

Inhaltsverzeichnis

Anforderungsanalyse	2
Use Case Diagram/ Use Case Table	2
Ist- und Soll-Analyse	3
INVEST- Kriterien (Qualitätskriterien einer User Story)	4
Funktionale und Nichtfunktionale Anforderungen	5
Lastenheft und Pflichtenheft	5
Verifikation und Validierung	5
Entwurf	6
Komponentendiagramm	6
Grobentwurf vs. Feinentwurf	6
Entwurfs- und Architekturmuster (Patterns)	6
Singleton Pattern	7
Composite Pattern	7
State Pattern	7
Drei-Schichten-Architektur	7
Pipe-Filter-Architektur	8
Microservices-Architektur	8
Domain Driven Design	9
Behaviour Driven Development	10
Test Driven Development	10
Akzeptanz	11
SOLID Principles von Robert C.Martin	11
Implementierung und Test	12
Git	12
DevOps	13
Continuous Deployment Patterns	14
Kontrollflussgraphen	15
Prüfresultate beim Testen	15
McCabes zyklomatische Komplexität	15
Projekt- und Produktmanagement	16
Vorgehensmodelle	16
Wasserfallmodell	16
iterative Entwicklung	16
inkrementelle Entwicklung	16
Scrum	17
Organisationsformen für Teams	18
Unternehmensorganisation	19
Begriffe im Software Engineering	19
Prinzipien	19
„Programming in the Large“ vs. „Programming in the Small“	20
Safety vs. Security	20
Stakeholder im Software Engineering	20
Qualitätenbaum	21
Die QS-Ingenieur*in	21
Software-Review	21

Prüfungsfragen	22
1. Aspekte, die ein Entwurf gegenüber den Anforderungen zusätzlich enthalten sollte.	22
2. Software ist immateriell. Erkläre	22
3. Prozessqualität vs. Produktqualität	22
4. 2 Analysetechniken, die sich besonders gut zur Feststellung des Ist-Zustandes eignen	22
5. Was ist Software Alterung?	22
6. Nenne ein Dokumententyp, der nicht geeignet ist durch ein Review geprüft zu werden	22
7. Erkläre die Make or Buy-Entscheidung und Argumente für und gegen die Neuentwicklung	22
8. Fehlhandlung(Mistake) und Ausfall(Failure)/ Fehlerursache(Fault) und Fehlerzustand(Error)	23
9. Alles zu Risikomanagement	23
10. Was ist ein Arbeitspaket, woraus besteht es (durch welche Informationen ist es definiert)?	23
11. Nenne 3 Beispiele für nichtfunktionale Anforderungen	24
12. Aufwandsschätzung	24
13. Beschreiben Sie, was Planning Poker ist.	24
14. Fasse Prototyping zusammen.	24
15. Ebenen beim Entwurf (einer Software)	24
16. Durch was wird der Nutzen einer Software beeinflusst?	24
17. Use Cases vs. User Stories	24
18. Was ist die Rolle des Software Architekten?	25
19. Software-Krise und Ursachen	25
20. Der Begriff der Qualitätssicherung war in drei Teilbegriffe(Schwerpunkte) zerlegt worden. Welche?	25
21. Welche Nachteile hat es, im Test Fehler zu beheben sobald sie aufgetreten sind?	25
22. Welche Voraussetzungen müssen erfüllt sein damit mehrere Personen ein demokratisches Team bilden können?	26
23. Was versteht man unter einem quantifizierten Qualitätsmodell, und wie kann man es nutzen?	26
24. Warum sind Meilensteine in der Software-Entwicklung wichtig?	26
25. Geben Sie zwei Gründe an, warum Anforderungen schriftlich festgehalten werden sollten.	26
26. Entwicklungs- und Evolutionsmodell für Mitarbeiter	26

Anforderungsanalyse

a) Use Case Diagram/ Use Case Table

Mit dem Use Case Diagramm werden von außen sichtbare Interaktionen von Akteuren visualisiert. Damit beschreibt es in einer hohen Abstraktion, welche Funktionen und Dienste ein System für einen Anwender bereitstellt.

Ein Use Case Table (Anwendungsfalltabelle) ist ein Werkzeug, das in der Softwareentwicklung und im Projektmanagement verwendet wird, um verschiedene Anwendungsfälle eines Systems zu dokumentieren. Diese Tabelle hilft dabei, die Anforderungen und Interaktionen zwischen den Benutzern (Akteuren) und dem System zu verstehen und zu spezifizieren.

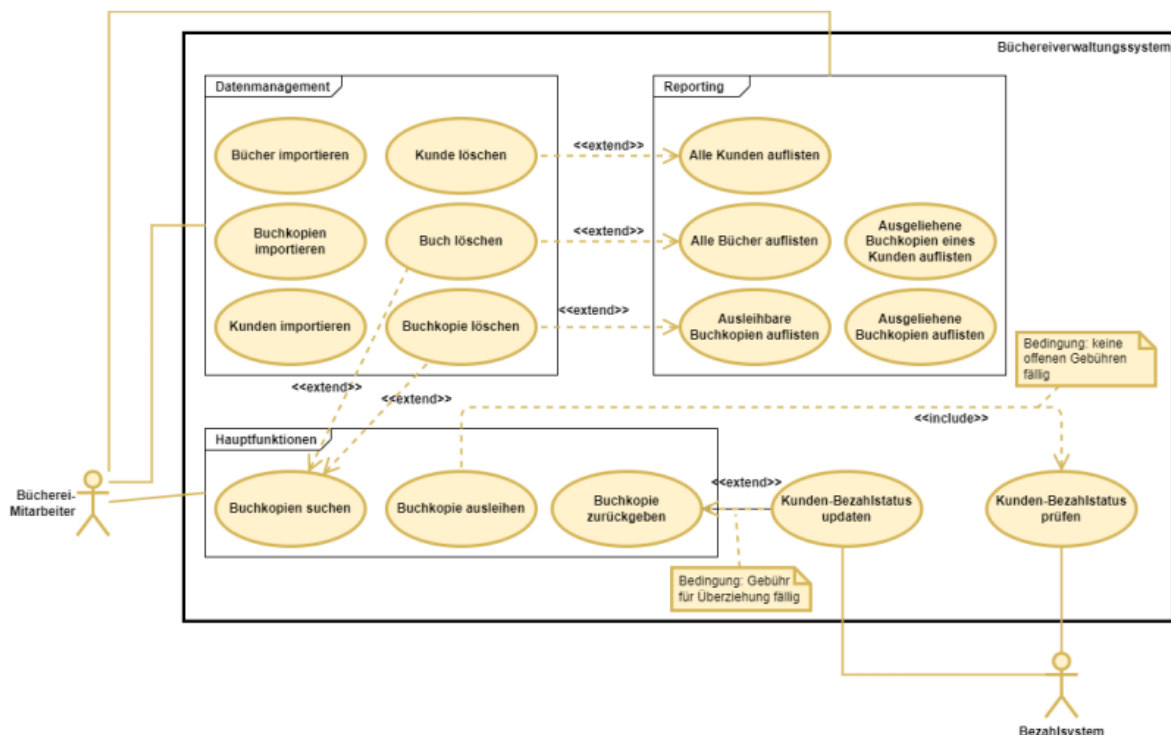


Abbildung 2: Use Case Diagramm des BVS

Die «include»-Beziehung zeigt an, dass ein Anwendungsfall den Ablauf eines anderen Anwendungsfalls einschließt. Das bedeutet, dass der inkludierte Anwendungsfall immer dann ausgeführt wird, wenn der Hauptanwendungsfall ausgeführt wird.

Syntax: Ein gestrichelter Pfeil mit einem offenen Pfeilkopf, der von dem Hauptanwendungsfall zum inkludierten Anwendungsfall zeigt, mit der Beschriftung «include».

Die «extend»-Beziehung zeigt an, dass ein Anwendungsfall den Ablauf eines anderen Anwendungsfalls erweitert. Dies bedeutet, dass der erweiternde Anwendungsfall optional ist und nur unter bestimmten Bedingungen ausgeführt wird.

Syntax: Ein gestrichelter Pfeil mit einem offenen Pfeilkopf, der von dem erweiternden Anwendungsfall zum erweiterten Anwendungsfall zeigt, mit der Beschriftung «extend».

User Stories

Story Name	ID:	357
Authentifizieren		
As a	Kunde	
I want to	meine Bankkarte einführen und meinen PIN eingeben	
so that	ich Zugriff auf mein Konto habe	
Estimate		Actual
3		4

Hierbei bezeichnet 'Estimate' die Aufwandsabschätzung.

Tabelle 1: Use Case Tabelle für die Ausleihe einer Buchkopie

Name	Ausleihe einer Buchkopie
Ziel	Gewählte Buchkopie ist für den Kunden als ausgeliehen eingetragen
Akteure	Büchereimitarbeiter
Vorbedingung	<ul style="list-style-type: none"> • System ist im Hauptmenü gestartet • Kunden-ID und Buchkopie-ID sind bekannt
Nachbedingung	<ul style="list-style-type: none"> • Buchkopie ist mit Datum als ausgeliehen markiert und dem Kunden zugeordnet • Erfolgsmeldung wird angezeigt • Hauptmenü wird angezeigt
Normalablauf	<ol style="list-style-type: none"> 1. Auswahl des Menüpunkts „Ausleihe“ 2. Eingabe von Kunden-ID und Buchkopie-ID 3. Prüfung, ob alle Kriterien erfüllt sind (keine offenen Gebühren, nicht mehr als 5 Buchkopien, nicht mehrere Kopien des gleichen Buchs) 4. Setzen des Ausleihstatus und -datums für Buchkopie und Zuweisung zum Kunden 5. Erfolgsmeldung anzeigen 6. Navigation zum Hauptmenü
Sonderfall 3a	Kunde erfüllt Kriterien nicht
Nachbedingung im Sonderfall 3a	<ul style="list-style-type: none"> • Buchkopie ist weiterhin verfügbar • Fehlermeldung wird angezeigt
Ablauf im Sonderfall 3a	<ol style="list-style-type: none"> 3a.1. Anzeige des genauen Fehlergrunds 3a.2. Weiter mit Punkt 2

Ist- und Soll-Analyse

Eine Soll-Ist-Analyse in der Softwareentwicklung ist eine Methode zur Identifikation und Analyse von Unterschieden zwischen dem aktuellen Zustand eines Systems oder Prozesses (Ist-Zustand, bspw. Auswertung vorhandener Daten, Dokumente) und dem gewünschten oder geplanten Zustand (Soll-Zustand, bspw. Prototyping). Diese Analyse wird oft eingesetzt, um Verbesserungsmöglichkeiten zu identifizieren, Abweichungen zu verstehen und entsprechende Maßnahmen zur Anpassung oder Optimierung zu entwickeln.

b) INVEST- Kriterien (Qualitätskriterien einer User Story)

Independent: Jede User Story sollte unabhängig von anderen sein

Negotiable: Storys müssen verhandelbar sein und Details müssen während der Entwicklung anpassbar sein

Valuable: stellt sicher, dass jede Story einen Wert für den Kunden hat, sollte Mehrwert für Nutzer und Produkt darstellen (auch zukünftig)

Estimable: Man sollte in der Lage sein die Arbeit abzuschätzen (sonst split/slicing)

Sized Appropriately (Small): Story sollte so klein wie möglich sein

Testable: Story sollte testbar sein (Akzeptanzkriterien)

Anzeigen der Buchausleih-Historie eines Kunden

Independent (Unabhängig):

- **Bewertung:** Erfüllt
- **Begründung:** Diese Anforderung kann eigenständig umgesetzt werden, ohne dass sie von anderen Anforderungen direkt abhängig ist. Sie bezieht sich ausschließlich auf die Anzeige von Daten, die bereits im System vorhanden sind, und benötigt keine gleichzeitige Implementierung anderer Features.

Negotiable (Verhandelbar):

- **Bewertung:** Erfüllt
- **Begründung:** Die genaue Darstellung und Funktionalität der Buchausleih-Historie kann diskutiert und angepasst werden, z.B. welche Details angezeigt werden oder wie die Historie sortiert und gefiltert wird. Es gibt Spielraum für Verhandlungen über die konkreten Anforderungen und Details der Umsetzung.

Valuable (Wertvoll):

- **Bewertung:** Erfüllt
- **Begründung:** Diese Funktion bietet einen klaren Mehrwert für die Bibliothek und ihre Mitarbeiter, da sie einen schnellen Überblick über die bisherigen Ausleihen eines Kunden ermöglicht. Dies kann hilfreich sein, um Kundenfragen zu beantworten und den Ausleihprozess zu verwalten.

