Software Engineering

Qualität

Software-bezogene Qualität:

- **Prozessqualität:** Qualität des Projekts, bzw. des **Prozesses** indem/mit dessen Hilfe die Software entwickelt wird.
- **Produktqualität:** Qualität des **Ergebnisses** der Herstellung , d.h. die **Software** selber.

Produktqualität:

- Brauchbarkeit
    - Bedienbarkeit (wie gut die Software in der Lage ist, Aufgaben zu erfüllen)
        - Einfachheit (hoch, wenn SW dem Benutzer konzeptionell einfach erscheint)
        - Verständlichkeit (Benutzer versteht rasch, wie er mit SW umgehen muss)
        - Konsistenz (SW verhält sich ähnlich in ähnlichen Situation)
        - Handbuchvollständigkeit (Alle Fragen des Benutzers können beantwortet werden)
    - Nützlichkeit
        - Leistungsvollständigkeit (SW erbringt alle geforderten Leistungen)
        - Sparsamkeit (SW benötigt kaum Speicherplatz und andere Betriebsmittel)
        - Effizienz (SW benötigt kaum Rechenzeit, als minimal erforderlich wäre)
    - Zuverlässigkeit
        - Genauigkeit (Resultate vom mathematisch korrekten Resultat nur wenig Abweichungen)
        - Ausfallsicherheit
        - Korrektheit (Spezifikation zutreffend, SW korrekt)
- Wartbarkeit (Fähigkeit der Software, in Zukunft geändert/gewartet zu werden, ohne großen Aufwand und Kosten zu verursachen)
    - Portabilität
        - Abgeschlossenheit
        - Geräteunabhängigkeit
    - Änderbarkeit
        - Lesbarkeit
        - Knappheit
        - Simplizität (hoch, wenn nur wenige schwer verst. Konstruktionen enthalten sind)
        - Strukturiertheit (hoch, wenn SW logisch abgeschlossen und in hohem Zusammenhalt und geringer Kopplung gegliedert ist)
    - Prüfbarkeit
        - Testbarkeit (Ausführung ist reproduziertbar. Vollständiges Erfassen der Resultate)
        - Lokalität (hoch, wenn Fernwirkungen in SW vermieden sind.)
        - Spezifikationsvollständigkeit

Architektur

Architekturmuster

- sind Muster, die für die Organisation und Strukturierung von Software-Systemen verwendet werden.
- beschreiben, wie die Komponenten eines Systems miteinander verbunden sind und wie Steuerungsfluss und Datenfluss gestaltet sind.
- sind auf höherer Ebene als Desing-Patterns.
- sind normalerweise unabhängig von einer Programmiersprache.
- Beispiele: Model-View-Controller(MVC), Event-Driven Architecture(EDA), Microservices Architecture
- **Zusammenfassend: Architekturmuster beziehen sich auf die große Struktur des Systems.**

Architekturmuster - Aufteilung

- Strukturierende Architekturmuster
    - **Pipes und Filters** (Methode, um eine Aufgabe in kleineren Unteraufgaben (Filters) zu unterteilen, die dann in einer bestimmten Reihenfolge (Pipes) ausgeführt werden)
        - **Kontext:** System mit vielfältigen Aufgaben
        - **Probleme:**
            - Aufgaben sind fast immer unterschiedlich
            - System muss flexibel bleiben
        - **Lösung:**
            - Nacheinanderschalten verschiedener (unabhängiger) Komponenten für einzelne Teilaufgaben
        - **Vorteile:**
            - Einfacher Aufbau
            - Gute Wiederverwendbarkeit
            - Unabhängigkeit der Einzelschritte erleichtert Wartung
            - Parallelisierung der Einzelschritte möglich
        - **Nachteile:**
            - Datenformate meist rudimentär
            - Geschwindigkeit und Latenzzeiten abh. vom langsamsten Teil
        - **Beispiele:**
            - Compiler, Konsolenbefehle der Unix-Shell, Funktionale Programmierung, Verteilte Systeme

