Zusammenfassung SQS

Inhaltsverzeichnis

Grundlagen der Qualitätssicherung (VO 1)

Qualitätssicherung ist ein wesentlicher Bestandteil moderner Softwareentwicklung.

Stellenwert gewachsen durch:

- Steigende Größe und Komplexität von Softwareprojekten
- Hoher Grad an Integration und Vernetzung zwischen Systemen
- Zusammenspiel vieler Technologien
- Steigender Einsatz von Technologie im Alltag (IOT)
- Regulatorische Anforderungen (Vorschriften, Standards, Gesetze)

Schaden durch Qualitätsmangel: (Bsp. VO1, S.5)

- Finanzielle Folgen
- Personenschaden
- Verlust an Reputation eines Unternehmens
- Verlust des Vertrauens der Benutzer

Ziel ist die Vermeidung bzw. frühzeitige Identifikation von Qualitätsproblemen.

Software Qualität

Definition

Wann hat eine Software gute Qualität?

- Keine einheitliche Begriffsdefinition
- Multidimensionaler Begriff, **Gesamtqualität** setzt sich **aus vielen einzelnen Qualitätsaspekten** zusammen
- Ein „high Quality"-System ist **kontext**- und **projektabhängig** und oft subjektiv geprägt
- Komplexität auf zwei Ebenen
    - Welche Eigenschaften eines Produkts definieren Qualität?
    - Welche Ausprägung müssen diese Eigenschaften annehmen?
- **Messbarkeit** und **Überprüfbarkeit** muss gegeben sein

Definition nach **IEEE-730**

Software Qualität gem. IEEE-730 bedeutet erfolgreiche: (Bsp. Vo1, S.10)

- **Verifikation**: Konformität gegenüber den spezifizierten Anforderungen
    - *Bauen wir das Produkt richtig?*
- **Validierung**: Konformität gegenüber dem vorgesehenen Zweck und den Bedürfnissen der Stakeholder („Fitness for use")
    - *Bauen wir das richtige Produkt?*

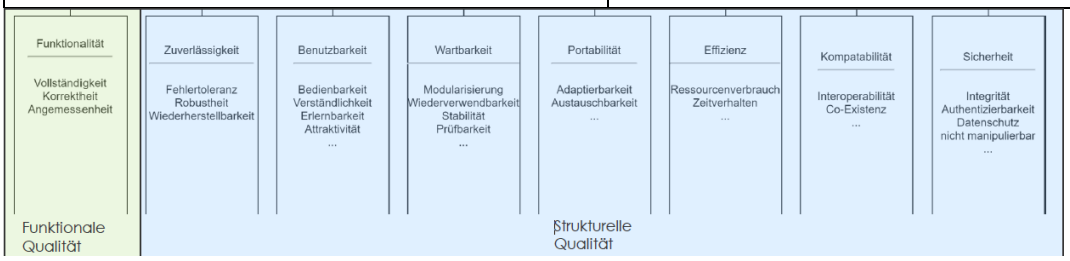
Nur eines davon ist nicht ausreichend, um von qualitativer Software zu sprechen.

Definition nach **ISO/IEC 25010**

- Definiert ein **Set an Qualitätsfaktoren und** zugehörigen **Qualitätskriterien**
  - Qualitativ oder quantitativ messbar
  - Helfen bei der Spezifikation von Anforderungen
  - Ermöglichen Beurteilung des Erfüllungsgrades
- Umfassendes Rahmenwerk zur Definition und Bewertung der Software Qualität
- Projektumfeld und Domäne steuern den Stellenwert der einzelnen Qualitätsfaktoren
- Unterteilung in zwei Untergruppen:
  - **Funktionale** Qualität
  - **Strukturelle** Qualität

Qualitätsfaktoren:

Funktionale Qualität:	Strukturelle Qualität:
- Die äußere Sicht (Benutzersicht) auf ein System - Externe Merkmale des Systems - Definieren das “was?” (Bezug zu funktionalen Anforderungen)	- Die innere Sicht auf ein System - Interne Merkmale und Eigenschaften eines Systems - Definieren das “wie gut?” (Bezug zu nicht-funktionalen Anforderungen)



Software Qualitätssicherung

Definition

- Maßnahmen und Methoden, die der Überprüfung und Überwachung der Software Qualität dienen.
- Sicherstellung, ob und zu welchem Grad die Qualitätsfaktoren erfüllt sind.
- Kontinuierlicher Prozess, der den gesamten SW-Lebenszyklus begleitet.

Klassifikation Qualitätssicherung

Organisatorische Methoden (konstruktive QS-Maßnahmen)

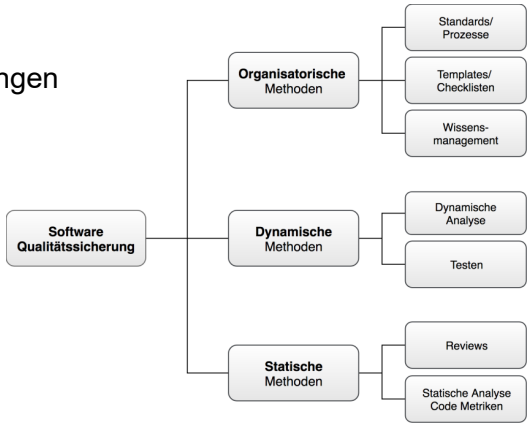
- Schaffen notwendige Infrastruktur und Rahmenbedingungen
- Definieren und steuern den Qualitätsprozess

Dynamische Methoden (analytische QS-Maßnahmen)

- Software bzw. Teile davon müssen ausführbar sein
- Fokus auf das Verhalten zur Laufzeit (“äußere Qualität”)

Statische Methoden (analytische QS-Maßnahmen)

- Software muss nicht aus-führbar bzw. integriert sein
- Fokus auf die interne Struktur („innere Qualität“)



Ziel:

- Prävention von Qualitätsproblemen
- Identifikation von Qualitätsmängeln
- Kontinuierliche Überwachung einzelner Qualitätsaspekte

Ergänzen einander durch unterschiedliche Sichtweisen:

- Äußere Qualität: Benutzersicht, Systemverhalten
- Innere Qualität: interne Sicht, u.a. Architektur, Wartbarkeit, Verständlichkeit, …

Fehlerkosten

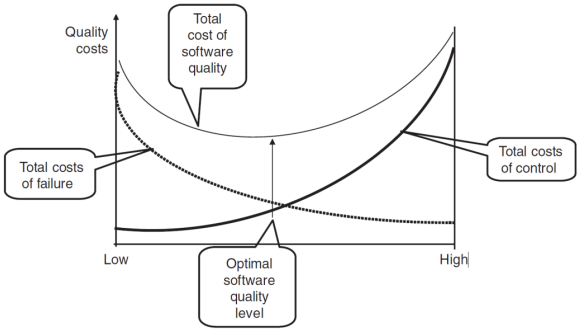
- QS-Maßnahmen sollten möglichst frühzeitig und begleitend zu den Phasen eines SW- Projekts stattfinden
- Fehlerkosten steigen, je später ein Fehler identifiziert/behoben wird

Qualitätskosten

- Gesamtheit aller geplanten und ungeplanten Kosten der Qualitätssicherung
  - Prüfkosten (geplante Kosten)
  - Fehlerkosten (ungeplante Kosten)
- auch unterlassene Qualitätssicherung verursacht Kosten

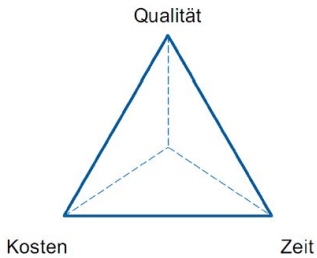
Qualitätskosten –Balance

- Proaktive Investition in Qualitätssicherung vs. hohe Fehlerkosten
- Ziel ist ein ausgewogenes Verhältnis zwischen Prüfkosten und Fehlerkosten



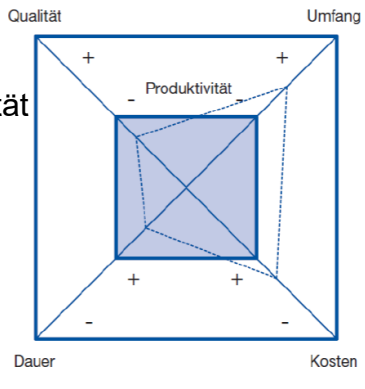
Einflussfaktoren –Magisches Dreieck

- In einem Projekt konkurrieren die Faktoren **Qualität**, **Zeit** und **Kosten** um die verfügbare Gesamtproduktivität
- Qualität (Umfang und Inhalt) steht im Spannungsverhältnis mit den anderen Faktoren
- Verzerrungen in eine Dimension gehen zu Lasten einer oder beider anderen



Einflussfaktoren –Teufelsquadrat nach Sneed

- Erweiterung des magischen Dreiecks, durch Trennung von **Qualität** und **Umfang**
- Mehr Handlungsoptionen bei Verzerrung von Dauer oder Kosten:
  - Qualität kann gehalten werden, unter Reduktion des Umfangs
  - (Umfang kann gehalten werden, unter Reduktion der Qualität)
- Die Grundfläche ist fix, sie symbolisiert die verfügbare Produktivität



