

XION IT SYSTEMS

AKTIENGESELLSCHAFT

Dresdnerstraße 81-85/8.Stock
A-1200 Wien

Tel: 0664-8242-600

E-mail: office@xion.at

Web: xion.at

Festnetz: +43/1/333 91 99-0

Fax: +43/1/333 91 99-199

x i o n . i t systems . a g



Software Wartung und Evolution


Dipl.-Ing. Dr. techn. Johannes Weidl-Rektenwald
Xion IT Systems AG

Organisatorisches 1/2

- LVA Info
 - VU 2.0, 184.169
 - Wahlfach: 033 522, 066 922, 066 933, 066 937
- Vorlesungsteil
 - 7 VO Termine
 - 06.03., 13.03, 10.04., 24.04., 08.05., 15.05., 29.05.
 - Jeweils Donnerstag, 16:15 - 17:45, pünktlich!
 - EI 4 Reithoffer Hörsaal

Organisatorisches 2/2

- Übungsteil
 - 1 Übungsbeispiel
 - Gruppenarbeit
 - Abschlusspräsentation in der Xion
- Prüfung
 - Donnerstag, 19. Juni 2008, 16:15 – 17:15, EI 4
- VU Web Page
 - <http://www.infosys.tuwien.ac.at/Teaching/Courses/SWE/swe.html>

Application Services 

Maintenance

Wartung der Anwendungen verbraucht 83% des IT Budget

"Too much money for maintenance. With just 27% of 2004 spending planned for new development, maintenance costs are choking IT productivity."


Gartner, Executive Summary, September 2004

60% des IT-Budget geht auf für Legacy Wartung

"Between 60 and 80 percent of an average company's IT budget is spent on maintaining existing mainframe systems and applications."

Gartner Group, Analyst Report, March 2002

BCS / AS Application Services 11. November 2005 © 2005 IBM Corporation

Application Services 

Legacy

Legacy gibt es (fast) immer und überall

- 70% der weltweiten Geschäftsdaten werden mit COBOL prozessiert
- Es gibt über 200 Milliarden COBOL "Lines of Code"
- 30 Milliarden COBOL Transaktionen werden täglich prozessiert – mehr als alle Zugriffe auf Webseiten

Legacy Modernisierung ist eine strategische Notwendigkeit

"It is clear that new technology will no more replace legacy in total than legacy technology alone will satisfy the needs of today's Web-savvy users. The truth is that modernized legacy applications play a crucial role."

Giga, Analyst Report, September 2002

4 BCS / AS – Application Services 11. November 2005 © 2005 IBM Corporation

Inhalt der Vorlesung: Überblick

- Was versteht man unter Software Wartung / Software Evolution / Wartbarkeit?
- Was sind die speziellen Probleme?
- Welche adäquaten Technologien, Prozesse und Tools gibt es, um diesen zu begegnen?
- Was sind die Best Practices der Software Wartung?
- Wie managt man Software Wartung?

Inhalt der Vorlesung: Themen

- (1) *Software Wartung*: Motivation, Definition, Arten, Probleme
- (2) *Software Wartung*: Aspekte, Aktivitäten, Wartungskrise, Legacy Systeme, *Reverse Engineering*
- (3) *Restructuring*, *Re-Engineering*, *Organisation der Wartung*: Software Life Cycle Modelle
- (4) *Tool Demos*, *Organisation der Wartung*: Defect Tracking, Software Configuration Management, Produktivstellung
- (5) *Software Evolution*: E-type Programs, Laws of Software Evolution Explained, Change Patterns
- (6) *Spezielle Kapitel der Software Wartung*: Program Comprehension, Change Impact Analysis, Qualitätsmerkmal „Wartbarkeit“
- (7) *Best Practices*: Design for Change, MDA und Wartung, *Software Wartung im unternehmerischen Kontext*: Rollen, Gewährleistung, Software-Wartungsverträge

Chapter 1

- Inhalte
 - Motivation: Software Wartung und Evolution
 - Abgrenzung der Begriffe Software Wartung und Software Entwicklung
 - Definition „Software Wartung“
 - Arten der Software Wartung
 - Probleme der Software Wartung

Software Engineering vs. Software Maintenance

- Suche nach dem Begriff „Software Engineering“ bei amazon.de, Kategorie „Englische Bücher“
 - Treffer: 9218 (2007: 8405; 2006: 1510; 2005: 1405; 2004: 1078)
 - Top Treffer: „Software Engineering: Update (Ian Sommerville et al.; 2006)“
 - Treffer 2: „A Handbook of Software and Systems Engineering.“
 - Treffer 3: „Practical Software Engineering. Analysis and Design for the .NET Platform.“
- Suche nach „Software Maintenance“
 - Treffer: 403 (2007: 374; 2006: 90; 2005: 92; 2004: 92)
 - Top Treffer: „Effective Software Maintenance and Evolution: A Reuse-Based Approach“ (Stanislaw Jarzabek; 2007)
 - Top Treffer 2007: „Macs for Dummies“ (Baig; 2006)
 - Top Treffer 2006: „Troubleshooting your PC“ (1994)
- Search for „Software Evolution“
 - Treffer: 91 (2007: 81; 2006: 23; 2005: 20; 2004: 15)
 - Top Treffer: „Software Evolution and Feedback. Theory and Practice.: Theory and Practice“ (Nazim H. Madhavji, M. M. Lehmann, und J. Ramil; 2006)
 - Top Treffer 2007: „The Old New Thing. Practical Development Throughout the Evolution of Windows“ (Chen et al.; 2007)
 - Treffer #7/2007: „Successful Evolution of Software Systems“ (Yang et al.; 2003)
 - Top Treffer 2006: „Software Engineering: Evolution And Emerging Technologies: 130“ von K. Zielinski (26. Dez. 2005!)

© J. Weidl-Rektenwald 02-08

9

Annäherung an den Begriff „Software Wartung“

- Software Wartung hat mit dem geläufigen Begriff der technischen Wartung wenig gemein
 - Software hat keine Verschleißteile
 - Software zeigt keine Abnutzungserscheinungen („wear-out“)

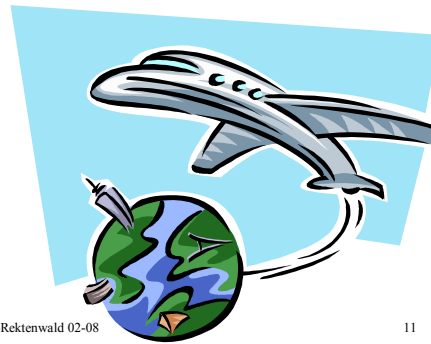


© J. Weidl-Rektenwald 02-08

10

Annäherung an den Begriff „Software Wartung“

- Software gilt im Gegensatz zu physischen technischen Artefakten als „leicht“ änderbar



© J. Weidl-Rektenwald 02-08

11

Software ist leicht änderbar?

```
/**
 * Wrong name: should be getRechnungToAuftragIDAndKeyAccountID
 */
public static Rechnung getRechnungToAuftragIDAndPersonID (Integer iAID,
    Integer iKAID) throws ExceptionPersist {

    Vector v;
    Rechnung r = new Rechnung();
    r.setAuftragID(iAID);
    r.setKundenID(iKAID);
    v = r.search();

    [...]
```

© J. Weidl-Rektenwald 02-08

12

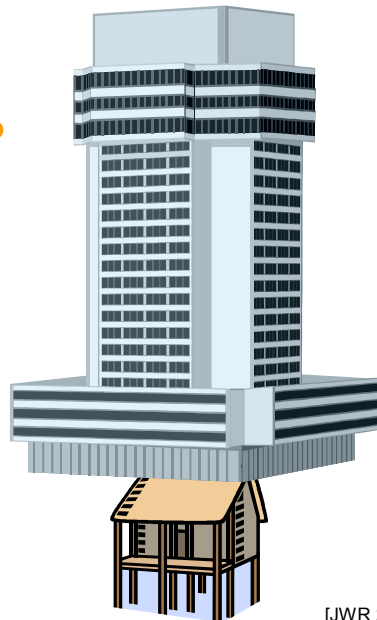
Software ist leicht änderbar?

```
for(int j=0;j<vtz.size();j++) {  
  
    tzkbez = ((DtZeile)vtz.elementAt(j)).getDtKurzBezeichnung();  
    iZeichPos = tzkbez.indexOf(':');  
  
    if (tzkbez.substring(0,iZeichPos).equals(tzkbez.substring(iZeichPos+1)))  
        ivSortDt = 1;  
    else if (tzkbez.substring(0,iZeichPos).compareTo(tzkbez.substring(iZeichPos+1))>0)  
        ivSortDt = 0;  
    else  
        ivSortDt = 2;  
  
    if (ivSortDt == 0) iZeichPos = 0;  
    else iZeichPos++;  
  
    int i;  
    for(i=0;i<vSortDt[ivSortDt].size();i++) {  
        if  
            (((DtZeile)vSortDt[ivSortDt].elementAt(i)).getDtKurzBezeichnung().substring(iZeichPos).compareTo(tzkbez.substring(iZeichPos))>0)  
            break;  
        }  
  
    if (i<vSortDt[ivSortDt].size())  
        vSortDt[ivSortDt].insertElementAt(vtz.elementAt(j),i);  
    else  
        vSortDt[ivSortDt].add(vtz.elementAt(j));  
}  
}
```

© J. Weidl-Rektenwald 02-08

13

Software ist leicht änderbar?



[JWR 2002]

© J. Weidl-Rektenwald 02-08

14

Annäherung an den Begriff „Software Wartung“

- „Programs, like people, get old“
- „Software aging will occur in all successful products“
 - David Lorge Parnas in „Software Aging“ [Parnas 1994]
- Es ist einsichtig, dass, was altert, irgendwie up-to-date gehalten werden muss.
- Die Frage ist: Wie altert Software und warum?

© J. Weidl-Rektenwald 02-08

15

Two Causes of Software Aging

- The first is caused by the failure of the product's owners to modify it to meet changing needs
 - „Lack of movement“
- The second is the result of the changes that are made
 - „Ignorant surgery“

[D. L. Parnas, 1994]

© J. Weidl-Rektenwald 02-08

16

Symptoms and Costs of Software Aging

- Inability to keep up
 - “As software ages, it grows bigger“
 - More code to change
 - More difficult to find routines that must be changed
- Reduced performance
 - More machine resources are needed
 - Poor design causes performance bottlenecks
- Decreasing reliability
 - „As the software is maintained, errors are introduced“

[D. L. Parnas, 1994]

© J. Weidl-Rektenwald 02-08

17

Preventive Medicine

- Design for success (aka „Design for change“)
 - Information hiding, abstraction, **separation of concerns** (SOC), data hiding, object orientation, ...
- Documentation
 - Problem: Most documentation is ignored because not being accurate
- Second opinions – Reviews
 - Reviews often are neglected because of time pressure

[D. L. Parnas, 1994]

© J. Weidl-Rektenwald 02-08

18

„Software Geriatrics“

- Stopping deterioration
 - Requires techniques and resources!
- Retroactive documentation
 - Lack of formal basis and influence of short-term interests
- Retroactive incremental modularization
- Amputation
- Restructuring

[D. L. Parnas, 1994]

© J. Weidl-Rektenwald 02-08

19

Software Wartung vs. Software Evolution

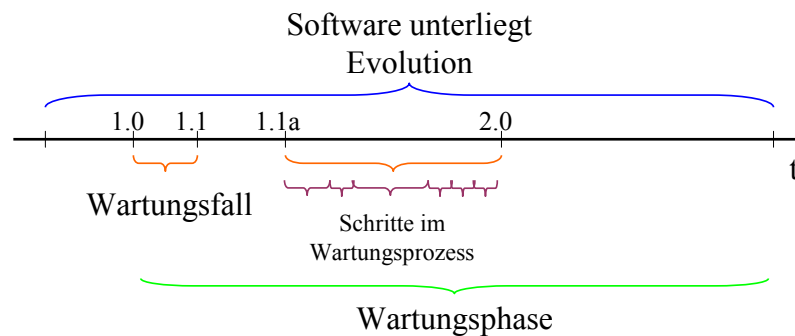
- Software Wartung
 - bezeichnet als **Tätigkeit** eine Änderung an einem Softwaresystem nach dessen Auslieferung (die Durchführung einer Änderung bezeichnet man als „Wartungsfall“)
 - bezeichnet als **Prozess** die Schritte, die in einem Wartungsfall sequentiell durchzuführen sind
 - bezeichnet als **Phase** den Abschnitt des Lebenszyklus eines Softwaresystems von dessen Auslieferung bis zur Stilllegung
- Software Evolution
 - bezeichnet den **Prozess** der Veränderung eines Softwaresystems von der Erstellung bis zur Stilllegung
 - umfasst: Entwicklung, Wartung, Migration, Stilllegung

[JWR 2002]

© J. Weidl-Rektenwald 02-08

20

Software Wartung vs. Software Evolution



Warum Evolution?

