Software engineering

Ziel: create a solution to a problem

Das Problem muss jedoch nicht unbedingt ein Software-Problem sein.

Beispiel: uber – ist kein Software-Problem gewesen

https://www.youtube.com/watch?v=BCYlUmzMmyc

1. Was macht einen guten Software-Engineer aus?
   Programmierkenntnisse, wie viele Programmiersprachen kann man, aber unbedingt auch die Kommunikation - sprechen mit Auftraggeber und Teammitglieder. Dazu gehört auch Dokumentation der Arbeit. Würde jemand kündigen, dann wüsste der Nachfolger vieles nicht.
2. Man muss die Fähigkeiten richtig zeigen – also auch eigenes Marketing betreiben
3. Ein guter Software-Engineer ist ein Kreator (Erschaffer). Es steht die Frage im Vordergrund „Warum“ sollte ich dieses Problem lösen nicht „wie“ löse ich es.

Was ist Software-Engineering:

Wie entsteht Software? Vor allem in einem Team? Einfach drauf los programmieren?

Praktische Anwendung wissenschaftlicher Erkenntnisse, um Software wirtschaftlich zu entwickeln, einzusetzen und zu warten.

Es geht nicht nur um Programmieren, sondern auch viel um **Projektmanagement und um Kundenzufriedenheit**. Die Zielgruppe kann sehr unterschiedlich sein, z.B. kann es sich um Spieleentwicklung usw. handeln.

Ziele:

- Erstellen von Qualitätssoftware
- Aufwände minimieren
- Kundennutzen maximieren

Aufgabe:

Aus einer Idee soll eine lauffähige Anwendung entwickelt werden.

- Umwandlung der Idee in definierte Anforderungen
- Entwurf einer Zielarchitektur
- Entwicklung des eigentlichen Produktes
- Testen und
- Bereitstellung in einer Zielumgebung und
- anschließende Überwachung des Betriebs

Diese 6 Schritte werden (in der Theorie) nacheinander (sequenziell) durchlaufen (Wasserfall). Jeder dieser Schritte erzeugt ein Ergebnis, ein Artefakt. Das kann ein Dokument, ein Quellcode eine Liste oder die fertige Anwendung.

Es gibt 2 Vorgehensmodelle:

1. Wasserfallmodell
2. Scrum

1)Wasserfallmodell:

Die Phasen folgen zeitlich nacheinander, nicht nebeneinander. Es ist somit ein lineares Modell. Alles der Reihe nach.

Vorteile:

- die Phasen sind klar voneinander abgegrenzt und lassen sich sehr gut kontrollieren. Systematisches Vorgehen möglich.
- Da die Ziele und Meilensteine definieret sind, können die Kosten gut geplant werden.

Nachteile:

- klare Abgrenzung der Phasen sind nicht immer möglich in der Realität; man könnte somit nicht in die Programmierung zurückspringen, um Änderungen vorzunehmen. Die nötige Flexibilität ist begrenzt.
- In der kompletten Durchführungszeit dürften sich keine Anforderungen ändern, die z.B. auf neuen Geschäftsmodellen oder neue Technologien reagieren sollten.

Schritte im Vorgehensmodell Wasserfall:

1. **Projektfindung**
   Eventuell ist ein Auftrag vorhanden, es gibt schon Vorentwicklungen oder einen Forschungsdurchbruch.
   Ziel ist, aus einer Idee eine lauffähige Anwendung zu entwickeln.
   Es beginnt mit der Idee. Aber um das Risiko von Missverständnissen zu vermeiden und um den Gesamtaufwand in der Softwareentwicklung „planbar“ zu machen, müssen diese Ideen in ein festes Format aufgenommen werden und als Anforderungen aufgeschrieben werden -> siehe nächster Punkt.

