T 01 Modul 450: Testkonzept

In den folgenden Kapiteln werden die grundlegenden Inhalte eines Testkonzepts für Softwareprojekte beschrieben. Die Inhalte konzentrieren sich auf die Rolle des Testens im Softwareentwicklungsprozess, die verschiedenen Teststrategien, Testmethoden und die dafür notwendigen Rahmenbedingungen. Es soll eine Idee vermittelt werden, um alle relevanten Aspekte des Testens abzudecken, von der Fehlervermeidung bis hin zur Fehlerklassifizierung und den benötigten Testumgebungen.

Inhaltsverzeichnis

1. Gründe zum Testen

Softwaretests sind ein essentieller Bestandteil jeder Softwareentwicklung. Sie dienen dazu, sicherzustellen, dass die entwickelte Software korrekt funktioniert, den Anforderungen entspricht und robust gegenüber potenziellen Fehlern ist. Fehler, die in der Produktion auftreten, können gravierende Folgen haben:

- **Hohe Kosten**
  Fehler, die erst in der Produktionsumgebung entdeckt werden, verursachen hohe Behebungsaufwände. Dies liegt nicht nur an den technischen Herausforderungen, sondern auch an den organisatorischen Hürden, wenn Änderungen an bereits ausgelieferten Softwareversionen vorgenommen werden müssen.
- **Rufschädigung**
  Softwarefehler können die Kundenzufriedenheit stark beeinträchtigen und das Vertrauen in die Softwarelösung mindern.
- **Komplexität der Behebung**
  Fehler in der Produktion sind oft schwieriger zu identifizieren und zu beheben, da die Fehlerursachen tief in der Architektur oder der Interaktion mit anderen Systemen verwurzelt sein können.

Durch systematisches Testen verhindern wir also, dass kritische Fehler den Entwicklungsprozess unbemerkt durchlaufen und in die Produktivumgebung gelangen.

2. Arten des Testens

Softwaretests sind nicht nur Aufgabe einer spezifischen Person oder eines Teams. Jeder Entwickler ist verantwortlich dafür, seinen Code auf korrekte Funktionalität zu prüfen. Dies geschieht idealerweise durch verschiedene Testmethoden:

- **Unit-Tests**
  Kleine, isolierte Tests, die einzelne Module oder Funktionen auf ihre korrekte Funktion überprüfen.
- **Integrationstests**
  Überprüfen das Zusammenspiel verschiedener Module oder Systeme.
- **Systemtests**
  Stellen sicher, dass das gesamte System wie geplant funktioniert.
- **Akzeptanztests**
  Beziehen den Kunden ein, um sicherzustellen, dass die gelieferten Lösungen den definierten Anforderungen entsprechen.

Die Frage ist also nicht nur, ob, sondern auch wie wir testen.

3. Aussagen von Tests

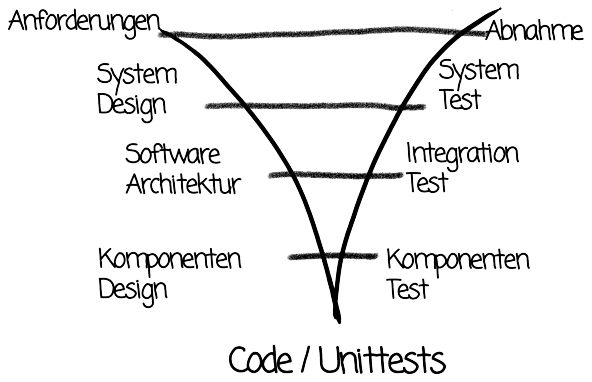
Softwaretests dienen verschiedenen Zwecken:

- **Fehlererkennung**
  Der Hauptgrund für Tests ist es, Fehler zu erkennen, bevor sie in die Produktion gelangen.
- **Qualitätssicherung**
  Durch Tests wird die Qualität der Software gesichert und nachweisbar gemacht.
- **Sicherstellung der Anforderungen**
  Tests helfen zu überprüfen, ob die entwickelten Features und Funktionalitäten mit den ursprünglichen Anforderungen übereinstimmen.
- **Wartbarkeit und Stabilität**
  Gut durchgeführte Tests erleichtern die Wartung der Software und stellen sicher, dass zukünftige Änderungen keine unerwünschten Nebenwirkungen haben.