Estimable (Schätzbar):

- **Bewertung:** Erfüllt
- **Begründung:** Der Aufwand für die Umsetzung dieser Anforderung kann realistisch geschätzt werden. Es ist klar definiert, welche Daten angezeigt werden sollen, und die technische Umsetzung (Datenbankabfragen, Benutzeroberfläche) ist gut abschätzbar.

Small (Klein):

- **Bewertung:** Erfüllt
- **Begründung:** Die Anforderung ist klein und überschaubar genug, um in einem kurzen Zeitraum umgesetzt zu werden. Sie umfasst im Wesentlichen die Entwicklung einer Abfrage und einer Darstellung in der Benutzeroberfläche, was in einem Sprint realisiert werden kann.

Testable (Testbar):

- **Bewertung:** Erfüllt
- **Begründung:** Die Anforderung ist testbar, da klare Akzeptanzkriterien definiert werden können. Es kann überprüft werden, ob die Historie korrekt angezeigt wird und ob alle relevanten Daten (wie Ausleihdaten, Buchdetails) vollständig und korrekt dargestellt werden.

(6 Punkte) Bewerten Sie anhand der INVEST-Kriterien (Wake, 2003) mit einer kurzen Begründung pro Kriterium die folgende Anforderung:

Import und Export der „Mensch ärgere Dich nicht“-Spielhistorie als Text-Datei

Solution:

- **Independent:** Eher nicht erfüllt, weil eine Abhängigkeit zur Spielfunktionalität und dem Anlegen der Historie besteht
- **Negotiable:** Erfüllt, weil die Details des Exports noch festgelegt werden können (z.B. welche Attribute oder ob CSV oder JSON als Format)
- **Valuable:** Eher nicht erfüllt, weil es für die Endnutzer (Spieler) wenig Mehrwert liefert (eher für die Entwickler)
- **Estimable:** Erfüllt, weil der Umfang sehr klar ist und es wenig unbekannte Einflüsse gibt
- **Sized Appropriately:** Erfüllt, weil die Größe angemessen für ein Arbeitspaket ist
- **Testable:** Erfüllt, weil man klare Akzeptanzkriterien festlegen kann und den erfolgreichen Import/Export gut prüfen kann

c) Funktionale und Nichtfunktionale Anforderungen

Funktionale Anforderungen(Problemorientiert, Abstrakt) beschreiben die spezifischen Funktionen und Verhaltensweisen, die ein System ausführen soll. Sie definieren, was das System tun soll und wie es auf bestimmte Eingaben reagieren soll. Funktionale Anforderungen beziehen sich auf konkrete Features, Aufgaben oder Interaktionen des Systems mit Benutzern oder anderen Systemen.

Nichtfunktionale Anforderungen(Konkret, Präzise) beschreiben die Qualitätseigenschaften und die allgemeinen Bedingungen, unter denen das System arbeitet. Sie definieren, wie das System seine Funktionen ausführen soll und beziehen sich oft auf Leistungsmerkmale, Sicherheitsanforderungen, Wartbarkeit, Skalierbarkeit und Benutzerfreundlichkeit.

Beispiele des BVS:

Funktionale Anforderungen:

1. Importieren einer CSV-Datei mit Büchern: • via absoluten oder relativen Pfad zu einem CSV-File • Inhalt wird gelesen und ins System eingespeist (eine Zeile entspricht einer Buchkopie) • Struktur: isbn;title;authors;year;city;publisher;edition

Nicht-Funktionale Anforderungen:

Wichtige nicht-funktionale Anforderungen sind zum einen die Verwendung der Programmiersprache Java und die Kompatibilität mit Java 17 SE. Weiterhin sind Effizienz und Benutzerfreundlichkeit der Kommandozeilensteuerung sehr wichtig. Die Menüanordnung und -auswahl sollen auf die wichtigsten Prozesse (8, 9, 7) ausgerichtet sein, sodass diese möglichst schnell erreichbar sind.

d) Lastenheft und Pflichtenheft

Das Lastenheft ist ein Dokument, das vom Auftraggeber erstellt wird und beschreibt, was das System leisten soll. Es enthält die Anforderungen und Wünsche des Auftraggebers an das System.

Das Pflichtenheft ist ein Dokument, das vom Auftragnehmer erstellt wird und beschreibt, wie die im Lastenheft definierten Anforderungen technisch umgesetzt werden sollen.

Zweck:

Lastenheft: Beschreibt die Anforderungen und Wünsche des Auftraggebers an das System (Was soll das System leisten?).

Pflichtenheft: Beschreibt die technische Umsetzung der Anforderungen des Auftraggebers (Wie sollen die Anforderungen umgesetzt werden?).

Inhaltliche Tiefe:

Lastenheft: Enthält allgemeine Anforderungen und Rahmenbedingungen.

Pflichtenheft: Enthält detaillierte technische Spezifikationen und Implementierungspläne.

e) Verifikation und Validierung

Verifikation bezieht sich auf den Prozess der Überprüfung, ob die Software gemäß den Spezifikationen und Entwürfen korrekt implementiert wurde. Es geht darum, sicherzustellen, dass das Produkt "richtig gebaut" wurde. Verifikation wird oft durch verschiedene technische Aktivitäten wie Code-Inspektionen, Reviews, statische Analysen und Unit-Tests durchgeführt.

Frage: Bauen wir das Produkt richtig?

Validierung bezieht sich auf den Prozess der Überprüfung, ob die Software die tatsächlichen Bedürfnisse und Erwartungen der Benutzer erfüllt. Es geht darum, sicherzustellen, dass das Produkt "das richtige Produkt" ist. Validierung wird oft durch Aktivitäten wie Systemtests, Akzeptanztests und Benutzerfeedback durchgeführt.

Frage: Bauen wir das richtige Produkt?

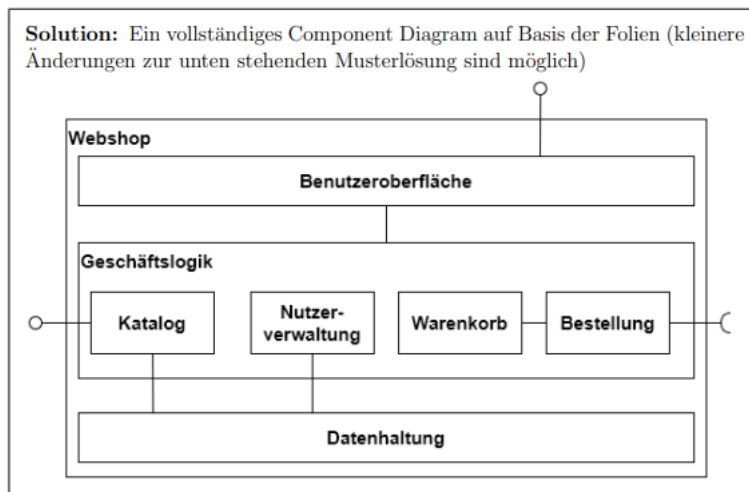
Entwurf

a) Komponentendiagramm

Eine **Komponente** ist ein ersetzbarer und modularer Bestandteil eines Systems. Eine Komponente ist eine Einheit, die eine bestimmte Funktionalität kapselt und in einem größeren System eingebaut werden kann. Sie kann aus einer Sammlung von Klassen bestehen, muss es aber nicht.

Eine **Schnittstelle** ist eine Methode oder ein Satz von Methoden, die von einer anderen Komponente verwendet werden. Eine bestimmte Komponente kann anderen Komponenten Schnittstellen zur Verfügung stellen und/oder Schnittstellen von anderen Komponenten benötigen.

- (a) (8 Punkte) Zeichnen Sie ein hierarchisches Komponentendiagramm für die Architektur des beschriebenen Webshops (*Component Diagram*). Verwenden Sie dabei die drei Schichten (*Layers*) „Benutzeroberfläche“, „Geschäftslogik“ (mit internen Komponenten) und „Datenhaltung“. Zeichnen Sie auch existierende Schnittstellenverbindungen (*Interfaces*) ein, sowohl zwischen internen Komponenten als auch von internen Komponenten nach extern. Externe Akteure / Systeme müssen NICHT eingezeichnet werden.



b) Grobentwurf vs. Feinentwurf

Detailtiefe:

Grobentwurf: Behandelt die Systemarchitektur und die Hauptkomponenten auf einer hohen Abstraktionsebene. Er legt die Struktur und die wesentlichen Interaktionen fest, ohne in die Details der Implementierung zu gehen.

Feinentwurf: Behandelt die Details der einzelnen Komponenten und deren Implementierung. Er liefert konkrete Anweisungen für die Programmierung und beschreibt die spezifischen Designaspekte.

Zielsetzung:

Grobentwurf: Ziel ist es, eine klare, strukturierte Übersicht über das System zu bieten und die grundlegenden Designentscheidungen zu treffen.

Feinentwurf: Ziel ist es, die genaue technische Umsetzung der Komponenten zu planen und die Implementierung vorzubereiten.

Ergebnisse:

Grobentwurf: Architekturdiagramme und grundlegende Designentscheidungen.

Feinentwurf: Detaillierte Diagramme, Spezifikationen und Anweisungen für die Implementierung.

c) Entwurfs- und Architekturmuster (Patterns)

Singleton Pattern

- Es soll sichergestellt werden, dass es zu einer Klasse höchstens eine Instanz gibt
- Diese Instanz soll global verfügbar sein, d.h. es muss eine Möglichkeit für alle Objekte geben, diese Instanz anzusprechen.

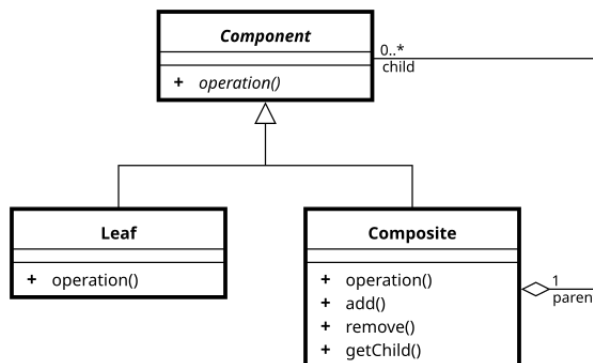
```

1 public class Singleton {
2     private static Singleton instance;
3     private Singleton() { // privater Konstruktor verhindert Instanzierung}
4
5     public static synchronized Singleton getInstance() {
6         if (instance == null) {instance = new Singleton();}
7         return instance;
8     }
9 }

```

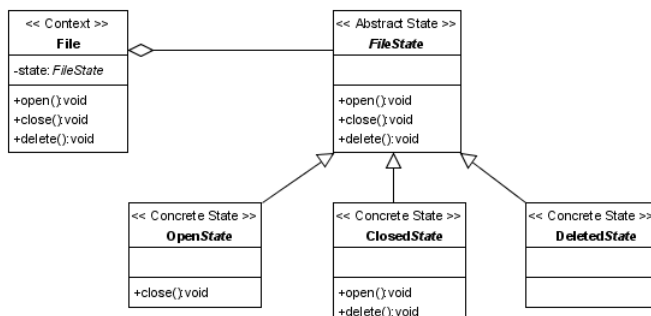
Composite Pattern

- Designmuster, mit dem man Objekte in baumartige Strukturen zusammensetzen kann, um Teil-Ganzes-Hierarchien darzustellen. Es ermöglicht Clients, einzelne Objekte und Objektkompositionen einheitlich zu behandeln. Mit anderen Worten: Unabhängig davon, ob es sich um ein einzelnes Objekt oder eine Gruppe von Objekten (Composite) handelt, können Clients sie austauschbar verwenden.
- Es wird verwendet, um Baumstrukturen darzustellen und die Arbeit mit Einzelobjekten und zusammengesetzten Objekten einheitlich zu gestalten.



State Pattern

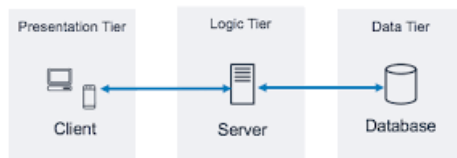
- Ein Objekt soll sich abhängig von seinem Zustand verschieden verhalten.
- Der Zustand wird als Klasse gekapselt. Spezielle Zustandsklassen mit überladenen Methoden implementieren die jeweilige Zustandsabhängigkeit.



