Seminar zur Ursache und Vermeidung von Fehlern in der Softwaretechnik

Kerstin Becker
Institut für Informatik
FU Berlin
02.04.2006

Übersicht

- **Einführung in das Thema**
- Geschichtlicher Rückblick
  - ... in das Jahr 1975
  - ... in das Jahr 1987
  - ... in das Jahr 2000
- Zusammenfassung
- Fragen

Einführung

- Software-Prozess, v.a. Qualitätssicherung besteht fast ausschließlich aus nachträglichen Schritten

- Weshalb man bisher keine Mittel und Wege gefunden hat, das Einbauen von Defekten zu verhindern?

- In diesem Seminar soll untersucht werden, was die Ursachen für das Einbauen von Defekten sind, in welchen Ausprägungen diese auftauchen und welche (neuen wie alten) Strategien entwickelt wurden, sie zu vermeiden.

Ablauf des Seminars

- **Ursachenforschung**
  1. Auftreten von Defekten (Software)
  2. Auftreten von Fehlern (Programmierer)

- **Vermeidung von Fehlern, Verhinderung von Defekten**
  1. Prozessorientierte Mittel
  2. Kodierorientierte Mittel

Übersicht

- Einführung in das Thema
- **Geschichtlicher Rückblick**
  - ... in das Jahr 1975
  - ... in das Jahr 1987
  - ... in das Jahr 2000
- Zusammenfassung
- Fragen

Geschichtlicher Rückblick

- Vermeidung von Fehlern bei der Entwicklung von Software in unterschiedlichen Ansätzen
  - Anleitung zum Schreiben von korrekten Programmen
  - Grundlegende und Erfolg versprechende Neuerungen auf unterschiedlichen Gebieten
  - Radikales Ändern von Erwartungen und Annahmen

- Diese drei Ansätze aus sehr unterschiedlichen Zeiten werden näher betrachtet und sollen im Anschluss diskutiert werden.

Übersicht

- Einführung in das Thema
- Geschichtlicher Rückblick
  - **… in das Jahr 1975**
  - … in das Jahr 1987
  - … in das Jahr 2000
- Zusammenfassung
- Fragen

Rückblick: 1975

- Harlan D. Mills: „How to write correct programs and know it“
- Abhandlung über das Schreiben von defektfreier Software.
- Vorhersage: „Der professionelle Programmierer von morgen wird sich, mehr oder weniger lebhaft, an jeden seiner Fehler in seiner Karriere erinnern können.“

Fehlerfreie Programme: Aber wie?

Freie Universität Berlin

- Voraussetzungen:
  - Klare Spezifikation
  - Bekannte Hardware (Prozessor etc.)
  - Das Wissen, dass man korrekte Programme schreiben kann
  - Das Wissen, dass wenn man ein korrektes Programm geschrieben hat, dies nicht für immer und ewig korrekt sein wird
  - Sich des Unterschieds zwischen Korrektheit (correctness) und Fähigkeit (capability) bewusst sein

Fehlerfreie Programme: So erreicht man sein Ziel

Freie Universität Berlin

- Funktionen benutzen
- Strukturiertes Programmieren
  - devide-and-conquer
  - kein GOTO
- Vermeidung von Syntaxfehlern

Fehlerfreie Programme: Wirklich?

- Gute Vorschläge, aber entstehen damit defektfreie Programme?
- Strukturiertes Programmieren ist an der Tagesordnung und muss nicht extra propagiert werden.
- Höhere Programmiersprachen setzen teilweise an diesem Gebiet an.

Übersicht

- Einführung in das Thema
- Geschichtlicher Rückblick
  - … in das Jahr 1975
  - **… in das Jahr 1987**
  - … in das Jahr 2000
- Zusammenfassung
- Fragen

Rückblick: 1987

- Frederick P. Brooks, Jr.: “No Silver Bullet: Essence and Accidents of Software Engineering“
- Brooks untersucht in seiner Abhandlung grundlegende Probleme bei der Softwareentwicklung
- Sein Aufhänger sind Werwölfe, die man nur mit silbernen Kugeln und damit auf magische Weise besiegen kann.

Desillusionierung, leider!