2. **Anforderungsanalyse (requirements engineering)**
   Planung: Setzen von Meilensteinen, Projektplan, Kostenrechnung, Lastenheft
   Ziel:
   Kundenanforderungen ermitteln, dokumentieren, abgestimmt und verwalten
   Stakeholder sind Personen, die ein Interesse daran haben, dass die Software neue Funktionen bekommt, dass Fehler behoben werden oder sonstige Änderungen durchgeführt werden. Sie sind Ideengeber.
   Diese Ideen sind oft nicht formal genug und um Missverständnisse zu vermeiden, müssen diese Ideen in einem **festen Format** aufgenommen und als sogenannte Anforderungen verschriftlicht und als Pakete für die Entwicklung eingeplant werden. Das feste Format hat einen Titel, eine Beschreibung und mehrere Akzeptanzkriterien.
   Arten von Anforderungen: funktionale und nicht-funktionale

a. **Funktionale** Anforderungen: beschreiben, was das System tun soll, wie Systemverhalten, Daten (wie werde sie ein- oder ausgegeben), Input/Output z.B. durch Klicken von Button. Bespiel: die Software soll basierend auf einer Formel die Kreditwürdigkeit berechnen.
b. Neben diesen gibt es auch **nicht-funktionale** Anforderungen. Diese sind teils qualitative Aussagen, wie z.B. Ausfallsicherheit, Performance.
Beispiel: Anforderung: ein User soll sich zu einem Workshop anmelden können, z.B. mit E-Mail- und Passwort-Feldern

Die Requirements (=Anforderung) sollen keine Widersprüche aufweisen und eindeutig (nicht: „soll etwas berechnen und schnell sein“) sein.

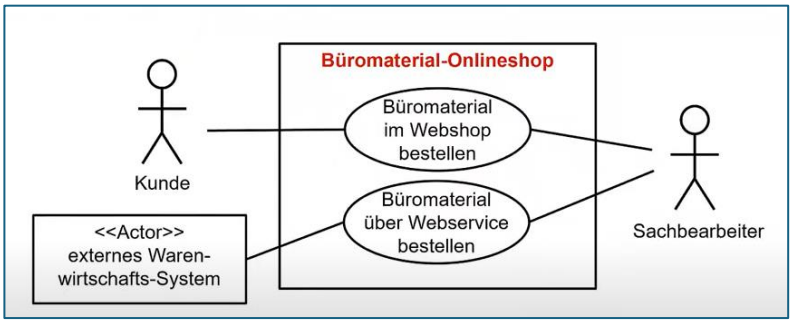
Beispiel für Anforderungen:

- Funktionale:
  - Kunden Konto-Management
    - Kunde kann sich registrieren mit Name, E-Mail und Passswort
    - E-Mail Verification für Sicherheit
    - Erlaubt dem Kunden einzuloggen und auszuloggen
  - Inhalt Management
    - Kunde kann Inhalt erstellen, ansehen, updaten und löschen
    - Jeder Inhalt sollte einen Namen und Beschreibung haben
    - Kunde kann den Inhalt als fertig und komplett markieren
  - Teilen
    - Kunde kann den Inhalt teilen
  - Responsives Design
    - Sicherstellen, dass die App auf verschiedenen Devices funktioniert
- Nicht funktionale Anforderungen:
  - Performance
    - Schnelles Laden und gute responsive Interaction für ein gutes Kunden-Erlebnis
  - Skalierbarkeit
    - Das Backend für eine steigende Anzahl von Kunden erstellen
  - Sicherheit
    - Sichere Kundendaten mit angemessener Verschlüsselung und sicheren Kommunikations-Protokollen
    - Schutz vor SQL-Injections und XSS (Cross-Site Scripting: z.B. wird das schädliche Skript sofort ausgeführt, wenn der Benutzer auf einen präparierten Link klickt oder eine präparierte URL aufruft. Führt z.B. zu Datendiebstahl, Malware-Verteilung))

Use Case

- Use Cases werden nach der Anforderungsbeschreibung erstellt. Danach wird ein Use Case Diagramm gezeichnet.
- Anwendungsfall, Geschäftsvorfall – ist eine zusammenhängende Menge an Aktivitäten, wird genutzt, um das System anforderungsseitig zu beschreiben
- Wird immer aus Nutzersicht formuliert. Nutzer sind Aktoren.

