Kapitel 10

Analyse

Anforderungsanalyse

Ziel
Grundlage
Prinzip der hierarchischen Strukturierung (Rene Descartes, 1637) : „Man soll jede schwierige Frage in so viele Teilfragen wie möglich zerlegen, damit man jede einzelne lösen kann."
Ziel
Problemzerlegung (**Analyse**); unterscheide:

- funktionale Anforderungen
- nichtfunktionale Anforderungen

WAS/WIE-Trennung ist entscheidend!

Aufgaben

1. lösungsorientierte Erhebung von Ist- und Sollzustand
2. Skizzierung der Systemarchitektur
3. Feststellung der Machbarkeit
4. grobe Abschätzung des Umfanges

Validierung und **Verifikation** der Anforderungen ist wichtig!

Eigenschaften einer guten Anforderungsanalyse

- Abgrenzen des Projektumfanges
- Achten auf Modularität
- Achten auf Änderbarkeit
- Festlegen der Schnittstellen
- Festlegen von Rahmenbedingungen („constraints")
- Prüfen der Realisierbarkeit

Schwierigkeiten

- erständnis für das Anwendungsgebiet
- zwischenmenschliche Kommunikation
- sich ständig ändernde Anforderungen
- Wiederverwendbarkeit der Ergebnisse („Eine neu erarbeitete Lösung rechnet sich erst nach drei Projekten!")

Techniken

Das Beherrschen der Komplexität

- Abstraktion (prozedurale Abstraktion,Datenabstraktion)
- Kapselung (Information Hiding)
- Vererbung (Aufbau von Taxonomien)
- Assoziation
- Nachrichtenkommunikation
- Organisationsmethoden
  - Objekt / Eigenschaften
  - Objekt / Bestandteile
  - Klassifikation von Objekten
- Verhaltenskategorien
  - unmittelbare Verursachung
  - ähnliche Evolution
  - ähnliche Funktionalität

Gedächtnisstützen

Zielerforschung: Was ist das Ziel?
Analyse: Was ist zu erstellen?
Design: Wie soll das Ergebnis aussehen?
Implementierung: Wie wird das Ergebnis realisiert

Werkzeuge

Papier CRC-Karten, Haftnotizen-Objekte, Scripting, (textuelle Beschreibung durch Szenarien)
Computer UML

Ergebnisse

Analysebericht
Lastenheft
Es wird also definiert, WAS WOFÜR zu lösen ist.
Das Lastenheft ist **vom Auftraggeber vollständig und widerspruchsfrei** zu erstellen. In der Praxis wird jedoch vielfach der Auftragnehmer mit der Erstellung beauftragt. Der **Auftraggeber** prüft das Lastenheft nur und gibt es frei.

Gliederungsvorschlag (gemäß VDI):

1. Einführung in das Projekt, Projektziele
2. Beschreibung des Istzustandes
3. Beschreibung des Sollzustandes
4. Schnittstellen
5. Systemanforderungen
6. Anforderungen für Inbetriebnahme und Einsatz
7. Qualitätsanforderungen
8. Anforderungen an die Projektentwicklung
9. Anhang

Vorprüfung durch den Auftragnehmer
Machbarkeit, Vollständigkeit, Minimalität, Konsistenz, Eindeutigkeit, Plausibilität, Überprüfbarkeit, Modifizierbarkeit

Machbarkeitsstudie
Teilmenge des Lastenhefts, wird bei unsicherer Machbarkeit vorab erstellt, konzentriert sich auf technische und wirtschaftliche Durchführbarkeit des Projekts oder definiert ein machbares Teilprojekt.

Planung

Ziel

Vorgaben festlegen für: Projektorganisation, Aufgaben, Aufwände, Termine, Ressourcen, Kosten, Finanzierung, begleitende Maßnahmen, vorsorgende Maßnahmen.

Planung bzw. Pläne müssen mit Projektfort- schritt t kontinuierlich verfeinert und aktualisiert werden.

Aufgaben

- Organisationsplanung
- Ziel- und Aufgabenplanung (Teilaufgaben)
- Zeitplanung
- Ablauf- und Terminplanung
- Ressourcenplanung (Sachmittel, Personal)
- Kosten- und Finanzplanung
- begleitende und vorsorgende Planung

Ziel- und Aufgabenplanung
Zielplanung: Festlegung der Zielwerte inkl. Ausmaß und Termin der geplanten Zielerreichung
Aufgabenplanung: Gliederung und Schätzung von Teilaufgaben, Festlegen der Aufgabenübergänge ⟶Aufgabenplan

Zeitplanung
Ermittlung des Zeitbedarfs für die Aufgaben (bzgl. Schwierigkeit, Sachmittel, Personal) mittels **Aufwandsschätzungen** auch **Aufwandsplanung** genannt; basiert auf **Aufgabenplan**!

Ablauf- und Terminplanung
auch **Meilensteinplanung** genannt; basiert auf Aufgaben- und Zeitplan!
Ablaufplanung: Erstellen eines Ablaufgraphen durch Verknüpfung der Aufgaben
Terminplanung: Festlegung von Anfangs- und Endterminen für jede Aufgabe

Ressourcenplanung
auch **Kapazitätsplanung** genannt.

Kosten- und Finanzierungsplanung
Kostenplanung Ermittlung und terminliche Zuordnung von Kosten (aus verplanten Ressourcen)
Finanzierungsplanung Budgetierung, Steuerung der Finanzmittelbereitstellung (aus Kostenplan)

Techniken

Aufwandsschätzung
Verfahren sehr n umstritten bzgl. Genauigkeit und Sinnhaftigkeit (Beispiele: Software Lifecycle Management (SLIM), Consolidated Cost Model v. 2 (COCOMO II), Function Points (FP), Object Points (OP), Proxy-based Engineering (ProBE)**.** Aufwandsschätzung nach Faustregeln („g Planning Poker", „ Magic Estimation" etc.) ist zumindest gleich gut!