---

    - **Schichten**
        - **Kontext:** Ein großes/komplexes System, das aufgeteilt werden muss
        - **Probleme:**
            - Abhängigkeiten zwischen High- und Low-Level-Funktionalität
            - Austauschbare Komponenten
            - Integration bestehender Systeme
        - **Lösung:**
            - Einteilung in verschiedene angemessene Schichten
            - Schitenanordnung übereinander

- **Vorteile:**
  - Klare Aufgabenteilung
  - Klarer Datenfluss
  - Austauschbarkeit der Schichten
  - Interface-Änderungen betreffen nur benachbarte Schicht
- **Nachteile:**
  - Schichtentrennung nicht einfach
  - Overhead (alle Schichten werden durchlaufen)
  - Cascading Changes bei Redesign
- **Beispiele:**
  - Netzwerkprotokolle: OSI-Schichtenmodell, Betriebssystemschichten: Kernel, Treiber, Anwendungen

---

- Architekturmuster für interaktive Systeme
  - **MVC**
    - **Kontext:** Austauschbare Benutzerschnittstellen
    - **Probleme:**
      - Häufige Änderung der Anforderungen an die Benutzerschnittstelle
      - Mehrere Benutzerschnittstellen
    - **Lösung:**
      - Aufteilung in Komponenten: Datenmodell (Datenbank), Anwendungslogik und Präsentation
      - Reduzierung von Abhängigkeiten
    - **Vorteile:**
      - Leichter austauschbare Benutzerschnittstellen
      - Verwendung mehrerer Benutzerschnittstellen gleichzeitig
    - **Nachteile:**
      - Komplexe technische Realisierung
      - Enge Bindung von View und Controller an Modell
    - **Beispiele:**
      - Webframeworks (Spring MVC, Ruby on Rails, Django)

- Architekturmuster für verteilte Systeme
  - **Vermittler-Prinzip**

    Kontext: Verteiltes System mit (unabhängigen) Komponenten, die zusammenarbeiten
    - **Probleme:**
      - Komponenten sollen andere Dienste anderer Komponenten abrufen können
      - Weitere Dienste sollen zur Laufzeit ausgetauscht, hinzugefügt oder entfernt werden
      - System soll keine Implementierungsdetails preisgeben
    - **Lösung:**
      - Einführung eines Vermittlers (Broker oder Mediator)
      - Service Provider teilen ihre Dienste einem Vermittler mit

        - Clients fordern Dienste vom Vermittler and, dieser leitet die Clients an passende Service Provider weiter.
    - **Vorteile:**
        - Einfache Kombination von Diensten
        - Leicht erweiterbar
    - **Nachteile:**
        - Standardisierung der Kommunikation notwendig
        - Overhead für Vermittlung
    - **Beispiele:**
        - Chat-Server

- **Client-Server**

  Kontext: Verteilte Benutzer, zentrale Anwendung
    - **Probleme:**
        - Verteilte Nutzer, verschieden Funktionalität
        - Skalierbarkeit (Anzahl Nutzer)
        - Konsistenz, Vertraulichkeit
    - **Lösung:**
        - Verteilzung der Aufgaben auf Netzwerkkomponenten
    - **Vorteile:**
        - Verteilt
        - Ein Dinest, viele Nutzer
        - Skalierbar
    - **Nachteile:**
        - Performanz und Fehlerbehandlung im Netzwerk
        - Gleichzeitige Nutzer vs Daten/Zustands-Konsistenz
        - Sicherheit
        - Veritelungsmanagement
    - **Beispiele:**
        - WWW(HTTP, FTP), E-Mail(IMAP,POP3)

- Dienstorientierte Architektur
    - **Peer-to-Peer**

Design-Patterns

- sind Muster, die auf der Implementierung von einzelnen Komponenten eines Systems auf Basis von bestimmten Problemen aufbauen.
- beschreiben, wie bestimmte Probleme in einer bestimmten Programmiersprache gelöst werden können.
- sind auf tieferer Ebene als Architekturmuster.
- sind für bestimmte Programmiersprachen gültig.
- Design Pattern mit "globalem" Einfluss auf SW werden oft auch als Architekturmuster angesehen.
- Beispiele: Singleton, Factory Method, Observer

- **Zusammenfassend: Design-Patterns beziehen sich auf kleinere Strukturen und die Implementation von einzelnen Komponenten.**

Asynchrones Programmieren

Futures und Promises

Konzepte, die es ermöglichen asynchrone Operationen auszuführen und das Ergebnis zu einem späteren Zeitpunkt zu erhalten.