4. Das V-Modell – Ein bewährtes Modell im Softwaretest

Das V-Modell ist ein traditionelles Testmodell, das die Testaktivitäten in Beziehung zu den Entwicklungsphasen setzt. Es zeigt, dass Testphasen parallel zu den Entwicklungsphasen ablaufen und eng miteinander verknüpft sind. Der zentrale Aspekt des V-Modells ist die Verbindung zwischen der Testhöhe und der Anforderungshöhe. Je höher die Tests im Modell angesiedelt sind, desto umfassender und abstrakter sind die Anforderungen.

Die Phasen des V-Modells:

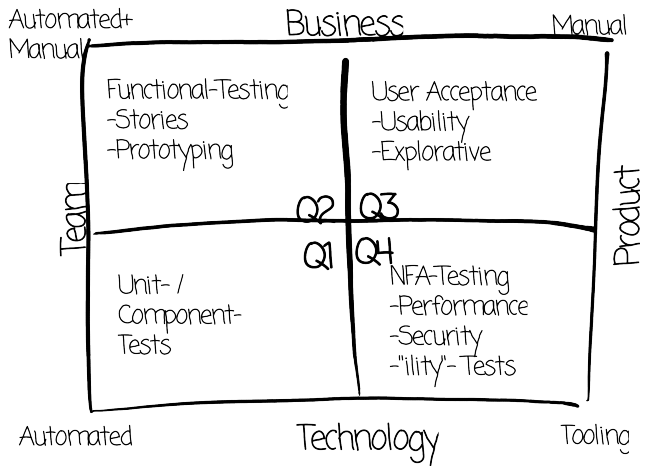


1. **Unit-Tests**
   Überprüfen die kleinsten Einheiten des Codes, wie einzelne Funktionen oder Methoden.
2. **Komponententests**
   Validieren das Zusammenspiel von Modulen oder Bibliotheken.
3. **Integrationstests**
   Prüfen das Zusammenspiel mehrerer Komponenten.
4. **Systemtests**
   Überprüfen das gesamte System, indem alle Teile in einer realistischen Umgebung getestet werden.
5. **Abnahmetests**
   Der Kunde validiert das Endprodukt und stellt sicher, dass alle Anforderungen erfüllt wurden.

Das V-Modell zeigt auch auf, dass der Aufwand zur Fehlerbehebung exponentiell ansteigt, je später der Fehler im Entwicklungsprozess entdeckt wird. Daher ist es entscheidend, Tests frühzeitig und regelmässig durchzuführen.

5. Quadrantenmodell – Testarten strukturiert darstellen

Das Quadrantenmodell im Software-Testing ist eine Methode, um verschiedene Testarten strukturiert darzustellen und zu kategorisieren. Es wurde von Lisa Crispin und Janet Gregory im Rahmen der agilen Testmethoden entwickelt und bietet einen Überblick über die unterschiedlichen Testaktivitäten und -ziele in vier Quadranten. Es hilft Teams, ihre Teststrategie umfassend zu planen und sicherzustellen, dass sowohl technische als auch geschäftliche Anforderungen berücksichtigt werden.



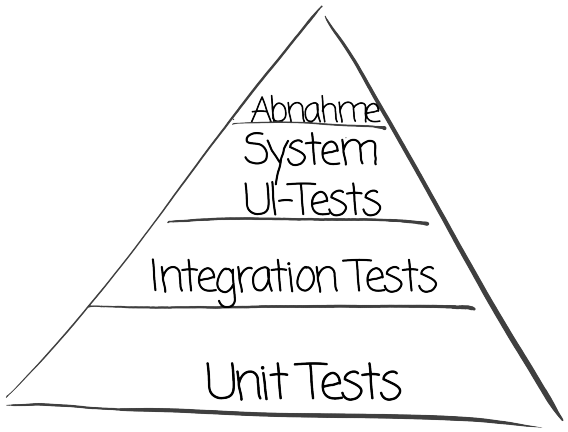
Das Quadrantenmodell nach Crispin und Gregory unterteilt Tests in vier Quadranten.

