheit gut, weil der Text an Lebendigkeit gewinnt.

Und falls sich irgendwo die Gelegenheit ergibt, ein bißchen Humor in einem Sachtext unterzubringen, wird das der Interessantheit nicht schaden — aber bitte nur, wenn es auch hinpaßt.

9.1.4 Adäquatheit

Selbst wenn ein Dokument einen korrekten, verständlichen Inhalt hat, gut organisiert und interessant geschrieben ist, stellt sich noch eine wichtige Frage: Ist er auch angemessen?

Jedes technische Dokument sollte einen möglichst genau bestimmten **Zweck** haben und muß sich fragen lassen, ob es diesen effektiv und effizient verfolgt. Dagegen gibt es drei Haupthindernisse:

1. Ein Dokument kann zu lang sein. Beispiel: ein „Einführungshandbuch" von 400 Seiten.
2. Ein Dokument kann falsch ausgerichtete inhaltliche Schwerpunkte aufweisen. Beispiele: Ein Entwurfsdokument für eine Bibliothek, das zu sehr auf die Benutzung der Objekte eingeht anstatt auf die Entwurfsentscheidungen, oder umgekehrt ein Benutzerhandbuch, das zu sehr auf die interne Implementation der Objekte eingeht anstatt auf die Benutzung.

[1] Eine andere alberne Variante ist der pluralis majestatis (oder sollte es pluralis academicus heißen?): „wir" statt „ich". Das wirkt mindestens steif, schlimmstenfalls sogar peinlich. Davon zu unterscheiden ist jedoch ein wünschenswerter Einsatz des „wir", bei dem nämlich die Autoren (auch, wenn es nur einer ist) den Leser oder die Leserin gewissermaßen bei der Hand nimmt und mit ihm oder ihr gemeinsam etwas untersucht.

3. Ein Dokument kann eine unangemessene (obwohl gute) Organisation aufweisen. Beispiele: Referenzhandbücher ohne Stichwortverzeichnis, mit Zwischenüberschriften, die erst aus dem Zusammenhang verständlich werden, oder mit zu knappem Inhaltsverzeichnis.

Folglich muß der Zweck eines Dokuments beim Schreiben als zentrale Leitlinie jederzeit im Auge behalten werden.

Über diese Hauptanforderungen (Inhalt, Organisation, Interessantheit, Adäquatheit) hinaus sind folgende Unterpunkte von Belang:

9.1.5 Korrektheit

Einer der größten Feinde von Verständlichkeit sind Fehler. Dies ist für den **Inhalt** offensichtlich: Wenn die Fehler innere Widersprüche hervorrufen, geht die Verständlichkeit verloren, andernfalls wird beim Leser ein falsches Verständnis erzeugt. Aber auch auf der Ebene des Sprachstils ist Korrektheit notwendig, da sprachliche Fehler die Aufmerksamkeit vieler Leser stark ablenken und dadurch effektiv die Interessantheit des Textes verschlechtern.

Zur Rolle der Korrektheit in **Rechtschreibung, Zeichensetzung und Grammatik** kann man drei Haupttypen von Haltungen unterscheiden: Die *Vorschreiber* bestehen darauf, unbedingt an jeder Stelle alle Regeln genau einzuhalten; jeden kleinsten Fehler kreiden sie dick mit rot an und regen sich furchtbar über jeden Text auf, der zu viele davon aufweist. Die *Zulasser* nehmen die gegenteilige Haltung ein: „Ist doch egal, ob das so stimmt oder nicht. Hauptsache, man kann es verstehen.“. Dazwischen liegen die *Pragmatiker*, die zwar Korrektheit anstreben, aber auch abweichende Lösungen zulassen bzw. diejenige auswählen, die ihnen *im vorliegenden Fall* am angemessensten erscheint. Die gleichen Haltungen gibt es auch im Bezug auf Stilregeln wie die unten aufgeführten.

Ich halte die Vorschreiber-Haltung für kleinlich und ineffizient: Man kann seine Energien fast immer für etwas anderes einsetzen, das wesentlich nutzbringender ist. Bei Stilregeln kann zudem ihre sklavische Einhaltung auch zu schauerlich schlechten Resultaten führen. Die Zulasser-Haltung andererseits ist gefährlich für die Verständlichkeit, denn für einen anderen Leser kann eine Unkorrektheit hinderlich für das Verständnis sein, auch wenn sie es für mich nicht ist. Ich halte deshalb einen gesunden Pragmatismus für die einzig sinnvolle Haltung.

9.1.6 Knappheit

Zumeist ist das wichtigste Mittel zum Erreichen einer adäquaten Form, das Dokument bei gegebener Organisation so knapp wie möglich zu halten. Das bedeutet *nicht*, Sätze zu verstümmeln oder alle weniger wichtigen Fakten einfach wegzulassen, sondern nur, Ausschweifungen zu vermeiden — was den meisten Schreibern gar nicht leicht fällt. Insbesondere sollte inhaltliche Redundanz nur sehr gezielt dort eingesetzt werden, wo sie wirklich nötig erscheint.