Futures

- Um asynchrone Operationen in **asynchronen Umgebungen** auszuführen
- einfacher und schneller zu implementieren als Promises, da sie nur das Ergenis einer asyn. Operation bereitstellen, aber **keine Möglichkeit** habem, das Ergebnis zu ändern.
- Beispiel:

```
const { setTimeout } = require("timers");
const { Future } = require("fluture");

const fetchData = () =>
Future((rej, res) =>
    setTimeout(() => res("Data retrieved successfully"), 2000)
);

fetchData().fork(console.error, console.log);
```

In diesem Beispiel wird die Funktion fetchData() eine Future zurückgeben die nach 2 Sekunden das Ergebnis "Data retrieved successfully" liefert. Die fork Methode wird aufgerufen, um das Ergebnis zu erhalten, die erste Funktion ist für einen Fehlerfall zuständig die zweite für den Erfolgfall.

Listenable Futures

- Können nützlich sein, um asynchrone Operationen in einer bestimmten Reihenfolge auszuführen.
  → Listeners werden in der Reihenfolge registriert, in der sie aufgerufen werden.
- Beispiel;

```
const listenableFuture = new ListenableFuture();

listenableFuture.addListener(result => {
    console.log(result);
});

listenableFuture.setResult("Data retrieved successfully");
```

In diesem Beispiel erstellt die ListenableFuture() eine neue Listenable Future, und addListener(result => {}) registriert einen Listener, der aufgerufen wird, wenn das Ergebnis der asynchronen Operation verfügbar ist. Die setResult Methode wird aufgerufen, um das Ergebnis der asynchronen Operation zu setzen und den registrierten Listener aufzurufen.

Promises

- Um asynchrone Operationen in **synchronen Umgebungen** auszuführen
- ähnlich wie Futures, aber mit dem Unterschied, das man eine Promise ändern kann.
- haben meisten eine methode `then` um eine Callback-Funktion zu registrieren.
- Beispiel:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data retrieved successfully");
        }, 2000);
    });
}