Netzplantechnik
Eigenschaften

- Einteilung in Vorgänge und Ereignisse (Meilensteine)
- Zuordnung von Zeit und Ressourcen
- Erstellung eines Vorgänger-/Nachfolgergraphen; spezielle Abhängigkeiten: Ende-Anfang (EA), AA, EE, [AE]

Bekannte Beschreibungsmethoden: **Metra-Potenzial-Methode** (MPM) (nach Fa. Metra, heute Atos), **Critical Path Method** (CPM), **Program Evaluation and Review Technique** (PERT)

Balkenplantechnik
tabellarische, grafische Darstellung von Aufwänden durch:

- Balkendiagramme
- Histogramme
- Gantt-Charts (nach H. L. Gantt)

Eigenschaften: gut für Ressourcenplanung, schlecht für komplexe Zusammenhänge

Werkzeuge

Projektplanungssysteme (auch: **Projektmanagementsysteme**) gut zur Dokumentation und zur Planadaptierung.

Allgemeine Vorgehensweise :

1. neues Projekt anlegen (Name, Beginn, Kalender, ...)
2. Vorgänge eingeben und gruppieren (Aufgabenplanung)
3. Dauer zuordnen (Zeitplanung)
4. Vorgänge verknüpfen (Ablaufplanung)
5. Meilensteine festlegen (Terminplanung)
6. Ressourcen zuordnen (Ressourcenplanung)

7. Ressourcen- und Terminkonflikte auflösen (Kapazitätsabgleich)
8. Kosten eingeben (Kostenplanung)
9. Kostenüberschreitungen und Finanzierungslücken beseitigen (Finanzierungsplanung)

Ergebnisse

interne Ergebnisse: Aufgabenplan, Terminplan, Ressourcenplan, Kosten- und Finanzplan

externe Ergebnisse: Meilensteinliste

Risiko

Ziel

- Vorab **Überlegen** der **Reaktionen auf potenzielle** Probleme inkl. Treffen dervorbereitenden Maßnahmen
- **Abwägen von Risiken und Chancen**; Eingehen der Risiken mit bestem Chancenpotenzial
- **quantitative Bewertung der Risiken;** zur Auswahl der für die Behandlung sinnvollsten

Risikobegriff / Definition

Risiko. „Auswirkungen von Unsicherheiten auf die Ziele eines Systems"

Risiko = Möglichkeit eines zukünftigen Verlusts oder Schadens (ein potenzielles Problem)

Problem = eingetretenes Risiko

vorhersehbares Problem = unausweichliche negative zukünftige Konsequenz

Chance = Möglichkeit, Erwartungen (an das System) zu übertreffen

Risikoprofil = Summe der Einzelrisiken eines Projektes

Ein erfolgreiches Projekt behandelt – die richtigen! – Risiken erfolgreich.

Risikoeigenschaften

- Wahrscheinlichkeit („Erwartungswert")
- Auswirkung („Schadenskoeffizient")
- Zeitrahmen
- Behandlungsmöglichkeit
- Kopplung

$$Risikofaktor = Erwartungswert * Schadenskoeffizient$$

Risikobehandlung

Risikobehandlung = proaktive Berücksichtigung wahrscheinlicher Probleme

"Softwareentwicklung ohne Überraschungen" ⟶ reife Projektorganisation

Situierung der Risikobehandlung in der Projektentwicklung

- Risikobehandlung ist proaktiv.
- Auch viele Techniken der Projektentwicklung reduzieren das Projektrisiko, aber reaktiv.

Unterscheide

- allgemeine Risiken
- projektspezifische Risiken

Risikobehandlung nur für projektspezifische Risiken

Aufgaben

1. **Risiko-Bewertung**
   (a) **Risiko-Identifikation**
      i. **Quellen**: Dokumente wie bspw. Analysebericht, Lastenheft, Projektplan,"lessons learned" aus ähnlichen Projekten
      ii. **Vorgehen**: **Bottom Up:** Risikonennung durch (alle) Mitarbeiter; führt zu langen Listen, hoher Konsolidierungsaufwand! **Top-Down:** Risikoermittlung aus der Bilanz und den Unternehmenskennzahlen; dabei wird leicht Wichtiges übersehen!
   (b) **Risiko-Analyse**
      i. Ermittlung der Schadenswahrscheinlichkeit und des potenziellen Schadensausmaßes (+ Kopplungen)
      ii. durch mehrere Mitglieder in einem Treffen
      iii. am besten quantitativ
   (c) **Risiko-Evaluierung**
      i. Reihung der Risikoelemente entsprechend ihres
      ii. Projekteinflusses (Risikofaktor!, Hausverstand anwenden)
      iii. Entscheidend ist das Vertrauen in die Zahlen!
      iv. Ergebnis: „watch list" (Top-n-Liste)
2. Risiko-Beherrschung
   (a) **Risiko-Vorsorgeplanung**
      i. Festlegung eines Aktivitätenplans
      ii. für jedes behandelnswerte Risiko
   (b) **Risiko-Überwachung**
      i. Überprüfen der Risiken und Vorsorgepläne auf notwendige Aktivitäten und Planadaptierungen (beginnend spätestens ab Analysephase)
      ii. periodisch und bei Anlass
      iii. Oft wird nicht rechtzeitig eingestanden, dass ein Risiko zu einem Problem wird oder keines mehr ist!
   (c) **Risiko-Überwindung**
      i. Anwenden der im Vorsorgeplan festgelegten Technik der Risikobehandlung
      ii. Auftraggeber mit einbeziehen

Design

Ziel

Design legt fest, **WIE** die Anforderungen für die Implementierung aufbereitet werden (Grundlage für Codierung).
Codieren soll möglichst klar strukturiert und in einfachen Schritten möglich sein!

Aufgaben

Erstellung des **Designmodells**, das die **physische Sicht** (Aufbau) und **logische Sicht** (Verhalten) des zu entwickelnden Systems beschreibt
Das Designmodell umfasst:

- Systemmodell (Systemarchitektur aus Entwicklersicht)
- Komponentenmodell (Komponentenmodell und deren Schnittstellen)#

Dementsprechende Aufgaben:

- Erstellen des Systemdesigns
- Erstellen des Komponentendesigns

Vorgehen nach schrittweiser Verfeinerung (Wirth)

Gliederung

Systemdesign	Komponentendesign
Architektur Schnittstellen Datenhaltung	Komp.struktur Komp.verhalten

Prioritäten festlegen, Kompromisse schließen!

