ISTQB TECHNICAL TESTANALYST

Zusammenfassung

Version: 1.0

Donnerstag, 6. Oktober 2016

Autor:
Raphael Grüter

FIRMTEC
SOLUTIONS AG

Inhaltsverzeichnis

Historie

Version	Datum	Bemerkungen
1.0	06.10.2016	Initiale Version

Template Version: 1.0

Disclaimer:

Dieses Dokument wurde von FirmTec Solutions AG mit grösstmöglicher Sorgfalt erstellt. Dennoch übernimmt die FirmTec Solutions AG keine Gewähr für die Aktualität, Vollständigkeit und Richtigkeit der bereitgestellten Seiten und Inhalte.

Für Fragen und Anregungen stehen wir Ihnen gerne unter der Emailadresse info@firmtec.ch zur Verfügung.

FIRMTEC
SOLUTIONS AG

1. TTA beim Risikoorientierten Test

1.1. Einführung

- Risikoidentifikation | Risikobewertung (-analyse) | Risikobeherrschung
- Iterativer Prozess, Fokus auf Produktrisiken, Änderung der Prioritäten, regelmässige Bewertung und Kommunikation des Risikostatus
- TTA → technische Risiken (→ Wahrscheinlichkeit), mit Fokus auf Sicherheit, Zuverlässigkeit, Performanz

1.2. Risikoidentifizierung

- TTA → spezifische technische Fähigkeiten → besonders gut um Experten-Interviews zu machen, Brainstorming mit Kollegen & aktuelle sowie vergangene Erfahrungen zu analysieren
- TTA arbeiten mit Fachkollegen des technischen Bereichs (→ Entwickler, Systemarchitekten, Betriebsingenieure) zusammen, um die technischen Risiken zu bestimmen
- Risiken, die zu identifizieren sind:
  - Performanzrisiken (z.B. Antwortzeiten bei hoher Last)
  - Sicherheitsrisiken (z.B. Offenlegung vertraulicher Daten durch Sicherheitsangriffe)
  - Zuverlässigkeitsrisiken (z.B. Anwendung erfüllt nicht die in SSL beschriebene Verfügbarkeit)

1.3. Risikobewertung (-analyse)

- Bestimmung von Eintrittswahrscheinlichkeit (technisches Risiko) & Schadensausmass (geschäftliches Risiko)
- TTA → Eintrittswahrscheinlichkeit bestimmen
- Faktoren, die bei Risikobewertung berücksichtigt werden:
  - Komplexität der Technologie | Codestruktur
  - Konflikte zwischen Stakeholdern bez. technischer Anforderungen
  - Kommunikationsprobleme → räumliche Trennung der MA's
  - Werkzeuge & Technologie
  - Zeitdruck, knappe Ressourcen, Druck durch Management
  - Fehlen einer frühzeitigen Qualitätssicherung
  - Häufige Änderungen in Zusammenhang mit technischen Qualitätsmerkmalen
  - Technische Schnittstellen-/Integrationsproblematik

1.4. Risikobeherrschung

- Reduzierung der Risiken durch priorisierte Ausführung der Tests bez. Risikostufe & Umsetzung der Massnahmen zur Risikobeherrschung & Vorkehrung gegen Risiken

- Bewerten der Risiken durch während Projektverlauf hinzugewonnener Erkenntnisse

2. Strukturbasierter Test

2.1. Einführung

- Werden auch als White-Box oder codebasierte Testverfahren bezeichnet
- Grundlage: Code, Daten, Architektur &/ Kontroll- und Datenfluss
- Liefern Kriterien für den Überdeckungsgrad
- Folgende Testentwurfsverfahren werden hier behandelt:
    - Einfacher Bedingungstest | Bedingungs-/Entscheidungstest | Modifizierter Bedingungs-/Entscheidungstest (MC/DC) | Mehrfachbedingungstest
      → basieren alle auf WAHR/FALSCH-Bedingungen, decken gleiche Arten von Fehlerzuständen auf
      → Verfahren werden immer gründlicher → mehr Testfälle nötig
    - Pfadtest | API-Test