- Viele Software Systeme bilden Geschäftsprozesse der realen Welt nach
- Geschäftsprozesse unterliegen ständigen Änderungen
 - passiv durch Adaption an neue Gegebenheiten (neue rechtliche Gegebenheiten, Marktsituation, Euro, Basel II, ...)
 - aktiv durch die Einführung neuer Produkte, neuer Prozesse (Business Process Reengineering (BPR))
- Daher muss die Software laufend an die sich ändernden Geschäftsprozesse angepasst werden

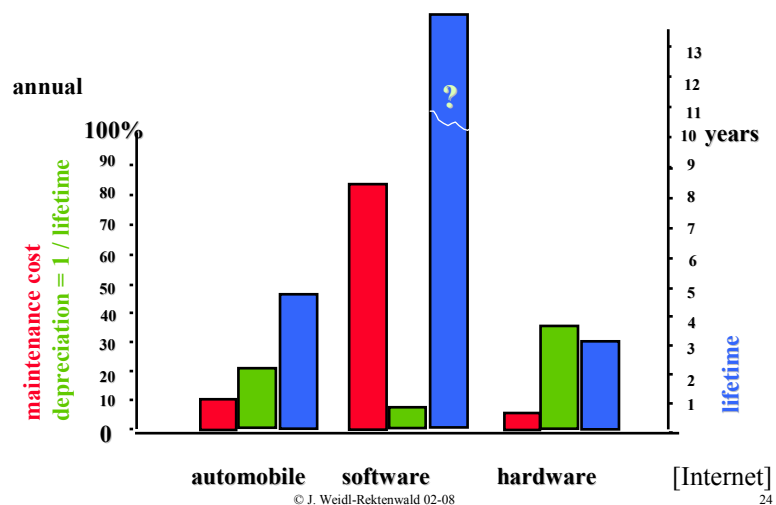
Warum Beschäftigung mit Software Wartung/Evolution?

- „Nevertheless, the industrial track record raises the question, why, despite so many advances, [...]
 - satisfactory functionality, performance and quality is only achieved over a *lengthy evolutionary process*,
 - software maintenance *never ceases* until a system is scrapped
 - software is still generally regarded as the *weakest link* in the development of computer-based systems“.
- [Lehman et al., 1997]

© J. Weidl-Rektenwald 02-08

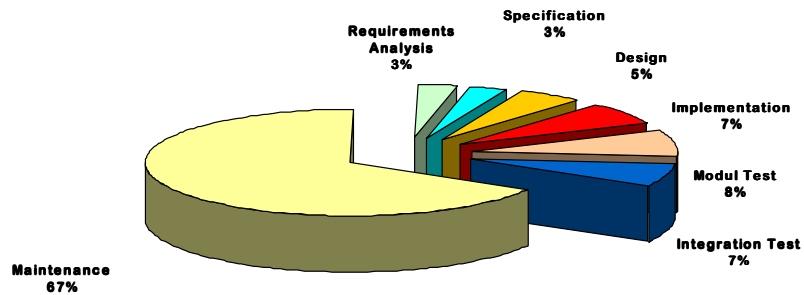
23

Software Wartung im Vergleich



24

Cost Distribution in the Software Life-Cycle

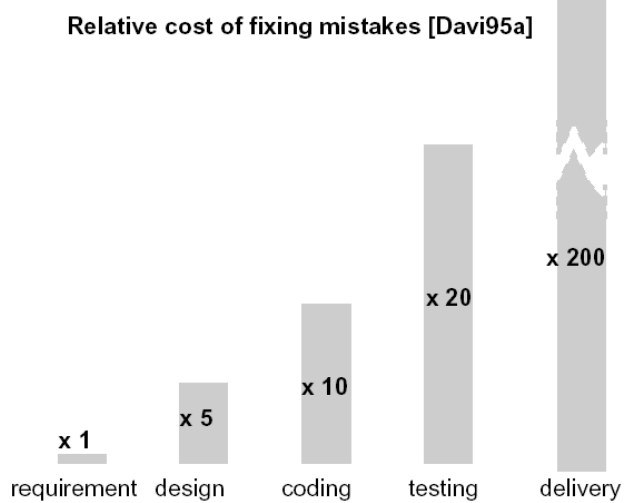


Cost Distribution in the Software-Life-Cycle

Source: Principles of Software Engineering and Design, Zelkovits, Shaw, Gannon 1979
© J. Weidl-Rektenwald 02-08

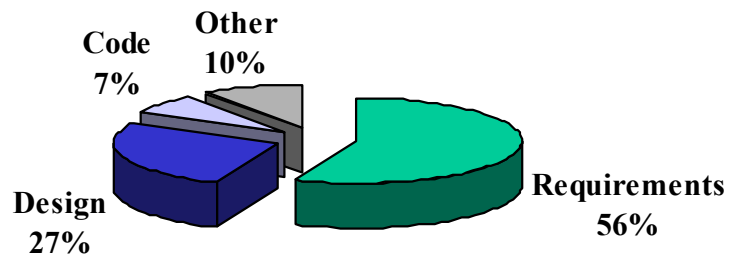
25

Cost of fixing bugs per phase



26

Distribution of Bugs



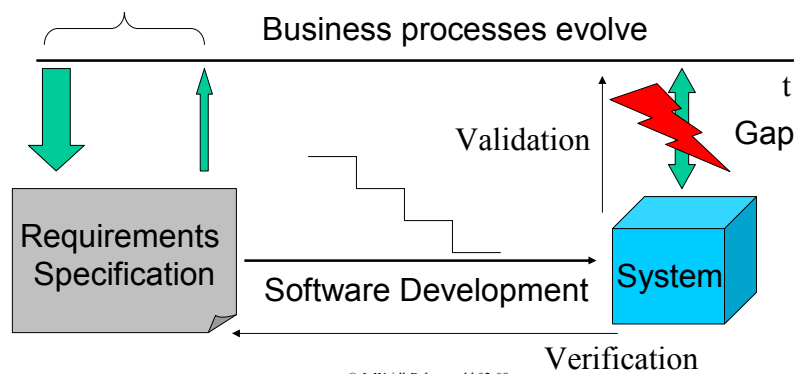
© J. Weidl-Rektenwald 02-08

Source: Klösch/Gall,
„Objektorientiertes Reverse
Engineering“, Springer

27

Classic Approach to Software Development

„Requirements Fixing“



© J. Weidl-Rektenwald 02-08

28

Difference software maintenance vs. software development

- Maintenance is similar to software development
 - Some unique skills and processes are employed:
 - Have intimate knowledge of system structure and content
 - Perform impact analysis and know the ripple effect
 - Problem solving skills
 - „Programmers have become part historian, part detective, and part clairvoyant.“ (Corbi 1989)
 - Track and control changes
 - Maintenance Outsourcing
 - Maintenance Cost Estimation

[Internet]

© J. Weidl-Rektenwald 02-08

29

Software Wartung: Definition

Definition: Software Wartung

- *Nach IEEE Std. 610.12-1990 bzw. IEEE Std. 1219-1998*
- Software Wartung ist die Modifikation eines Software-Produktes oder einer Komponente, nach der Auslieferung, mit dem Zweck
 - der Fehlerkorrektur
 - der Verbesserung der Performance oder anderer Systemattribute
 - der Adaptierung an eine geänderte Umgebung

© J. Weidl-Rektenwald 02-08

31

Definition: Software Wartung

- *Nach Barry Boehm*
 - “The process of modifying existing operational software while leaving its primary function intact”
- *Abstrakt*
 - „Preserve the value of software over time“
- Center for Software Maintenance
 - **Software maintenance** is the set of activities, both technical and managerial, that ensures that software continues to meet organizational and business objectives in a cost-effective way.

© J. Weidl-Rektenwald 02-08

32

Allgemeinere Definition [SWEBOK]

- Software Maintenance
 - The totality of activities required to provide cost-effective support to a software system.
 - Activities are performed during the predelivery stage as well as the postdelivery stage.
 - Predelivery activities include planning for the postdelivery operations, supportability, and logistics determination.
 - Postdelivery activities include software modifications, training, and operating a help desk.

© J. Weidl-Rektenwald 02-08

Source: *Software Engineering Body of Knowledge*,
<http://www.swebok.org/> 33

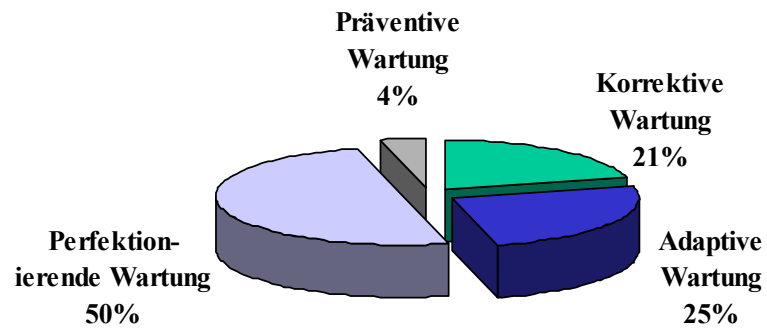
Arten der Software Wartung

- ❶ Korrektive Wartung
 - „Bug fixing“; reaktive Natur
- ❷ Präventive Wartung
 - Finden von latenten Fehlern, bevor sie effektive Fehler werden
- ❸ Adaptive Wartung
 - Neue Hardware, Betriebssysteme etc.; neue Anforderungen
- ❹ Perfektionierende Wartung
 - Verbesserungen von Performance und Wartbarkeit (Restructuring, Reverse Engineering, Dokumentationspflege, etc.)
- ❶ + ❷: Corrections / ❸ + ❹: Enhancements

© J. Weidl-Rektenwald 02-08

Source: *Klösch/Gall*,
„Objektorientiertes Reverse Engineering“, Springer 34

Arten der Software Wartung



© J. Weidl-Rektenwald 02-08

Source: Klösch/Gall,
„Objektorientiertes Reverse
Engineering“, Springer

35

Warum ist Software Wartung nicht-trivial?

Motivation

Typische Eigenschaften von Software

- Programme sind nur selten in sinnvolle **Modulstrukturen** aufgeteilt, und wenn, ist diese Aufteilung meist sehr willkürlich
- **Redundanzen** in Daten bzw. Funktionen sind ein steter Bestandteil eines Programms
- Die **Sichtbarkeitsbereiche** von Daten und Funktionen sind meist weiter ausgedehnt als für das Programm notwendig bzw. überhaupt sinnvoll

© J. Weidl-Rektenwald 02-08

37

Typische Eigenschaften von Software

- **Trace Ausgaben** sind spärlich, haben keinen Timestamp und keine Quellenangabe
- Durch Änderungen am Code verändert sich das Laufzeitverhalten durch **Gleichzeitigkeitsprobleme** („race conditions“) nichtdeterministisch
- Dieselbe **Methode** wird durch ein Flag für unterschiedliche Berechnungen genutzt
- **Methoden bleiben bei der Klasse** für die sie ursprünglich geschrieben wurden, bei einer Änderung der Funktionalität werden sie nicht zur am Besten geeigneten Klasse verschoben

© J. Weidl-Rektenwald 02-08

38

Typische Eigenschaften von Software

- Variablennamen sind semantisch wertlos
 - `int work = 0;`
- Variablen werden **kontext-abhängig** verwendet
 - Fall 1: work speichert Kundennummer
 - Fall 2: work speichert Faktorensomme
- Die **Dynamik** des Programmdurchlaufs ist während der statischen Code Inspektion nicht oder nur schwer ableitbar

© J. Weidl-Rektenwald 02-08

39

Schwierigkeiten in der Software Wartung

- Missing
 - Development environment (tools, scripts, etc.)
 - Build environment
 - Source code
 - Documentation
 - Design Decisions
 - Domain knowledge
 - Original programmer team / analysts

© J. Weidl-Rektenwald 02-08

40

Schwierigkeiten in der Software Wartung

- Wartung ist **ereignisgesteuert** (planbar?)
- In der korrektiven Wartung wird nach Meldung eines Fehlers verlangt, die Ursache so schnell als möglich zu finden und den Fehler zu beseitigen (also „**quick fix**“!), trotz des weiterlaufenden Tagesgeschäftes
- Das Wartungspersonal steht daher meist unter **Zeitdruck** und das Management und die Dokumentation des Wartungsfalls werden vernachlässigt

© J. Weidl-Rektenwald 02-08

41

Schwierigkeiten in der Software Wartung

- Laufende Wartung erhöht die „Software Entropie“
 - Verklärt
 - Architektur
 - Design
 - Modularisierung
 - Erhöht
 - Abhängigkeiten („Coupling“)
 - Vermindert
 - Orthogonale Trennung („Cohesion“)

© J. Weidl-Rektenwald 02-08

42

Schwierigkeiten in der Software Wartung

- Änderungen an einem Software System sind zwar operativ leicht durchführbar, die Schwierigkeit ist aber, **die richtige Änderung durchzuführen** - und **nur** diese.

Schwierigkeiten in der Software Wartung

- Viele Probleme der Software Wartung treten erst im Zusammenhang mit großen, alten, komplexen Software Systemen auf, so genannten „*Legacy Systemen*“.
- Diese wurden mit Methoden konzipiert und in Sprachen erstellt, die heute kaum mehr benutzt (und noch weniger gelehrt) werden
 - Strukturierte Analyse, proprietäre Analyseansätze
 - Mumps, FORTRAN, PL/I

State-of-the-Art: „7x24“

- 7 Tage in der Woche 24 Stunden online
- Hardware Hersteller werben mit **99,999** Prozent Verfügbarkeit ihrer Systeme (entspricht 5 Minuten 20 Sekunden Downtime im Jahr)
- **Wann** werden dann neue Versionen eingespielt?
- Trend zu Applikationsservern, die Wartung **zur Laufzeit** unterstützen
 - 2. Instanz mit adaptierter Software fährt hoch
 - Die 2. Instanz übernimmt zur Laufzeit
 - Die ursprüngliche Instanz geht außer Betrieb
 - Problem der Datenmigration!