9.1.7 Sprache

Unzählige Bücher befassen sich mit dem Thema der guten, verständlichen Schriftsprache. Es sei jedem empfohlen, sich mindestens eines davon zu Gemüte zu führen, z.B. das sehr unterhaltsame „Deutsch für Profis“ [Sch82].

Ich stelle hier nur kurz die allerwichtigsten Regeln zusammen, nicht mit Blick auf sprachliche Korrektheit, sondern auf Verständlichkeit und Interessantheit.

1. Es gibt Sätze der *richtigen* Länge; nicht zu lang, aber auch nicht zu kurz. Wenn man seine Sätze kritisch prüft, kann man sich da einfach auf sein Gefühl verlassen. Also zum Beispiel bitte nicht

> In den meisten Fällen ist es das wichtigste Mittel zum Erreichen einer adäquaten Form, wenn man sich bemüht, das Dokument bei gegebener Organisation so knapp wie möglich zu halten, was aber *nicht* bedeutet, Sätze zu verstümmeln oder alle weniger wichtigen Fakten einfach wegzulassen, sondern nur, Ausschweifungen zu vermeiden, was den meisten Schreibern gar nicht so leicht fällt.

und auch nicht

> Meist ist das wichtigste Mittel für Adäquatheit: Knapphalten. Das bedeutet *nicht*, Sätze zu verstümmeln. Auch nicht, alle weniger wichtigen Fakten einfach wegzulassen. Es bedeutet nur, Ausschweifungen zu vermeiden. Das fällt den meisten Schreibern nicht leicht.

sondern eher wie oben im Abschnitt 9.1.6 über die Knappheit gelesen.

2. Kauderwelsch vermeiden, auch wenn es „im Fach“ üblich erscheint. Oft gibt es ein gutes und verständliches deutsches Wort für ein Fremdwort oder einen englischen Fachbegriff. Dokumentspezifisches Fachvokabular knapp und ordentlich einführen und dabei nach Möglichkeit deutsche Begriffe verwenden — diese Möglichkeit besteht durchaus nicht immer. Abschreckendes Beispiel:

> Um den Workflow managen zu können, muß das System write-Zugriff auf alle Records der User-Datenbank haben, um auch die von den Pointern referenzierten Felder updaten zu können.

Übertrieben? Zumindest nicht sehr. Besser wäre es jedenfalls zum Beispiel so:

> Um den Arbeitsablauf verwalten zu können, muß das System Schreibrecht für alle Sätze der Benutzerdatenbank haben, um auch die über Zeiger angesprochenen Felder aktualisieren zu können.

3. Einfache Wörter oder Formulierungen sind besser als pompöse. Auch ohne Kauderwelsch kann man sich verklausulieren. Sehr oft sieht man Formulierungen wie

> Zu diesem Zeitpunkt erfolgt die Darstellung des Fensters. Hinsichtlich der Effizienz ist es vorzuziehen, die Variable zur Verwendung zu bringen, als einen Prozeduraufruf durchzuführen.

Au weia, denn es ginge ja auch so:

> Nun wird das Fenster angezeigt. Die Variable zu benutzen, ist effizienter als ein Prozeduraufruf.

Auch betont exotische oder altmodische Wörter sollte man lieber vermeiden.

4. Aktiv ist besser als Passiv. Passivsätze werden typischerweise im Amtsdeutsch verwendet; von ihnen wird ein steifer und lebloser Eindruck auf den Leser ausgeübt. Ein aktiver Satz hingegen lebt und zieht den Leser mit in den Text hinein. (Übung: Warum steht hier kein Beispiel?)

5. Verben sind besser als Substantive. Also nicht

> Ein Erfordernis für die Darstellung des Fensters ist die vorherige Aktualisierung des Widgets.

sondern lieber

> Um das Fenster darzustellen, muß man zuvor das Widget aktualisieren.

Von allen diesen Regeln gibt es naturlich Ausnahmen. Siehe die Diskussion über Korrektheit oben.

9.1.8 Schönheit

Als einen besonderen Aspekt von Interessantheit sollte man anstreben, daß auch ein technischer Text ästhetisch befriedigend ist. Wenn man dies erreicht (das ist aber schwer!), kann daraus eine Zugeneigtheit der Leser erwachsen, die die Interessantheit und damit die effektive Verständlichkeit enorm erhöht. Merke: Beruf und Vergnügen können (und sollten) durchaus gemeinsam auftreten!

9.1.9 Also?

Um zu der zu Beginn gestellten Frage zurückzukehren: Das Schreiben eines technischen Dokumentes ist insofern fast das gleiche wie das Schreiben eines Zeitungsartikels, als die effektive Verständlichkeit im Mittelpunkt steht. Die Anforderungen sind sogar recht ähnlich denen für das Schreiben eines guten Romans; allerdings muß beim technischen Text im Konfliktfall (aber auch nur dann) die Korrektheit und Knappheit stets der Schönheit vorgezogen werden.