Qualitätssicherung in der Softwareentwicklung

Vorgehensmodelle in der Softwareentwicklung

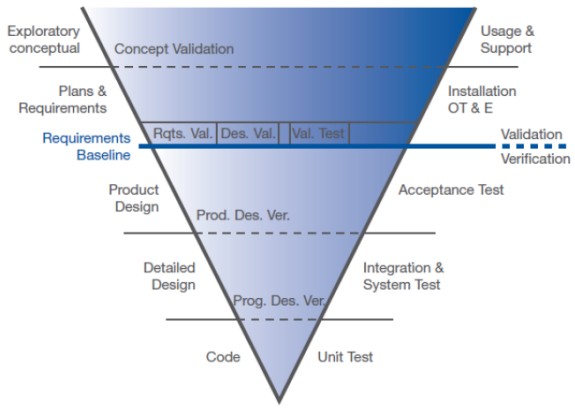
- Vorgehensmodelle geben SW-Projekten einen einheitlichen Rahmen und Ablauf
- Arten von Vorgehensmodellen:
  - Sequenzielle (historische) Modelle (z.B. Wasserfall, V-Modell)
  - Iterative/Inkrementelle Modelle (z.B. Spiralmodell, V-Modell XT)
  - Agile Modelle (z.B. Scrum, Kanban)
- Kein Universalmodell: Wahl und Anpassung des Vorgehensmodells je nach Projekt
- Fundamentale Phasen sind in unterschiedlicher Form in jedem Modell aufgegriffen:

Einbettung Qualitätssicherung

- QS erfordert systematisches Vorgehen nach bewährten Methoden und Maßnahmen
- QS ist ein kontinuierlicher Prozess (begleitet alle Phasen)
- QS – Aktivitäten finden in jedem Vorgehensmodell Anwendung, sind jedoch unterschiedlich gelagert und integriert
  - Analyse: Prüfung von Anforderungen und Spezifikationsdokumenten
  - Entwurf: Prüfung von technischer Spezifikation, Ableitung von Testfällen
  - Implementierung: Analyse von Source Code, Implementierung von Tests

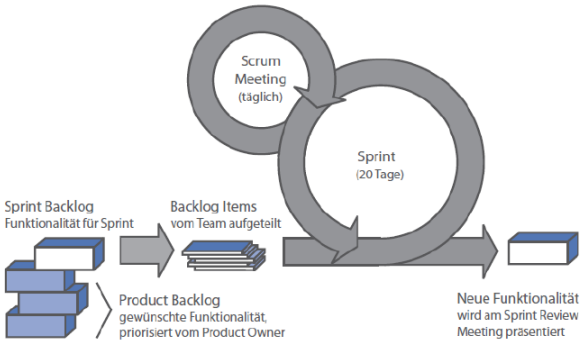
Überblick – QS im V-Modell

- Grundlage für modernere Modelle
- Jede *konstruktive* Phase der SW-Entwicklung (linke Seite) hat eine korrespondierende *prüfende* Phase der Qualitätssicherung (rechte Seite)
- Erstes Modell, das QS-Aktivitäten als Feedback-Schleifen integriert
- Miteinbezug von Verifikation und Validierung
- Risiko der Fehlentwicklung durch späte Validierung bleibt



Überblick – QS bei Scrum

- Repräsentant für agile SW-Entwicklung
  - Leichtgewichtiges Rahmengerüst
  - Iterative Entwicklung in Zyklen (Sprints)
- QS ist kein separater Prozess, sondern eng mit den Prinzipen und Praktiken der agilen SW-Entwicklung gekoppelt
- QS-Aktivitäten sind Teil des Sprints
- Teams sind oft cross-functional und selbstorganisiert



Reviews (VO 2)

Anforderungen und QS

- Nahezu die Hälfte aller Fehler entstehen in der Phase der Anforderungsanalyse.

Begriffsdefinition

Anforderungen sollten den SMART-Kriterien entsprechen

- **S**–Specific (präzise und eindeutig formuliert)
- **M**–Measurable (messbar, überprüfbar, verifizierbar)
- **A**–Achievable (erstrebenswert, erreichbar)
- **R**–Realistic (realisierbar)
- **T**–Time-Related (innerhalb definierter Zeitspanne erreichbar)

Klassifikation (WH aus VO1)
Funktionale Anforderungen (Funktionale Qualität)

- Beschreiben das Verhalten des Systems
- Eingaben, Verarbeitungsschritte, Ausgaben des Systems
- Verhältnismäßig „einfach“ zu ermitteln
- Definieren das „was?“

Nicht-funktionale Anforderungen (Strukturelle Qualität)

- Beschreiben unter welchen Bedingungen etwas bereitgestellt werden muss, große Breite
- Erstrecken sich oftmals quer über die gesamte funktionale Anforderungsbasis
- Wesentlich schwieriger zu ermitteln (Stakeholder-Awareness, Messbarkeit)
- Definieren das „wie gut?“

Gute Anforderungen (Bsp. VO2, S.10)

- Vollständigkeit: alle Aspekte sind definiert
- Konsistenz: es liegen keine Widersprüche und Konflikte vor

häufige Fehler:

- Fehlende Präzision (z.B. nicht messbar)
- Vermischung (funktionale und nicht funktionale Aspekte)
- Zusammenführung (z.B. fehlende Abgrenzung)
- Mehrdeutigkeit
- Selbstverständlichkeit

QS von Anforderungen

<u>Konstruktive Ansätze</u>

Nutzung von Methoden und Prozessen, um Entstehung von Fehlern bereits während der Anforderungserhebung zu vermeiden, z.B.

- Einsatz von Checklisten, Schablonen, Templates, …
- Workshops, Prototyping

<u>Analytische Ansätze</u>

Alle QS-Tätigkeiten, die auf bereits spezifizierte Anforderungen nachgelagert angewandt werden, z.B.

- Reviews
- Definition von Abnahmetestfällen

Reviews (Statische Qualitätssicherung)

Begriffsdefinition

- Systematische Überprüfung von Artefakten in einem SW-Projekt
- Verschiedene Ausprägungen, variieren hinsichtlich
    - Formalität (von informell bis formal)
    - Zweck (z.B. Fehlerfindung, Wissensverteilung, Prozessverbesserung)
    - Flexibilität
    - Rollen

Zielsetzungen

- Frühzeitige Identifikation von Fehlern, Widersprüchen, Redundanzen und Abweichungen
- Anerkennen von guten Lösungen
- Erhöhung der Produktivität
- Reduktion von Kosten
- Bessere Wissensverteilung im Team
- Gemeinsame Verantwortlichkeit (Collective Ownership)
- Kontinuierlicher Verbesserungsprozess

Anwendungsbereiche

- Grundsätzlich für alle Arbeitsergebnisse in einem Projekt
- komplexe Situationen zu erfassen und zu bewerten
- wenn keine Überprüfung durch Werkzeuge/Tools erfolgen kann oder als Ergänzung dazu

Bsp.: Anforderungen und User Stories, Technische Spezifikationen und Entwürfe, Code, Testdokumente

Häufige Fehlertypen (Mehr: VO2, S.17)

- Fehler in Anforderungen (z.B. Abweichungen, Widersprüche, Lücken, etc.)
- Fehler im Entwurf/Design (z.B. Abweichung von Anf., Sicherheitslücken, Schlechte Modularisierung, etc.)
- Abweichungen von Standards, Normen und Konventionen

Erfolgsfaktoren

- Objektivität und sachliche Kommunikation
- Festlegung klarer Ziele
- Definition messbarer Bewertungskriterien
- Festhaltung und Nachvollziehbarkeit der Ergebnisse
- Auswahl des passenden Review-Typs
- Unterstützung und Akzeptanz in der Unternehmenskultur

Review Typen

Traditionell

- Resultierend aus Forschung der 1980/1990er
- Formalisiert, schwergewichtig, meetingzentriert
- Definierte Rollen und Prozesse
- Lesetechniken als Performance-Booster
- In Einklang mit sequenziellen Vorgehensmodellen

Modern

- Getrieben aus Praxis und Industrie
- Leichtgewichtig, toolgestützt, kontinuierlich stattfindend
- Erlauben Asynchronität und verteilte Teams
- In Einklang mit agilen Praktiken
- Insb. für Source-Code-Reviews

Grundlage für moderne Reviews

Inspektion	Technisches Review	Walk-Through	Peer Review
sehr formal	formal	wenig formal	informell
Identifikation von Fehlern und Anomalien Prüfung auf Erfüllung der Spezifikation Prüfung von Anforderungen	Identifikation von Fehlern und Anomalien Prüfung techn. Artefakte auf Konformität mit der Spezifikation Prüfung der Einhaltung von Standards, Regulationen, Guidelines	Identifikation von Fehlern und Anomalien Verbesserung der Qualität Erwägen von Alternativen Wissensverteilung (Ausbildung, Einschulung)	Identifikation von Fehlern und Anomalien Verbesserung der Qualität Erwägen von Alternativen Verschiedene Ausprägungen möglich
Formaler Prozess Definierte Rollen Formaler Bericht Einsatz fachlicher Experten	Formaler Prozess Definierte Rollen Einsatz technisch qualifizierter Experten	Keine generellen Vorgaben hinsichtl. Prozess und Rollen, meist nur Autor und Gutachter Oft in kleineren Teams etabliert	Keine generellen Vorgaben hinsichtl. Prozess und Rollen, meist nur Autor und Gutachter Ad-hoc Selbstorganisiert

Formale Reviews – Rollen

- Rollen und Verteilung variieren je nach Review Art
- Einsatz von Werkzeugen kann Rollen überflüssig machen

Autor

- Urheber des Review-Objekts
- Unterstützt bei Unklarheiten und führt Korrekturen und Verbesserungen durch

Moderator

- Organisiert und koordiniert das Review
- Kommunikations- und Vermittlungsinstanz zw. den beteiligten Personen
- Meist Qualitätsmanager oder QS-Experte

Gutachter

- Führt das Review durch und präsentiert die Ergebnisse
- Meist Fachexperten (techn. / fachl.), z.B. andere Entwickler, Tester, Analysten, Betreiber

Manager

- Obliegt die Freigabe des Review-Objekts
- Meist leitende Position, Projektleiter, Testleiter, Entwicklungsleiter

Protokollant

- Dokumentiert und protokolliert das Review-Meeting
- Hält alle wichtigen Erkenntnisse und Entscheidungen fest

