Software Konstruktion - Falk Hower
Zusammenfassung

Maximilian Springenberg
TU-Dortmund WS18/19

0 Vorwort

Diese Zusammenfassung basiert auf den Vorlesungsfolien des Moduls Software Konstruktion, gehalten von Falk Hower im Wintersemester 2018/2019 an der TU-Dortmund. Die Unterlage dient als Lernergänzung, ich bin nicht Eigentümer des Gedankengutes.

1 Teil1

1.1 Projektmanagement

Entwicklungsphasen:

Weil unterteilen den Weg von der Idee/ dem Problem hin zum Betrieb in Phasen.

- Analyse
- Design
- Umsetzung
- Test

Dazu werden wir uns Methodik, Metriken und Werkzeuge anschauen. Das Querschnittstheme dieser Sektion ist das Projektmanagement, später werden wir noch Methodik im Vorgehen genauer betrachten.

Anforderungen an Methoden:

Die Methode sollte korrekt (Qualität), im Budget (Kosten) und termingerecht (Termin) umgesetzt werden können.

1.2 Projektplanung

1.2.1 Wasserfallmodell

Beim Wasserfallmodell werden Phasen sukzessiv abgearbeitet, ohne Rückkanten.

1. Anforderung
2. Entwurf
3. Implementierung
4. Überprüfung
5. Wartung

1.2.2 V-Modell

Das V-Modell beschreibt ein linear Steigende und dann linear fallendes Verhältnis von Zeit und umfang/ Detaillierung vom Entwurf. Zeitlich mittig und mit der größten Detaillierung steht der software-Entwurf. Anforderungen, System-Architektur, System-Entwurf, sowie Software Architektur gehen disem zuvor und Unit-Tests, Integrations-Tests, Sytem-Integration, sowie Abnahme und Nutzung folgt offensichtlich.

1.2.3 Iterative Modell

1.2.4 SCRUM

1.2.5 Vergleich der Modellvarianten

Kriterium	Wasserfall	Spiral	Scrum	Simultan Engineering
Ablauf	sequentiell	iterativ	iterativ	parallel
Phasentrennung	ausgeprägt	schwach	schwach	fehlt
Durchlaufzeit	lang	lang	kurz	kurz
Feststellung von Fehlern	spät	früher	früh	spät
Aufwand Planung/ Kommunikation				

1.2.6 Ereignis Knoten Netz

Das Ereignis Knoten Netz ist ein Graph, der die Abhängigkeiten von Aufgaben zueinander darstellt. Kann ein Ereignis nicht geschehen, ohne dass zuvor ein anderes abgelaufen ist, so hat dessen Knoten im Graphen eine eingehende Kante von dem Ereignis, das zuvor geschehen sein muss.

Crititcal-Path-Method
Die Crititcal-Path-Method stützt sich auf das Ereignis Knoten Netz, bzw. erweitert dessen Syntax und Semantik. So wird in jedem Knoten die Ereignisnummer, der früheste - und späteste Ereignistermin und der Puffer eingetragen. Die Kanten erhalten zudem noch die Dauer des davor liegenden Verfahrens als gewichtung, sowie eine ID.

j : Ereignisnummer
F : frühester Ereignistermin
S : spätester Ereignistermin
P : Puffer
D : Dauer

1.3 Schätzen, Messen und Steuern

1.3.1 Schätzen

Das Aufwands-Aufrags-Dilemma behandelt das Problem, dass die Zeit die für einen Auftrag angegeben wird den Auftrag sichert. Eine kurze angebene Auftagszeit sichert den Auftrag, birgt jedoch ein hohes Risiko, vice versa eine lange angegebene Auftragszeit.

Es gibt folgende Schätzmethoden:

Methode	Beispiel
Intuitiv	Raten
Vergleich	mehr/ weniger als (unsicher, einfach)
Kennzahlen	Einflussparameter $A = \sum_i c_i E_i$
Zerlegen	Summe von Einzelschätzungen $A = \sum_i A_i$
Skalieren	Dimensionen reduzieren und auf anschauliche Größen reduzieren
Kombinieren	kombination von Methoden
Gruppe	Gruppen schätzen besser als eine Person (Delphi Methode)

Punktschätzung und Aufwand

Es geht darum den Aufwand $E(x)$ und die Standardabweichung $S(x)$ aus gegebenen Parametern zu schätzen. Allgemein gilt

$$E(x) = \frac{a + r \cdot c + b}{2 + r}, V(x) = \frac{(b-1)^2}{u^2}, S(x) = \frac{(b-a)}{u}$$

, wobei $a \leq c \leq b$

Spezialfälle sind:

Parameter	*Formeln*
a, b	$E(x) = \frac{a+b}{2}, S(x) = \frac{b-a}{6}$
a, c, b	$E(x) = \frac{a+4 \cdot c+b}{6}, S(x) = \frac{b-a}{6}$

1.3.2 Reaktionsmöglichkeiten

1. Fall über Plan

Zunächst werden geringe Zeitvorteile als zusätzliche Puffer verwendet. Danach wird der größere Zeitvorteil zur Planrevidierung verwendet.

