

Softwarereengineering

Vorwort

Dieses Dokument stellt eine Mitschrift der Vorlesung „Softwarereengineering“ von PD. Dr. M. Riebisch an der TU Ilmenau im Sommersemester 2007 dar.

Inhaltsverzeichnis

Vorwort	1
Inhaltsverzeichnis	1
1 Einordnung	1
1 Situation	1
2 Entwicklungsprozess und Wartungsphase	2
2 Legacy-Systeme	2
1 Was ist ein Legacy-System?	2
2 Strukturen von Legacy-Systemen	3
3 Bewertung von Legacy-Systemen	3
3 Analyse und Assessment	4
1 Analyse auf Systemebene	5
2 Abdeckungsanalyse	5
3 Software-Visualisierung	5
3.1 Statische Programm-Visualisierung	5
3.2 Dynamische Programm-Visualisierung	6
4 Analyse auf Modell- und Architekturebene	6
4.1 Programmverstehen	6
4.2 Architekturekonstruktion	6
5 Analyse auf Code-Ebene	7
5.1 Compilertechniken	7
5.2 Programm-Slicing	7
5.3 Clone-Detection	7
4 Maßnahmen oberhalb der Codeebene	7
1 Traceability	8

2 Wrapping	8
5 Reengineering-Maßnahmen auf Codeebene	9
1 Refactoring	9
2 Clone Consolidation	10
3 Modularisierung	10
4 Restrukturierung des Steuerflusses	10
5 Datenabstraktionen und Daten-Reengineering	11
6 Managment und Organisation	11
1 Bezug zum Projektmanagment	11
1.1 Planung und Durchführung von Reengineering-Projekten	11
1.2 Kosten/Nutzen	12
1.3 Risikomanagment und Entscheidungsprozess	12
1.4 (Änderungen/) Change Management	13
2 Migrationsstrategien	13
3 Teststrategien	13
4 Einbindung Prozessmodelle	13
5 Teststrategien	13
6 Werkzeuge	14

1 Einordnung

1 Situation

Die Rolle von Änderungen ist gekennzeichnet durch:

- Softwaresysteme sind sehr komplex (Anforderungen und Änderungen sind komplex)
 - es gibt viele Module
 - Öffnung von Systemen für das Web
 - Kopplung mit (externen) Systemen
 - Zusammenführung von verwandten Ausgaben aus verschiedenen Systeme
- Bedeutung
 - Geschäftsprozesse von Software abhängig ⇒ **geschäftskritische SW-Systeme**
 - hohe Änderungsdruck der Geschäftsprozesse ⇒ **Änderungsdruck auf SW-Entwickler**
 - Änderungen risikoarm und zukünftige Änderungen ermöglichen

2 Entwicklungsprozess und Wartungsphase

Das klassische Entwicklungsmodell umfasst:

- Konzeption
- Anforderungsanalyse
- Entwurf
- Implementation (Verwirklichung)
- Integration
- Inbetriebnahme
- **Wartung**

Die Wartungsperspektive ist im **Staged Model** (siehe 1. Foliensatz) enthalten.

Definition 1. (*Wartbarkeit*)

Die **Wartbarkeit** ist ein Qualitätsmerkmal nach ISO 9126. Es beschreibt den Aufwand der betrieben werden muss, um Änderungen zu ermöglichen. Sie wird unterteilt:

- *Verständlichkeit*
- *Werkzeugunabhängigkeit*
- *Durchführbarkeit*
- *Testbarkeit*

Im Weiteren werden noch wichtige Definitionen vorgestellt, die auf dem ersten Foliensatz nachlesbar sind.

2 Legacy-Systeme

1 Was ist ein Legacy-System?

Definition 2. Im Wesentlichen ist Legacy ein Vermächtnis. Ein **Legacy-System** ist ein altes SW-System, dessen Versagen ernste Auswirkungen auf tägliche Abläufe im Unternehmen. Das SW-System ist nicht mehr wie ursprünglich geliefert, sondern hat schon zahlreiche Änderungen durchlaufen. Solche Änderungen können z.B. durch Märkte und Managementphilosophien eintreten.

Was ist dabei zu beachten?

