Ursache und Vermeidung von Fehlern in der Softwaretechnik.

Eine Einführung

Kerstin Becker
Institut für Informatik,
Freie Universität Berlin
schmackmo@gmx.de

Kurzfassung

Die Betrachtung von Fehlern in Software findet heutzutage meist erst am Ende der Entwicklungsphase statt, nämlich in der Form von Software-Tests und Code-Durchsichten. Doch um diesen Prozess zu vereinfachen oder weniger zeitintensiv zu gestalten, wäre es wünschenswert, Fehler, oder wohl eher Defekte, entweder zu vermeiden oder falls dies nicht möglich ist, die Fehlerquellen zu minimieren.

Diese Arbeit bietet einen knappen Überblick über Lösungsansätze aus unterschiedlichen Jahrzehnten und man kann eindeutig feststellen, dass es nicht möglich ist, Fehler völlig zu vermeiden. Genauso wie es nicht möglich ist, die Fehlerquellen gänzlich zu umgehen.

Inhalt

1. Einführung

Im Rahmen des zugrunde liegenden Seminars „Ursachen und Vermeidung von Fehlern in der Softwaretechnik“ sollen Mittel und Wege gefunden werden, wie man bei der Entwicklung von Software Fehler entweder vermeiden oder entsprechende Fehlerquellen ausschalten oder minimieren kann.

Diese Arbeit soll nun einen Überblick verschaffen über Lösungsansätze zu diesem Thema. Grundlegend für diese Arbeit waren drei Texte aus drei unterschiedlichen Jahrzehnten: Harlan D. Mills Abhandlung „How to write correct programs and know it“ von 1975 [1] in der der Autor einen Weg beschreibt, wie es möglich sein wird, korrekte Programme zu schreiben. Die Abhandlung von Frederick P. Brook, Jr: „No Silver Bullet: Essence and Accidents of Software Engineering“[2] von 1987. Hier untersucht Brooks grundlegende Probleme bei der Softwareentwicklung, beschreibt Durchbrüche in der Vergangenheit, die sich jedoch nicht auf die essentiellen Probleme beziehen, sondern auf eher nebensächliche Probleme und bietet einen Ausblick auf in der Zukunft erwartete Durchbrüche, sowohl bei essentiellen als auch bei nebensächlichen Problemen. Schließlich liegt noch der Text von Boris Beizer: „ Software is different“[3] aus dem Jahr 2000 zugrunde. Beizer beschreibt in seiner Abhandlung gängige Annahmen, die aus der „normalen“ ingenieurwissenschaftlichen Welt übertragen werden auf die softwarebezogene Ingenieur-Welt. Sein Hauptanliegen besteht darin, den Bereich der Software als eigenständigen Bereich zu betrachten und hier nach entsprechenden Regeln und Lösungen für Probleme zu suchen, gänzlich unabhängig von anderen Ingenieurwissenschaften. Diese Lösungen bestehen für ihn weitestgehend darin, das Denken aller an Software Beteiligten zu ändern.
Ob und wie die jeweiligen Ansätze, im Bezug auf Ursachen und Vermeidung von Fehlern, hilfreich sind und wie die einzelnen Ansätze zu bewerten sind, soll im Folgenden geklärt und diskutiert werden.
Aus diesem Grund werden die drei Texte chronologisch auf die Frage nach der Brauchbarkeit im Bezug auf die Ursachen und Vermeidung von Fehlern hin untersucht werden. Im Anschluss an die Analyse sollen angefallene Fragen kurz gestellt werden, so dass ein Ausblick auf die in Zukunft anfallenden Forschungen entsteht, bevor das Fazit schließlich den Abschluss dieser Arbeit bildet.

2. How to write correct programs and know it

Mills beschreibt in seiner Abhandlung aus dem Jahr 1975 sehr enthusiastisch und vor allem sehr optimistisch ein Verfahren mit dem es möglich sein soll, in Zukunft fehlerfreie bzw. korrekte Software schreiben zu können. Seine Prognose lautet: „The professional programmer of tomorrow will remember, more or less vividly, every error in his career." [p. 363]

2.1. Voraussetzungen

Hier stellt sich die Frage, wie Mills dieses Ziel erreichen will und es stellt sich heraus, dass es sehr vieler erfüllter Vorraussetzungen bedarf, damit dieses Ziel erreichbar ist.

