Einführung in die Softwaretechnik
TUM Sommersemester 2011
Dozent: Florian Matthes

Janosch Maier

7. August 2011

Inhaltsverzeichnis

1 Einführung und Überblick

Software, die eine Menge von Funktionen anbietet und auf einer vorgegebenen Hardware laufbar ist, zur Lösung einer festgelegten Aufgabe, bestehend aus ausführbaren Programmen, Ressoucren und Anwenderdaten

Klassifikation von Software

- Gelöste Problemstellung
- Programmiersprache
- Hardware
- Betriebsumgebung
- Qualitätseigenschaften
- Vermarktung
- Nutzergruppe

Softwareengineering Ingeneurmäßiges Vorgehen beim Entwickeln von Software. Kosten ↔ Qualität ↔ Termine. Fachlichkeit, Softwaretechnik, Projektmanagement.

Einsatz wissenschaftlicher Methoden (Theorien, Modelle), Wirtschaftlichkeit, Nutzen, Beherrschung von Terminen, Qualität, Kosten

Projektmanagement, Anforderungsmanagement, Softwarearchitektur, Teilintegration, Qualitätssicherung, Erprobung, Wartung

Fehleranfälligkeit durch Komplexität, Unkenntniss, Interessenskonflikt

Standardsoftware vs. Individualsoftware

Standardsoftware Für Markt entwickelt, Vertrieb, breiterer Einsatz, Standardgeschäftsprozesse, Anpassung

Individualsoftware Für Unternehmen entwickelt, spezieller Prozess, Ergebnis eines Projektes für Auftraggeber, Individuelle Pflege und Anpassung

2 Requirements Engineering

Finden, Bewerten, Dokumentieren, Strukturieren und Verwalten von Anforderungen.

Fehler passieren am häufigsten während des Requirements Engineerings und des Design; je später sie entdeckt und entfernt werden, desto teurer sind sie.

2.1 Anforderung

Eigenschaft, die ein **Stakeholder** fordert, **vertraglich** festgehalten

2.1.1 Anforderungsermittlung

Lastenheft → Pflichtenheft → Anwendungsfälle → Konzeptuelles Klassendiagramm

2.1.2 Arten von Anforderungen

Funktional Drückt Funktion aus. Formulierbar, als Aktivität. *Wenn der User Drucken klickt, muss gedruckt werden*

Nicht Funktional Beschreibt Eigenschaft des Systems, die keine Interaktion ist. Formulierbar als negative Zusicherung. *Antwortzeit muss unter einer Sekunde liegen*

Bsp: Verwendbarkeit, Robustheit, Wartbarkeit, Verfügbarkeit, Performanz
→ Messbarkeit muss sichergestellt sein!

Beschränkung Bestimmt Lösungsraum *Die Implementierung muss in Java erfolgen*

2.1.3 Qualitätseigenschaften

Ist die Anforderung korrekt? Verständlich? Eindeutig? Testbar?
Priorisierung: Muss-, Soll-Anforderungen; Anforderungen niedriger Priorität
Zielkonflikte:

- Funktionsumfang ⇔ Benutzbarkeit
- Kosten ⇔ Robustheit
- Effizienz ⇔ Portabilität
- Entwicklungsdauer ⇔ Funktionsumfang
- Wiederverwendbarkeit ⇔ Kosten
- Abwärtskompatibilität ⇔ Wartbarkeit

2.2 Lasten- und Pflichtenheft

Lastenheft	Pflichtenheft
Ziele, Anforderungen, Rahmenbedingungen	Detaillierte Anforderungen
Projektplanung	Detaillierte Projektplanung
abstrakt	konkret, messbar
vom Auftraggeber erstellt	vom Auftragnehmer erstellt
Teil der Ausschreibung	Vertragsbestandteil

2.2.1 Inhalt:

Lastenheft	Plichtenheft
Zielbestimmung	Zielsetzung
Produkteinsatz (Akteure + Anwendungsbereiche)	Allgemeine Beschreibung (Systemgrenzen, Generelle Funktion, Benutzer)
Produktumgebung (benachbarte Systeme)	Gegenwärtiges System (falls Vorhanden)
Produktfunktionen (funktionale Anforderungen)	Vorgeschlagenes System (Anforderungen, Systemmodell)
Produktdaten (erwartete Datenmenge)	
Produktleistung (nicht funktionale Anforerungen)	
Qualitätsanforderungen (Priorisierung d. Anfd.)	
	Erwartete Evolution
Glossar	Glossar

Anforderungen müssen eindeutig identifizierbar sein (Nummerierung)
Spezifikation muss strukturiert, vollständig und widerspruchsfrei sein.