1. **Technikgetriebene Tests**
   Konzentrieren sich auf die technische Qualität der Software und werden früh im Entwicklungsprozess durchgeführt.
   (z. B. Unit-Tests, API-Tests)
2. **Geschäftsgetriebene Tests**
   Stellen sicher, dass die Software die Anforderungen des Kunden erfüllt.
   (z. B. Funktionstests, Akzeptanztests)
3. **Explorative Tests**
   Tests, die ohne festgelegte Testfälle durchgeführt werden, um die Grenzen des Systems auszuloten.
4. **Nicht-funktionale Tests**
   Überprüfen Aspekte wie Leistung, Sicherheit und Skalierbarkeit.

Das Quadrantenmodell hilft uns, die Testarten und ihre jeweiligen Ziele klar zu strukturieren und zu planen. Dabei soll es uns helfen, den Fokus zu setzen. Dies bedeutet nicht, dass die anderen Quadranten ausser acht gelassen werden sollen.

6. Die Testpyramide – Masse mit Klasse

Die Testpyramide ist eine bewährte Struktur, um die verschiedenen Testebenen zu organisieren. An der Basis der Pyramide stehen Unit-Tests, die zahlreich und automatisiert sind. Je weiter man die Pyramide hinaufgeht, desto weniger Tests werden durchgeführt, da die höheren Tests (z. B. Systemtests) aufwendig und ressourcenintensiv sind.



Die Testpyramide von Mike Cohn.

1. **Unit-Tests**
   Unit-Tests überprüfen einzelne Code-Einheiten, wie Methoden oder Funktionen, um sicherzustellen, dass sie isoliert korrekt arbeiten.
2. **Integrationstests**
   Integrationstests prüfen das Zusammenspiel mehrerer Komponenten oder Module, um sicherzustellen, dass diese korrekt interagieren.
3. **Systemtests**
   Systemtests überprüfen das Gesamtsystem als Ganzes, inklusive aller Komponenten, um sicherzustellen, dass es als Einheit den funktionalen Anforderungen entspricht.
4. **Abnahmetests**
   Abnahmetests sind die letzte Teststufe und werden durchgeführt, um sicherzustellen, dass das entwickelte System den Anforderungen der Benutzer oder des Kunden entspricht. Sie dienen der formalen Abnahme des Systems durch den Kunden.

Eine gut ausbalancierte Testpyramide sorgt dafür, dass Fehler frühzeitig entdeckt werden und die Tests insgesamt effizient und ressourcenschonend bleiben.

7. Testumgebung und -infrastruktur

Eine stabile und konsistente Testumgebung ist entscheidend für verlässliche Testergebnisse zum einen. Zum anderen ist die Kenntnis über die Umgebung wichtig um reproduzierbare Ergebnisse zu erzielen. Folgende Aspekte spielen dabei eine Rolle:

- **Betriebssysteme**
  Welche Betriebssysteme und Versionen werden unterstützt? Tests sollten unter verschiedenen Systemen durchgeführt werden (z. B. Linux, Windows, Mac).
- **Server, Datenbanken, etc**
  Welche Server und Datenbanken werden genutzt? Welcher Versionen und konkrete Typen werden oder wurden verwendet. Es ist wichtig, die Performance und Skalierbarkeit unter möglichst realistischen Bedingungen zu testen. Und Aussagen machen zu können, mit welchen Komponenten die Applikation funktionsfähig ist.
- **Hardware**
  Welche Hardware wird genutzt? Die Systemarchitektur (32/64-Bit, ARM vs. x86) sowie die Leistungsmerkmale wie CPU und RAM können die Testergebnisse beeinflussen.
- **Netzwerk**
  Die Netzwerktopologie (lokal vs. remote) und die Anbindung (LAN, WAN) haben ebenfalls Einfluss auf das Testverhalten. Es ist ebenfalls wichtig die Netzwerkumgebung zu kennen um allfällige Störer zu kennen.