Systemdesign - Modellierung der Architektur

auch **Architekturdesign**; vielfach durch evolutionäre Prototypen

Erweiterung des Strukturmodells

- Entwickeln der Systemarchitektur aus dem Strukturmodell der Analyse (Analysebeschränkungen und Implementierungsvorgaben beachten)
- wieder zu verwendende Komponenten/Untersysteme einplanen

Erweiterung des Verhaltensmodells

- Zerlegung in Untersysteme inkl. deren zeitlichem Zusammenspiel (logische Systemgliederung in physische Untersysteme umsetzen ⟶ verteilte Systeme)
- danach Abbildung auf Prozesse (Echtzeitverhalten beachten!)

Modellierung der Schnittstellen

siehe Folien K11 S 11

Modellierung der Datenhaltung

Organisation des Ressourcenmanagements v. a. saubere Modellierung der Datenhaltung und -ablage wichtig:

- **Datenstrukturen** im Hauptspeicher
- **Dateien** im Dateisystem / Netzwerk / Cloud
- **Datenbanken** inkl. Anbindung (zentral / verteilt)
- **Export und Import** von Daten

Komponentendesign - Modellierung der Struktur

Festlegung der **Komponentenstruktur**:

- zuerst Abhängigkeiten und Schnittstellen zwischen den Komponenten festlegen und Kontrollfluss spezifizieren
- wieder zu verwendende Komponenten auswählen

- Komponentendefinition aus Analyse vervollständigen (wenn notwendig Strukturen ergänzen)
- iterativ Strukturmodell adaptieren (abstrakte Komponenten einführen)

Komponentendesign - Modellierung des Verhalten

Festlegung des **Komponentenverhaltens**:

- Beschreibung der Abläufe (Methoden)
- normalerweise im Design nur deklarativ, z. B.:
  - uses: Eingabeparameter
  - changes: Ausgabe-/Übergabeparameter
  - informs: erzeugte Nachrichten
  - invariant: Invarianten
  - requires: Vorbedingungen
  - ensures: Nachbedingungen
  - produces: Rückgabewerte
- nur bei schwierigen Algorithmen oder Performanzproblemen (Speicher, Zeit) funktionale Beschreibung
- zusätzlich Schnittstellenbeschreibung (in UML Schnittstellenklassen - „Interfaces“ - und Lollipop-Notation)

Techniken

Erstellen von Prototypen

vor allem **experimentelles Prototyping und evolutionäres Prototyping** für

- Architekturdesign und
- Benutzerschnittstellendesign

Prototyping ist Grundlage aller iterativ- inkrementellen Vorgehensmethoden!

Modellgesteuertes Entwickeln

„Model-Driven Development“ (MDD): reales System wird durch reine Transformationen aus einem Modell erzeugt

Basis: Model-Driven Architecture (MDA); diese ist Teil des UML2-Standards.

Bedenke: UML2 ist eine vollständige Programmiersprache!

Vorteile

- plattformunabhängige Entwicklung (Trennung von Modell und Implementierungsdetails in untersch. Plattformen nutzbar)
- Simulation auf Modellbasis möglich
- Plattform-Deployment automatisierbar

Vorgehensweise

1. Erfasse Anforderungen, Modelle und Prozesse in einem plattformunabhängigen Modell (Platform-Independent Model – PIM).
2. Lege Rahmenbedingungen als Metadaten fest (Meta Object Facility – MOF).
3. Erzeuge daraus ein plattformspezifisches Modell (Platform-Specific Model – PSM).
4. Generiere ausführbaren Code vollautomatisch.

Endbenutzer-Entwicklung / End User Development

vor allem durch Maßschneidern und Befüllen von Systemrahmen (Application Frameworks)
ührt vielfach zu schlechten Ergebnissen (Unerfahrenheit der Benutzer bzgl. gutem Designs, speziell bei Architektur, aber auch bei Richtlinien zur Benutzerinteraktion)

Wiederverwendung

Development by Investment - **Wiederverwendbarkeit** (Reuse) muss gegeben sein!

Vorgehen

- Wiederverwenden von Bausteinen
- Abwandeln von Mustern
- Verwenden von Schablonen
- Konkretisieren von Systemrahmen

Wiederverwenden von Bausteinen (Toolkits - Modulbibliotheken - Klassenbibliotheken) oder Abwandeln von Mustern (Designmuster - Analysemuster - Prozessmuster)

Werkzeuge

UML - sonst K11 S 33

Ergebnisse

Designbericht

- Erweiterung des Analyseberichts um die Designmodelle und um die Prototypen
- normalerweise intern, Auszüge für Auftraggeber zur besserenVerdeutlichung möglich.

Pflichtenheft

Pflichtenheft (bei IEEE: „System/Software Design Specification“ – SDS)

- ist „die Beschreibung der Realisierung aller Anforderungen des Lastenhefts“ (VDI) und „beinhaltet“ es somit.
- Es wird also definiert,**WIE** und **WOMIT** die Anforderungen zu realsieren sind

Das Pflichtenheft wird vom **Auftraggeber abgenommen**; - bei agilem Vorgehen ggf. Alternativen überlegen (Abnahme von Zwischenprodukten, Protokollen etc.).

