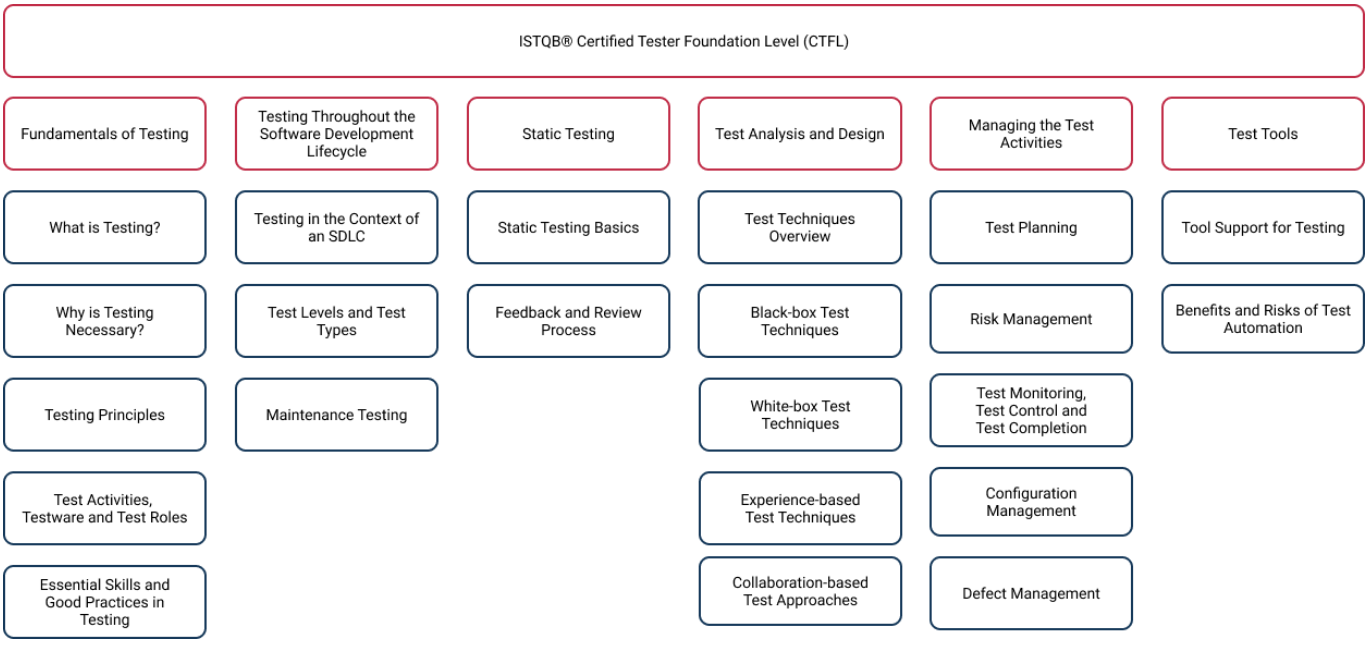
Allgemeines

- Aktueller Lehrplan ist Version 4.0.1. Dieser wurde im Jahr 2023 veröffentlicht.
- Ab Juni 2024 können Prüfungen nur noch nach dem neuen Lehrplan abgelegt werden.
- Bis einschließlich Mai 2024 sind noch Prüfungen mit Lehrplan 3.1 möglich.
- ca. 70% des Lehrplans sind zwischen v.3.1 und v4.0 identisch geblieben.
- Die Prüfung dauert 60 Minuten, wobei 40 Multiple-Choice-Fragen zu beantworten sind.
    - Um die Prüfung zu bestehen, müssen 65% der Gesamtpunktzahl erreicht werden, also 26 Punkte.
    - Für jede Frage gibt es genau einen Punkt.
    - Bei manchen Fragen braucht man deutlich mehr Zeit, um sie zu beantworten, als bei anderen. Trotzdem ist die Punktzahl stets 1.
- **Hinweis:** Wenn man die Prüfung in einer Sprache abhält, die nicht die eigene Muttersprache ist, kann man eine Verlängerung der Prüfungszeit um 25% beantragen. Das sind bei 60 Minuten Prüfungszeit 15 Minuten zusätzlich.

Lehrplan

- Der Lehrplan ist in 6 Kapitel unterteilt.
- Die Kapitel 4 und 5 sind am wichtigsten, da sich ca. 50% der Prüfungsfragen auf diese beiden Kapitel beziehen.



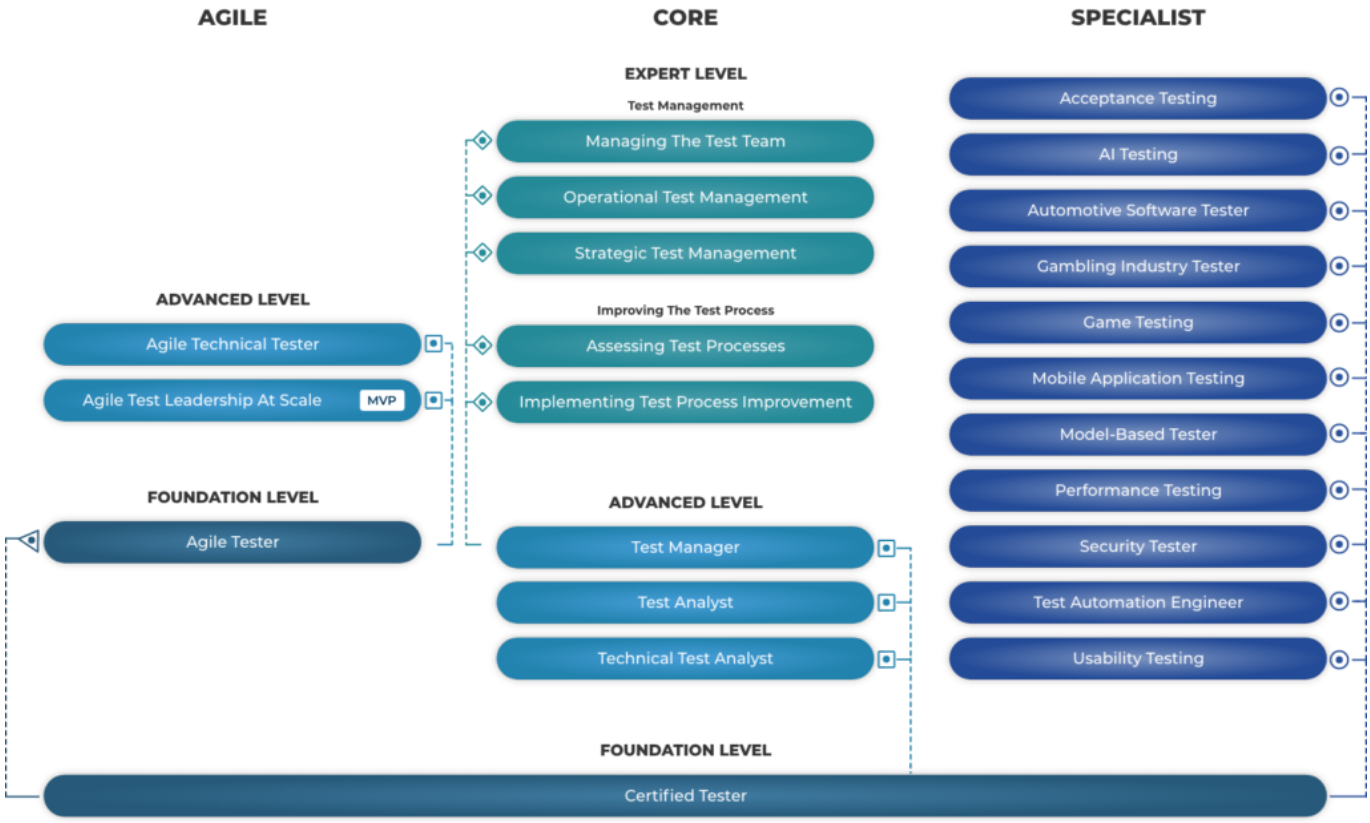
Weiterbildungsmöglichkeiten

Die Foundation Level Zertifizierung ist die Basis für alle weiteren Zertifizierung des ISTQB. Es gibt folgende Ausbildungsstufen:

- Foundation Level
- Advanced Level

- Expert Level

Zusätzlich sind Module für Spezialisierungen wie Game Testing und Usability Testing möglich. Außerdem sind auch im Bereich Agile Softwaretesting separate Zertifizierungen durchführbar.



Unterschied zwischen Efficiency and Effectiveness

- **Efficiency**: Ein gutes Verhältnis zwischen Nutzen eines Tests und die dafür benötigten Ressourcen wie Zeit, Personal, Budget etc.
- **Effectiveness**: Damit meint man die Wirksamkeit eines Tests. Ein Test ist wirksam, wenn er genau testet, was gewünscht ist und akkurate sowie vollständige Ergebnisse liefert.

Chapter 1

Was ist Testing?

