Software Engineering
Zusammenfassung

Bernhard Fischer
Andreas Rain

4. Februar 2012

Inhaltsverzeichnis

Teil I

Software Engineering

1 Software Crisis

Typen von Software-Problemen

- Fehlschlagen von Software-Projekten
- Fehler von Software-Systemen

Umfrage IEEE 1995

- 30% Software-Porjekte abgebrochen
- 50% sind über dem Budget
- nur 60% der Funktionalitäten sind im Endprodukt

Unterscheidung zwischen Fault und Failure

- **Fault:** Fehler, auch Error, wird beim Programmieren oder beim Software-Design verursacht, welcher dazu führt, dass die Software in einen unerwünschten Zustand gerät.
- **Failure:** Wird oft weitaus später, als beim tatsächlichen auftreten entdeckt. System verhält sich dann nicht wie spezifiziert (unerwünschte Ergebnisse).

1.1 Ergebnis: Software Crisis

- Software-Produkte werden spät geliefert
- Software-Projekte übersteigen das Budget
- Gelieferte Software erfüllt oft nicht seinen Zweck
- Software-Produkte sind defekt bei Lieferung
- Große Projekte werden vor Lieferung abgebrochen

2 Software

2.1 Charakteristiken von Software

- Software wird entwickelt
  - meistens angepasst an den Kunden
  - kleineres „Komponenten-Zusammenfügen“
- Software verschleißt nicht, kann aber veralten..
- Software ist komplex

- Software ist ein bestimmender System-Faktor

Warum ist Software schwierig zu produzieren?

- Es wurde bisher kein ähnliches System entwickelt
- Die Anforderungen sind nicht eindeutig
- Anforderungen ändern sich
- Komplexe Interaktionen zwischen Services und Komponenten
- Natur der Systeme, bsp. integriert oder Informationssystem
- Software ist leicht verformbar
- Software ist ein diskretes Artefakt (Korrektheit kann nicht abgeschätzt werden..)

2.2 Große Software Development Projekte

Zwei Hauptcharakteristiken

Kommunikation: (involvierte Parteien, verantwortliche im Software-Projekt)

- Kunden
- Endnutzer
- Software-Designer
- Software-Entwickler
- usw. ..

Komplexität:

- Derzeitige Software-Artefakte sind sehr groß und komplex
- Bsp: Betriebssysteme, Switches, Automotor Steuerung, medizinische Software..
- Konsequenz: SE muss Möglichkeiten bieten solche komplexe Systeme zu designen.

3 Konsequenzen

- Man versucht die Software Produktion auf einem „Engineering Ansatz“basieren zu lassen
- Software in einem Prozess entwerfen der folgendes unterstützt:
  - Korrektheit und Abhängigkeit des Software Produkts
  - kosten-effektiv
  - die Komplexität „schmiegsamer“machen
  - Langlebigkeit des Software Produktions Lebenszyklus, gerade bei sich ändernden Anforderungen
  - Kommunikation zwischen den Verantwortlichen
- Das führt zur Defintion eines Software Engineering Prozess-Models

3.1 Build and fix

- Keine Prozess Schritte (Keine Spezifikation, Dokumentation etc.)
- Keine Verteilung der Arbeit (Teamwork unmöglich, ..)
- Kann nicht mit großer Komplexität umgehen..

3.2 Wasserfall-Modell

- System Design
  - Problem- Systemrequirements
  - Durchführbarkeits-Studie
  - Hauptsysteme definieren
  - System Design Document
- Requirements
  - Functional, Non-Functional (Remember? =))
- Design
  - wie?, architektur, deatiliertes Design
- Implementation
  - Implementieren und testen
- Integration
  - Integrieren und testen
- Maintenance

3.3 Vorteile von Prozess-Modellen

- Klare Definition der unterschiedlichen Aufgaben
- Spezifikation und Dokumentation
- Ermöglicht Einführung weiterer Konzepte (Verifikation, Prototyping, ..)

3.4 Alternative Projekt-Modelle

Wasserfall ist leider nicht sehr praktisch da:

- man von einem Schritt nicht zurück kann
- es nicht flexibel ist (sondern sehr abhängig von dem was bisher gemacht wurde)
- es eine große Zeitspanne zwischen anfang und ende des Projekts gibt

Es gibt ein modifziertes Wasserfall-Modell, das es ermöglicht von einem Schritt auf den vorherigen zu springen.
Weitere Alternativen sind: Modifiziertes Wasserfall-Modell, V-Model (XT), Evolutionäre Prozess-Modelle, Spiral-Modell, Rational Unified Process, Agile Prozesse..