fetchData().then(result => {
    console.log(result); // "Data retrieved successfully"
}).catch(error => {
    console.log(error);
});
```

In diesem Beispiel erstellt die Funktion fetchData() eine neue Promise, die nach 2 Sekunden das Ergebnis "Data retrieved successfully" liefert. Die Methode then der Promise wird aufgerufen, sobald das Ergebnis verfügbar ist und gibt das Ergebnis als Argument an die übergebene Callback-Funktion weiter.

Event basierte Architektur

Event Notification

Idee:

- Sendung von Event-Nachrichten, um über bereits vorgenommene Veränderungen zu benachrichtigen.
- Ausgelöstes Event enthält `keine Details`
  - **Vorteile:**
    - Einfach, neue Systeme umzusetzen
  - **Nachteile:**
    - nicht einfach festzustellen, welche Systeme alle von Daten abhängen.
    - Resource Contention

Event Carried State Transfer

Idee:

- Für die Aktualisierung notwendigen Daten werden im Event mitgeschickt.
    - **Vorteile:**
        - einfach, neue Systeme umzusetzen.
        - Jedes abhängige System kann seine Sicht auf die Daten vorhalten.
          → Vermeidung von „Resource contention”
        - widerstandsfähiger, weil Empfänger unabhängig vom Sender funktionieren
    - **Nachteile:**
        - nicht einfach festzustellen, welche Systeme alle von Daten abhängen.
        - nurnoch "eventually consitent"
        - Großes Datenmengen im Event

Event-Sourcing

Idee:

- Modifizierung durch Events
- Event-Logs (Aufzeichnung der Änderungen) → "Zeitreisen" möglich. Vergangenheit kann wiederhergestellt werden
    - **Vorteile:**
        - Überprüfbarkeit
        - Debugging
        - Skalierbarkeit
        - Live Backup Systeme
    - **Nachteile:**
        - ungewöhnlich
        - Events müssen langlebig sein
        - Alles muss als Event repräsentiert werden
        - Namensgebung

CQRS

Idee:

- Vergleichbar: Command Query Seperation (CQS)
- Idee von CQS: Mit einer Datenstruktur nur Lesend, oder nur Schreibend interagieren.
- Idee von CQRS: Es gibt ein Datenmodell zum Lesen, und ein Datenmodell zu Schreiben.
- Trennung der Modelle → komplexe Ansichteneffektiv im Lese-Modell realisieren, ohne das Schreib-Modell zu beeinflussen.
- Lese-Modell = gut skalierbar, d.h. es kann z.B. auf dem Client mithilfe von einem „Push” basierten Synchronisationsmechanismus und Events kontinuierlich aktuell gehalten werden.

Architektur dokumentieren (arc42)

---

1. Einführung und Ziele

    - Aufgabenstellung
        - Was macht die Software?
    - Qualitätsziele
        - Wesentliche Qualitätsziele
    - Stakeholder
        - Aufzählung von Stakeholders, ggf. Einsatz von Personas

2. Randbedingungen

    - Technische Randbedingungen
        - Verwendete Hardware
        - Clouddienste
        - Betriebssysteme, VMs
    - Organisatorische Randbedingungen
        - Mitglieder des Teams
        - Zeitliche Vorgaben
        - Lizenzen
        - Einsatz verwendeter Werkzeuge
    - Konventionen
        - Allg. Regeln:
            - z.B. Open-Source Bibliotheken sind Closed-Source Bib. vorzuziehen
        - Terminologie
        - Coding Conventions

3. Kontextabgrenzung

    - Fachlicher Kontext
        - Welche Benutzer hat das System (`Use-Case`)
        - Mit welchem Fremdsystem interagiert das System
    - Technische- oder Verteilungskontext
        - Welche Fremdsysteme?
        - Technische Standards? Welche Dateiformate?
    -

4. Lösungsstrategie

    - Wie funktioniert die Lösung
    - Fundamentale Lösungsansätze

5. Bausteinsicht

    - Statische Struktur
    - Aufteilung in Subsysteme oder Komponenten
    - Beschreibung wichtiger Bestandteile (`Klassendiagramme, Komponentendiagramme`)

6. Laufzeitsicht

    - Darstellung dynamischer Aspekte
    - (`Sequenzdiagramm, Zustandsdiagramm, Aktivitätsdiagramm`)

7. Verteilungssicht
    - Welche Artefakte laufen auf welchem Knoten (`Verteilungsdiagramm`)
8. Konzepte
    - Wiederkehrende Muster und Strukturen
    - Fachliche Strukturen
9. Entwurfsentscheidungen
    - Darstellung wichtiger Entscheidungen
    - Funktionale/Veränderbare Datenstrukturen
    - Aufzeigen von Alternativen
    - Analyse der Konsequenzen
10. Qualitätsszenarien
    - Beinhaltet Qualitätsszenarien
    - Qualitätsbaum
    - Bewertungsszenario
11. Risiken
    - Unsicherheiten werden dargelegt