- Eine Software zu testen bedeutet nicht nur, die Software auszuführen und auf Fehler zur prüfen, sondern auch die Dokumentation zu bewerten, den Quelltext zu lesen und nach logischen Fehlern zu suchen.
- Unterschied Verifizierung und Validierung:
    - Beim **Verifizieren** wird geprüft, ob die Software ihre Spezifikation bzw. die Anforderungsliste korrekt umsetzt.
    - Bei der **Validierung** wird geprüft, ob die Software das tut, was ihre Endanwender von ihr erwarten.
    - Mit anderen Worten: Wenn eine Software laut Spezifikation korrekt implementiert wurde (Verifikation), aber nicht das tut, was der Kunde sich gewünscht hat (Validierung), dann hat die

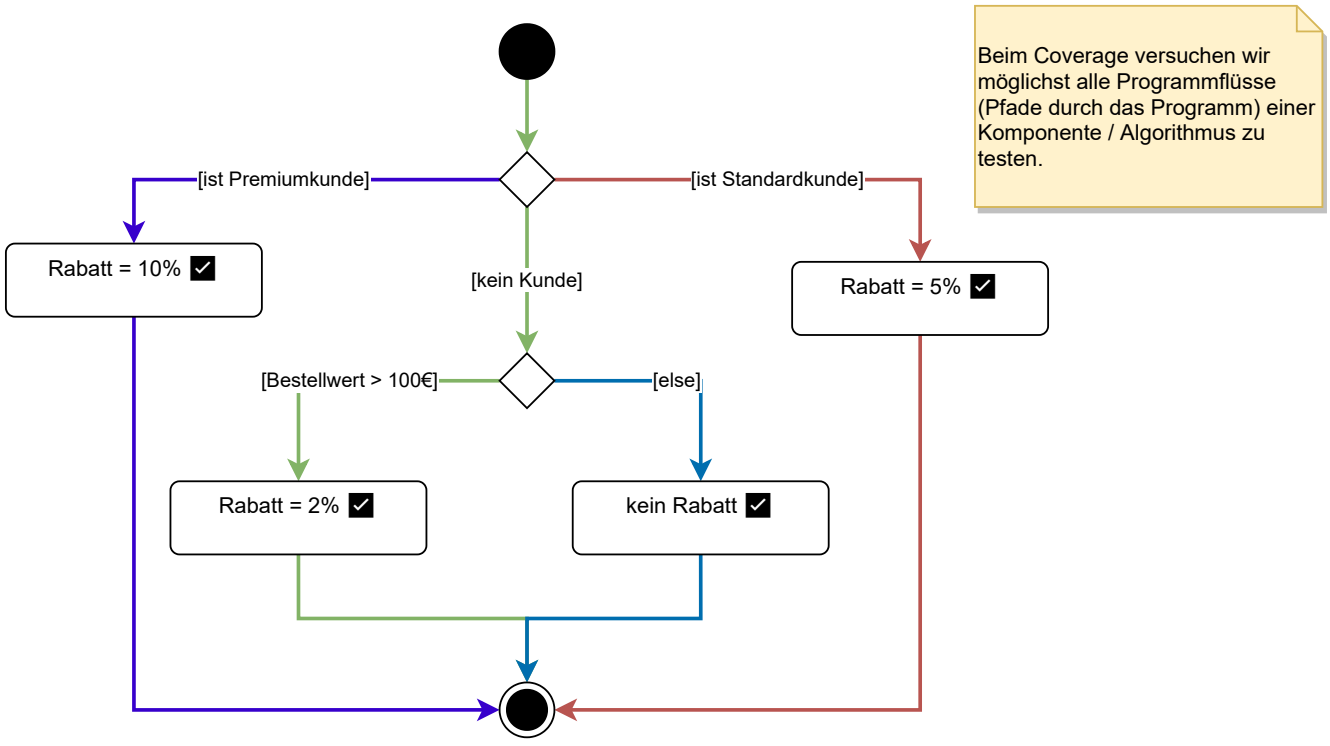
Software keinen wirklichen Nutzen und somit ihr Ziel verfehlt.

- **dynamisches Testen**: Die Software befindet sich in der Ausführung, während getestet wird.
- **statisches Testen**: Die Software wird ohne Ausführung getestet, z.B. durch gemeinsame Reviews des Source Codes. Es wird vor allem nach logischen Programmierfehlern im Quelltext gesucht. Beispiel: Im Quelltext wird eine Division ausgeführt, aber der Quelltext prüft nicht, ob der Teiler Null ist (Fault, Defect). Wenn nun versucht wird, durch Null zu teilen, stürzt das Programm ab (Failure/Ausfall/Fehlfunktion).

Ziele von Software-Testing

- Bewertung von Arbeitsergebnissen, wie Anforderungsdokumente, User Stories, Entwürfe (UI und Softwarearchitektur), Quelltext etc.
- Beispiel User Story: As a manager (Rolle), I want to be able to understand my colleagues progress (Was soll das System leisten?), so I can better report our sucess and failures (Intent, Begründung für Anforderung).
- Mängel (Defects) in der Software finden: z.B. logischer Programmierfehler im Quelltext (`ArrayIndexOutOfBounds`, `DivisionByZero`, Endlosschleifen etc.).
- Fehlverhalten (Failures) in der Software auslösen: z.B. Eingabe eines Wortes statt einer Zahl führt zum Absturz eines Programms (`Exception`) oder zu einem falschen Ergebnis (z.B. `NaN` in JavaScript).
- Möglichst alle Pfade durch das Programm testen (Test Coverage), sofern technisch durchführbar.

Beispiel für ein Test Coverage eines Algorithmus zur Berechnung von Kundenrabatten:



Unterschied zwischen Testing und Debugging

Ein **Debugger** ist ein Programm, mit dem der Softwareentwickler ein Programm schrittweise ausführen und analysieren kann (Variableninhalte anzeigen, Speicherverbrauch etc.). Ein Debugger wird dazu verwendet, logische Programmierfehler aufzuspüren, also Defects zu finden. Durch einen Debugger ist es i.d.R. erheblich

einfacher, die Ursache für ein Fehlverhalten eines Programms zu ermitteln. **Debugging wird nicht als Testing angesehen, sondern ist für die Ermittlung der Fehlerursachen erforderlich**.

Testing ist umfassender als Debugging. Beim Testen kann auch eine statische Analyse vorgenommen werden, also eine Analyse ohne Ausführung des Programms. Beim Debugging ist die Ausführung immer erforderlich. Debugging ist lediglich ein Teilaspekt des Testens und wird in den meisten Fällen nicht direkt vom Tester, sondern vom Entwickler durchgeführt.

Wenn der Entwickler den gefundenen Defect behoben hat ("Das müsste jetzt funktionieren!"), also den Quelltext entsprechend modifiziert hat, dann muss der Tester den Testfall erneut durchführen und prüfen, ob der Fehler tatsächlich behoben ist.

Was ist Regression Testing?

Wenn man Quelltext modifiziert bzw. erweitert, kann es passieren, dass man unbeabsichtigt neue Defects in das Programm einbaut. Das kann auch an Stellen im Source Code auftreten, an denen man gar nichts geändert hat, aber die aufgrund von Querabhängigkeiten von der Veränderung indirekt betroffen sind. Regression Testing prüft, ob nach einer Modifikation das Programm noch ordnungsgemäß funktioniert, also "nichts kaputt gemacht wurde". Eine **Regression** ist ein solcher "neu eingebauter" Defect.

Beispiel: Eine kleine Funktion `boolean isAdult(int age)` zur Bestimmung der Volljährigkeit wird derart modifiziert, dass erst ab einem Alter von 21 eine Volljährigkeit besteht. Da diese Funktion in vielen anderen Teilen bzw. Komponenten der Software verwendet wird, kann dies zu unerwarteten Fehlern bzw. fehlerhaften Ergebnissen im Programm führen. Ein Regression Testing prüft hier, ob die anderen "Teile" bzw. "Komponenten" noch ordnungsgemäß funktionieren, also keine Regression stattfindet.

Warum ist Testing notwendig?

Unterschied Quality Control und Testing

Qualitätskontrolle (Quality Control) schließt alle Maßnahmen und Tätigkeiten ein, mit denen **die Qualität eines Systems bzw. Produkts** geprüft und sichergestellt werden kann. **Testing hingegen ist nur *eine* Form der QA**: eine Tätigkeit, die während des Softwarelebenszyklus durchgeführt wird, um die Qualität der Software sicherzustellen. Neben dem Testing gibt es noch andere Formen von QC: Simulationen, Prototyping (eine unvollständige Software, die nur einen bestimmten Aspekt des Systems implementiert, z.B. UI Mockups), Eigenschaften prüfen an einem Modell (z.B. aus einem Material wie Plastik, Holz etc.), mathematische Korrektheitsbeweise etc.

Im Gegensatz zur Qualitätskontrolle (QC) **bezieht sich Quality Assurance (Qualitätssicherung) auf den gesamten *Entwicklungs- und Testprozess***. Beim QA geht es darum, den Entwicklungsprozess derart zu optimieren, dass unter dessen strikter Einhaltung qualitativ hochwertige (Software-) Produkte entstehen. Jeder, der am Softwareprojekt beteiligt ist, ist auch indirekt am QA beteiligt.

Testberichte dienen als Feedback für die Qualitätskontrolle sowie das QA. Beim QC werden Testberichte verwendet, um Defects zu beheben. Beim QA werden die Testergebnisse verwendet, um den Entwicklungs- und Testprozess zu optimieren bzw. anzupassen.

Errors, Defects, Failures, Root Cause

Wenn ein Nutzer eine Software verwendet und etwas "falsch macht", dann bezeichnet man das als **Error**. Beispiel: Der Nutzer soll in ein Eingabefeld eine Zahl eingeben, aber tippt versehentlich einen Text ein. Die Errors eines Nutzers können die **Defects** einer Software auslösen (triggern). Wenn ein solcher Defect ausgelöst wird, dann führt das in der Regel zu einem Fehlverhalten / Fehlfunktion der Software (**Failure**). Beispiel: Ein Programm fordert den Nutzer auf, eine Zahl einzugeben, der Nutzer gibt aber versehentlich einen Text ein. Da das Programm die Benutzereingaben nicht validiert (Defect), stürzt das Programm bei einer Berechnung ab (Failure).

Defects treten hauptsächlich in folgenden Dokumenten auf:

- Source Code
- Anforderungsdokumente (z.B. Pflichtenheft)
- Softwaredokumentation (z.B. Instruction Manual)
- Build-Dateien (Skripte, die die Software übersetzen, in ein Paket verpacken und ggf. ausführen)

Hinweis: Manche Defects treten gar nicht, sehr selten oder nur in ganz bestimmten Situationen auf. Je eher man einen Defect erkennt, umso besser. Software Testing sollte über alle Entwicklungsphasen hinweg durchgeführt werden, also beginnend ab der Analysephase.

Wenn das System einen Failure hat, kann das auch bedeuten, dass die Software etwas tut, was sie gar nicht machen soll. Beispiel: Eine fehlerhafte Produktionsmaschine baut nicht-funktionierende Platinen.