Formale Reviews – Prozess

1. Planung
    a. Definition der **Arbeitspakete** und **Review** Inhalte
    b. Erarbeitung von **Checklisten**, **Richtlinien**, **Ziele** etc.
    c. Festlegung des **Teams** und der benötigten **Rollen**
2. Kick-Off
    a. Initiierung des Reviews nach Erfüllung der Eingangskriterien
    b. Verteilung an das Team,
    c. Festlegung der Ziele und Aufgaben
3. Vorbereitung
    a. Durchführung des Reviews (asynchron), intensive Durcharbeitung
    b. Dokumentation von Fehlern, Rückfragen und Kommentaren
    c. Nutzung der bereitgestellten Checklisten etc.
4. Meeting
    a. Präsentation der Review Ergebnisse
    b. Besprechung und Bewertung der identifizierten Fehler
    c. Erarbeitung von Empfehlungen und Verbesserungsvorschlägen
5. Nacharbeit
    a. Behebung der identifizierten Mängel
    b. Umsetzung von Verbesserungsvorschlägen
6. Follow-Up
    a. Nachkontrolle der durchgeführten Arbeiten
    b. Freigabe des Reviewobjekts (oder Einberufung neues Reviews)
    c. Sammlung von Metriken/Lessons-Learned (kontinuierlicher Verbesserungsprozess)

Lesetechniken

- Hilfestellung, wie beim Review vorgegangen werden soll
- Soll die Effizienz steigern (Fehlerrate, Fokus, Dauer, …)

Beispiele: (Bsp. VO2, S.29)

- Ad Hoc Ansatz
  - Wenig bis keine Anleitung
  - Erfordert wenig Vorbereitung
  - Outcome stark vom Skillset und Sichtweise des Reviewer abhängig
- Checklisten-basierter Ansatz
  - Review basierend auf vorgegebenen Checklisten
  - Set an Ja/Nein Fragen, die auf potenzielle Fehlerkategorien abzielen
- Perspektiven-basierter Ansatz
  - Betrachten des Review Objekts aus der Sicht unterschiedlicher Stakeholder (z.B. Kunde, Entwickler, Tester)
  - Ermöglicht Auffinden verschiedener Fehlertypen

Moderne Reviews

- Weiterentwicklung bzw. Ausprägung von Peer Reviews
- Einsatzgebiet besonders bei Source Code Reviews
- Zentrale Eigenschaften
  - Werkzeuggestützt
  - Informell
  - Asynchron
  - Kontinuierlich
  - Änderungsbasiert
- Nutzen Funktionen von Versionskontrollsystemen
  - Diff-View
  - Merge/Pull Requests

Moderne Reviews –Evolution (Vergleich)

	traditionelle, formale Ansätze	**moderne, leichtgewichtige Ansätze**
Durchlaufzeit	Mehrere Wochen	Tage/Stunden
Anzahl Reviews	Wenige zu fixende Meilensteine	häufig und kontinuierlich
Anzahl involvierten	5-30 (!) Personen	1-2 Reviewer
Charakter	Prüfung eines gesamten Artefakts	änderungsbasierte Reviews mit kleinem Umfang

Moderne Reviews – Ausprägungen

Pre-Commit („review-then-commit“)

- Review erfolgt, bevor Änderungen in die gemeinsame Code Basis (z.B. develop/main-Branch) kommen
- Forciert einen Review-Prozess
- Kurze Zeitspanne zw. Entwicklung und Review
- Hohe Motivation zur Behebung, da es Abschluss blockiert
- Nachteil: kann entwicklungsverzögernd wirken

Ohne Toolsupport

- Walkthrough der Änderungen
- Export und Übermittlung eines Änderungs-Patches
- Sonderform: Pair-Programming

Mit Toolsupport

- Nutzung von Source-Code-Management-Plattform (z.B. GitLab…)
- Spezielle Pre-Commit-Reviewtools, z.B. Gerrit Code Review

Post-Commit („commit-then-review“)

- Review erfolgt, nachdem Änderungen in der gemeinsamen Code Basis sind
- Ermöglicht schnellere Integration
- Nachteil: schlechte Qualität muss nachträglich bereinigt werden, Reviews können bei Zeitdruck „wegdiskutiert“ werden

Software Testen Grundlagen (VO 3)

Definition: ausführen einen Programms, um Fehler zu finden

Verifikation & Validierung (V&V): Überprüft, ob ein Software System der Spezifikation entspricht und den vorgesehenen Zweck erfüllt

- Verifikation
  - „Bauen wir das Produkt richtig?“
  - „Entspricht es den Spezifikationen?“
- Validierung
  - „Bauen wir das richtige Produkt?“
  - „ist das Produkt für seine Zwecke geeignet?“

Ziele

- Fehler finden
- Fehler verhindern
- Vertrauen bzgl. der Qualität der Software zu erhalten
- Informationen für weitere Entscheidungen bereitzustellen

Testen ist ein konstruktiver Prozess

- „Zerstört“ nicht das Produkt
- Ein gefundener Fehler ist keine Kritik

- Kommunikation muss konstruktiv bleiben
- Während des Testens gefundene Fehler sparen Zeit und Geld

Potenzielle Fehlerquellen

- Testen ist nur ein Teil der Qualitätssicherung
- ISTQB Testprinzipien zeigen
    - Vollständiges Testen ist nicht möglich
    - Testen zeigt die Anwesenheit von Fehlern, nicht deren Abwesenheit
- Fehler in den Anforderungen
- Fehler im Design
- Fehlerhafte Testdaten

Grundsätze des Testens:

ISTQB Testprinzipien:

1. Testen zeigt Fehler auf
    a. Beweist aber nicht, dass keine Fehler mehr vorhanden sind
2. Vollständiges Testen ist nicht möglich
    a. Sämtliche Kombinationen von Eingaben, Vorbedingungen und Zuständen zu testen ist nicht umsetzbar/praktikabel
    b. Stattdessen wird der Testfokus mittels Risiken und Prioritäten festgelegt
3. Frühzeitig Testen
    a. Testaktivitäten sollten so früh wie möglich begonnen werden
    b. Frühzeitig gefundene Fehler sind einfach und günstig zu beheben
4. Fehlerhäufungen beachten
    a. Fehler sind nicht gleichmäßig verteilt. Werden in einem Modul einige Fehler gefunden, sind dort meist weitere zu erwarten
5. Veränderung statt Wiederholung
    a. Das wiederholte Ausführen von Testfällen wird keine neuen Fehler identifizieren
6. Testen ist kontextabhängig
7. Trugschluss: Fehlerlose Systeme sind brauchbar
    a. Fehler zu finden und zu beheben, garantiert nicht, dass das Produkt den Anforderungen und Bedürfnissen des Benutzers entspricht

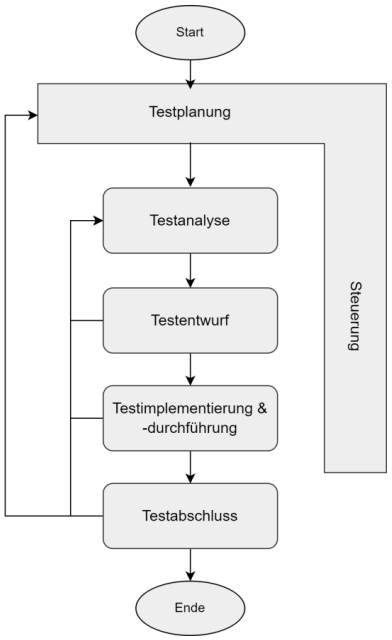
Testaktivitäten

Testaktivitäten existieren vor und nach der Testausführung

Testaktivitäten umfassen

- Testplanung
- Auswahl der Testbedingungen
- Testüberwachung / -steuerung
- Entwurf / Ausführung von Testfällen
- Überprüfen der Ergebnisse
- Bewertung von Ausstiegskriterien
- Reporting über den Testprozess und den Status des zu testenden Systems
- Abschlussaktivitäten nach einer Testphase

ISTQB fundamentaler Test Prozess



- Testen ist ein Prozess, keine einmalige Aktivität
- Testausführung ist der sichtbarste Teil des Testens
- Auch Tests müssen geplant, vorbereitet und ausgewertet werden
- Aktivitäten können überlappend/gleichzeitig stattfinden
- Testprozess an Anforderung des Projekts anpassen

Teststufen:

Teststufen – Komponententests

- Auch Modul- oder Unit-Tests
- Testen einer einzelnen Komponente auf korrekte Funktionalität
- Testen einer Komponente isoliert vom Rest des Systems
- Erfordert Zugang zum Source Code
- Von Entwicklern erstellt und ausgeführt

Teststufen – Integrationstests

- Testen, um Fehlerzustände in den Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten aufzudecken
- Zeigt Fehler in Interfaces und Interaktionen zwischen Modulen auf
- Fokus auf Integration mehrerer Komponenten
- Von Entwicklern erstellt und ausgeführt

Teststufen –Systemtests

- Testen eines **integrierten Systems**, um sicherzustellen, dass es spezifizierte Anforderungen erfüllt.
- Testumgebung sollte ein **Abbild der Produktivumgebung** sein, um das Risiko umgebungsspezifischer Fehler zu minimieren.
- Von unabhängigem Testteam ausgeführt

Teststufen –Akzeptanztests

- Auch Abnahmetest oder Kundenakzeptanztest
- Testet, ob das System bereit für den **produktiven Einsatz** ist und den **Abnahmekriterien des Kunden** entspricht
- Dient nicht dazu, Fehler im System zu finden
- Vom Kunden/Benutzer durchgeführt

Testentwurfsverfahren

Black Box Methoden

- Basieren auf Spezifikation
- Kein Wissen über innere Struktur vorhanden
- Daten-getrieben
    - Variationen von Eingabeparametern
    - Überprüfung des Ergebnisses mit einem erwarteten Wert
- Anforderungsüberdeckung