2.2. Einfacher Bedingungstest

- Entscheidungstest (→ Zweigtest) → Betrachtung der Entscheidung als Ganzes
- Einfacher Bedingungstest → Befassung damit, wie die Entscheidung getroffen wird
- Jede Entscheidung besteht aus einer oder mehreren atomaren Bedingungen, die zu einem booleschen Wert führen

Anwendbarkeit:

- Nur in abstrakter Hinsicht interessant

Einschränkungen/Schwierigkeiten:

- 100% Bedingungsüberdeckung ([A], [B]: WAHR/FALSCH) ohne 100% Entscheidungsüberdeckung ([A und B]: Nur FALSCH):

	Atomare Teilbed. [A]	Atomare Teilbed. [B]	Entscheidung [A und B]
Test 1	FALSCH	WAHR	FALSCH
Test 2	WAHR	FALSCH	FALSCH

2.3. Bedingungs-/Entscheidungstest

- Sowohl 100% Bedingungs- als auch 100% Entscheidungsüberdeckung muss erreicht werden
- 100% Bedingungsüberdeckung ([A], [B]: WAHR/FALSCH) und 100% Entscheidungsüberdeckung ([A und B]: WAHR/FALSCH):

	Atomare Teilbed. [A]	Atomare Teilbed. [B]	Entscheidung [A und B]
Test 1	WAHR	WAHR	WAHR

Test 2	FALSCH	FALSCH	FALSCH

Anwendbarkeit:

- Wenn der Code zwar wichtig, aber nicht kritisch ist

Einschränkungen/Schwierigkeiten:

- Braucht möglicherweise mehr Testfälle → Problematisch wenn Zeit knapp wird

2.4. Modifizierter Bedingungs-/Entscheidungstest

- Liefert höhere Kontrollflussüberdeckung
- Wenn N atomare Teilbedingungen vorliegen, braucht es normalerweise N+1 Testfälle
- Neben 100% Bedingungs-/Entscheidungsüberdeckung, müssen noch folgende Punkte abdeckt werden:
    1. Mind. 1 Test, bei dem sich der Entscheidungsausgang ändert, wenn atomare Teilbed. X WAHR ist
    2. Mind. 1 Test, bei dem sich der Entscheidungsausgang ändert, wenn atomare Teilbed. X FALSCH ist
    3. Für jede atomare Teilbed. Gibt es Tests die die oberen beiden Punkte erfüllen
- 100% Bedingungsüberdeckung ([A], [B], [C]: WAHR/FALSCH) und
  100% Entscheidungsüberdeckung ([(A oder B) und C]: WAHR/FALSCH):

	At. Teilbed. [A]	At. Teilbed. [B]	At. Teilbed. [C]	(A oder B) und C
Test 1	WAHR	FALSCH	WAHR	WAHR
Test 2	FALSCH	WAHR	WAHR	WAHR
Test 3	FALSCH	FALSCH	WAHR	FALSCH
Test 4	WAHR	FALSCH	FALSCH	FALSCH

- In Test 1 ist A WAHR und der Ausgang ist WAHR. Wenn A FALSCH wird (wie in Test 3; andere Werte bleiben unverändert), wird das Ergebnis FALSCH
- In Test 2 ist B WAHR und der Ausgang ist WAHR. Wenn B FALSCH wird (wie in Test 3; andere Werte bleiben unverändert), wird das Ergebnis FALSCH
- In Test 1 ist C WAHR und der Ausgang ist WAHR. Wenn C FALSCH wird (wie in Test 4; andere Werte bleiben unverändert), wird das Ergebnis FALSCH

Anwendbarkeit:

- In der SW Entwicklung für die Luft- und Raumfahrtindustrie sowie für andere sicherheitskritische Systeme
- Einsetzbar für sicherheitskritische SW, wo Fehler eine Katastrophe auslösen könnte

Einschränkungen/Schwierigkeiten:

- Schwierig wenn ein spezifisches Element in einem Ausdruck mehrfach vorkommt → gekoppeltes Element
- Je nach Entscheidungsanweisung im Code nicht möglich das gekoppelte Element so zu variieren, dass dies allein zu einer Änderung des Entscheidungsausgangs führt
  → Mögliche Lösung: Nur atomare Teilbedingungen testen, welche keine solche Kopplung enthalten

→ Alternative Lösung: Jede Entscheidung mit gekoppelten Elementen wird von Fall zu Fall analysiert

- Verkürztes Auswertungsverfahren → Wenn Entscheidung «A und B» bewertet werden soll, gibt es keinen Grund B zu bewerten, wenn A als FALSCH bewertet wird. Kein Wert von B kann den endgültigen Wert ändern
  → Ausführungszeit kann eingespart werden, wenn B nicht bewertet werden muss

2.5. Mehrfachbedingungstest

- Alle möglichen Kombinationen von Wahr/Falsch-Bedingungen testen
- Anzahl benötigter Tests = 2^N (N = Anzahl ungekoppelter atomarer Teilbedingungen)
- Folgende Tabelle heisst Wahrheitstabelle / Wahrheitstafel / Wahrheitsmatrix
- 100% Bedingungsüberdeckung ([A], [B], [C]: WAHR/FALSCH) und
  100% Entscheidungsüberdeckung ([(A oder B) und C]: WAHR/FALSCH) und
  100% Überdeckung aller Kombinationen:

	At. Teilbed. [A]	At. Teilbed. [B]	At. Teilbed. [C]	(A oder B) und C
Test 1	WAHR	WAHR	WAHR	WAHR
Test 2	WAHR	WAHR	FALSCH	FALSCH
Test 3	WAHR	FALSCH	WAHR	WAHR
Test 4	WAHR	FALSCH	FALSCH	FALSCH
Test 5	FALSCH	WAHR	WAHR	WAHR
Test 6	FALSCH	WAHR	FALSCH	FALSCH
Test 7	FALSCH	FALSCH	WAHR	FALSCH
Test 8	FALSCH	FALSCH	FALSCH	FALSCH

- Bei verkürzter Auswertung verringert sich die Anzahl Tests, je nach Reihenfolge und Gruppierung der logischen Operationen

Anwendbarkeit:

- Traditionell für embedded SW, welche lange, zuverlässig, ohne Abstürze laufen soll (→ Telefonswitches für 30a)
- Wird bei den meisten kritischen Anwendungen durch den modifizierten Bedingungs-/Entscheidungstest ersetzt

Einschränkungen/Schwierigkeiten:

- Überdeckungsgrad einfach bestimmbar, jedoch sehr viele Testfälle nötig → mod. Bed.-/Entsch.-überdeckung geeigneter

2.6. Pfadtest

- Identifikation von Pfaden im Code → Abdeckung durch Tests
- Ideal alle Pfade abzudecken, bei nichttrivialen Systemen kann #Testfälle sehr gross werden
- Empfohlen, das möglichst viele Pfade durch das SW Modul von Anfang bis Ende durchlaufen werden
- Folgende Vorgehensweise empfiehlt sich dafür:

1. Als ersten Pfad den einfachsten, funktional sinnvollen Pfad von Anfang bis Ende auswählen
2. Jeden zusätzlichen Pfad als leichte Variation des vorherigen Pfades auswählen. Dabei möglichst nur ein Zweig im Pfad verändern. Kurze Pfade bevorzugt. Funktional sinnvolle Pfade bevorzugen
3. Funktional sinnlose Pfade nur wählen, wenn dies für die Überdeckung erforderlich ist
4. Bei Pfadauswahl intuitiv vorgehen → Wahrscheinlichste Pfade zuerst auswählen