Beispiel Use Case Diagramm: Onlineshop für Büromaterial



https://www.youtube.com/watch?v=PmYtAh4dV6s

Es gibt 2 Use Cases: 1.) Büromaterial im Webshop bestellen
2.) Büromaterial im Webservice bestellen

Der auslösende Aktor ist ein menschlicher Kunde, der im Webshop bestellt. Der sekundäre Aktor ist der Sachbearbeiter, der alles zusammenstellt und die Adresse prüft.

Der 2. Use Case wird durch einen nicht menschlichen Aktor ausgelöst. Das könnte ein automatischer Vorgang einer Bestellsoftware eines Kunden sein, der über eine API angeschlossen ist.

Typische Use Case Bezeichnungen sind „Material bestellen“ (Namenskonvention: Objekt + Verb im Infinitiv).

Beispiel:
Use Case 1: Kunde manuell anlegen
Use Case 2: Foto hochladen

3. **Architektur, Design, Planung, Entwurf – noch kein Code**
   Dabei wird eine Art Bauplan mit einer Struktur entworfen. Genaue Strukturen und Use-Case werden erstellt.
   Beispiel: Wie genau wird eine Eingabe gespeichert?
   Weitere Überlegungen sind:
   Design/Entwurf, Layout. Lastenheft verfeinern, Welche Programmiersprache? Welche Datenbank soll verwendet werden? ER-Modell für Datenbank.

   a. **Entwurfsprinzip: KISS**
      Keep it simple and stupid
      fordert, dass etwas verständlich und einfach ist. Das heißt nicht, dass der Code einfach sein soll. Die „simple“ Lösung soll die Qualitätsattribute (nicht-funktionale Attribute) einfach umsetzen können. Dazu sollen einfache Werkzeuge reichen. Das betrifft auch die Programmierfähigkeiten von Mitarbeitern.
      Man sollte sich die nötigen Erfordernisse überlegen, wie zum Beispiel
      - zum Speicher von Daten in einer Anwendung reicht eine JSON-Anwendung und es muss nicht gleich eine relationale Datenbank sein
      - braucht man Docker, auch wenn das keiner im Team wirklich gut beherrscht?

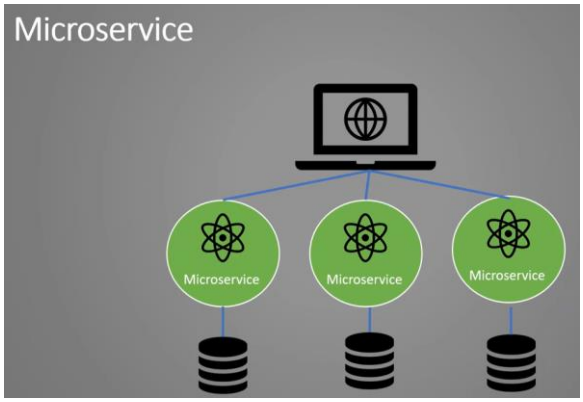
- Kann ein durchschnittlicher Programmierer damit dann auch umgehen?
- Kann er es einfach verstehen warte und erweitern oder testen?
- Muss der Programmierer sehr erfahren sein oder reicht ein durchschnittlicher Kollege?

b. **Entwurfsprinzip**: Design Patterns
Pattern/Muster: Lösungen für bestimmte Probleme – Standardlösung, aber müssen nicht in jedem Projekt vorkommen
Design Entwurfsmuster sind bereits erprobte und vorhandene Lösungen und man kann darauf zugreifen. Diese sind katalogisiert und können für bestimmte Problemlösungen übernommen werden. Dabei werden vorhandene Lösungen verwendet und man einigt sich im Team darauf, diese zu verwenden. Dabei sollten alle Mitglieder diese kennen und einsetzen können.