9.2 Vorgehen

Ähnlich wie bei Software kann man auch bei technischer Dokumentation eine produktorientierte oder eine prozeßorientierte Sichtweise einnehmen. Die produktorientierte wurde im vorherigen Abschnitt verwendet: Im Mittelpunkt stehen erwünschte und unerwünschte Eigenschaften des Dokuments. Diese Betrachtungen stecken zwar ein Ziel ab, geben jedoch nur schwache Hinweise, wie man *vorgehen* sollte, um zu einem guten Dokument zu kommen.

Im diesem Abschnitt besprechen wir nun die prozeßorientierte Sichtweise des Schreibens, also das Vorgehen. Verblüffenderweise kann man sehr viel von den Vorgehensmodellen der Softwareentwicklung direkt auf das Schreiben übertragen. [LPD91] beschreibt diese Analogie ausführlicher und enthält auch eine umfangreiche kommentierte Liste von Literatur zum Thema Schreiben.

9.2.1 Wie entwickelt man doch gleich Software???

Das Phasenmodell (Wasserfallmodell) der Softwareentwicklung kann man in seinen Einzelteilen recht gut auf die Erstellung technischer Dokumente übertragen. In den folgenden Abschnitten werde ich genau das tun. Anschließend bespreche ich auch noch die wichtigsten Vorgehensalternativen, die vom Phasenmodell abweichen.

Ebenso wie bei der Softwareentwicklung ist es auch beim Schreiben schwierig, eine lineare Abfolge der Schritte einzuhalten. Allerdings ist dies beim Schreiben ein viel geringeres Problem: Wegen der geringeren Strenge natürlicher Sprache im Vergleich zu Computerprogrammen können leichte Schwächen ohne weiteres hingenommen werden und wegen der geringeren Größe des Entwicklungsteams (oft nur eine Person) sind nachträgliche Änderungen schon abgeschlossener Phasen viel einfacher.

9.2.2 Analyse

In der Analysephase wird eine **Problem- und Aufgabenbeschreibung** erstellt. Diese Texte sind nicht eigentlich Bestandteil des herzustellenden Dokuments, sondern dienen dem Autor zu seiner eigenen Orientierung. Teile der Beschreibungen finden sich jedoch normalerweise später in der Einleitung des Dokuments wieder.

Die Problem- und Aufgabenbeschreibung enthält

1. Eine Beschreibung des größeren Zusammenhangs, in dem das Dokument angesiedelt ist.
2. Die Beschreibung des konkreten Problems, das im Dokument behandelt wird.
3. Die Charakterisierung der Ziele, die das Dokument erreichen soll (z.B. der Information, die es liefern soll).
4. Eine Begründung, warum und für wen das Dokument wichtig ist.
5. Die Charakterisierung des Vorwissens der angepeilten Leserschaft.
6. Eine Auflistung relevanter Randbedingungen: Zeitbeschränkungen, Umfangsbeschränkungen, technische Randbedingungen (Medien etc.), äußere Vorgaben (Standards) für Stil, Organisation oder Format.

Von zentraler Bedeutung in der Analysephase ist die richtige Charakterisierung der Leserschaft, insbesondere ihres Vorwissens. Schreibt man ein Dokument für eine Gruppe von Gleichen, wie es in der Softwareentwicklung häufig vorkommt, so ist dies relativ einfach. In anderen Fällen empfiehlt sich ein zweistufiges Vorgehen: Man nimmt ein Modell eines gewünschten Lesers an und sammelt dann, mit welchen Unterschieden dazu bei der übrigen Leserschaft zu rechnen ist.

9.2.3 Planung

Bei größeren Dokumenten empfiehlt sich die Aufstellung eines Prozeßplans. Das gilt besonders, wenn mehr als eine Person am Dokument schreiben soll. Der Prozeßplan enthält die Zeitplanung und fixiert Entscheidungen im Hinblick auf Rahmenbedingungen wie Softwarewerkzeuge und Versionsverwaltung. Er enthält außerdem eine Liste benötigter Unterlagen sowie eine Beschreibung der Konventionen zur Entscheidung von Zweifelsfällen sprachlicher, inhaltlicher oder textstruktureller Art. Die letzteren beiden Listen müssen während der Erstellung des Dokuments ständig aktualisiert werden und sind insofern Schreibhilfsmittel, die über einen reinen Plan hinausgehen.

9.2.4 Entwurf

Als Entwurfsmethode eignet sich bei Texten das Top-Down-Vorgehen; man stellt also zu Beginn die Liste der Grobabschnitte oder Kapitel auf, etwa so, wie sie später im Inhaltsverzeichnis erscheinen wird. Dann wird schrittweise verfeinert, indem man die Überschriften der Unterabschnitte und eventuell Unter-Unterabschnitte hinzufügt. Das so entstandene Inhaltsverzeichnis bildet quasi den Grobentwurf. Dieser kann nun zu einem Feinentwurf erweitert werden, indem man zu jedem Abschnitt eine Reihe von Stichworten oder -phrasen aufschreibt, die den Inhalt des Abschnitts näher bestimmen.