1. Eine klare und eindeutige Spezifikation und
2. bekannte Hardware, denn: „A correct program defines a procedure for a stated processor to satisfy a stated specification. If you don't know what a program is supposed to do, or don't know how the processor is supposed to work, you can't write a correct program." [p. 363]
3. Das Wissen und die Sicherheit, dass es möglich ist, korrekte Programme zu schreiben, schließlich wird man es nie beweisen können und
4. Das Wissen, dass ein korrektes Programm nicht für immer und ewig korrekt sein wird, denn die Anforderungen können und werden sich ändern.[1] [p. 364]
5. Sich des Unterschieds zwischen Korrektheit (correctness) und Fähigkeit (capability) bewusst sein. Dies bedeutet, dass es einen Unterschied gibt, ob das Programm korrekt arbeitet, oder ob es so arbeitet wie man es erwartet. Hier ist zu beachten, dass Erwartungen weitaus schwieriger zu realisieren sind als bestimmte und vorgegebene Verhaltensweisen, da sie meistens nicht explizit genannt werden. [ebd.]

2.2. Der korrekte Code

Aufbauend auf diese Voraussetzungen schlägt Mills nun Vorgehensweisen vor, mit denen man das Ziel korrekten Code zu produzieren, erreichen kann.

1. Funktionen benutzen, bei grundlegenden Problemen und bei grundlegendem Verhalten in der Programmlogik[2]. Damit werden unerwünschte und oft unbemerkte Nebeneffekte (wie Datenkorruption) vermieden. [p. 365]
2. Strukturiertes Programmieren benutzen. Strukturiertes Programmieren beinhaltet viele kleine Schritte die beim Entwicklungsprozess helfen sollen, Fehler zu vermeiden. So z.B. divide-and-conquer, kein Benutzung von GOTO-Anweisungen. [p. 366]
3. Vermeidung von Syntaxfehlern: Wenn man in dem weitaus einfacher zu realisierenden Bereich der Syntax auf Präzision achtet, so hat dies unweigerlich Auswirkungen auf den ganzen restlichen Bereich der Arbeit und man macht automatisch auch inhaltlich weniger Fehler, sodass in der Software weniger oder keine Defekte zu finden sind.

[1] Dies sollte jedoch kein allzu großes Hindernis sein, schließlich kann man bei sich ändernden Anforderungen und mit entsprechend gutem Design und Erklärungen den Code auch ändern.

[2] Hier merkt man dem Text sein Alter sehr deutlich an. Mills geht in seinem Text von maschinennahen Programmiersprachen aus. Mit den heute üblichen höheren Programmiersprachen, stellt dieser Punkt jedoch kein Problem mehr dar.

2.3. Beurteilung

Mills weist in seiner Arbeit auf grundlegende Probleme in der Software-Entwicklung hin. So z.B. auf eine ausreichend gute und detaillierte Spezifikation oder auch der Hinweis, dass erwartet nicht gleich korrekt ist. Und obwohl seine Vorschläge zur Produktion von korrekter Software größtenteils so praktiziert werden, ist doch klar, dass heutzutage keineswegs auch korrekter Code geschrieben wird. Die von ihm vorgeschlagenen Punkte sind heute kein Thema mehr aufgrund von höheren Programmiersprachen, aber auch aufgrund von Entwicklungsumgebungen und Hilfswerkzeugen, die den Programmierer in seiner Arbeit unterstützen. Man merkt dem Text deutlich sein Alter an, denn heute käme wohl kaum einer auf die Idee die Möglichkeit von vornherein korrektem Code in Erwägung zu ziehen, doch sind die bereits angesprochenen Themen noch immer aktuell. Eine gute Spezifikation trägt wesentlich dazu bei, ein korrekt arbeitendes Programm zu schreiben. Doch was Mills nicht anspricht ist das größte Problem bei der Herstellung von Code, nämlich die Komplexität der Aufgabe. Doch wird dies in den folgenden Texten eine wichtige Rolle spielen.

3. No Silver Bullet

Brooks untersucht in seiner Abhandlung grundlegende Probleme, die bei der Softwareentwicklung auftreten. Er führt Durchbrüche auf diesem Gebiet an, die in der Vergangenheit gelungen sind und stellt fest, dass diese Durchbrüche jedoch nicht die essentiellen Probleme bei der Entwicklung betreffen, sondern eher nebensächliche Probleme behandeln. So wurden zwar große Fortschritte in der Effizienz gemacht, und dem Entwickler mächtige Werkzeuge an die Hand gegeben, die ihm die Arbeit erleichtern, aber das grundlegende Problem der Komplexität blieb bestehen. Aufhänger für seinen Essay ist übrigens eine Geschichte über Werwölfe, die nur mithilfe einer magischen Waffe (in Form von silbernen Geschossen) besiegt werden können. Eben dieses magische „Etwas“ sucht man auch in der Softwareentwicklung, bisher konnte aber leider noch keines gefunden werden. Leider ist sein Ausblick auch eher zweifelnd, ob es ein solches „Etwas“ je geben wird.