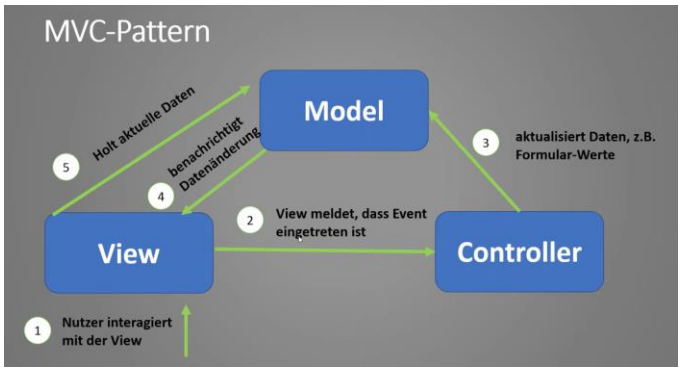
c. **Architektur:**
Architektur ist die Struktur eines Softwaresystems.
Dazu gehört das Zusammenspiel von Diensten, Schichten und Komponenten. Welche Komponenten hat man, wie kommunizieren diese miteinander und wie ist deren Laufzeitverhalten. Komponenten sind der Überbegriff für Klassen und Methoden, die darin gesammelt werden.

System-Architektur:

- Monolith - ist eine einheitliche, zusammenhängende Anwendung, bei der alle Teile eng miteinander verbunden sind
- Client-Server Architektur - teilt die Anwendung in zwei Hauptkomponenten auf, die Client und Server, die über ein Netzwerk miteinander kommunizieren.
- N-Tier-Architektur: unterteilt in mehrere logischen Schichten oder “Tiers”. Jede Schicht hat eine spezifische Aufgabe und kommuniziert nur mit den benachbarten Schichten. Sehr häufig wird die 3 Schichten Architektur verwendet, wo der Client, der Server und die Datenbank autonome Einheiten sind. Der Hauptvorteil liegt darin, dass jeder Bereich von einem separaten Entwicklungsteam entwickelt wird.
- Microservices - Zerlegt die Anwendung in kleine, unabhängige Dienste, die über APIs kommunizieren. Jedes Microservice implementiert genau eine Funktion. Es ist möglich, dass jedes Microservice eine andere Programmiersprache und Datenbank nutzt. Sie können unabhängig voneinander entwickelt werden, wodurch die Entwicklerteams ebenfalls unabhängig arbeiten können.



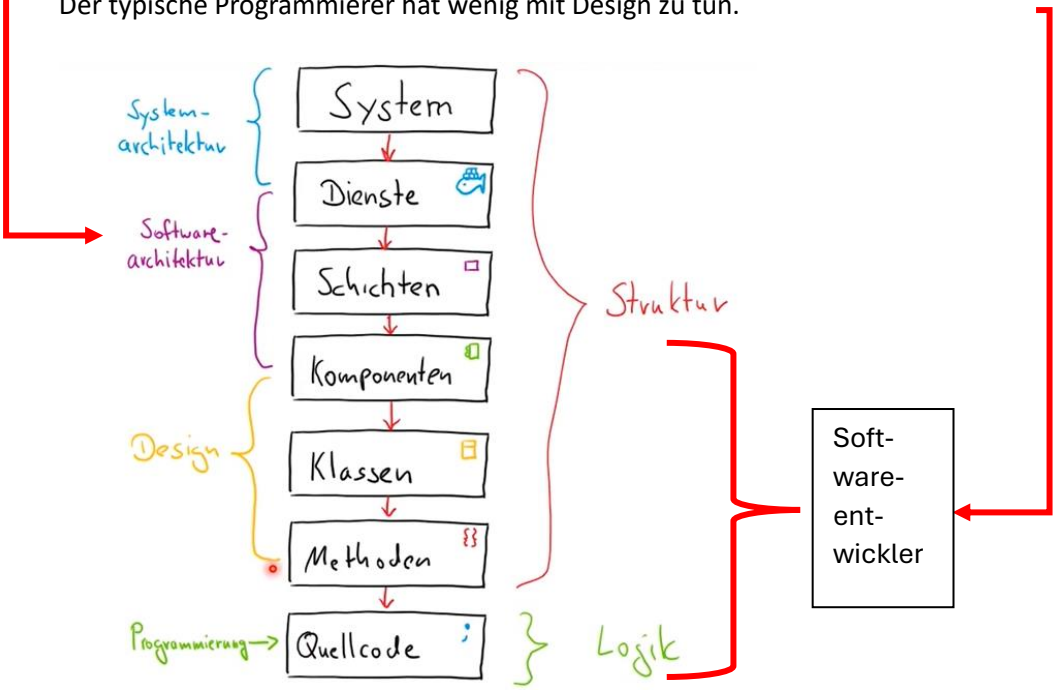
- Model-View-Controller (MVC)