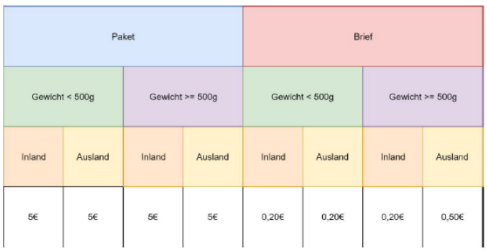
Methoden:

- Äquivalenzklassen
- Grenzwerte
- Entscheidungstabellen
- Anwendungsfall-basiertes Testen
- Zustandsbasiertes Testen

Klasse 1: Typ
•A1 Paket
•A2 Brief

Klasse 2: Gewicht
•B1 Gewicht < 500g
•B2 Gewicht >= 500g

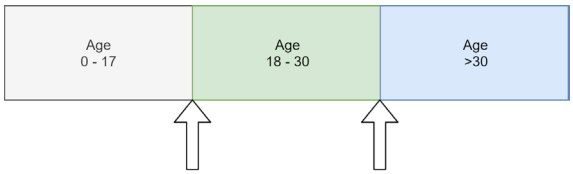
Klasse 3: Zustellungsort
•C1: Inland
•C2: Ausland



Äquivalenzklassen (Bsp. VO3, S.23)

- Einteilung möglicher Eingabewerte in Klassen/Partitionen (Äquivalenzklassen)
- Alle Werte einer Klasse/Partition führen zu demselben erwarteten Ergebnis
- Äquivalenzklassen für gültige und ungültige Werte
- Jeder Wert kann nur zu einer Äquivalenzklasse gehören
- Es ist i.d.R. ausreichend, einen repräsentativen Wert pro Klasse auszuwählen
- Verwendung in Kombination mit der Grenzwertanalyse

Grenzwertanalyse



- Erweiterung der Äquivalenzklassenbildung
- Jeweils das Minimum und Maximum einer Äquivalenzklasse sind die Grenzwerte
- Das **Verhalten** an den **Grenzen** von Äquivalenzbereichen ist mit größerer Wahrscheinlichkeit **anders** als das Verhalten **innerhalb** der Bereiche.

Entscheidungsbasierte Tests

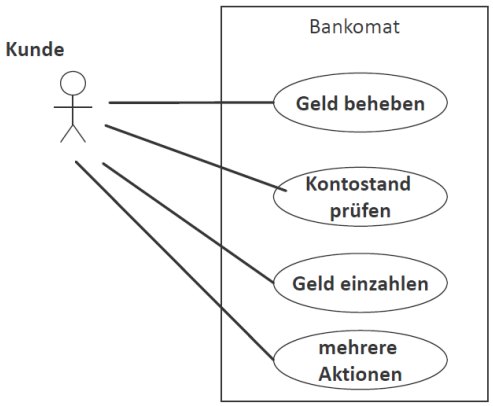
- Gute Möglichkeit komplexe Regeln abzudecken
- Abhängigkeiten zwischen Eingabewerten werden bei Grenzwertanalysen oder Äquivalenzklassen nicht berücksichtigt
- Vollständiges Testen ist nicht möglich
    - Entscheidungstabellen sollen helfen, die Menge aller möglichen Kombinationen zu reduzieren
- Ursache-Wirkungs-Graph (Bsp. VO3, S.27)
    - grafische Darstellung, die Zusammenhänge zwischen Eingaben (Ursachen) und Ausgaben (Wirkung) darstellt
- Entscheidungstabellen (Bsp. VO3, S.29)
    - Tabelle, die zur Darstellung von Bedingungen (Eingaben) und den daraus resultierenden Aktionen (Ausgaben) verwendet wird

Zustandsübergangstest (Bsp. VO3, S.34)

- Darstellung des Verhaltens eines Systems durch Zustände
- Zeigt i.d.R. nur gültige Übergänge
- Jeder Pfad entspricht einem Testfall
- Verwendet für:
    - Menübasierte Anwendungen
    - Prozessabläufe (z.B. BPMN)

Anwendungsfallbasiertes Testen

- Testfälle werden aus Anwendungsfällen abgeleitet
- Jeder Anwendungsfall beschreibt ein Szenario der Systeminteraktion
- Anwendungsfälle enthalten:
    - Vorbedingungen
    - Erwartete Ergebnisse
    - Nachbedingungen
- Abdeckung abgedeckte AF / Anzahl AF



White Box Methoden (Kontrollflussorientierte Testverfahren)

Auch White Box / Überdeckungstests

Es handelt sich um strukturorientierte Testmethoden

- Basieren auf Analyse des Quellcodes
- Orientieren sich am Kontrollflussgraphen des Programms
- Relevant auf Unit Test Ebene
- Definieren keine Regeln für die Erzeugung von Testdaten
- Dienen der Ermittlung der Testabdeckung

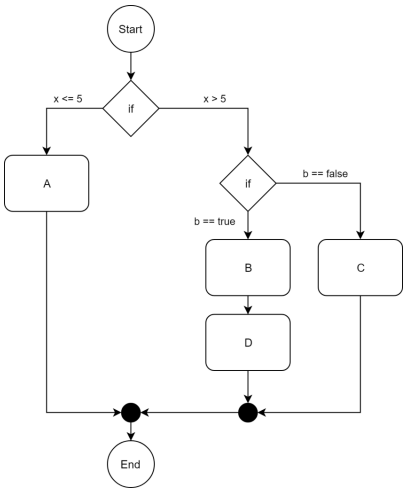
Testarten:

- Anweisungsüberdeckung
- Zweigüberdeckungstest
- Pfadüberdeckungstest
- Bedingungsüberdeckungstest

Kontrollflussgraph

- Gerichteter Graph
- Beschreibt Kontrollfluss eines Programms
- Dient zur Programmoptimierung

```
if(x <= 5) {
    A();
}
else {
    if(b) {
        B();
        D();
    }
    else {
        C();
    }
}
```



Anweisungsüberdeckung (Bsp. VO3, S.40)

- Statement Coverage
- Ziel: jede Anweisung muss mind. einmal durchlaufen werden
- Vollständige Anweisungsüberdeckung liegt vor, wenn sämtliche Anweisungen mindestens einmal durchlaufen werden
    - Zeigt ob toter Code existiert
        - Anweisungen, die niemals durchlaufen werden können
- Zu schwaches Kriterium für sinnvolle Testdurchführung

Zweigüberdeckung (Bsp. VO3, S.42)

- Branch Coverage
- Umfasst Anweisungsüberdeckung vollständig
- Ziel: Jede Kante muss mind. einmal durchlaufen werden
    - Zeigt nicht ausführbare Programmzweige auf
    - Hilft oft durchlaufene Programmteile gezielt zu optimieren
- Im Gegensatz zum Anweisungsüberdeckungstest muss jede Entscheidung mindestens einmal true und false annehmen
- Problematik:
    - Unzureichender Test von Schleifen
    - Komplexe Logik in Statements wird nicht berücksichtigt

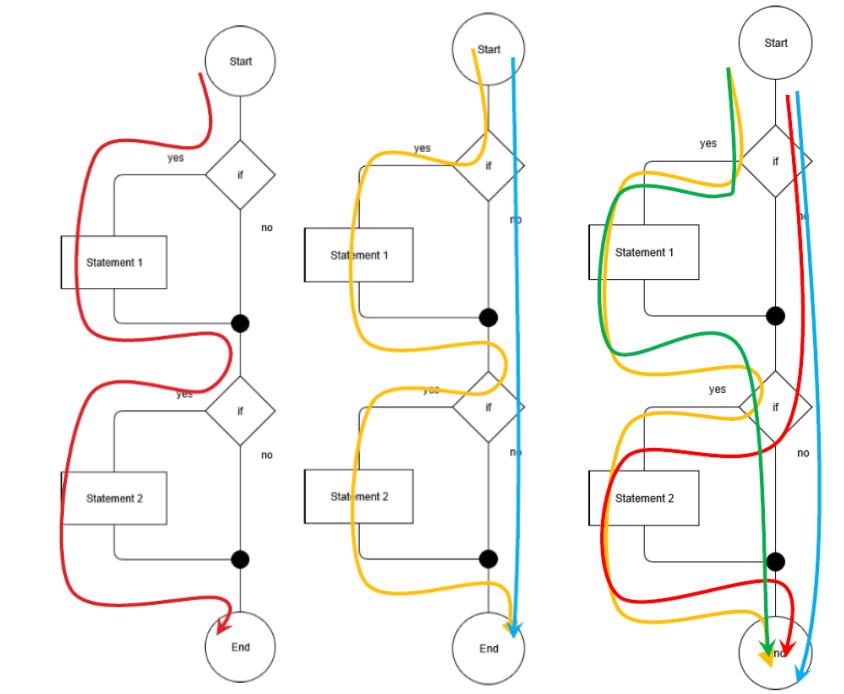
Pfadüberdeckung (Bsp. VO3, S.44)

- Path Coverage
- Ziel: Abdeckung aller Pfade von Start-bis Endknoten
- Für komplexe Softwaremodule meist nicht durchführbar
    - unendlich hohe Anzahl von Pfaden
    - Schleifen bieten extrem viele Pfade

Bedingungsüberdeckung (Bsp. VO3, S.46)

- Condition Coverage
- Ziel: Überprüfung zusammengesetzter Entscheidungen, Teilentscheidungen

- Einfacher Bedingungsüberdeckungstest
    - Fordert den Test aller atomaren Teilentscheidungen gegen true und false
- Mehrfach-Bedingungsüberdeckungstest
    - Fordert den Test aller Wahrheitswertekombinationen der atomaren Teilentscheidungen
    - Sehr aufwändig