4 Weiteres

4.1 Software Qualitäten

- Korrektheit (Verhält sich wie gewünscht)
- Zuverlässigkeit (Garantierte Qualität, ist schwächer als Korrektheit)
- Robustheit
- Wartbarkeit
- Leistung
- Wiederverwendbarkeit
- Interoperabilität

4.2 Prinzipien des Software Engineering

- Systematisches Vorgehen
- Formalisierung
- Aufteilung der Arbeit
- Abstraktion
- Modularität
- Generalisierung

Teil II

Anforderungen und Analyse

5 Anforderungs-Spezifikation

Man sollte sich hier auf das Schnittstellen-Verhalten konzentrieren!

5.1 Typen

- Nutzer - Oft in natürlicher Sprache, evtl. Diagramme. Geschrieben für den Kunden
- System - Ein strukturiertes Dokument, detailierte Beschreibungen etc.

Weiter unterscheidet man zwischen funktionalen und nicht-funktionalen Anforderungen:

- funktional
  - Angabe welche Services das System zur Verfügung stellt
  - Interaktion zwischen System und Umgebung
  - Wie sich das System in unterschiedlichen Situationen verhält

- nicht-funktional
  - nicht zwangsweise mit System-Funktionen verbunden
  - bezieht sich auf bspw:
    * Zuverlässigkeit
    * Genauigkeit von Ergebnissen
    * Leistung / Timing
    * Mensch-Computer Schnittstellen Probleme
    * Laufzeit und physische Bedingungen
    * Portability und interoperability
    * Standards...

5.2 Domänen Anforderungen

Abgeleitet von der Applikations-Domäne. Beschreibt System-Charakteristiken und Features, die die Domäne wiedergeben. Beschränkung existierender Anforderungen.
Falls Domänen-Anforderungen nicht eingehalten werden, läuft das System evtl. nicht.

Probleme

- Verständlichkeit (Sind oft in der Sprache der Applikations-Domäne geschrieben, bsp. Physiker schreiben Formeln)
- Selbstverständlichkeit (Domänen-Anforderungen werden für selbstverständlich erachtet)

5.3 Eigenschaften der Anforderungs-Spezifikation

- Korrektheit - Alles was spezifiziert wurde, ist auch so gewollt
- Unmissverständlichkeit - Keine mehrfachen Interpretationen möglich
- Komplettheit - Jede Eigenschaft die verlangt wird, ist auch in der Spezifikation
- Nachweisbarkeit - Es gibt manuelle oder automatische Möglichkeiten, nachzuweisen dass das System den Spezifikationen nach kommt (viele Anforderungen sind nicht nachweisbar korrekt)
- Beständigkeit - Anforderungen widersprechen sich nicht gegenseitig
- Traced (Begriff) - Trifft zu wenn der Ursprung jeder Anforderung klar ist
- Zurückführbar - SRS ist so geschrieben, dass jede Anforderung leicht verweisbar ist
- Design-Unabhängigkeit - Es wird keine Plattform oder ein Algorithmus vorgegeben, das liegt in der Hand der Software Entwickler

5.4 Auszuführende Schritte während der Requirement-Analysis

- Starting Point
  - System Specification Document (HW und SW)
  - Customer requerements (abstract)
- Aktivitäten:
  - Aufstellen der Requirements (Interviews, Szenarien, Marktbeobachtung usw.

- Aufstellen und Verhandlung der Requirements (welche der eventuell gegensätzlichen Requirements sind wichtig)
- Requirement Dokumenation und Spezifikation (verständliches Requirements-Dokument, formale und nicht-formale Spezifikation)
- Requirement Validierung (Konsistenz, Vollständigkeit, Übereinstimmung mit den dokumentierten Requirements und abstrakte Customer / User Requirements

5.5 Requirements Elicitation (Findungsphase)

- Gespräche und Verhandlungen über Requirements mit verschiedenen beteiligten Personen
- Prototyping (der Kunde weiß nicht was er will, stellt sich aber grob etwas vor)
  Fester Bestandteil moderner Prozessmodelle
- Brainstorming
- FAST (Facilitated Application Specification Technique)
  Methode, in welcher sich eine Gruppe an Beteiligten trifft und nach bestimmten Grundregeln die Requirements zusammenstellt.
- JAD (Joint Application Design)
  Von IBM-Entwickelter Ansatz für FAST
  Hauptziele:
  - Gruppendynamik nutzen
  - Kommunikation verbessern
  - rationaler, organisierter Prozess (wiederholbar)
  - Standardisierte Dokumentation

  JAD soll bis zu 20% Gesamtkostenreduktion ermöglichen.