Manche Failures werden nicht durch den Nutzer ausgelöst, sondern durch Ereignisse, die in der IT Infrastruktur aufreten: z.B. der Ausfall eines Datenbankservers kann dazu führen, dass eine Software nicht ordnungsgemäß funktioniert, weil die Software für ihre Aufgabe die Daten aus der Datenbank benötigt. Auch physikalische Beeinträchtigungen wie kosmische Strahlung (z.B. im Weltall) oder elektromagnetische Felder (z.B. bei Funktürmen) können die Funktionsweise eines Hard- und Softwaresystems derart einschränken, dass das System seine Aufgabe nicht mehr erfüllen kann.

Die **Root Cause** (Hauptursache) ist die eigentliche Ursache für das Auftreten eines Fehlers. Manche Fehler sind lediglich "Symptome" der Root Cause. Solange man nicht die Root Cause findet und behebt, werden immer wieder Fehler auftreten. Beispiel: Eine Software muss umfangreiche Daten in Echtzeit in ein Datenbanksystem abspeichern. Es kommt hin und wieder vor, dass das Datenbanksystem überlastet ist und Anfragen nicht bearbeiten kann, wodurch Fehler entstehen. Eine kurzfristige Lösung wäre, wenn die Software die Datenbankanfragen künstlich verzögert, so dass keine Überlastung auftreten kann. Die eigentliche Hauptursache für das Problem ist jedoch die mangelnde Leistung des Datenbanksystems. Dieses müsste ausgetauscht oder repliziert werden, so dass Anfragen auf mehrere Server verteilt und schneller bearbeitet werden können.

Testing Principles

1. **Testing kann nur die Anwesenheit von Defects nachweisen**, nicht jedoch deren Abwesenheit bzw. die Fehlerfreiheit bzw. vollständige Korrektheit der Software! Testing reduziert lediglich die Wahrscheinlichkeit, dass Defects unentdeckt bleiben.
2. **Vollumfängliches Testen ist praktisch unmöglich**: Ein Programm mit allen möglichen Eingaben zu testen und alle Programmpfade zu durchlaufen, ist in der Praxis nicht möglich und auch nicht sinnvoll. Stattdessen sollte man versuchen seine Tests zu priorisieren und nach Risiko zu gewichten. Jene Komponenten, deren Fehlverhalten zu massiven Auswirkungen führen könnten, sollte man im Idealfall als erstes testen.

3. **Frühes Testen spart Zeit und Geld**. Beispiel: Du entwickelst eine Softwarebibliothek für deine Kollegen im Unternehmen. Deine Kollegen verwenden diese Bibliothek für ihre Programme. Falls deine Bibliothek gravierende Defects enthält, wird das natürlich auch Auswirkungen auf die Software deiner Kollegen haben. Hier wäre es von großem Vorteil, wenn du die Bibliothek vor der Weitergabe ausführlich testest und auf Korrektheit prüfst.
4. **Defects haben "Hot Spots"**: Es wurde beobachtet, dass sich die meisten Defects innerhalb einiger weniger Systemkomponenten konzentrieren (sogenannte Hot Spots). Diese Defects führen hauptsächlich zum Fehlverhalten der Software. Achtung: Je mehr Tests man durchführt, umso mehr Defects wird man sehr wahrscheinlich auch finden.
5. **Tests verlieren eventuell an Wirksamkeit**: Bestehende Tests sorgen in der Regel nicht dafür, neue Defects zu entdecken. Wenn man also eine Software erweitert, muss man zwangsläufig auch neue Tests entwickeln. Bestehende Tests können jedoch für das Regression Testing hilfreich sein.
6. **Testing ist abhängig vom Kontext**: Testverfahren bzw. Testmethoden unterscheiden sich je nach Softwareprojekt bzw. auch nach Umfang der Software und den zeitlichen, finanziellen und gesetzlichen Rahmenbedingungen. Beispiel: Eine Software mit grafischer Schnittstelle wird anders getestet, als eine Software, die lediglich kommandozeilenbasiert arbeitet. Eine Software, die im medizinischen Bereich eingesetzt wird und deren Fehlverhalten zum Tode eines Menschen führen kann, wird deutlich strikter geprüft, als ein PC-Spiel.
7. **Abwesenheit von Fehlern bedeutet nicht zwangsläufig, dass die Software korrekt bzw. zielgerichtet entwickelt wurde**: Nur weil die Software die Spezifikation vermeintlich fehlerfrei erfüllt, heißt das nicht, dass die Software auch dem Endkunden den gewünschten Nutzen bringt und ihn bei seiner alltäglichen Arbeit unterstützt. Neben der Verifikation ist auch immer eine Validierung notwendig.

Test Activities, Testware and Test Roles

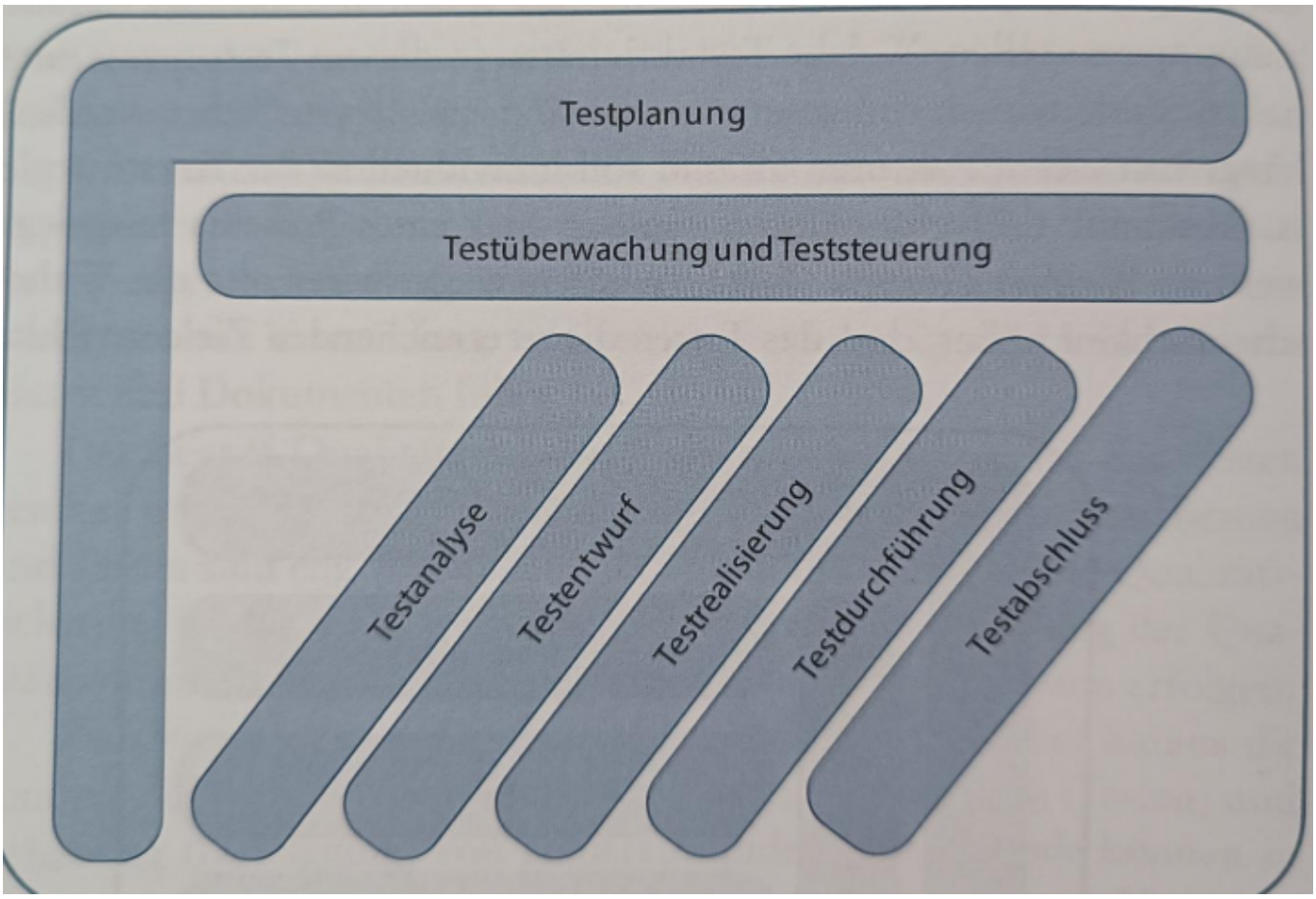
Eine **Test Basis** ist eine Menge von Dokumenten, die für die Erstellung von Testfällen und das Test Design verwendet wird. Ohne Test Basis kann keine Test Analysis durchgeführt werden. Beispiele für eine Test Basis sind: Anforderungsdokumente (Lastenheft, Pflichtenheft), User Manuals, Source Code, Use Cases (Anwendungsfall), User Stories, Technische und funktionale Spezifikation (eigentlich im Pflichtenheft enthalten).