- Manche Pfadsegmente werden mehrfach ausgeführt → Jedes Codesegment mindestens 1x ausführen

Anwendbarkeit

- Dieser Partielle Pfadtest wird häufig für sicherheitskritische SW-Anwendungen durchgeführt
- Gute Ergänzung zu den anderen Methoden, da hier Pfade betrachtet werden anstatt einzelne Entscheidungen

Einschränkungen/Schwierigkeiten

- Obwohl ein Kontrollflussgraph verwendet werden kann, um die Pfade zu bestimmen, wird in der Realität eher ein Werkzeug benötigt, um die Pfade von komplexen Modulen zu berechnen

Überdeckung

- Bei genügend Tests zur Überdeckung aller Pfade gibt es auch 100% Anweisungs- und Zweigüberdeckung
- Partieller Pfadtest liefert gründlicheres Testen als der Zweigtest bei einer nur geringfügig erhöhten Anzahl Tests

2.7. API-Test

- API = Application Programming Interface / Programmschnittstelle
- Beim Umgang mit APIs sind Negativtests häufig von entscheidender Bedeutung
- Programmierer wollen ev. diese Schnittstelle für nicht vorgesehene Art und Weise benutzen
  → robuste Fehlerbehandlung unbedingt erforderlich
- Gründliches Testen der Wiederherstellungs- und Wiederholungsmechanismen nötig, da APIs häufig lose miteinander gekoppelt sind, was zu verlorenen Transaktionen und Timing-Fehler führen kann
- Häufig gründliche Zuverlässigkeitstest seitens API-Anbieter notwendig

Anwendbarkeit

- API-Test gewinnt an Bedeutung, da mehr Systeme verteilt arbeiten / remote
- Beispiele: Betriebssystemaufrufe, service-orientierte Architekturen (SOA), Programmfernaufrufe (Remote Procedure Calls (RPC)), Webservices, etc. → API-Test besonders für Multisysteme geeignet

Einschränkungen/Schwierigkeiten

- Normalerweise spezialisierte Werkzeuge nötig

- Kein GUI → Werkzeuge: für Aufsetzen der Anfangsumgebung, Daten-Marshalling, API-Aufrufe auszuführen und Ergebnisse zu bestimmen

Überdeckung

- API beschreibt Testart, kein Überdeckungsgrad bestimmbar

Fehlerarten

- Sehr unterschiedliche Fehler aufdeckbar, häufig Schnittstellenprobleme, Probleme mit Datenhandling, Timing, Verlust / Duplizierung von Transaktionen

2.8. Strukturbasierte Verfahren auswählen

- Je kritischer ein System, desto höher der benötigte Überdeckungsgrad, desto mehr Aufwand nötig
- Avionik Standards: DO-178B / DO-178C bzw. ED-12B für Europa → fünf Stufen der Kritikalität:
    - A- Katastrophal: Das Versagen verursacht das Fehlen einer kritischen Funktionalität, die für einen sicheren Flug oder eine sichere Landung benötigt wird
    - B- Gefährlich: Das Versagen hat eine grosse negative Auswirkung auf Sicherheit oder Performanz
    - C- Bedeutend: Das Versagen ist bedeutend, aber nicht so schwerwiegend wie A oder B
    - D- Geringfügig: Das Verssagen ist wahrnehmbar, hat aber weniger schwerwiegende Auswirkungen als C
    - E- Keine Auswirkung: Das Versagen hat keine Auswirkung auf die Sicherheit
- Stufe A → modifizierte Bedingungs-/Entscheidungsüberdeckung (MC/DC)
- Stufe B → Entscheidungsüberdeckung vorgeschrieben, mod. Bedingungs-/Entscheidungsüberdeckung optional
- Stufe C → Anweisungsüberdeckung vorgeschrieben
- Internationaler Standard: IEC-61508 → funktionale Sicherheit von programmierbaren, elektronischen und sicherheitsbezogenen Systemen. Wurde in vielen verschiedenen Branchen angepasst, z.B. für Automobilindustrie, Eisenbahnbranche, Produktionsprozesse, Kernkraftwerke, Maschinenbau. Safety Integrity Level (SIL) von 1 (gering) bis 4 (höchste Kritikalität):
    1. Anweisungs- und Zweigüberdeckung empfohlen
    2. Anweisungsüberdeckung sehr empfohlen, Zweigüberdeckung empfohlen
    3. Anweisungs- und Zweigüberdeckung sehr empfohlen
    4. Modifizierte Bedingungs-/Entscheidungsüberdeckung sehr empfohlen
- API-Test immer nötig wenn ein Teil der Verarbeitung remote ist → Kritikalität bestimmt dabei den Testaufwand