Bei dieser Vorgehensweise entdeckt man ständig Fehler oder Ungeschicktheiten im Aufbau, die durch entsprechendes Ändern des Entwurfs ausgebügelt werden müssen. Dieses Problem entsteht aus der (im Vergleich zu Software-Entwürfen) relativen Detailarmut des Textentwurfs, durch die Inkonsistenzen erst spät sichtbar werden. Ebendiese Detailarmut ist aber zugleich ein Segen, denn sie macht Änderungen auch sehr leicht durchführbar. Die meisten guten Autoren verwenden das Top-Down-Vorgehen nur als eine Art Gerippe für ihren Schreibprozeß und schreiben in Wirklichkeit weitgehend assoziativ: Der jeweils nächste Entwicklungsschritt wird an dem Abschnitt gemacht, zu dem einem der jeweils letzte Schritt gerade den nächsten sinnvollen Gedanken eingegeben hat. Die Rückkehr zum Top-Down-Vorgehen wird immer dann vorgenommen, wenn eine Lücke in diesem Prozeß entsteht. Die Rückkehr verhindert, sich im assoziativen Durcheinander zu verzetteln.

9.2.5 Implementation

Weniger noch als beim Programmieren läßt sich beim Schreiben eine klare Grenze zwischen Entwurf und Implementation angeben. Durch das Top-Down-Vorgehen geht eine Aktivität fast stufenlos in die andere über. Zumindest muß man aber irgendwann anfangen, die einzelnen Abschnitte „endgültig" auszuformulieren, was eindeutig eine Implementationstätigkeit ist. Egal wie gründlich die Vorarbeiten waren: Stets tauchen dabei noch neue Ungereimtheiten auf. Da paßt hier ein Gedanke nicht ohne Lücke an den nächsten und da wird dort eine Grundlage benötigt, die weiter oben noch nicht gelegt wurde. In diesen Fällen muß der Entwurf entsprechend korrigiert werden, bevor man weiter fortfahren kann. Bei solchen Änderungen erfordert es einige Disziplin, sie tatsächlich auf der entsprechenden Entwurfsebene durchzuführen, anstatt im Text zu flickschustern.

Im Prinzip gilt bei der Implementation das gleiche Spannungsverhältnis von assoziativem und Top-Down-Vorgehen. Allerdings ist hier ein assoziatives Arbeiten zusätzlich erschwert, weil so viele Details beachtet werden müssen. Deshalb sollte der Volltext als Faustregel in ununterbrochenen Arbeitsblöcken geschrieben werden, die mindestens einen ganzen Absatz umfassen.

Es gibt allerdings auch ganz andere Herangehensweisen an die Implementation als die Orientierung am Top-Down-Modell: Bei einer Arbeitsweise, die mit dem **Rapid Prototyping** verwandt ist, werden zunächst ohne viel Rücksicht auf die Qualität längere Textpassagen mit möglichst wenig Unterbrechungen geschrieben. Danach erst erfolgen Aufräum- und Verbesserungsarbeiten, die den an sich schon vollständigen Text seiner endgültigen Form annähern.

Der genau gegenteilige Ansatz ist mit dem **Risikomodell** (Spiralmodell) der Softwareentwicklung verwandt: Hier werden zuerst diejenigen Teile bearbeitet, die am schwierigsten und wichtigsten zu sein versprechen. Sie werden sehr gründlich hergestellt und gut ausgefeilt, bevor darauf aufbauend andere Teile in Angriff genommen werden.

Beim Ansatz mit **Wiederverwendung** steht am Beginn der Texterzeugung der Versuch, möglichst viele benötigte Teile aus früher geschriebenen Texten direkt oder mit Änderungen zu übernehmen. Diese Möglichkeit besteht dann, wenn man mehrere Texte zu engverwandten Themen zu schreiben hat, so daß sich etwa Definitionen oder Erläuterungen von Grundlagen übernehmen lassen. In diesem Fall sind manchmal ganze Abschnitte ohne Änderungen übernehmbar. Ein anderer Fall ist die Wiederverwendung von Textschablonen, d.h. Absätzen oder Abschnitten, in denen ein bestimmter Gedankengang verfolgt wird und die durch die Änderung von Teilsätzen oder Sätzen an das aktuelle Thema angepaßt werden können. Beispiel: Ein Abschnitt über Zweck und Verwendung einer Prozedur X hat die Form *Zweck, Vorbedingung, Nachbedingung, Besonderheiten* und kann oftmals unabhängig von X die gleichen Formulierungen zur Verbindung dieser Teile verwenden; eventuell gibt es mehrere solcher Bausteine für ähnliche Zwecke, um erstens für Variabilität in der Formulierung sorgen zu können und zweitens für unterschiedliche Fälle eine angemessene Variante bereit zu haben.

Bei Wiederverwendung besteht eine gefährliche Tendenz, daß der erzeugte Text nicht gut seiner Leserschaft angemessen ist, wenn wiederverwendete Bausteine für eine andere Leserschaft geschrieben waren. Außerdem wirkt der Text selten richtig wie aus einem Guß. Wiederverwendung ist deshalb mit großer Vorsicht zu genießen. Bei der Wiederverwendung von Text anderer Autoren (auch aus der eigenen Organisation) sind die Regeln der Fairneß und die Urheberrechtsgesetze zu beachten.