3.1. Grundlegende Probleme

Für Brooks gibt es vier grundlegende Probleme:

1. Komplexität:
    a. Diese unterteilt er noch einmal in die Komplexität der einzelnen Komponenten, die z.B. entsteht durch die Größe der Komponenten, durch deren komplexen Inhalt oder durch die Wichtigkeit der Laufzeiteffizienz etc.
    b. und in die Komplexität durch die Interaktion der Komponenten; hier sind die emergenten[3] Eigenschaften nicht bekannt und dies führt zu erhöhter Komplexität und damit zu Problemen

   Mögliche Folgen dieser Komplexität:
   Das Risiko, dass der Entwickler bei seiner Arbeit etwas übersieht, steigt an. Dadurch wiederum, kommt es zu Unzuverlässigkeiten. Ein weiteres Problem gesteht in der Benutzbarkeit oder auch Handhabbarkeit. Diese kann fraglich werden, wenn das Programm aufgrund der gestiegenen Komplexität auch in der Bedienung komplexer und damit schlechter/schwieriger zu bedienen ist. Mit steigender Komplexität wird die Erweiterbarkeit des Produktes kritisch. Stichwörter hierzu sind „side effects“ oder auch emergente Probleme. Betrachtet man die Kommunikation zwischen Teamkollegen, kann eine steigende Komplexitätsrate ebenfalls zu Problemen führen. Hier spielen mehrere

[3] Emergente Probleme, sind Probleme die auftreten durch das Zusammenspiel der einzelnen Komponenten. D.h. die einzelnen Komponenten können an sich korrekt und defektfrei sein, Defekte treten erst auf, wenn zwei oder mehrere Komponenten zusammenarbeiten.

Faktoren eine Rolle. Zum einen die möglichen, unterschiedlichen intellektuellen Fähigkeiten eines jeden Entwicklers und das Sprachproblem[4]. Aufgrund der bisher angesprochenen Probleme, kann es daher zu Produktfehlern kommen, die wiederum dazu führen, dass es zu einer Kostenexplosion für das betroffene Produkt kommt. Spätestens zu diesem Zeitpunkt spielt die Zeit dann eine Rolle. Denn wenn es zu Produktfehlern kommt, dann wird schnell die Zeit knapp. In extremen Fällen reicht sie vielleicht nicht einmal aus. Aber nicht nur aus Entwicklersicht führt eine erhöhte Komplexität zu Problemen, auch aus Management-Sicht können Probleme auftauchen, nicht zuletzt kann der Überblick über ein sehr komplexes System oder Projekt verloren gehen. [vgl. p.2]

2. Konformität

   Auch hier wird unterschieden in Konformitätsprobleme zu bereits bestehenden Interfaces, an die man den Code eventuell anpassen muss und in Konformitätsprobleme mit der Benutzeroberfläche, weil es z.B. eine Vorgänger-Version gibt oder ähnliche Produkte auf dem Markt sind. [vgl. p.3]

   Die Folge ist hier eine Erhöhung des Komplexitätsgrades. Durch das Anpassen der Software muss mehr intellektuelle Energie aufgebracht werden, was im Endeffekt heißt, dass das Problem komplexer geworden ist. Und die Folgen der Komplexität wurden oben bereits beschrieben.

3. Änderbarkeit
   a. Änderungen durch neue Anwendungsmöglichkeiten: „All successful software gets changed. Two processes are at work. First, as a software product is found to be useful, people try it in new cases at the edge of or beyond the original domain. The pressures for extended function come chiefly form users who like the basic function and invent new uses for it." [ebd.]
   b. Anpassungs-Änderungen: "Second, successful software survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, then at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicle opportunity." [ebd.]
      Die Folge auch hiervon ist eine erhöhte Komplexität.

4. Unsichtbarkeit

   Software ist unsichtbar und nicht sichtbar zu machen. Versucht man die Struktur zu erfassen, braucht man viele Diagramme, die meist nicht aufeinander aufbauen oder sich nicht gut ergänzen. Meist sind sie nicht oder wenig hierarchisch. [vgl. p. 3-4]

   Die Folgen sind hier: Für eine Person alleine ist es fast unmöglich, das Design zu gestalten. Es kommt hinzu, dass die Kommunikation zwischen den Beteiligten schwierig wird, da hierdurch Sprachbarrieren und Deutungsunterschiede entstehen. Eine vereinheitlichte und leicht zu verstehende Syntax wäre hier eine Lösung.