Drei-Schichten-Architektur

Sie teilt die Anwendung in drei logische Schichten auf, um die Komplexität zu reduzieren und die Wartbarkeit zu verbessern. Diese Schichten sind:

- Präsentationsschicht (client tier, Front-End): Ist für die Präsentation und die Eingabe der Daten (die Benutzerschnittstelle) verantwortlich.
- Logikschicht (application-server tier): Verarbeitung der Daten. Hier ist die Anwendungslogik vereint.
- Datenhaltungsschicht (data-server tier, back end): Sie enthält die Datenbank und ist verantwortlich für das Speichern und Laden von Daten.



Pipe-Filter-Architektur

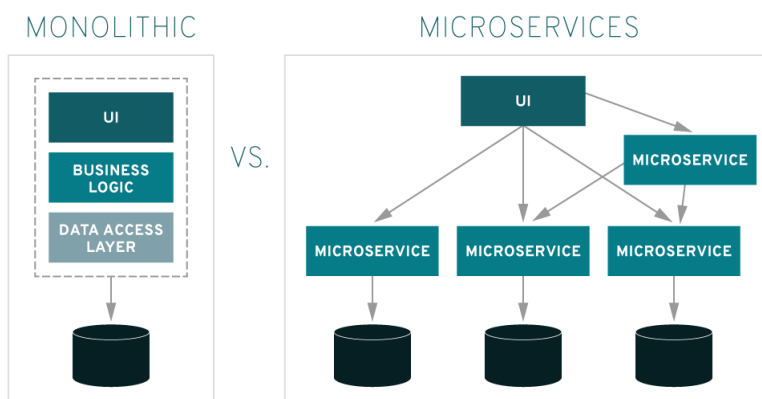
Das Pipe-Filter-Architekturmuster dient dazu, eine Anwendung zu strukturieren, die Daten auf einem virtuellen Fließband verarbeitet.

- Die Verarbeitungsschritte werden in den sogenannten Filtern realisiert. Ein Filter verbraucht und erzeugt Daten.
- Pipes leiten die Ergebnisse des Filters an nachfolgende Filter weiter.
- Das erste Filter bekommt seine Daten aus der Datenquelle, das letzte liefert sie an die Datensenke.



Microservices-Architektur

Die Microservices-Architektur ist ein Ansatz in der Softwareentwicklung, bei dem eine Anwendung als eine Sammlung von kleinen, unabhängigen Diensten oder Microservices entwickelt wird. Jeder dieser Dienste ist für eine spezifische Geschäftsaufgabe verantwortlich und kann unabhängig voneinander entwickelt, bereitgestellt und skaliert werden. Diese Architektur hat sich als Alternative zur monolithischen Architektur etabliert, insbesondere bei komplexen und großen Systemen.



d) Domain Driven Design

Domain-Driven Design (DDD) ist ein Ansatz zur Softwareentwicklung, der den Fokus auf die Modellierung der Domäne (das spezifische Fachgebiet, für das die Software entwickelt wird) legt. DDD wurde von Eric Evans in seinem Buch Domain-Driven Design: Tackling Complexity in the Heart of Software eingeführt. Die Geschäftslogik, Anforderungen und der Bereich, den das System adressiert stehen im Mittelpunkt der Softwareentwicklung. Entwickler und Fachexperten arbeiten eng zusammen, um ein tiefes Verständnis der Domäne zu entwickeln.

Kernkonzepte:

- 1) Ubiquitous Language (Allgegenwärtige Sprache)
Eine gemeinsame Sprache, die von allen Beteiligten (Entwicklern und Fachexperten) verwendet wird. Diese Sprache umfasst alle relevanten Begriffe und Konzepte der Domäne und wird in allen Kommunikationsformen und im Code konsequent verwendet.
- 2) Bounded Context (Begrenzter Kontext)
Ein Bounded Context definiert klare Grenzen für ein bestimmtes Domänenmodell und die zugehörige Ubiquitous Language. Innerhalb eines Bounded Contexts sind die Begriffe und Regeln einheitlich und klar definiert. Dies hilft, die Komplexität großer Systeme durch klare Abgrenzungen zu reduzieren und Missverständnisse zu vermeiden.
- 3) Entity
Eine Entity ist ein Objekt, das durch seine Identität definiert ist. Es hat eine eigenständige Lebensdauer und kann sich im Laufe der Zeit ändern, behält aber seine Identität bei. Beispiele sind Kunden, Bestellungen oder Produkte.
- 4) Value Objects
Objekte, die durch ihre Eigenschaften definiert sind und keine eigene Identität besitzen. Beispielsweise könnte eine Adresse ein Value Object sein, da sie nur durch die Kombination ihrer Felder (Straße, Stadt, Postleitzahl) definiert ist.
- 5) Aggregate
Ein Aggregate ist eine Gruppe von zusammengehörigen Objekten (Entities und Value Objects), die als Einheit behandelt werden. Es hat eine Wurzel-Entity (Aggregate Root), die die Konsistenz und Integrität des gesamten Aggregates gewährleistet.
- 6) Repository
Ein Repository ist eine Abstraktion für die Speicherung und das Abrufen von Aggregates. Es bietet eine Sammlung ähnlicher Objekte und kapselt die Implementierung der Datenzugriffslogik.
- 7) Service
Ein Service ist eine Operation oder eine Gruppe von Operationen, die nicht natürlich zu einer Entity oder einem Value Object gehören. Services sind oft zustandslos und bieten domänenspezifische Funktionalitäten.
- 8) Domain Event
Ein Domain Event signalisiert, dass innerhalb der Domäne etwas Bedeutendes passiert ist. Es ermöglicht die Kommunikation zwischen verschiedenen Teilen des Systems und kann verwendet werden, um Ereignisse über Kontextgrenzen hinweg zu verbreiten.
- 9) Commands
Im zweiten Schritt fügen wir den Domain Events Commands hinzu. Diese lösen die Domain Events aus und sind immer einem Domain Event zugeordnet. Sie sind als Verben formuliert. Manche Domain Events haben auch keine auslösenden Commands, sondern werden bspw. durch einen Timer ausgelöst. Während diesem Hinzufügen der Commands können auch neue Domain Events hinzugefügt werden

Vorteile von Domain-Driven Design (DDD): erleichtert eine präzise Abbildung der Geschäftsanforderungen in der Software, verbessert die Kommunikation zwischen Entwicklern und Fachexperten durch eine einheitliche Sprache und fördert Flexibilität und Anpassungsfähigkeit. Durch klar definierte Grenzen und Verantwortlichkeiten werden komplexe Systeme besser beherrschbar.

Nachteile von Domain-Driven Design (DDD): Implementierung zeitaufwendig und komplex, erfordert eine intensive Einarbeitung und eine kontinuierliche enge Zusammenarbeit zwischen den beteiligten Teams.

e) Behaviour Driven Development

Behavior Driven Development (BDD), auch bekannt als verhaltensgetriebene Softwareentwicklung, ist eine agile Entwicklungstechnik, die die Zusammenarbeit zwischen Qualitätsmanagement und Business-Analyse stärkt. Durch BDD werden die Anforderungen, Ziele und Ergebnisse der Software in einer spezifischen Textform festgehalten, die später als automatisierte Tests ausgeführt werden können. Diese Tests dienen dazu, die korrekte Implementierung der Software zu überprüfen. Die Softwareanforderungen werden oft in Form von Wenn-Dann-Sätzen verfasst, die auf der Sprache des Domain-driven Designs basieren. Das Ziel ist es, eine Form dafür zu wählen, die auch von Nicht-Softwareexperten verstanden werden kann. Dies erleichtert den Übergang zwischen der Sprache der fachlichen Anforderungen und der Programmiersprache, die zur Umsetzung verwendet wird. Insgesamt zielt BDD darauf ab, die Qualität der Software zu verbessern, indem es eine klare Kommunikation zwischen den verschiedenen Stakeholdern ermöglicht und die Softwareentwicklung auf die Bedürfnisse der Benutzer ausrichtet.

Vorteile von Behavior Driven Development (BDD): BDD fördert eine enge Zusammenarbeit zwischen Qualitätsmanagement und Business-Analyse, verbessert die Kommunikation zwischen den Stakeholdern und ermöglicht eine klare Definition der Softwareanforderungen in Form von automatisierten Tests. Durch die Verwendung von "Wenn-Dann-Sätzen in der Sprache des Domain-driven Designs wird der Übergang von fachlichen Anforderungen zur Implementierung erleichtert.

Nachteile von Behavior Driven Development (BDD): Die Einführung von BDD erfordert Zeit und Schulungsaufwand, um die beteiligten Personen mit der neuen Vorgehensweise vertraut zu machen. Zudem kann die Erstellung und Wartung umfangreicher Testfälle zeitaufwändig sein und die Entwicklung verlangsamen.

f) Test Driven Development

Test Driven Development (TDD) ist eine agile Softwareentwicklungsmethode, bei der Tests vor der Implementierung des eigentlichen Codes geschrieben werden.

Kernkonzepte:

1) Red-Green-Refactor:

Dies ist der zyklische Prozess von TDD. Zuerst schreibt man einen Test (Red), dann implementiert man den minimalen Code, um den Test zum Bestehen zu bringen (Green) und schließlich wird der Code verbessert und vereinfacht (Refactor).

2) Regressionstests:

Die bereits geschriebenen Tests werden regelmäßig ausgeführt, um sicherzustellen, dass neu hinzugefügter Code keine bestehende Funktionalität beeinträchtigt (Regression). Implementierung

3) Kleinere Schritte:

Die Implementierung erfolgt in kleinen Schritten, wobei jedes Inkrement ein neuer Test und eine entsprechende Code-Änderung ist.

Vorteile von Test Driven Development (TDD): TDD führt zu höherer Codequalität, indem es frühzeitig potenzielle Fehler aufdeckt und die Software stabiler macht. Zudem bietet es schnelles Feedback über den Codestatus und fördert eine inkrementelle und iterative Entwicklung, was letztendlich zu einer besseren Software führt.

Nachteile von Test Driven Development (TDD): Einführung erfordert zunächst zusätzliche Zeit und Ressourcen für die Erstellung und Wartung der Tests. Entwickler könnten Schwierigkeiten haben, sich daran zu gewöhnen.

g) Akzeptanz

Akzeptanztests sind eine Art von Softwaretests, die durchgeführt werden, um zu überprüfen, ob eine Softwareanwendung die Geschäftsanforderungen und Spezifikationen erfüllt. Das Ziel dieser Tests ist es sicherzustellen, dass das System für die Bereitstellung und den Einsatz in der realen Umgebung bereit ist. Akzeptanztests werden typischerweise vom Endbenutzer oder dem Kunden durchgeführt und sind oft die letzte Testphase vor der endgültigen Freigabe der Software.

In der Akzeptanztest-getriebenen Entwicklung nehmen wir die Akzeptanztests, typischerweise auf Systemebene, als Grundlage für die ganzen weiteren Entwicklungsaktivitäten. Wir leiten aus der Anforderung –meist in Form einer User Story – Akzeptanzkriterien ab, wie wir es in der Anforderungsanalyse bereits kennengelernt haben.

Akzeptanztests: Fokussieren sich auf die Erfüllung der Geschäftsanforderungen und die Benutzerfreundlichkeit aus der Sicht des Endbenutzers.

Systemtests: Konzentrieren sich auf die technische und funktionale Überprüfung des gesamten Systems als Einheit.

h) SOLID Principles von Robert C.Martin

Single Responsibility Principle (SRP)

- 1) Prinzip: Eine Klasse sollte nur eine einzige Verantwortlichkeit haben, also nur einen Grund zur Änderung.
- 2) Erklärung: Jede Klasse sollte nur für eine bestimmte Funktion oder Aufgabe zuständig sein. Dadurch wird die Klasse leichter verständlich und änderbar.

Open Closed Principle (OCP)

- 1) Prinzip: Softwaremodule sollten offen für Erweiterungen, aber geschlossen für Veränderungen sein.
- 2) Erklärung: Bestehender Code sollte nicht verändert werden müssen, um neue Funktionen hinzuzufügen. Stattdessen sollten Erweiterungen durch Vererbung oder durch Implementierung von Schnittstellen möglich sein.

Liskov Substitution Principle (LSP)

- 1) Prinzip: Objekte einer Basisklasse sollten durch Objekte ihrer abgeleiteten Klassen ersetzbar sein, ohne dass sich das Verhalten des Programms ändert.
- 2) Erklärung: Eine Unterklasse muss die Verträge der Basisklasse erfüllen. Methoden der Unterklasse sollten die gleichen Erwartungen wie die der Basisklasse erfüllen.

Interface Segregation Principle (ISP)

- 1) Prinzip: Viele spezialisierte Schnittstellen sind besser als eine allgemeine.
- 2) Erklärung: Anstatt eine große Schnittstelle zu definieren, sollten mehrere kleinere, spezifische Schnittstellen verwendet werden. Dadurch müssen Klassen nur die Methoden implementieren, die sie tatsächlich benötigen.