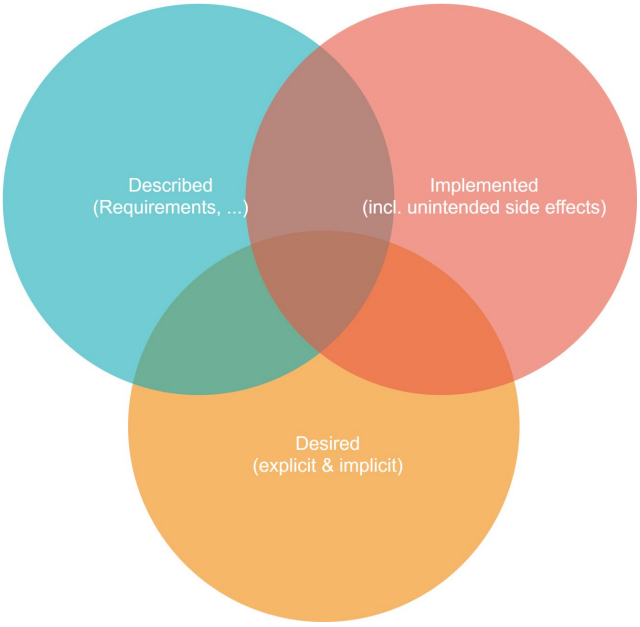


Anweisungsüberdeckung	Zweigüberdeckung od Einfach Bedingungsüberdeckung	Pfadüberdeckung od. Mehrfach. Bedingungsüberdeckung

Überdeckung

- Quantitative Metrik vs. qualitative Methodik
- Coverage allein ist kein ausreichendes Kriterium

Erfahrungsbasiertes Testen

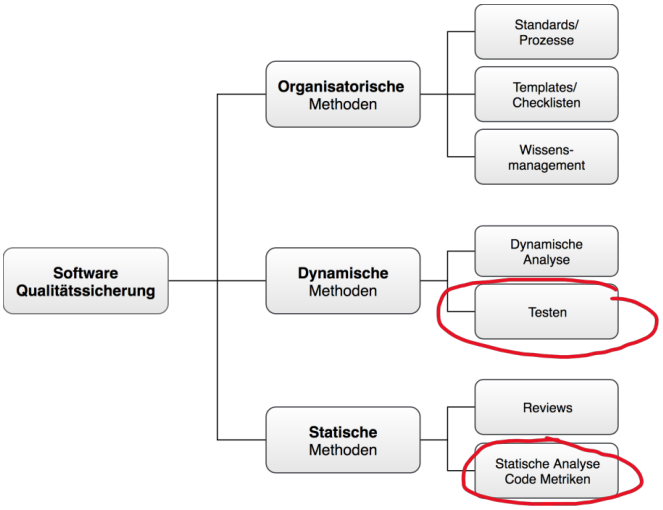


- Auch **Exploratives Testen**
- können mit verschiedenen Testentwurfsverfahren kombiniert werden
- Testfälle basieren auf Intuition und Erfahrung des Testers
    - Wo könnte es Probleme geben?
    - Wo waren Fehler in der Vergangenheit? (zB in anderen Projekten)
    - Welche Komponenten sind unter Zeitdruck entstanden?
- Erfahrungsbasiertes Testen muss dokumentiert werden

desired = gewünscht

Einführung in Software Testen (VO 4)

Klassifikation Qualitätssicherung



Qualitätsmetriken

- Metriken schaffen ein gemeinsames Verständnis von Qualitätsaspekten
- Metriken **quantifizieren** verschiedene Aspekte der Qualität
- Werden von Tools berechnet, die in die Entwicklungs-/Erstellungsumgebung integriert sind

- Aber:
    - Metriken müssen mit Sorgfalt / im richtigen Kontext interpretiert werden

Metriken dienen unterschiedlichen Zwecken:

- Verstehen des Status Quo (gegenwertigen Zustand) eines Projekts
- Verfolgen von Trends im Laufe der Zeit
- Vergleich von Status und Trends zwischen ähnlichen Projekten/Portfolios
- Unterstützung von Managemententscheidungen
- Ermöglichung korrigierender Maßnahmen

Beispiele:

- Lines of Code (LOC)
- Number of Statemens (NOS)
- Zyklomatische Komplexität
- Kognitive Komplexität
- Testabdeckung (Coverage)
- Anzahl gefundener Fehler
- Technische Schuld
- …

Statische Qualitätssicherung

Statische Analysen

- Statische Analysen werden durch Tools unterstützt
- Tools zur Überprüfung von Sourcecode Konventionen
    - Checkstyle
- Tools zur Überprüfung der Testabdeckung
    - JaCoCo, Clover und Cobertura
- Tools zum Finden von Fehlermustern
    - FindBugs/SpotBugs, SonarLint und SonarQube
- Frameworks für die Analyse der Systemarchitektur
    - ArchUnit

Fehlermuster (FindBugs/SpotBugs, SonarLint, SonarQube und ArchUnit → Bsp. VO4, S.11)

Statische Code Analyse ermöglicht die Identifikation häufiger Fehlermuster im Code:

- Variablen mit undefiniertem Wert
- Komplexe Konstrukte
- Toter Code
- Potenzielle Endlosschleifen
- Security Schwachstellen
- Unused Code
- …

Vorteile:

- Verbesserung der Code Qualität (z.B. Wartbarkeit)
- Auffinden von Fehlern
- Vermeidung/Reduktion zukünftiger Fehler

Sourcecode Qualitätskriterien

Coding Conventions (Checkstyle → Bsp. VO4, S.19)

- Einheitlicher Stil innerhalb eines Projekts
    - Variierender Stil erschwert Lesbarkeit/Verständlichkeit des Codes
- Definieren Konventionen/Guidelines für eine spezifische Programmiersprache

Code Guidelines:

- Einrückung (space/tab)
- Max. Zeilenlänge
- Import Reihenfolge
- Methoden / Feld Reihenfolge in Klassen
    - Basierend auf Modifiern
    - Basierend auf Sichtbarkeit/Veränderbarkeit
- System.out Vermeidung
- Logging
- Namenskonventionen
- Code Formatierung

Commit Conventions

Commit Messages dienen zumindest drei wichtigen Aspekten:

- Beschleunigen den Review-Prozess
- Unterstützen das Erstellen von Release Notes
- Erleichtern zukünftige Wartung

Commit Message – Best Practice

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Gute Commit Messages: (Bsp. VO4, S.25)

- Warum wurde diese Änderung gemacht?
- Was wurde geändert?
- Enthalten nur eine logische Änderung pro Commit
- Idempotent (in sich geschlossen)
  - Zwingen Sie niemanden dazu externe Tools (z.B. Bugtracking Tools, …) zu benutzen, um einen Commit zu verstehen

Conventional Commits:

- Basierend auf den Angular Konventionen
- IntelliJ Plugin

Vorteile:

- Changelogs
- Nachvollziehbarkeit von Änderungen
- Erhöht Wartbarkeit
- Vereinfacht die Zusammenarbeit/Beteiligung

Dynamische Qualitätssicherung

Unit Tests

Testframework

- Komponententests werden i.d.R. automatisiert ausgeführt
- Implementierung wird mittels Unit Test Frameworks umgesetzt

xUnit Frameworks

- Überbegriff für Unit-Test-Frameworks in verschiedenen Programmiersprachen mit ähnlichem Design

Testframework –Benennung (Bsp. VO4, S.31)

Benennungsschema für Testmethoden

- Einheitliches Benennungsschema erhöht
  - Lesbarkeit
  - Nachvollziehbarkeit

Zusätzliche Informationen, falls notwendig:

- Spezielle Inputs
- Spezielle Zustände
- …

JUnit

Lifecycle Annotations

- @BeforeAll
- @AfterAll
- @BeforeEach
- @AfterEach
- …

JUnit–Struktur eines Tests

Aufbau eines (Unit) Tests

- Given / When/ Then
- Arrange/ Act / Assert

Unterteilen des Tests in klare Abschnitte für

- Vorbedingung / Aufbau
- Ausführung
- Validierung

Vorteil:

- Erhöhte Lesbarkeit / Nachvollziehbarkeit

Nachteil:

- Nicht immer klar anwendbar

JUnit–Assertions

- **Zusicherungen** zur Überprüfung, ob das zu testende Objekt den **korrekten Zustand** besitzt
- Statische Methoden der Klasse org.junit.jupiter.api.Assertions;
- Können für erhöhte Lesbarkeit mittels statischem Import importiert werden

AssertJ (Vergleich VO4, S.37)

- Zusätzliches Assertion-Framework mit FluentAPI
- Verbessert Lesbarkeit und Nachvollziehbarkeit

Grouped Assertions/ Soft Assertions

```
assertAll(
    () -> assertTrue(. . .),
    () -> assertEquals(. . .)
)
```

- Gruppierungen von zusammenhängenden Assertions
- Fehlermeldungen werden gesammelt ausgewertet

JUnit–Testen von Exceptions

- Tests sollen nicht nur positive Fälle, sondern auch Fehlerfälle abdecken

JUnit–Parametrisierte Tests

- Tests werden mehrmals mit unterschiedlichen Testdaten ausgeführt
- Benötigt eigene Dependency
    - junit-jupiter-params
- Benutzt die Annotation **@ParameterizedTest**

Unterschiedliche Quelle der Testdaten (Bsp. VO4, S.41)

- @ValueSource(ints = { 1, 2, 3 })
- @EnumSource(TimeUnit.class)
- @MethodSource("stringProvider")
- @CsvFileSource(resources = "/data.csv")

JUnit–Extensions (Bsp. VO4, S.44)

Erweiterungspunkte für Unit Tests

- org.junit.jupiter.api.extension
    - TestWatcher
    - BeforeTestExecutionCallback
    - AfterTestExecutionCallback
    - ParameterResolver