3.2. Durchbrüche in der Vergangenheit

Zu den erfolgreichsten und effizientesten Entwicklungen der Vergangenheit zählt Brooks

1. Höhere Programmiersprachen: Die Erfindung, Einführung und Nutzung dieser Sprachen hat die Produktivität mindestens um den Faktor fünf erhöht, die Produkte zuverlässiger und verständlicher gemacht. [p.4] Da man sich nun nicht mehr um die grundlegenden also maschinennahen Probleme kümmern muss, wird ein ganzer Komplexitätslevel übersprungen. Doch erhält man hierdurch auch mehr Werkzeuge an die

[4] Dies ist zwar nicht unbedingt ein Problem der Komplexität, denn Sprache ist leider etwas sehr ungenaues. Doch muss man auch hier festhalten, dass es mit einer steigenden Komplexität noch schwerer wird, sich auf der sprachlichen Ebene auf dieselben Voraussetzungen und Definitionen zu beziehen.

Hand, mit denen man immer komplexere Aufgaben lösen kann, die vorher nicht möglich waren. So entsteht aber wieder neue Komplexität, nur auf einer anderen Ebene.

2. Time-sharing (Zeitteilverfahren): Auch hier wird die Produktivität gesteigert und der Programmierer behält eher den Überblick, denn es entstehen keine großen Wartezeiten, in denen er das gerade Gedachte wieder vergessen könnte, doch ändert sich auch hier nichts daran, dass die Aufgabe die er zu meistern versucht komplex bleibt. [ebd.]
3. Standardisierte Programmierumgebungen: Auch hier wird die Produktivität deutlich verbessert, denn durch den Einsatz von standardisierten Schnittstellen zu einem Betriebssystem wird die Verwendung von Code auf unterschiedlichen Plattformen stark vereinfacht.

Doch wie man sehr gut erkennen kann, bieten diese Entwicklungen keine Lösungen für das Grundproblem der Komplexität. Diese Durchbrüche bieten lediglich Lösungs- oder Verbesserungsmöglichkeiten für nebensächliche Probleme.

3.3. Die magische Waffe

Angelehnt an die Einführung über die Werwölfe beschreibt Brooks nun Entwicklungen, die im Allgemeinen als aussichtsreich angesehen werden, also als magische Waffe bei der Softwareentwicklung und beurteilt diese kurz.

1. Fortschritt bei den höheren Programmiersprachen: Dies führt wahrscheinlich zu höherer Produktivität aber wie oben bereits angesprochen auch zu weiterer Komplexität, nur auf einer anderen Ebene (komplexere Aufgaben möglich, Design bleibt komplex). Den größten Nutzen sieht Brooks darin, dass Programmierer hiermit gezwungen werden, sich mit Software-Design Techniken auseinanderzusetzen.
2. Objektorientiertes Programmieren: Auch hier bleibt das Problem des Software-Designs bestehen.
3. AIs oder Expertensysteme: Ohne mehr über Expertensysteme wissen zu müssen, genügt es zu wissen, dass man ohne einen Experten, egal auf welchem Gebiet, ein solches auch nicht bauen kann. Auch hier gibt es keine Lösung für komplexe Aufgaben.
4. Graphisches Programmieren: Das Problem ist hier, wie macht man Software sichtbar? Auch wenn man heute, unabhängig von diesem Text, UML als Teil dieser Lösung hat, bleibt doch das Problem bestehen, dass man komplexe Probleme erst einmal verstehen muss, um sie bearbeiten zu können und ein Modell daraus machen zu können.
5. Verifikation: Eine Verifikation soll (meist mathematisch) beweisen, dass ein Programm der Spezifikation entspricht. Hier entstehen jedoch zwei Probleme. Erstens: Um ein Programm verifizieren zu können, muss eine Spezifikation bestehen. Zweiten: Ist diese Spezifikation nicht vollständig, oder fehlerhaft, kann ein Programm dieser Spezifikation entsprechen, aber trotzdem nicht das erwartete Verhalten aufweisen. Hier besteht ein Problem in der Anforderungsanalyse und in der Auswahl von Architektur und Design.
6. Umgebungen und Werkzeuge: Diese unterstützen den Programmierer wesentlich in seiner Arbeit, unter Anderem darin, dass er nicht jedes einzelne Konstrukt einer Programmiersprache kennen muss. Doch das Denken muss er noch immer alleine übernehmen. Komplexe Probleme werden hierdurch nicht gelöst.
7. Workstations: Man hat mehr Rechenleistung zur Verfügung. Das macht das Entwickeln vielleicht schneller, aber wird die Software dadurch wirklich besser?

Die Hoffnungen werden also deutlich enttäuscht. Das Grundproblem wurde nicht gelöst und damit wurde auch keine, sehnlichst erhoffte, magische Waffe gefunden. [vgl. p.5-10]

3.4. Viel versprechende Ansätze

Hier stellt Brooks nun seine eigenen Hoffnungen für die Zukunft dar.