6 Object-Oriented Analysis

Was sind die wichtigen Objekte und was tun sie? (Häufig in UML)

Prinzipien:

- Objekte
- Klassen
- Kapselung
- Information-Hiding
- Vererbung
- Polymorphismus

Software-Modelle werden im gesamten Entwicklungszyklus benutzt, für:

- Validierung der Requirements
- Simulation des erwarteten Systemverhaltens
- Dokumentation der Software-Architektur

- Einschätzen der Performance und des Verhaltens des Designs
- Automatisches erzeugen von Stubs, Skeletons
- Definieren was getestet wird
- Existierendes System für spätere Fortentwicklung dokumentiern.

Bei der Form von Software Modellen legt man wert auf mathematische Formulierungen, grafische und textuelle Notationen ($\longrightarrow$ UML!).

6.1 OO Modeling und UML

Visuelle Notation für die Analyse und Design. Definiert von der **Object Management Group (OMG)**. Vorgänger hiervon sind Booch, Jacobson und Rumbaugh.

6.1.1 Models und Views

Model: Abstraktion die ein System oder eine Teilmenge davon beschreibt.
View: Schildert ausgewählte Aspekte eines Modells (Sozusagen eine Art ZoomIN).

6.1.2 Konzepte

Statische Struktur: Schlüssel Konzepte einer Applikation, interene Struktur und Beziehungen (Use Case, Static: Class Diagrams, Implementierung: Component Diagram).

Dynamisches Verhalten: wie interagiert das Objekt mit der Umgebung und Kommunikationsmuster einer Menge von Objekten (Statechart Diagram, Activity, Sequence, Collaboration).

6.1.3 Use Case

- Häufig genutzt als Einstieg
- Nutzungs Szenario
- Szenarios sind atomare Entitäten
- Für den Nutzer als Dokumentation

Bestandteile: Aktuere, Use Cases, Pfeile geben an wer was nutzt (communicate).
Include: gibt an wenn ein Use Case ein anderes nutzt.
Extend: gibt an wenn ein Use Case nur optional ein anderes nutzt.

Use Cases finden: Was sind die Funktionen die ein Aktuer durchführen soll? Systeminformationen für Akteur (Ändern, Hinzufügen, Löschen)?...

Zusätzlich schildert das Use Case Interaktionen und Abhängigkeiten zwischen Akteuren und Use Cases.

6.1.4 Class Daigram

Phenomenon: Objekt aus der realen Welt
Concept: beschreibt Eigenschaften von Phänomenen, ist ein 3er Tupel aus Name, Grund(Eigenschaften) und Mitglieder (Phänomene)

Weitere Begriffe:

- **Type:** Name: int, Grund: Ganzzahlen, Mitglieder: $-\infty, \infty \in \mathbb{N}$
- **Instanz:** Beziehung zwischen Type und Instanz ist ähnlich wie zwischen Phäonmen und Konzept.
- **Abstract Data Type:** Spezieller Typ dessen Implementierung versteckt wird.