2. Fall unter Plan (Plan vs. Realität)

Falsche Realität/ Rückstand aufholen:
Überstunden, mehr Personal und/ oder bessere Arbeitsleistungen werden benötigt.

Falscher Plan/ Plan revidieren:
einmalige Verschiebung des Zeitplans bei einmaliger Fehlplanung, Dehnung des Gesamtplans bei systematischer Fehllanung.

Umgang mit Abweichungen

Kleine Abweichungen puffern, mittlere im Projektteam lösen, große mit dem Auftragsgeber.

1.3.3 Risiko

Das Risiko erreichnet sich aus der Risikowahrscheinlichkeit p_i und dem Schadensausmaß S_i

$$R = \sum_{i}^{N} p_i S_i$$

1.4 Anforderungen

1.4.1 Typen und Eigenschaften

Funktional	Nicht Funktional
Benutzer • Eingabe-/ Ausgabeverhalten • Was soll nicht passieren bei Benutzereingaben?	
	Unternehmensanforderungen • Anforderungen an den Entwicklungsprozess
System • **Wie** realisiert das System Benutzeranforderungen?	
	Produktanforderungen • Performanz des Produkts
	Externe Anforderungen • Rechtliche Rahmenbedingungen

Funktionale Anforderungen umfassen Leistungen, die das System abieten soll.
Benutzeranforderungen beschreiben die Dienste, die eine Software leisten soll. Außerdem werden Rahmenbedingungen, unter denen die Software betrieben werden soll, definiert.
Systemanforderungen definieren was wie implementiert werden soll.

Nicht funktionale Anforderungen umfassen Qualitätsanforderungen an Produkt und Prozess.
Unternehmensanforderungen umfassen Anforderungen an den Entwicklungsprozess, sowie an die (Programmier-)Umgebung des Produkts.
Produktanforderungen umfassen Performanz, Zuverlässigkeit, Benutzbarkeit.
Externe Anforderungen umfassen Regulatorische und ethische Anforderungen.

1.5 Spezifikation von Anforderungen

1.5.1 Sophisten Satzschablonen

SYSTEM - sollte/ muss/ wird/ ... - _/ fähig sein/ die Möglichkeit bieten/ ... - OBJEKT - PROZESSWORT

1.5.2 Formale Sprachen

1.6 Erheben und Management von Anforderungen

2 Teil2

2.1 Software Architektur

2.1.1 Architekturmuster

Wir haben folgende Architekutrstile kennen gelernt:

- Objekt orientiert
- Client-Server
- Pipe and filter
- Ereignisorientiert
- Schichten
- Repository
- Process controll

Objekt orientierte Architektur wird beispielsweise bei abstrakten Datetypen verwendet. So wird die interne Datendarstellung von der Außenwelt abgekapselt. Auf mehrere Agenten zerlegbar. Ein Nachteil jedoch ist, dass Objekte die Identität anderer Objekte kennen müssen, um zu interagieren.

Client-Server Architektur verteilt die Anwendungslogik auf eine Menge von Clients und Serversubsystemen. In der Regel werden Clients uns Serversubsysteme auf verschiedenen Maschinen realisiert und kommunizieren über ein Netzwerk. Nachteil hierbei ist, dass die Clients die Identität des Servers kennen müssen.

Peer to Peer, wobei Peers durch einen Server oder Broadcast gefunden werden. Engpässe werden reduziert und die Architektur ist robust gegenüber Peer-Ausfällen. Nachteile sind jedoch, dass der Server ein Engpass werden kann und Peers nur eine unvollständige Sicht-Synchronization ermöglichen.

Pipe and Filter wird u.a. bei Unix Pipes, Compiler Ketten und Signal Prozessoren verwendet. Filter können parrallel implementiert werden und müssen nichts über die verbundenen Elemente wissen.

Ereignisorientierte Architektur findet beispielsweise bei Debuggern und Graphischen Benutzerschnittstellen Anwendung. Es wird auf Ereignisse gelauscht, wobei die Ereignisquelle die Ereignissenken nicht kennen muss. Ein großer Nachteil ist, dass die Komponenten keine Kontrolle über die Berechnungsreihenfolge haben (Endlosschleifen).

Schichtenarchitektur beschreibt die Einteilung von Software in verschieden Schichten (e.g. Benutzerschnittstelle, Benutzerschnittstellenverwaltung, Buisinesslogik, OS/DB) Ein Beispiel für die Schichtenarchitektur wäre das ISO/OSI Modell aus RvS.