Benutzerdokumentation

Implementierung

Codieren

Ziel

1. Umsetzen der Designergebnisse in Programmcode (Quellcode) →Codieren des Designs soll möglichst klar strukturiert und in einfachen Schritten möglich sein!
2. Übersetzen (lassen) des Quellcodes ingeeigneten Code für das Zielsystem

Aufgaben

Schrittweises, korrektes Umsetzen des Designs:

- **Umsetzen** des **Systemdesigns**
- **Umsetzen** des **Komponentendesigns**
- (komponentenweises) **Übersetzen** des Quellcodes in Zielcode
- **Einzeltest** jeder Komponente
- **Zusammenbau** zum Gesamtsystem

Techniken

Transformieren des Designs

klassische Kernaufgabe der Programmierung

Einfügen logischer Bedingungen

Assertionen sind logische Aussagen, die Bedingungen im Programm entsprechen

- Vorbedingungen
- Nachbedingungen
- Invarianten

Instrumentieren des Codes

Versehen des Codes mit zusätzlichen Codeteilen zum

- Feststellen der Codelokalität zur Laufzeit
- Prüfen der Pfadabdeckung
- dynamischen Analysieren (Profiling)
- Prüfen der Testqualität (Bebuggen, Mutieren)

Hauptprobleme:

- neue Codeteile können zusätzliche Fehler enthalten
- neue Codeteile verändern manchmal das Programmverhalten

Code-Restrukturierung („Refactoring“)

Code-Restrukturierung = „disziplinierte Änderung eines existierenden Codes zur Designverbesserung ohne Verhaltensänderung“ [Fowler]

- stammt aus Smalltalk (~1980); ab ~2000 eng mit eXtreme Programming (und später anderen agilen Vorgehensmethoden) verknüpft
- Erkennen restrukturierungsbedürftigen Codes durch Muster (bad smells)
- **positiv**: einleuchtende Muster mit detaillierten Vorschlägen zur Restrukturierung (teilweise sogar automatisierbar, vgl. z.B. Eclipse)
- **negativ**: Kommentare werden auch als Zeichen eines schlechten Codes gewertet! (eher missverstandener Zweck der Kommentierung)

Restrukturierung darf Lauffähigkeit nicht verändern!

- nur lauffähige Programme restrukturieren
- automatisierte Tests verwenden (Test-driven Development)

Weiteres Problem: Bei unerfahrenen Programmierern steigt Restrukturierungsaufwand überproportional! [Boehm]

Restrukturierung kann Architekturprobleme nicht lösen!

⟶ sinnvollste Anwendung zur Verbesserung des Komponentendesigns

Erstellen von Testrahmen
Zweck: Erstellen temporärer Programm(teil)e und Daten zu Testzwecken
- Programmierumfang kann umfangreich sein (bis zu gleich viel wie der zu testende Code)
- Code von Testrahmen nicht löschen, sondern nur ausblenden

Komponenten von Testrahmen:
- Stumpf („**stub**“): Testcode für gerufene Routine
- Treiber („**driver**“): Testcode für rufende Routine
- Attrappe („mock-up“, „**mock** object“): Testcode für Komponenten („mock-up“ wird oft für nichtausführbare Benutzerschnittstellen-Prototypen verwendet)
- Dummydatei („**dummy file**“): Datei mit Testdaten

Kontinuierliche Integration
engl. „continuous integration“: Jeder Entwickler pflegt kleine Codeänderungen laufend ein (früher „daily build“, jetzt „continuous build“).

Voraussetzungen
- eine gemeinsame Codebasis
- Übersetzen und Binden auf Knopfdruck (automatisiert)
- funktionale Tests hoch automatisiert
- neueste Programmversion laufend zugänglich

In der Praxis nur Werkzeug-gestützt sinnvoll möglich (z.B. Apache Ant, CruiseControl, Jenkins)!

Testgesteuerte Entwicklung
dee (engl. Test-driven Development – TDD): Testfälle werden vor dem entsprechenden Programm(teil) erstellt (eigentlich programmiert)!

Zweistufiges Vorgehen:
- Erstellen von Systemtests für Anwendungsfälle (idealerweise durch AG)
- Erstellen von daraus abgeleiteten Komponententests (durch AN)

In der iterativ-inkrementellen Entwicklung oft **Ersatz für genaue Produktspezifikation**:
- Produkt erfüllt immer genau die spezifizierten Testfälle.
- Jeder neue Testfall führt zu einer (möglichst kleinen) Produkterweiterung (ggf. Code-Restrukturierung).

Testgesteuerte Entwicklung beeinflusst Art des Designs und der Implementierung und ist daher keine Technik des Testens!

Vorteile
- Tests sind **automatisch ausführbar** und daher leicht wiederholbar.
- Tests werden vor der Realisierung erstellt ⟶ Erzeugung **testbaren Codes.**
- Tests dienen als **Spezifikation und Dokumentation** („besser als nichts“).
- Die Implementierung wird in jedem Inkrement getestet (**einfaches Regressionstesten**).
- Code-Änderungen sind (auch lokal) rasch überprüfbar.
- Fehler sind rasch lokalisierbar; Folgefehler rasch erkennbar.
- Fehler bei der Code-Restrukturierung werden normalerweise rasch erkannt.
- Man verliert **weniger Zeit mit Debuggen.**
- Zeit für die Testfallentwicklung wird zu Beginn investiert. Damit fällt eine Belastung zu Iterationsende weg.

Herausforderungen
- **Testfallerstellung** und -auswahl werden **nicht unterstützt**.
- Testfallqualität ist schwer beurteilbar.
- **Testfälle hängen** teils voneinander und auch von (geänderten) Anforderungen ab.
- Notwendige Änderungen an Testfällen sind schwer erkennbar.
- Eine große Anzahl von Einzeltests **ersetzen nicht** Integrations- und **Systemtests**.
- Techniken der systematischen Testfallerstellung werden oft vernachlässigt.
- „Einige“ korrekt verlaufene Testfälle „sichern“ die Programm-Korrektheit!

Weitere Erkenntnisse
- optimistisch
  - estklassen sind gleichzeitig Testimplementierung und -dokumentation der zu implementierenden Klassen.
  - Testklassen können als Ersatz für eine nicht vorhandene Spezifikation dienen.
  - Tester und Entwickler arbeiten enger zusammen.
  - Es gibt keinen Code ohne Test.
  - Testklassen liefern Sicherheit bei der Code-Restrukturierung.
  - Man entwickelt nicht zu viel und nicht zu wenig Code auf einmal.
- pessimistisch
  - Die Implementierung wird eher nach hinten verschoben.
  - Man versucht, durch viele, einfache Tests die unangenehmen, komplizierten Tests zu ersetzen.
  - Man glaubt, dass ein Testfall, der einmal passt, auch immer passt.
  - Eine große Anzahl von Einzeltests ersetzt nicht Integrations- und Systemtests.
  - Testgesteuerte Entwicklung verleitet zu Einstellungen wie
  - „Warum denn weitertesten, wenn es eh schon läuft?“.
  - **Größtes Problem: Wer testet die Tests?**

Testfallqualität ⟶ Testqualität ⟶ Softwarequalität!