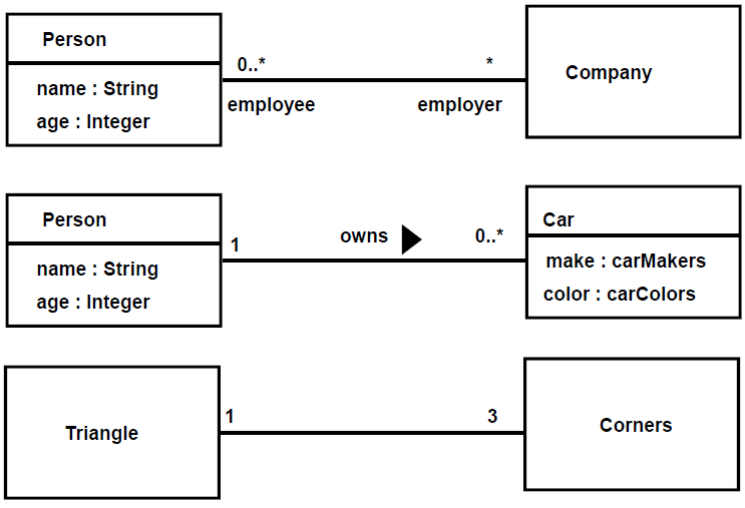
Klasse Klasse besteht aus Attributen und Methoden. Ein konkretes Objekt einer Klasse ist eine Instanz. Die Attribute haben dann konkrete Werte.
Hierbei repräsentiert eine Klasse ein **Konzept**. Sie kapselt Attribute und Operationen. Jedes Attribut hat einen Typ und jede Operation hat eine Signatur. Ein Objekt hingegen repräsentiert das Phänomen.

Ziele:

- Generalisierung
- Abstraktion

Nutzung während Requirements Analysis und Problem Domain s.o. Während System design und Object design.

Assoziation Stellt eine Beziehung zwischen zwei Klassen dar. Dabei gibt es Rollennamen und Kardinalitäten. Siehe Grafik:



Wichtig: Man kann auch bereits hier definieren wie die Variable heißen wird in der die Objekte einer anderen Klasse gespeichert werden. Dies geht durch eine qualifizierte Assoziaiton.

Bei n:n muss man Klassen für eine Assoziation erzeugen!.

Aggregation Ist-Teil-Von-Beziehung. Ansonsten möglichkeiten wie Assoziation.
Darstellung mittels weißer Raute.

Komposition Besteht-Aus-Beziehung. Hierbei sind die Teile Pflicht.
Darstellung mittels gefüllter Raute.

Generalisierung Vererbung. Vererbte Attribute müssen bei den Erbenden nicht dargestellt werden.
Wird durch weißen Pfeil dargestellt.

Sichtbarkeit Plus für public. Minus für private und # für protected.

Mehrfachvererbung Wird von UML zugelassen. Alles wird vererbt. Viele OO lassen dies jedoch nicht zu.

Abstrakte Klasse Abstract in den Klassenkopf in geschweiften Klammern. Es kann keine Instanz von dem Objekt erzeugt werden.

6.1.5 Sequenzdiagramm

Klassen werden durch Spalten repräsentiert.
Kommunikation mittels Pfeil.
Aktivierungen werden durch Rechtecke in der Lebenslinie dargestellt und die Lebenslinien durch gestrichelte Linien.

6.2 Use-Case getriebene Analyse

- Entitäten identifiziern (Persistente Information über System)
- Kontroll-Objekte (Realisierung des Use-Case)
- Rand-Objekte (Interaktion zwischen Akteur und System)
- Use-Cases auf Sequenz Diagramme Mappen
- Assoziationen, Aggregationen und Attribute herausfinden
- Zustands-abhängiges Verhalten individueller Objekte modellieren
- Vererbungs-Beziehungen modellieren

6.3 Objekte herausfinden

Applikations-Domäne analysieren ist hilfreich. Ebenso Allgmeinwissen und Intuition!

7 Analyse mit UML

7.1 Modelle definieren

Funktionales Modell: Use-Cases
Dynamisches Modell: Sequenz-Diagramm, Zustands-Diagramm
Objekt Modell Klassendiagramm

7.2 Text Analyse / Linguistische Analyse

Hierbei untersucht man einen Text auf bspw. Nomen für Objekte, Verben für Methoden etc. Hierbei muss beachtet werden, dass sowohl Attribute als auch Objekte Nomen sind, daher muss hier entschieden werden. Prinzipiell ist dies ein kreativer Prozess.

Hat man die Informationen aus dem Text gezogen, werden unnötige Assoziationen, Objekte etc. eliminiert. Daraufhin wird das Modell aufgebaut.

7.2.1 Verfeinern

Hat man das Modell aufgestellt, ist es nicht zwangsweise das best Mögliche. Deshalb sollte man dieses noch verfeinern.