9.2.6 Testen und Überarbeitung

Wenn das Produkt (sprich: Dokument) „fertig“ ist, erfolgt ähnlich wie beim Programmieren eine Testphase. Gute Schreiber beginnen damit schon während der Textproduktion und führen viele Prüfungen ständig nebenher oder gelegentlich zwischendurch aus. Beim Testen wird das Dokument auf verschiedenen Ebenen untersucht, um die Einhaltung der Ziele zu überprüfen und ggf. durch Nachbesserung zu erreichen.

Auf der **Mikroebene** geht es um lokale Eigenschaften des Textes wie

1. Stimmen Rechtschreibung, Zeichensetzung, Grammatik?
2. Ist die Wortwahl angemessen?
3. Sind die Sätze verständlich (Länge, Aufbau) und angenehm lesbar?

Diese Prüfung können alle Personen vornehmen, die ein wenig vom Themengebiet verstehen. Der Autor selbst ist allerdings weniger gut geeignet, weil er für seine eigenen Mikroebenenfehler weitgehend blind ist.

Auf der **Makroebene** werden die globalen Anforderungen an den Text geprüft. Hierzu werden die Notizen aus der Analysephase herangezogen.

1. Stellt er sein Thema und seine Ziele klar?
2. Paßt der Text zu seiner angestrebten Leserschaft (Inhalt, Detaillierungsgrad, Ton)?

3. Ist der Text seinem Zweck angemessen (Umfang, Inhalt, Aufbereitung, Form)?

4. Ist die Organisationsstruktur klar ersichtlich?

5. Ist alle benötigte Information enthalten?

6. Hat der Gedankengang Lücken?

Für die Effizienz eines Tests ist es wichtig, daß man sich vor seinem Beginn entscheidet, ob man die Mikro- oder die Makroebene prüfen will. Wer beides zugleich versucht, überfordert sich in der Regel und übersieht dann wichtige Schwächen. Bei Prüfungen auf der Makroebene sollte man sich sogar auf einen oder zwei der angeführten Punkte pro Prüfdurchgang beschränken. Die gleichen Anmerkungen gelten analog für das Vorgehen bei der Korrektur der Schwächen.

9.2.7 Wartung

Viele Dokumente müssen über eine längere Lebensdauer hinweg mehrfach angepaßt, erweitert oder geändert werden. Dabei tauchen ganz ähnliche Probleme auf, wie bei der Wartung von Software

1. Die Entwurfsentscheidungen und der entstandene Entwurf selbst sind nicht gut genug dokumentiert, um bei der Wartung verstanden zu werden. Deshalb zerstört die Wartung allmählich immer mehr von der ursprünglichen Struktur.

2. Selbst wenn dieser Fall nicht eintritt, werden die Entwurfsbeschreibungen oft nicht ausreichend mitgepflegt. Das gleiche gilt für die Anforderungsbeschreibungen.

3. Warter haben die Neigung, Änderungen möglichst lokal zu machen. Auch dies schwächt auf die Dauer die globale Struktur.

4. Eine wohldokumentierte Versions- und ggf. Konfigurationsverwaltung ist notwendig.

5. Sind mehrere Varianten eines ähnlichen Textes zu pflegen, verstärkt sich das Problem entsprechend.

Die besten Mittel zur Lösung dieser Probleme sind leicht erfaßbare, kompakte und fest an das Dokument gekoppelte Beschreibungen der Anforderungen und des Entwurfs sowie ein möglichst modularer Aufbau der Dokumente, der Zusammenhänge zwischen Teilen so gering wie möglich hält und so explizit wie möglich sichtbar macht.

Anforderungs- und Entwurfsbeschreibungen werden am besten als Annotationen technisch direkt mit ins Dokument integriert (d.h. in die entsprechende Datei, nicht in den Ausdruck). Zusammenhänge, sowohl innerhalb oder zwischen Dokumentteilen als auch zwischen Dokument und obigen Beschreibungen, kann man über technisch verfolgbare und prüfbare Querverweise explizit machen.

9.2.8 Bilder und andere Medien

Die Regel „Ein Bild sagt mehr als tausend Worte“ ist zwar auch für technisches Schreiben ein nützlicher Hinweis, muß aber stets mit Vorsicht umgesetzt werden. (Unter „Bild“ wollen wir hier jede Art graphischer Darstellung verstehen.)

Zunächst einmal kann in vielen Fällen in der Tat eine sehr große Informationsmenge mit einem Bild transportiert werden, die als Text erheblich mehr Platz verschlungen hätte. Krasse Fälle dieser Art sind aber eher selten.

