Software Engineering

Terminologie und Herausforderung

- Software „unsichtbar“ daher meinst gedanke, alles ist machbar
- Umso Komplexer desto mehr Konflikte zwischen Steakholdern
- Billig Programmieren und schnell aber in jeder Umgebung einsetzbar (kompatiblität)

Lösung Vorgehensmodelle:

Softwareentwicklung

Planung:

- Anforderungsanalyse
- Systementwurf

Realisierung:

- Programmierung
- Testen
- Einführung/Wartung

Plannungsphase:
Es wird Versucht, die Anforderungen an die Software zu eruieren und zu dokumentieren. Die Sammlung der Anforderungen ist verbindlich und dient als Grundlage für die Abnahme.

Realisierungsphase:
Im Einklang mit den dokumentierten Anforderungen wird angefangen zu Programmieren. Unabhängig vom Anwendungsgebiet und den Anforderungen werden zunehmend Programmierrichtlinien verfolgt, um, Wartbarkeit, Wiederverwendbarkeit und Erweiterbarkeit zu gewährleisten.

Sequenzielle Phasenmodelle (Wasserfall):
Phase für Phase…

- Do it twice (Prototyp entwickeln bspw)
- Document the Design (Viel und gute Dokumentation)
- Plan, Control, and Monitor Testing
  Quellcode statisch Testen anschließend dynamisch. Testgruppen sollten Leute sein die beim Design und der Programmierung nicht beigetragen haben, und es sollten Qualitätsisicherungstechniken und -werkzeuge angewendet werden
- Involve the Customer (Endbenutzer sollte nach der Anfoderungsbestimmung auch in den Phasen des Systementwurfs und Testens involviert sein

Wachstumsmodell (Spiralmodbel)

Möglichst schnell ein funktionsfähiges Teilsystem zu entwickeln, das grob formulierte, kritische Erstanforderungen erfüllt. In jeder Iteration nähert sich der Prototyp an das gewünschte Endprodukt an.

- Jede Phase kann mehrfach durchlaufen werden

Agiles Vorgehensmodell n

- Flexibel auf änderungen reagieren
- Einbeziehung von Kuden/Stakeholdern

Agiles Manifest = 12 Prinzipien

Anfoderungsermittlung

Funktionale vs nicht funktionale Anforderungen

Funktionale Anforderungen:

- Sind Anfoderungen, die eine konkrete Funktionalität beschriebe, die von der Software erfüllt werden muss
- Bspw. App muss die möglichkeit haben sachen zu ergänzen, entfernen

Nicht funktionale Anforderungen:

- Anfoderungen die nicht direkt eine Funktion der Software beschreibt. Sie beschreibt vielmehr externe und interne Qualitätsanforderungen. (Effizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit, Portabilität)
- Bspw. System soll auf allen gängigen Webbrowsern laufen