- zahlreiche beteiligte Entwickler
- Dokumentation ständig abnehmender Qualität
- Struktur zunehmend verschlechtert

Es gibt verschiedene Risiken einer Neuentwicklung:

- Teile nicht im Neusystem enthalten (Spezifikation fehlte)
- Arbeitsabläufe müssen geändert werden
 - Änderungen an Geschäftsprozessen
 - Abläufe waren nicht genau beschrieben
 - ⇒ **hoher Aufwand**

- wichtige Geschäftsregeln sind nicht enthalten (waren in Software eingebettet, aber nicht dokumentiert) ⇒ **Folgekosten, wirtschaftliche Nachteile**
- **hohe Kosten**, aber für den Nutzer kein wirklich sichtbarer Mehrwert
- Risiko einer Neuentwicklung: **Budget + Termineinhaltung**

Wir wollen das System also weiter betreiben, jedoch einer **Überarbeitung** unterziehen. Auch dies ist jedoch ebenso mit **hohen Kosten** und **schwer abschätzbaren Risiken** verbunden, da

- Dokumentation nicht vorhanden/schlecht ⇒ hoher Aufwand für **Analyse/Verstehen**
- Fehlerrisiko (Komponenten können falsch verstanden werden)
- System schlecht **strukturiert**, Abhängigkeiten versteckt (als Folge von vielen Änderungen) ⇒ Änderungen aufwendig, unvollständig und fehlerhaft
- Anpassung der Architektur (Methoden und Schnittstellen) wären notwendig, aber aufwendig ⇒ zusätzliche Aufwände und Risiken in eigentlich unbeteiligten Bereichen

2 Strukturen von Legacy-Systemen

Die „alten“ Systeme sind meist nicht objektorientiert aufgebaut, sondern eine Sammlung von Funktionen und Unterprogrammen. Diese sind meist nach funktionalen Kriterien gegliedert (*strukturierter Entwurf* (Top-Down)). Dort finden wir häufig einen zentral gehaltenen Systemzustand. Weitere Merkmale:

- funktionale Details gekapselt, Systemzustand global zugreifbar
- lokale Statusinformationen in Funktionen nur während der Ausführung aufrecht erhalten

Einsatzbereich und Vorteile des funktionalen Entwurfs.

- wenn gemeinsame Datenhaltung gewollt
- wenn Signale/Impulse/Eingaben den Ablauf bestimmen
- wenige Abhängigkeiten von früherem Zustand („Module ohne Gedächtnis“)
- weitere Vorteile:
 - Gliederung EVA entspricht häufigen Aufgabenstellungen

3 Bewertung von Legacy-Systemen

Diese Bewertungen sind wichtig um Entscheidungen treffen zu können (Weiterentwicklung, Ablösung). Sie ist wichtig um den Budgeteinsatz und die Wertermittlung zu ermöglichen. Strategien für Weiterentwicklung/Nutzung:

Abbildung 1.

1. System ausmustern
2. System pflegen
3. System sanieren

Die **Beurteilung des Geschäftswertes** unterliegt folgenden Kriterien:

- es ist subjektiv, d.h. es gibt keine zuverlässige, objektive Methode
- Meinungen/Aspekte einbeziehen
 - Endbenutzer:
 - wie effektiv bei der Unterstützung von Geschäftsprozessen

- Kunden:
 - System für Kunden transparent oder wird Beziehung beeinträchtigt
- Bereichsmanager:
 - leistet System effektiven Beitrag zum Erfolg der Abteilung
- IT-Manager:
 - wie schwierig ist es, Mitarbeiter zur Bedienung/zum Betrieb zu finden
- Geschäftsführer:
 - leistet System wirksamen Beitrag zu Geschäftszielen
- Ermittlung:
 - Meinungen einholen
 - Meinungen wichten
 - Gesamteindruck/-bewertung

Die Beurteilung der Systemqualität:

- alle Systemebenen einbeziehen (1. Folie)
- Beurteilung Geschäftsprozess
 - eng mit Geschäftswert verbunden
 - Verbesserung Geschäftsprozess erforderlich?
 - Verbesserung der Unterstützung des Geschäftsprozesses erwünscht?
- Beurteilung der Umgebung (Anwendersoftware, Unterstützungssoftware (Compiler, Linker, Wartungswerkzeuge etc)) (siehe Folie)
 - wie gut, Perspektiven, Update-/Migrationsbedarf
 - wichtig, weil Umgebungsänderungen häufig sind
- Beurteilung der Qualität der Anwendungssoftware (Kriterien auf Folie)
 - Entwicklungs-QS nicht direkt einsetzbar
 - zusätzlich: quantitative Daten erheben, z.B. Anzahl Änderungen/Änderungswünsche, Anzahl unterschiedlicher Bedienermasken, Umfang verwendeter Daten
 - bei Erhebung Aufwand vs. Nutzen beachten
 - keine Richtwerte zum Vergleich
 - Alter und Größe (Umfang) einzubeziehen