Werkzeuge
Entwicklungsumgebungen, auch Programmierumgebungen genannt
für Codieren benötigt
- Editor
- Compiler / Linker
- Versions- und Konfigurationsverwaltungssystem

Ergebnisse
Quellcode
- Gestaltung muss bestimmten Regeln unterworfen werden
- Kommentierung muss Design widerspiegeln

Tipp: Möglichst früh entscheiden, ob Quellcode dem Auftraggeber übergeben wird!

Systemdokumentation
Systemdokumentation („Software/System Reference Manual“)

Zweck:
- Erklärung der Implementierung
- Dokumentation des Know-hows
- Grundlage für Wartung
- Unterstützung der Kommunikation im Entwicklerteam (zusätzlich Entwicklerkollegen als Kontrollleser)

Komponenten:
- Beschreibung der Systemstruktur
- Beschreibung des dynamischen Verhaltens
- Beschreibung der Komponenten
- Beschreibung der verwendeten externen Datenhaltung
- Beschreibung der zentralen Begriffe
- Beschreibung der Installation (Detaillierung des Installationshandbuchs)

Die Systemdokumentation darf **nicht nur Ergebnisse**, sondern muss **auch die Gründe** dafür (**WARUM**) dokumentieren.

Debuggen

Ziel
Finden der Ursachen **von Fehlern** und Mängeln **inkl. deren Behebung,** meist vom Codeersteller selbst durchgeführt

Aufgaben
Sechs Schritte für effizientes Debuggen:
1. Fehler stabilisieren
2. Fehler lokalisieren
3. Korrektur ausarbeiten
4. Korrektur durchführen u. dokumentieren
5. Korrektur testen
6. auf ähnliche Fehler prüfen

Unterschiede beim (Blackbox-)Testen:
- andere Person testet
- Fehler nicht (genau) lokalisiert
- Fehler nicht behoben

Fehler als Chance zur Leistungssteigerung!

Techniken
- Statische Programmanalyse (ohne Programmausführung)
  - Desk Checking
  - Technisches Review
  - Komplexitätsanalyse
  - Strukturanalyse
  - Datenflussanalyse
- Dynamische Programmanalyse (mit Programmausführung)
  - Postmortem-Analyse
  - Singlestepping
  - Topdown-/Bottomup-Test
  - Whitebox-Test

Statische Programmanalyse
Desk Checking **(„Schreibtischtest“)** manuelles Durchgehen „am Schreibtisch“, oft anhand konkreter Beispiele (durch Implementierer oder andere Person)
Aber: **Fachverständnis** (Domänenwissen) ist **wichtig**!

Technisches Review Präsentation von Code in einer Gruppe („First do reviews, then compile!“)
- Walkthrough: Implementierer präsentiert Programm Schritt für Schritt (Alternative: anderes Gruppenmitglied präsentiert)
- Codeinspektion: Gruppe prüft Code gegen (verschiedene) Checklisten
- Codereview: Gruppe prüft Code vorab und diskutiert ihn mit Implementierer

Zeitlimit ist wichtig!

Komplexitätsanalyse Bewertung mittels Komplexitätszahlen (Lines of Code (LOC), Software Sciences (Halstead), Cyclomatic Complexity (McCabe))

Viele Maße beruhen auf keinem realistischen Modell. Es wird eher das gemessen, was leicht zu messen ist! Aber: Eine schlechte Messung ist (meist) besser als gar keine!

Strukturanalyse Finden von Strukturanomalien (z.B. unerreichbarem, „totem“ Code, Endlosschleifen)

Datenflussanalyse Finden von Datenflussanomalien (z.B. verwendeten Variablen ohne Wert, Variable ohne Nutzung)

Statische Programmanalyse
Postmortem-Analyse („Leichenbeschau“) Auswerten der Ausgaben eines instrumentierten Codes (auch nach Absturz)

Werkzeuge
Statische Analysatoren
erstellen z. B. Objektbäume, Aufrufgraphen etc. zur Komplexitätsanalyse

Listengeneratoren
erstellen z. B. Listen von Komponenten, Aufrufen, Kreuzreferenzen, Programmstrukturpläne

Speicherprüfer
ermitteln verdächtige Speicherzugriffe und -verwendung

Testrahmensysteme
Bibliotheken zur Testautomatisierung, die das Schreiben von Testtreibern vereinfachen und vereinheitlichen

Debugger
erlaubt Ablaufverfolgung während des Programmablaufs (meist spezielle Übersetzung notwendig)
Funktionalität
- Postmortem-Analyse
- Quellcode-Anzeige
- Zugriff auf Laufzeitwerte
- Darstellung von Aufrufketten
- Singlestepping, Setzen von Break Points und Watch Points
- Analyse dynamischer Objektstrukturen
- interaktives Ändern von Werten

Ergebnisse

Lauffähiges Programm übersetzter Quellcode, der zum Testen (durch andere) freigegeben wird

Singlestepping schrittweises Durchgehen eines Programms, z.B. mit Debugger

Topdown-/Bottomup-Test Verwendung (von Teilen) des Testrahmens

Whitebox-Test Programmlauf mit verfügbarem Quellcode (vielfach mittels Debugger)

Fehlerliste
strukturierte Sammlung begangener oder möglicher Fehler es gibt Standardlisten, jeder Softwareentwickler soll jedoch seine individuelle Fehlerliste anfertigen (gereiht nach Häufigkeit – Pareto-Prinzip)
Anwendung:
- zur strukturierten Fehlersuche
- zur Generierung von Lösungsideen
- zur Erstellung projektspezifischer Checklisten
- zur Schwerpunktsetzung bei knapper Testzeit

Testen

Testen = systematisches Aufzeigen von Fehlern in minimaler Zeit und mit minimalem Aufwand unter realen Bedingungen

Man kann nur die Anwesenheit von Fehlern zeigen; Fehlerfreiheit lässt sich niemals zeigen! (Dijkstra)
- Jedes Programm enthält Fehler („mentale Irrtumsrate“, Halstead).
- Forderung nach „100% Fehlerfreiheit“ demotiviert Programmierer!