1. Buy-or-build:

   Beim Bau eines neuen Systems sollte man sich auf dem Markt umsehen, ob es vielleicht ein passendes Produkt gibt und abwägen, ob es nicht billiger ist, dies zu kaufen, als selbst eines zu bauen. Kauft man ein fertiges Produkt, hat man selbst weniger Aufwand, der Komplexitätsgrad sinkt. Doch in einem solchen Fall nur für einen selbst und nicht allgemein, irgendwer muss die Denkarbeit ja machen. Es gibt vieles was für Kaufen spricht, aber auch viele Gründe, eine Software selbst zu bauen. Dies jedoch an dieser Stelle zu diskutieren, würde den Rahmen dieser Arbeit sprengen.

2. Anforderungen verfeinern und Prototyping
   Bei exakten, vollständigen und verständlichen Anforderungen ist es leichter, eine gute und korrekte Software herzustellen. Der Softwareprozess wird insofern erleichtert, dass ein Großteil der Denkarbeit bereits bei der Anforderungsanalyse gemacht wurde. Defekte in der Software werden so wahrscheinlich minimiert. Was das Prototyping anbelangt, lernt man natürlich sehr viel über das zu bauende System und Fehler die man beim Prototype macht und Defekte die daraus entstehen, werden in der nächsten Version nicht mehr gemacht, die Defekte entstehen erst gar nicht.

3. Großartiges Design und großartige Designer:
   Gutes Design verhindert wahrscheinlich einige Fehler, doch bleibt auch hier die Frage, wie kommt man zu gutem Design, und wie bekommt man gute Designer. Und es stellt sich weiterhin die Frage, ob gutes Design die Komplexität, mit der Entwickler zu tun haben, wirklich auch mindert, oder ob es nicht einfach nur hilft, den Überblick zu behalten und Mittel und Wege bereit zu stellen, wie ein Problem gelöst werden könnte. Die eigentliche Lösung bleibt doch das Problem.

3.5. Beurteilung

Die Beobachtungen die Brooks macht scheinen mir größtenteils einsichtig und richtig, doch fehlt mir der kritische Blick auf seine eigenen Vorschläge, zumindest, wenn man sie auf die Frage hin liest, „Was sind die Ursachen und wie vermeide ich Fehler in der Software?“. In diesem Fall, bleibt, wie oben bereits angedeutet, jeweils das Problem der Komplexität bestehen und wie Brooks bereits selbst gesagt hat, gibt es hierfür keine Lösung, denn dies ist das eigenste Wesen von Software. Die von ihm vorgeschlagenen Ansätze deuten aber ganz gut an, was Ursachen für Fehler und Defekte sein können. Bauen statt kaufen, schlechte Anforderungsanalyse und schlechtes Design. Dies gibt einen Hinweis in welcher Richtung gute Ansätze, zwar nicht für die Lösung des Problems Komplexität, gefunden werden können, aber wo Lösungen für das Problem „Defektvermeidung oder Defektfrüherkennung“ zu finden sind.

4. Software is different

Beizer beschreibt in seiner Abhandlung die gängigen und meist falschen Annahmen im Software-Bereich sowohl bei Entwicklern und Managern als auch bei Anwendern. Er versucht Argumente und Wege zu finden, eingefahrenes und starres Denken zu stoppen und/oder zu ändern. Hauptsächlich will er jedoch erreichen, dass Software als eigenständige Disziplin angesehen wird, unabhängig von anderen Ingenieurwissenschaften. Schließlich ließen sich viele Regeln aus diesen Wissenschaften nicht auf Software übertragen. Außerdem plädiert er dafür, dass gut funktionierende Methoden beibehalten werden, wie etwa konservatives Vorgehen, wenn es um die Sicherheit von Software geht.

4.1. Falsche Annahmen

Zunächst versucht Beizer mit den Annahmen aufzuräumen, die gemeinhin bei Software gemacht werden. Allen voran stellt er die physikalischen Schranken, denen Software im Allgemeinen nicht unterliegen.

4.1.1. Lokalität

1. Raum-Abhängigkeit: Die Symptome eines Defekts können beliebig weit von der Ursache entfernt sein.
   Beispiel aus der physikalischen Welt: „If you have a problem with the windshield wipers of your car, you do not expect it to affect the radio." [p.296]
2. Zeit-Abhängigkeit: Die Symptome eines Defekts können beliebig lange nach Ausführung des fehlerhaften Codes auftreten.
   Beispiel aus der physikalischen Welt: „The tire blows out causing the car to lurch to the right immediately." [p.297]
3. Konsequenzen und Proportionalität: Die Konsequenzen eines Defekts können, was die Proportionalität betrifft, willkürlich mit der Ursache in Beziehung stehen.
   Beispiel aus der physikalischen Welt: „Our tires wear out gradually, so we gradually lose road-handling; not catastrophically and suddenly when the tread depth falls below 4.3 mm." [ebd.]