2.2.2 Alternatives Requirements Engineering

Bei kleinem Team, wenig Requirements Engineering möglich. Beispiel: Story Cards mit User Stories.

2.3 Anforderungs- und Systemanalyse mit UML

Szenario Interaktion zwischen System und Akteuren

Beispielszenario Konrete Interaktionsfolge

Typszenario Menge möglicher Interakionsfolgen

Use Case Typszenario, welches von einem Akteur angestoßen wird

Akteur Mensch mit Zielen gegenüber des Systems / Rolle, die ein Mensch gegenüber dem System hat

2.3.1 Beispielszenario

Freier Text, flexibel, unpräzise, leicht missverständlich, Fehler leicht zu übersehen

2.3.2 Typszenario

- Name:
- Beschreibung
- Akteure
- Auslöser
- Vorbedingung

- Nachbedingung
- Ablaufschritte
- Alternative Ablaufschritte

2.3.3 Sichten eines Systems

- Struktursicht
  - Statische Struktur (Daten-, Klassen- und Objektmodelle)
  - Struktur und Zusammenhang von Komponenten
  - Protionierung in Pakete und Subsysteme
  - Physische Struktur
- Verhaltenssicht
  - Interaktion externer Akteure mit System
  - Zeitlich-dynamisches Verhaltenssicht
  - Interaktion ausgewählter Objekte
  - Aktivitäten und Ablauf

2.3.4 Anwendungsfall

Wähle ein oder mehrere Szenarien und abstrahiere diese.

- Aktiv verwenden (Akteure)
- Kausale Folge der Schritte
- Kompletter Ablauf
- Systemgrenzen + externe Komponenten
- Glossar

TBD Anwendungsfalldiagramm Beispiel

2.3.5 Objekte und Klassen

Klassen, Attribute, Methoden, Beziehungen, Kardinalitäten, Vererbung identifizieren

Klassendiagramm Beschreibung eines Schemase

Objektdiagramm Beschreibung eines Szenarios / Beispiels
TBD Klassendiagramm Objektdiagramm
vgl. UML Kompakt

Abbott's Technique – Klassen identifizieren

1. Anwendungsfall wählen
2. Nomen → Klassen / Objekte; Verben → Methoden
3. Klassen / Objekte Trenne
4. Klassen Typisieren (Entity (Speicherung), Boundary (Schnittstelle) und Control (Beobachtete Vorgänge) Objects)

⇒ Konzeptuelles Klassendiagramm

2.3.6 Zusammenfassung

Szenarien → Benutzersicht; Objekt-/Klassenmodelle → Struktur, Daten, Operationen ⇒ Verknüpfung nötig.

3 Architekturentwurf

- Entwicklung (≠ Implementierung) einer softwaretechnischen Lösung bestehend aus Softwarearchitektur und Systemkomponenten (mit Schnittstellen nach Außen)
- Softwarearchitektur ist abstrahiert Struktur, Verhalten und Verteilung des Systems, welche Komponenten und deren Beziehungen beschreibt.
- Beschränkungen durch Programmiersprache, Fachlichen Standards, Altsystemen, etc.

3.1 Dekomposition in Bausteine

- Geringe Kopplung (zwischen Bausteinen)
- Hohe Kohäsion (Nahe Elemente im selben Baustein)
- Zulänglichkeit (Baustein erfüllt Aufgabe)
- Vollständigkeit
- Einfachheit

Bei hoher Kopplung zweier Komponenten evtl. Zusammenfassung in eine Komponente sinnvoll. Information-Hiding zur Linderung von Kopplung.