© J. Weidl-Rektenwald 02-08

45

Geschichte der Software Wartung

- Die Wartung von Programmen galt von jeher als *unbeliebte* Tätigkeit im Software Life-Cycle
 - Historisch die Arbeit von neu rekrutierten bzw. nicht so erfahrenen Programmierern
- Das heißt
 - **wenig Know-How**
 - **keine Werkzeuge**
- Folgen
 - **„quick fix“ Modell wird angewandt**
 - **Wartung führt zu noch mehr Fehlern**
 - **Wartung kann aufgrund von steigender Komplexität gar nicht mehr durchgeführt werden**

© J. Weidl-Rektenwald 02-08

46

Evolutionäre Probleme der Software Wartung

- Die Komplexität wächst mit jedem Wartungseingriff
- Daher vergeht zwischen den Wartungseingriffen immer mehr Zeit
- Die Produktivität der Wartungseingriffe sinkt, die Kosten pro Eingriff steigen
- Dies alles geschieht nach Gesetzmäßigkeiten („**Laws of Software Evolution**“)

© J. Weidl-Rektenwald 02-08

47

Das Pareto Prinzip im Software Engineering bzw. in der Wartung

- 20% der Requirements bedingen 80% der Komplexität
- 80% des Systems sind in 20% der Zeit fertig gestellt
- 20% des Codes beinhalten 80% der Fehler
- 80% der Fehler werden in 20% der Zeit behoben
- Nach Vilfredo Pareto (1848-1923)
 - Italienischer Ökonom und Gesellschaftstheoretiker
 - Dieses Prinzip besitzt auch in vielen anderen Bereichen Gültigkeit (z.B. Zeitmanagement, Verkauf)

© J. Weidl-Rektenwald 02-08

48

POEM - David H. H. Diamond


- The fellow who designed it,
Is working far away;
The spec's not been updated,
For many a livelong day.
- They haven't kept the flowcharts,
The manual's a mess,
And most of what you need to
know
You'll simply have to guess.
- The guy who implemented it is
Promoted up the line;
And some of the
enhancements
Didn't match to the design.
- We do not know the reason,
Why the bugs pour in like rain,
But don't just stand here gaping,
Get out there and MAINTAIN.

© J. Weidl-Rektenwald 02-08

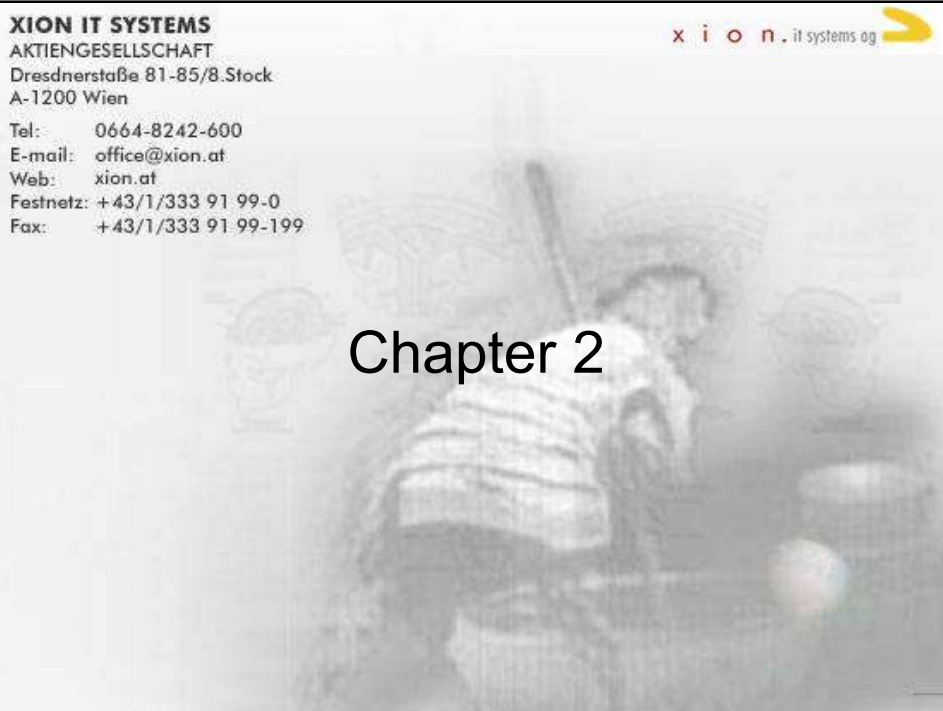
49

XION IT SYSTEMS
AKTIENGESELLSCHAFT
Dresdnerstraße 81-85/8.Stock
A-1200 Wien

Tel: 0664-8242-600
E-mail: office@xion.at
Web: xion.at
Festnetz: +43/1/333 91 99-0
Fax: +43/1/333 91 99-199

x i o n . it systems og 

Chapter 2



Chapter 2

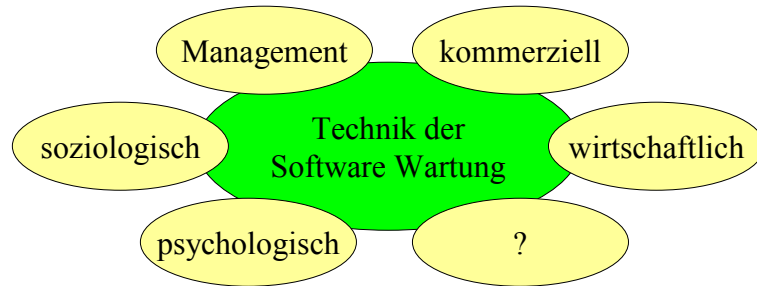
- Inhalte
 - Aspekte der Software Wartung
 - Aktivitäten der Software Wartung
 - Software Wartungskrise
 - Legacy Systeme
 - Reverse Engineering
 - Redocumentation
 - Design Recovery

Aspekte der Software Wartung

Technik der
Software Wartung

... und das war's?

Aspekte der Software Wartung



- Berücksichtigung aller Aspekte führt zur *holistischen* Betrachtung von Software Wartung

© J. Weidl-Rektenwald 02-08

53

Aspekte der Software Wartung

- Technical
 - Understanding existing code
- Managerial
 - Reactive work context
- Economic
 - Justifying remedial work
- Commercial
 - How to estimate the effort of a change request?

© J. Weidl-Rektenwald 02-08

54

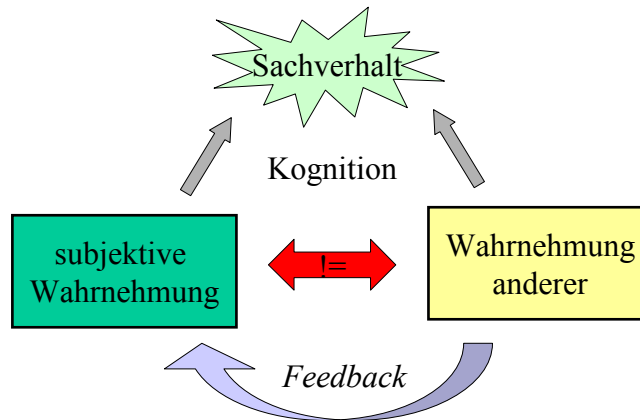
Aspekte der Software Wartung

- Sociological
 - Second hand / apprentice's work
 - Maintaining morale
- Psychological
 - Hesitate to touch a huge work of art
 - Fear to damage a working system constantly
 - „Kognitive Dissonanz“

Kognitive Dissonanz

- **Definition (nach L. FESTINGER)**
 - K. D. ist die Bezeichnung für einen unangenehm erlebten Zustand, der aus **widersprüchlichen Erfahrungen oder Einstellungen** in Bezug auf den gleichen Gegenstand hervorgeht und nach einer Veränderung verlangt.
 - Um den als unangenehm erlebten Zustand aufzulösen, müssen neue, **spannungslösende Schritte** eingeleitet werden. Dies lässt sich auf mehrere Arten erreichen, z.B. durch die Suche nach neuen **Informationen**, durch eine Einstellungsänderung, durch ein Handlungsänderung usw.
 - Die Theorie der k. D. beschäftigt sich mit der Verarbeitung von wichtigen Informationen, nachdem eine Entscheidung gefällt wurde.
 - Die Theorie besagt, dass nach einer getroffenen Entscheidung vorzugsweise die Informationen ausgewählt werden, die die Entscheidung als richtig erscheinen lassen, und dass gegenteilige Informationen nicht mehr beachtet werden.

Kognitive Dissonanz



© J. Weidl-Rektenwald 02-08

57

Kognitive Dissonanz: Beispiel

- Sachverhalt: **Die Software funktioniert nicht so, wie sie sollte**
- **Wahrnehmung Programmierer:** „Ich bin mit dem Programm fertig. Es funktioniert.“
- **Wahrnehmung Quality Assurance:** „Dieser Testfall funktioniert nicht.“
- **Feedback QA:** „In ihrem Programm ist ein Bug“
- -> kognitive Dissonanz: „Ich bin fertig und das (weniger das Ergebnis, sondern der Zustand) gefällt mir“ **versus** „Du hast Mist gebaut!“
- **Auflösung der KD:**
 - „Der Bug liegt sicher nicht bei mir! Fragen sie mal Kollegen X.“
 - „Es steht aber genau so in den Anforderungen! Lesen sie die mal!“
 - „Das ist kein Bug, die User haben es mir so erklärt.“
 - „Sie wissen ja nicht einmal, was ein Bug ist!“
- Kognitive Dissonanz führt also zu einer **Wirklichkeitskonstruktion**, die versucht, die Dissonanz aufzulösen

© J. Weidl-Rektenwald 02-08

58

Egoless Programming

- **Über-Identifikation** mit den Artefakten der Arbeit ist ein unerwünschter Nebeneffekt
- -> „Egoless Programming“
- Modern: Extreme Programming mit dem Leitfaden „Embrace Change“ [Kent Beck]
 - Ist psychologisches Problem, nicht rein technisches!

Beispiel für den psycho-sozialen Aspekt

- aus Petr Kroha „Softwaretechnologie“ [Kroha97]
- **Windows of Opportunity** [Yourdon92] - beschreiben günstigsten Zeitpunkt zur Neuentwicklung eines Softwaremoduls:
 - „der zuständige Programmierer geht in Rente oder kündigt
 - wenn die Komponente extrem gravierende Fehler aufweist und der Programmierer die Suche aufgegeben hat
 - *wenn es endlich möglich ist, den Widerstand leistenden Programmierer zu entlassen.“*

Kommerzieller Aspekt



Budget, Time-
to-market



Software
Qualität

© J. Weidl-Rektenwald 02-08

61

Aktivitäten der Software Wartung

Aktivitäten der Software Wartung

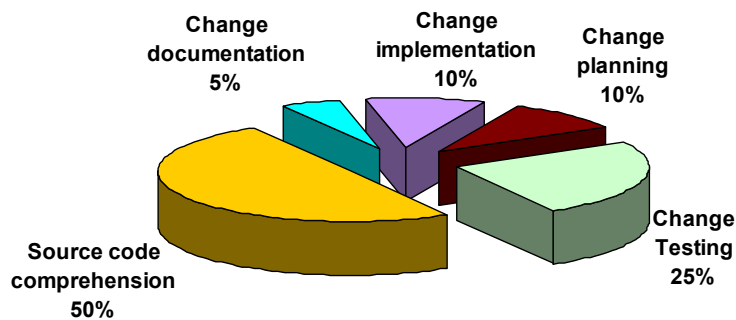
- Analyse bzw. Planung der Änderung
 - Verstehen des Systems, der „Architektur“
 - Source Code Verstehen, Bilden von Hypothesen
 - Verifizieren von Hypothesen
 - Change Impact Analysis
- Implementierung der Änderung
 - Restructuring
 - Change Propagation
- Verifikation und Validierung
- Re-Dokumentation

M
a
n
a
g
e
m
e
n
t

© J. Weidl-Rektenwald 02-08

63

Activities in Software Maintenance



Activities in Software Maintenance

Source: Principles of Software Engineering and Design, Zelkovits, Shaw, Gannon 1979

© J. Weidl-Rektenwald 02-08

64

Software Wartungskrise

Software Wartungskrise

- Unter der „Software Wartungskrise“ versteht man die **Unangemessenheit der Prozesse und Werkzeuge**, um den Vorgang der Software Wartung auch für komplexe, umfangreiche Systeme qualitativ hochwertig (ökonomisch und technisch) durchführen zu können.