4.1.2. Komplexität

1. Emergenz
   Jede Komponente für sich genommen ist korrekt und arbeitet auch korrekt. Doch die Zusammenarbeit der einzelnen Komponenten, oder die gleichzeitige Nutzung mehrerer Komponenten hat Probleme oder gar ein völliges Versagen der Software zur Folge. In der physikalischen Welt kann man durch die Betrachtung der einzelnen Elemente jedoch meistens sehr gute Rückschlüsse auf das Ganze ziehen.
2. Proportionale Komplexität
   Während in der physikalischen Welt zwei Komponenten A und B die jeweiligen Komplexitäten C_A und C_B haben und eine Kombination der beiden ein additives Verhältnis zueinander haben (C_A+C_B) so ergibt sich in der Software-Welt eher ein mutipliziertes Verhältnis ($C_A x C_B$).
3. Komplexität durch Funktionalität
   Erhöht man den Funktionsumfang eines Produktes, so ergeben sich Komplexitätszuwächse in mehreren Bereichen. Erstens in der Form, das mehr Code produziert wird und die Übersichtlichkeit dadurch sinkt. Zweitens in der Form von erhöhter Komplexität in den Algorithmen und in erhöhter intellektueller Anstrengung und schließlich eine erhöhte Komplexität in der Architektur. Zusammengefasst bedeutet dies einen großen Anstieg in der internen Komplexität

4. Sicherheitslimits
   Hier stellt sich die Frage, wo das Sicherheitslimit für Software ist. So lange wie man hierfür keine Lösung gefunden hat, hilft eigentlich nur das Zurückgreifen auf altbewährte Tugenden (wie z.B. beim Brückenbau): Konservatives, d.h. normales Vorgehen, zumindest da, wo dies möglich ist. [vgl. p. 299-303]

4.1.3. Qualität

Hier lautet die Fragestellung: Qualität wovon? Oder auch Wie stellt man Qualität fest?
Es gibt drei gängige, allerdings nicht sehr aussagekräftige Qualitätsmerkmale.

1. Defekte pro Zeile Code: Doch erfährt man hier nichts über die Schwere der Defekte, außerdem erfährt man nichts über emergente Probleme
2. Defektfindungs-History: Hier wird aufgezeichnet wann man welchen Fehler gefunden hat. Mit der Zeit sinkt erfahrungsgemäß die Rate der gefundenen Fehler. Dies bedeutet jedoch nicht, dass es keine Fehler mehr gibt, sondern dass man sie entweder nicht findet, weil sie emergent sind, oder auch weil man schlichtweg einfach keine Zeit, oder keine Lust oder einfach auch keine Ideen mehr hatte.
3. Die Defektrate von Defekten, die den Anwendern aufgefallen sind: Hier kann ein Problem sein, dass den Anwendern Dinge auffallen und als Defekt deklarieren, die eigentlich gar keine sind, sondern dass Verhalten entdeckt wurden, die durchaus so gewollt waren, für den Anwender aber nicht unbedingt Sinn machen. Des Weiteren möchte man zum Zeitpunkt der Auslieferung eines Produkts möglichst keine Defekte mehr in seinem Produkt wissen. Dies ist also einfach ein zu später Zeitpunkt um Defekte aufzudecken. [vgl. p.304-305]

Es zeigt sich, dass Qualität auch ein Komplexitätsproblem ist, da Qualität zu einem großen Teil vom emergenten Verhalten abhängig ist und damit nur sehr schwer messbar.

4.2. Lösungen

Lösungen für die Probleme in der Softwareentwicklung werden wahrscheinlich nicht von heute auf morgen gefunden und noch wahrscheinlicher werden sie nicht in einem Bereich zu suchen und zu finden sein. Beizer macht jedoch Vorschläge, wie man den Entwicklungsprozess entzerren kann und wie man den Druck, der oftmals auf den Entwicklern lastet mindern könnte. Beizer spricht Missverhältnisse und Fehlverhalten auf verschiedenen Ebenen oder aus verschieden Sichten an:

4.2.1. Die Entwicklersicht

Leider muss man wohl akzeptieren, dass es in der Welt in der wir leben keine einfachen Lösungen gibt. Man kann jedoch versuchen, die Denkmuster zu ändern. Aus Entwicklersicht führt er an, dass man die Einschränkungen die gegebenenfalls durch das Design entstehen, akzeptieren muss, das Design auch anwenden muss. Durch Software-Design erhält man einen besseren Überblick über die Software, es hilft oft auch, die Software besser zu verstehen. Als Verhaltensregeln fordert er Strukturiertes Programmieren, Starke Typisierung (strong typing), die Vermeidung von globalen Daten, Style-Regeln und Style-Checker und Kapselung. Doch schränkt Beizer hier gleich ein, dass dies keine ultimative Liste für gutes Programmieren ist und nicht sein will, aber es ist eine Liste, die den Entwicklungsprozess übersichtlicher gestalten kann und dadurch vielleicht hilft, Fehler zu vermeiden.