Dependency Inversion Principle (DIP)

- 1) Prinzip: Abhängigkeiten sollten zu Abstraktionen hin invertiert werden, nicht zu konkreten Implementierungen.
- 2) Erklärung: Hochrangige Module sollten nicht von niederrangigen Modulen abhängen, sondern beide sollten von Abstraktionen abhängen. Dies führt zu einer lockeren Kopplung und einer besseren Modularität des Codes.

Implementierung und Test

a) git commands

- 1) git init (zum Initialisieren eines neuen Git-Repositorys)
- 2) git clone (zum Klonen eines vorhandenen Repositorys)
- 3) git status (zum Anzeigen des Status des Arbeitsbereichs)
- 4) git add (zum Hinzufügen von Änderungen zur Staging- Area)
- 5) git commit (zum Speichern von Änderungen im Repository als Snapshot)
- 6) git push (zum Hochladen von lokalen Änderungen auf ein Remote-Repository)
- 7) git remote (zum Anzeigen, Hinzufügen oder Entfernen von Remote-Repositorys)
- 8) git fetch (zum Herunterladen von Änderungen aus einem Remote-Repository)
- 9) git pull (zum Herunterladen von Änderungen und Zusammenführen mit dem lokalen Branch)
- 10) git branch (zum Anzeigen, Erstellen oder Löschen von Branches)
- 11) git checkout (zum Wechseln zwischen Branches)
- 12) git merge (zum Zusammenführen von Branches)

b) drei Zustände

- 1) **working directory:** Eine Datei befindet sich im Zustand **modified**, wenn sie seit dem letzten Commit bearbeitet wurde, aber die Änderungen noch nicht für den nächsten Commit vorgemerkt wurden.
- 2) **staging area:** Eine Datei befindet sich im Zustand **staged**, wenn die Änderungen für den nächsten Commit vorgemerkt wurden. Dies wird durch den Befehl git add erreicht. Die Staging Area (auch Index genannt) ist eine Zwischenstation, in der Änderungen gesammelt werden, bevor sie committet werden.
- 3) **git directory:** Eine Datei befindet sich im Zustand **committed**, wenn die Änderungen dauerhaft in das lokale Repository gespeichert wurden. Dies geschieht durch den Befehl git commit. Ein Commit speichert die momentanen Inhalte der Staging Area im Repository und markiert sie mit einer eindeutigen Commit-ID.

c) git merge

- 1) **Branches**
Hauptzweig (Mainline, offiziell für alle verfügbare Version), Feature Branch, Peer-Reviewed Commit (Jeder Commit in die Mainline wird in einem Review geprüft), Release Branch (letzte Fehlerkorrekturen vor Release in separatem Branch), Maturity Branch (mehrere Ebenen von Branches je nach Reife)
- 2) **Was ist ein merge-conflict?**
Ein Merge-Konflikt tritt in Git auf, wenn zwei verschiedene Zweige (Branches) unabhängig voneinander Änderungen an denselben Stellen derselben Datei vorgenommen haben und diese Änderungen nicht automatisch zusammengeführt werden können. Git kann dann nicht entscheiden, welche Änderungen beibehalten werden sollen, und benötigt daher manuelle Eingriffe, um den Konflikt zu lösen.

d) Schwäche von Test Coverage/ Code Coverage

Test Coverage gibt nur den Prozentsatz des Programms an, der im Testlauf ausgeführt wird (z.B. die Lines oder Branches), trifft aber keine Aussage über die Qualität oder Sinnhaftigkeit dieser Tests. Außerdem können Tests selber Fehler enthalten und Sonderfälle nicht berücksichtigen. Ein Programm kann 100% Testabdeckung haben, aber in den Tests trotzdem wichtige Facetten von Funktionalität oder potentielle Fehlerfälle nicht berücksichtigen. Der Test zeigt nur die Existenz von Fehlern und nicht die Fehlerursache oder dass keine vorhanden sind. 100 % Zweigüberdeckung garantiert keine vollständige Pfadüberdeckung, stellt nicht sicher, dass alle möglichen Kombinationen von Bedingungen/Pfaden getestet werden oder Berücksichtigt keine Schleifen oder ähnliches.

e) DevOps

DevOps ist ein Kofferwort aus den Begriffen Development (engl. für Entwicklung) und IT Operations (engl. für IT-Betrieb). Unter DevOps versteht man diverse Praktiken, Tools und eine Kulturphilosophie, die die Prozesse zwischen Softwareentwicklungs- und IT-Teams automatisieren und integrieren. Im Vordergrund stehen dabei Teambefähigung, teamübergreifende Kommunikation und Zusammenarbeit sowie Technologieautomatisierung.

Continuous Integration (CI) ist ein Ansatz der agilen Softwareentwicklung. Der Code wird regelmäßig in den Quelltext integriert und automatisch getestet. Auf diese Weise versuchen Entwicklerteams Fehler frühzeitig zu erkennen, zu beheben und so die Ergebnisqualität zu erhöhen. Continuous Integration ist ein DevOps-Verfahren in der Software-Entwicklung, bei der Entwickler alle Codeänderungen regelmäßig in einem zentralen Repository zusammenführen. Diese Änderungen werden dann automatisiert erstellt und getestet.

Continuous Delivery (CD) ist die konsequente Weiterführung von Continuous Integration, da hier auch die Bereitstellung geregelt ist, nicht nur Entwicklung und Test: Continuous Delivery ist ein Software-Engineering-Ansatz, bei dem Teams sicherstellen, dass der Code jederzeit in einem Release-fähigen Zustand ist. Dies bedeutet:

- Automatisierte Tests: Der Code wird kontinuierlich getestet, um sicherzustellen, dass er fehlerfrei ist.
- Continuous Integration: Änderungen werden kontinuierlich in ein gemeinsames Repository integriert, und jede Integration wird verifiziert, um Fehler so früh wie möglich zu erkennen.
- Release-fähiger Zustand: Der Code kann jederzeit manuell in die Produktion gebracht werden. Das Operations-Team oder eine andere verantwortliche Instanz entscheidet, wann und wie oft dieser Code tatsächlich bereitgestellt wird.

Continuous Delivery zielt darauf ab, den Bereitstellungsprozess so zu optimieren, dass neue Funktionen, Bugfixes und andere Änderungen schnell und zuverlässig bereitgestellt werden können, sobald sie fertig sind.

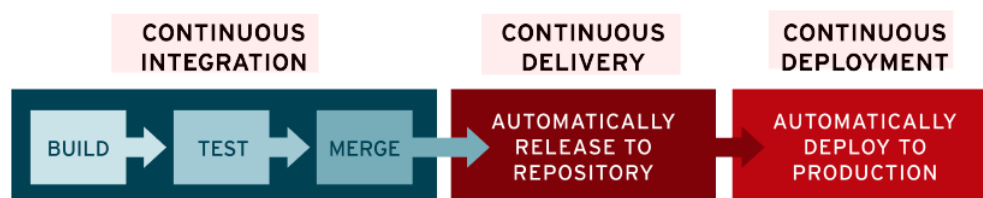
Continuous Deployment (CD) geht einen Schritt weiter als Continuous Delivery:

- Automatisierte Bereitstellung: Jede Änderung, die den automatisierten Testprozess durchläuft und bestanden hat, wird automatisch in die Produktionsumgebung bereitgestellt. Es gibt keinen manuellen Freigabeprozess mehr.
- Schnelle Feedback-Schleifen: Änderungen gelangen schneller zum Endbenutzer, was eine schnellere Rückmeldung ermöglicht und eine schnellere Iteration unterstützt.

Continuous Deployment automatisiert den gesamten Release-Prozess, sodass jedes Mal, wenn ein Entwickler Änderungen an den Code vornimmt und diese den Testprozess bestehen, sie automatisch live gehen.

Zusammengefasst:

Continuous Delivery: Der Code ist jederzeit in einem release-fähigen Zustand und kann nach manueller Freigabe durch ein Operations-Team in die Produktion gebracht werden. Continuous Deployment: Der Code wird automatisch in die Produktion gebracht, sobald er alle automatisierten Tests bestanden hat.



f) Continuous Deployment Patterns

Blue-Green Deployments

- **Problem:** Beim Deployment sollte der Cut-Over, also das Umschalten, sehr schnell gehen, damit die Software schnell wieder verfügbar ist. Auch in die andere Richtung, also das Zurückrollen (Rollback) muss schnell gehen, falls das neue Release fehlerhaft ist.
- **Lösung:** Halte immer zwei lauffähige Systeme vor, zwischen denen schnell, z.B. direkt am Router über das Netzwerk umgeschaltet werden kann. Ein System hat die bisherige, bewährte Version, das andere System eine neu deployte Version.

Canary Releasing

- **Problem:** Beim Deployment einer neuen, fehlerhaften Version mit Features, die Nutzer*innen auswählen, sind die Auswirkungen sehr groß, da schnell alle Nutzer*innen betroffen sind.
- **Lösung:** Um das Risiko zu senken, werden nicht sofort alle Nutzer*innen auf die neue Version umgestellt, sondern nur ein kleiner Anteil. Dieser Anteil wird beobachtet, und sobald wir sicher genug sind, dass keine Probleme auftreten, wird die neue Version auf alle Nutzer*innen ausgerollt.

Dark Launching

- **Problem:** Beim Deployment einer neuen, fehlerhaften Version mit Features, die die Nutzung ergänzen, sind die Auswirkungen sehr groß, da schnell alle Nutzer*innen betroffen sind.
- **Lösung:** Um das Risiko zu senken, werden die neuen Features für die Nutzer*innen unsichtbar deployt. Echte Nutzer*innen-Interaktionen werden zusätzlich zur alten Version auch an das Feature im Backend weitergegeben und das Verhalten analysiert. Erst wenn das neue Feature sich auch mit den echten Interaktionen verhält wie erwartet, schalten wir auf das neue Feature um.

A/B Testing

- **Problem:** Ich habe mehrere Möglichkeiten der Umsetzung eines Features oder der Benutzeroberfläche, aber ich habe keine gute Basis mich für eine Variante zu entscheiden.
- **Lösung:** Ähnlich dem Canary Release testen wir die Varianten gegeneinander. Ein Teil der Nutzer*innen wird zu Variante A geleitet, ein anderer Teil zu Variante B. Wir vergleichen dann anhand passender Messungen, welche Variante besser ist.

Feature Toggles

- **Problem:** Beim kontinuierlichen Deployment wird die Mainline bis hin zur Produktion automatisch veröffentlicht. Wenn wir in der Entwicklung aber auf der Mainline arbeiten sind dort wahrscheinlich unfertige Features, die noch nicht benutzt werden können.
- **Lösung:** Wir bauen in unser System die Möglichkeit einzelne Features ein- und auszuschalten. Damit können unfertige Features im Deployment deaktiviert werden.
- **Alternative Namen:** Feature Flags, Feature Bits, Feature Flippers

g) Kontrollflussgraphen

Ein Kontrollflussgraph ist eine grafische Darstellung, die zeigt, wie die Ausführung von Programmanweisungen oder Codeblöcken miteinander verbunden ist. Er hilft dabei, den Ablauf eines Programms zu visualisieren und die möglichen Ausführungspfade zu verstehen.

h) Prüfergebnisse beim Testen

Ein **richtig positiver** Fehler tritt auf, wenn ein Test tatsächlich einen Fehler findet, der auch tatsächlich existiert.

Ein **falsch positiver** Fehler tritt auf, wenn ein Test einen Fehler meldet, obwohl in Wirklichkeit kein Fehler vorhanden ist.

Ein **richtig negativer** Fehler tritt auf, wenn ein Test keinen Fehler findet, und es gibt tatsächlich keinen Fehler.



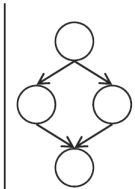
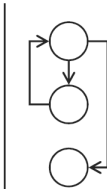
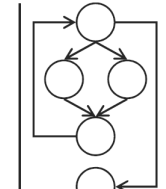
Ein **falsch negativer** Fehler tritt auf, wenn ein Test keinen Fehler findet, obwohl ein Fehler tatsächlich vorhanden ist.

i) McCabes zyklomatische Komplexität

Die zyklomatische Komplexität misst die Anzahl der linearen, unabhängigen Pfade durch den Programmcode. Diese Pfade entsprechen den möglichen Ausführungspfaden, die durch Verzweigungen (wie if-Bedingungen oder Schleifen) entstehen.

$$V(G) = E - N + 2$$

wobei E die Anzahl der Kanten und N die Anzahl der Knoten im Kontrollflussgraphen ist.