Legacy Systeme

Eigenschaften von Legacy Systemen

- Größe: > 1 Mio. LOC
- Alter: > 10 Jahre
 - Verwendung veralteter Programmiersprache wie COBOL, FORTRAN, PL/I, ...
 - Verwendung veralteter Datenspeicherung, z.B.
 - Flat Files
 - Hierarchische Datenbanken
 - Dokumentation veraltet
- Strategische Bedeutung
 - Bilden kritische Geschäftsprozesse ab – Unternehmen ohne System nicht vorstellbar
 - 7x24 Verfügbarkeit

Eigenschaften von Legacy Systemen

- **Kosten**
 - Wartungskosten bei Legacy Systemen typischerweise über 95% der Gesamtkosten
- **Systemumgebung**
 - Limitierte Hardwareressourcen
 - Begrenzte Anbindungsmöglichkeiten an Kommunikationsprotokolle (Middleware - z.B. CORBA, ESB)
- **Komplexität**
 - Neue Anforderungen können im System nicht mehr verwirklicht werden
 - Unzufriedenstellende Performanz (z.B. Transaktionsrate)

© J. Weidl-Rektenwald 02-08

[Kroha97]

69

Probleme bei der Ablöse von Legacy Systemen

- Das neue System muss **funktional äquivalent** zum alten System sein
- Das neue System muss zusätzlich alle **aktuellen Anforderungen** implementieren
- Die **Daten** der Legacy Applikation müssen übernommen werden
- Die neue Applikation soll **State-of-the-art Techniken** verwenden (Datenbank, 3-Tier, Objektorientierung, Middleware, AOP, SOA, ESB, Clustering, etc.)
- Die **Ausfallszeit** durch die Ablöse soll minimal sein

© J. Weidl-Rektenwald 02-08

[Kroha97]

70

Gründe für das Scheitern von Legacy Neuimplementierungen

- Um die Mittel bewilligt zu bekommen, müssen **umfangreiche Erweiterungen** direkt mit der Migration versprochen werden
- Neuentwicklung dauert lange – in der Zwischenzeit **ändern** sich die Anforderungen
- Es existieren **versteckte Abhängigkeiten** von vielen Programmen zur Legacy Applikation
- **Politische** Einflüsse verhindern eine Fertigstellung
- Management von Software Großprojekten ist im Allgemeinen schwierig

© J. Weidl-Rektenwald 02-08

[Kroha97]

71

Migration von Legacy Systemen

- Meist inkrementell auf per-Modul Basis
 - Reverse Engineering
 - Restrukturierung und Reuse
 - Entwicklung neuer Komponenten
 - Integration
 - Datenmigration
 - Inbetriebnahme (Installation, Schulung, ...)

© J. Weidl-Rektenwald 02-08

[Kroha97]

72

Beispiel: Technologien Erste Bank

- 1990 waren bei der Ersten Bank Systeme in folgenden Technologien im Einsatz
 - Assembler
 - PL/I
 - Cobol
 - Fortran-TRX
 - TRX-GEN 1/2
 - „Entscheidungstabellen, DELTA, TIMESHARING, DMS1100“
 - » Aus: Spezielle Aspekte der Informationsverarbeitung in der Wirtschaft, W. Konvicka 1995

© J. Weidl-Rektenwald 02-08

73

Reverse Engineering

Reverse Engineering: Definition

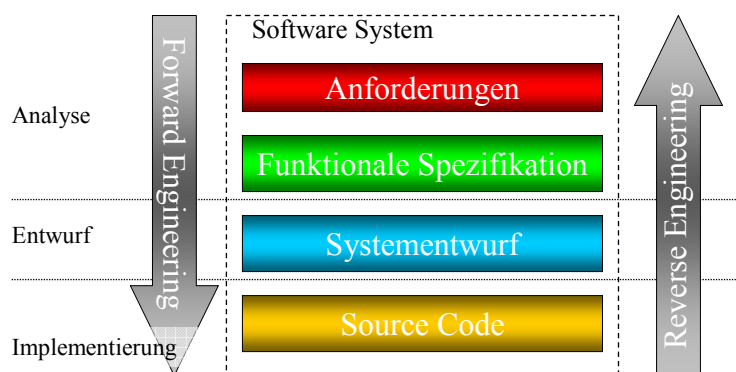
- Unter Reverse Engineering versteht man den Prozess der Analyse eines bestehenden Systems, mit dem Zweck
 - der Identifikation von **Systemkomponenten** und deren Beziehungen untereinander, sowie
 - der Erzeugung von **Darstellungen** des untersuchten Systems auf unterschiedlichen, höheren Abstraktionsstufen.

© J. Weidl-Rektenwald 02-08

[Chikofsky/Cross]

75

Forward- und Reverse Engineering

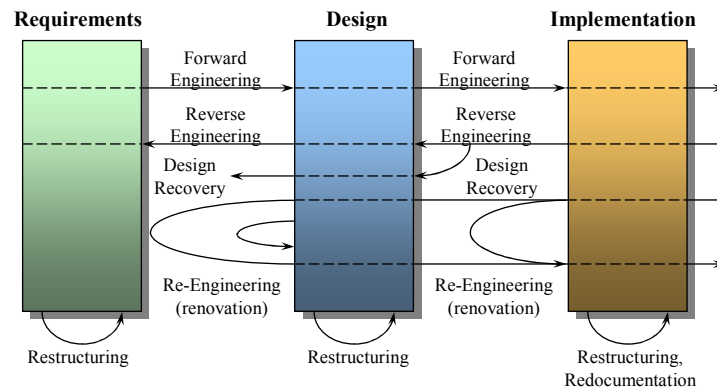


© J. Weidl-Rektenwald 02-08

[Klösch/Gall95]

76

Reverse Engineering Terminologie



© J. Weidl-Rektenwald 02-08

[Chikofsky/Cross90]

77

Motivation

- Ca. **75%** of software development time and expense goes toward **maintenance**
- Of this, ca. **50%** goes to **understanding** the software and the bug/enhancement
- Consequently, we want to devise tools and techniques to support improved understanding of software

© J. Weidl-Rektenwald 02-08

78

Difficulties

- Dimensions
 - application domain (what) and program (how)
- Information
 - Specific (machine-level) and abstract (design-level)
- Formality
 - Formal structure of programs and informal human understanding

Reverse Engineering verlangt Wissen

- Programmiersprache
 - Syntax
 - Semantik
- Programmierung (Algorithmen, Datenstrukturen, Patterns, ...)
- Application Domain („Domänenwissen“)

Reverse Engineering: Subprozesse

- Redocumentation
- Design Recovery

© J. Weidl-Rektenwald 02-08

81

Redocumentation

- Erzeugung oder Überarbeitung von **semantisch äquivalenten Repräsentationen** des Systems innerhalb desselben Abstraktionsniveaus
 - Datenfluss- und Kontrollflussdiagrammen, Cross Reference Listings, etc.
 - Automatisch generiert, meist ohne zusätzliche Informationen

© J. Weidl-Rektenwald 02-08

82

Design Recovery

- Ist ein Prozess, in dem **zusätzliche Information** verwendet wird, um Abstraktionen des Systems zu generieren u. zw. auf höheren Abstraktionsstufen
 - Wissen über Anwendungsdomäne, informale Beziehungen
 - Wissen von Applikationsexperten über Architektur, Modulstruktur, etc.

© J. Weidl-Rektenwald 02-08

83

Design Recovery: Repräsentationen

- Repräsentationen durch Design Recovery
 - Structure Charts
 - Nested Trees
 - Datenflussdiagramme
 - Kontrollflussdiagramme
 - Entity Relationship, ...
 - Informale Beschreibungen des Software Systems
 - Text
 - Diagramme
- Angereichert durch informale Information*

© J. Weidl-Rektenwald 02-08

84

Arten von Design Recovery

- Modellbasiert
 - Desire von Biggerstaff [Biggerstaff89]
- Wissensbasiert
 - Basierend auf zentraler Wissensbasis (Repository)
 - Extrahierung von Programmlichés
 - z.B. Sortieralgorithmen, Listen, Bäume
- Formale Methoden
 - Transformation von v.a. COBOL in Z oder Z++
 - siehe ESPRIT Projekt „REDO“
- Objektorientiert
 - Objekt Identifizierung

© J. Weidl-Rektenwald 02-08

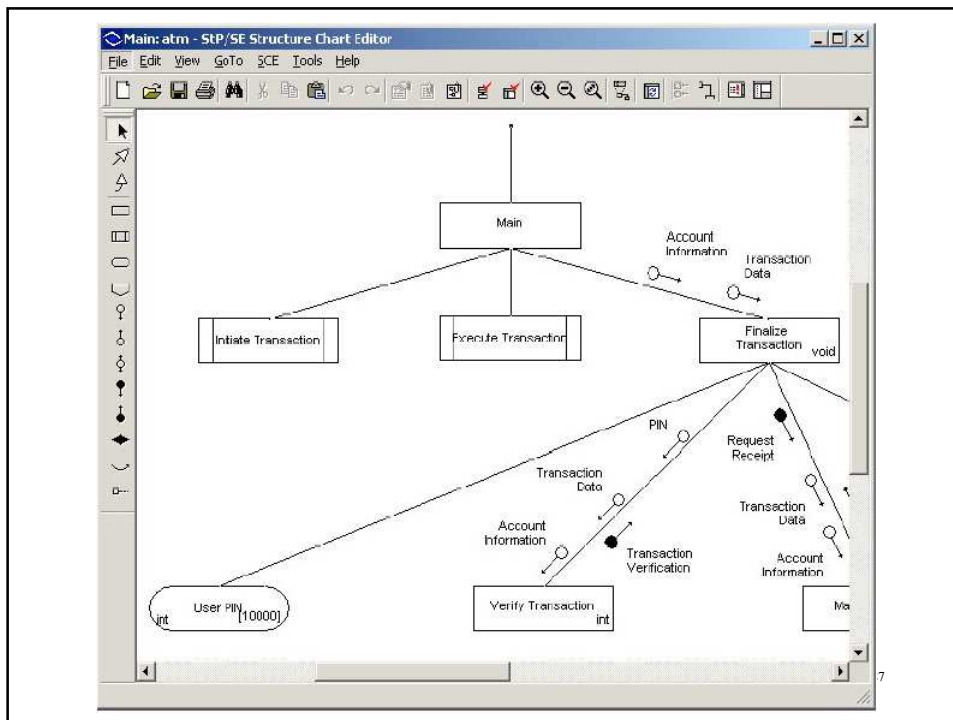
85

Structure Charts

- Aus dem Strukturierten Design [Coad/Yourdon79]
- Stellen die **Modulstruktur** nach der funktionalen Dekomposition [Parnas72] in einer hierarchischen Form dar
- Beinhalten Information, welche Daten zwischen den Modulen ausgetauscht werden
- Black box Ansicht von Modulen, Verhalten wird durch Input/Output beschrieben
- In UML: Klassen + Event Trace Diagramme

© J. Weidl-Rektenwald 02-08

86



Nested Trees

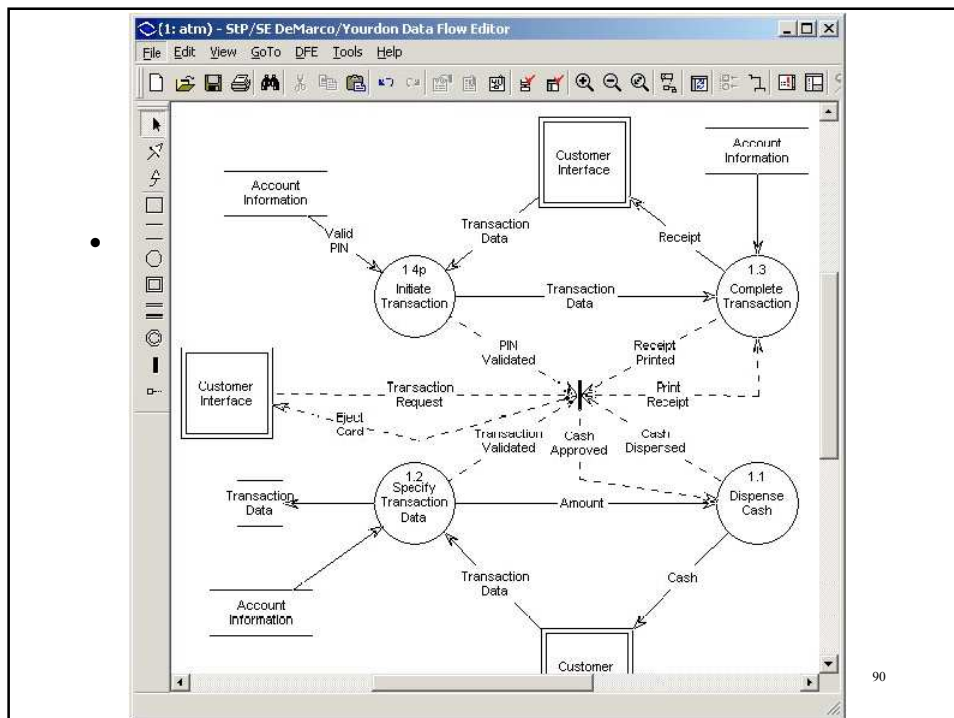
- Stellen den impliziten Datenaustausch über globale Variablen dar
- Knoten
 - Repräsentieren Module
- Kanten
 - Repräsentieren die Definition eines Moduls in einem anderen
- Nested Tree ermittelt
 - Verschachtelung der Deklarationen von Prozeduren und Funktionen
 - Kann Gültigkeitsbereich für jede Variable im System darstellen

Datenflussdiagramme

- Stellen Datenfluss und Datentransformation dar
 - Datentransformation: Ersetzen der formalen Parameter von Prozeduren durch die aktuellen Parameter
- Verschiedene Ansätze zur „bottom-up“ (i.e. reverse) Generierung finden sich in der Literatur
 - z.B. Benedusi, Cimitile und De Carlini [BCD89]

© J. Weidl-Rektenwald 02-08

89



90

Kontrollflussdiagramme

- Visualisierung des Kontrollflusses
 - Zwischen Prozeduren
 - Call tree
 - Innerhalb einer Prozedur
 - „Logische Wege“ durch eine Prozedur werden visualisiert
 - Visualisieren die zyklomatische Komplexität (McCabe Metrik)

Design Decisions

- During design and implementation decisions are made according to specific design rationales
 - Formal representation: Design Decision Tree (DDT)
- Tools for making design decisions persistent during the development process are only in experimental stage
 - Therefore, most of the design decisions almost always have to be extracted when examining existing code
- Reverse engineering design decisions deals with **automated decision extraction and injection**, knowledge repositories, knowledge management