4.2.2. Die Managementsicht

Prioritäten bei der Softwareentwicklung aus der Sicht des Managements waren bisher oftmals: „1. Develop it as fast as possible, 2.Make it run as fast as possible, 3. Build it as tight as

possible, 4. Put in as many features as you can, 5. Do it at the lowest possible cost, 6. Worry later. Bugs will get fixed."[p.309]
Brooks stellt eine ganz andere Liste zusammen, nämlich eine Liste von Fragen, die sich auf die Qualität und den Inhalt des Produktes beziehen und so wahrscheinlich den Druck auf die Entwickler mindern wird, die dadurch möglicherweise weniger Defekte im Entwicklungsprozess produzieren: „1. Can it be analyzed? Is its behavior predictable? 2. Can it be tested? 3. Does it have a composable model? 4. Does it work? 5. Have feature, component, and data interaction been reduced to the absolute minimum? 6. Does it have the features the users need (as contrasted to want)?" [p.310]

4.2.3. Die Benutzersicht

Dies lässt sich leider nur sehr salopp ausdrücken, um klarzumachen, was Beizer hier aussagen möchte. Vielleicht ist es irgendwann einmal möglich, den Benutzern von Software ein Produkt zu verkaufen, das als großes Plus nicht bloß noch mehr und noch eindrucksvollere Features aufzuweisen hat. Vielleicht ist es möglich, den Benutzern klar zu machen, das eine hohe Stabilität, Zuverlässigkeit oder Bedienbarkeit tolle Eigenschaften der Software sind, diese aber zu einem bestimmten Preis gekauft werden müssen. Vielleicht ist dem Benutzer auch klarzumachen, dass er nicht unbedingt einen Texteditor braucht, der nebenher auch noch Kaffee kochen kann. Zugegeben, das ist alles ziemlich schwierig und sieht nicht unbedingt aus, als könnte es von Erfolg gekrönt sein.
Schließlich führt Beizer noch das Schlagwort die Ehrlichkeit ein. Ihm geht es hier hauptsächlich um die Ehrlichkeit der Entwickler dem Management, den Marketing-Leuten und Benutzern gegenüber. Die Entwickler müssen sich und auch anderen gegenüber die Wahrheit eingestehen. Wenn man als Entwickler nicht weiß, wie etwas funktioniert, dann muss man dies auch äußern und nicht behaupten, dass man das System schon zum laufen brächte. Beizers Fazit: Wenn Marketing, Management und Benutzer überzogenen Vorstellungen haben, dann ist der Entwickler in solch einem Fall selbst daran Schuld.

4.3. Beurteilung

Beizer rückt in seiner Abhandlung manch eine Vorstellung zurecht. Dass physikalische Gesetze nicht unbedingt auf ein Softwaresystem anwendbar sind, ist einem Entwickler meist klar, aber den Anwendern, Managern und Marketing-Leuten auch? Hierfür versucht Beizer in seinem Text zu sorgen. Durch seinen saloppen Stil und seine meist witzigen Beispiele ist sein Anliegen auch für Nicht-Entwickler verständlich und nachvollziehbar. Doch stellt man sich die Frage, ob seine Vorschläge auch wirklich von Erfolg gekrönt sein werden.

Aus Entwicklersicht lässt sich sagen, dass die Vorschläge zum „besseren" Programmieren wahrscheinlich fruchtbar sind, sie werden größtenteils eingesetzt oder zumindest propagiert, doch erfordern sie Disziplin und auch ein gewisses Maß an Weiterbildung zu dem der Arbeitgeber, aber auch der Entwickler selbst bereit sein muss.

Ob das Management seine Prioritäten in die vorgeschlagene Richtung ändert, lässt sich für mich schwer einschätzen, da ich in diesem Bereich keine Kenntnisse habe. Ich kann mir jedoch vorstellen, dass auch hierfür viel Zeit, vor allem für die Schulung von Managern benötigt wird. Wahrscheinlich ist auch ein eventuell nicht vorhandenes Fachwissen der Grund, weshalb Situationen falsch eingeschätzt werden.[5]

Bleibt noch die Benutzersicht: Die Frage nach dem Qualitätsbewusstsein der Benutzer ist recht einfach zu bewerten, wenn man von sich selbst ausgeht. Wir alle (als Benutzer von Software) haben uns daran gewöhnt, dass die Software einen immer größeren Funktionsumfang hat, dass sie leichter zu bedienen ist und dass sie immer billiger wird. Wollen wir das

[5] Z.B. Wenn dem Management klar ist, dass mehr Features mehr Aufwand beim Programmieren und vor allem bei Testen bedeutet, kann darauf auch Rücksicht genommen werden.

wirklich aufgeben? Wenn ich ehrlich bin, dann sage ich: Nein, ich will das alles und noch viel mehr. Die Erwartungen der Benutzer zu dämpfen, könnte auch ein Stillstand in der Software-Entwicklung bedeuten. Wenn die Benutzer nicht mehr verlangen, weshalb sollte dann mehr gemacht werden, man kann sich bequem auf seinen Lorbeeren ausruhen. Innovationen bleiben auf der Strecke.