3. Analytische Testverfahren

3.1. Einführung

- Es gibt 2 Arten von Analyse: Statische & Dynamische

3.2. Statische Analyse

- Ziel der statischen Analyse:
  - A- Tatsächliche oder potentielle Fehlerzustände in Programmcode und Systemarchitektur zu finden
  - B- Wartbarkeit/Änderbarkeit des Codes verbessern
- Statische Analyse wird im Allgemeinen durch Werkzeuge unterstützt

3.2.1. Kontrollflussanalyse

- Statisches Analyseverfahren, bei dem der Kontrollfluss eines SW-Programms, entweder mit Hilfe eines Kontrollflussgraphen oder mit Werkzeugunterstützung analysiert wird
- Auffindbare Anomalien: schlecht konzipierte Schleifen (z.B. mit mehreren Eingangspunkten), unklare/inkorrekt deklarierte Ziele von Funktionsaufrufen in bestimmten Sprachen, inkorrekte Ablaufsequenzen, etc.
- Kontrollflussanalyse wird oft verwendet um die zyklomatische Komplexität zu ermitteln → Positive ganze Zahl, welche die Anzahl unabhängiger Kontrollflusspfade in einem stark zusammenhängenden Kontrollflussgraphen angibt
- Schleifen und Iterationen werden nicht mehr berücksichtigt nachdem sie einmal durchlaufen wurden
- Jeder unabhängige Pfad von Anfang bis Ende ist ein einzigartiger Pfad und sollte getestet werden
- Je höher die zykl. Komplexität, desto schwieriger die Wartung, desto mehr Fehlerzustände sind zu erwarten
- NIST (National Institute of Standards and Technology) empfiehlt <=10 → Höhere Zahl → Aufteilung des Moduls

3.2.2. Datenflussanalyse

- Statische Analyse der Variablen, welche jeweils in 3 best. Aktionen unterteilt werden (→ Define-Use-Verfahren)
  - D (Define): Wenn Variable definiert / initialisiert wird
  - U (Use): Wenn Variable verwendet / gelesen wird
  - K (Killed): Wenn Variable gelöscht oder zerstört wird oder nicht mehr erreichbar ist
- Diese 3 Aktionen werden zu Paaren zusammengefasst («Definition-Verwendungspaar»), um Datenflusspfade darzustellen. Bsp.: du-Pfad → Programmcode für Definition und Verwendung wird angeschaut
- Mögliche Datenflussanomalien: Durchführung der korrekten Aktion der Variablen zum falschen Zeitpunkt oder Datenverwendung einer Variablen bei einer inkorrekten Aktion
- Mögliche Anomalien sind:
  - Zuweisung eines ungültigen Wertes zu einer Variablen
  - Verwendung einer Variablen ohne das ihr zuvor ein Wert zugewiesen wurde
  - Inkorrekte Pfade aufgrund eines inkorrekten Werts in einer Kontrollfluss-Entscheidung
  - Versuch, eine Variable zu verwenden nachdem diese zerstört wurde
  - Referenzierung von Variablen wenn diese nicht mehr erreichbar sind
  - Deklarieren und Zerstören von Variablen ohne diese zu verwenden