Ziele von Reverse Engineering

- Beherrschung der System **Komplexität**
- Erzeugen von fehlender oder alternativer **Dokumentation**
- **Wiedergewinnung** verlorener Information
- Erkennung von Seiteneffekten und **Anomalien**
- **Migration** auf eine andere Hardware/Software Plattform bzw. Integration in eine CASE Umgebung
- Erleichterung der Software-Wiederverwendung

Reverse Engineering Kandidaten

- Schlecht strukturierter Source Code
- Umfassende korrektive Wartung
- Veraltete Dokumentation
- Design Infos fehlen oder sind unvollständig
- Module sind unüberschaubar komplex
- Migration auf eine neue Plattform
- System soll durch ein neues abgelöst werden

Reverse Engineering: Vorteile

- **Kosteneinsparung** in der Software Wartung
- Ermöglichen weiterer **Software Evolution**
- **Qualitätsverbesserung**
- **Wiederverwendung** von Software Komponenten
- Vorteile im Wettbewerb
- Investitionssicherung

© J. Weidl-Rektenwald 02-08

[Klösch/Gall95]

95

Reverse Engineering: Probleme

- Umfangreiches Source Code **Volumen**
- Mangelndes **Wissen** über das Programm
- Inkonsistente Entwicklungs**standards**
- Nicht aktuelle oder fehlende **Dokumentation**
- Hohe Redundanz

© J. Weidl-Rektenwald 02-08

[Klösch/Gall95]

96

Reverse Engineering bzw. Code Comprehension Tools

- Imagix4D (Imagix Corp.)
 - www.imagix.com
- Software Refinery, Refine/C (Reasoning Systems Inc.)
- Sniff+ (Wind River)
 - http://www.windriver.com/products/development_tools/ide/sniff_plus/
- Source Navigator
 - <http://sourcnav.sourceforge.net>
- Rational Rose
 - <http://www.ibm.com/software/rational>
- Software through Pictures SE (Aonix: StP/SE)
 - http://www.aonix.com/stp_se.html
- Sotograph
 - www.software-tomography.com

© J. Weidl-Rektenwald 02-08

97

Die Tool Falle

- *“A fool with a tool is still a fool”*
 - English proverb
- Give software tools to good engineers. You want bad engineers produce less, not more, poor-quality software.
 - [Davis95]: Principles of Software Development

© J. Weidl-Rektenwald 02-08

98

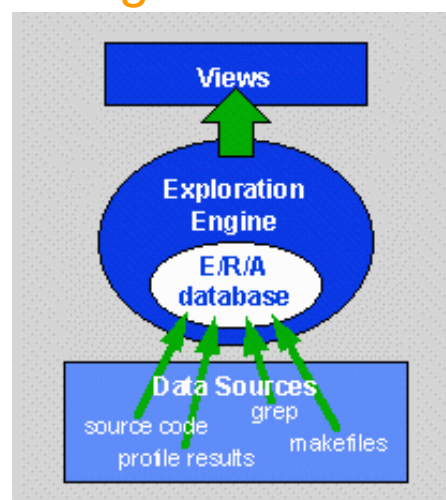
Imagix 4D: Funktionen

- Only for C/C++
- Provides graphical 3D output of analysed source
- Control Flow Analysis
 - Generation of augmented call graphs
- Annotated Flow Charts
 - Shows the logic flow for complex constructs
- Use Browser
 - Shows where and how a symbol is used
- Automated generation of HTML documentation

© J. Weidl-Rektenwald 02-08

99

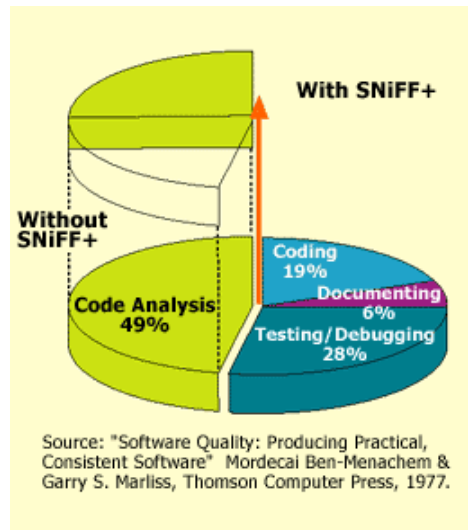
Imagix 4D: Toolkit



© J. Weidl-Rektenwald 02-08

100

Sniff+



101

Sniff+: Funktionen

- Source code editor
- Class browser
- Inheritance hierarchy browser
- Xref – Cross reference browser (uses/used)
- Include browser (Java: imports)
- Retriever – search for symbol names
- Grep
- Interface to debugger, build environment, revision control

© J. Weidl-Rektenwald 02-08

102

Software Refinery


- Programming-language specific parsers (Fortran, C, Cobol, Ada)
- Standard analyzers (xref, def/use, call tree)
- OO Repository for holding results of analyses
- Wide-spectrum language for writing custom analyses
- Extensions: dialect (parser), intervista (GUI)

© J. Weidl-Rektenwald 02-08

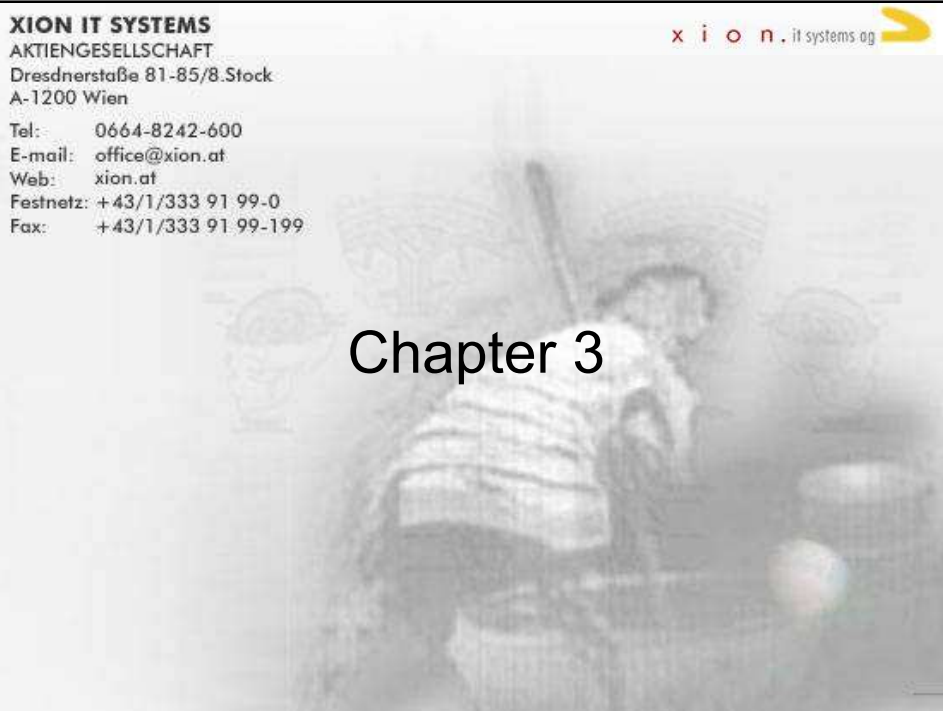
103

XION IT SYSTEMS
AKTIENGESELLSCHAFT
Dresdnerstraße 81-85/8.Stock
A-1200 Wien

Tel: 0664-8242-600
E-mail: office@xion.at
Web: xion.at
Festnetz: +43/1/333 91 99-0
Fax: +43/1/333 91 99-199

x i o n . i t systems ag 

Chapter 3



Chapter 3

- Inhalte
 - Restructuring
 - Refactoring
 - Reengineering
 - Dimensionen der Neuentwicklung
 - Organisation der Wartung
 - Management des Wartungsfalles
 - Life Cycle Modelle der Software Wartung
 - Produktivstellung

© J. Weidl-Rektenwald 02-08

105

Restructuring

Restructuring: Definition

- Restructuring is the transformation of
 - one representation form to another
 - at the **same relative abstraction level**,
 - while preserving the subject system's **external behavior** (functionality and semantics).

© J. Weidl-Rektenwald 02-08

[Chikofsky/Cross]

107

Restructuring

- Abstraktionsniveau bleibt erhalten
- Code-to-code transformation
 - Beispiele
 - Ersetzung von goto statements
 - IF zu CASE Transformation
 - Verbesserung der Modularisierung
- Data-to-data transformation
 - Datennormalisierung (z.B. Transformation in 3. Normalform)
- Ziele
 - Verbesserte Struktureigenschaften, Modularisierung
 - Dadurch erhöhte Lesbarkeit, Testbarkeit, Effizienz

© J. Weidl-Rektenwald 02-08

108

Restructuring: Historie

- 1966 Boehm/Jacobini
 - Jeder synchrone Ablauf eines Algorithmus kann mit 3 Konstrukten a) Sequenz b) Selektion c) Iteration logisch äquivalent zu hergebrachten mit goto programmierten Formen dargestellt werden
- 1968 Dijkstra (*goto* statement considered harmful)
 - Programme ohne *goto* sind übersichtlicher und besser lesbar
- 1972 Ashcroft und Manna
 - Alle Kontrollstrukturen in einem Programm können durch einen Algorithmus in while-Strukturen ersetzt werden

© J. Weidl-Rektenwald 02-08

109

Restructuring: Historie

- 1975 Erste Restructuring Tools kommen auf den Markt
 - „Structured Engine“ für Fortran
 - Neater/2 für PL/I
- 1980 Belady et al
 - „A graphic representation of structured programs“
 - Bedeutsam für die grafische Unterstützung des Restrukturierungsprozesses
- Modern
 - Refactoring (entspricht Restructuring im OO Paradigma)

© J. Weidl-Rektenwald 02-08

110

Refactoring

Refactoring: Motivation and Definition

- “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
- Definition “Refactoring”
 - “Improving the design of a program after it has been written”
 - A change made to the internal structure of software to make it **easier to understand** and **cheaper to modify** without changing its observable behaviour
- Refactoring is
 - Perfective maintenance
 - Object-oriented restructuring
 - Functionality preserving

[Fowler99]

Refactoring

- Refactoring basiert auf Arbeiten von Ward Cunningham, Kent Beck und William Opdyke
- Es existiert eine große Anzahl von vordefinierten Refactoring Maßnahmen
 - z.B. Extract Method, Move Method, Move Field, Remove Parameter, Collapse Hierarchy, Remove Middle Man, ...
 - M. Fowler, "Refactoring – Improving the design of existing code", Addison-Wesley, 1999
 - www.refactoring.com

© J. Weidl-Rektenwald 02-08

113

Refactoring: Advantages

- Advantages of Refactoring
 - Improves the Design of Software
 - Without refactoring, the design of software will decay
 - Makes Software Easier to Understand
 - Easier understanding means lower maintenance costs
 - Helps You Find Bugs
 - As you refactor, you better understand code
 - Helps You Program Faster
 - Based on the assumption that good design allows rapid development

[Fowler99]

© J. Weidl-Rektenwald 02-08

114

When to Refactor

- „The Rule of Three“
 - Refactor when you **add a function**
 - Refactor when you **need to fix a bug**
 - Refactor as you do a **code review**

[Fowler99]

© J. Weidl-Rektenwald 02-08

115

When to Refactor

- Duplicate Code
 - Extract method, Pull up field
- Long Method
 - Extract method, Introduce parameter object
- Large Class
 - Extract class, Extract subclass
- Switch Statements
 - Replace type code with subclasses/state
- ...

[Fowler99]

© J. Weidl-Rektenwald 02-08

116

Refactoring: Problems

- Databases
 - Flexibility in refactoring depends on the [level of indirection](#)
 - Changing the database schema forces you to [migrate the data](#)
- Changing Interfaces
 - Many refactorings change the [interface](#) of a class
 - Public interfaces vs. published interfaces
 - Solution: leave the old interface unchanged and publish a new one
- Design changes that are difficult to refactor
 - Boils down to the question: Can everything be solved by refactoring?