8. Testdaten

Testdaten spielen eine zentrale Rolle im Testprozess. Sie können aus bestehenden Datensätzen (z. B. Abzug aus Produktionsdaten) oder eigens für die Tests generiert werden. Wichtige Aspekte sind dabei:

- **Reproduzierbarkeit**
  Testdaten müssen konsistent und reproduzierbar sein, um Vergleichbarkeit zwischen den Testläufen zu gewährleisten.
- **Datenschutz**
  Insbesondere bei der Nutzung von echten Produktionsdaten sind datenschutzrechtliche Vorgaben zu beachten.

9. Fehlerklassifizierung und Dringlichkeit

Nicht jeder Fehler hat die gleiche Bedeutung. Die Fehlerklassifizierung hilft, die Schwere eines Fehlers und seine Dringlichkeit zu bewerten. Dabei werden Fehler nach ihrer Ursache (versehentlich vs. mutwillig), der Schwere (fehlerfrei bis kritisch) und ihrer Auswirkungen (Feststellbarkeit, Auftretenshäufigkeit, Kundenreaktion) kategorisiert.

9.1. Fehlerklassifizierung

Die Fehlerklassifizierung bezieht sich darauf, wie ein Fehler im System eingeordnet wird, um seine Natur und den betroffenen Bereich zu verstehen. Die Klassifizierung hilft, den Fehler einzugrenzen und die Verantwortung für die Behebung zuzuweisen. Die wichtigsten Dimensionen der Fehlerklassifizierung sind:

9.1.1. Schweregrad

Der Schweregrad beschreibt, wie schwerwiegend die Auswirkungen des Fehlers auf das System sind. Die Schweregradstufen variieren in der Praxis, aber sie lassen sich üblicherweise in folgende Kategorien einteilen:

- **Kritisch**
  Das System oder eine wesentliche Funktion ist nicht nutzbar. In diesem Fall ist eine sofortige Aufmerksamkeit und eine schnelle Behebung notwendig, da der Fehler die Nutzung der Software unmöglich macht.
- **Hoch**
  Wichtige Funktionen sind beeinträchtigt, aber das System bleibt grundsätzlich funktionsfähig. Dieser Fall führt zu einer Priorisierten Behebung, da dies den Geschäftsbetrieb erheblich beeinträchtigen kann.
- **Mittel**
  Der Fehler verursacht Unannehmlichkeiten oder kleinere Probleme, die das System beeinträchtigen, aber der Kern der Anwendung funktioniert weiter. In diesem Fall wird der Fehler behoben, aber mit geringerer Priorität.
- **Niedrig**
  Der Fehler hat minimale Auswirkungen auf das System und verursacht wenig bis gar keine Störungen. Die Behebung des Fehler findet bei Gelegenheit statt, oft werden diese gesammelt und in Wartungstelease eingearbeitet.

9.1.2. Art des Fehlers

Fehler können auch nach ihrer Art klassifiziert werden, um zu verstehen, welches Problem genau vorliegt.

- **Funktionale Fehler**
  Betrifft die Hauptfunktionen der Anwendung
- **Usability-Fehler**
  Beeinträchtigt die Benutzerfreundlichkeit

- **Sicherheitsfehler**
  Betrifft die Sicherheitsmechanismen des Systems
- **Performance-Fehler**
  Betrifft die Geschwindigkeit oder Effizienz des Systems

9.1.3. Ursache des Fehlers

Es ist auch wichtig die Ursache für einen Fehler zu kennen. Nur so kann verhindert werden, dass dieser wieder eintritt. Dabei gibt es folgende Hauptkategorien.

- **Anforderungsfehler**
  Der Fehler ist entstanden, weil schon in den Anforderungen ein Fehler vorhanden war. Ebenso kann es sein, dass beim interpretieren der Anforderung ein Fehler passiert ist.
- **Böswilliger Fehler**
  Der Fehler wurde absichtlicht verursacht. Die Ursachen hierfür können extrem unterschiedlich sein. (Bereicherung, Schädigung, …)

9.1.4. Lokalisierung des Fehlers

Fehler können auch nach dem betroffenen Modul oder der Komponente oder wie schwierig es ist diesen Fehler zu Finden klassifiziert werden.