- Fehlende Objektklassen: Asymmetrie und Generalisierungen
  ungleiche Attribute/Operationen innerhalb einer Klasse → Klasse splitten
  doppelte Assoziationen...
- unnötige Klassen
- Fehlende Assoziationen
- Unnötige Assoziationen (redundante bspw.)
- Inkorrekte Platzierung von Assoziationen und Attributen

8 Dokumentieren von Anforderungen

8.1 Zweckmäßigkeit von SRS

- Vertragsmäßige Sicherheit
- Genutzt bei Rechtsstreit
- Vereinbaren von Anforderungen und Prioritäten,
- Rückschau im Falle sich ändernder Anforderungen,
- Liste an Kriterien, die für die Abnahme der Software wichtig ist,
- Budget- und Zeitplanung
- Optional kann man Customer Acceptance Tests und User Doku machen
- ebenso das User Interface definieren und einen Prototyp entwickeln

9 Zusammenfassung von Anforderung und Analyse

1. **Erster Schritt:**
   - Anforderungs- und Analyseprozess organisieren
2. **Zweiter Schritt:**
   - Use-Cases herauskitzeln
   - Nicht-funktionale Anforderungen herauskitzeln

- Domänen-Anforderungen
- Priorisiere Use-Cases und Anforderungen
- Sorge für Rückführbarkeit!

3. **Dritter Schritt**
   - Ausarbeiten und verfeinern von Use-Cases
4. **Vierter Schritt**
   - Relevante Objekte identifizieren (Entität-, Rand- und Kontroll-Objekt)
   - Linguistische Analyse auf verfeinerten Use-Cases
   - Analyse der Anwendungs-Domäne
5. **Fünfter Schritt**
   - Dynamisches Verhalten des Systems modellieren
   - Sequenz-Diagramme, Zustands-Diagramme

   Sechster Schritt
   - Klassendiagramm
6. **Siebter Schritt**
   - Klassenmodell aufräumen
7. **Achter Schritt**
   - Analyse dokumentieren

Teil III

Design

10 Design Objectives

Software soll den Anforderungen entsprchen. Man muss die Komplexität meistern.

10.1 Desing Prinzipien

- Tunnelblick vermeiden
- Rückführbar zum Analyse Modell
- Man soll das Rad nicht neu erfinden
- Einheitlichkeit und Intefration darlegen
- Design möglichst günstig für Änderungen gestalten
- Design ist nicht gleich Coding
- Design sollte dafür sorgen dass Semantik minimiert wird??! (Nehmen an dass weniger über die Semantik nachgedacht werden muss, da bereits im Design arbeit abgenommen wurde?)

11 Klassisches und Objekt-Orientiertes Design

11.1 Design Ansätze

Hauptdesignansätze sind:

- Procedural Design
  - Dekomposition
  - Beschreibung von Modulinteraktionen
  - Prozedur-Definitionen
  - Keine Abstrakten Datentypen, Vererbung, jedoch Interfaces
  - C, Pascal..
- Object-oriented Design
  - Modellierung in Klassen und Objekten
  - Beschreibung Objektschnittstellen
  - Kapselung, Abstrakte Datentypen
  - Wiederverwendung durch Vererbung
  - Polymorphismus

11.2 Klassisches, prozedurales Design

- Architektur-Design: Struktur Elemente bestimmen und Beziehungen
- Schnittstellen-Design
- Daten-Design
- Prozedurales Design

11.2.1 Modul

Ist eine wohl-definierte Komponente, die eine diskrete Funktionalität bieten. Es besteht aus einem Namen, Interface und Körper.

Die Implementierung wird außerdem versteckt.

Warum? $\longrightarrow$ Komplexität, Doku, Teamwork

11.2.2 Coupling and Cohesion

Cohesion:
Unter Kohäsion versteht man beim Software Engineering ein Maß dafür, wie gut die Funktionalitäten eines Moduls zusammenhängen, d.h. ob sie gemeinsame Funktionalitäten bzw. nur eine Funktionalität haben.

Coupling:
Unter Kopplung versteht man beim Software Engineering ein Maß für die Abhängigkeit eines Moduls von anderen Modulen.