Rollenverteilung

- Der Implementierer soll nicht testen!
- Der Tester soll nicht korrigieren!

Aufgaben

Erkennen von Analyse- und Designfehlern
- Testen „von außen“ bedeutet intensives Arbeiten mitder Benutzerschnittstelle.
- Inkonsistenzen und fehlerhafte Bedienungsabläufe sind oft Analysefehler oder Designfehler!

Erkennen von Implementierungsfehlern
- strukturiert suchen (z.B. Versuch alle Fehlermeldungen zu generieren, Testen von Kompatibilität)

Bestimmen des Produktstatus
- Alphatesten, Betatesten

Produktstatus
Alphatesten
Testen der einzelnen Funktionalitäten bottomup Produkt als Gesamtes noch eher instabil (Alphaversion), Testen einzelner funktionaler Anforderungen ist jedoch bereits möglich

Betatesten
Testen des Produkts als Gesamtes anhand realer Beispielsabläufe (Betaversion; nicht unbedingt volle Funktionalität)

Produktperformanz
Performanztest – teilweise bereits während des Debuggens, während des Testens aber aus Anwendersicht
- Ermitteln notwendiger Performanzverbesserung als Vorgabe für notwendiges Tunen (Datenprüfung bei Eingabe in Maske vs. Prüfung vor Übernahme in DB)
- Speichertests testen die Speicheranforderungen
- Volumentests testen maximal bewältigbaren Aufgabenumfang
- Belastungstests testen Stabilität bei hoher Systemaktivität
- Benchmarkvergleiche vergleichen Produktversionen

Vorbereiten der Abnahme
Abnahmevortest nimmt die Abnahmesituation beim Auftraggeber vorweg (Test der **Abnahmesuite**)
Abnahmesuite wird **idealerweise vom Auftraggeber erstellt;** in der Praxis oft gemeinsam mit dem Auftragnehmer

Techniken

Testfalldesign
Ein guter Testfall
- macht Programmfehler offensichtlich,
- findet wahrscheinlich einen Fehler,
- ist weder zu einfach noch zu komplex.

Testprozess soll möglichst systematisch sein!

Gängigste Techniken
- Äquivalenzklassen bilden
- Grenzwerte analysieren
- Ursache-Wirkungs-Zusammenhänge analysieren

Testsuitedesign
Erstellen eines Verzeichnisses geeigneter Testfälle (“**Testfallsammlung**“) Man kann nie alle Anwendungsmöglichkeiten eines Systems testen (zeitliche und ressourcentechnische Einschränkungen).

Gliederung in
- **Konformanztests** (müssen korrektes Ergebnis liefern),
- **Nonkonformanztests** (müssen Fehler liefern) und
- **Qualitätstests** (nichtfunktionale Anforderungen).

Erstellen von Testfällen
Vorgehensweisen
- aus Pflichtenheft und Dokumentation extrahieren und Grenzwerte für die Eingabewerte festlegen
- mit Referenzprogramm korrekte Testergebnisse erstellen
- interaktiv mit System arbeiten
- Konkurrenzprodukte zu Vergleichszwecken heranziehen
- korrekte Ergebnisse für spätere Vergleiche aufheben
- alle Ergebnisse immer elektronisch aufheben

Testablaufplanung
Qualifikationstest
prüfen, ob eine Programmversion stabil genug fürs Testen ist ⟶ Erstellen einer Qualifikationstestsuite, Testen im Qualifikationstest

Testzyklus
Schritte:
1. „Start with a bang!“ – Implementierer rasch mit Fehlerbehebungsaufträgen „versorgen“
2. Produktstabilität und -zuverlässigkeit abschätzen (Prognose für notwendige Testzyklen)
3. Offensichtliche Fehler aufdecken (Simulation des ersten Arbeitstages beim Anwender, Inkonsistenzen zur Dokumentation)
4. Testsuite(n) durchlaufen
5. Ergebnisse dokumentieren (möglichst elektronisch und geeignet für automatische Wiederholung/Prüfung)
6. Testsuite(n) adaptieren

Verifikation
esten der Funktionalität „Haben wir das Produkt richtig erstellt?“
- Einzeltest („Unit Test“)
- Modultest, Programmtest
- Integrationstest (Testen des Zusammenbaus)

Validierung
esten des Gesamtsystems („Haben wir t das richtige Produkt erstellt?“) ⟶ Überprüfung, ob ein Softwareprodukt den realen Anforderungen genügt („Brauchbarkeit“; Boehm) daher auch „Systemtest “, “Gesamttest“

Techniken
- (Testen weitgehend unabhängiger Subsysteme und deren Schnittstellen)
- Systemtest, Integritätstest (Testen auf Einsetzbarkeit und Brauchbarkeit beim Anwender)
- (Benutzer-)Akzeptanztest (Testen der realen Verwendung)
- Zertifizierung (wenn gewünscht / erforderlich)

Regressionstesten
auch: Retesten

Zweck:
- Überprüfung der Fehlerkorrektur
- Absicherung gegen neu eingeschleppte Fehler

Ergebnisse

Testplan

Testplan = „Beschreibung von Aufgabenumfang, Vorgehensweise, Ressourcen und Ablauf der beabsichtigten Testaktivitäten“ [ANSI/IEEE]

Erstellung benötigt viel Zeit:

- parallel zu Design und Implementierung durchführen
- auf Pflichtenheft aufbauen
- laufend aktualisieren

Inhalte

- Beschreibung von Produktziel und Testziel
- Festlegung des Testumfangs (auch was nicht getestet wird!)
- Auflistung der abgedeckten / nicht abgedeckten Risiken
- Beschreibung von Teststrategie, -ansatz und -kriterien
- Beschreibung der Testumgebung
- Planung des Ablaufs inkl. benötigter Ressourcen
- Festlegung der zu liefernden Testdokumentation

Testsuite

Testsuite = „nach Testklassen geordnetes Verzeichnis von Testfällen“ [ANSI/IEEE]

Testlisten

angefertigt für persönlichen oder entwicklerinternen Gebrauch, erstellt aus dem Pflichtenheft sowie aus Benutzer- / Systemdokumentation

Fehlerbericht

Fehlerbericht = standardisierte Beschreibung eines entdeckten Fehlers mit Angaben zur Fehlerbehebung

Fehler muss reproduzierbar sein!