- Dieser magische Gegenstand existiert nicht, vor allem nicht in der Software-Welt
- Fortschritte werden, wie das Wort schon sagt, nur in Schritten erzielt, man braucht dazu große Anstrengungen, Hartnäckigkeit und Disziplin
  - Wie sonst überall auch

Grundlegende Probleme: Übersicht

- Komplexität
- Konformität
- Änderbarkeit
- Unsichtbarkeit

Problem: Komplexität

- Komplexität
  - Komplexität der einzelnen Komponenten
  - Komplexität durch Interaktion der Komponenten
- Folgen:
  - Das Risiko, etwas zu übersehen steigt. Es kommt zu Unzuverlässigkeiten
  - Gute Benutzbarkeit wird fraglich
  - Erweiterbarkeit wird kritisch (side effects)
  - Probleme bei der Kommunikation zwischen Teamkollegen, Produktfehlern, Kostenexplosion, Zeitknappheit, Managementprobleme (Überblick über Projekt)
  - u.v.m.

Problem: Konformität

- Konformität zu bereits bestehenden Interfaces
- Konformität der Benutzeroberfläche
  - z.B. durch Vorgänger-Versionen, ähnliche Produkte
- Folgen:
  - Erhöhte Komplexität

Problem: Änderbarkeit

Freie Universität Berlin

Neue Anwendungsmöglichkeiten

- Erfolgreiche Software wird ständig geändert
  - Anwender benutzen die Software anders als gedacht, oder abseits der eigentlichen Domäne
  - Anwender finden die Software zwar gut, haben aber zusätzlich noch Wünsche

Anpassungs-Änderungen

- Erfolgreiche Software wird ständig geändert,
  - Lebenszyklus der ursprünglichen Maschine ist kürzer
  - Neue oder geänderte Peripherie-Geräte

- Folgen: Erhöhte Komplexität

Problem: Unsichtbarkeit

- Software ist unsichtbar und nicht sichtbar zu machen
  - Struktur durch viele, uneinheitliche Diagramme repräsentiert
  - Folgen: Für eine Person alleine ist es fast unmöglich, das Design zu gestalten.

Durchbrüche in der Vergangenheit

Freie Universität Berlin

- Höhere Programmiersprachen
- Standardisierte Programmierumgebung
- ABER: Wie bereits gesagt, diese Durchbrüche waren nur bei nebensächlichen Problemen zu erzielen.

Potenzielle magische Waffen

Freie Universität Berlin

- Fortschritt bei den höheren Programmiersprachen
- Objektorientiertes Programmieren
- AI: Expertensysteme
- Graphisches Programmieren
- Verifikation
- Umgebungen und Werkzeuge
- Workstations

Viel versprechende Ansätze nach Brooks

Freie Universität Berlin

- Buy or build
- Anforderungen verfeinern und Prototyping
- Großartiges Design und großartige Designer

Übersicht

- Einführung in das Thema
- Geschichtlicher Rückblick
  - … in das Jahr 1975
  - … in das Jahr 1987
  - **… in das Jahr 2000**
- Zusammenfassung
- Fragen

Rückblick: 2000

- Boris Beizer: „Software is different“
- Plädoyer für eine Änderung des Denkens.
- Software als eigenständige Disziplin, unabhängig von anderen Ingenieurwissenschaften
- Aber: Nicht alles gleich wegwerfen, sondern auch aus vergangenem lernen.

Falsche Annahmen: Übersicht

- Physikalische Einschränkungen
- Lokalitätsprinzip
- Komplexität
- Qualität

Problem: Physikalische Schranken

- Software gehorcht nicht den physikalischen Gesetzen
- Software ist eher verwandt mit der psychischen Welt des menschlichen Verhaltens
- Die erste falsche Annahme, die man fallen lassen muss

Problem: Lokalität

- Raum-Abhängigkeit
  - Die Symptome eines Defekts können beliebig weit entfernt sein von der Ursache
    - Hat man am Auto Probleme mit den Scheibenwischern, erwartet man nicht, dass sich dies auf das Radio auswirkt
- Zeit-Abhängigkeit
  - Die Symptome eines Defekts können beliebig lange nach Ausführung des fehlerhaften Codes auftreten.
    - Platzt ein Reifen am Auto, erwartet man, dass das Auto sofort nach einer Seite ausbricht, nicht Stunden oder Tage später
- Konsequenzen und Proportionalität
  - Die Konsequenzen eines Defekts können willkürlich mit der Ursache in Beziehung stehen
    - Die Reifen am Auto verlieren nach und nach an Profil, nicht auf einmal. Außerdem explodieren sie nicht, sobald das Profil unter 4,3 mm tief ist

Problem: Komplexität

- Emergente Probleme
- Proportionale Komplexität
- Komplexität durch Funktionalität
- Sicherheitslimits

Problem: Qualität

- Aber Qualität wovon, wie stellt man Qualität fest?
  - Defekte pro Zeile Code
  - Defektfindungs–History
  - Defektrate von Defekten, die den Anwendern aufgefallen sind
  - Qualität ist auch ein Komplexitätsproblem, da Qualität auch vom emergenten Verhalten abhängig ist und damit nur schwer messbar.