Allerdings kann man sehr oft mit einem Bild eine gewisse Information erheblich *schneller* übermitteln, was gerade für technische Dokumentation außerordentlich wünschenswert ist. Insofern sollten Bilder möglichst häufig eingesetzt werden. Dabei sind jedoch zwei Fallen unbedingt zu vermeiden:

Erstens brauchen die meisten Bilder eine Erläuterung, zumindest, wenn mehr als ein ungefährer Eindruck von etwas übermittelt werden muß. Diese Notwendigkeit kann aber den Vorteil des Bildes wieder völlig zunichte machen, wenn die Erklärung entweder zu lang, zu umständlich oder zu verklausuliert ist oder wenn sie sich schwer auffindbar irgendwo im Text versteckt. Eine gute Erklärung sollte sich direkt in der Bildunterschrift unterbringen lassen und möglichst ohne Rückgriff auf den Text verständlich sein. Kriegt man das partout nicht hin, ist wahrscheinlich das Bild überfrachtet. Abbildung 9.1 ist ein abschreckendes Beispiel für ein völlig überfrachtetes Bild (Bitte nicht nach einem Sinn suchen. . .).

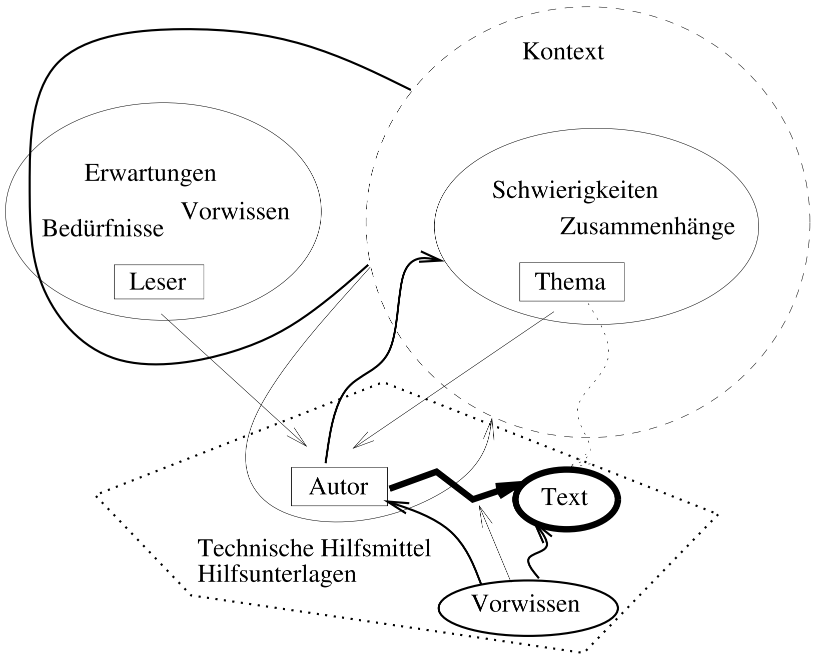

Abbildung 9.1: Zusammenhänge der beim Schreiben relevanten Faktoren (siehe Text)

Zweitens erzeugt der Druck zum Einsatz von Bildern manchmal die Neigung, triviales in einer Abbildung darzustellen. Davon fühlt sich der Leser oft auf den Arm genommen und versagt dann eventuell dem Rest des Textes die ernsthafte Aufmerksamkeit. Abbildung 9.2 ist ein abschreckendes Beispiel dieser Art.

Für Multimedia/Hypermedia-Dokumente gelten weitgehend die gleichen Regeln, wie sie bisher unter der Annahme von Papierdokumenten besprochen wurden:

- Selbst ein Hypermedia-Dokument mit seinen zahlreichen Querverweisen sollte idealerweise eine gut organisierte, linear lesbare Struktur aufweisen, weil nur diese ein vollständiges Erfassen durch den Benutzer garantieren kann.

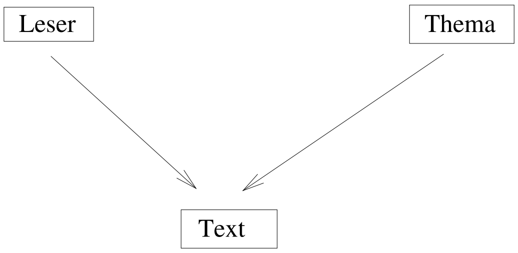

Abbildung 9.2: Sowohl Leserschaft als auch Thema beeinflussen den Autor eines Textes

- Auch wenn man die Abstützung auf geschriebenen Text zur Informationsübermittlung verringert, entstehen die gleichen Probleme der Angemessenheit der Darstellung bezüglich des Dokumentenzwecks und des Zielpublikums.
- Die vermeintlich viel größere Interessantheit von Multimedia-Dokumenten ist zum Teil eine Illusion, die aus der relativen Neuheit dieser Dinge entsteht. Ein wirkliches inhaltliches Interesse aufrechtzuerhalten (anstatt nur eine Art unterhaltsames Rauschen zu erzeugen), ist aber mit Multimedia ungefähr genauso schwierig wie mit Papier.

9.2.9 „Goldene Regeln“

Es gibt umfangreiche Sammlungen von Regeln mit dem Ziel, korrektes, gutes, verständliches oder interessantes Deutsch (oder Englisch) zu schreiben. Da diese Sammlungen häufig zu umständlich oder umfangreich sind, gebe ich hier nur eine kleine Zahl von Meta-Regeln, mit denen man aber schon ziemlich weit kommt:

1. Habe ein Anliegen (neudeutsch: *message*).
2. Orientiere Dich an Deinen Lesern.
3. Schreibe wie Du sprichst und feile das Ergebnis dann aus.
4. Schreibe nur einen Gedanken pro Satz.
5. Halte Dich nie sklavisch an einer Regel fest.