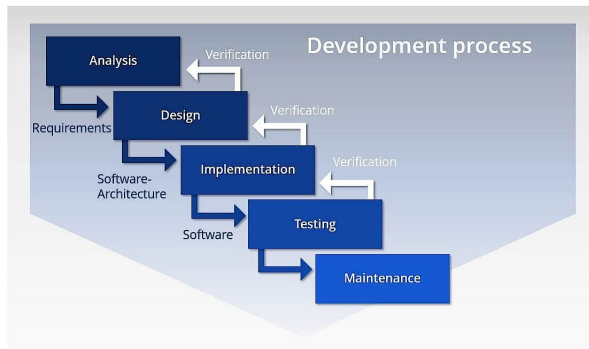
					
Anzahl Kanten	0	2	4	3	6
Anzahl Knoten	1	3	4	3	5
McCabe Zahl	1	1	2	2	3

Projekt- und Produktmanagement

Vorgehensmodelle

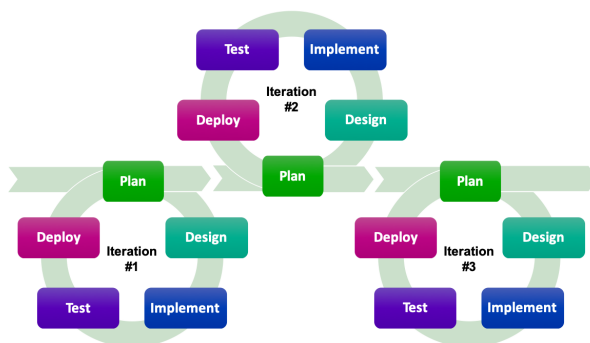
a) Wasserfallmodell

Im Wasserfallmodell „fließt“ eine Projektphase in die Nächste – immer in eine Richtung, ohne die Reihenfolge zu verändern, wie ein Wasserfall eben. Diese Methode ist das Gegenstück zum agilen Projektmanagement, bei dem Abläufe flexibel gestaltet und nicht direkt von Beginn an bis ins Detail definiert sind. Jede Phase im Wasserfallmodell baut auf der vorherigen auf und muss vollständig abgeschlossen sein, bevor die nächste Phase beginnen kann.



b) iterative Entwicklung

Das Prinzip der iterativen Entwicklung beruht darauf, dass ein Produkt durch permanente Wiederholungen (Iterationen) optimiert und zur Anwendungsfähigkeit gebracht wird. In das Endprodukt gehen also alle Erfahrungen aus dem ersten Durchgang ein und auch neuere Entwicklungen. In jeder Iteration werden die Tätigkeiten Analysieren, Entwerfen, Codieren und Testen ausgeführt, und das resultierende System wird erprobt. Nach jeder Iteration sollte eine Abnahme mit dem Kunden erfolgen und so die Anforderungen und die Qualität der Software überprüft werden.



c) inkrementelle Entwicklung

Inkrementelle Entwicklung ist ein Softwareentwicklungsansatz, bei dem die Software schrittweise erstellt wird. Jede Entwicklungsphase liefert ein vollständiges und funktionsfähiges Produktinkrement, das auf den vorherigen Inkrementen aufbaut und zusätzliche Funktionen bietet.

Ein Beispiel für den **Unterschied zwischen iterativ und inkrementell**:

Stellen wir uns vor, eine Dating-Website zu erstellen. Bei einer iterativen Arbeitsweise würde das Team ein wenig an jedem Teil der Seite arbeiten – Profilverwaltung, Suchfunktion, Werbeanzeigen etc. Danach kommt das Team dann auf alle einzelnen Teile zurück und verbessert diese.

Bei dieser rein iterativen Arbeitsweise wird also die gesamte Seite verbessert.

Wenn diese Website inkrementell erstellt würde, würde das Team beispielsweise erst die Profilverwaltung perfektionieren, bevor es die Suchfunktion fertiggestellt usw. Es wird also immer erst die gesamte Arbeit für einen Bereich abgeschlossen, ehe mit etwas Neuem begonnen wird.

d) **Scrum**

Scrum ist ein Vorgehensmodell/Rahmenwerk zur agilen Softwareentwicklung für die Zusammenarbeit von Teams basierend auf einer Definition von Rollen, Meetings und Werkzeugen, die einem Teamstruktur und einen klar definierten Arbeitsprozess basierend auf agilen Prinzipien geben.

Rollen/Scrum Team	Ereignisse	Artefakte
<ul style="list-style-type: none"> • Produktowner • Scrum Master • Developers 	<ul style="list-style-type: none"> • Der Sprint • Sprint-Planung • Sprint- Review • Sprint- Retrospektive • Tägliches Scrum-Meeting 	<ul style="list-style-type: none"> • Product Backlog • Sprint Backlog • Inkrement

Ein Kunde hat einen Wunsch und geht zum **Product Owner** des Scrum-Teams. Zum Produktwunsch des Kunden gibt es meist eine Vision, die langfristig angibt, wohin sich das Produkt entwickeln soll. Aus dieser Vision, dem Produktziel und den Wünschen des Kunden erstellt und pflegt der Product Owner das **Product Backlog**. Es ist eine Liste von allen bekannten Anforderungen und zum aktuellen Zeitpunkt. Das Product Backlog ist mit Wünschen und Aufgaben gefüllt, die man **Product Backlog Items** nennt. Diese können im Format der User-Story gefüllt sein. Das Product Backlog besteht aus **Product Goal** (Ziel), welches das Commitment zum Product Backlog ist.

Sprint Planning: Diese besteht aus 3 Themen: Warum?, Was? und Wie?

Das **Sprint Backlog** ist immer eine Teilmenge vom Product Backlog. Während des Sprint Planning Meetings wählt das Scrum-Team dann eine Teilmenge der Items aus dem Product Backlog aus, die sie im kommenden Sprint bearbeiten möchten. Das Ergebnis des Sprint-Planning ist das Sprint Backlog und das dazugehörige Sprint Goal.

Während des Sprints arbeiten alle Entwickler (oft im Pair-Programming) zusammen an Lösungen. Jeden Tag zur gleichen Zeit findet das **Daily Scrum** statt. Dort fragen sich die Entwickler, wie der Fortschritt auf dem Weg zum Sprint Goal ist. Dabei beantwortet jeder 3 Fragen:

- Was habe ich gestern gemacht?
- Was werde ich heute für das Sprintziel tun?
- Welche Probleme oder Hindernisse gibt es?

Irgendwann nach einer Zeit läuft die Zeitscheibe des Sprints ab und wir erzeugen ein **Increment**, was die fertige nutzbare Arbeit des Sprint darstellt. Damit die Qualität eingehalten wird, gibt es die **Definition of Done**, die sagt wann Arbeit als fertig gilt. Im **Sprint-Review** laden wir User, Stakeholder und Kunden ein und lassen uns über das Inkrement Feedback geben. Das Feedback fließt wieder in das Product Backlog ein. Während wir im Review auf das Produkt schauen, legen wir den Wert in der **Retrospektive** auf den Prozess. Wie hat die Zusammenarbeit geklappt? Verbesserungen gelangen wieder in das Sprint Backlog.

Der **Scrum Master** achtet hierbei auf das ganze Scrum- Konstrukt und die Sprints. Er ist verantwortlich für die Einhaltung von Scrumwerten und -techniken und repräsentiert das Management. Die typische Sprintdauer beträgt 2-4 Wochen.

Organisationsformen für Teams



17

Der Einzelkämpfer Überall, wo sehr begrenzte Aufgaben zu bearbeiten oder einfach nicht mehr Leute verfügbar sind, werden „Einzelkämpfer“ eingesetzt.

- **Merkmale:** Eine Person, arbeitet sehr selbstständig an einer Aufgabe.
- **Vorteil:** Fast kein Kommunikationsaufwand.
- **Nachteile:** Keine Gesprächspartner, kein Dokumentationsdruck, Risiko des Ausfalls.
- **Typisch für:** kleine Projekte, Aufgabenpakete, die sich nicht weiter sinnvoll über mehrere Personen verteilen lassen, unregelmäßige Wartungsaufgaben, Wartung und Weiterentwicklung von Produkten.
- **Empfehlung:** Einzelkämpfer einbinden, vernetzen.



18

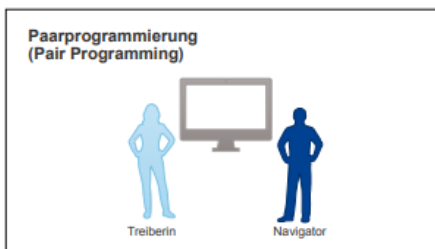
Merkmale: Entstehung der Gruppe

- durch den freien Beschluss der Beteiligten zur Zusammenarbeit. Keine Führungsrolle! („Doppel“) oder
- durch die Weisung an einen Helfer („Sherpa“), einen Spezialisten zu unterstützen. („Tandem“)

Vorteile: Gespräch möglich, implizite QS, keine Katastrophe durch Ausfall einer Person. Sinn des Tandems kann es auch sein, das Wissen des Spezialisten auf zwei Köpfe zu verteilen. Folge: Sherpa = Schüler

Nachteile: Schwierig, wenn sich die beiden nicht gut verstehen

Typisch für: Pair Programming; Einarbeitung.



19

Beim Paarprogrammieren wird die Entwicklung in einer Gruppe von zwei Personen durchgeführt. Dies kann als genereller Entwicklungsansatz oder für spezielle und/oder sehr komplizierte Teile der Software verwendet werden. Dabei ist immer eine Person **Treiber** (Driver) und die andere Person **Navigator** oder **Beobachter** (Navigator, Observer). Die Treiberin ist dabei die durchführende Person, die Code schreibt, während der Navigator parallel alles begutachtet und Hinweise gibt. Man kann es als eine Art kontinuierliches Review betrachten. Die Rollen können dabei jederzeit getauscht werden, was auch häufig passiert.

Inzwischen werden auch viele andere Aktivitäten im Paar durchgeführt (Pair Testing, Pair Modelling). Es gibt Evidenz, dass Paarprogrammieren die Produktivität und Qualität erhöhen kann.



20

- **Merkmale:** Die Entwickler arbeiten im Wesentlichen autonom, nach eigenen Vorgaben und Maßstäben. Hierarchische Beziehungen fehlen oder werden faktisch ignoriert, weil der Wille, die Zeit und/oder die Fähigkeit zur Führung fehlen.

- **Vorteile:** Entwickler sind selbstbestimmt, keine Hierarchie-Probleme, kaum bürokratische Hemmnisse.

- **Nachteile:** Standards, Normen lassen sich nicht durchsetzen. Die Entstehung der erforderlichen Resultate ist Glückssache (d. h. gewisse Dokumente entstehen in aller Regel **nicht**). Die Organisation insgesamt ist nicht lernfähig, Planung, Einführung neuer Methoden und Werkzeuge sind von der Laune der Mitarbeiter abhängig.

- **Typisch für:** Organisationen mit schwacher Führungsstruktur, d. h. Kleingruppen; Einzelkämpfer können in der Regel als anarchisch eingestuft werden; das gilt auch für Studenten; Behörden, Forschungseinrichtungen, auch Firmen.



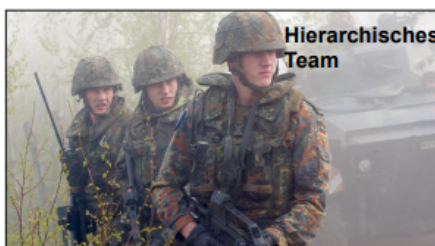
21

- **Merkmale:** Die Beteiligten sind grundsätzlich gleichberechtigt. Sie erzielen durch ausreichende Kommunikation einen Konsens über die Ziele und Wege, und sie verhalten sich diszipliniert.

- **Vorteile:** Die Fähigkeiten der Beteiligten werden optimal genutzt, Probleme werden frühzeitig erkannt und gemeinsam bekämpft.

- **Nachteile:** Hoher Kommunikationsaufwand. Unter Umständen Paralyse (Dissens, Fraktionsbildung)

- **Typisch für:** Forschungsgruppen (unter günstigen Umständen), auch kleine Software-Unternehmen.



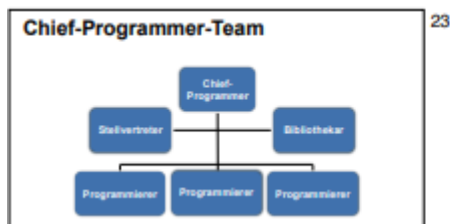
22

- **Merkmale:** Die Gruppe steht unter der Leitung einer Person, die für die Personalführung, je nach Projektform auch für das Projekt verantwortlich ist. Varianten: Gruppenleiter übernimmt Stabs- oder Linienfunktion.