        - Mögliche Strategie zeigen, falls Risiko eintreten sollte
12. Glossar

Testen und Verifizieren

Begrifflichkeiten

- Fehlbedienung (Error/Mistake): Person bedient Programm nicht wie vorgesehen.
  → bessere UI, Handbuch, bessere Fehlermeldung
- Defekt (Fault/Defect/Bug): Fehlverhalten des Programms
  → Review, Programmanalyse
- Fehler (Failure): Fehlverhalten des Programms gegenüber der Spezifikation, dass während seiner Ausführung auftritt
  → Test

Unterschied Fehler zu Defekt:
Defekt ist ein unbefriedigendes Verhalten oder eine Abweichung von den Anforderungen, während ein Fehler die Ursache des Defekts beschreibt. Ein Defekt kann also durch mehrere Fehler verursacht werden und umgekehrt kann ein Fehler mehrere Defekte verursachen.

- Beispiel: Ein Defekt kann sein das Programm stürzt ab, ein Fehler kann sein das eine Variable nicht richtig zugewiesen wurde und dadurch stürzt das Programm ab.

Assert

- **Definition:**
  Aktuelle Wert einer oder mehrerer Speicherstellen wird gegen erwartete Werte getestet.
  → `Vergleich mit Erwartungswerten`
- **Einsatz von nativen Asserts:**
  - Überprüfung von internen Invarianten
  - `Fail-Fast:`
  - Defekte sind einfacher zu finden
  - Es gibt wenige "tote Defekte"
  - Nachteil: "Tote Defekte" eskalieren zu einem Fehler

Manuelles Testen

Freies Testen

- **Definition:**
  Freies Testen (Keine/wenig Vorgaben)
- **Einsatz von manuelles Testen:**
  - Nützlich um Feedback zu Einfachheit, Verständlichkeit, Konistenz zu erhalten
  - Immer von neuen Personen durchführen lassen
  - Release einer Alpha- und Betaversion

Angeleitetes Testen

- **Definition:**
  Testplan um Ablauf und Erwartung des Ergebnisses genau festzugelegen.
- **Einsatz von angeleitetes Testen:**
  - Nützlich um Funktionstests durchzuführen
  - Überprüfung der Implementierung
  - Nachteil: Spezifikation überprüfen nicht möglich
  - Ziel: Vermeidung von „peinlichen” Fehlern im Release.
- **Testplan:**
  - Einführung
  - Überblick über das System
  - zu testende Aspekte
  - Testkriterien
  - generelle Anweisungen zum Testablauf
  - Voraussetzungen für die Tests, z.B. Software- und Hardwareanforderungen
  - Liste von Testfällen, pro Testfall konkrete Beschreibung des Testablaufs, und des erwarteten Verhaltens

Unittests

- **Definition:** Konzentration auf einzelne Komponenten um Qualität zu verbessern
- **Vorteile:**
  - Besserer Fokus
  - Übersichtlich
  - Umsetzbar während der Entwicklung
  - Gut um Kernfunktionalität zu testen

    - Lokale Fehlersuche möglich
- **Nachteile:**
    - Kann nicht das Zusammenspiel mit anderen Komponenten sicherstellen
    - Entspricht nicht Sicht des Nutzers; eher technische Sicht.

Integrationstests

- **Definition:**
  Testet mehrere Module, die im Zusammenspiel korrekt funktionieren sollen.

Funktionsorientierte Tests

Testgrundlage: Spezifikation

- **Vorgehensweisen:**
    - Äquivalenzklassenbildung
        - Unterteilung der Eingabeparameter in Äquivalenzklassen
        - Klasse sollte Werte enthalten, für die die Spezifikation aussagt, dass das Verhalten für alle Elemente der Klasse gleich sein sollte
        - Einteilung in gültige und ungültige Klassen (gültige/ungültige Eingaben)
    - Zustandbasierter Test
        - Zustandsautomat (Test auf gültige Übergänge)
        - Problem: Zyklische Automaten sind nicht endlich
          → Beschränkung auf Abläufe einer gewissen Länge/Dauer
    - Ursache-Wirkungsanalyse
        - Je nach Sicht, wird die Kombination von mehreren Parametern nicht in die Äquivalenzklassenbildung einbezogen.
          → Abhängigkeiten der Parameter mithilfe von Entscheidungstabellen und Ursache-Wirkungs-Graphen darstellen

Strukturorientierte Tests

- **Kontrollflussgraph (KFG):**
    - wie Ablaufdiagramm
        - Knoten entsprechen Anweisungen im Quellcode
        - Kante von Knoten1 zu Knoten2, wenn nach Anweisung1 die Anweisung2 ausgeführt werden kann
    - Anweisungsüberdeckungstest:
        - Jede Anweisung im KFG muss durch einen Test überdeckt werden
    - Zweigüberdeckungstest:
        - Jede Kante im KFG muss getestet werden
    - Bedinungsüberdeckungstest:
        - Jeder Ausdruck wird in atomare Teilausdrücke zerlegt.
            - Es wird verlangt, dass alle Teilausdrücke jeweils mit `True` oder `False` ausgewertet wurden