Die ersten vier Regeln bilden zusammen den Schlüssel zu interessantem, lesbaren Text: Nur wenn man genau weiß, was genau es ist, das man seinen Lesern sagen will, schafft man es, sich wirklich auf das Wesentliche zu konzentrieren und eine klare Struktur in den Text zu bringen. Und nur, wenn es einem wichtig ist, das „rüberzubringen“, was man sagen möchte, schafft man es auch, die Präsentation interessant zu gestalten. Dies sichert das Interesse des Lesers auf der Ebene der Textorganisation. Die dritte und vierte Regel liefern die Methode, mit der man sich auf der Ebene einzelner Sätze vor Unverständlichem und damit Uninteressantem bewahren kann.

Die letzte Regel besagt lediglich, daß keine Regel gut genug ist, um immer zu gelten. Wenn es also einen genügend guten Grund gibt, brich sie! Ohne Augenmaß und Entschlußkraft kommt man bei größeren Vorhaben selten zu befriedigenden Ergebnissen.

9.3 Weitergehende Information

An Literatur sei vor allem nochmals das wirklich sehr gute und unterhaltsame „Deutsch für Profis“ von Wolf Schneider erwähnt [Sch82]. Ebenfalls besonders empfehlenswert ist [DLLS92], ein recht knappes Büchlein, das sehr auf die Bedürfnisse von Informatik-Studierenden zugeschnitten ist und nicht nur zum Schreiben etwas zu sagen hat.

Materialien auf und für Englisch sind [LPD91], das als
ftp://ftp.sei.cmu.edu/pub/education/cm23.ps.Z
online verfügbar ist, sowie [You89].

Literaturverzeichnis

[AASB93] Rita L. Atkinson, Richard C. Atkinson, Edward F. Smith, and Daryl J. Bem. *Introduction to Psychology.* Harcourt Brace College Publishers., San Diego, CA, 11th edition, 1993.

[AJEG83] Allan J. Albrecht and Jr. John E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, November 1983.

[All95] C. Dennis Allen. Succeeding as a clandestine change agent. *Communications of the ACM*, 38(5):81–86, May 1995.

[And94] Ross J. Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[BBHL95] Barry Boehm, Prasanta Bose, Ellis Horowitz, and Ming June Lee. Software requirements negotiation and renegotiation aids: A Theory-W based spiral approach. In *Intl. Conf. on Software Engineering*, pages 243–253, Seattle, WA, April 1995. ACM Press.

[BCW95] Francoise Brun-Cottan and Patricia Wall. Using video to re-present the user. *Communications of the ACM*, 38(5):61–71, May 1995.

[Beh83] Charles A. Behrens. Measuring the productivity of computer systems development activities with function points. *IEEE Transactions on Software Engineering*, SE-9(6):648–652, November 1983.

[BH95] Hugh R. Beyer and Karen Holtzblatt. Apprenticing with the customer. *Communications of the ACM*, 38(5):45–52, May 1995.

[BJ94] Judith G. Brodman and Donna I. Johnson. What small businesses and small organizations say about the CMM. In *16th Intl. Conf. on Software Engineering*, pages 331–340, Sorrento, Italy, May 1994. IEEE CS Press.

[BR89] Barry Boehm and Rony Ross. Theory-W software project management: Principles and examples. *IEEE Trans. on Software Engineering*, 15(7):902–916, July 1989.

[Bro96] Fred P. Brooks. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, March 1996.

[BW84] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. on Software Engineering*, 10(6):728–738, November 1984.

[Car93] Neil R. Carlson. *Psychology — The Science of Behavior.* Allyn and Bacon, Simon and Schuster, Needham Heights, MA, 4th edition, 1993.

[CDS85] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models.* Benjamin/Cummings, Menlo Park, CA, 1985.

[Chr94] Larry B. Christensen. *Experimental Methodology.* Allyn and Bacon, Needham Heights, MA, 6th edition, 1994.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6):476–493, June 1994.

[CM90] Richard H. Cobb and Harlan D. Mills. Engineering software under statistical quality control. *IEEE Software*, pages 44–54, November 1990.

[CS91] John C. Cherniavsky and Carl H. Smith. On Weyuker's axioms for software complexity measures. *IEEE Trans. on Software Engineering*, 17(6):636–638, June 1991.

[DLLS92] Marcus Deininger, Horst Lichter, Jochen Ludewig, and Kurt Schneider. *Studien-Arbeiten: ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik.* vdf Verlag der Fachvereine und Teubner, Stuttgart, 1992.

[Dye92] Michael Dyer. *The Cleanroom Approach to Quality Software Development.* Series in Software Engineering Practice. Wiley, New York, 1992.

[Fen91] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman and Hall, London, 1991.

[Fey91] Richard P. Feynman. *Kümmert Sie, was andere Leute denken?* Piper, 1991.