- **Frontend-Fehler**
  Betrifft die Benutzeroberfläche (z. B. ein Layout-Fehler in der Web-App).
- **Backend-Fehler**
  Betrifft serverseitige Logik, Datenbanken oder APIs.
- **Integrationsfehler**
  Tritt bei der Kommunikation zwischen verschiedenen Systemkomponenten auf.
- **Häufigkeit**
  Der Fehler tritt extrem häufig oder nur sehr selten auf. (Evtl. wird diese Funktionalität nur von einer Randgruppe verwendet)

9.2. Dringlichkeit

Die Dringlichkeit beschreibt, wie schnell ein Fehler behoben werden sollte. Sie wird nicht nur durch die Schwere des Fehlers bestimmt, sondern auch durch geschäftliche Prioritäten. Oft gibt es Fälle, in denen ein Fehler mit einem niedrigen Schweregrad dennoch eine hohe Dringlichkeit hat, weil er für den Kunden von besonderer Bedeutung ist. Die Dringlichkeit wird durch die Fehlerklassifizierung gesteuert. Typische Dringlichkeitsstufen sind:

1. **Blocker**
   Dieser Fehler muss sofort behoben werden, da er wesentliche Geschäftsprozesse stoppt oder das System unbrauchbar macht. Es sollte alles andere stehen und liegen gelassen werden um den Fehler zu beheben.
2. **Hoch**
   Der Fehler beeinträchtigt zentrale Funktionen, die für den Benutzer oder den Geschäftsbetrieb

wichtig sind, aber eine Notlösung oder Workaround ist möglich. Der Fehler muss schnell behoben werden, wird aber nach den Blocker-Fehlern priorisiert.

3. **Mittel**
   Der Fehler hat keine sofortigen, gravierenden Auswirkungen auf das Geschäft, verursacht jedoch Unannehmlichkeiten für Benutzer. Ein solcher Fehler wird in einem regulären Release- oder Wartungszyklus behoben.

4. **Niedrig**
   Der Fehler hat nur geringe Auswirkungen oder betrifft Funktionen, die selten oder kaum genutzt werden. Es besteht kein Handlungsdruck. Dieser Feher wird bearbeitet, wenn die Zeit es zulässt, oft während eines Wartungszeitraums.

10. Testrollen und Verantwortlichkeiten

Tests sind nur so effektiv wie die Personen, die sie durchführen. Zudem gibt es verschiedene Aufgaben und Verantwortlichkeiten, die durch spezifische Rollen abgedeckt werden müssen. Die Rollen helfen, sicherzustellen, dass jeder weiss, was von ihm erwartet wird und welche Aufgaben er zu erfüllen hat. Durch die Definition von Rollen und deren jeweiligen Zuständigkeiten wird der Kommunikationsfluss innerhalb des Teams und mit anderen Projektbeteiligten verbessert. Jeder kennt die Ansprechpersonen für bestimmte Fragen oder Probleme, was den Informationsaustausch beschleunigt und Missverständnisse vermeidet. Dies ist insbesondere bei grösseren Projekten oder verteilten Teams von grosser Bedeutung.

- **Tester**
  Führt die Tests durch und dokumentiert die Ergebnisse.
- **Testmanager**
  Koordiniert den gesamten Testprozess und sorgt für die Einhaltung der Testpläne.
- **Testautomatisierer**
  Implementiert automatisierte Tests.
- **Entwickler**
  Verantwortlich für Unit-Tests und die Behebung gefundener Fehler.
- **Kunde**
  Führt Akzeptanztests durch, um sicherzustellen, dass die Software den Anforderungen entspricht.

10.1. Verantwortlichkeit für Entscheidungsfindung

In einem Testprozess müssen oft wichtige Entscheidungen getroffen werden, wie zum Beispiel:

- Wann der Testzyklus startet und endet
- Welche Fehler als kritisch eingestuft werden
- Ob ein Release freigegeben werden kann

Die Rolle des Testmanagers sorgt dafür, dass solche Entscheidungen auf der Basis klarer Kriterien und einer fundierten Risikoabschätzung getroffen werden.