- Serverless-Architektur – heißt nicht, dass kein Server benutzt wird. Der Fokus liegt auf dem Schreiben von Quellcode, nicht auf Infrastrukturaspekte.

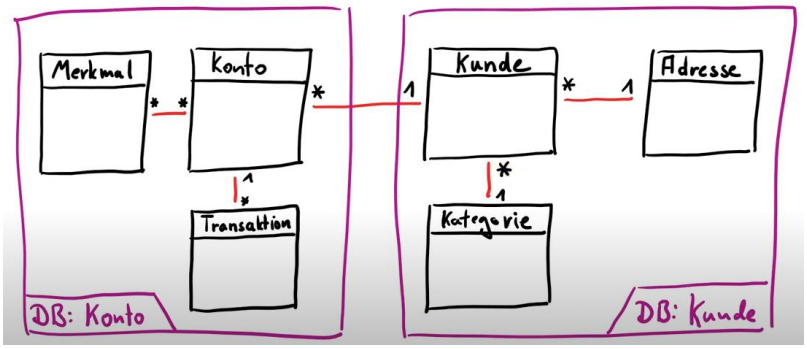
Der Software-Architekt kommuniziert mit dem System-Architekt und dem Software-Entwickler. Er kümmert sich z.B. um die Wiederverwendbarkeit von Komponenten und die Austauschbarkeit von Komponenten (Beispiel: für eine bestimmte Zeit wird der Umsatzsteuersatz reduziert).

Der typische Software-Entwickler deckt die Bereiche Design und Programmierung ab. Der typische Programmierer hat wenig mit Design zu tun.



Beispiel:
Anlegen (Onboarding) eines neuen Kunden bei einer Bank und gleichzeitig Anlegen eines neuen Girokontos. Ein Teil ist der Domain-Bereich (DB) Kunde, der andere Domain-Bereich der vom Konto. Domains sind strikt voneinander getrennt.

- In der DB-Kunde kann man z.B. neue Kunden anlegen, löschen, bearbeiten.
- Kategorie: Firmenkunde, Neukunde, Bestandskunde…
- Merkmale: überzogen, Kreditlimit
- Transaktionen: Eingang, Abgang-Auszahlung



Domains sind eigentlich die Kernbereiche eines Unternehmens oder einer Software, wie Kunde, Konto, Lager, Shop, Versand…

Es kann z.B. folgende Domain-Komponenten geben: Kredite, Kreditkarte, Kunden, Sicherheiten, Buchhaltung usw.

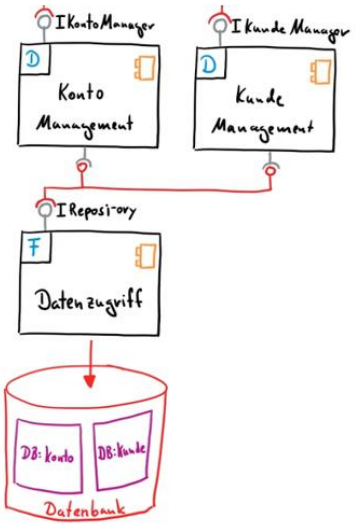
Eine Domain kann dann für andere Projekte oder woanders in einem großen Projekt **wiederverwendet** werden. Das kann dann als Service bzw. Microservice dort wiederverwendet und eingebaut werden.