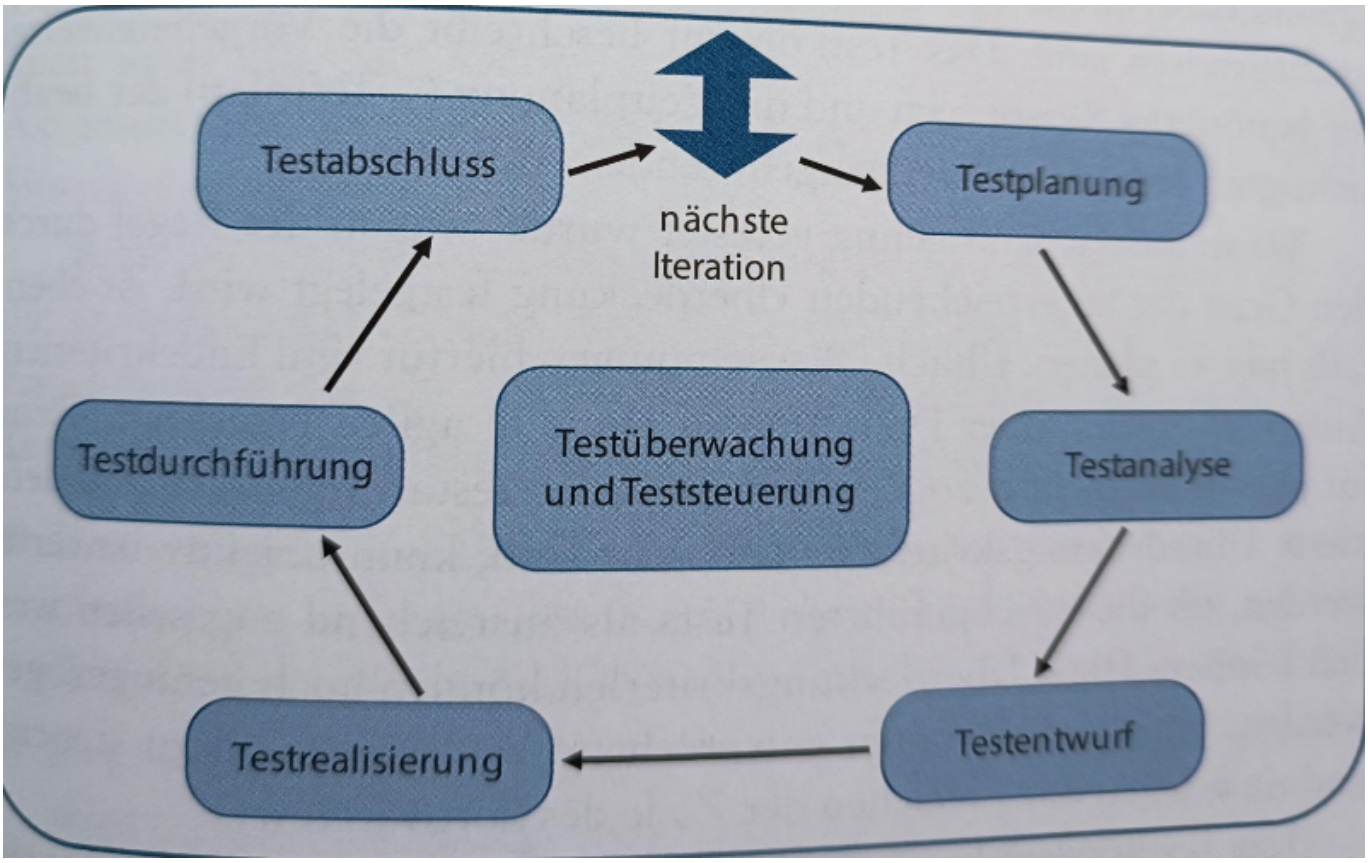
Testware ist eine Zusammensetzung aus den Worten Testing und Software. Darunter versteht man eine Menge von Dokumenten und Werkzeugen, die während des Testprozesses erstellt oder verwendet werden. Beispiel: Testprotokoll bzw. Testbericht, Test Runner zum automatischen Durchlauf von Tests, Tools für Dokumentation und Bug-Tracking.

Ein **Test Charter** ist ein Dokument, das festhält, welche Ziele mit einer Test Session verfolgt werden, was zu testen ist und wie beim Testen vorgegangen werden soll. Man kann es sich wie eine Anleitung zum Testen vorstellen.

Eine **Anomaly** ist eine Abweichung von einem erwarteten Testergebnis.

Ein **Testprozess** besteht aus diversen **Testaktivitäten**, die sich in folgende Kategorien einordnen lassen:





- Test Planning:
  - Testziele festlegen und Testverfahren auswählen
- Test Monitoring und Test Control:

    - **Sicherstellen**, dass der Testprozess voranschreitet und **die Testziele bzw. der Testplan erfüllt werden**.
    - Beim Test Controlling werden Maßnahmen festgelegt, die durchzuführen sind, wenn der Testplan vom Soll abweicht.
- Test Analysis:
    - Analysiere die Test Basis und identifiziere Aspekte, die sinnvoll getestet werden können.
    - Dazu gehören auch die Test-Rahmenbedingungen, die für einen Testfall gelten müssen (z.B. 1000 Tests pro Sekunde mit einer Response Time von 50ms).
    - Priorisieren von Testfällen nach Risiko.
    - **Was soll getestet werden** und wie genau oder wie umfangreich ist zu testen?
- Test Design (Testentwurf):
    - Konkrete Ausarbeitung von Testfällen und Testware.
    - Dazu gehören auch die Festlegung von Eingaben für den Testfall, die Wertebereiche für Testeingaben etc.
    - Festlegung der Test Environment (z.B. Testdatenbank, Datenbank-Mocks, Dateien / Verzeichnisse etc.)
    - Festlegung der benötigten Test Tools.
    - **Wie soll getestet werden?**
- Test Implementation:
    - Die konkrete Umsetzung eines Tests.
    - Hier wird Testware (Testberichte, Testanforderungen) erzeugt bzw. bezogen, d.h. Testdaten generiert oder aus einer Datenbank geholt.
    - Hier werden automatisierte Tests (z.B. Unit Tests) programmiert und ausgeführt.
    - Testfälle werden kategorisiert zu Test Suites, sofern sinnvoll.
    - Angemessene Testumgebung erstellen (z.B. einen Test Server aufsetzen)
    - Testskripte erstellen.
    - Reihenfolge festlegen, in der die Tests auszuführen sind.
    - **Hier wird alles vorbereitet, um die Tests ausführen zu können.**
- Test Execution:
    - Die Tests in der vorher festgelegten Reihenfolge durchlaufen.
    - Tests können händisch oder automatisch durchgeführt werden.
    - Erwartete und tatsächlich erhaltene Ergebnisse vergleichen und aufzeichnen.
    - Ergründen, weshalb ein Testergebnis nicht dem erwarteten Ergebnis entspricht.
- Test Completion:
    - Hier werden Aktivitäten ausgeführt, die i.d.R. erst am Ende einer Entwicklungsphase bzw. Iteration sinnvoll sind.
    - Testdokumente werden archiviert bzw. weitergereicht an andere Teams.
    - Der gesamte Testprozess wird kritisch bewertet und ggf. optimiert.
    - Die Test Environment wird zurückgesetzt.
    - Ein Abschlussbericht wird erstellt und an die Stakeholder übergeben.

Vorlage für einen **Test-Plan**:

```
<Dokumenttitel>
Author: <Name>
```

```
1. Introduction
  - Test Objectives
2. Testing Resources
  - Personal, Testrollen, Testumgebunge
3. Scope of Testing
  - Test Objects (was soll getestet werden?)
  - Was soll _nicht_ getestet werden
4. Testing approaches
  - Welches Testverfahren ist anzuwenden?
  - Welche Testtechniken sind anzuwenden?
  - Welche Arten von Tests sind durchzuführen?
    - Performance
    - Functional
    - Usability
    - Security
    - Portability
5. Test Schedule
  - Wann soll was und womit getestet werden?
  - Was sind die Deadlines für die Testphasen?
  - Wann ist eine Testiteration abgeschlossen?
6. Risks & Issues
  - Welche Risiken oder Probleme könnten in den Testphasen auftreten?
  - Wie soll mit diesen Problemen umgegangen werden? (Risikostragie)
  - Notfallplan
```

Was beeinflusst das Software Testing?

- Stakeholder
- Zeit, Budget, Personal
- Zur Verfügung stehende Testing Tools
- Branche (z.B. im medizinischen Bereich ist intensiver zu testen als im Gaming Bereich)
- der verwendete Softwareentwicklungsprozess (Wasserfall vs. Scrum)
- Das Entwicklerteam und deren Einstellung zum Thema Testing
- Die Vorgaben der Geschäftsführung bzw. des Unternehmens
- Technische Faktoren (Softwareart: IoT Device ist schwerer zu updaten als ein PC Game)

EXKURS: Automatisierte Tests mit dem JUnit Testing Framework entwickeln

Schaue dir hierzu die Programme im Verzeichnis `programs/UnitTestExamples` an. Wir verwenden `JUnit` in der Version `5`.

Um den Source Code testbar zu machen, verwenden wir *Java Interfaces*. Durch den Einsatz eines Interfaces können wir Softwarekomponenten mit wenig Aufwand austauschen. In unseren Beispielprogrammen tauschen wir die Original-Datenquelle durch eine Fake-Datenquelle aus. Das beschleunigt den Test und sorgt für gleiche Ausgangssbedingungen beim Testen.

Testware

Eine **Test Procedure** ist eine Sequenz von Test Cases, die in einer bestimmten Ausführreihengolge angeordnet sind. Zur Test Procedure gehören Vor- und Nachbedingungen für die einzelnen Test Cases, sowie eine

Initialisierung der Testumgebung und ein "Tear-Down" der Testumgebung. Analogie: Denke an eine JUnit Test Class mit `BeforeAll`, `AfterAll`, `BeforeEach` und `AfterEach` sowie `Test`.

Eine **Test Suite** ist eine Menge von Test Procedures oder Test Scripts, die als eine Einheit auszuführen ist.