Annotation der Tests mit

- @ExtensWith(…)

Ermöglicht eine zentrale Behandlung von Erweiterungen

- z.B. Logging der Zeit pro Testfall
- Tasks basierend auf Testergebnissen
- Tasks die vor/nach einem Test ausgeführt werden sollen

JUnit–ConditionalExecution

Bedingungen für Testausführung, z.B.:

Beispiel:

- @Enabled/Disabled OnOS(WINDOWS)
- @Enabled/Disabled ifSystemProperty(named= "file.separator", matches= "[/]")

JUnit–Best Practices

- Annotation Wrapping (Bsp. VO4, S.49)
    - Erstellen von Meta-Annotationen, um häufig genutzte Annotationen zusammenzufassen
    - Erhöht die Verständlichkeit und Wartbarkeit
- Testfälle sind isoliert
    - Jeder Testfall testet nur einen Aspekt bzw. Zustand
    - Keine Abhängigkeiten zwischen Testfällen
    - Jeder Testfall bereitet seine Daten vor und räumt sie auf
- Einhaltung von Namenskonventionen
- Klare Fehlermeldungen bei Assertions

- Tests sind Code –auf Lesbarkeit achten
    - Einhaltung von Code Guidelines / Refactorings etc.
- Klassen und Testklassen liegen im selben Package

JUnit–Bad Practices

- Test testet nicht das gewünschte Verhalten
    - Test erfolgreich, unabhängig vom Ergebnis
- Test testet mehrere Zustände gleichzeitig
    - Mehrere Testfälle in einer Testmethode
- Zufällige Logik
    - Testdaten aus Zufallsdaten anstatt Äquivalenzklassen
    - Testdaten hardcoded im Test statt in sprechenden Variablen
- Überprüfung der Vorbedingungen
    - z.B. Überprüfung der in @BeforeEach ausgeführten Aktionen
- Logik in Tests (Abfragen, Schleifen, try/catch)

Test-Driven-Development (TDD) (Bsp. VO4, S.53)

RED

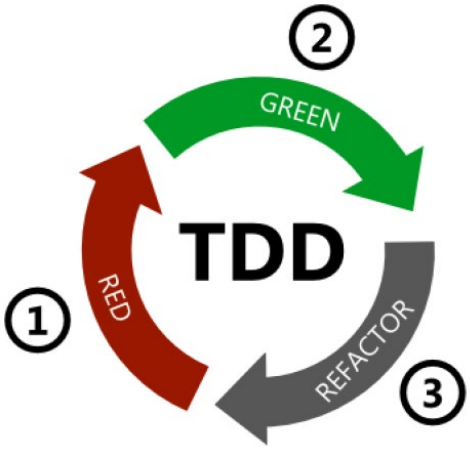
- Implementierung und Ausführung der Testfälle
- Tests müssen fehlschlagen

GREEN

- Implementierung der Funktionalität
- Testfall erfolgreich

REFACTOR

- Optimierung der Implementierung
- Keine Änderung des Verhaltens
- Testfälle dürfen nicht fehlschlagen



Testabdeckung

- CoverageReports

Einführung in Software Testen II (VO 5)

Probleme bei Komponententests

- Interagieren mit anderen Komponenten
- Rufen externe Schnittstellen auf
- Verwenden nicht deterministische Werte (z.B. Systemzeit)
- Interagieren mit einer Datenbank

Wie können diese Abhängigkeiten ausgeblendet werden?

Test Doubles

Generischer Begriff für den Austausch einer Komponente des Systems durch eine alternative, meist simplifizierte Implementierung für Testzwecke

Isolation

- Isoliertes Testen einzelner Komponenten
- Bessere Steuerung der zu testenden Komponente
- Reduktion von Abhängigkeiten zu anderen System(teil)en

Effizienz

- Reduktion von Komplexität
- Reduktion der Ausführungszeit
- Reduktion von Kosten/Service-Gebühren

Flexibilität

- Leichteres Testen von Randfällen und Fehlerfällen
- Kontrolle nicht deterministischer Werte (z.B. Datum/Zeit)

Test Doubles –Typen

Dummy Objekt (Bsp. VO5, S.8)

- „Leerer“ Platzhalter ohne Funktionalität
- Wird in die Komponente gereicht, jedoch nicht zur Verwendung intentioniert

Fake Objekt

- Ausführbare Implementierung
- Dient jedoch nicht zur Steuerung/Überprüfung des Testobjekts

Einsatzzwecke:

- Reale Implementierung zu langsam
- Reale Implementierung nicht (bzw. nicht in Testumgebung) verfügbar

Beispiele: Simulation einer Datenquelle, In-Memory Datenbank

Spy

- Kein Ersetzen einer Komponente durch eine einfachere Variante
- Erweiterung einer Komponente um „Überwachungsfunktionen“
    - Interaktionen zw. Testobjekt und abhängiger Komponente werden überwacht
    - Aufrufe werden jedoch an die reale Komponente weiter delegiert
    - Vorteil: Verhalten der realen Komponente wird nicht beeinflusst→realitätsnäheres Testsetup)
    - Nachteil: Komplexität/Abhängigkeit bleibt bestehen

Ermöglicht: Prüfung aufgerufener Methoden, Prüfung der übergebenen Aufrufparameter, Reihenfolge des Aufrufs, Anzahl der Aufrufe

Stub (Bsp. VO5, S.12)

- Ausführbare Implementierung
- Liefert vordefinierte Werte/Exceptions an den Aufrufer
    - Responder-Stub (Gutfall, Lieferung valider Werte)
    - Saboteur-Stub (Fehlerfälle, Werfen von Fehlern/Exceptions)
- Ermöglicht indirekt die Steuerung der zu testenden Komponente
    - Steuerung, welcher Pfad durchlaufen wird
    - Erlaubt dadurch das Testen von Pfaden in der Komponente, die von außen nicht beeinflusst werden könnten

Mock (Bsp. VO5, S.16)

- Ausführbare Implementierung
- Liefert vordefinierte Werte/Exceptions an den Aufrufer
- Im Gegensatz zum Stub werden
    - die Interaktionen des Testobjekts mit dem Mock überwacht
    - die an den Mock übergebenen Parameter überprüft
    - Erwartungen an das Verhalten des Testobjekts geprüft
- Ein Mock ist somit selbst Teil des Tests
- Ein Mock erkennt, wenn unerwartete Werte eingehen
    - Direkte Verwendung von Assertions
    - Auslösen von Exceptions

Mocking Frameworks: vereinfachen die Verwendung von Mock Objekten für Tests

- Mockito, JMockit, etc.
- (Bsp. VO5, S.19)

Continuous Integration (CI)

- Ursprünglich im Kontext von Extreme Programming(XP) etabliert, mittlerweile Standard in agilen Projekten
- Entwicklungsansatz, bei dem Code Änderungen laufend integriert, getestet und gebaut werden
- Jede Integration wird durch einen automatisierten Buildprozess begleitet

Ausgestaltung

Grundprinzipien

- Häufige Commits („Commit Daily –Commit Often“)
- Häufige Builds („Build Early –Build Often“)
- Stabile Builds („Keep it Green“)

Ausgestaltung abhängig von

- Projektkontext
- Techn. Setup/Technologien
- Verfügbares Versionsverwaltungs- und CI-Tool
- Gewünschter Umfang, z.B.
    - Kompilieren / Bauen des Projekts
    - Ausführung automatisierter Tests (z.B. Unit-/Integration-/Systemtests)
    - Statische Analysen (z.B. Sonar, Checkstyle)
    - Security Analysen (z.B. Schwachstellenscan)

Vorteile

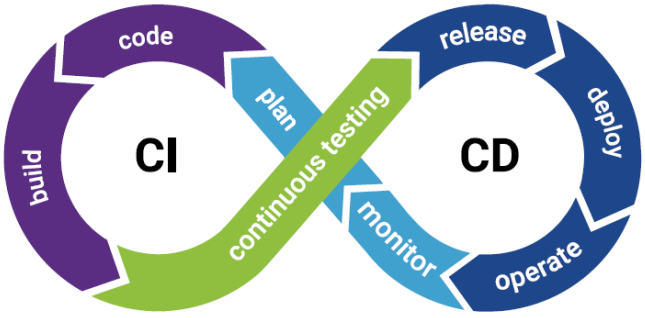
- Schnelles und häufiges Feedback
- Reduktion manueller Tätigkeiten
- Frühzeitige Identifikation von Fehlern
- Bessere Ausgangssituation für Refactorings
- Vermeidung einer späten „Integrationshölle“
- Vermeidung einer „works on my maschine“ Problematik
- Mehr Transparenz über Status und Qualitätsniveau
- Einblick in Projekt-Historie

Abgrenzung CI/CDE/CD

CI fokussiert auf den automatisierten Build der Software bis an den Punkt, wo ein integrierter, lauffähiger SW-Stand vorliegt

Darauf aufbauende Ansätze

- Continuous Delivery (CDE)
    - Automatisierung des Packaging-/Release-Prozesses
- Continuous Deployment (CD)
    - Automatisierung des Deployment-Prozesses
    - Automatisierte Bereitstellung neuer SW-Versionen auf Produktivumgebungen



Relevanz für DevOps

strikte Trennung zwischen Entwicklung und Betrieb

- Entwicklung zuständig für Bereitstellung des Produkts
- Betrieb zuständig für Deployment, Installation, Infrastruktur und Support

Problematischer Gap

- Fehlende Berücksichtigung betrieblicher Aspekte während Entwicklung
- Fehlendes Testing betrieblicher Aspekte
- Späte Identifikation von Blockern
- Fehlende Transparenz + fehlendes Know-How

DevOps fördert teamübergreifende Kooperation und reduziert Durchlaufzeit bis zur Bereitstellung der Software

- CI/CD ist ein wesentlicher Baustein

Beispiele CI-Server: Jenkins, Circle CI , GitLab CI/CD, GitHub Actions, etc.