anderes Beispiel: amazon
Betrifft viele Domains, nämlich

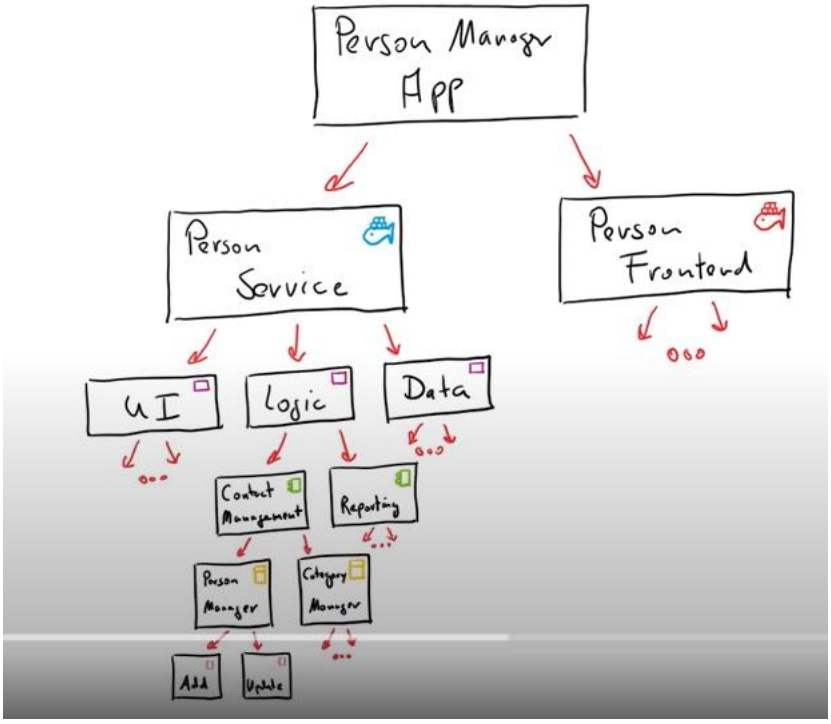
- Lagerverwaltung
- Buchhaltung
- Shopsystem
- Versand

https://www.youtube.com/watch?v=DCdu9IM2gVk

Einer der ersten Tätigkeiten ist die Erstellung des Datenzugriffs. Dabei dient der „Datenzugriff“ als Schnittstelle. Damit ist die Datenbank angeschlossen, wäre aber separat und ist austauschbar.



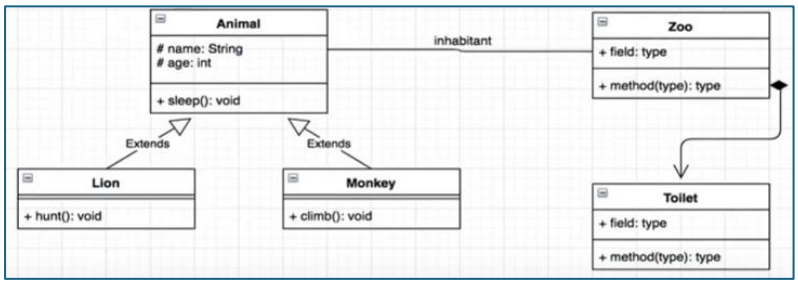
Beispiel: Modularisierung



Style einer Architektur: 3 Schichtenarchitektur

1. Schicht: Button (UI),
2. Schicht: Logik - was passiert, wenn man den Button drückt? (User, Domain),
3. Schicht: Infrastruktur (z.B. Datenbank Zugriff)

d. Ansatz, um das Projekt abzubilden (Systemmodellierung)
Visualisierung – visuelle Darstellung
UML: unified modeling language
Standardisierung von Diagrammen bei Modellierung in UML, Klassendiagramm (object model, Vererbung, Datenbeziehungen)



- UML unterstützt bei der Planung und Gestaltung der Systemarchitektur, was zu einer besseren Struktur und Organisation führt.
- Es fördert die Zusammenarbeit zwischen verschiedenen Teams, da es eine gemeinsame Sprache für Entwickler, Designer und Stakeholder bietet.