Lösungen: Übersicht

- Die Welt in der wir leben
  - Entwickler
  - Management
  - User
- Ehrlichkeit

Lösungen: Entwickler

- In der Welt in der wir leben gibt es keine einfachen Lösungen.
  - Man kann jedoch versuchen, die Denkmuster zu ändern
  - Änderungen aus Entwicklersicht:
    - Akzeptieren und Anwenden von Design-Einschränkungen
    - z.B. Strukturiertes Programmieren, Strong typing (starke Typisierung), Vermeidung von globalen Daten, Style-Regeln und Style-Checker, Kapselung

Lösungen: Management

Prioritäten für die Softwareentwicklung sehen im Moment ungefähr so aus:

- Entwickle so schnell wie möglich
- Programmgeschwindigkeit so hoch wie möglich
- Baue das System so kompakt wie möglich
- Baue so viele Features ein, wie du nur kannst
- Baue es so billig wie möglich
- Sorge dich später. Bugs will get fixed

- Vorschlag von Fragen die bei der Entwicklung eine Rolle spielen sollten:
  - Kann man das System analysieren? Verhält es sich vorhersehbar?
  - Kann man es testen?
  - Funktioniert es?
  - Sind Features, Komponenten und Dateninteraktion auf das absolute Minimum reduziert?
  - Hat es die Features, die der Benutzer braucht (anstelle derer, die er will)?

Lösungen: Benutzer

- Benutzer-Sicht
  - Qualität statt Quantität
  - Nicht sinnvolle Features
  - Das ist aber alles ziemlich schwierig!

Lösungen: Ehrlichkeit

- Ehrlichkeit
  - Die Entwickler müssen sich und auch anderen gegenüber die Wahrheit eingestehen.
  - Wenn man (noch) nicht weiß, wie etwas funktioniert oder wann es funktioniert oder ob es funktioniert, dann muss man das auch sagen.
  - Wenn Marketing, Management und Benutzer überzogene Vorstellungen haben ist man in solch einem Fall selbst dran Schuld.

Übersicht

- Einführung in das Thema
- Geschichtlicher Rückblick
  - ... in das Jahr 1975
  - ... in das Jahr 1987
  - ... in das Jahr 2000
- **Zusammenfassung**
- Fragen

Zusammenfassung

- Die Komplexität von Software hat sich als größtes Problem im Entwicklungsprozess herausgestellt
- Verringert man die Komplexität, etwa durch neue Konzepte, stehen mehr Möglichkeiten zur Verfügung, die auch ausgenutzt werden
- Dadurch steigt jedoch wieder der Grad an Komplexität
- Und diese Spirale dreht sich und dreht sich und dreht sich ...
- Wie man im Vorfeld Fehler vermeiden kann wurde bisher nicht geklärt, aber erste Ansatzpunkte wurden geliefert.

Übersicht

Freie Universität Berlin

- Einführung in das Thema
- Geschichtlicher Rückblick
  - ... in das Jahr 1975
  - ... in das Jahr 1987
  - ... in das Jahr 2000
- Zusammenfassung
- **Fragen**

Fragen? Diskussion!

- Wie brauchbar sind die angestrebten Denkänderungen von Beizer? Tragen sie wirklich zu einer fehlerfreieren Software bei?
- Werden Ansätze geboten, wenn die Software wirklich mal schnell fertig werden muss?
- Wie bereit ist man selbst, diese veränderte Wahrnehmung anzunehmen und die Verhaltensmaßnahmen einzuhalten? Sie eventuell auch einem späteren Chef gegenüber durchzusetzen oder einzuhalten?
- Führt eine gute Architektur oder gutes Design automatisch zu guter Software?
- Spielt gutes Management eine Rolle für gute Software?

Danke!