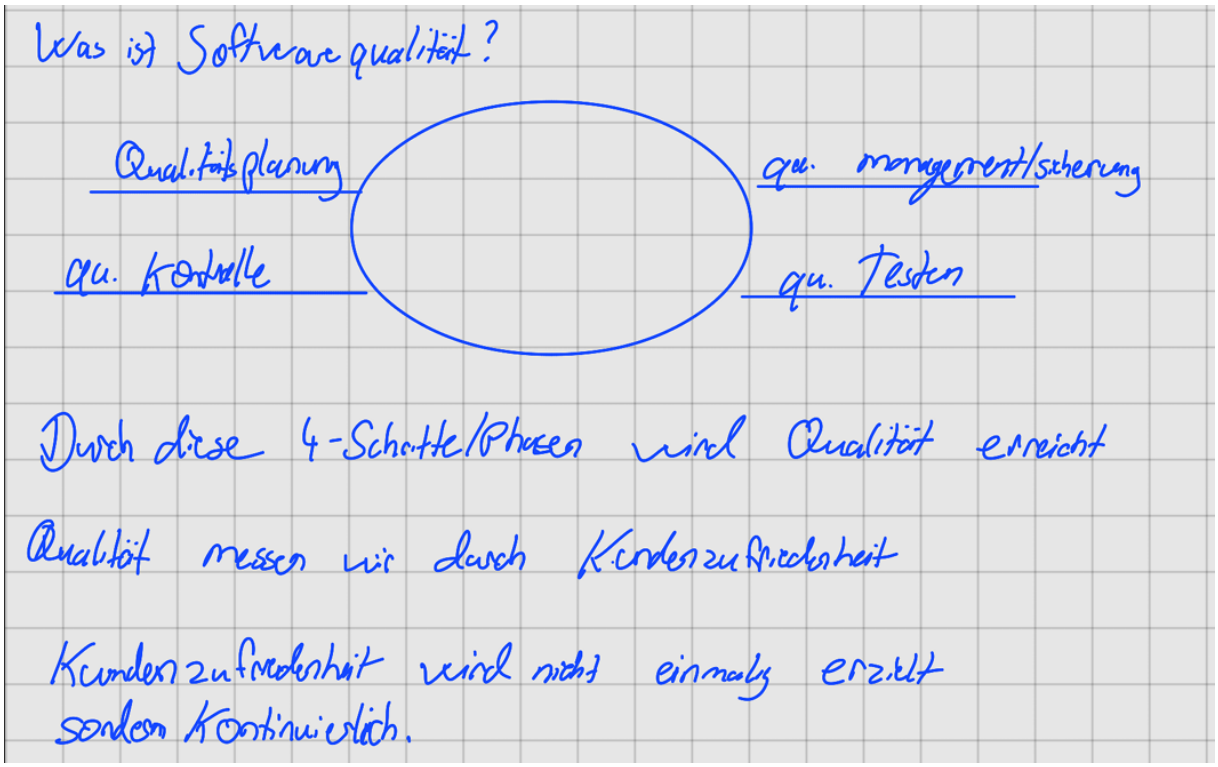
Rahmenbedingungen

- Keine Anforderungen, sondern Fakten bzw- Umstände, die den möglichen Lösungsraum einschränken (Hardwareumstände, existierende Geschäftsprozesse, Gesetze, Standards, Budgetrahmen

Prozessmodell	Primäres Ziel	Antreibendes Moment	Benutzerbeteiligung
Sequenzielles Modell	minimaler Managementaufwand	Dokumente	gering
Wachstumsmodell	Risikominimierung	Prototyp	mittel bis hoch
V-Modell	maximale Qualität	Dokumente	gering
Rational Unified Process (RUP)	Risikominimierung	Dokumente, Prototyp	mittel
Agile Modelle	minimale Entwicklungszeit, Risikominimierung	Kunde, Prototyp	hoch

Größe	Vorgehensmodell	Erfolgreich	Eventuell abgeschlossen	Gescheitert
alle Projektgrößen	Agile	39 %	52 %	9 %
	Wasserfall	11 %	60 %	29 %
große Projekte	Agile	18 %	59 %	23 %
	Wasserfall	3 %	55 %	42 %
mittelgroße Projekte	Agile	27 %	62 %	11 %
	Wasserfall	7 %	68 %	25 %
kleine Projekte	Agile	58 %	38 %	4 %
	Wasserfall	44 %	45 %	11 %

Qualitätssicherung von Software



Verschiedene Ansätze für den Arbeitsablauf im Qualitätsmanagement

- Kontinuierlicher Ansatz
- Modellbasierte Ansätze
- Agiles Vorgehensmodelle

Qualitätsmerkmale und -metriken für Software

Allgmemeine Nützlichkeit

- Portabilität
- Gebrauchsnutzen

    - Verlässlichkeit
    - Effizienz
- Wartbarkeit
    - Testbarkeit
    - Verständlichkeit
    - Veränderbarkeit

Software Produktqualität

- Funktionalität
    - Vollständigkeit
    - Korrektheit
    - Angemessenheit
- Kompatibilität
    - Koexistenz
    - Interoperabilität
- Benutzbarkeit
    - Erlernbarkeit
    - Bedienbarkeit
    - Barrierefreiheit
- Zuverlässigkeit
    - Reife
    - Verfügbarkeit
    - Fehlertoleranz
- Wartbarkeit
    - Modularität
    - Wiederverwendbarkeit
    - Prüfbarkeit
- Sicherheit
    - Vertraulichkeit
    - Integrität
- Leistungseffizienz
    - Zeitverhalten
    - Ressourcennutzung
    - Kapazität
- Portabilität

    - Anpassbarkeit
    - Austauschbarkeit
    - Installierbarkeit

Methoden zur Qualitätssicherung

Konstruktive Qualitätssicherung

- Software-Richtlinien
- Typisierung
- Portabilität
- Dokumentation

Analytische Qualitätssicherung

- Software-Tests
    - Balck-Box-test
    - White-Box-Test
- Statische Analyse
    - Software-Metriken
    - Konformitätsprüfung
    - Exploit-Analyse