Fehlerbericht immer sofort beim Auftreten eines Fehlers schreiben (Fehlerverhalten „noch auf dem Bildschirm sichtbar“)!

Fehlerlogbuch

Fehlerlogbuch = zeitlich geordnete Auflistung aller gefundenen Fehler inklusive ihrer Behebung (auch: Testlogbuch)

Produkteinführung

Ziel

Produkteinführung = Abschluss eines Projekts durch die erfolgreiche Aufnahme des Betriebs eines Softwareprodukts beim Auftraggeber/Kunden

für Auftragnehmer: Know-how-Erweiterung durch Nachanalyse

In Betrieb befindliche Software benötigt Wartung und Pflege!

Aufgaben

1. Lieferung
2. Installation und Inbetriebnahme
3. Schulung
4. Abnahme und Abschluss
5. Wartung (und Pflege)
6. Evaluierung
7. Archivierung

Lieferung

umfasst **Übergabe** und **Übernahme** von Produkt inklusive Dokumentation

Der AG erwartet bei der Lieferung ein **brauchbares Produkt.** ⟶Erster Eindruck beeinflusst (positive) Einstellung

Oft muss Zuverlässigkeit abgeschätzt werden (vom Projektleiter):

1. hohe Zuverlässigkeit
2. mittlere Zuverlässigkeit
3. niedrige Zuverlässigkeit
4. unbekannte Zuverlässigkeit (am schlechtesten!)

Zumindest **minimale Zuverlässigkeit** schriftlich festlegen Basis für die Abnahme.

Projektleiter muss Unternehmensleitung über Risiken klar informieren!

Installation und Inbetriebnahme

Installation = Einrichtung eines Produkts in der (simulierten bzw. echten) Zielumgebung (noch ohne es auszuführen)

Aufgaben

- **Überprüfung** der Zielhardware (Vollständigkeit, Funktionsfähigkeit)
- **Konfiguration** von Hardware und Software
- kundenspezifische **Installation** (Treiber, Konfigurationen)
- Prüfung auf Funktionsfähigkeit des Geräts (**ohne** neue **Software**)

Installation und Inbetriebnahme

Inbetriebnahme = Vorbereitung für den Regelbetrieb (erste Ausführung des Produkts)

Aufgaben

- **Starten** und **Testen** der Komponenten
- Erstellen von **Testausdrucken**
- Prüfung auf Funktionsfähigkeit des Geräts (**mit** neuer Software)

Vor und nach der Installation ein **Backup** anfertigen! Auch **Wiederinbetriebnahmetest** durchführen.

Schulung

Schulung = Einweisung in geeignete Verwendung

Bediener haben verschiedene Rollen (vgl. s Personas aus dem Design!):

- **Benutzer** (user) braucht Anwendungsfunktionalität
- **Betreuer** (administrator) braucht Systemfunktionalität, aber auch als Benutzer schulen!

Schulung möglichst **vor der Abnahme** durchführen! ⟶ Sie kann verrechnet werden (Angebot!) und erleichtert die Abnahme.

Abnahme und Abschluss

Abnahme = rechtskräftige Annahme eines Produkts durch den AG; – führt automatisch zum Abschluss!

Abnahme dauert max. 1 Tag – immer ein Kompromiss zwischen optimalem und akzeptablem Produkt (schriftliche Dokumentation – Protokoll! – wichtig)

In der Praxis **erst** sinnvoll **nach Inbetriebnahme** (und Schulung) beim Auftraggeber.

Wartung

Wartung (+ Pflege) = Produktverbesserung nach dem Abschluss

Jede Software erfordert Wartung (jedoch anderer Wartungsbegriff als bei materiellen Produkten, da keine Abnützung).

Wartung im weiteren Sinn =

- Wartung im engeren Sinn (korrigierende Tätigkeiten) +
- Pflege (erweiternde Tätigkeiten)

Aktivitäten der Wartung (im engeren Sinn):

- Stabilisierung (Beseitigung enthaltener und neu eingebrachter Fehler, ereignisgesteuert)
- Optimierung (wenn zur bestimmungsgemäßen Verwendung nötig)

Aktivitäten der Pflege:

- Anpassung (bei Bedarf; enthält auch „Optimierung“ über die Anforderungen hinaus)
- Erweiterung (längerfristig planbar)

Pflegeaktivitäten umfassen 60 - 80% der Gesamttätigkeit!

Kosten für Wartung und Pflege:

Wartung/Pflege macht **mehr als 50% der Gesamtkosten** eines Produkts (über die gesamte Entwicklungs- und Lebenszeit) aus.

- Seriöse Kostenschätzung nur bedingt möglich
- In der Praxis werden vielfach 7-15% der Entwicklungskosten pro Jahr kalkuliert
- Wartungsfreundlichkeit ist wichtig

Wartungs- und Pflegeaktivitäten sollten voneinander getrennt werden (unterschiedliche Charakteristika) – in der Praxis jedoch schwierig:

- Wartung nach Aufwand (Stunden) verrechnen
- Pflege als Folgeprojekt kalkulieren (Fixpreis)

Wartungsdauer („End-of-Life“-Problematik):

Bei „alten“ Produkten ist zu entscheiden:

- Wartung („Instandhaltung“)
- Sanierung („Instandsetzung“, „Reengineering“)
- Evolution („Weiterentwicklung“, „Pflege im Großen“)
- Migration („Übertragung“)
- Einstellung (durch neues Produkt ersetzen)

Evaluierung

Evaluierung = auftragnehmerinterne Nachanalyse und Nachkalkulation (Soll-/Ist-Vergleich des Projektaufwands)

- daraufhin Adaptierung der Planungsparameter

Projekt wird kostenmäßig abgerechnet, die Zeitplanung normalisiert.