  - Variablen erneut definieren bevor diese verwendet wurden
  - Nicht-Löschung einer dynamisch zugewiesenen Variablen (Ursache für mögliche Speicherlecks)
  - Ändern einer Variablen, was zu unerwarteten Nebenwirkungen führt (z.B. Nachwirkungen und Folgeprobleme nachdem eine globale Variable geändert wurde, ohne alle Verwendungen dieser Variablen zu bedenken)
- Die Programmiersprache beeinflusst die Datenflussanalyse, z.B. durch 2x definierte Variablen, 1x in spez. Pfad
- Datenflusstest verwendet Kontrollflussgraphen, um die unplausiblen Datenaktionen zu erforschen
- Durch Multiprozessoren & Multi-Tasking gibt es Echtzeitsituationen, die durch Datenfluss- / Kontrollflussanalyse nicht aufgedeckt werden können

3.2.3. Wartbarkeit/Änderbarkeit durch statische Analyse verbessern

- Mit Werkzeugunterstützung kann die Einhaltung von Programmierkonventionen und -richtlinien analysiert werden
  → Ziel: Wartbarkeit des Codes verbessern
- Statische Analysewerkzeuge zeigen im Allgemeinen eher Warnungen als Fehlerzustände
- Modularer Aufbau verbessert die Wartbarkeit, statische Analysewerkzeuge unterstützen dabei:
  - Suche nach Wiederholungen im Code → Strukturverbesserungen (Refactoring) zu Modulen
    → Vorsicht: Overhead der Modulaufrufe kann sich negativ auf die Laufzeit auswirken!
  - Erzeugung von Metriken für Kopplung und Kohäsion, die wertvolle Indikatoren sind
    → Gute Wartbarkeit = weniger gekoppelte Module (→ Module, die während Ausführung des Codes aufeinander angewiesen sind)
    → Gute Wartbarkeit = hohe Kohäsion (→ eigenständige Module für die Durchführung einer einzigen Aufgabe)
  - Zeigen bei objektorientiertem Programmcode an, wo abgeleitete Objekte zu viel / wenig Sichtbarkeit zur übergeordneten «Elternklasse» haben
  - Identifikation von Bereichen mit hoher struktureller Komplexität (zyklomatische Kompl.) → Indikator für schlechte Wartbarkeit & höhere Fehlerdichte
- Auch Wartung von Webseiten kann verbessert werden → Baumstruktur der Webseite ausgeglichen? → Unausgewogenheit kann:
  - Testaufgaben schwieriger machen
  - Höheren Wartungsaufwand erzeugen
  - Navigation für den Nutzer erschweren

3.2.4. Aufrufgraphen

- = Statische Repräsentation der Kommunikationskomplexität
- Sind gerichtete Graphen, in denen Knoten = Programmeinheiten, Kanten = Kommunikationsbeziehungen zwischen den Einheiten
- Komponententest → versch. Funktionen / Methoden rufen einander auf