3.2 Konzepte des Architekturentwurfs

3.2.1 Komponente

Besitzt Schnittstellen, Status, Adressraum, Lebenszyklus, Identität. Kann aus anderen Komponenten bestehen.

3.2.2 Schnittstelle

Grenze zwsischen Komponenten. Beschreibt Informationsfluss zwischen Komponenten. Schnittstellenklasse über Entwurfsmuster Fassade möglich

3.2.3 Design by Contract

Informationhiding. Zugriff auf Komponente nur über Schnittstelle. Implementierung der Komponente unabhängig von Anwendung.

Komponentenschnitstelle (Architekturentwurf) ⇔ API (Implementierung)

3.2.4 Kompatibilität von Methoden

- Syntaktisch: Importierendes Klasse: Aufruf einer Methode T' subtype T
- Semantisch: Exportierende Klasse / Unterklasse: Vorbedingung ist weniger strikt, Nachbedingung ist strikter

3.2.5 Sichten

- Black-Box: Nur Schnittstelle bekannt
- White-Box: Implementierung bekannt

3.3 Architekturdokumentation

UML zur Notation von Softwarearchitekturen

3.4 Softwarearchitekt

Schwerpunkt auf objektorientiertem Entwurf, nicht auf Platform oder Technologie

3.4.1 Kommunikator

- Programm-Level: Programmierer, Tester, etc. ⇒ Technische Fähigkeiten
- Softwarearchitektur-Level: Softwarearchitekt ⇒ Abstraktions- / Kommuniakationsfähigkeiten
- Projektmanagement-Level: Projektmanager / Gruppenleiter ⇒ Geschäftskenntniss + soziale Kompetenz

3.5 Architekturmuster

Beschreiben erprobte Lösungsideen häufiger Entwurfsprobleme (Grob-/Feinentwurf). Verwandt mit Entwurfsmustern (implementierungsnah).

3.5.1 Wichtige Architekturmuster

- Repository: Zugriffssteuerung und Protokollierung bei mehreren Benutzern mit Schreibzugriff auf gemeinsame Datenbanis
- Model-View-Controller: Schichten zur Datenerhaltung (Model), Darstellung (View) und Steuerung (Controller)
- Schichten-Architektur: Aufteiung in Schichten, Zugriff nur auf benachbarte Schichten

- Client/Server: 2-Schichten-Architektur
- Strukturelle Entwurfsmuster

  Fassade: Eine Schnittstelle für System aus mehreren Komponenten

  Kompositum: Komponenten, die Instanzen von sich enthalten

3.5.2 Schichten und Partitionen

Schicht ist Systembestandteil, der Dienste anbietet. Schicht ist nur von darunter liegenden Schichten abhängig. Schicht hat keine Kentniss höher liegender Schichten.

Schicht kann aus mehreren Komponenten bestehen ⇒ Partitionen (meist schwach gekoppelt)

Strenge Schichtenarchitektur vs. Offene Schichtenarchitektur Zugriff nur auf direkt darunter liegende Schicht (Wartbarkeit + Flexibilität) ⇔ Zugriff auf alle darunter liegenden Schichten (Performanz)

3-Schichtenarchitektur Darstellungsschicht ↔ Anwendungsschicht ↔ Datenzugriffsschicht

3.6 Architekturentwurf

1. Analyse + Grobentwurf (Datenflussdiagramme)
2. Grobentwurf (Komponentendiagramme)
3. Feinentwurf (Klassendiagramme)

4 Objektorientierter Entwurf von Subsystemen

4.1 Überblick

Implementierungsnahe Klassendiagramme + Zustands- / Interaktionsdiagramme. CRC-Cards und Entwurfsmuster als Hilfsmittel

4.2 Dynamische Modellierung mit UML

vgl. UML Kompakt

4.3 CRC Cards

- Verantwortlichkeiten (abstrakte Beschreibung des Zwecks einer Klasse)
- Zusammenarbeit (Beziehung mit anderer Klasse zur Erfüllung der Verantwortlichkeit)

4.4 Entwurfsmuster

4.4.1 Einführung

Dokumentation von bewährten Problemlösungen

Klassifikation Aufgabe (Erzeigungsmuste, Strukturmuster, Verhaltensmuster), Gültigkeitsbereich (Klassenbasiert, Objektbasiert)