Beispiele für Cohesion:
Zufällige Kohäsion: Die Funktionalitäten innerhalb des Moduls/ der Klasse haben nichts oder nur sehr wenig gemeinsam.(*low cohesion*)
Prozedurale Kohäsion: Die Elemente des Moduls sind sich ähnlich, müssen jedoch in einer bestimmten Reihenfolge ausgeführt werden. (*medium cohesion*)
Funktionale Kohäsion: Wenn alle Modulelemente für eine bestimmte, wohldefinierte Aufgabe ausgeführt werden müssen, spricht man von funktionaler Kohäsion.(*high cohesion*)

Beispiele für Coupling
geringe Kopplung: Hier geringe Abhängigkeit von einem anderen Modul.(*low coupling*)
Kontrollkopplung: Ein Modul kann durch Parameter Einfluss auf den Kontrollfluss eines anderen Moduls nehmen.(*medium coupling*)
Inhaltskopplung: Ein Modul ist von der Abarbeitung von Daten durch ein anderes Modul abhängig und verlässt sich somit auf den Code in einem anderen Modul. Oder ein Modul ändert die Daten eines anderen Moduls.(*high coupling*)

Wünschenswert?
Die Kopplung sollte möglichst gering gehalten werden, wobei die Kohäsion maximiert werden soll. Deshalb ist high cohesion mit low coupling wünschenswert.

11.3 OO Design

OO Aktivitäten: Analyse, Design, Programming.

11.3.1 Generelle Schritte im OO Design Prozess

- Analysis-Model in Teilsystem aufteilen
- Teilsysteme den Aufgaben zuteilen
- Design User Interface
- Basisstrategie für Datenmanagement wählen
- Globale Ressourcen identifizieren
- Kontrollsystem entwickeln
- Randbedingungen

11.3.2 Vorteile OOD

Wartung vereinfachen, Wiederverwendbarkeit, Semantik

12 Design mit UML

12.1 OO Design Prozess

- Dekomposition
- Kontext und Modi des Systems
- System Architektur
- System Objekte identifizieren

- Design Modelle entwickeln
- Objektschittstellen spezifizieren

12.2 Welches Diagramm für was?

- Um ein System in Teilsysteme aufzuteilen, eignet sich das Paketdiagramm wie es auf Folie 3-36 beschrieben wird, da man auf kleine aber übersichtliche Weise die Beziehung zwischen den Teilsystemen darstellen kann. Außerdem kann so ein Package Klassen beinhalten und man kann auch komplexere Modelle erstellen.
- Um den Zusammenhang zwischen den Teilsystem zu beschreiben, eignet sich das umfangreichere Paketdiagramm aus Folie 3-37, in dem man auch in die Teilsysteme sehen kann und deren Klassen bzw. Teilsysteme sieht und wie diese mit anderen Klassen und Teilsystemen interagieren.
  Für Nutzungsmodelle eignen sich besonders UML Use Case Diagramme, da hier auf einfache weise die Interaktion mit dem Teilsystem dargestellt werden kann.
- Um die System Architektur darzustellen, eignet sich wieder das UML Package Diagramm. Hier sieht man Teilsysteme als einzelne Packages und Beziehungen zwischen diesen. Allerdings sollte dieses Package Diagramm, auch ein Package für das gesamte System beinhalten, sodass man schnell sieht welche Packages überhaupt miteinander kolaborieren. Ein Beispiel für dieses Diagramm ist auf Folie 3-43 zu finden.
- Es ist durchaus Sinnvoll die Objekte des Systems direkt in einem Klassendiagramm darzustellen, da ein Objekt einer Klasse entspricht.
- Um Design-Modelle zu entwickeln kann man auf eine Reihe an UML Diagrammen zugreifen. Darunter, Sequenz Diagramme, Zustands Diagramme, Aktivitäts Diagramme und Komponenten Diagramme. Alle diese Diagramme rechtfertigen ihren eigenen Zweck, wie z.B. der Ablauf von Operationen und Interaktionen zwischen Teilsystemen in einem Sequenzdiagramm.
- Für spezielle Objekt-Interface sind wieder Komponenten Diagramme geeignet, wie sie auf Folie 3-55 beschrieben wird, da man hier Interface Beziehungen im Diagramm darstellen kann.

13 Interfaces Contracts

Wozu?: Remote Object Invocation, Modularisierung, Contract

13.1 Ziele Middleware

Transparenz: Fakt verbergen, dass eine implementierte Nachricht an einem fernen Ort aufgerufen wird.

13.2 CORBA

13.2.1 CORBA Aufgaben

Lokalisieren, Aktivieren, Kommunizieren und Antworten.

Static invocation: Remote Object von CORBA ist bekannt
Dynamic invocation: Ist nicht bekannt

13.2.2 IDL

Interface Definition Language.
Enthält Namen von Attributen und Methoden.