10.2. Verantwortlichkeit

Eine klare Rollenzuweisung ermöglicht es, Aufgaben und Verantwortlichkeiten nachzuvollziehen. Wenn beispielsweise ein Testfehler auftritt oder bestimmte Testziele nicht erreicht werden, ist klar, wer die Verantwortung trägt und wer zur Lösung beitragen muss. Dies erhöht die Verantwortlichkeit im Team und trägt zu einer besseren Nachverfolgbarkeit der Testergebnisse bei.

11. Testfall

Folgende Punkte sollten in einem Testfall enthalten sein. Dies beschreibt einen Testfall wie er im Testkonzept resp. für eine Ausführung definiert werden sollte. Nicht wie dieser im Protokoll erscheinen soll.

11.1. Testfall-ID

Eindeutige Kennung für den Testfall.

11.2. Beschreibung des Testziels

Kurze Zusammenfassung des Ziels des Testfalls.

Beispiel: "Prüfen, ob der Login-Button korrekt funktioniert, wenn gültige Anmeldedaten eingegeben werden."

11.3. Voraussetzungen

Alle Bedingungen, die erfüllt sein müssen, bevor der Test durchgeführt wird.

Beispiel: "Der Benutzer ist registriert und die Anwendung ist gestartet."

11.4. Testdaten

Daten, die während des Tests verwendet werden.

Beispiel: * Benutzername: 'user123' * Passwort: 'password456'

11.5. Testschritte

Detaillierte Anweisungen zur Durchführung des Tests.

Beispiel: . Öffnen Sie die Login-Seite. . Geben Sie 'user123' im Feld 'Benutzername' ein. . Geben Sie 'password456' im Feld 'Passwort' ein. . Klicken Sie auf 'Login'.

11.6. Erwartetes Ergebnis

Das spezifische Ergebnis, das erwartet wird.
Beispiel: "Der Benutzer wird erfolgreich auf die Startseite weitergeleitet."

12. Internationale Standards und Best Practices

In der Welt des Softwaretests gibt es zahlreiche Standards, die sicherstellen sollen, dass Tests effizient und konsistent durchgeführt werden. Zu den wichtigsten Standards zählen:

12.1. ISO/IEC/IEEE 29119 - Software Testing

Ein umfassender Standard, der Testprozesse, Testdokumentation und Testtechniken definiert. Er umfasst: * **Testprozesse**
Planung, Durchführung und Steuerung. * **Testdokumentation**
Vorlagen für Testpläne, Protokolle und Berichte. * **Testtechniken**
Statische und dynamische Methoden.

12.2. ISO/IEC 25010 - Qualitätsmodell

Dieser Standard beschreibt Qualitätsmerkmale von Software (Funktionalität, Zuverlässigkeit, Benutzbarkeit) und hilft Testern, relevante Eigenschaften zu definieren und zu testen.

12.3. ISO/IEC 12207 - Software-Lebenszyklusprozesse

Definiert Verifizierungs- und Validierungsprozesse und integriert Tests in den Entwicklungszyklus. Wesentliche Aspekte sind Qualitätssicherung und Prozessintegration.

12.4. IEEE 829 - Testdokumentation

Bietet Vorlagen für Testpläne, Testfallbeschreibungen und Fehlerberichte. Obwohl durch ISO 29119 ersetzt, ist dieser Standard weiterhin eine wichtige Referenz für Testdokumentation.

12.5. ISTQB

Das ISTQB-Rahmenwerk liefert weltweit anerkannte Leitlinien und Begriffe für das Softwaretesten, die in vielen Testkonzepten verwendet werden.

12.6. CMMI und TMMi

Diese Modelle zur Reifegradbewertung fokussieren sich auf die Verbesserung der Testprozesse. TMMi bewertet speziell die Reife des Testprozesses, während CMMI allgemeine Entwicklungsprozesse abdeckt.

12.7. HERMES

HERMES ist ein Projektmanagement- und Testmodell aus der Schweiz. Es integriert Testaktivitäten in verschiedene Projektphasen und legt Rollen im Testprozess fest. Es wird vor allem im

öffentlichen Sektor eingesetzt.

Diese Standards bieten eine Orientierung, sollten jedoch flexibel auf die spezifischen Projektanforderungen angepasst werden.