3 Analyse und Assessment

- Produkt-Definition: Anforderungsbeschreibung
- Folie: Horseshoe-Modell

1 Analyse auf Systemebene

Definition 3. Eine *Metrik* ist die Bewertung von Merkmalen nach quantitativen Kenngrößen.

Merkmale für Qualität sind genormt (siehe Folie). Messbar sind dabei:

- Lines of Code

- Komplexität
- Entwicklungsaufwand
- Anzahl und Struktur der Daten

Aus Merkmalen versuchen wir Kenngrößen abzuleiten (siehe Folie Ableitung ... für Wartbarkeit).

Bemerkung 4. 80/20 - Regel:

- diejenigen Systemteile mit 80% der Probleme liegen in 20% der Programmen

2 Abdeckungsanalyse

Black-Box-Betrachtung zur Ermittlung von

- Zusammenhang Anforderungen \leftrightarrow Systembestandteilen (use cases, Szenarien durchlaufen, Profiler(ermittelt, in welchen Funktionen etc. ein Programm abläuft)
 \Rightarrow Zusammenhang mit Geschäftswert ermitteln
- Zusammenhang Anforderungen \leftrightarrow Testfälle
 \Rightarrow fehlende Testfälle, Verifikation, wo weitere Analysen notwendig

Festlegung weiterer Analysemaßnahmen (80/20-Regel).

3 Software-Visualisierung

Warum benötigen wir die Softwarevisualisierung?

Software besitzt eine hohe *Komplexität*, diese wollen wir beherrschen und nutzen deswegen Visualisierungsmöglichkeiten. Weiterhin können wir damit gut die Informationsmenge erfassen. *Softwarevisualisierung* bietet eine Lösung dafür, in dem sie relevante Informationen hervorhebt und man durch mehrere Sinne das Erfassen und Verstehen zu unterstützen.

Wir unterscheiden die Visualisierung in *statische Programmvisualisierung* und *dynamische Programmvisualisierung*.

3.1 Statische Programm-Visualisierung

Dazu gehören Dinge die „fest“ im Programm sind:

- Struktur
- Aufbau
- Metriken, z.B. *Änderungshäufigkeit*
- Abhängigkeitsvisualisierung wären hilfreich (zur Zeit aber noch nicht möglich)

Dies ist abhängig von der Programmiersprache:

- OOP: Klassen, Methoden, Paket, Projekte
- prozedural: Prozeduren, Funktionen, Module, Pakete, Projekt

Folien: „*Statische Visualisierung* {1,2}“

3.2 Dynamische Programm-Visualisierung

Zur *dynamischen Programmvisualisierung* gehört:

- Verhalten

- Abläufe
- Zeit

Ziel dieser Visualisierungstechnik ist z.B.:

- Performance-Analyse
- Diagnose bei Speicherwaltungs-Problemen („Memory-Leaks“)
- Objekt-Interaktionen (z.B. dynamisch Abhängigkeiten verdeutlichen)

Die grundlegende Aufgabe eines Visualisierungstools ist es, den Ablauf eines Programms zu erfassen. Die wird meist erreicht durch *Code-Instrumentierung* (z.B. durch *Logging*) und nachfolgende *Analyse*. Dabei gibt es jedoch verschiedene Probleme:

- Datenmenge
- Vollständigkeit („ist mein ersteller Ablauf vollständig?“)
- Verfälschung durch Instrumentierung (z.B. Verlangsamung)

4 Analyse auf Modell- und Architekturebene

Durch unsere hohe Komplexität bedingt, müssen wir Abstraktionen einführen und den Überblick erhalten bzw. ermöglichen. Häufig sind dabei Modelle nicht aktuell oder nicht vorhanden und der Code ist die einzige zuverlässige Informationsquelle \Rightarrow *Bottom-Up-Vorgehen* (vom Code ausgehend Programm verstehen).