Repositories reduzieren die Notwendigkeit komplexe Daten zu duplizieren. Das Blackboard verwaltet dabei alle Daten zentral im einzelnem, gemeinsamen Repository.

Model View Controller beschreibt ein Zentrales Modell mehrerer Schichten, dem Model, View und Controller. Der Controller verarbeitet Updates der Nutzerschicht, der Controller updated das Model und notifiziert die View. Das Modell füllt die View.

REST

2.1.2 Qualität von Architektur

Die Menge von Beziehungen zwischen Subsystemen nennen wir Kopplung, sie ist ein Maß für die Verbundenheit von Elementen. Im allgemienem ist starke Kopplung schlecht, da mit ihr Klassen nur schwer isoliert verstanden werden können, Klassen ohne verbundene Klassen nicht wieder verwendet werden können. Im Gegenzug ist lose Kopplung gut, da sie die Wartbarkeit verbessert und Klassen unanbhängig sind.

Kohäsion ist ein weiterer wichtiger Begriff, geringe Kohäsion bedeutet, dass ein ELement wenig ider unzusammenhängende Funktionalität bietet. Es erschwert Verständnis und Wartbarkeit. Elemente mit hoher Kohäsion sind also einfacher Verständlich/ weniger Komplex.

2.1.3 Weitere Muster

- Vertikale Dekomposition: Anwendung in mehrere unabhängige Anwendungen von vornherein zerlegen.
- Distributed Computing: Horizontale Schnittstelle z.B. Kommunikation über REST
- Sharding: Aufteilung der Eingabe auf dedizierte Prozesse (e.g. Aufteilung nach Anfangsbuchstaben)
- Load Balancing: dynamische Zuordnung der Bearbeitung (e.g. nach Auslastung)
- Microservice Architektur: Applikationen sind Folge unabhängiger Dienste
  - Komponentenzerlegung via Dienste möglich
  - Organisiert um betriebswirtschaftliche Funktionen
  - Entwicklung eher Produkte als Projekte
  - Smarte Endpunkte und dumme Verbindungen
  - Ausfallsicher und Evolutionsfähig
  - Dezentrale Anwendungs- und Datenverwaltung

Die Varianten Sharding und Load Balancing können auch kombiniert werden, um Skalierbarkeit zu erhöhen.

2.2 Konfigurationsmanagement

2.2.1 Source Code Management

Eine Versionsverwartlung hat unter anderem folgende Aufgaben

- Identifizierung der Version
- Festlegung aller zu verwaltenden Bestandteile
- Namenskonventionen und Relation zwischen den Komponenten
- Verschiedene Versionen einer Komponenete müssen abgespeichert werden

- beliebige abgespeicherte Versionen müssen bereitgestellt werden können
- Dokumentation von Änderungen
- Festlegung und Kontrollieren von Zugriffsrechten
- Verwaltung des Komponenten-Repositories

Commits:

Ein Commit enthält mindestens ein Datum, den Author, eine neue Versionsnummer und eine optionale Begründung. Der Commit beschreibt also die Menge von Änderungen.

Merge vs. Rebase:

Bei einem Rebase wird der aktuelle Entwicklungsstand des Masterbranches akzeptiert und davon aus weiter gearbeitet. Jedoch kann bei Parraleler entwicklung ein Merge Konflikt auftreten. Ferner geht bei einem Rebase sämtliche Information eines Branches verlore/ ein Branch wird zerstört.
Bei einer Merge Operation wird ein Branch in Stand gehalten und die kritischen Sektionionen reviewed/ gesolved bevor ein Update/ Push zum Masterbranch erfolgt.

Beim Arbeiten mit GIT wird oftmals GIT-Workflow verwendet. Dabei existieren Branches für master (Production), hotfixes, release, development und einzelne Features.

Der Rosinenpick ist eine Operation, bei der ein einzelner Commit in einen anderen Branch übernommen wird, ohne dabei die anderen Commits mit zu übernehmen.

2.2.2 Build-Automatisierung und Dependency Management

Nicht relevant.

2.2.3 Release-Strategien

Continuous integration

Ein ständiges zusammenführen von Komponenten zu einer ANwendung, dabei ist das Ziel die Steigerung der Softwarequalität. Oftmals in Kombination von Tests und Qualitätsmetriken.

Continuous Delivery

Maunuelle Software Deployments und Konfiguration der Produktivumgebung. Software wird erst nach ihrer Fertigstellung in einer der Produktivumgebung ähnlichen Testumgebung getestet.

Deployment Pipeline

2.3 Meta-Modellierung und Domänenspezifische Sprachen

2.3.1 Meta-Modellierung

Nach Stachowiak hat ein Meta Modell 3 Merkmale: Abbildungs-, Verkürzungs- und pragmatisches Merkmal. Das Modell soll die Beschreibung der Realität vereinfachen. Die Welt wird also auf eine diskrete Struktur abgebildet. Dabei werden Struktur, Beziehungen und Verhalten modelliert.