[Fowler99]

© J. Weidl-Rektenwald 02-08

117

Refactoring: Tools

- IDE embedded
 - Borland JBuilder
 - IntelliJ IDEA
 - Eclipse etc.
- Plugins
 - RefactorIt (NetBeans, Forte, JDeveloper, JBuilder)
 - JafaRefactor (jEdit)
- Tools are available for Java, Smalltalk, .NET, Visual Basic, Python, Self, ...
- [www.refactoring.com]

© J. Weidl-Rektenwald 02-08

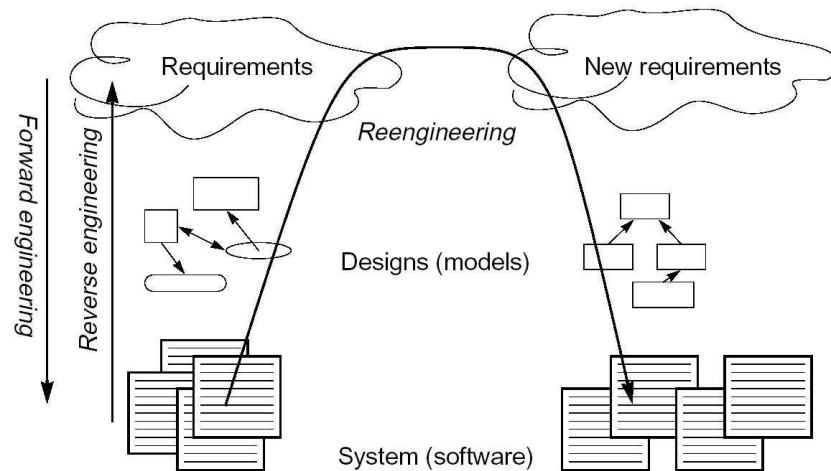
118

Re-Engineering

Re-Engineering: Definition

- Re-Engineering ist
 - die **Untersuchung und Änderung** eines bestehenden Systems
 - mit dem Ziel, das System in einer neuen, geänderten Form zu implementieren
 - **Strukturiert in Reverse Engineering und Forward Engineering Phase**

Reverse and Reengineering



© J. Weidl-Rektenwald 02-08

121

Re-Engineering: Probleme

- Komplexe Systeme können meist nur **schrittweise migriert** werden
 - Anbindung über *forward-* und *reverse-gateways*
- Reverse Engineering Ergebnisse sind unter Umständen dürrtig
 - Fehlendes Know How, Werkzeuge
 - Hohe Komplexität der untersuchten Systeme
 - Zeitdruck
 - Im durch Forward Engineering erstellten System fehlt daher oft Funktionalität

© J. Weidl-Rektenwald 02-08

122

Reengineering: Examples

- Data migration
 - Flat Files Data Repository to Database
- Programming language migration
 - e.g. 3GL to 4GL (Cobol to C++ or Java)
- User interface migration
 - Command line to GUI
- Procedural to object-oriented paradigm
 - „Re-architecting“

© J. Weidl-Rektenwald 02-08

123

Dimensionen der Neuentwicklung

- Neuentwicklung
 - From scratch / Auf der grünen Wiese
 - Durch Anforderungsanalyse, typisches Forward Engineering
- Re-Engineering
 - Reverse Engineering von
 - Design
 - Architektur
 - Anforderungen
 - Anschließendes Forward Engineering
- Transformation
 - Intra-paradigmatisch vs. inter-paradigmatisch

© J. Weidl-Rektenwald 02-08

124

Organisation der Wartung

„Organisation“

- *Organisation* als *Tätigkeit* („organisieren“) heißt, einem zielorientierten, sozio-technischen System eine *dauerhaft wirksame Struktur* zu geben. Diese Struktur entsteht durch formalisierte (verbindliche) *generelle Regelungen*, in dem die *Beziehungen* von *Aufgabenträgern, Informationen und Sachmitteln* bei der Aufgabenerfüllung festgelegt sind.

Organisation und Management

- *Organisation* als *Ergebnis* des Organisierens ist die **dauerhaft wirksame Struktur** von zielorientierten sozio-technischen Systemen.
- Gegenstand der Managementlehre ist die Gestaltung von **Organisationen**

© J. Weidl-Rektenwald 02-08

127

Aktivitäten des Wartungsfalles (vgl. Lecture 1)

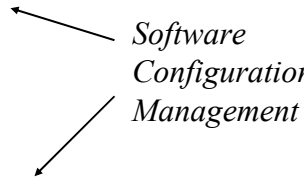
- Analyse bzw. Planung der Änderung
 - Program Comprehension
 - Change Impact Analysis
- Implementierung der Änderung
 - Restructuring
 - Change Propagation
- Verifikation und Validierung
- Re-Dokumentation

} M
a
n
a
g
e
m
e
n
t

© J. Weidl-Rektenwald 02-08

128

Management des Wartungsfalles

- Fehlermeldung bzw. Änderungsantrag
 - Life Cycle Management (Defect and Change Tracking)
 - Evaluierung/Reporting
 - Analyse bzw. Planung der Änderung
 - Program Comprehension
 - Change Impact Analysis
 - Implementierung der Änderung
 - Restructuring
 - Change Propagation
 - Verwalten der Artefakte (Software Artifact Management)
 - Verifikation und Validierung
 - Re-Dokumentation
 - Produktivstellung der Änderung
- Software Configuration Management*
- 

© J. Weidl-Rektenwald 02-08

129

Life Cycle Modelle der Software Wartung

Life Cycle Modelle

- Life Cycle Modelle
 - beschreiben den zeitlichen Ablauf von Teil-Prozessen in einem Gesamtprozess
 - bieten einen Top-Level View auf einen Gesamtprozess

Life Cycle Modelle der Software Entwicklung

- Wasserfall Modell [W.W. Royce 1970]
- Spiralmodell [Boehm]
- Rapid prototyping
- OO
 - Fountain Model
 - Inkrementell-iterative Entwicklung (vgl. Rational Unified Process - RUP)
 - Extreme Programming (Kent Beck et al.)

Life Cycle Modelle der Software Wartung

- Quick-fix Model [Basili90]
 - Änderung erfolgt **sofort** auf der Code Ebene
 - Änderung wird in Dokumentation typischerweise **nicht** nachgezogen
- Iterative Enhancement Model [Basili90]
 - Beginnt eine Änderung bei den **Anforderungen** und zieht Änderungen definiert bis in die **Testphase** nach

© J. Weidl-Rektenwald 02-08

133

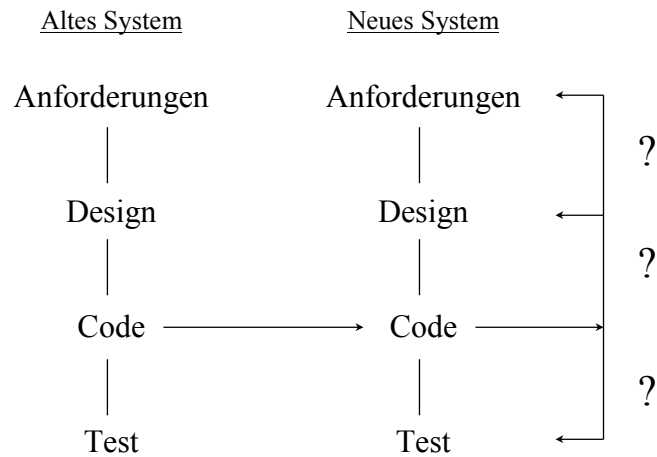
Life Cycle Modelle der Software Wartung

- Full-reuse Modell [Basili90]
 - Reengineering Ansatz
 - Klassisches Forward-Engineering beginnend mit den Requirements und Wiederverwendung von möglichst großen Teilen des Altsystems
 - Voraussetzung
 - **Altsystem ist in wieder verwendbare Teile strukturiert**

© J. Weidl-Rektenwald 02-08

134

Quick Fix Model



© J. Weidl-Rektenwald 02-08

135

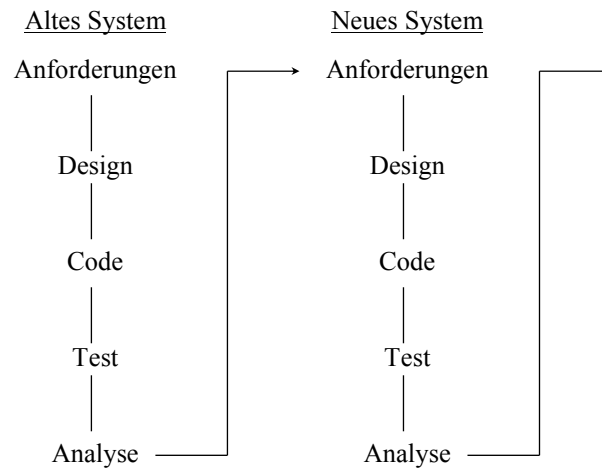
Quick Fix Model: Folgen

- Führt zu **unübersichtlichen** Systemen
- Mit jeder Änderung werden weitere Änderungen **erschwert**
 - Dokumentation zu den vorangegangenen Änderungen (warum?, warum hier?, warum nicht anders?) ist nicht vorhanden

© J. Weidl-Rektenwald 02-08

136

Iterative Enhancement Model



© J. Weidl-Rektenwald 02-08

137

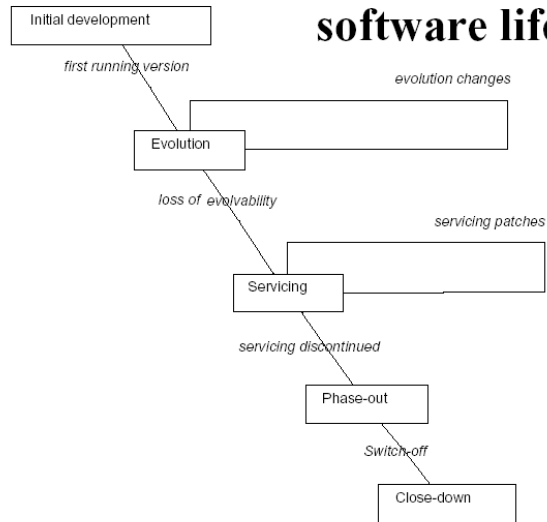
Staged Life Cycle Model

- Nach [Bennett/Rajlich]:
- „If changes can be anticipated at design time
 - They can be built in by parameterisations etc.“
 - (i.e. they can be planned for)
- „However, 40 years of hard experience confirms:
 - Many changes cannot even be **conceived** by the original designers
 - **Inability** to change software quickly and reliably means that business opportunities are lost
 - Our solution: base the software life cycle on the fact that many changes cannot be predicted“

© J. Weidl-Rektenwald 02-08

138

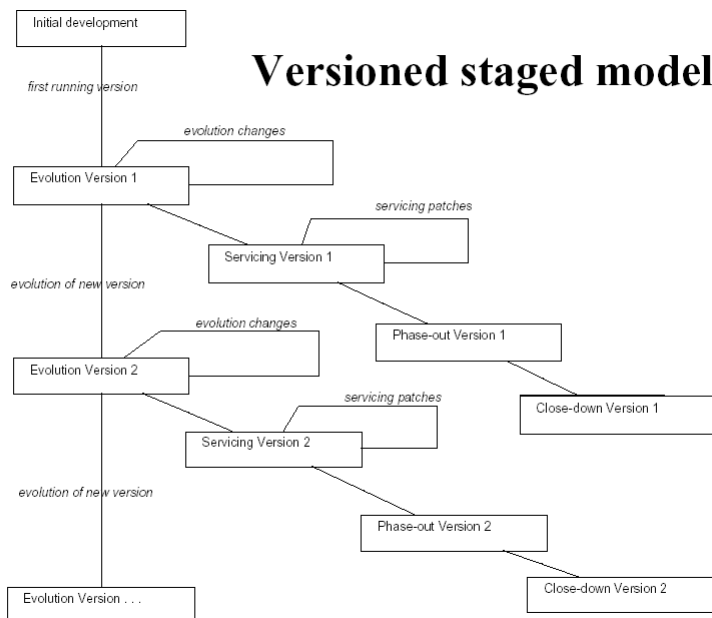
Staged model of software lifecycle



© J. Weidl-Rektenwald 02-08

139

Versioned staged model



© J. Weidl-Rektenwald 02-08

140

Produktivstellung

Wartungsfenster

- Ein Wartungsfenster ist ein **begrenzt**es Zeitintervall, in dem ein Produktionssystem für Wartungsarbeiten außer Betrieb geht
 - Das System muss **definiert** außer Betrieb genommen werden (Ankündigung und Wartungsmeldung)
 - Die Wartungsarbeiten müssen genau **geplant** werden (Projektmanagement)
 - Definition des „Point of no return“
 - Ab diesem Zeitpunkt kann nicht mehr auf das alte System zurückgestellt werden (außer durch Einspielen eines Backups)
 - Definition des Wiederanlaufes

Checkliste Wartungsfenster


- **Voraussetzung:** Freigabe der neuen Produktions-Release
- Projektplan erstellen
 - **Ressourcen/Rollen** festlegen
 - Detaillierter **Ablaufplan** (Work Breakdown Structure), Definition von Go/No-Go bzw. **Rollback** Punkten
 - **Dauer** definieren
- Projektplan durch **Tests** verifizieren
- Termin mit dem Systemverantwortlichen bzw. den Benutzern (intern, eventuell extern) vereinbaren
- Verteilen des **Projektplans** an die Akteure (Betriebsführungspersonal, Wartungspersonal, etc.)