Refactoring (Bsp. VO5, S41)

Systematischer Ansatz zur Steigerung der Code Qualität

Verbesserung der internen Struktur des Codes hinsichtlich

- Lesbarkeit
- Verständlichkeit
- Komplexität
- Architektur/Design

Keine Änderung des sichtbaren Verhaltens

- Keine neue Funktionalität
- Keine Behebung von Fehlern

Keine Änderungen, die nicht das Ziel haben, die Struktur des Codes zu verbessern

- z.B. Performanceverbesserungen

Technische Schuld

Metapher-Begriff, um nicht-techn. Stakeholdern Refactoring zu erklären:

- Schlechte Code Qualität = sich in Schuld begeben
- Schulden müssen (zeitnah) zurückgezahlt werden

Problematik –2 Aspekte

- Code „verwahrlost“ über die Zeit
- Entstehung schlechter Qualität wird als bewusster Kompromiss in Kauf genommen, um kurzfristig schneller liefern zu können (VO2, Teufelsquadrat)

Anhäufung von Qualitätsmängeln führt langfristig zu

- Verlust an Produktivität
- Verlust an Wartbarkeit

- Höheren Fehlerraten
- Explosion von Fehlerkosten

Refactoring–Motivation

- Verbesserung des Designs
- Erhöhung der Verständlichkeit
- Auffinden von Fehlern
    - Mehr Verständnis über den Code ermöglicht Fehler zu identifizieren

Refactoring–Vorgehensweise

1. Identifikation
2. Testabdeckung
    a. Kontrolle ob genug Tests für den Code das von Refactoring betroffenen ist (Ergänzen bei Bedarf)
3. Durchführung
    a. Schrittweise Verbesserung des Codes, regelmäßiges Ausführen der Tests zur Überprüfung, dass die Funktionalität nicht verändert wird

Refactoring – Bad Smells

Indikatoren für die Notwendigkeit von Refactoring

Unterteilung in Gruppen, z.B.

- Bloaters
    - Code, der über die Zeit hinweg erweitert ist
- Object-Orientation Abusers
    - Falsche oder unvollständige Nutzung objektorientierter Paradigmen
- Couplers
    - Zu starke Abhängigkeiten zwischen Klassen
- Change Preventers
    - Änderungen an einer Stelle haben Auswirkungen auf viele andere Stellen
- Despensables
    - Überflüssiger Code, negative Auswirkungen auf Lesbarkeit und Wartbarkeit

Refactoring–Patterns

Beschreiben allgemeine Lösungsvorschläge für wiederkehrende Probleme

Gruppe	Fokus	Beispiele
Composing Methods	Struktur/Schnitt von Methoden	• Extract Method • Extract Variable • Replace Temp with Query
Moving Features between Objects	Verschieben von Funktionalität zwischen Klassen	• Move Method • Move Field • Extract Class
Organizing Data	Verwaltung/Kapselung von Daten	• Encapsulate Field • Replace Magic Number
Simplifying Conditional Expressions	Vereinfachen von logischen Ausdrücken	• Decompose Conditional
Simplifying Method Calls	Vereinfachen von Methodenaufrufen	• Rename Method • Add Parameter
Dealing with Generalization	Erzeugen von Vererbungshierarchien Verschieben von Funktionalität zwischen Vererbungsstrukturen	• Pull Up Field • Pull Up Method • Extract Superclass

Agiles Testen (VO 6)

Grundlagen der agilen Software-Entwicklung

Herausforderungen in IT-Projekten

- Anforderungen und Ziele sind nicht immer klar
- Anforderungen können sich schnell ändern
- Anforderungen können sich spät im Projekt ändern
- Technologien ändern sich schnell
- Geringe Wiederverwendbarkeit zwischen Projekten
- Software-Entwicklung liegt zwischen Handwerk und Kunst

Traditionelles Projekt-Management

- Top-Down-Ansatz („command and control principle“)
- Projektmanager sind für das Projekt verantwortlich
- Der Kunde liefert die Anforderungen
- Push-Prinzip: Der Projektmanager weist die Tätigkeiten den Teammitgliedern zu
- Tests und QS werden erst spät im Projekt tätig
- Das Produkt wird erst zu einem späten Zeitpunkt an den Kunden geliefert
- Der Projektmanager versucht Unsicherheiten zu reduzieren und gleichzeitig die Vorhersagbarkeit zu erhöhen
- Oftmals organisatorische Trennung zwischen Entwicklung, Test und Betrieb
- Prozesse und Dokumentation dienen im Problemfall als Verteidigung

Manifest für Agile Softwareentwicklung

Es definiert zentrale Werte für die Softwareentwicklung:

-Individuen und Interaktionen mehr als Prozesse und Werkzeuge
-Funktionierende Software mehr als umfassende Dokumentation
-Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung
-Reagieren auf Veränderung mehr als das Befolgen eines Plans

„Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.“

12 Prinzipien des Agilen Manifests

1. Kundenzufriedenheit durch die frühe und kontinuierliche Auslieferung von wertschaffender Software
2. Änderungen an den Anforderungen sind selbst spät in der Entwicklung willkommen
3. Funktionierende Software wird regelmäßig innerhalb von Wochen/Monaten ausgeliefert (kürzere Intervalle sind bevorzugt)
4. Fachexperten und Entwickler müssen täglich zusammenarbeiten
5. Das Team benötigt eine entsprechende Umgebung und Unterstützung, sowie Vertrauen, dass sie die Arbeit erledigen
6. Die effizienteste und effektivste Methode für die Weitergabe von Informationen ist direkte und persönliche Kommunikation
7. Der Projektfortschritt wird anhand der funktionierenden Software gemessen
8. Durch eine konstante Arbeitsgeschwindigkeit soll eine nachhaltige Entwicklung erreicht werden

9. Durch Einfachheit wird die Menge nicht-getaner Arbeit maximiert (unnötige Arbeit vermieden)
10. Technische Exzellenz und gutes Design erfordern kontinuierliche Aufmerksamkeit
11. Selbstorganisierte Teams liefern die besten Anforderungen, Architekturen und Entwürfe
12. Das Team reflektiert in regelmäßigen Abständen, um effektiver zu werden

Agile Vorgehensmodelle

Vorgehensmodelle für einzelne Teams

- Extreme Programming
- Scrum
- Kanban
- Crystal Family
- Lean Software Development

Vorgehensmodelle für mehrere Teams

- Scrum of Scrums
- Nexus Framework
- Scaled Agile Framework (SAFe)
- Large Scale Scrum (LeSS)
- Scaling Agile @ Spotify

Extreme Programming(XP)

- Definiert 5 Werte und leitet davon 14 Prinzipien ab, die wiederum die Basis für 23 Hauptpraktiken und 11 davon abgeleiteten Praktiken bilden
- Praktiken haben sich teilweise bereits vorher bewährt
- Wesentlicher Einfluss auf das agile Manifest

Praktiken:

- Einbeziehung des Kunden
- Iterative Entwicklung (Wöchentliche & quartalsmäßige Zyklen)
- Requirements als Stories
- Einzelne Code-Basis
- Pair Programming
- Inkrementelles Design
- Test-First Programming
- Continuous Integration

Scrum

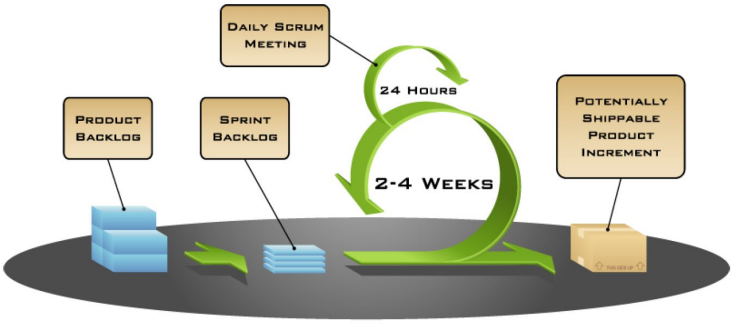
- Iterativer Prozess unterteilt in Sprints mit fixer Zeitdauer („timeboxed“)
- Mittels „Definition of Done“ wird festgelegt, welches Maß an Qualität benötigt wird, um etwas abzuschließen

Rollen:

- Product Owner
    - Schnittstelle zwischen Kunden & Team
    - Sammelt und priorisiert Anforderungen (User Stories)
    - Fungiert als Ansprechpartner für fachliche Fragen

- Scrum Master
    - Sorgt für die Einhaltung des Prozesses
    - Moderiert die Meetings, trifft aber keine Entscheidungen
    - Beseitigt Blockaden (eng. Impediments) für das Team
- Team Member
    - Schätzt Anforderungen
    - Arbeitet am Projekt

Prozessmodell



Sprints

- Fixe Timebox von 1 – 4 Wochen mit definiertem Umfang
- Beginnt mit Sprint Planning Meeting, bei dem die umzusetzenden Items geschätzt und festgelegt werden
- Daily Scrum Meeting zur internen Abstimmung
- Endet mit
    - Sprint Review Meeting: Ergebnis des Sprints wird Stakeholdern präsentiert
    - Sprint Retrospective: Zur Verbesserung der Qualität und Effektivität
- Sprints können nur abgebrochen, aber nicht verlängert werden
- Nicht fertig gestellte Items werden in den nächsten Sprint verschoben

Kanban

- Pull-System für konstante Arbeitsgeschwindigkeit
- Leichtgewichtiges System, um Widerstand bei der Einführung zu vermeiden
- Oftmals in Verbindung mit Scrum verwendet => Scrumban