- **Vorteile:** Einfache Kommunikationsstruktur, klare Zuständigkeiten, auf Mitarbeiterebene gute Ersetzbarkeit.

- **Nachteile:** Lange Kommunikationswege (d. h. oft schlechte Information), der Gruppenleiter stellt ein hohes Risiko dar; Gruppenmitglieder sind kaum zur Kooperation motiviert.

- **Typisch für:** Unternehmen und Behörden — die traditionelle Organisationsform.



23

- **Merkmale:** Spezielle Variante des hierarchischen Teams. Die wesentlichen Unterschiede sind die Differenzierung der Rollen in der Gruppe und die Entwickler-Funktion des **Chief-Programmers**.
- Die Gruppe besteht aus dem Chief-Programmer und seinem **Stellvertreter**, einem **Bibliothekar**, der alle Verwaltungsfunktionen übernimmt, sowie aus einigen **Programmierern**.
- Zusätzlich kann es weitere **Spezialisten** geben, z.B. einen Projekt-Verwalter, "Werkzeugmacher", Dokumentierer, Sprach- oder System-Experten, Tester.
- **Vorteile:** Die Gruppe kann — wie ein Operationsteam, das Vorbild dieser Struktur war — außerordentlich effizient arbeiten.
- **Nachteile:** Hohe Ansprüche an die Disziplin, vermutlich auch die Gefahr, dass der Chief-Programmer „abhebt“, sich überschätzt.
- **Typisch für:** ??? (mit Einschränkungen: Open-Source-Projekte) Diese Struktur wurde bei IBM in USA erfunden, es ist unklar, ob sie anderswo auch eingesetzt wurde und mit welchem Erfolg.

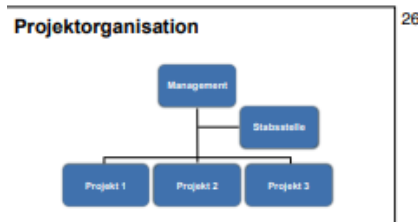
Unternehmensorganisation



25

Hersteller hat immer wieder **ähnliche Aufgaben**. Gruppen oder Abteilungen mit Spezialisten für die Teilaufgaben. Das Produkt entsteht durch das Zusammenwirken der Abteilungen. Die Linienorganisation reicht aus, eine Sekundärorganisation ist nicht erforderlich. Die Strukturen sind projektunabhängig; es gibt **kein Projekt!** Jede Person hat eine definierte (feste) Rolle. Die Zwischenresultate fließen von Abteilung zu Abteilung.

- **Vorbilder** außerhalb der Softwaretechnik: Fabrik mit Serienfertigung, Behörde
- **Vorteile:** stabile Zuordnung der Mitarbeiter, keine Konflikte um Prioritäten
- **Nachteile:** Alles, was am Projektbegriff hängt, fehlt: Identifikation mit einem Projekt, Reaktion auf Probleme, Motivation durch Ziel.



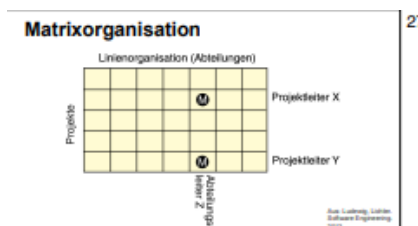
26

Hersteller muss eine **spezielle Aufgabe unter vorgegebenen Bedingungen** lösen. Die Aufgabe wird einem **Projektteam** übertragen. Das Projektteam bildet eine temporäre Struktur (Sekundärorganisation) in der (schwachen) Primärorganisation. Im Extremfall kann die Primärorganisation ganz fehlen (wenn Mitarbeiter von Fall zu Fall nur für ein einziges Projekt eingestellt werden). Das Projekt bestimmt die Struktur, d. h. die Organisation wird nach den Erfordernissen des Projekts gebildet. Mit dem Abschluss des Projekts wird die Struktur aufgelöst.

Vorbild: Die Task-Force, die Spezialeinheit

Vorteile:

- Ein Projekt kann auf die Umgebung (Änderungen der Aufgabe, der Umgebungsbedingungen) sehr rasch reagieren; der Projektleiter hat alle Kompetenzen, um das Projekt zu führen.
- Das Team ist auf das Projekt zugeschnitten, seine Mitglieder identifizieren sich mit dem Projekt. Erfolg oder Misserfolg sind leicht zuzuordnen, die Mitarbeiter sind hoch motiviert,



27

Schwierigkeiten zu vermeiden oder zu überwinden.

Nachteile: Start und Ende des Projekts sind schwierig. Dilemma des Projektleiters: Zu Beginn zu viele Leute oder später zu wenige. Spezialisten sind nicht sinnvoll dauernd einzusetzen. Zwischen Projekten kommt es zu Kämpfen um Köpfe.

In großen Firmen will man weder die Trägheit der funktionalen Organisation noch das Durcheinander der Projektorganisation. Jeder Mitarbeiter steht mit einem Bein in der **Linienorganisation**, mit dem anderen in (mindestens) einem **Projekt**. Die **Primär-**(Linienstruktur) und die **Sekundärorganisation** (Projekte) bilden also eine **Matrix**. Daher spricht man von einer **Matrix-Organisation**. Jeder Mitarbeiter ist für jede Beteiligung an einem Projekt durch einen Punkt repräsentiert.

- **Vorteile:** große Flexibilität, Zugriff auf Spezialisten leicht organisierbar, auch für sehr kleine Projekte anwendbar; jeder Mitarbeiter hat eine stabile Heimat im Unternehmen.
- **Nachteile:** Da jeder Mitarbeiter (mindestens) zwei Vorgesetzte hat, entstehen Zielkonflikte.
- **Typisch für:** Größere Unternehmen und Großunternehmen; auch für Projekte, die „nebenherlaufen“, z. B. Machbarkeitsuntersuchungen, langfristige Konzeptionen.

Begriffe im Software Engineering

Prinzipien

- Rationalität:** Ingenieur*innen sind Feinde des Aberglaubens, Verehrer der Zahlen und Formeln, also Kinder der Aufklärung
- Problemlösen:** Scientists build to learn, engineers learn to build
- Kostenbewusstsein statt Perfektionismus**

- d) **Universeller Anspruch, der nicht vor Fachgrenzen Halt macht:** Ingenieur*innen kapitulieren nicht, wenn das Problem die Grenzen ihres Fachs überschreitet.
- e) **Qualitätsbewusstsein als Denkprinzip**
- f) **Die Einführung und Beachtung von Normen**
- g) **Denken in Baugruppen**

Weshalb sind beim „Programming in the Large“ grundsätzlich andere Techniken nötig sind als beim „Programming in the Small“?

Beim „Programming in the Large“ sind andere Techniken nötig, weil die Komplexität und der Umfang größer sind, was eine sorgfältige Planung der Architektur und umfassende Dokumentation erfordert. Zudem arbeiten hier oft mehrere Entwickler zusammen, was Koordination und Kommunikation mittels spezieller Werkzeuge und Methoden notwendig macht. Im Gegensatz dazu ist „Programming in the Small“ einfacher strukturiert, oft von Einzelpersonen durchgeführt, und benötigt weniger formale Prozesse.

Was ist der Unterschied zwischen Safety und Security?

Safety (Sicherheit im Sinne von Betriebssicherheit/ technische Sicherheit): Bezieht sich auf die Gewährleistung, dass ein System auch unter ungewöhnlichen oder fehlerhaften Bedingungen sicher funktioniert und keine Schäden verursacht. Es geht darum, Unfälle, Fehlfunktionen und Ausfälle zu verhindern, die zu physischen oder materiellen Schäden führen könnten.

Security (Sicherheit im Sinne von IT-Sicherheit/ Informationssicherheit): Bezieht sich auf den Schutz eines Systems vor böswilligen Angriffen, unbefugtem Zugriff und Datenverlust. Es geht darum, die Vertraulichkeit, Integrität und Verfügbarkeit von Daten und Diensten zu gewährleisten.

Beispiel für eine Software, bei der Security eine wichtige Anforderung ist, nicht jedoch Safety:

Online-Banking-Anwendung: Hier ist Security besonders wichtig, um die finanziellen Daten der Benutzer vor unbefugtem Zugriff und Betrug zu schützen. Safety ist weniger relevant, da keine physischen oder materiellen Schäden drohen.

Beispiel für eine Software, bei der Safety in besonderem Maß wichtig ist:

Medizinische Geräte-Software (z.B. für Herzschrittmacher): Hier ist Safety besonders wichtig, da Fehlfunktionen oder Ausfälle direkt zu lebensbedrohlichen Situationen für die Patienten führen können. Security ist ebenfalls wichtig, aber die unmittelbare Betriebssicherheit hat oberste Priorität.

Stakeholder (Interessenten an der Entwicklung und dem Betrieb)

Stakeholder sind Personen oder Gruppen, die ein Interesse oder einen Einfluss auf ein Projekt haben.

Endbenutzer: Personen, die die Software täglich nutzen und deren Anforderungen erfüllt werden müssen.

Kunden/Klienten: Auftraggeber der Software, die sicherstellen möchten, dass das Produkt ihren Bedürfnissen entspricht.

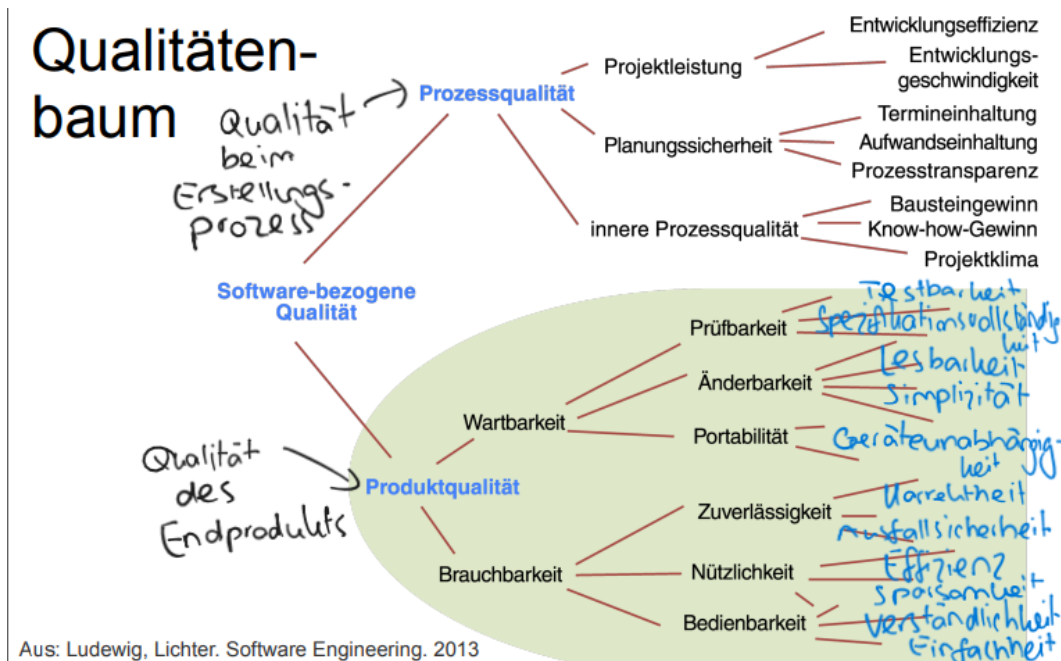
Projektmanager: Verantwortlich für die Planung, Durchführung und Überwachung des Projekts, um sicherzustellen, dass es termingerecht und im Budget bleibt.

Projektleiter: Verantwortlich für die Gesamtplanung, Durchführung und Überwachung des Projekts. Sorgt dafür, dass das Projekt termingerecht, im Budget und entsprechend den Anforderungen abgeschlossen wird. Koordiniert Kommunikation zwischen den Stakeholdern und löst auftretende Probleme.

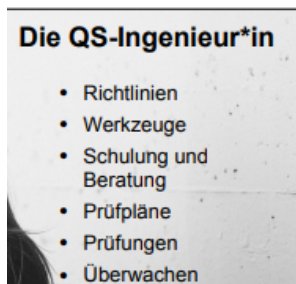
Entwicklungsteam: Entwickler, die für das Design, die Implementierung und das Testen der Software verantwortlich sind.

QA-Team: Qualitätssicherungsmitarbeiter, die Tests durchführen, um sicherzustellen, dass die Software den Qualitätsstandards entspricht.