© J. Weidl-Rektenwald 02-08

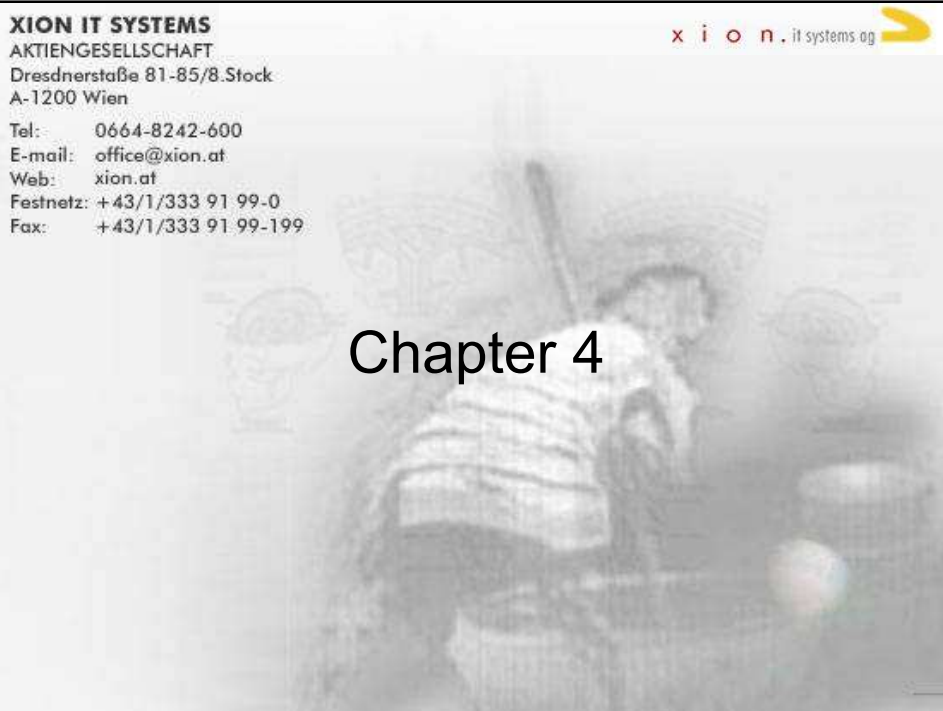
143

XION IT SYSTEMS
AKTIENGESELLSCHAFT
Dresdnerstraße 81-85/8.Stock
A-1200 Wien

Tel.: 0664-8242-600
E-mail: office@xion.at
Web: xion.at
Festnetz: +43/1/333 91 99-0
Fax: +43/1/333 91 99-199

x i o n . i t systems ag 

Chapter 4



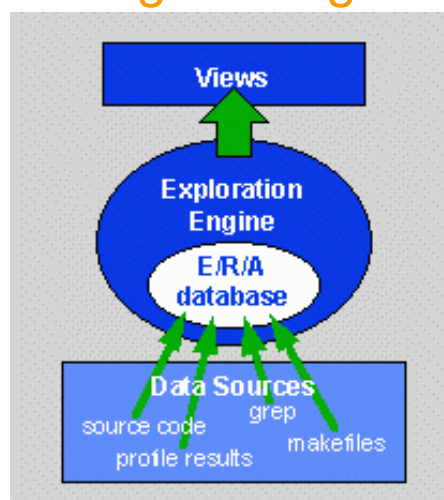
Chapter 4

- Inhalte
 - Tool Demo
 - Reverse Engineering mit Imagix4D
 - Refactoring mit IntelliJ IDEA
 - Architektur- und Evolutionsanalyse mit dem Sotograph
 - Software Analyse mit dem Eclipse CREOLE Plugin

© J. Weidl-Rektenwald 02-08

145

Reverse Engineering: Imagix 4D



© J. Weidl-Rektenwald 02-08

146

Refactoring: IntelliJ IDEA

```

public class ListUsersAction extends Action {
    /**
     * Base Action performs(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)
     */
    public ActionForward perform(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {
        // Validate the request parameters specified by the user
        ActionErrors errors = new ActionErrors();

        ListUsersForm myform = (ListUsersForm) form;

        try {
            if (!StringUtil.isLoggedin(request)) {
                UserSession user = UserSessionUtil.getSession();
                UserSession us = user.create();

                Collection result = null;

                String id = request.getParameter("id");
                if (id == null) {
                    result = us.searchAllUsers();
                } else {
                    result = new Vector();
                    result.add(us.searchUsersById(id));
                }

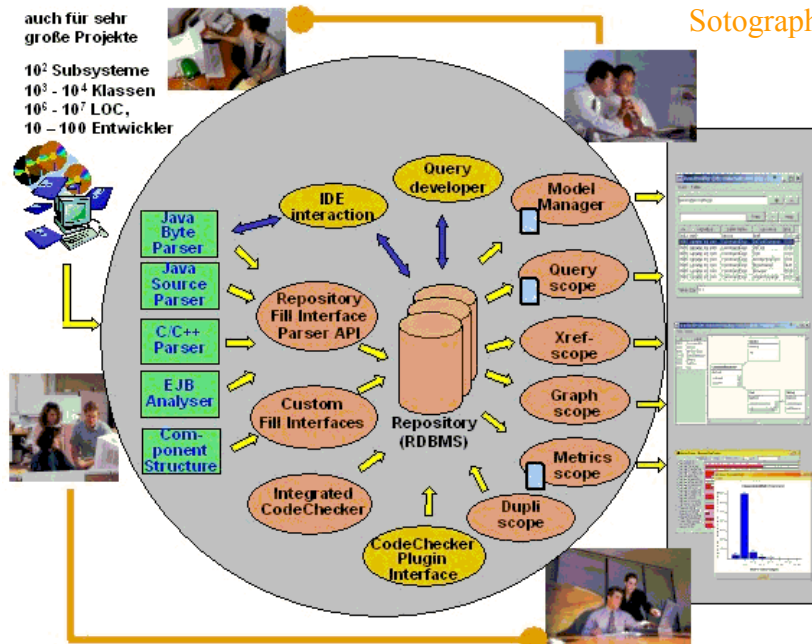
                myform.setExistingUsers(result);
            }
        }
    }
}
    
```

© J. Weidl-Rektenwald 02-08

147

auch für sehr große Projekte
 10² Subsysteme
 10³ - 10⁴ Klassen
 10⁶ - 10⁷ LOC,
 10 - 100 Entwickler

Sotograph



CREOLE

- Eclipse plugin by „The Chisel Group“
 - Computer Human Interaction & Software Engineering Lab
 - [<http://www.thechiselgroup.org/creole>]
- „Creole is the term used to describe our plug-in to the Eclipse platform which integrates SHriMP with the Eclipse platform's Java Development Tools (JDT).“
- SHriMP = **S**imple **H**ierarchical **M**ulti-**P**erspective

© J. Weidl-Rektenwald 02-08

149

XION IT SYSTEMS

AKTIENGESELLSCHAFT

Dresdnerstraße 81-85/8.Stock
A-1200 Wien

Tel.: 0664-8242-600

E-mail: office@xion.at

Web: xion.at

Festnetz: +43/1/333 91 99-0

Fax: +43/1/333 91 99-199

x i o n . i t systems ag 

Chapter 5

Chapter 5

- Inhalte
 - Software Evolution
 - Definition
 - Types of Programs
 - Laws of Software Evolution
 - Change Patterns and Evolutionary Narratives

Fallbeispiel: „Adaption auf der falschen Abstraktionsstufe“

- Gegeben: Datenbankmodell
 - Datenbankfeld: „SA_EINZEL“
 - Vermutete Semantik
 - SA ... Sportart
 - EINZEL ... Einzelsportart
 - Typ: char(1), keine Constraints
 - Wertbelegung in der Datenbank
 - „J“ ... Folgerung: Ja, Sportart ist eine Einzelsportart
 - „N“ ... Folgerung: Nein, Sportart ist keine Einzelsportart
 - „T“ ... **Überraschung!**: Sportart ist eine „Tennissportart“ ☹

Software Evolution

Evolution: General Definition 1/2

- Evolution is the process of *progressive change* over time in characteristics, attributes, properties of some material or abstract, natural or artificial, entity or system or of a sequence of these
 - Changes are *progressive* when they result in a definable trend of, for example, increasing value, growing precision or better fit to a changing domain or context

Evolution: General Definition 2/2

- Entities include objects or collections of objects (e.g. population) such as natural species, societies, cities, artefacts, concepts, theories, ideas or systems of these
- Change process will, in general, be continual with **relatively slow rate** of change, or discrete with individual incremental changes, small relative to entity as a whole
- Source: Lehman and Ramil 2001

Software Evolution: Definition 1/2

- Keine genormte Definition
- Nach Lehman/Ramil
 - Software Evolution is the process of continual fixing, adaption, enhancement to maintain stakeholder satisfaction
 - In response to changes in domains, needs, expectations
- Nach Bennet/Rajlich
 - Maintenance means general post-delivery activities
 - Evolution refers to a particular phase in the staged model where substantial changes are made to the software

Software Evolution: Definition 2/2

- Nach Godfrey
 - Evolution is what happens while you are busy making other plans
 - Maintenance is the *planned* set of tasks to effect changes
 - Evolution is what actually happens to software

Types of Programs

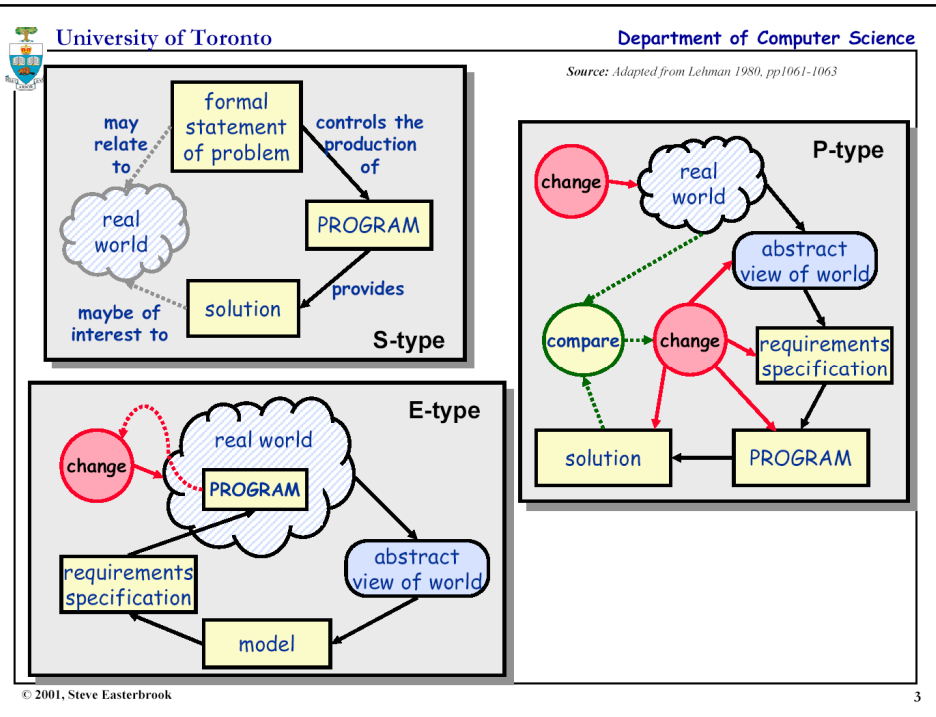
- *Nach Lehman, Belady 1980, pp. 1060-1076*
- S-type Programs („Specifiable“)
 - Problem can be stated formally and completely
 - Acceptance: Is the program correct according to its specification?
 - This software does not evolve
 - A change to the specification defines a new problem, hence a new program

Types of Programs

- P-type Programs („Problem-solving“)
 - Imprecise statement of a real-world problem
 - Acceptance: Is the program an acceptable solution to the problem?
 - This software is likely to evolve continuously
 - Because solution is never perfect, and can be improved
 - Because the real-world changes and hence the problem changes

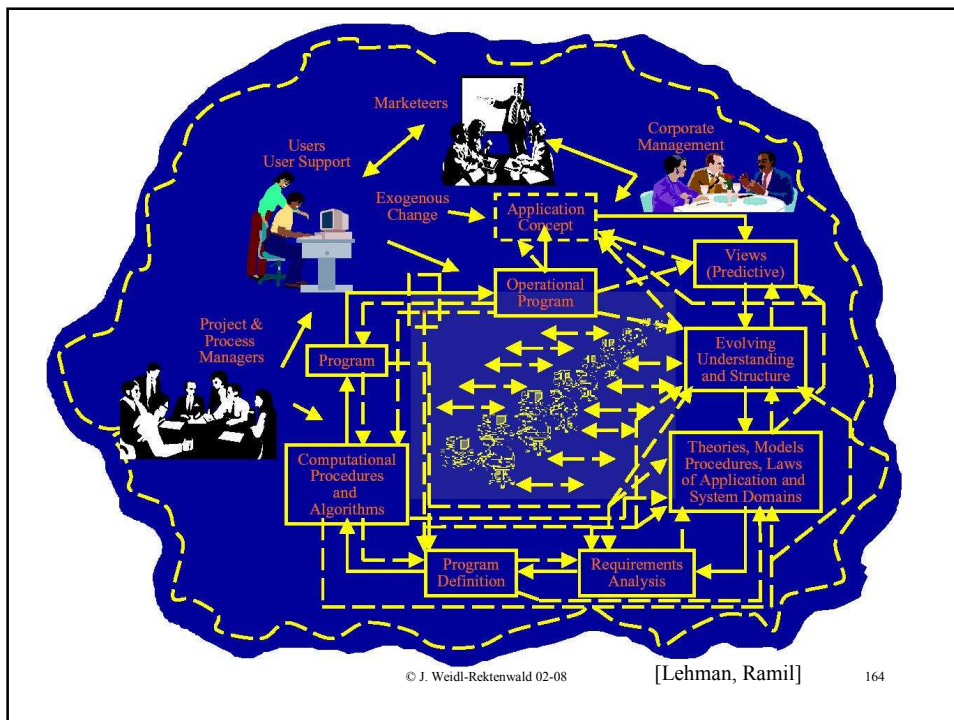
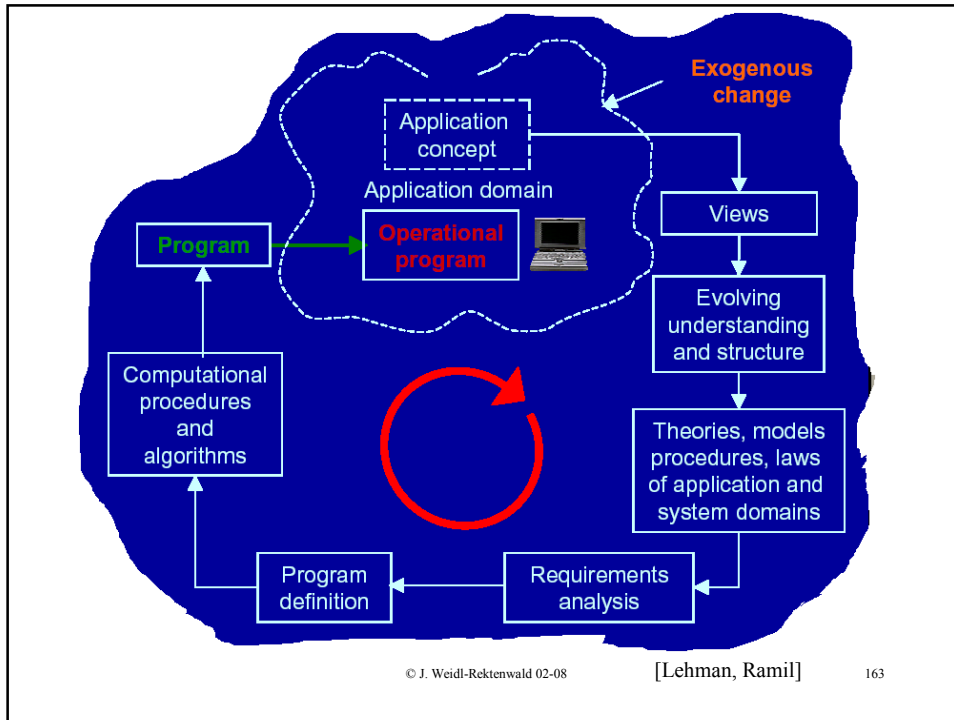
Types of Programs