Während des gesamten Testprozesses werden Dokumente (Testware) erzeugt:

- In der **Test Planning Phase**:
  - Test Plan
  - zeitliche Planung der Tests (Test Schedule),
  - Risk Register (Kategorisierung nach Risiko),
  - Vorbedingungen für Tests
  - Definition, wann Test abgeschlossen ist (wann wurde ausreichend getestet?)
- In der **Test Monitoring / Test Control Phase**:
  - Test Progress Reports
  - Maßnahmen, die getroffen wurden, um entdeckte Probleme zu beheben (Control Directives)
  - Einschätzung der Risiken (Das Risiko kann sich aufgrund geänderter Rahmenbedingungen wie Personal etc. ändern)
- In der **Test Analysis Phase**:
  - Testfälle (Test Cases) erstellen
  - Test Charters (Anleitung zum Testen, Vorgabe)
  - Coverage Items (z.B. Methoden und Klassen, die getestet werden sollen oder eine Webseite einer Web Application)
  - Testdaten (z.B. eine CSV Datei, Datenbank, eine API zum Generieren von Fake Data)
  - Vorgaben für die Test Environment (z.B. Vorhandensein bestimmter Test Server, Anlegen von Dateien und Verzeichnisstrukturen etc.)
- In der **Test Implementation Phase**:
  - Test Procedures (z.B. Implementierung einer JUnit Test Class)
  - Testskripte (z.B. ein Skript, das Benutzereingaben simuliert (e.g. AutoHotKey))
  - Test Data (z.B. in Form einer @MethodSource in JUnit)
  - Aufbau der Testumgebung bzw. der Testinfrastruktur (z.B. Client Rechner, Server Rechner, und entsprechende Software, sowie Netzwerkverbindung)
  - Entwicklung von Stubs, Mocks, Simulatoren etc. Ziel: Ersetzung von anderen Softwarekomponenten / Hardwarekomponenten durch spezielle Testimplementierungen (z.B. Interface `IDataSource` mit Fake Implementierung `FakeDataSource`).
    - macht Test schneller
    - macht Test reproduzierbar (im Idealfall herrscht immer dieselbe Ausgangssituation für einen Testfall)
    - macht Test überhaupt erst durchführbar (z.B. weil es keinen echten Test Server bzw. keine echte Test Datenbank gibt).
- In der **Test Execution Phase**:
  - Testberichte, Logs.
  - Gefundene Defects in Form von *Bug Reports* melden.
- In der **Test Completion Phase**:
  - Erstellen eines Test Completion Report (Abschlussbericht).
  - Erstellung von Change Requests (Modifikationen bzw. Erweiternugen der Software)

        - ein Change Request wird in Form von Product Backlog Items hinterlegt (z.B. bei Softwareentwicklungsprozess SCRUM)
    - Aus dem Review des Testprozesses:
        - was kann das nächste Mal besser gemacht werden?
        - wie lässt sich der Testprozess optimieren?

Beispiel für ein **Product Backlog**:

User story	Story point(s)	Priority
As a user, I am able to search for documents so I can find them more easily	2	1
As a site visitor, I can compare different types of accounts to see which account type suites me best	1	2
As a user, I can submit questions through the website so I know how to better use the product	1	3
As a site visitor, I am shown what I can do in the product so I know whether or not this product will fill my needs	2	4
As a user, I want to be able to retrieve documents that were deleted so I can reclaim documents that were deleted on accident	3	5
As a site visitor and user, I can sign up for newsletters to remain up to date on the product	2	6
As a user, I am notified when a new feature is released so I know what is possible	1	7
As a user, I can change my user name if desired	3	8
As an admin, I need the ability to update which team a user belongs to so I can make sure all teams are up to date	3	9
As a user, I can enable spell check so I can be confident my final document has no spelling errors	4	10

Beispiel für einen **Bug Report**:

```
Programmversion: 1.0.3
Betriebssystem: Windows 11 24H1 Professional

Führe folgende Schritte aus, um Fehler zu reproduzieren:
- Öffne Dialogfenster.
- Gib in das Eingabefeld "Alter" die Zeichenkette "xx" ein.
- Klicke auf den Button "Volljährigkeit prüfen".

Erwartetes Ergebnis (Expected Behaviour):
- Programm fordert Nutzer auf, seine Eingabe zu korrigieren.
```

```
Tatsächliches Ergebnis (Actual Behaviour):
- Programm stürzt mit einer `NumberFormatException` ab.
```

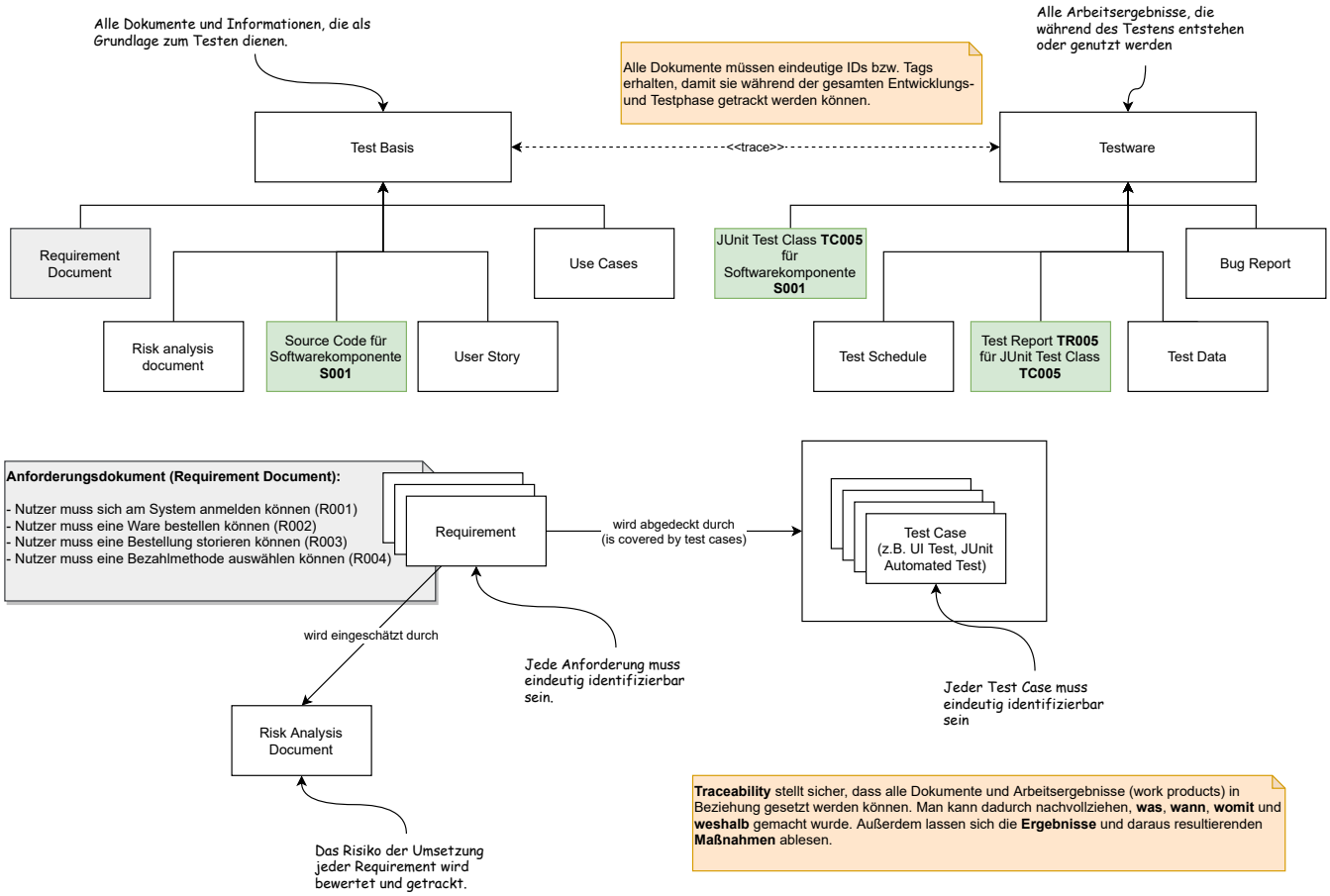
Traceability zwischen Test Basis und Testware

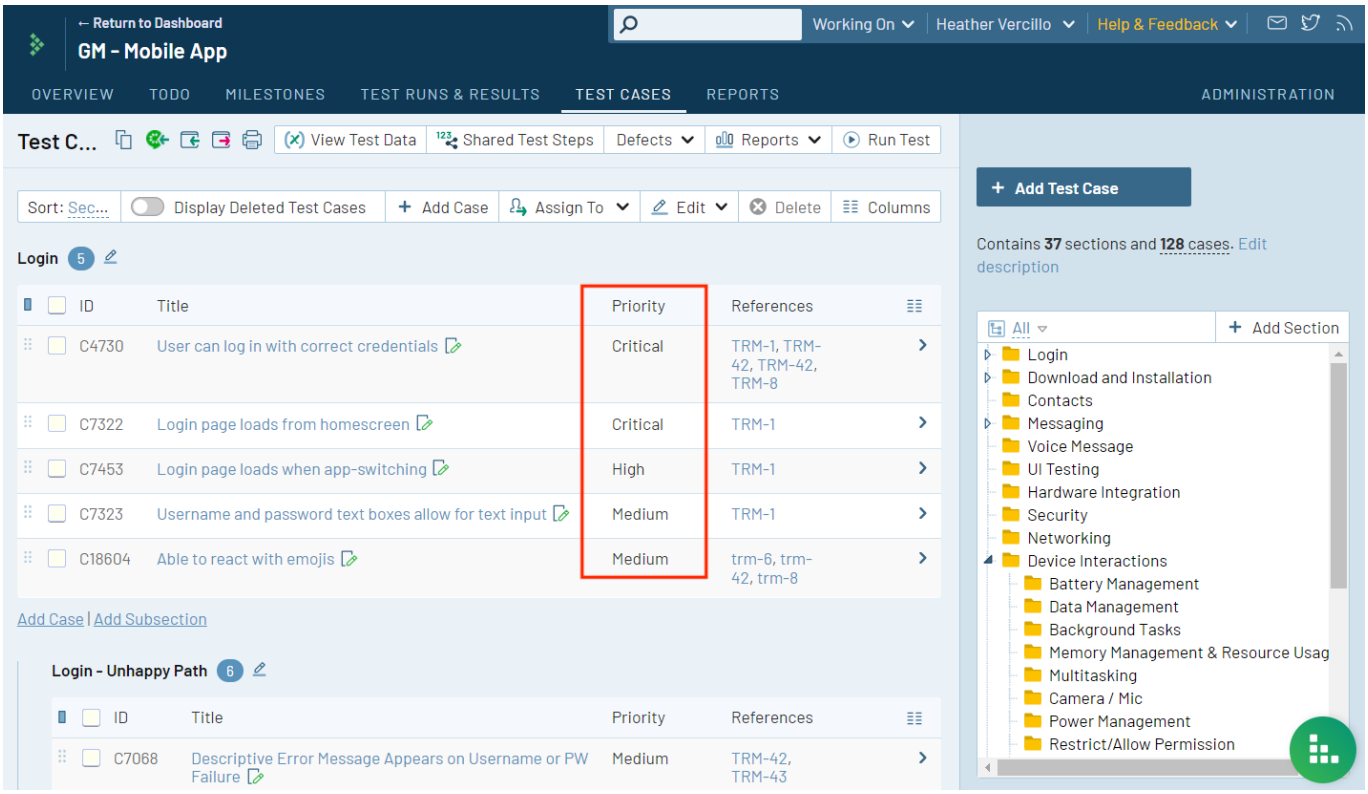
Mit **Traceability** stellen wir logische Verbindungen zwischen den Testobjekten und den Arbeitserzeugnissen des Testprozesses her. Beispiel: Eine Anforderung (Requirement) im Anforderungsdokument wird mit einer Reihe von Testfällen (Test Cases) verknüpft. Die Ergebnisse (Test Reports, Test Logs, Bug Reports) der durchgeführten Testfälle werden wiederum mit den Test Cases verlinkt.