13.3 Interface

Warum? → Ermöglicht besseres Teamwork, Dokumentation ist gleich vorhanden, Qualität wird gesichert, Konsistenz von Operationen in einer Klasse

13.4 Contract

Ein Contract hat eine Precondition. Wenn diese erfüllt ist, muss gewährleistet sein, dass auch die Postcondition erfüllt ist. Weiterhin gib es invarianten, die für die Korrektheit sorgen.

13.5 Object-constraint Language

Hiermit definiert man invarianten, pre- und postconditions in UML.

13.6 Subcontractor

Erbende Klasse muss min. Contract der Superklasse erfüllen.
Die Precondition darf hier nicht weiter eingeschränkt werden, die Postcondition kann verstärkt werden.

14 Software Architektur und Muster

14.1 Typische Architektur Styles

- Pipe unf Fitler
- Call und Return (Fan out, Fan in, Tiefe, Breite..)
- Ebenen hierarchischer Struktur
- Client/Server Kommunikation

14.2 Vertikale Partitionierung

Separate Teilbäume werden gewählt. Kontroll-Module koordinieren die Kommunikation zwischen den Funktionen.

- Vorteile:
  - einfaches Testen und Warten
  - einfaches Erweitern
- Nachteile:

  - größerer Datenfluß und komplizierte Kontrollflüße bei häufigem wechsel zwischen Funktionen

14.3 Horizontale Partitionierung (Factoring)

Man unterteilt das System in Schichten. Das erleichtert z.B. das Anpassen der Anwendung auf Änderungen, da in den unteren Ebenen öfter etwas geändert wird als in den oberen.

14.4 Schichten

Beispiel: Linux

- Vorteile:
  - Ermöglicht geringe Kopplung
  - Hohe Kohäsion
  - Zyklische Abhängigkeiten werden vermieden
- Nachteile:
  - Funktionalität zur Kommunikation zwischen den Ebenen ist Overhead.

14.5 Closed Architecture (wie VMs)

Prinzip der geschlossenen / sichtdichten Schichten (jede Schicht kann nur die nächst untere Layer aufrufen).

Vorteile:

- Seperation of Concerns
- Wartbarkeit (da unabhängiger)
- Flexibilität

14.6 Open Architecture (wie VMs)

Prinzip des transparenten Layerings (jede Schicht kann auf alle anderen Schichten darunter zugreifen).

Vorteile:

- Laufzeit / Effizienz (kein Message-Passing durch Weiterleitung, keine Parametrisierung)

14.7 Client-Server

...

14.8 Patterns (Design-Patterns)

Realisiert das Wiederverwenden von Umsetzungen für schon gelöste, häufig vorkommende Probleme durch bewährte Design-Muster.

Erzeuger:

- Builder
- Singleton
- Factory
- Prototype

Strukturelle:

- Adapter
- Bridge
- Decorator
- Facade
- Proxy

Verhalten:

- Interpreter
- Mediator
- Observer
- State
- Strategy

Für mehr Info: s. Jans ZFS.

15 Von Design zum Code

15.1 Mapping Contracts

Die meisten OO Sprachen unterstützen nicht direkt Contracts (außer Eiffel).

15.2 Implementing Contracts

- Precondition implementieren, also prüfen ob sie erfüllt ist.
- Postcondition testen, am ende des codes
- Jede Invariante prüfen!
- Vererbung handlen. Alle pre- und postconditions auch in Methoden die von der Klasse aufgerufen werden prüfen.

16 Design dokumentieren

Siehe SDD.

Teil IV

Testing und Quality Assurance

16.1 SQA

SQA (Software Quality Assurance), es wird in einem Projekt immer Fehler geben, daher versucht man mit SQA Fehler zu finden.

Techniken:

- Verifikations- und Validierungstechniken
  - Code Analyse
  - Testing
  - Formale Verifikation
- Code Quality Assessment
  - Software Metrics

Verifikation: Beweisen (formal, Korrektheit), dass das Produkt der Spez. entspricht
Validierung: Experimentell zeigen, dass das Programm den Anforderungen entspricht.

Ziele:

- Korrektheit zeigen
- Je früher, desto besser!