- E-type Programs („Embedded“)
 - A system that becomes part of the world it models
 - Acceptance: Depends entirely on opinion and judgement; criterion is the satisfaction of stakeholder needs
 - This software is *inherently* evolutionary
 - Changes in the software and the world affect each other



Software Systeme als komplexe Feedback Systeme

- Der Entwicklungs- und Evolutionsprozess eines Software Systems wird von Lehman als
 - Multi-level
 - Multi-loop
 - Multi-agent
 - Feedback System bezeichnet.
- Feedback technisch: Die Rückführung eines (transformierten) Teils des Ausgangssignals als Eingangssignal in ein System
 - („Feedback: The return of a portion of the **output**, or processed portion of the output, of a (usually active) device to the **input**“)



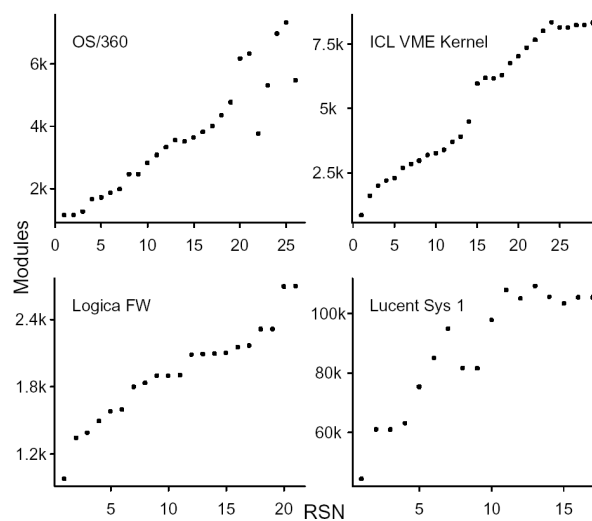
Laws of Software Evolution

- In den späten sechziger Jahren untersuchen *Lehman und Belady* die *Release History Daten* von IBM OS/360 mittels bestimmter Metriken und stoßen auf Eigenschaften im Evolutionsprozess, die bei anderen Systemen in späteren Untersuchungen ebenfalls nachvollzogen werden können
- Diese Eigenschaften scheinen Gesetzmäßigkeiten zu folgen und wurden als „Laws of Software Evolution“ postuliert
- Die „Laws of Software Evolution“ ergeben sich aus der Beobachtung von *E-type programs*

© J. Weidl-Rektenwald 02-08

165

Laws of Software Evolution



166

Laws of Software Evolution

- Warum „Gesetze“?
 - Die entdeckten Phänomene der Evolution werden als Gesetze bezeichnet, da sie technologie- und prozessunabhängige Mechanismen bezeichnen

Laws of Software Evolution

- *Nach Lehman, Belady 1980, pp. 1061-1063 und spätere Publikationen*
- 1) Law of continuing change
 - “A system that reflects some external reality undergoes continuing change or becomes progressively less useful
 - The change process continues until it becomes more economical to replace it by a new or restructured system.”
- 2) Law of increasing entropy (or: complexity)
 - “The entropy of a system increases with time unless specific work is executed to maintain or reduce it.”

Laws of Software Evolution

- 3) Fundamental law of software evolution
 - Software evolution is self-regulating with statistically determinable trends and invariants
- 4) Conservation of organisational stability (invariant work rate)
 - During the active live of a software system the *average effective global activity rate* is roughly constant

Laws of Software Evolution

- 5) Conservation of familiarity
 - In general, the average incremental growth rate (growth rate trend) tends to decline
 - As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery.
- 6) Continuing growth
 - The functional content of E-type systems must be continually increased to maintain user satisfaction

Laws of Software Evolution

- 7) Declining quality
 - The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes
- 8) Feedback System
 - E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base

© J. Weidl-Rektenwald 02-08

171

Lehmans Approach: Formal

- Lehman describes software evolution on a formal level
 - Based on observations
 - Empirical generalisations are made
 - They provide basis for axioms in a formal theory
 - Possible inferences are proposed
 - Derived from the formal models
 - Basis for potential theorems in formal theory
 - Try to fully prove theorems

© J. Weidl-Rektenwald 02-08

172

Formal Models of Software Evolution: Growth

- Inverse Square Model [Turski 1996]
 - $\underline{S}_1 = S_1$
 - $\underline{S}_i = \underline{S}_{i-1} + e / (\underline{S}_{i-1})^2$
 - S ... Size (often number of modules)
 - i ... Release sequence number (1..n, n = max release nr.)
 - e ... Model parameter
 - S_i and \underline{S}_i stand for actual and predicted size at release i
- Other model: Normalised size as a function of the normalised work rate [Lehman 2001]
 - $\underline{S}_i / S_1 = (H_i / H_1)^{1/g'}$ for $i \geq 1$
 - H ... Work rate as indirect effort indicator (e.g. elements handled)
 - g' ... Model parameter

© J. Weidl-Rektenwald 02-08

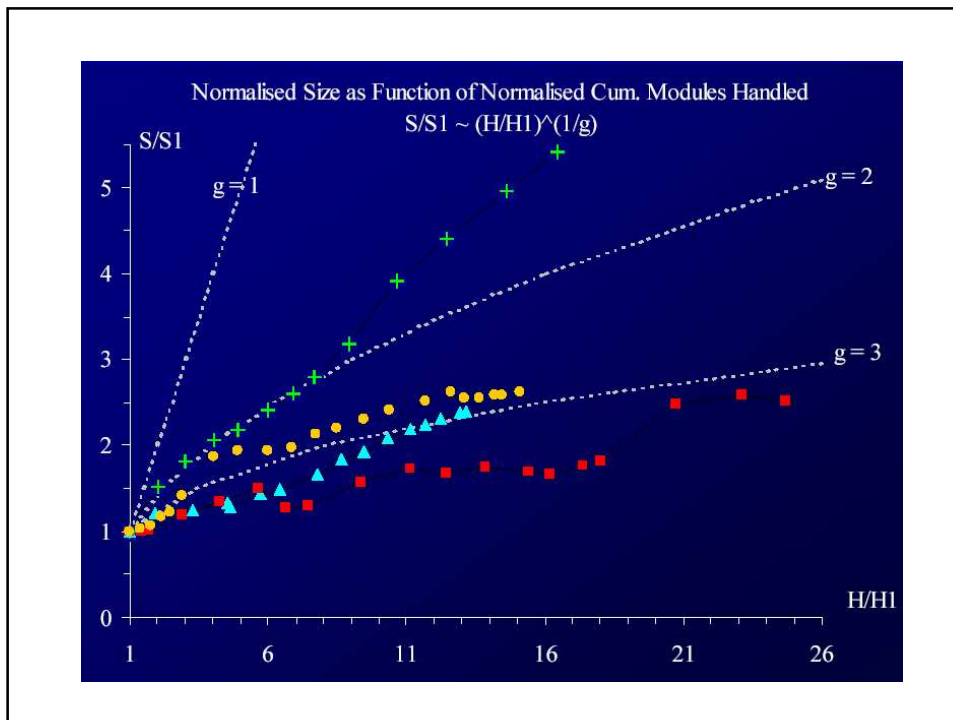
173

Formal Models: An Example

- Next slide shows the *normalised size as a function of the normalised work rate*
 - size measured in number of modules
 - work rate measured in modules handled
- For four industrially evolved systems
- Three different organisations
- Three different applications domains
- Data taken from release data history

© J. Weidl-Rektenwald 02-08

174



Formal Models: Use?

- Formal Models provide means for
 - Evolution planning
 - Simulation, visualisation, release planning
 - Process Management and Control
 - Long term prognosis
 - Overall process improvement
 - Tools

Research Areas in Software Evolution

- The driving force guiding the work will be the search for formally supported techniques:
 - logic-based declarative description and reasoning techniques
 - formal models for software evolution based on rewriting systems
 - software metrics
 - visualisation techniques
 - generation of design documents and source code
 - extraction of design and analysis documentation
 - migration to component-based and web-based systems
 - the use of metamodels as a general integration technique

© J. Weidl-Rektenwald 02-08

177

Change Patterns and Evolutionary Narratives

Change patterns and evolutionary narratives

- Cathedral style [Raymond]
 - careful control and management
 - debugging done before committing code
 - evolution is slow, planned, rarely undone
- Bazaar style
 - lots of low-level changes, frequent fixes
 - lots of “building around” rather than wholesale changing, occasional redesigns
 - creeping feature-itis, “complete” dependency graph

© J. Weidl-Rektenwald 02-08

[Godfrey 2001]

179

Change patterns and evolutionary narratives

- Band-aid evolution (just add a layer)
 - quick & dirty way to add new functionality, esp. if system is not well understood
e.g. Y2K fixing, adding portability, new features
- “Vestigial features”
 - design artifact persists after rationale dies
e.g. whale fin bone structure resembles hand

© J. Weidl-Rektenwald 02-08

[Godfrey 2001]

180

Change patterns and evolutionary narratives

- “Adaptive radiation” [Lehman]
 - when conditions permit, encourage wild variation for a while
 - later, evaluate and let “best” ideas live on.
e.g. Linux kernel evolution
- “Convergent evolution”
 - compare similar systems to reference architecture (or to each other)
e.g. everyone grows an XML generator in response to market pressure

© J. Weidl-Rektenwald 02-08

[Godfrey 2001]

181

Change patterns and evolutionary narratives

- Radical redesigns (localized and global)
 - aka “refactoring”
 - little new functionality added, but structure changes significantly, legacy cruft dissipates
 - likely “goodness” (design metrics) improves
- Migration patterns
 - look out for known translation idioms, especially if migration is not one big bang
e.g. procedural-to-OO idioms

© J. Weidl-Rektenwald 02-08

[Godfrey 2001]

182

Change patterns and evolutionary narratives

- Reuse patterns
 - components are (re)used in different systems
e.g. build COTS interface, throw out homebrew DB

© J. Weidl-Rektenwald 02-08

[Godfrey 2001]

183

Ausgewählte Kapitel der Software Wartung

- Program Comprehension
 - Span of Understanding
- Maintainability (Wartbarkeit)
- Design for Change
 - Design Metriken
 - Coupling und Cohesion

© J. Weidl-Rektenwald 02-08

184

Span of Understanding

- Span of Understanding
 - Nennt man die Zeitspanne, die der Programmierer zum Verstehen eines definierten Programmstückes benötigt
- Ziel der Forschung: Wie kann man den *Span of Understanding* verkürzen bzw. minimieren

Definition: Maintainability

- Maintainability is the ease of maintenance
- Can be decomposed as
 - Repairability
 - Ability to correct defects in reasonable time
 - Evolvability
 - Ability to adapt software to environment changes and to improve it in reasonable time

Design Metrik: Cohesion

- Beschreibt den Grad der logischen Abhängigkeiten innerhalb eines Software Moduls
- Je größer die Cohesion desto besser das Software Design
- Hinter einem Interface sollte die Cohesion maximal sein
- Große Cohesion erlaubt die einfache Wiederverwendung von Software Modulen

© J. Weidl-Rektenwald 02-08

187

Design Metrik: Coupling

- Coupling beschreibt den Grad der logischen Abhängigkeiten zwischen verschiedenen Software Modulen
- Je größer das Coupling desto schlechter das Software Design
- Das Coupling über Interfaces hinweg sollte minimal sein
- Großes Coupling verhindert die einfache Wiederverwendung von Softwaremodulen

© J. Weidl-Rektenwald 02-08

188

Computer science...

- Computer Science is the discipline that believes all problems can be solved by adding one more layer of indirection.
 - *Dennis DeBruler*

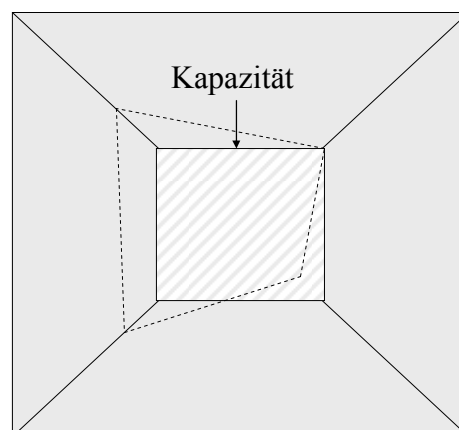
© J. Weidl-Rektenwald 02-08

189

Teufelsquadrat

Qualität

Ressourcen



Projekt-dauer

Wirtschaft-lichkeit¹⁹⁰

© J. Weidl-Rektenwald 02-08