Es gibt auch einfachere Varianten stattUML.
Kostenlose Modellierungssoftware: https://www.draw.io
https://www.eraser.io/

Wichtige Aufgaben für ein erfolgreiches Erstellen einer Architektur:

- Schreibe ein Design-Dokument als Plan, Welche Architekturstil? Z.B. monolith; Zeit, wann fertig? Welche Programmiersprache/Referenzen nutzt man? Z.B. vue.js, node.js und MongoDB; Wer sind die Stakeholder?
- Lerne ein Diagramm zu erstellen – sind eine universelle Sprache und helfen das komplexe System zu visualisieren, wie die verschiedenen Teile interagieren.

  https://www.youtube.com/watch?v=k3hKLd7vYZ8

Beispiel für ein leeres Template mit gängigen Abschnitten:

https://www.mimacom.com/de/ressourcen/software-architektur-template

4. **Entwicklung, Implementierung des Codes**
   Ab hier wird Code geschrieben.
   Ausgangspunkt sind die User-Stories aus der Anforderungsanalyse und das Ergebnis (der Entwurf) aus der Architekturphase. Daraus wird mit Berücksichtigung der entworfenen Architektur der Code geschrieben.1
   Dabei kommen verschiedene Programmiersprachen und Plattformen, sowie Tools, Technologien und Frameworks zusammen. Das Ergebnis ist der Quellcode.

5. **Testen**
   „Testen zeigt das Vorhandensein von Bugs, nicht ihre Absenz."
   a. Fehler erkennen: Nach dem Zusammensetzen der Software, eventuell von verschiedenen Programmierern, könnten Fehler auftreten.
      Anforderungsfehler, Entwurfsfehler, Implementierungsfehler (Bug im Code).
   b. Verifizieren heißt „Korrektheit beweisen" der User-Stories und der Architektur, ob diese erfüllt wurden, z.B. werden die Use-Cases auf der Oberfläche durchprobiert auf Usability, Benutzbarkeit usw.
      Beispiel: in einem String kommen Umlaute vor und verursachen Probleme (utf8).

   Es werden verschiedene Arten von Tests durchgeführt:

   - Unittest – strikt lokal für nur 1 Klasse, vermeiden Infrastruktur, damit sie sehr schnell sind und dauernd in der Entwicklung laufen können.
   - Integrationstest - testen ob verschiedene Module korrekt zusammenarbeiten, z.B. ob API funktioniert. Datenbanken werden mitgetestet.
   - Systemtest (echte Produktionsumgebung, aber mit Testdaten, nicht mit den echten; ob System funktionsfähig ist, nicht localhost)
   - Abnahmetest.
   - Für die nicht-funktionalen Anforderungen werden z.B. Lasttests, für die Benutzbarkeit werden Usability-Tests durchgeführt. Das Budget begrenzt hier die Vielfältigkeit der

Tests, da diese einiges an Geld kosten können.
https://www.youtube.com/watch?v=z0r5XqPk8jA automatisierte Tests

https://www.youtube.com/watch?v=4h0yYXPtapo Unit testing

6. **Bereitstellung, Abnahme und Einführung (deployment)**
   hier wird mit einem Build-Server“ die ausführbare Anwendung erstellt und danach für den Endnutzer bereitgestellt. Die Bereitstellung kann z.B. mittels FTP auf einen Server geladen werden oder in der Cloud bereitgestellt.

7. **Betrieb, Wartung**
   Die Ausgestaltung hängt davon ab, was für eine Anwendung entwickelt wurde, also eine Desktop-, Web-, Mobile-, IoT- oder Cloudanwendung. Meist gehören hier neben der Nutzung auch die Überwachung und Auswertung dazu.
   Sind die Server zu schwach?

Quellen:

https://www.youtube.com/watch?v=CinPOlgq0kQ – David Tielke
https://www.youtube.com/@DavidTielke

https://www.youtube.com/watch?v=Sl3zCFveUa4&list=PL_pqkvxZ6ho05rbgNaakWmxFmT9qEXzIo

https://programmieren-starten.de/

https://www.youtube.com/watch?v=OIafuLDsXgg&t=183s Architekturmuster

https://www.youtube.com/watch?v=DCdu9IM2gVk