16.2 Was sind Reviews?

Reviews - Walkthrough (Durchgehen) und Inspection

17 Testing und Strategies

17.1 Test-Prinzipien

- White Box - mit Kenntnis des Source-Codes
- Black Box - ohne Kenntnis des Source-Codes
- Grey Box - wie Back Box, nur mit Kenntnis einiger interner State-Variablen.

17.2 Typen des Testings

- Unit-/Modul-Testing (durch Object Design Document)
  - Test der Spezifikation eines Moduls
  - häufig schon während des Programmierens
- Integrations-Test (durch System Design Document)
  - Testen beim Zusammensetzen der Module
  - Zusammenarbeit der Module testen
  - Testen ob Spezifikation erfüllt
  - wird von System-Integratoren und Testspezialisten ausgeführt
- System Testing (durch Requirements Analysis Document)
  - Gesamtumfang des Systems testen
- Akzeptanz Testing (durch Erwartung des Kunden)
  - Ausgeführt durch den Kunden
  - Manchmal in Requirements oder in Verträgen festgelegt
- Regression-Testing
  - Ausgeführt während Wartung, wenn eine Komponente verändert wurde
  - Eine fixe Menge an Tests, welche garantiert, dass alle Features auch nach der Änderung noch funktionieren.
- Stress Testing
  - System unter extremen Bedingungen testen, z.B. unüblich hohe Nutzanzahl

17.3 Exhaustive Testing

Gerade wenn Schleifen in einem Programm vorhanden sind, kann das Programm, selbst wenn es nicht sonderlich komplex ist, nicht vollständig getestet werden.

Generell kann man sagen: Man kann durch Testen niemals die Abwesenheit von Bugs zeigen kann, sondern lediglich die Anwesenheit, wenn man einen findet.

Ziel von Testing ist also das Finden von Bugs.

17.4 Formalisiertes Testing

- Test case D
  - einzelner Testfall (z.B. eine Abfrage)
- Test set / Test suite T
  - Ansammlung / Teilmenge von Testcases
- Succesful
  - Suite ist erfolgreich, wenn alle enthaltenen Test Cases erfolgreich waren
- Selection Criterion C
  - ist eine Menge aller endlichen Teilmengen aller Test Cases

- Consistent
  - Ein Criterion ist consistent, wenn für alle Testsuits in C gilt, dass alle immer das gleiche Ergebnis halten
- Complete
  - Ein Criterion ist complete, wenn die in C enthaltenen Testsuits alles Abdecken, also jeder Testcase getestet wird.

17.5 Unit Testing

Individuelle Einheiten des Systems werden getestet. (Methoden in einem Objekt o. Klasse, Komponenten bestehend aus internen Schnittstellen und multiplen Klassen).

Beim Unit Testing geht es nicht um das Testen des ganzen Systems.

Complete Coverage kann hier erreicht werden in dem,

- Alle Methoden mit allen möglichen Kombinationen von Parametern testen
- Alle möglichen Werte für Attribute setzen und testen
- Alle möglichen Zustände

Heuristiken

- Zeigen, dass die Komponente das tut was sie soll
- Zeigen, dass sie robust ist

18 Test Coverage

Coverage-Arten:

18.1 White Box Testing

18.1.1 Strukturelles Testing

(Unit) Testing bei bekanntem Code: Code ausführen um Fehler zu entdecken. Modul muss tun was es soll. Test cases werden vom Programm code abgeleitet.

Fehler können nur in Teilen des Programms, die getestet werden, gezeigt werden. Die Anzahl Test Cases muss realistisch sein.

18.1.2 Selective Testing

Wieviele Pfade (entsprechen TC) werden ausgewählt. Welche werden ausgewählt? (s. Coverage Criteria)

18.1.3 Statement-Coverage

- Jeder Knoten wird einmal erreicht

18.1.4 Edge-Coverage

- Jede Kante wird einmal besucht

18.1.5 Condition-Coverage

- Jede Condition eines Knotens wird jeweils einmal zu False und einmal zu True ausgewertet

18.1.6 Multiple Condition-Coverage

Hier wird tatsächlich jede Kombination durchgeführt.

18.1.7 Path-Coverage

Jeder Pfad muss ein mal besucht werden.

18.1.8 Loop Testing - Simple Loop

Loop soll einmal garnicht, einmal nur einmal und dann mehrfach ausgeführt werden.

18.2 Black-Box Testing

19 Reviews and Inspections

20 Software Metrics