Durch die Traceability können wir jederzeit nachvollziehen, inwieweit eine Anforderung durch Tests abgedeckt wird bzw. wie gut diese Anforderung bisher umgesetzt wurde.

Tracebility hat u.a. folgende Vorteile:

- Nachvollziehbarkeit und Verständlichkeit des Testprozesses
- Fördert die Produktqualität und das Erreichen der Geschäftsziele
- Vereinfacht das Testen, wodurch Zeit eingespart und Kosten reduziert werden.
- Hilft beim Nachweis von Konformität (z.B. mit Tests lässt sich nachweisen, dass Datenschutzrichtlinien beachtet wurden)





Rollen, die beim Testen eingenommen werden

Im Wesentlichen werden hier zwei Rollen unterschieden: Test Manager und Tester.

Der **Test Manager** kümmert sich um die Planung, das Monitoring und das Controlling, während der **Tester** sich mit der Testanalyse, dem Testentwurf und der Testimplementierung beschäftigt. Außerdem führt der Tester die Tests durch und erstellt die Testberichte bzw. Testlogs.

In großen Projekten, bei denen viele Teams miteinander arbeiten, gibt es meist eine Person, die die Rolle des Test Managers übernimmt. Das Entwicklerteam kümmert sich um die Testimplementierung und ein Tester führt die Tests durch.

In kleineren Firmen werden die Aufgaben des Test Managers und des Testers im Entwicklerteam aufgeteilt.

Achtung: Es ist eher ungünstig, wenn Entwickler und Tester ein und diesselbe Person sind. Der Entwickler kann zwar seine Unit Tests schreiben und ausführen, aber das gesamte Programm - inklusive UI - sollte durch eine andere Person getestet werden.

Essential Skills & Good Practices in Testing

- Grundlagen des Testings beherrschen, d.h. Testverfahren und Testmethoden kennen. (z.B. ISTQB Foundation Level 😉)
- Gute Kommunikation, d.h. Kritik konstruktiv äußern und allen Beteiligten aktiv zuhören.
- Sorgfalt, analytisches Denken, Genauigkeit, Neugier
- technisches Wissen für den Umgang mit Software, Hardware und den entsprechenden Test Tools
- Domain Knowledge, d.h. umfangreiches Wissen über die Branche bzw. das Projektumfeld
    - Beispiel 1: Wenn eine Software zu testen ist, die einen PKW Motor steuert, dann muss ich als Tester ein gutes Verständnis von Motoren und PKWs haben.

    - Beispiel 2: Wenn ich eine Software testen soll, die Lohnbescheide erstellt, dann muss ich mich gut in Lohn- und Steuerberechnungen auskennen.

Whole-Team Approach

Ein **Acceptance Test** prüft, ob das bisher entwickelte System den gewünschten Sollzustand erreicht hat. Beispiel bei SCRUM: Man legt für einen 4 Wochen Zyklus eine Menge von zu implementierenden Features fest. Am Ende des Zyklus wird geprüft, ob diese Features vollständig und korrekt umgesetzt wurden.

Ein Acceptance Test kann auch bedeuten, dass der Kunde die ausgelieferte Software selbst testet und dem Entwicklerteam mitteilt, ob die Software seine Erwartungen erfüllt hat.

Beim **Whole-Team** Ansatz "ziehen alle Teammitglieder an einem Strang", d.h. alle sind bemüht, die Qualität der Software aufrecht zu erhalten. Alle Team Member streben nach dem selben Ziel. Man unterstützt sich gegenseitig und fördert die persönlichen Stärken.

Der Whole-Team Ansatz funktioniert am besten, wenn alle Teammitglieder in einem Raum arbeiten, sich also räumlich nah sind. Dadurch kann besser kommuniziert und miteinander gearbeitet werden.

Die Tester stehen im ständigen Kontakt mit dem Entwicklerteam, dem Kunden sowie dem Projektmanagement-Team.

Independence of Testing

Wenn Tester und Entwickler diesselbe Person sind oder sich beide Personen sehr nahe stehen, kann sich das negativ auf die Effektivität des Testens auswirken. Beispiel: Ein Entwickler weiß sehr genau, wie sein Programm funktioniert. Er weiß, welche Eingaben das Programm erwartet und mit welchen Ausgaben zu rechnen ist. Eine nicht direkt involvierte Person hat einen ganz anderen Blick auf die Software und findet vielleicht Fehler, die dem Entwickler niemals aufgefallen wären. Fazit: **Eine gewisse Unabhängigkeit (Independence) zwischen Entwickler und Tester ist also empfehlenswert bzw. notwendig.**

Beispiele für Independence Levels:

- Tester ist Entwickler: no independence
- Tester ist Entwickler des eigenen Teams: low independence
- Tester arbeitet außerhalb des Entwicklerteams, aber innerhalb des Unternehmens: medium independence
- Tester arbeitet außerhalb des Unternehmens: very high independence

Man unterscheidet folgende Arten von Tests:

- **Unit Test**: Testet eine Methode, eine Klasse, ein Modul oder eine ganze Komponente
- **Integration Test**: Testet, ob mehrere Units bzw. Komponenten, die miteinander verknüpft werden, korrekt funktionieren. (z.B. Funktioniert meine Softwarekomponente mit der Datenbankkomponente?)
- **System Test**: Testet das gesamte System als eine Einheit (Beispiel: ein System bestehend aus Frontend, Backend, Authentication Service, Ressource Server, Database Server etc.)
- **System Integration Test**: Testen, ob das entwickelte System mit Fremdsystemen im Unternehmen reibungslos funktioniert. Beispiel: Bediensoftware für eine Produktionsmaschine, muss nicht nur mit der Maschinensteuerung, sondern auch mit der Software für die Produktionssteuerung reibungslos

funktionieren. Beispiel 2: Mein Onlineshop muss mit dem Finanzdienstleistersystem PayPal funktionieren.

Auf jedem Independence Level sollte getestet werden:

- Entwickler führt Unit Tests und Integration Tests durch.
- Test-Team führt System Test und System Integration Test durch.
- Auftraggeber bzw. Vertreter des Auftraggebers führt Acceptance Testing durch.

Terminologie

- **Coverage**: Wie umfangreich (Angabe in %) wurde ein Testobjekt getestet? Beispiel: Wie viele Methoden und Code Zeilen einer Class wurden mit Unit Tests geprüft?
- **Debugging**: Mit einem Debugger durchläfut man den Programmcode Anweisung für Anweisung und spürt logische Fehler bzw. Root Causes auf.
- **Defect**: Ein Mangel bzw. ein Bug in der Software, der ausgelöst werden kann, aber nicht muss. Beispiel: Programm validiert keine Eingaben.
- **Error**: Eine fehlerhafte Bedienung durch den User. Beispiel: User soll Zahl eintippen, aber gibt stattdessen ein Wort ein.
- **Failure**: Das von der Software gezeigte Fehlverhalten. Beispiel: Software zeigt Exception an, stürzt ab oder berechnet offensichtlich falsche Ergebnisse.
- **Quality**: Wie gut oder schlecht erfüllt die Software die Erwartungen bzw. die Vorgaben des Kunden? Kriterien für Softwarequalität sind u.a:
    - Korrektheit (korrekte Ergebnisse berechnen)
    - Verlässlichkeit (wie gut funktioniert Software in anderen Umgebungen?)
    - Robustheit (Software darf bei Nutzerfehlern nicht abstürzen)
    - Benutzerfreundlichkeit (übersichtlich, anpassbar, barrierefrei, zielgerichtet)
    - intuitiv zu benutzen bzw. verständlich (einfach zu bedienen, erklärt sich von selbst)
    - wartbar, d.h. einfach zu verändern bzw. zu erweitern (bezieht sich auf den Source Code)
    - testbar (sowohl auf Source Code Ebene als auch auf Benutzerbene)
    - portierbar (Software kann auf unterschiedlichen Betriebssystemen und Hardwarearchitekturen ausgeführt werden)
    - effizient (gutes Verhältnis von Nutzen und benötigter Ressourcen wie Speicher und CPU)
    - ausfallsicher bzw. verfügbar (zu wie viel Prozent ist die Software pro Jahr nutzbar?)
    - dokumentiert (wie gut ist Source Code und Software dokumentiert?)
- **Quality Assurance**: Sämtliche Maßnahmen, die der Sicherung der Qualität dienen. Beispiel: Testen, Testberichte auswerten und Bug Fixes anfordern, Personal an Sorgfaltspflicht erinnern, Personal schulen, Testprozess optimieren, Softwareentwicklungsprozess optimieren etc.
- **Root Cause**: Die Wurzel des Problems, bzw. die Hauptursache für ein Fehlverhalten des Systems. Beispiel: Nutzer kann keine Dateien speichern. Hauptursache ist eine zu kleine Festplatte, aber Administrator versucht immer nur temporäre Dateien zu löschen, wodurch das Problem nur kurzfristig gelöst wird, aber dann doch immer wiederkehrt.
- **Test analysis**: Eine Phase innerhalb eines Testprozesses. Hier werden die Testbedingungen auf Grundlage der Test Basis ermittelt.
- **Test Basis**: Sämtliche Informationen und Dokumente, die verwendet werden, um daraus Testfälle und Testbedingungen abzuleiten. Beispiel: Anforderungsdokument (Lastenheft, Pflichtenheft),

Benutzerhandbuch, Use Cases (diese wurden in der Softwareanalyse ermittelt), User Stories, sonstige Spezifikationen