Qualitätenbaum



Die QS-Ingenieur*in



12

Die Aktivitäten der Qualitätssicherung werden definierten **Rollen** im Projekt (z. B. dem Tester) und in der Organisation zugeordnet. Eine besondere Stellung hat der/die **QS-Ingenieur*in**. Er ist insbesondere für die **organisatorischen Qualitätssicherungsmaßnahmen** verantwortlich. Zu den Aufgaben dieser Rolle zählen:

- Erstellen von **Richtlinien**, Musterdokumenten, Checklisten usw.
- Auswählen und Einführen von **Werkzeugen**
- **Schulung und Beratung** der Projektmitarbeiter in Methoden und Techniken der Qualitätssicherung
- Aufstellen von Plänen, insbesondere von **Prüfplänen**
- Mitwirkung oder Teilnahme an **Prüfungen**, insbesondere bei den formalen Prüfungen an **Meilensteinen**
- **Überwachen** aller Maßnahmen, die im Interesse höherer Qualität durch Pläne, Richtlinien usw. vorgesehen sind

Software-Review

Eine Software-Review ist eine formelle oder informelle Methode zur Bewertung der Qualität, des Designs und der Funktionalität von Software durch eine oder mehrere Personen, die nicht direkt an der Entwicklung der überprüften Software beteiligt sind. Ziel ist es, Fehler und Schwächen frühzeitig zu identifizieren, die Qualität zu verbessern und sicherzustellen, dass die Software den festgelegten Anforderungen und Standards entspricht.

Zuständigkeiten und Verantwortlichkeiten sind im Review klar durch folgende Rollen definiert:

Der/die **Moderator*in** leitet das Review, ist also für den ordnungsgemäßen Ablauf verantwortlich.

Der/die **Notar*in** führt das Protokoll.

Die **Gutachter*innen** sind Kolleg*innen, die den Prüfling beurteilen können.

Der/die **Autor*in** ist Urheber/in des Prüflings oder ein/e Repräsentant*in des Teams, das den Prüfling erstellt hat.

Der/die **Manager*in** (Linienvorgesetzte oder Projektleiter*in) hat den Auftrag zur Erstellung des Prüflings gegeben und somit auch die Verantwortung für die Freigabe des Prüflings. **Er/sie sollte** (vor allem in den ersten Versuchen) **nicht am Review teilnehmen!**

Das Review-Team bilden alle Teilnehmer*innen des Reviews außer dem/r Autor*in.

Nachteile von Tests im Vergleich zu Review z.bsp.

Aussagekraft des Tests wird überschätzt, zeigt nicht Korrektheit, denn schon die Zustandsräume kleiner Programme sind riesig, Man kann nicht alle Anwendungssituationen nachbilden, Der Test zeigt nur die Existenz von Fehlern und nicht die Fehlerursache oder dass keine vorhanden sind.

Testabdeckung misst nur den Teil des Codes der ausgeführt wird, aber nicht wie gründlich sie sind.

Prüfungsfragen

1. Aspekte, die ein Entwurf gegenüber den Anforderungen zusätzlich enthalten sollte:

Anforderungen: Fokus auf **was** das System tun soll.

Entwurf: Fokus auf **wie** das System die Anforderungen technisch umsetzen wird.

Was ist im Entwurf, aber nicht in den Anforderungen?: Diagramme, Art der Datenspeicherung, Architektur (Struktur), Aufteilung der Komponenten, verwendete Programmiersprache, APIs

2. Software ist immateriell. Erkläre

Software ist kein physisches Objekt, das man anfassen oder sehen kann. Stattdessen besteht Software aus digitalen Daten, die auf einem Computer oder einem anderen digitalen Gerät gespeichert sind. Sie läuft also auf physischen Geräten und man kann mit ihr interagieren. Die immaterielle Natur von Software hat weitreichende Auswirkungen auf ihre Entwicklung, Verteilung, Nutzung und rechtliche Behandlung.

3. Prozessqualität vs. Produktqualität

Während Prozessqualität darauf abzielt, die Effizienz, Konsistenz und Verbesserung der Abläufe zu maximieren, indem zB Fehler vermieden und behandelt werden, konzentriert sich Produktqualität auf die Eigenschaften und die Zufriedenheit der Endnutzer des Produkts, also die Zuverlässigkeit, Benutzerfreundlichkeit, Funktionalität und Haltbarkeit. Beide Aspekte sind eng miteinander verbunden, da eine hohe Prozessqualität oft zu einer besseren Produktqualität führt. Ein gut strukturierter und effizienter Prozess kann die Wahrscheinlichkeit von Fehlern reduzieren und die Produktion von qualitativ hochwertigen Produkten sicherstellen.

4. 2 Analysetechniken, die sich besonders gut zur Feststellung des Ist-Zustandes eignen

Die SWOT-Analyse bietet eine umfassende Bewertung der internen und externen Faktoren, die den aktuellen Zustand beeinflussen.

- Bietet eine strukturierte und umfassende Analyse.
- Einfach anzuwenden und zu verstehen.
- Hilft, sowohl interne als auch externe Faktoren zu berücksichtigen.

Prozessanalyse ermöglicht eine detaillierte Untersuchung der bestehenden Abläufe.

- Detaillierte und klare Visualisierung der aktuellen Abläufe.
- Ermöglicht die Identifikation spezifischer Probleme und Verbesserungsmöglichkeiten.
- Grundlage für die Prozessoptimierung und Neugestaltung.

5. Was ist Software Alterung?

Software-Alterung beschreibt den Prozess, bei dem Software im Laufe der Zeit an Leistungsfähigkeit und Zuverlässigkeit verliert, auch wenn sie nicht aktiv genutzt wird. Dieser Effekt kann durch verschiedene Faktoren verursacht werden, die sowohl auf der Software selbst als auch auf ihrer Umgebung basieren.

Ursachen: Ressourcenlecks, Datenkorruption, Software- und Systemänderungen, Änderungen in der Nutzung

Symptome: Leistungsabfall, Erhöhte Fehlerraten, Ressourcenerschöpfung, Unerwartetes Verhalten

6. Nenne ein Dokumententyp, der nicht geeignet ist durch ein Review geprüft zu werden

Ein Dokumententyp, der nicht geeignet ist durch ein Review bzw. durch einen Test geprüft zu werden, sind persönliche Notizen.

Warum nicht für Reviews geeignet? Subjektiver Inhalt, Nicht für die Öffentlichkeit bestimmt, Mangel an formaler Struktur

Warum nicht für Tests geeignet? Kein formalisierter Inhalt, Fehlende Zielsetzungen, Unstrukturierte Informationen

7. Erkläre die Make or Buy-Entscheidung und Argumente für und gegen die Neuentwicklung

Die Make-or-Buy-Entscheidung im Software Engineering bezieht sich auf die strategische Entscheidung, ob eine Softwarelösung intern entwickelt (Make) oder extern beschafft (Buy) werden soll. Diese Entscheidung basiert auf einer Vielzahl von Faktoren, darunter Kosten, Zeit, Ressourcen und strategische Ziele des Unternehmens.

- für die Neuentwicklung: Eine intern entwickelte Software kann exakt auf die spezifischen Bedürfnisse und Anforderungen des Unternehmens zugeschnitten werden. Dies ermöglicht eine höhere Anpassungsfähigkeit und Flexibilität, um spezifische Geschäftsprozesse und -anforderungen zu erfüllen.
- gegen die Neuentwicklung: Die Entwicklung einer maßgeschneiderten Softwarelösung kann sehr kostspielig und zeitaufwendig sein. Dies umfasst nicht nur die initialen Entwicklungs- und Implementierungskosten, sondern auch die langfristigen Wartungs- und Updatekosten.

8. Fehlhandlung(Mistake) und Ausfall(Failure)/ Fehlerursache(Fault) und Fehlerzustand(Error)

- Fehlhandlung(Mistake): Eine Fehlhandlung ist eine menschliche Aktion, die zu einem Fehler führt. Dies kann durch Unachtsamkeit, Missverständnisse oder mangelndes Wissen geschehen, kann aber auch durch andere menschliche Fehler verursacht werden, wie z.B. bei der Anforderungsanalyse, beim Design oder bei der Dokumentation.
- Fehlerursache(Fault,Bug): Eine Fehlerursache ist die zugrundeliegende Ursache für einen Fehlerzustand. Es ist der eigentliche Defekt im System, der durch eine Fehlhandlung eingebracht wurde. Die Stelle, die falsch implementiert ist, ist dann die Fehlerursache.
- Fehlerzustand(Error): Ein Fehlerzustand ist der Zustand eines Systems, der von der Spezifikation abweicht, verursacht durch eine Fehlhandlung, aber noch nicht notwendigerweise ein Ausfall passiert ist. Ein Fehlerzustand repräsentiert eine interne Abweichung des Systems.
- Ausfall(Failure): Ein Ausfall tritt auf, wenn ein System sein vorgesehenes Verhalten nicht erfüllt, also wenn ein Fehlerzustand nach außen sichtbar wird und die Funktionalität des Systems beeinträchtigt. Ein Ausfall kann sich auf verschiedene Arten äußern, z.B. als Absturz, als falsche Berechnung, als unvollständige Ausgabe oder als unerwartetes Verhalten. Der Benutzer bemerkt, dass das System nicht (mehr) den geforderten Dienst erbringt.
- Defekt (Defect): Generische Bezeichnung für Ausfall und Fehlerursache

9. Alles zu Risikomanagement

Das Ziel des Risikomanagements in der Softwareentwicklung ist es, potenzielle Risiken frühzeitig zu identifizieren, zu bewerten und geeignete Maßnahmen zu ergreifen, um die negativen Auswirkungen dieser Risiken auf das Projekt und das Endprodukt zu minimieren oder zu verhindern. Dadurch sollen die Wahrscheinlichkeit und die Folgen unerwünschter Ereignisse reduziert werden, um die Projektziele zu erreichen und die Qualität der Software zu gewährleisten.

Warum ist Risikomanagement wichtig? Vermeidung von Projektverzögerungen, Kosteneffizienz, Qualitätssicherung, Stakeholder-Vertrauen, Kontinuierliche Verbesserung

Top 10 Risk Items

1. Personelle Probleme
2. Unrealistische Zeit- und Kostenpläne
3. Entwicklung der falschen Funktionalität
4. Entwicklung der falschen Benutzeroberfläche
5. Gold Plating
6. Kontinuierlicher Strom von Anforderungsänderungen
7. Probleme mit extern durchgeführter Aufgaben
8. Probleme mit extern entwickelten Komponenten
9. Probleme mit Echtzeit-Performanz
10. Überforderung der Möglichkeiten der Informatik

Gegenmaßnahmen:

1. Top-Talente als Mitarbeiter, passende Profile, Team-Building, Morale-Building, Cross-Training, Schlüsselpersonen früh ins Projekt
2. Detaillierte Kosten- und Zeitplanung basierend auf verschiedenen Quellen, Design to Cost, inkrementelle Entwicklung, Wiederverwendung, Anforderungen zurückstellen
3. Organisationsanalyse, Nutzungsanalyse, Formulierung eines Einsatzkonzepts, Nutzerbefragung, Prototyping, frühes Benutzerhandbuch
4. Prototyping, Szenarien, Nutzungssanalyse
5. Anforderungen zurückstellen, Prototyping, Kosten/Nutzen-Analyse, Design to Cost
6. Hohe Änderungsschwelle, inkrementelle Entwicklung
7. Referenzen prüfen, Team Building, Prüfungen vor der Auftragsvergabe, Prototyping, Beauftragung mehrerer Zulieferer
8. Benchmarking, Inspektionen, Referenzen prüfen, Kompatibilitätsanalyse
9. Simulation, Benchmarking, Modellierung, Prototyping, Instrumentierung, Tuning
10. Technische Analyse, Kosten/Nutzen-Analyse, Prototyping

10. Was ist ein Arbeitspaket, woraus besteht es (durch welche Informationen ist es definiert)?

Ein Arbeitspaket repräsentiert eine spezifische Aufgabe oder eine Gruppe von Aufgaben, die durchgeführt werden müssen, um ein Projektziel zu erreichen. Arbeitspakete sind dabei so klein und spezifisch, dass sie klar definiert und handhabbar sind. In der Software-Entwicklung bestehen Arbeitspakete typischerweise aus folgenden Elementen:

Titel/Bezeichnung, Beschreibung, Ziele und Akzeptanzkriterien, Verantwortlichkeiten/Rollen, Zeitplan, Abhän-

gigkeiten, Ressourcen, Ergebnisse/Deliverables, Risiken und Annahmen, Aufwandschätzung

Ein Arbeitspaket hilft dabei, ein Projekt in überschaubare Teile zu zerlegen, die effektiv geplant, überwacht und kontrolliert werden können. Es fördert die klare Kommunikation und Verantwortlichkeit innerhalb des Teams und stellt sicher, dass alle Beteiligten ein gemeinsames Verständnis der Aufgaben und Ziele haben.