Prinzipien

- Visualisierung des Workflows
- Limitierung des Work-in-Progress
- Verwaltung des Flows
- Regeln des Prozesses
- Gemeinsame Verbesserung

Kanban-Board (Bsp. VO6, S.19)

- Bietet Überblick über alle offenen Aufgaben und deren aktuellen Stage
- Stages definieren in Form von Spalten den Workflow
- Optionale Swimlanes entlang einer Zeile können zur Gruppierung von Aufgaben dienen
- Können physisch oder in Form einer Software verwendet werden

Fallstricke

- Auch agile Entwicklung ist keine Silver-Bullet
- Agile Entwicklung wird oft mit Scrum gleichgesetzt
- Dark Scrum: Missbrauch der Ideen von Scrum
    - Keine Änderung der Kultur, sondern nur Umbenennung, z.B. Projekt-Manager wird zu Scrum Master
    - Keine Selbstorganisation des Teams
    - Micromanagement, z.B. Daily wird als Status-Meetingmissbraucht
    - Zombie Scrum: Stures Befolgen der Regeln ohne Agilität zu leben

Agiles Testen

- Überbegriff für qualitätssichernde Maßnahmen in agilen Projekten
- Baut auf bestehendem Stand der Technik auf
- Folgt den Prinzipien der agilen Software-Entwicklung, u.a.
    - Fokus auf Zusammenarbeit und Kommunikation
    - Qualitätssicherung von Beginn des Projekts an
    - Iterative Entwicklung
- Innovation und Weiterentwicklung des Testens hauptsächlich im Bereich des agilen Testens

Die Rolle des agilen Testers

- Im ganzen Lebenszyklus involviert
- Bringt Wissen und Erfahrung aus dem Testen ins Projekt
- Sorgt für Einführung & Einhaltung der qualitätssichernden Maßnahmen im Projekt
    - Teststrategie
    - Auswahl und Unterstützung bei Verwendung der Testmethoden
    - Aufbereitung Testdaten
    - Erstellen von Reports und Berichten
- Identifiziert Risiken im Produkt
- Unterstützt Analysten bei der Definition von User Stories & Akzeptanztests
- Unterstützt Entwickler bei Entwicklung von automatisierten Tests

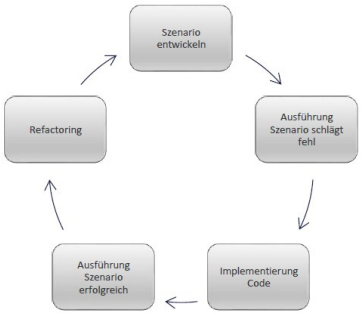
Whole-Team-Approach

- Selbstorganisiertes Team, das Fachbereich-Experten, Entwickler, Tester und weitere Spezialisten einbezieht → „Power of Three“
- Gesamtes Team ist für die benötigte Qualität verantwortlich
- Auch Entwickler benötigen Wissen über Testen

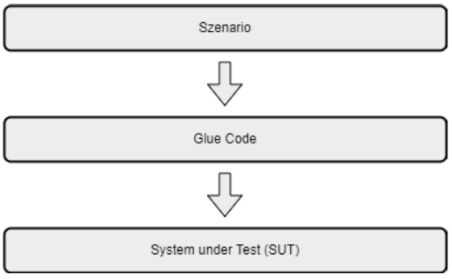
Agile Testmethoden

- Test-Driven Development (TDD) →(VO4)
    - Entwickler-fokussiert
- Behaviour-Driven Development (BDD) (Verhaltensgetriebene Entwicklung)
    - Kunden-fokussiert
- Acceptance Test-Driven Development (ATDD) (Abnahmetestgetriebene Entwicklung)
    - Kunden-fokussiert

Behaviour-Driven Development (BDD)



- Product Owner bzw. Analyst, Entwickler und Tester entwickeln gemeinsam Beispiele (= **Szenarien**) wie das System zu funktionieren hat
- Szenarien werden möglichst natürlichsprachlich formuliert. Werden auch als „Living Documentation“ bezeichnet
- Entwicklung einer gemeinsamen Sprache zwischen Beteiligten



- Szenario: Formulierung in eigener Sprache
- GlueCode: Übersetzung des Szenarios in Aufrufe an das zu testende System
- System under Test: System bzw. Funktionalität, die getestet werden soll

Cucumber (Framework) & Gherkin (Beschreibungssprache)(Bsp. VO6,S.32 bis 37)

Acceptance Test-Driven Development (ATDD)

- Von Product Owner bzw. Analyst, Entwickler und Tester werden zu den User Stories **Akzeptanztests** definiert
- Tests werden möglichst natürlichsprachlich formuliert, so dass sie auch von Nicht-Technikern geschrieben werden können
- Akzeptanztests können, aber müssen nicht automatisiert werden
- Ähnlich zu BDD, aber höherer Abstraktionslevel

Testautomatisierung

- Gemeint sind meist UI-Tests
- Auch oft unter dem Begriff e2e-Tests („end-to-end“)

Herausforderungen

- UI schwierig zu testen
- Hohe Volatilität von UI-Technologien
- Langsamere Ausführung im Vergleich zu anderen automatisierten Tests
- Leicht zu brechen, z.B. Umbenennung eines Buttons
- Cross-Browser-Kompatibilität

Testautomatisierung – Gestern & Heute

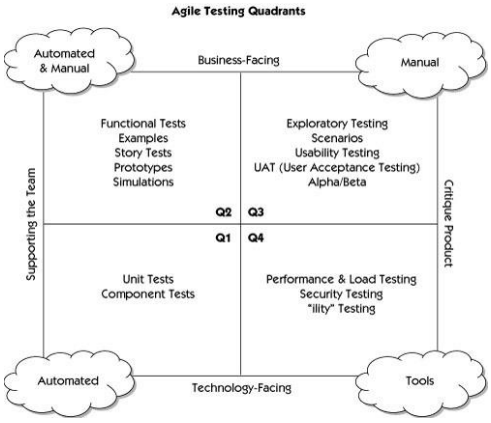
Gestern	Heute
- keine Unterstützung durch UI-Technologien - Tests eher unzuverlässig (False Positives) - Hauptsächlich kommerzielle Produkte	- Unterstützung durch UI-Technologien - False Positives weniger, aber noch möglich - Open Source-Frameworks (ggf. mit kommerziellem Support)

Testautomatisierung –Best Practices

- Erfordert Abstimmung bzw. Aufstellen von Regeln zwischen Entwicklern und Testern
- Testen des Positiv-Falls
- Für Setup Abkürzungen verwenden, um Laufzeit zu verringern (z.B. direkter Aufruf von REST-Services statt durch UI)
- Auf sauberen Code und gute Strukturierung achten
- Ergebnis von Record-and-Play-Funktionalität nicht 1:1, sondern nur als Ausgangsbasis für Implementierung verwenden

Agile Testquadranten nach Crispin & Gregory

- Einteilung der verschiedenen Test-Typen in unterschiedliche Quadranten
- Keine Priorisierung oder Reihung der einzelnen Quadranten => es werden alle Quadranten benötigt



Q2	Q3
Q1	Q4

Q1: Technische Tests, die das Team unterstützen

- Entwicklung von Unit-Tests z.B. mittels Test-Driven Development
- Regelmäßige Integration und Ausführen der Tests
- Zeigen die interne Qualität des Produkts
- Verwendete Werkzeuge
    - Versionskontrollsysteme (z.B. Git)
    - IDEs für Refactoring-Support
    - Build-Tools (z.B. Apache Maven,Gradle)
    - Continuous Integration Server (z.B. Jenkins, GitLabCI/CD, GitHub Actions)
    - Unit-Test-Frameworks (z.B. JUnit, TestNG)

Q2: Geschäftsorientierte Tests, die das Team unterstützen

- Automatisierte Akzeptanztests, die auch von Nicht-Entwicklern verstanden werden können
- Höheres Abstraktionslevel als Tests von Q1, jedoch Überschneidung zwischen Tests von Q1 und Q2 möglich
- Zeigen die externe Qualität des Produkts und festzustellen, ob man "done" ist
- Unterstützen gemeinsame Sprache zwischen Domäne und Entwicklung zu finden
- Werkzeuge:
    - manuell: Checklisten, Mind Maps, Tabellen, Mock-Ups, Diagramme
    - automatisiert: BDD-Frameworks, Werkzeuge zur Testautomatisierung

Q3: Geschäftsorientierte Tests, die das Produkt kritisieren

- Hauptsächlich manuelle Tests
- Test-Methoden:
    - Szenario-basierte Tests
    - Explorative Tests

    - Session-basierte Tests
    - Usability-Tests
- Testen von Schnittstellen (z.B. Web Services, RESTful-Services)

Q4: Technische Tests, die das Produkt kritisieren

- Testen der nicht-funktionalen Anforderungen
- Benötigen oftmals Unterstützung durch Spezialisten
- *ilities-Tests
    - Security
    - Maintainability
    - Reliability
    - Installability
- Performance-Tests
- Last-Tests
    - Unterschiedliche Arten mit unterschiedlichen Zielen: Load-, Stress-, Spike-, Capacity-Testing
    - Aufwändiges Test-Setup
    - Benötigen ähnliche Ausstattung wie Produktionssystem

Zusammenfassung

- Agiles Manifest bildet die Grundlage für agile Software-Entwicklung und agiles Testen
- Scrum & Kanban sind die vorherrschenden Vorgehensmodelle in der agilen Software-Entwicklung
- Tester sind Teil des Teams und unterstützen Projekt vom Beginn bis zum Ende
- Spezielle Testmethoden im agilen Bereich: TDD, BDD, ATDD
- UI-Tests werden besser unterstützt und haben an Bedeutung gewonnen
- Agile Testquadranten geben Hilfestellung welche Tests benötigt werden, um Qualität sicherstellen zu können