- **Test Case**: Ein Testfall. Hier wird aktiv getestet. Beispiel: Eine JUnit Test Method wird ausgeführt. Das erwartete Ergebnis wird mit dem tatsächlichen Ergebnis verglichen. Vorbedingungen (z.B. eine bestimmte Datei muss existieren oder eine Verbindung zu einem Server muss bestehen oder ein Rechner muss online sein) werden geprüft und Nachbedingungen (z.B. Datei wurde angelegt, Verbindung wurde getrennt, App installiert) sichergestellt.
- **Test Completion**: Eine Phase des Testprozesses. Hier wird ein Testabschlussbericht erstellt und an die Stakeholder übermittelt. Der Testprozess wird kritisch betrachtet und optimiert. Testware wird für spätere Verwendung gekennzeichnet und aufbewahrt. Testumgebung wird in Ursprungszustand zurückversetzt.
- **Test Condition**: Etwas, das man an einem System bzw. einer Software testen kann bzw. soll. Test Conditions können sehr allgemein aber auch sehr detailliert formuliert werden.
    - Beispiel für Testbedingungen: Funktion, Methode, Klasse, Feature, Einschränkungen
    - Konkrete Beispiele: "Teste Bezahlfunktion", "Teste Bezahlfunktion via PayPal", "Wenn Nutzername und Passwort eingegeben wurden, darf fortgefahren werden"
- **Test Control**: Kontrollieren, ob das Testen planmäßig voranschreitet. Bei Abweichungen müssen Gegenmaßnahmen getroffen werden. Beispiel: Wenn Defects festgestellt wurden, müssen diese zeitnah behoben werden, da ansonsten darauf aufbauende Tests nicht durchgeführt werden können.
- **Test Data**: Sämtliche Daten, die für die Ausführung von Test Cases benötigt werden. Beispiel: Zufallszahlen für die Berechnung einer Minimumsfunktion.
- **Test Design**: Eine Phase des Testprozesses. Hier werden konkrete Test Cases entworfen. Grundlage dafür bilden die Testbedingungen (Test Conditions). Beispiel: Aus der Testbedingung "Premiumkunden erhalten Rabatt von 5%" werden die konkreten Testfälle "Ist Premiumkunde und Bestellwert ist 300 Euro, dann 5% Rabatt" sowie "Ist kein Premiumkunde und Bestellwert ist 100 Euro, dann kein Rabatt" abgeleitet.
- **Test Execution**: Eine Phase des Testprozesses. Hier werden die im Design und in der Implementierung umgesetzten Test Cases ausgeführt und Test Logs bzw. Test Reports erstellt.
- **Test Implementation**: Eine Phase des Testprozesses. Hier werden die im Entwurf (Design) erstellten Test Cases programmiert und die Testumgebung (Test Environment) für die Testausführung vorbereitet. Beispiel: Ein programmierter Unit Test mit JUnit. Einrichten eines Datenbankservers mit Testdaten.
- **Test Monitoring**: Eine Phase des Testprozesses. Wird in Verbindung mit Test Control durchgeführt. Hier werden die Testergebnisse überwacht und Gegenmaßnahmen getroffen, sofern vom Testplan abgewichen wird.
- **Test Object**: Der Testgegenstand, also das "Ding", das getestet werden soll. Beispiel: eine konkrete Softwarekomponente, eine grafische Schnittstelle, ein Handbuch, ein Quelltext etc.
- **Test Objective**: Die Begründung bzw. die Ziele für das Testing.
    - Beispiel: Ein Test ist durchzuführen, um zu prüfen, ob die Software sich bei Fehlereingaben robust verhält. (Robustheit, Fehlertoleranz)
    - Allgemeine Ziele: Funktionalität sicherstellen, Performanz prüfen, Datensicherheit prüfen, Usability prüfen, Barrierefreiheit sicherstellen
- **Test Planning**: Eine Phase im Testprozess. Hier werden die Testziele, die Testobjekte, die Teststrategie, Techniken etc. festgelegt, sowie ein Zeitplan für das Testen erzeugt.
- **Test Procedure**: Eine Menge von auszuführenden Test Suites und Test Cases in einer vorgegebenen Reihenfolge. Hier wird auch definiert, ob und wie die Testumgebung zu initialisieren und am Ende wieder zurückzusetzen ist.

- **Test Result**: Das Ergebnis eines Tests und ggf. die daraus resultierenden Maßnhamen / Konsequenzen.
- **Testing**: Prüfen, ob ein System bzw. eine Komponente die Anforderungen erfüllt.
- **Testware**: Sämtliche Dokumente bzw. Artefakte, die während des Testprozesses erstellt werden oder zum Einsatz kommen. Beispiel: implementierte Test Cases, Test Reports, Coverage Data, Test Runner, Testing Framework, Bug Report, abgeänderter Testplan
- **Verification**: Ein Vorgang bei dem geprüft wird, ob das System die Anforderungsspezifikation erfüllt. Wurde die Software laut Plan korrekt und vollständig implementiert?
- **Validation**: Ein Vorgang bei dem geprüft wird, ob das System aus Sicht des Kunden die Erwartungen erfüllt und einen Nutzen hat. Wurde die richtige Software entwickelt?

Chapter 2: Testing in the Context of a Software Development Lifecycle

Ein **Vorgehensmodell** ist ein Modell, das beschreibt, wie man vorgehen sollte, um eine Lösung für ein Problem zu erarbeiten. Die durchzuführenden Aktivitäten sind meist sehr allgemein (abstrakt) beschrieben und müssen noch verfeinert bzw. angepasst werden. Beispiele:

- das sequenzielle Vorgehensmodell **Wasserfallmodell**
- das risikogetriebene und iterative Vorgehensmodell **Spiralmodell**
- das inkrementelle Vorgehensmodell **Unified Process**.

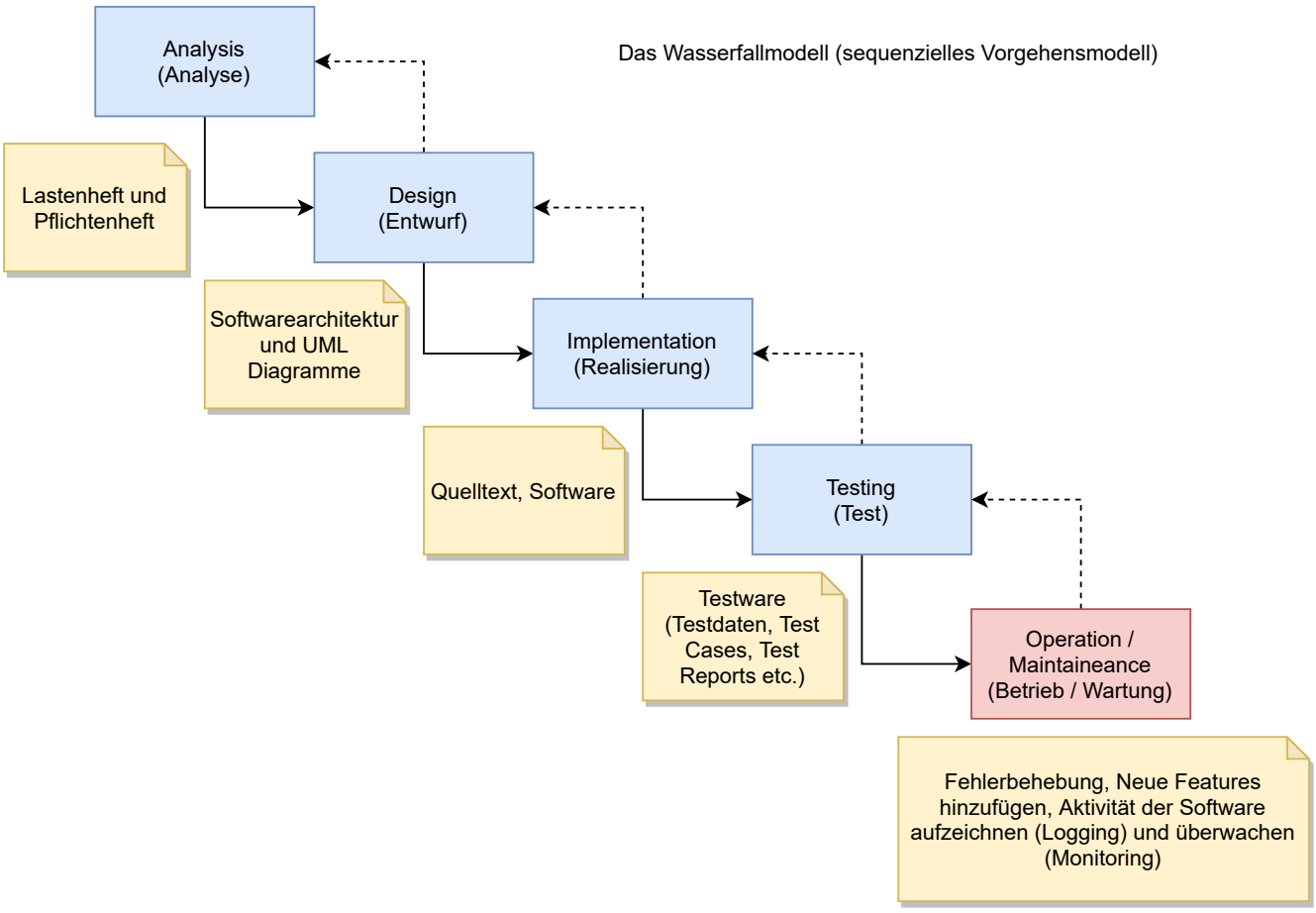
Ein **Inkrement** ist eine neue Softwareversion, die im Vergleich zur Vorgängerversion neue Features bereitstellt.

Ein **Softwareentwicklungsprozess** (**Software Development Process** oder auch **Software Development Lifecycle**) ist im Vergleich zu einem Vorgehensmodell wesentlich konkreter und detaillierter. Hier werden die Aktivitäten meist sehr genau beschrieben und die zeitliche Reihenfolge exakt festgelegt und die zu erstellenden Artefakte (Dokumente, Software etc.) genau vorgegeben. Beispiele: Scrum, Rational Unified Process.