Code-Richtlinien wie

- Namenskonventionen: Variablen- und Funktionsnamen, sollten aussagekräftige Namen haben
- Formatierungskonventionen: if-Abfragen sollten möglichst einzeilig sein, und Anweisungen in Blöcken eingerückt werden
- Kommentarkonventionen: Kommentare sind wichtig, aber sollten nicht unnötig verfasst werden, bei klarem Methodennamen sind Kommentare nicht notwendig

Statische Softwarequalitätskontrole

Warnsignale im Code

- Sehr lange funktionen
- Zu viele Funktionsparameter (regel mehr als 4 sind zu vermeiden)
- Zu tief verschachtelte Verzeigungen (Indikatoren von Logikfehlern oder komplexer Denkweise)
- Globale Variablen (funktionen sollten soweit wie möglich mit lokalen Variablen arbeiten, die als Parameter übergeben werden)
- Code-Duplikat (Gleicher Code an mehreren Stellen, dann lieber funktion erstellen)
- Sinnlose Kommentare

Statische Analysewerkzeuge auf Quellcode um schwachstellen anzuzeigen bzw. zu identifizieren.

Dynamische Qualitätskontrolle

Testgrundsätze

- Testen zeigt die Existenz von Fehlern, nicht deren Abwesenheit

- Vollständiges Testen ist nicht möglich
- Frühzeitiges Testen erspart Zeit und Geld
- Fehler häufen sich im Allgemeinen nah beieinander an

Arten des Testens

- Funktionale Tests
  Sicherstellung das Software den funktionalen Anforderungen entspricht
  Wichtig Tester muss Anforderungen verstehen
- Regressionstests
  Stellt sicher das Veränderungen keine negative Auswirkungen auf die Software haben
- Nicht-Funktionale tests
  Test bei denen nicht-funktionale Anforderungen getestet werden
    - Usability Tests sollen die Benutzerfreundlichkeit einer Software überprüfen
    - Performanz-Tests sollen Zeit- und Ressourceneffizienz einer Software sicherstellen (Reaktionsfähigkeit, Verlässlichkeit, Stabilität und Skalierbarkeit)
    - Sicherheitstests sollen die informationstheoretischen Sicherheitsanforderungen an die Software überprüfen
- Einhaltungstests (compliance testing)

Stufen des Testens

- Komponententests (Komponenten sind die kleinsten testbaren Teile einer Software, funktionen)
- Komponentenintegrationstests werden implementiert und ausgeführt, wenn die Zusammenarbeit von mehreren Komponenten getestet werden soll.
- Systemintegrationstests: Hier wird die Zusammenarbeit mehrerer Systeme getestet
- Systemtests (Alle Komponenten als Ganze werden von Anfang bis ende getestet)
- Akzeptanztest

Automatisiertes Testen und testgetriebene Entwicklung

Komponententests mit JUnit

Software-Architekturen

Softwareentwurf

Technischer Plan für die Entwicklung, um die Anforderungsanalyse technisch umzusetzen.

Im Softwareentwurf wird festgestellt welche:

- Programmiersprachen, Entwicklungsumgebungen, externe Funktionsbibliotheken, API und SDKs

Architekturmuster

Architekturmuster definieren die grundlegende Organisation und Interaktion zwischen den Komponenten einer Anwendung.

Adaptive Systeme, Verteilte Systeme

Adaptive Systeme

Unterstützen besonders die Erweiterungs- und Anpassungsfähigkeiten von Softwaresystemen.

Plug-In-Architektur

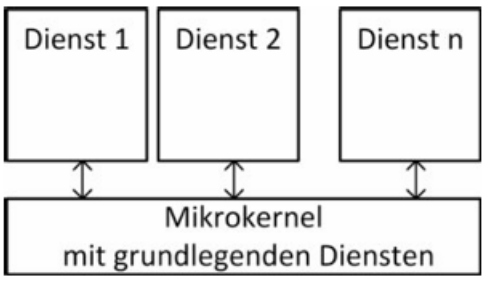
Dienste in einer Software, erweiterung durch Plug-Ins (Eclipse zb.)