4.1 Programmverstehen

Wir wollen *Struktur und Verhalten* erkennen, um damit die Beziehungen untereinander zu verstehen. Weiterhin benötigen wir den Bezug zur Aufgabenstellung und Anwendungsdomäne. Wichtig ist dabei, von Implementierungsdetails zu abstrahieren.

- Bezug Strukturelemente \leftrightarrow Aufgaben herstellen
dazu Aufgaben analysieren (Top-Down)
- Hypothesen aufstellen und verifizieren \Rightarrow Bezüge erfassen \rightarrow Traceability-Links
- hilfreich bei Vereinfachung/Abstraktion: etablierte Lösungen (Muster, Prinzipien (z.B. Abstraktionsschichten))

4.2 Architekturekonstruktion

Ziel dabei ist es, die höchste Abstraktionsebene zu verstehen. Wir wollen ein Modell für:

- Struktur (Komponenten, Schnittstellen)
- Verantwortlichkeiten (funktionale und qualitative Anforderungen)

Das Problem dabei ist, dass Architekturkonstrukte nicht im Code vorhanden sind. Unsicherheit von Informationen sind wie in 3.4.1 durch Traceability-Links gegeben (wir müssen anhängen, wie sicher wir uns sind).

Ablauf (Seacord 2003)

1. Sicherstellen von Informationen
 - Informationen auswählen
 - Informationen verbinden
 - Sichten schaffen (spätere Architekturbeschreibung)
2. Erkunden und Bearbeiten der Sichten

3. Muster erkennen (Abstraktionen finden)

5 Analyse auf Code-Ebene

5.1 Compilertechniken

- Scanner und Parser zur Analyse nutzen
- *Syntaxbaum* als Darstellung
 - unabhängig von Bezeichnen, Layout, Fragmentierung

5.2 Programm-Slicing

Beim *Programm-Slicing* zerlegen wir das Programm zum besseren Verständnis. Wir unterteilen grundlegend in *Forward-Slicing* und *Backward-Slicing* (siehe Folie).

5.3 Clone-Detection

Häufig wird Copy&Paste bei der Entwicklung genutzt \Rightarrow das stellt ein großes Problem dar (Code wird verdoppelt \Rightarrow wenn ein Fehler auftritt, muss dieser wohlmöglich an 2 Stellen geändert werden.)

- Fehlerkorrektur schwieriger und aufwendiger
- Wartung umfangreicher, Verstehen schwieriger
- Quelle subtiler Fehler (manchmal passt c&p nicht perfekt für den Anwendungszweck den man dann gerade hat)

Typen von Clones:

- Typ 1 - exakte Kopie bis auf Kommentare und Layout
- Typ 2 - Umbennungen, Struktur bleibt erhalten
- Typ 3 - Modifikation der Struktur (Kopie wird abgeändert)
- Typ 4 - Semantischer Klon - Implementierung unterscheidet sich, aber das Konzept bleibt gleich

4 Maßnahmen oberhalb der Codeebene

Mit Maßnahmen oberhalb der Codeebene sind vor allem Architektur- bzw. Strukturänderungen zu verstehen. Wozu sind diese Änderungen nötig?

- Performance-Anforderungen nicht (mehr) erfüllt
- Datenmengen-Anforderungen nicht (mehr) erfüllt
- Portierung, GUI-Änderung \rightarrow Einführung von Schichten
- Öffnung von Anwendungen \rightarrow Web-Services
- Integration zu „Anwendungs-Landschaften“

Maßnahmen dafür sind

- Architectural Recovery \rightarrow Modelle
- Strukturänderung \rightarrow Wrapping

1 Traceability

Damit gemeint ist die Wiederherstellung von Bezügen Anforderung - Entwurf - Implementation. Wichtig dabei sind *Traceability-Links*:

- Verweis, der den Arbeitsprozess des Entwicklers nachverfolgen lässt
- bidirektional (in beide Richtungen verfolgbar, Anforderung → Implementation, Implementation → Anforderung)
- tragen häufig Informationen zu Abhängigkeiten und Entwurfsentscheidungen
- können nur manuell erstellt werden bis heute

Diese Traceability-Links werden in der Praxis verwendet:

- Verfolgen von Anforderungen auf hoher Abstraktionsebene (Testfälle prüfen und konstruieren)
- nicht tiefer verfolgbar, da die Anzahl der Links hoch ist und die Prüfung der Links einen hohen Aufwand erfordert