Hinsichtlich UML gelten folgende Anwendungsbereiche:

- Anwendungsfalldiagramm: Anforderung an das System
- Klassendiagramm: Datenstruktur des Systems

- Objektdiagramm: ein Instanz des Models/ Systems
- Aktivitätsdiagramm: Steuerung des Ablaufs zwischen Komponenten

3 Teil 3 Modellieren

3.1 Domänenspezifische Sprachen/ Modellierung und Verhalten

3.1.1 Metamodelle

Ein Metamodell besteht im wesentlichen aus 3 Teilen, der konkreten C und abstrakten A Syntax, sowie einer Semantik S. Dabei reden wir von einer syntaktischen $M_S : A \to C$ und einer semantischen $M_C : A \to S$ Abbildung, die bestimmt was wie beschrieben wird, bzw. die Bedeutung der Beschreibung.

3.1.2 Modellierung von Verhalten

Im wesentlichen haben wir zum Modellieren des Datenfluss Blockschaltdiagramme und für den Kontrollfluss erweiterte endliche Automaten verwendet.

Als ein Negativbeispiel für Modellierung wurden State-Charts erwähnt, zu denen bis heute keine Semantik vollständig festgelegt wurde.

Blockschaltdiagramme:

Blockschaltdiagramme arbeiten ohne implizieten Speicher, es handelt sich um eine Abbildung auf logische Ausdrücke mit Variablen. Die Semantik wird durch die Auswertung dieser Ausdrücke unter einer Belegung der Variablen realisiert.

Erweiterte endliche Automaten:

Erweiterte endliche Automaten kombinieren Ausdrücke mit dem Automaten-System/Modell, die Semantik erfolgt über das Transitionssystem.

3.1.3 Domänenspezifische Sprachen

??? EMF ???

Code Generierung:

Für die Generierung des Codes wird ein Template basierter Ansatz verwendet. Sprich es existieren Dateien mit Vorgefertigtem Code/ Vorlagen, die durch Informationen aus dem Modell ausgefüllt werden.

??? Xtend ???

3.2 Qualität

3.2.1 Software Qualität

Die Qualität von Software ist immer relativ zu ihren Anforderungen. Es wird anhand der Conformance (Erfüllung der Anforderungen) die Qualität gemessen. Bei nicht Erfüllen einer Anforderungen wird zwischen Faults (Fehlern) und Failures (Ausfällen) unterschieden.

3.2.2 Code Qualität

- Reliability/ Resilliance: Wie gut geht die Software mit Fehlern um?

- Performance/ Efficiency: Wie Effizient ist die Software?
- Security: Wie sicher ist die Software für einen Benutzer?
- Maintanbility: Wie schwer ist es die Software zu pflegen/ warten?

3.2.3 Code Standards

Standards stellen sicher das Code wartbar/ lesbar ist und ggf. andere nicht-funktionale Anforderungen erfüllt. Code Standards helfen eine Basis für guten Code zu schaffen hinsichtlich:

- Dokumentation (Maintanbility)
- Übersichtlichkeit (Maintanbility)
- Konventionien für e.g. Passwörter (Security)
- Behandelung von Exceptions (Reliability)
- Nutzung von String-Buildern an Stelle von Konkatenationen (Efficiency)

3.2.4 Code Review

Reviews stellen gemeinsames Verständnis sicher und finden Fehler möglichst früh. Andere Menschen setzten sich mit dem Code eines Authors auseinander, dadurch werden Fehler und Bugs in der Regel schneller gefunden, sowie der Code verbessert und es Tritt kein Chaos in der Endphase eines Projekts auf.

Im Rahmen der Code Reviews haben wir folgende Methoden kennen gelernt:

- Ad-hoc: Einen Kollegen fragen
- Peer Desk-Checking: Ein Reviewer führt den Code/ das Produkt aus.
- Pair Programming: Einer programmiert und erzählt, die ander Person hört zu und checkt.
- Walkthrough: Der Author präsentiert vor Reviewern.
- Team-Review: Mehrere reviewen individuell, Resultate werden im Meeting besprochen.

3.2.5 Code Metriken

Metriken helfen bei der Bewertung der Qualität von Code.

Kennen gelernte Metriken sind:

- Lines Of Code (LOC): Anzahl der Zeilen
- Lack Of Cohesion (LCOM): Wie stark hängen alle Komponenten einer Klasse zusammen?
- Cyclomatic Complexity (CC): Wie Kompliziert ist eine Methode geschrieben?

LCOM:

$$LCOM = \frac{1/a \cdot \sum_{i=1}^{a} n(A_i) - m}{1 - m}$$

CC:

$$CC = e - n + 2 \cdot p$$