4.4.2 Kompositum (Composite)

Component, Composite, Leaf werden zu Baumstruktur zusammengefasst. ⇒ Betrachtung von Objekten (Leaf), als auch Kompositionen (Composite) von Objekten gleichermaßen (Operationen von Component) möglich

4.4.3 Strategie

Zusammenfassung von Klassen, die sich nur in Verhalten unterscheiden (Unterschiedliche Varianten eines Algorithmus)

4.4.4 Singleton

Es gibt genau ein Objekt dieser Klasse. Zugriff über getInstance()

4.4.5 Fassade

Bereitstellung einer Klasse (Fassade), welche eine einfach Schnittstelle für mehrere (komplizierte) Klassen bereitstellt

4.4.6 Observer Pattern

Observable Objekte informieren Observer über Änderungen in ihrem Status

5 Implementierung

5.1 Forward Engineering

Entwicklung von Code ausgehend von Modellen

5.2 Ingeneurmäßige Programmierung

5.2.1 Pragmatischer Programmierer

Vermeidet (erzwungene, unbeabsichtete, ungeduldige, durch mehrere Etwickler entstehende) Doppelung

5.2.2 Implementierungsmuster

- Aussagekräftiger Name (einer Methode)
- Methodensichtbarkeit überprüfen
- Zugriffe durch add() / remove() / count(), anstatt Zugriff auf gespeicherte Liste
- valid() / invalid() statt setValid(boolean newState)

5.2.3 Refactoring

Refactoring fügt keine neuen Funktionen hinzu.

Bad Smells

- Duplikate
- Lange Methoden
- Große Klassen
- Lange Parameterliste
- Switch Konstrukte
- Parallele Vererbungshierachien

Mögliches Refactoring

- Methoden extrahieren
- Temporäre Variable durch Methodenaufruf ersetzen
- Methode in andere Klasse verschieben
- Attribut in Vererbungshierachie anheben

6 Testing

6.1 Einführung

6.1.1 Fehler

- Fehlerwirkung / Failure (Fehlfunktion): Falsches Programmverhalten (Erkennbar durch Testing)
- Fehlerzustand / Fault (Bug): Ursache für Fehlerwirkung, führt nicht zwingend zu Fehlfunktion
- Fehlerhandlung / Error: Handlung, die zu Fehlerzustand führt (Programmierfehler)

6.1.2 Fehlerarten

Modulfehler Modul entpsricht nicht Spezifikation → Modultest

Architekturfehler Zusammenspiel der Module lässt Fehler auftreten → Integrationstest

6.1.3 Testing

Zeigt Fehler oder ist Indiz für Korrektheit. Vollständiges Testen meist nicht möglich

Testen Identifizierung von Fehlern; Automatisiert möglich; Testteam

Debugging Fehlerbeseitigung; Manuell, Programmierer

6.2 Terminologie

Testobjekt	Zu testendes System
Tesfall	Eingabe und Soll-Resultate des Testobjektes
Testdaten	Eingaben des Testfall
Test-Suite	Menge von Testfällen
Coverage	Abdeckung von Fällen einer Test-Suite

6.3 Modultest

Auswahl von repräsentativen, fehlersensitven, redundanzarmen und ökonomischen Testfällen ⇒ Möglichst wenig Testfälle zum finden möglichst vieler Fehler.

Einzelene Softwaremodule werden einzeln getestet. Erste Testphase nach Programmierung.

6.3.1 Funktionsorientierter Test (Black Box)

Testen der spezifizierten Funktionalität; Möglichst viele Eingaben auf korrekte Bearbeitung testen; Code nicht einsehbar; (auch für Integrations-/Systemtest)

Äquivalenzklassenbildung: Testfall aus Repräsentanten

Grenzwertanalyse : Max / Min Werte, Spezialfälle wie 0, etc.