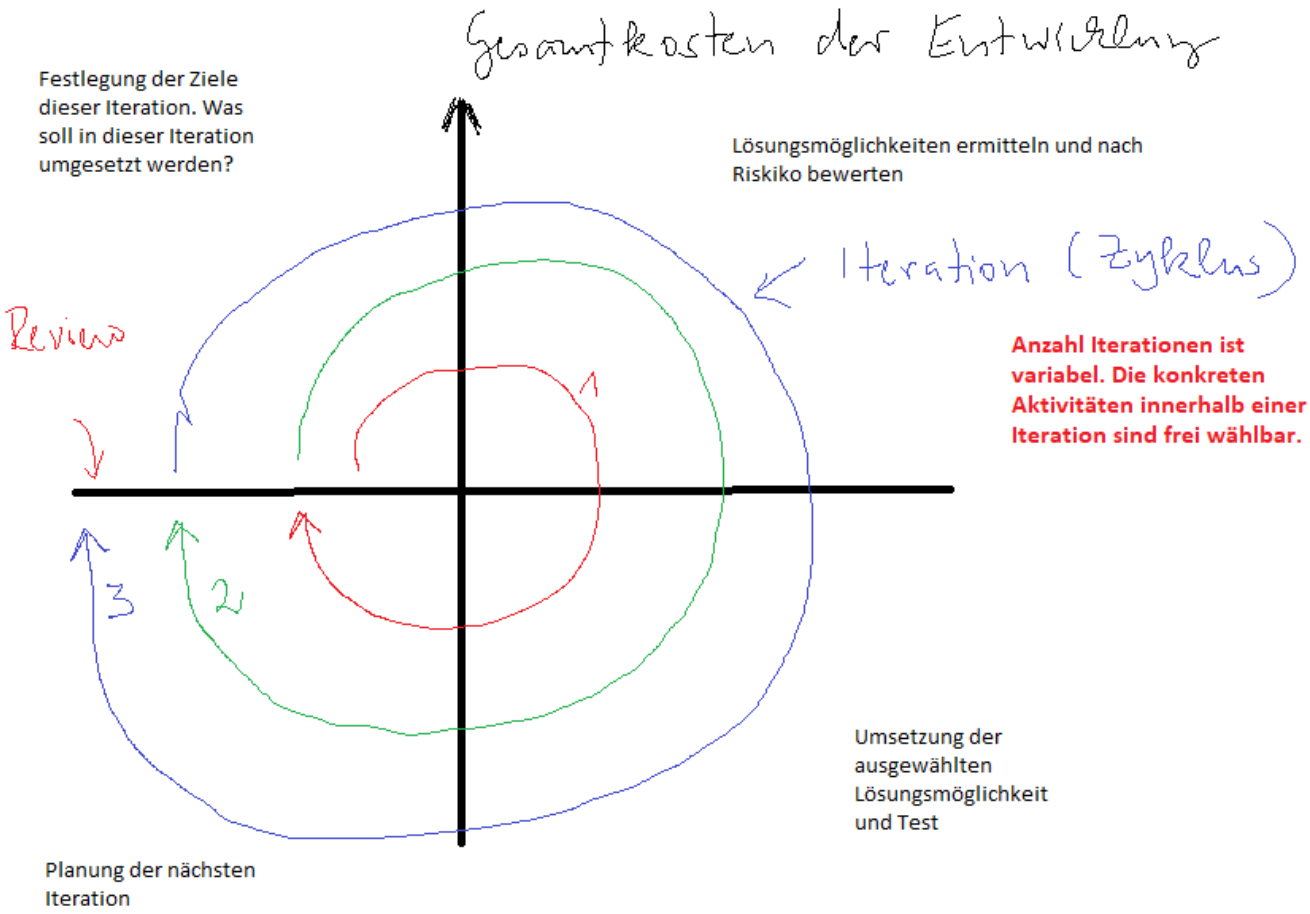
Neben den Softwareentwicklungsprozessen gibt es noch sogenannte Softwareentwicklungsansätze bzw. Methodiken (**Development Methods**, **Development Practices**). Beispiele:

- Test-Driven-Development (erst Unit Tests schreiben, dann zu testenden Code schreiben)
- eXtreme Programming (iterativ entwickeln, Pair Programming)
- Domain Driven Design (erst ein Domain Model programmieren und sich danach um Datenhaltung, grafische Schnittstelle und andere Services kümmern)

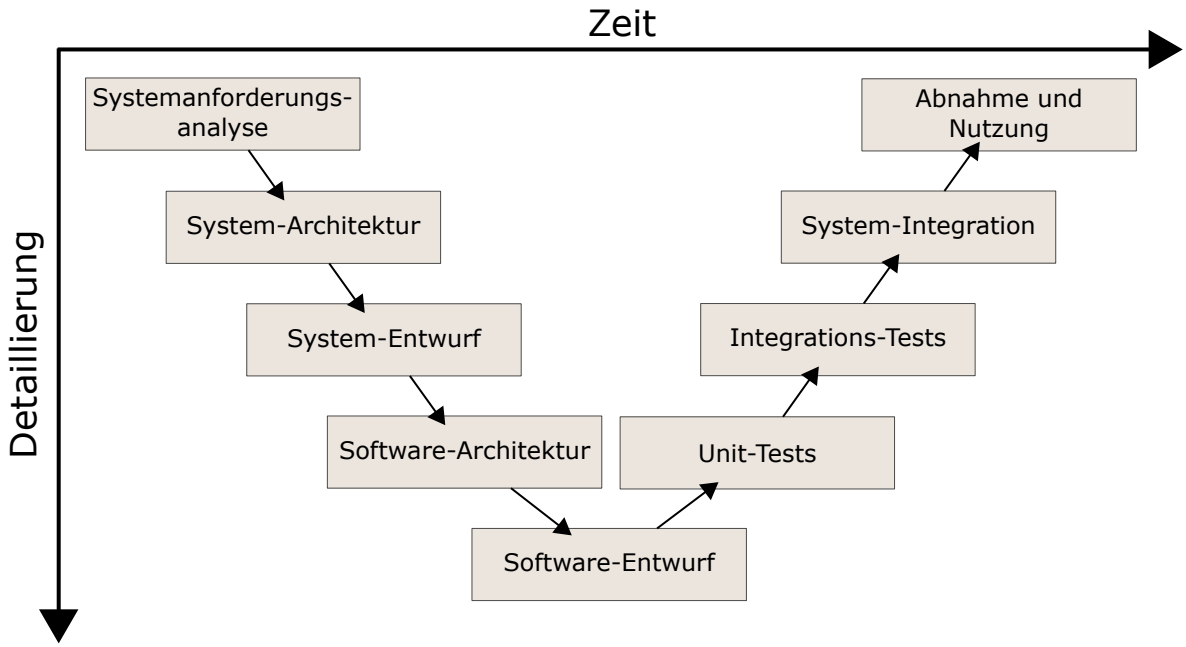
Das Wasserfallmodell:



Das Spiralmodell:



Das V-Modell:



Impact of the Software Development Lifecycle (SDLC) on Testing

Der verwendete SDLC legt Umfang und Zeitpunkt der Testaktivitäten fest. Er bestimmt außerdem, wie detailliert bzw. umfangreich getestet werden muss. Der SDLC bestimmt auch indirekt, wie intensiv

automatisiert getestet werden muss und wie stark ein Tester involviert ist.

Anmerkungen:

- bei Wasserfallmodell (sequenziell): Test Execution findet erst nach Softwareimplementierung statt, d.h. dynamische Tests sind erst sehr spät durchführbar. Testdokumentation steht im Vordergrund. Tester ist bei Anforderungsanalyse beteiligt.
- bei Spiralmodell (iterativ, inkrementell): Vollständig funktionierendes und getestetes Programm wird an Kunden nach jeder Iteration ausgeliefert. D.h. es müssen alle Testarten durchlaufen werden, also inklusive System Test und Acceptance Test.
- bei agilen Vorgehensmodellen: Im Fokus steht hier eine funktionierende Software, die möglichst oft und kontinuierlich an den Kunden ausgeliefert wird. Dadurch ist Dokumentation weniger relevant und automatisierte Tests umso wichtiger. Beispiel: Ein Commit in GitHub löst Buildprozess, Testing und automatisches Deployment der Software aus (**Continuous Deployment**). Testanalyse und Testentwurf geraten in den Hintergrund. Tests basieren auf Erfahrungswerten. Unit Tests, Integration Tests und System Tests finden vordergründig statt.

Good Testing Practices

- Für jede Entwicklungsaktivivät gibt es eine entsprechende Testaktivität (z.B. wird das im V-Modell so gemacht)
- Für jeden Test-Level gibt es spezifische Testziele. Ziel ist es, nach Möglichkeit alles zu testen, aber ohne Redundanz.
- Zu Beginn jeder Entwicklungsphase und für jeden Test-Level findet sofort Testanalyse und Testdesign statt. Prinzip: Early Testing.
- Tester sind am Review von Arbeitsergebnissen zu beteiligen, sobald diese vorliegen.

Testing as a driver for Software Development

Entwicklungsmethoden wie **TDD, ATDD und BDD** verfolgen dasselbe Ziel: Der Fokus liegt ganz klar auf dem Testen und die Tests treiben die eigentliche Entwicklung der Software voran. Andere Entwicklungsmethoden vernachlässigen Tests oder führen sie erst viel später durch, nachdem ein Großteil der Software bereits entwickelt wurde.

TDD, ATDD und BDD haben gemeinsam, dass die Tests **vor** der Implementierung der Software festgelegt und programmiert werden. Beispiel:

- bei TDD werden erst die Unit Tests geschrieben und erst danach der zu testende Code geschrieben. Im Gegensatz zu BDD formulieren hier die Entwickler die Tests.
- bei BDD werden die Tests in Form von "Given, When, Then" Sätzen formuliert, die dann in Unit Tests konvertiert werden. Die Tests werden im Gegensatz zu TDD von den Nutzern bzw. den Testern formuliert.
- bei ATDD leitet man die Tests aus der **Acceptance Criteria** ab. Diese werden von Nutzern und Business Representives formuliert. Die Ableitung geschieht während der Analyse und Entwurfsphase im SDLC.