Aufstellen von Links bei *Architectural Recovery*:

- Analyse von Dokumenten (Systemdokumente, Wartungshandbuch, Architekturdokumente)
- Erstellung einer ersten Architekturbeschreibung (wenig vertrauenswürdig und vollständig)
- Code-Analyse
 - Aufstellung von Hypothesen über Bezug zu Entwurf und Anforderungen → Traceability-Links
 - Verifikation: Code-Analyse, Tests, Profiling
 - Ergänzung der Architekturbeschreibung

Dieser Prozess wird iterativ immer wieder von vorn betrachtet bis man einen gewissen Stand der Code-Analyse erreicht wird.

2 Wrapping

Definition 5. *An Object Wrapper provides access to a legacy system through an encapsulation layer. The encapsulation exposes only those attributes and operation definitions desired by the software architect.*

Ziele des Wrappings sind:

- neue Schnittstelle ohne Änderung des Legacy-Systems
- Alternative zu Änderung und Neuentwicklung

Arten

- Daten-Wrapping (siehe Folien)
 - benötigt, wenn Daten und Code parallel zu migrieren
 - Synchronisieren zwischen modernisierten und Legacy-Datenbeständen
 - Mapping der Datenmodelle (-Strukturen)
 - Datenreplikation (Kopieren, Pflege über mehrere DB hinweg)
- Hybride Ansätze
 - Kombinationen aus Synchronisation, Replikation
 - separate Techniken je Problem

- Logische Adapter:
 - Umsetzung von Bedingungen und Zuständen
 - Ziel: Geschäftslogik arbeitet korrekt

5 Reengineering-Maßnahmen auf Codeebene

1 Refactoring

Beim Refactoring bezeichnet die manuelle oder automatisierte Strukturverbesserung von Programmquelltexten (Ändern der internen Struktur, um Software leichter verständlich zu machen und einfacher zu ändern/erweitern). Ziele sind dabei:

- Qualität des Entwurfs verbessern
 - klare Struktur
 - Duplikate entfernen
- Verbessern der Verständlichkeit des Codes
 - Menge der notwendigen Details verringern
 - klarere Darstellung von Konzepten
- Unterstützen des Findens von Fehlern (z.B. *defensives Programmieren* (Prüfen übergebener Parameter auf Gültigkeit (keine negativen Werte vor Wurzelziehen etc.))
- Erhöhung der Produktivität/Effektivität der Entwicklung

Wann wird ein solches *Refactoring* angewendet?

- als Zusatz zum Erweitern
- bei einem Codereview wurden Verständlichkeitsprobleme festgestellt
- wenn Fehler gesucht/gefunden werden, können dabei Strukturänderungen vorgenommen werden

Wichtig dabei ist, dass *Refactoring* Risiken verringert und nicht erhöhen darf. Der Ablauf sieht meist folgendermaßen aus:

- Analyseschritt (Codeinspektion von Hand) - *Bad Smells* (aus XP - wir suchen nach Stellen, die „nicht sauber“ sind). Folie „**Ablauf Refactoring**“
- Expansion
- Zusammenfassung (Expansion und Zusammenfassung sind Funktionserhaltend)
Dabei kann JUnit etc zur Überprüfung der Korrektheit benutzt werden

Wesentlich dabei ist, dass jeder Schritt mit definierten Änderungen verbunden ist. Von diesen Änderungen kann man nachweisen, dass sie die Funktion nicht beeinflussen (siehe Folie „**Refactoring-Katalog**“). *Automatische Tests* sollten dabei möglichst genutzt werden. Beispiele für „Bad Smells“ sind z.B.:

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change (eine Änderung zieht viele Folgeänderungen mit sich)
- Shotgun Surgery (kleinere Schäden an vielen Stellen durch kleinen Fehler/Änderungen - Änderungen sind überall verstreut)

- Parallel Inheritance Hierarchy

Wie gehen wir nun damit um, wenn wir so etwas finden?

Beispiel 6. Auf Long Method können wir mittels

- Extract Method (Teil einer Methode nehmen und in eigene Methode setzen)
- Replace Temp with Query (temporäre Variable kommt immer wieder vor \Rightarrow durch Instanzvariablen ersetzen)
- Replace Temp with Method Object (temporäre Variable durch Objekt ersetzen)
- Decompose Conditional (lange switch/case zerlegen)

reagieren.