Statistisches Testen : Zufällige Auswahl von Eingabedaten (+ Soll-Ausgaben), Umfangreiche Tests

Error Guessing : Tester versucht Fehler zu raten; Stark von Qualität des Testers abhängig

6.3.2 Strukturorientierter Test (White Box)

Testfälle aus Programmcode; Strukturelle Überdeckung testen; (Nur Modultest + evlt. Integrationstests)

Anweisungsüberdeckung : Jede Anweisung des Kontrollflussgraphen wird einmal aufgerufen

Zweigüberdeckung : Alle Kanten des Kontrollflussgraphen werden durchlaufen

Pfadüberdeckung : Meist Unmöglich, da Schleifen zu undendlich vielen Pfaden führen

Variablennutzung : Abdeckung von Variablendefinition / Berechnung / Bedingung (unüblich)

6.3.3 Automatisierung

Zum Beispiel durch jUnit, jcoverage

6.4 Integrationstest

6.4.1 Integration

- **Bottom-Up** Integration der Module nacheinander

  Fehler gut findbar, aber viele Umgebungen benötigt; Testtreiber für jede Umgebung

- **Dummies** Gesamtsystem mit Dummies modelliert

  Fehler gut findbar, aber Dummies benötigt; Nur ein Testreiber nötig, aber Dummies

- **Big-Bang** Alle Module direkt zusammengefügt

  Einfacher Integration, Fehler schwer zuzuordnen, Alle Module müssen fertig sein

- **Ad-hoc** Integration der Komponenten in Reihenfolge der Fertigstellung (zufällig)

  Zeitgewinn, Testtreiber und Dummies benötigt

6.4.2 Testziele

- Funktionsfehler: Aufruf falscher Komponenten
- Schnittstellenfehler: Übergabeparameter passen nicht zusammen
- Fehler im Datentausch: Falsche Interpretation von Daten, Zeitliche Fehler, …

6.5 Systemtest

Test des komplett integrierten Systems (aus Benutzersicht)

Funktionale Anforderungen Anforderungen als Testbasis. Ableiten von Systemtestfällen. (Alpha-Test)

Nicht Funktionale Anforderungen Stresstest, Lastmenge, Performanztest, Konfiguration, Kompatibilität, Sicherheit, Betriebsumgebung, Qualitätstest, Wiederherstellung

6.6 Akzeptanztest

Test auf Vertragliche Akzeptanz + Test auf Benutzerakzeptanz (Mehrere Benutzergruppen, Kunde $\neq$ Benutzer, etc; Beta-Test)

7 Qualitätssicherung

7.1 Softwarequalitätsmodelle

7.1.1 Produktqualität

Transzendenz (Erfahrung), Produktzentriert (Vergleich), Benutzerzentriert (Bedürfnisse), Herstellerzentriert (Qualitätsanforderungen), Wertzentriert (Preis)

7.1.2 Softwarequalität

ISO 9126

- **Funktionalität** Funktionen implementiert und funktionierend
- **Zuverlässigkeit** Fehlertolerant, Robust
- **Effizienz**
- **Benutzbarkeit** Verständlich, Attraktiv, Erlernbar, Bedienbar
- **Änderbarkeit** Analyse, Modifizierung
- **Übertragbarkeit** Anpassbar, Installierbar (auf verschiedenen Systemen)

Andere Qualitätsmodelle

- Verlässlichkeit
  - Safety (Technische Sicherheit)
  - Reliability (Zuverlässigkeit)
  - Security (Informationssicherheit)
    - Availability (Verfügbarkeit)
    - Confidentiality (Vertraulichkeit)
    - Integrity (Unversehrtheit)
  - Maintainability (Wartbarkeit)

7.2 Methoden der Verifikation

Validierung Richtiges System?

Verifikation System richtig?

7.2.1 Inspektion und Code Review

Reviews sind bei geringem Zeitaufwand effektiv ($\frac{1}{3}$ der Fehler werden gefunden); Früh im Entwicklungszyklus möglich.

Lesetechniken

- Checklist-Based Reading (Coding Standards)
- Scenario-Based Reading

  Perspective-Based Reading (Rollen)

  Defect-Based Reading (Fehlerklassen)

7.2.2 Automatisierte statistische Analyse

Zum Beispiel: Automatisches Auffinden von Bug-Patterns.