Schließlich bleibt noch zu sagen, dass es vermutlich immer mal wieder Projekte geben wird, die schnell fertig werden müssen oder die möglichst kostengünstig hergestellt werden müssen. Wirkliche Vorschläge für solch einen Fall macht Beizer nicht. Für ihn liegt die Lösung einzig und alleine im Ändern des Denkens und Ändern der Prioritäten. Ich denke, Beizer macht es sich etwas zu leicht, indem er sich hierauf konzentriert, zumal er anderen Lösungsmöglichkeiten weder Raum noch Zeit lässt. Innovationen bleiben so allzu leicht auf der Strecke und die Suche nach (anderen) Lösungsmöglichkeiten für das Fehler- und Komplexitätsproblem wird im Keim erstickt.

5. Zusammenfassung und Ausblick

In allen drei Texten hat sich herausgestellt, dass die Komplexität von Software das größte Problem im Entwicklungsprozess ist. Nicht nur auf der Programmierebene, sondern schon bei der Anforderungsanalyse und im Entwurf. Wird die Komplexität durch neue Konzepte, wie z.B. beim Übergang von sequenzieller zu objektorientierter Programmierung, verringert, entstehen mit dieser Weiterentwicklung auch gleichzeitig mehr Möglichkeiten, die natürlich auch wahrgenommen werden. Dadurch steigt aber wieder der Grad an Komplexität. Dies ist, so muss man leider feststellen, ein Teufelskreis.
Was die grundlegende Frage dieser Einführung betrifft, nämlich die Frage nach Ursachen und Vermeidung von Fehlern, so wurden doch zumindest Ansatzpunkte und Ideen geliefert. Zum einen hat Brooks dargelegt, dass in der Vergangenheit durchaus Wege gefunden wurden, den Entwicklungsprozess zu unterstützen. Was die Vermeidung von Fehlern anbelangt, sollten hier wohl Entwicklungsumbebungen und Tools genannt werden, die den Entwickler in seiner Arbeit unterstützen. Aber auch die Weiterentwicklung auf dem Gebiet der Programmiersprachen. Obwohl sich in diesem Feld die Komplexität und damit die Fehleranfälligkeit wohl nur verlagert.
Was die Ursachen von Fehlern anbelangt, scheint mir Beizer einen wichtigen Punkt anzusprechen, nämlich den Leistungsdruck, der von vielen Seiten auf die Beteiligten an einem Software-Produkt wirkt. Den Appell, das Denken der Menschen in diese Hinsicht zu verändern erscheint mir überambitioniert und überoptimistisch. Hier stellt sich auch die Frage, ob diese Vorschläge wirklich zu einer defektfreieren Software führen oder ob damit nicht nur die Fortschritte, die bisher gemacht wurden und in Zukunft gemacht werden sollen, gebremst werden. Und wenn man sich selbst fragt, ob man bereit ist, diese Verhaltensmaßnahmen für sich anzunehmen, muss man wahrscheinlich eingestehen, dass sie in der Theorie gut klingen und wahrscheinlich wirksam sind, dass man aber wohl doch zu undiszipliniert ist, sich daran zu halten. Und mal ehrlich, setzt man sich seinem Chef gegenüber wirklich durch und verlangt, dass der Leistungsdruck, der auf einem lastet gemindert wird, damit man bessere Arbeit abliefert oder will man nicht doch lieber seinen Job behalten? Und was passiert, wenn eine Software wirklich mal schnell fertig werden muss? An dieser Stelle bleiben recht viele Fragen offen und es scheint fast so, als könnte man zwar Ursachen für die Produktion von Defekten ausmachen, aber als sei es unmöglich, diese Quellen gänzlich auszuschalten.

6. Literatur

[1] Boris Beizer. Software is different. Annals of Software Engineering, Volume 10, Numbers 1-4, March 2000, Pages: 293 – 310

[2] Frederick P. Brooks, Jr.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, Vol. 20, No. 4 (April 1987) pp. 10-19

[3] Harlan D. Mills. How to write correct programs and know it. Proceedings of the international conference on Reliable software, Los Angeles, California, 1975, pp.363-370