2 Clone Consolidation

Eng verbunden mit Refactoring. Vorgehen dabei ist auch „Expansion&Zusammenfassung“.

3 Modularisierung

Ziele der Modularisierung sind ähnlich zu Zielen des Refactorings, primär die *Neuorganisation* eines Programms

- Zusammenfassen
- als Modul kapseln
- Interaktionen optimieren
- Redundanzen leichter entfernen

Module zeichnen sich aus durch

- Kapselung (*information hiding*)
- hohe Kohäsion (gleiche Dinge bleiben zusammen)
- geringe Kopplung

Für uns wichtige Modularten sind

- Datenabstraktionen \rightarrow global benutzte Bereiche entkoppeln
- Funktionsmodule (Module haben ähnliche Funktion)
- Hardwaremodule
- Module zur Geschäftsprozessunterstützung

4 Restrukturierung des Steuerflusses

- Struktur des Steuerflusses muss Struktur der Aufgabe entsprechen
 - vom Aufbau oder vom Ablauf
- Kommentare leisten allein keine ausreichende Verbesserung

Regel von Boehm und Jacopini (1966): „Jedes Programm kann mit *if/then/else* und *while* statt *goto* ausgedrückt werden“.

Automatische Code-Restrukturierung kann jedoch zu einigen Problemen führen:

- Verlust von Kommentaren

- Verlust von Bezügen zur Dokumentation
- Hohe Anforderungen an Rechenleistung

Eine Möglichkeit des Auswegs stellt dar:

- nur gezielt Komponenten überarbeiten

5 Datenabstraktionen und Daten-Reengineering

- Änderung durch Modularisierung → Abstraktionen bilden
- auch bei Daten-Inkonsistenzen
 - durch inkonsistente Datenupdates → Datenpflege betreiben
- Einschränkungen von Datenmenge und des Wertebereichs
- Weiterentwicklung der Architektur

In der Praxis ist der Begriff *Datenverschlechterung* gängig und bezeichnet

- Inkonsistente Standardwerte
- Inkonsistente physikalische Einheiten
- Inkonsistente Darstellung und Semantik, z.B. Validierungsregeln (z.B. Behandlung negativer Werte)

6 Managment und Organisation

1 Bezug zum Projektmanagment

- Zusammenfassung von Techniken zur Steuerung von Projekten entspricht dem Begriff **Projektmanagment**
- 3 Aufgaben (siehe Folie „Wiederholung Projektmanagment“)
 - Koordination
 - Projektführung
 - Tätigkeitsvoraussetzung (hä?)
- 3 Tätigkeiten

1.1 Planung und Durchführung von Reengineering-Projekten

- Reengineering-Projekte nicht unbedingt gleichzusetzen mit Entwicklungsprojekten
- Legacysysteme:
 - wertvoll wegen Geschäftswert (aber ständige Verbesserung nötig), Änderungen notwendig
 - Ziele:
 - Komplexität verringern (des Systems)
 - Kosten begrenzen, Kostenrahmen einhalten
 - Risiken verringern
 - Verständlichkeit und Wartbarkeit verbessern

- Zeitverhalten verbessern
- Portierung
- möglicherweise Umstrukturierung machbar machen

1.2 Kosten/Nutzen

- verlässliche Angaben zu Kosten soll ermittelt werden (erfordert natürlich genaues Wissen über die Aufgabe) → Kostenschätzung
- Nutzen bewerten
 - Geschäftswert
 - Systemqualität
 - → 4 Quadranten
- Aufwand ist abhängig von:
 - Wartungseigenschaften mit folgenden Faktoren:
 - breite Schnittstellen
 - hohe Komplexität
 - geringe Verständlichkeit
 - schlechte Struktur
 - erhöht den Aufwand deutlich
 - Anzahl (+ Umfang, Menge) der Änderungen
 - Vorhersagemöglichkeit von Traceabilitylinks
 - Vorhersagemöglichkeit von Schätzungen → insbesondere bzgl. Abschätzungen