[HG96] James D. Herbsleb and Dennis R. Goldenson. A systematic survey of CMM experience and results. In *Proc. 18th Intl. Conf. on Software Engineering*, pages 323–330, Berlin, Germany, March 1996. IEEE CS press.

[HKK89] Watts S. Humphrey, David H. Kitson, and Tim C. Kasse. The state of the software engineering practice: A preliminary report. In *Proc. 11th Intl. Conf. on Software Engineering*, pages 277–288. ACM, 1989.

[Hum89] Watts S. Humphrey. *Managing the Software Process.* SEI series in Software Engineering. Addison Wesley, 1989.

[Hum95] Watts Humphrey. *A Discipline for Software Engineering.* SEI Series in Software Engineering. Addison Wesley, Reading, MA, 1995.

[Hum96] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[In96] Barry Boehm Hoh In. Identifying quality-requirement conflicts. *IEEE Software*, 13(2):25–35, March 1996.

[KC95] Mark Keil and Erran Carmel. Customer-developer links in software development. *Communications of the ACM*, 38(5):33–44, May 1995.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[KL90] John C. Knight and Nancy G. Leveson. A reply to the criticisms of the Knight and Leveson experiment. *Software Engineering Notes*, 15(1):24–35, January 1990.

[Kol92] Arthur Kolb. Ein pragmatischer Ansatz zum Requirements Engineering. *Informatik Spektrum*, 15:315–322, 1992.

[KS95] Ralf Kneuper and Frank Sollmann. Normen zum Qualitẗsmanagement bei der Softwareentwicklung. *Informatik-Spektrum*, 18(6):314–322, December 1995.

[Lev95] Nancy G. Leveson. *Safeware — System Safety and Computers.* Addison Wesley, Reading, MA, 1995.

[LPD91] Linda Levine, Linda H. Pesante, and Susan B. Dunkle. Technical writing for software engineers. Technical Report SEI-CM-23, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1991.

[MM93] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics.* W.H. Freeman and Company, New York, 1993.

[Neu95] Peter G. Neumann. *Computer-Related Risks.* ACM Press, Addison Wesley, 1995.

[NJJ+96] Hans W. Nissen, Manfred A. Jeusfeld, Matthias Jarke, Georg V. Zemanek, and Harald Huber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, 13(2):37–48, March 1996.

[Pau95a] Mark C. Paulk. The evolution of the SEI's Capability Maturity Model for software. *Software Process – Improvement and Practice*, Pilot Issue:3–15, August 1995.

[Pau95b] Mark C. Paulk. How ISO 9001 compares with the CMM. *IEEE Software*, 12(1):74–83, January 1995.

[Pet92] Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design.* Vintage Books, Random House, New York, 1992. first published 1985.

[Pet95] Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs.* Random House, New York, 1995.

[RR96] James Robertson and Suzanne Robertson. *Vollständige Systemanalyse.* Hanser Verlag, München, Wien, 1996. (orig.: Complete Systems Analysis, Dorset House Publ., New York, 1994).

[SBB87] Richard W. Selby, Victor R. Basili, and F. Terry Baker. Cleanroom software development: An empirical evaluation. *IEEE Trans. on Software Engineering*, 13(9):1027–1037, September 1987.

[Sca89] D.A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6:28–36, September 1989.

[Sch82] Wolf Schneider. *Deutsch für Profis. Handbuch der Journalistensprache — wie sie ist und wie sie sein könnte.* 2. Auflage. Stern Buch, Gruner und Jahr, 1982.

[SKA96] S. Wayne Sherer, Ara Kouchakdjian, and Paul G. Arnold. Experience using Cleanroom software engineering. *IEEE Software*, 13(6):69–76, May 1996.

[SMMH77] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20(6):373–381, June 1977.

[SS92] Duane P. Schultz and Sydney Ellen Schultz. *A History of Modern Psychology.* Harcourt Brace Jovanovich College Publishers, Orlando, FL, 1992.

[TBS92] Carmen J. Trammell, Leon H. Binder, and Cathrine E. Snyder. The automated production control documentation system: A case study in Cleanroom software engineering. *ACM Trans. on Software Engineering and Methodology*, 1(1):81–94, January 1992.

[TLR93] Barbee Teasley, Laura Marie Leventhal, and Diane S. Rohlman. Positive test bias in software testing by professionals: what's right and what's wrong. In *Empirical Studies of Programmers: Fifth Workshop*, pages 206–221, Palo Alto, CA, December 1993. Ablex Publishing Corp.

[Wey88] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Trans. on Software Engineering*, 14(9):1357–1365, September 1988.

[WR94] Claes Wohlin and Per Runeson. Certification of software components. *IEEE Trans. on Software Engineering*, 20(6):494–499, June 1994.

[You89] Matt Young. *The Technical Writer's Handbook.* University Science Books, Mill Valley, CA, 1989.

[Zub97] Dave Zubrow. Trends in software process maturity: Process maturity profile of the software community 1996 year end update. In *Second Annual European Software Engineering Process Group Conf.*, pages (T201–a8)1–16, Amsterdam, Netherlands, June 16-17, 1997.