11. Nenne 3 Beispiele für nichtfunktionale Anforderungen

- Performance: Das System muss in der Lage sein, mindestens 1000 Transaktionen pro Sekunde zu verarbeiten. Antwortzeiten für Benutzeranfragen dürfen nicht länger als 2 Sekunden betragen.
- Sicherheit: Alle Datenübertragungen müssen durch SSL/TLS verschlüsselt werden. Benutzer müssen sich über ein zweistufiges Authentifizierungsverfahren anmelden.
- Zuverlässigkeit/Verfügbarkeit: Das System muss eine Verfügbarkeit von 99,9% während der Geschäftszeiten garantieren. Im Falle eines Systemausfalls muss das System innerhalb von 1 Stunde wiederhergestellt werden können.

12. Aufwandsschätzung

Wie viel Zeit wird benötigt, um das Arbeitspaket oder das gesamte Projekt abzuschließen? Welche Ressourcen werden benötigt, um die Aufgaben erfolgreich zu erledigen?

Warum brauchen wir in der Softwareplanung eine Aufwandschätzung?

- Projektplanung und -kontrolle: Eine präzise Aufwandsschätzung ermöglicht eine realistische Projektplanung. Sie hilft dabei, den zeitlichen Rahmen und die benötigten Ressourcen zu bestimmen, was wiederum die Erstellung eines detaillierten Projektplans erleichtert. Ohne eine solche Schätzung besteht die Gefahr, dass Projekte zu lange dauern oder die verfügbaren Ressourcen überschreiten.
- Budgetierung und Ressourcenmanagement: Aufwandschätzungen sind entscheidend für die Budgetplanung. Sie ermöglichen es, die Kosten für ein Projekt im Voraus zu berechnen und sicherzustellen, dass ausreichend finanzielle und personelle Ressourcen zur Verfügung stehen. Dies hilft, finanzielle Überziehungen zu vermeiden und sicherzustellen, dass die richtigen Fähigkeiten und Kapazitäten zur Verfügung stehen, um das Projekt erfolgreich abzuschließen.

13. Beschreiben Sie, was Planning Poker ist.

Planning Poker ist eine agile Methode zur Schätzung des Aufwands in Softwareentwicklungsprojekten, die oft in Scrum-Teams verwendet wird. Es ist eine kollaborative Technik, bei der das gesamte Team beteiligt ist, um den Arbeitsaufwand für User Stories oder andere Arbeitspakete zu schätzen. Der Prozess fördert die Diskussion und den Austausch von Wissen innerhalb des Teams und hilft, eine realistischere und konsensbasierte Schätzung zu erzielen.

14. Fasse Prototyping zusammen.

Prototyping ist ein iterativer Entwicklungsansatz in der Software-Entwicklung, bei dem ein vorläufiges Modell oder eine frühe Version eines Produkts erstellt wird, um verschiedene Aspekte des Endprodukts zu testen und zu validieren. Ziel des Prototyping ist es, Feedback von Nutzern und Stakeholdern zu erhalten, um Anforderungen besser zu verstehen und das Design zu verfeinern, bevor das vollständige System entwickelt wird.

15. Ebenen beim Entwurf (einer Software)

- Systemebene: Fokus auf das Gesamtsystem und seine Umgebung.
- Softwarearchitekturebene: Definiert die grundlegende Struktur und die Interaktionen der Hauptkomponenten der Software.
- Detailliertes Design: Spezifiziert das Design und die Implementierung einzelner Module und Komponenten.
- Implementierungsebene: Konkrete Programmierung und Codierung der Software.

Die Softwarearchitektur gehört zur Softwarearchitekturebene und bildet das Rückgrat des Softwareentwurfs, indem sie die Struktur, die Interaktionen und die grundlegenden Prinzipien festlegt, die die gesamte Softwareentwicklung leiten. Sie stellt sicher, dass das System den funktionalen und nicht-funktionalen Anforderungen entspricht und eine solide Basis für die weitere Detaillierung und Implementierung bietet.

16. Durch was wird der Nutzen einer Software beeinflusst?

Der Nutzen einer Software wird durch eine Vielzahl von Faktoren beeinflusst: Funktionalität, Benutzerfreundlichkeit, Zuverlässigkeit, Leistung, Sicherheit oder Wartbarkeit

17. Use Cases vs. User Stories

Use Cases und User Stories sind beide Methoden zur Dokumentation und Definition von Anforderungen in der Softwareentwicklung, unterscheiden sich jedoch in ihrer Detaillierung und Herangehensweise.

- User Stories sind kurze, einfache Beschreibungen von Funktionen oder Anforderungen aus der Sicht eines Endbenutzers. Sie folgen typischerweise dem Format: Als [Rolle] möchte ich [Funktion], um [Ziel] zu erreichen.
- Use Cases sind detaillierte Beschreibungen der Interaktionen zwischen einem Benutzer (oder einem externen System) und der Software, um ein bestimmtes Ziel zu erreichen. Sie umfassen oft die Schritte, die durchgeführt

werden müssen, um ein bestimmtes Ergebnis zu erzielen.

User Stories bieten eine schnelle, benutzerzentrierte Sicht auf die Anforderungen, während Use Cases detaillierte Szenarien und Abläufe liefern. User Stories sind ideal für agile Umgebungen, in denen Flexibilität und schnelle Anpassungen wichtig sind. Use Cases sind besser geeignet, wenn ein detailliertes Verständnis der Systeminteraktionen erforderlich ist, z.B. in einer Phase der detaillierten Planung oder Analyse.

18. Was ist die Rolle des Software Architekten?

Die Rolle des Softwarearchitekten ist entscheidend für den Erfolg eines Softwareprojekts. Er oder sie ist verantwortlich für die Entwicklung und Dokumentation der Softwarearchitektur, die als Grundlage für die Implementierung und Wartung des Systems dient.

- Architekturentwurf
- Anforderungsanalyse
- Technologiewahl
- Sicherstellung der Qualitätsmerkmale
- Kommunikation und Dokumentation
- Unterstützung der Implementierung
- Qualitätskontrolle und Refactoring
- Risikomanagement

19. Software-Krise und Ursachen

Die Software-Krise beschreibt eine Reihe von Herausforderungen und Problemen, die in den frühen Tagen der Softwareentwicklung auftraten, als die Industrie begann, zunehmend komplexe Softwareprojekte zu realisieren. Das Schlagwort Software-Krise entstand in den 1960er Jahren, als die Schwierigkeiten im Umgang mit der wachsenden Komplexität und dem Bedarf an zuverlässiger und wartbarer Software offensichtlich wurden.

Ursachen:

- Komplexität der Software - Unzureichende Entwicklungsprozesse - Mangelnde Planung und Dokumentation - Fehlende Wartbarkeit und Zuverlässigkeit - Unzureichende Testmethoden

20. Der Begriff der Qualitätssicherung war in drei Teilbegriffe(Schwerpunkte) zerlegt worden. Welche sind das?

- Organisatorische Maßnahmen: Diese umfassen die Strukturierung und Planung des gesamten Softwareentwicklungsprozesses. Dazu gehören die Definition von Prozessen, Standards und Richtlinien, die Organisation von Teams und die Implementierung von Managementmethoden, um sicherzustellen, dass alle Aspekte der Softwareentwicklung effizient und zielgerichtet ablaufen.
- Konstruktive Maßnahmen: Hierbei geht es um die Schaffung einer soliden Basis für die Softwareentwicklung durch gute Design- und Entwicklungspraktiken. Dies umfasst unter anderem die Anwendung bewährter Methoden in der Softwarearchitektur, das Einhalten von Coding-Standards und das Nutzen von Entwurfsmustern, um eine hohe Qualität des entwickelten Codes zu gewährleisten.
- Analytische Maßnahmen: Diese beziehen sich auf die systematische Überprüfung und Bewertung der Software, um Fehler zu identifizieren und die Qualität zu sichern. Dazu gehören Software-Prüfungen wie Unit-Tests, Integrationstests, Systemtests und andere Testmethoden, die eingesetzt werden, um sicherzustellen, dass die Software die festgelegten Anforderungen erfüllt und funktionsfähig ist.



21. Welche Nachteile hat es, im Test Fehler zu beheben sobald sie aufgetreten sind?

Das Beheben von Fehlern erst während der Testphase kann zu erhöhten Kosten, Zeitverzögerungen, zusätzlicher Komplexität und reduzierter Testeffizienz führen. Es beeinträchtigt auch die Fähigkeit, die zugrunde liegenden Ursachen von Problemen zu identifizieren und zu adressieren. Idealerweise sollten Fehler so früh wie möglich im Entwicklungsprozess identifiziert und behoben werden, um diese Nachteile zu minimieren. Dies kann durch präventive Maßnahmen wie sorgfältige Anforderungsanalyse, Designüberprüfung, kontinuierliche Integration und

frühzeitiges Testen erreicht werden.

22. Welche Voraussetzungen müssen erfüllt sein damit mehrere Personen ein demokratisches Team bilden können?

Die Beteiligten sind grundsätzlich gleichberechtigt. Sie erzielen durch ausreichende Kommunikation einen Konsens über die Ziele und Wege und sie verhalten sich diszipliniert.

- Klare Kommunikation und Transparenz
- Gegenseitiger Respekt und Vertrauen
- Gleichberechtigung
- Gemeinsame Ziele und Werte
- Teamarbeit

23. Was versteht man unter einem quantifizierten Qualitätsmodell, und wie kann man es nutzen?

Ein quantifiziertes Qualitätsmodell ist ein System zur Bewertung und Messung der Qualität von Software oder anderen Produkten auf einer quantitativen Basis. Es definiert und verwendet Metriken und Kennzahlen, um die verschiedenen Aspekte der Qualität zu messen und zu bewerten. Diese Art von Modell ermöglicht eine objektive und messbare Beurteilung der Qualität, im Gegensatz zu subjektiven Bewertungen oder allgemeinen Beschreibungen.

Nutzen:

- Qualitätsbewertung
- Fehleridentifikation und -management
- Qualitätsverbesserung
- Vergleich und Benchmarking
- Kommunikation und Reporting
- Risikomanagement

24. Warum sind Meilensteine in der Software-Entwicklung wichtig?

Meilensteine sind essenziell für die erfolgreiche Durchführung von Softwareprojekten. Sie bieten Struktur, ermöglichen Fortschrittsüberwachung, erleichtern die Kommunikation und helfen bei der Steuerung und Risikomanagement. Durch die Definition und das Erreichen von Meilensteinen können Teams und Projektleiter sicherstellen, dass das Projekt zielgerichtet und effizient voranschreitet.

25. Geben Sie zwei Gründe an, warum Anforderungen schriftlich festgehalten werden sollten.

Das schriftliche Festhalten von Anforderungen ist entscheidend, um Klarheit und Verständlichkeit zu gewährleisten sowie um Nachvollziehbarkeit und Verfolgbarkeit sicherzustellen. Es hilft, Missverständnisse zu vermeiden, Änderungen zu dokumentieren und als rechtlich verbindliches Referenzdokument zu dienen.

26. Stellen Sie sich vor, es wurde eine neue Abteilung an der Uni für Software-Qualitätssicherung eingerichtet. Da die meisten Mitarbeiter der Meinung waren, dass Software-Evolution und Wartung weniger wichtig sind, haben sich die Mitarbeiter nicht mit Evolution und Wartung beschäftigt. Welches Entwicklungs- und Evolutionsmodell würden Sie für die Mitarbeiter der Abteilung für Software-Qualitätssicherung empfehlen?

Die Abteilung für Software-Qualitätssicherung sollte ein agiles Entwicklungsmodell mit starkem Fokus auf Qualitätssicherung implementieren. Agile Methoden wie Scrum und Testgetriebene Entwicklung (TDD) ermöglichen iterative Entwicklung, kontinuierliche Verbesserung und enge Zusammenarbeit. Tools wie Continuous Integration/Continuous Delivery (CI/CD) und statische Codeanalyse unterstützen dabei, die Softwarequalität zu gewährleisten und schnelle Anpassungen an veränderte Anforderungen vorzunehmen. Durch regelmäßige Schulungen und einen Kulturwandel kann die Abteilung die Bedeutung von Software-Evolution und Wartung anerkennen und langfristig hochwertige, anpassungsfähige Software liefern.