1.3 Risikomanagement und Entscheidungsprozess

- wichtigster Punkt *Risikominimierung*:
 - Änderungen nur dort durchführen, wo sie unbedingt notwendig sind → schwierige Entscheidung für die Entwickler
- Einflussgrößen und Ziele sind oft unklar
- strukturierten, nachvollziehbaren Entscheidungsprozess durchführen
 - Folie „Vorgehen bei Entscheidungsfindung bei Unsicherheit“ (Präferenzen des Entscheiders → Richtung angeben)
 - 3 grundlegende Schritte + 4. Schritt: Entscheidung implementieren
 - Widerspruchslösungsdurchführung: Zerlegung der Ziele in Unterziele und priorisieren dieser
 - Alternativen bewerten bzgl. Zielen
 - Ziel: sichere Entscheidung treffen → **Risikominimierung!**
- Änderungs- bzw. Reengineering-Kandidaten
 - Auswahl nach Aufwand und Nutzen
 - Technik: 4 Quadranten (Abschnitt 2) → Geschwindigkeit, Systemqualität

1.4 (Änderungen/) Change Management

Notwendig ist es, Abhängigkeiten klar darzustellen → Traceability Links.

2 Migrationsstrategien

siehe Folie 6

3 Teststrategien

siehe Folie 7

4 Einbindung Prozessmodelle

- Folie 8
- V-Modell
- CMMI

5 Teststrategien

Testen von OO-Software

- Kapselung verringert Testbarkeit (kein direkter Zugriff auf Variablen, Zustand eines Objektes kann nicht direkt eingestellt werden)
 - Zustand erkennen
 - Zustand erreichen (Test-Ausgangszustand)
- Fehlerbehandlung verringert Testbarkeit (defensives Programmieren - immer vom schlechtesten Fall ausgehen)

Typische Testfälle sind:

- Klassentest (Testen der Methoden eines Objekts)
Methoden mit
 - einfachen Kontrollstrukturen
 - keine großen Abhängigkeiten voneinander, aber starke Verflechtung über die Attribute
- Zustandsbasierter Test
 - auch möglich für mehrere Klassen (Systeme von Klassen)
 - Zustandssequenzen werden durchlaufen, werden durch Zustandsautomat spezifiziert
 - minimale Testabdeckung:
 - alle Zustände werden abgedeckt
 - besser: alle Zustandsübergänge durchlaufen

6 Werkzeuge

Ziele:

- Routinetätigkeiten automatisieren
 - Fehlervermeidung
 - Aufwandsverringerung

- “geht schneller“
- Vereinfachung von Tätigkeiten
 - Visualisierung unterstützt Verstehen
 - Unterstützt Beherrschung der Komplexität (große Mengen an Informationen)
- verbesserte Kontrollmöglichkeit
 - Protokollierung
 - Arbeitsteilung: Entkopplung und Parallelität von Aufgaben

Möglichkeiten der Automatisierbarkeit:

Aktionen	automatisierbar	nicht automatisierbar
Forward Eng.	Teile der Codierung	Entwurf, Entscheidungen
Restrukturierung	Clone Detection, Realisierung	Finden, Entscheiden
Reverse Engineering	statische Analysen, Strukturanalyse	Herstellen von Bezügen
Dokumentation	Integration, Konv., Aktualisierung	Inhalte erstellen
Integration	Zusammenführung	Auswahl/Entscheidung Konfiguration
Test	Testdurchführung	Testfallerstellung

Tabelle 1.

Sinnvoller Einsatzpunkt von Werkzeugen:

- große Daten
- viele Wiederholungen einfacher Tätigkeiten

Maßnahmen zur Aufrechterhaltung von Modellen und Dokumentation:

- erstelle Richtlinien für Abnahmen und Kontrollen
- Motivation der Entwickler, u.a. Modell-Nutzung für Generatoren
- automatisierte Aktualisierung, Werkzeugunterstützung
 - Verknüpfung Realisierungs-Modell mit Traceability-Links
 - integrierte Dokumentation
- Hilfsmittel:
 - Template, Formulare

Chancen von Modellen:

- auf Basis von Modellen, können wir viele Fehler vermeiden
- Modelle nutzen für Testbasis etc.

Risiken:

- vollkommene Abhängigkeit vom Modell/Tools (proprietäre Modelle)
- Fehler im Werkzeug ⇒ Probleme?!

Langfristige Verbesserung des Reifegrades:

- “Wie reif ist eine Organisation?“
- Dokumentationsbasis verbessern ⇒ schrittweise
- Architekturqualität verbessern ⇒ schrittweises Refactoring
- Modellqualität verbessern