Vorteil:
- Erweitbarkeit, Vielfältig, Leichte Wartbarkeit, Gute Kappselung der Plug-Ins, Flexibiltät
Nachteil:
- Komplexität (Plug-in Struktur), Schad-Code kann leicht ins System gelanden (zb. Open Source Plug-Ins), Kompatibilität (Plug-In abhängig vom anderen Plug-In zb. Spezifische Versionen nötig)

Kern liegt auf dynamische Erweiterung

Mikrokernel

Hauptsächlich in Betriebssystemen aber auch für komplexe Software denkbar. Kern enthält nur absolut grundlegende Funktionen, bei OS (Speicher- und Prozessverwaltung und Grundfunktionen zur Synchronisation und Kommunikation). Alle weiteren Funktionen werden als eigene Prozesse eingebunden, die mit den nachfragenden Prozessen kommunizieren.



Vorteil	Nachteil
-Modularität und Entkoppelung der Dienste voneinander	-Komplexe Abhängigkeiten zwischen den Diensten - Verstärkte Kommunikation innerhalb Kernal

⇨ Diese Kombination sorgt für geringere Performance von Systemen auf der Basis von Mikrokernels

Reflexion

Programm kennt seine eigene Struktur und kann diese zur Laufzeit modifizieren. Dadurch bei Laufzeit abfragen auf Klassen, aus denen Objekte instanziiert werden

Reflexion ist eine Voraussetzung für die sogenannte dynamische Typsicherheit, dabei werde Datentypen zur Ausführungszeit überprüft und müssen reflexiv abgefragt werden können. (type(object))

Dependency Injection

Ein spezielles Entwurfsmuster der OOP, bei welchem die Abhängigkeiten zwischen Objekten regimentiert werden. Beispiel zwischen 2 Klassen, Verzeichnis und Datei, diese haben eine Komposition, das heißt die Klasse Datei kann ohne das Ganze, Verzeichnis, nicht existieren.

Class Verzeichnis public Verzeichnis(name)
Class Datei public Datei (name, verzeichnis v)

Schichtenarchitektur

Bspw. OSI Schichten Modell (7 Stück)

Schicht kann nur mit direkter schicht daneben kommunizieren.

TCP/IP Modell

Anwendung: (http; FTP, DHCP)
Transport: (UDP; TCP)
Internet: (IP)
Netzzugang: (Ethernet, WLAN)

3-Tier-Architektur

Anwender → Präsentation → Businesslogik → Datenzugriff → Datenbank

Interaktive Systeme

Helfen dabei, Interaktionen zwischen einem Menschen und einem Computer zu sturkturieren. Diese Muster kommen heutzutage vorallem in webbasierten Anwendungen vor.

Model-View-Controller (MVC)

Basis zur Entwicklung von komplexen Softwaresystemen etabliert, insbesondere im Webumfeld.

View: Darstellung von Daten und Realisierung der Benutzerinteraktion. (Client side)

Die View wird jedoch vom Server über einen Controller verwaltet, der durch http Requests angestoßen wird, die Daten aus dem Model holt und für die View aufbereitet und als http Response an die View zurückgeht

MVP (Model View Presenter)
Klarere Trennung der Schichten mit größerer Unanbhängigkeit der Schichten untereinander und verbesserter Testbarkeit der einzelnen Schichten.

Verteilte Systeme

Zusammenschluss unabhängiger Computer, die sich für den benutzer als ein einziges System präsentieren.

Echte Nebenläufigkeit realisieren, da mehrere Prozesse gleichzeitig ausgeführt werden. Besser Skalierbar als einzelne Computer, da durch Hinzufügen weiterer Rechner die Leistungsfähigkeit erhöht wird.

Client-Server-Model

Server ist ein Programm, das einen Dienst anbietet.

Beliebig viele Clients können nun diesen Dienst nutzen, indem Anfragen (Requests) an den Server gehen, dieser Antwortet drauf (Response)

Peer-to-Peer

Bspw. bei Filesharing, Einheiten kommunizieren direkt oder indirekt miteinander. Peer kann Server und Client funktionalitäten übernehmen.

Entwurfsmuster:

- Factory
- Proxy

Architekturmuster:

- MVP
- Peer-To-Peer

Antimuster:

- Onion
- Blob

Antimuster der Software-Architektur:

- Spaghetticode
- Blob (Objekt das zu viel Weiß)
- Software die keine erkennbare Software-Architektur besitzt, wird als großer Matschklumpen bezeichnet