Archivierung

wichtig für Wartung und Wiederauffinden von Know-how

- intern elektronische und Papierablagen „zusammenräumen“
- gesamtes Know-how dokumentieren (Wissen aus den Köpfen der MitarbeiterInnen)
- auftraggeberspezifische Informationen löschen

Techniken

Umstellung

Übertragung von Datenbeständen in die neue Umgebung

Strategien

- **Direkte Umstellung**: Wechsel vom alten aufs neue System zu einem bestimmten Zeitpunkt (vor allem bei nicht duplizierbaren Ressourcen)
- **Parallele Umstellung**: gleichzeitiger Betrieb des alten und des neuen Systems über einen gewissen Zeitraums, dabei laufende Synchronisierung des Datenbestands (bei sicherheitskritischen Systemen)
- **Testweise Umstellung**: gleichzeitiger Betrieb des alten und des neuen Systems über einen gewissen Zeitraums mit ein-/mehrmaliger Duplizierung der Daten, ohne regelmäßige Synchronisation

Probebetrieb
Evaluierung des Produkts durch den Auftraggeber

Arten:

- Installationstest
- Leistungstest (Benchmarktest)
- Pilottest

Auftraggeber darf kein fehlerfreies, sondern lediglich ein brauchbares System erwarten!

Kontinuierliche Freigabe
Englisch: Continuous Release, Continuous Delivery. Bei vielen neuen Softwaresystemen (z.B. Web, Mobile Apps):

Eigenschaften:

- aktueller Inhalt („content“) untrennbar mit Anwendung verknüpft
- Übergang von Entwicklung zur Wartung kaum terminisierbar
- Produkt wird regelmäßig aktualisiert („kontinuierlich freigegeben“)

Hilfreich:

- inkrementelles Prozessmodell (agiles Vorgehen)
- kurze Freigabezyklen (wenige Tage)

Herausforderungen bei Kontinuierlicher Freigabe:

- oft muss Anwendung durchgehend verfügbar sein („24x7 operation“)
- Wartung und Pflege im laufenden Betrieb schwierig

Empfehlungen:

- Konfigurations-/Versionsverwaltungssysteme verwenden (Änderungshistorie!)
- Entscheidungen schriftlich festhalten

Development & Operations („ DevOps)
Verschränkung von Entwicklung („Development“) und Betrieb („ Operations“)

- Anforderungen („Requirements“) und Änderungswünsche („Change Requests“) werden zunehmend gleich betrachtet und behandelt!
- **Übertragung der Betriebsfähigkeit und -qualität auf den Auftragnehmer**
- Ausweitung des Agilen Vorgehens auch auf den Betrieb („Agile System Administration”, „Agile Operations”; Application Lifecycle Management)

Abnahmetest
Gesamttest unter realen Einsatzbedingungen - prüft (**ideal**) Produkt gegen **Lastenheft**, **real** Produkt gegen aktuelle **Kundenwünsche**

- **Abnahme(test)** ist kein punktuelles Ereignis – genügend Zeit einplanen; Auftraggeber jedenfalls einbeziehen
- **Abnahmesuite**: Menge vorher festgelegter Testfälle – muss vom Auftraggeber dem Auftragnehmer rechtzeitig (VOR der Abnahme!) übermittelt werden
- **Abnahme-Schlussbesprechung**: Abnahme erfolgt, wenn entdeckte Fehler tolerierbar oder nachbesserbar sind. Vollständiges Testen ist niemals möglich!

Nachkalkulation
Soll-/Ist-Vergleich der Planwerte, um die Ursachen der Abweichung zu ermitteln. Verglichen werden: Anforderungen (Aufgaben), Aufwände, Termine, Ressourcen, Kosten, Finanzen

Werkzeuge

Spezielle Werkzeuge existieren nur beschränkt (z. B. **Erweiterungen** von **Continuous-Integration-Tools** wie Jenkins oder AntHill bzw. DevOps-Tools (z.B. Docker).

Eingesetzt werden beispielsweise:
Versionsverwaltungswerkzeuge, Installationsprogramme, Simulatoren, Monitorprogramme, Schnittstellenprüfer, Logfile-Ersteller, Protokolliersysteme

Ergebnisse

Betriebsversion
erste abgenommene Version inklusive Dokumentation (**keine „Vollversion“ “!**)

Wichtig

- korrekte Version der Dokumentation liefern
- Systemdokumentation (notwendig bei Lieferung des Quellcodes)
- Testdokumentation beilegen
- Readme-Dateien nur im Notfall beilegen
- Benutzer über Fehler informieren

Installations- - und Inbetriebnahmeprotokoll
dokumentiert chronologisch Installation und Inbetriebnahme (durch Auftragnehmer)

Auftraggeber führt Bedienerprotokoll

- Weiterführen in den ersten Betriebstagen
- Dient als Basis für Fehlerkorrekturen bzw. Änderung in Schulungsunterlagen bzw. notwendigen Nachschulungen

Abnahmeprotokoll
Dokumentiert detailliert die **Abnahme**. Wird **unabhängig vom Erfolg der Abnahme** erstellt (Teilabnahmen sind möglich)
Ist ein gemeinsames Protokoll nicht möglich (zu unterschiedliche Ansichten), so werden **Sachverhaltsdarstellungen** erstellt.

Abschlussbericht
interner Bericht des Auftragnehmers, von allen Mitarbeitern erstellt (manchmal auch als Protokoll einer Abschlussbesprechung), schließt Informationen der **Nachkalkulation** und der **Qualitätssicherung** ein

Projektarchiv
dient als Basis für Wartung und Dokumentation des erarbeiteten Know-hows

Wichtig:

- alle notwendigen Werkzeuge in den richtigen Versionen mit archivieren
- rechtliche Klärung der Speicherung auftraggeberspezifischer Daten ist notwendig

Inhaltsverzeichnis