Model Checking Anforderung formulieren (Nie beide Ampeln einer Kreuzung auf grün), alle erreichbaren Zustönde darauf überprüfen.

7.2.3 Verifikation durch Beweis

Sehr aufwändig aber grundsätzlich für alle Programme möglich.

7.3 Software-Metriken

7.3.1 Laufzeitmetriken

- Transaktionszeit
- Mean-Time-Between-Failure (MTBF)
- Verfügbarkeit

7.3.2 Designmetriken

- Depth of Inheritance
- Number of Children

7.3.3 Codemetriken

- Lines of Code (LOC)
- McCabes Zyklomatische Komplexität V eines gerichteten Graphen; e = Edges, n = Knots, p = Nicht verbundene Komponenten

$$V(G) = e - n + 2p \tag{1}$$

7.3.4 Objektorientierte Designmetriken

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT): Wiederverwendung, Wartungsaufwand
- Number of Children (NOC): Einfluss der Klasse, Wiederverwendung, Abstraktionsgrad
- Coupling between Objects (CBO): Modularisierung, Wartungsaufwand

- Response for a Class (RFC)
- Lack of Cohesion in Methods (LCOM): Kapselung; LCOM = (Methodenpaare ohne gemeinsame Variablen) - (Methodenpaare mit gemeinsamen Variablen) ≥ 0
- Verhältnis Statischer Methoden zu Anzahl aller Methoden

$\Rightarrow$ Design- und Codemetriken zeigen keine Fehler, aber zeigen wo Review sinnvoll (vor allem bei „Ausreißern“)

7.4 Prozessqualität

Qualität des Entwicklungsprozesses wichtig für Produktqualität. Qualitätsmesspunkte während des Entwicklungsprozesses

7.4.1 CMMI (Capability Maturity Model Integration)

Level des Softwarereifegrads: Initial, Managed, Defined, Quantitatively Managed, Optimized

8 Software-Wartung und Evolution

8.1 Relevanz

Verbesserung von entdeckten Fehlern, Hinzufügen von Funktionen bei Änderung der Anforderungen (ca. $\frac{3}{4}$ aller Change Requests); Wartungskosten übersteigen im Allgemeinen nach einigen Jahren die Entwicklungskosten.

8.2 Terminologie

8.2.1 Softwarevolution

Umfasst Softwareentwicklung und **Softwarewartung**

8.2.2 Reengineering

- Reverse Engieering (Informationsanalyse) [Ablöse: Reverse Engineering + Forward Engineering]
- Restrukturierung (Nachträgliche Modularisierung)
- Transformation (Änderung der Programmiersprache)
- Migration (Anpassung für andere Platform)

8.2.3 Wartungsfälle

- Perfective Maintenance (Anforderungsänderung)
- Adaptive Maintenance (Technologieänderung)
- Corrective Maintenance (Fehlerbeseitigung)
- Preventive Maintenance (Qualitätsverbesserung)

8.3 Herausforderungen

Software wird „alt“. Probleme der Wartung vorwiegend nicht technisch (Fähigkeiten der Entwickler, Termine, Dkumentation, etc.)

Meist Intuitiv (durch Entwickler), Wartung durch Spezialistenteam u.U. effizienter.

Wartbarkeit (Analysierbarketi, Änderbarkeit, Stabilität, Testbarkeit, Vorgaben-Konformität) hat großen Einfluss auf Wartungskosten, aber schwer zu prüfen.

8.4 Wartungsprobleme

vgl. Bad Smells

9 Versions- und Konfigurationsmanagement

9.1 Konfigurationsmanagement

Konfiguration ist Zusammenstellung zusammenpassender Versionen. Speicherung von Projektplänen, Konfigurationen, Design, Software, Testsuiten, Changerequests.

Konfigurationsdatenbank bietet Informationen über KM-Prozess (Wieviele Changerequests für eine Version, etc.)

9.2 Releasemanagement

Release ist für Kunden freigegebene Version. Ausführbarer Code, Konfigurationsdateien, Installer, Dokumentation.

Erstellen von Releases und Dokumentieren

9.3 Änderungsmanagement

Wie werden Changerequests behandelt?