- Integrations- & Systemtest → separate Module rufen sich einander auf
- Systemintegrationstest → separate Systeme rufen einander auf
- Zwecke von Aufrufgraphen:
  - Tests entwerfen, die ein bestimmtes Module oder System aufrufen
  - Herausfinden wie viele Stellen in der SW das Modul / System aufrufen
  - Struktur des Gesamtcodes und seiner Architektur bewerten
  - Vorgehensweisen für die Reihenfolge der Integration vorschlagen
- Es gibt bisher 2 Kategorien des Integrationstests: inkrementell (Top-Down, Bottom-Up, usw.) & nicht-inkrementell (Big-Bang) → Inkrementelle Methoden sind vorzuziehen da sie den Code in Inkrementen integrieren, was die Fehlereingrenzung erleichtert
- Hier werden nun 3 weitere Methoden vorgestellt, die Aufrufgraphen verwenden:
  - Paarweiser Integrationstest → Konzentration auf Komponentenpaare, die miteinander interagieren
  - Umgebungsintegrationstest → Basieren auf Knoten, welche mit einem best. Knoten verbunden sind. Basis vom Test = alle Vorgänger- & Nachfolger-Knoten eines bestimmten Knotens
  - **McCabe's Entwurfsansatz** → Nutzt die Theorie der zyklomatischen Komplexität. Erstellung eines Aufrufgraphen nötig, der darstellt, wie Module sich gegenseitig aufrufen können:
    - Bedingungsloser Modulaufruf → Modulaufruf durch anderes Modul findet immer statt
    - Bedingter Modulaufruf → Modulaufruf durch andere Modul findet manchmal statt
    - Sich gegenseitig ausschliessender bedingter Modulaufruf → Modul ruft von versch. Modulen eines auf (nur eines!)
    - Iterativer Modulaufruf → Modul ruft ein anderes mindestens einmal auf
    - Iterativer bedingter Modulaufruf → Modul kann anderes Modul 0 – x Male aufrufen

3.3. Dynamische Analyse

3.3.1. Überblick

- Wird eingesetzt, um Fehlerwirkungen aufzudecken, deren Symptome nicht sofort ersichtlich sind
- Bsp.: Speicherlecks können durch statische Analyse erkennbar sein, mit dynamischer Analyse leicht zu erkennen
- Nicht ohne weiteres reproduzierbare Fehlerwirkungen können erhebliche Konsequenzen für Testaufwand haben
- Ursache kann sein: Speicherlecks, inkorrekter Einsatz von Zeigern, System-Stacks, etc.
- Diese Ursachen können allmähliche Verschlechterung der Systemleistung oder zu Systemabstürzen führen
- Dynamische Analyse wird gemacht um:
  - Fehlerwirkungen zu verhindern, indem fehlerhafte «wilde» Zeiger aufgedeckt werden
  - Systemausfälle zu analysieren, die nicht leicht reproduzierbar sind
  - Netzwerkverhalten zu bewerten

  - Systemleistung zu verbessern, indem Info über Laufzeitverhalten des Systems erfasst werden
- Dynamische Analyse kann in jeder Teststufe gemacht werden. Es sind technische Fähigkeiten und Systemkenntnisse erforderlich, um folgende Aufgaben zu erfüllen:
  - Spezifizieren der Testziele für dyn. Analyse
  - Geeigneter Zeitpunkt für Beginn und Ende der Analyse zu bestimmen
  - Testergebnisse zu analysieren

3.3.2. Speicherlecks aufdecken

- Speicherlecks treten auf, wenn der Speicher (RAM) zwar allokiert, aber nicht wieder freigegeben wird
- Durch Garbage Collector wird diese Aufgabe heutzutage vom Programm übernommen
- Es kann schwierig sein, Speicherlecks zu identifizieren, wenn vorhandener, allokierter Speicherplatz durch automatische Speicherbereinigung freigegeben wird
- Speicherlecks verursachen Probleme die erst allmählich erkennbar werden, z.B. wenn SW erst kürzlich installiert oder System neu gestartet wurde, was beim Testen oft geschieht
- Symptom für Speicherlecks → kontinuierliche Verlangsamung der Antwortzeiten
- Identifikation für Speicherlecks → dyn. Analysewerkzeuge / einfacher Speichermonitor
- Andere Engpässe, die beachtet werden müssen: Dateibezeichner, Zugriffsgenehmigungen (Semaphore), Verbindungspools für Ressourcen

3.3.3. Wilde Zeiger aufdecken

- Folgen von wilden Zeigern (Objekte gibt es nicht mehr / falscher Speicherort):
  - Programm funktioniert nicht wie erwartet
  - Programm kann abstürzen
  - Programm funktioniert nicht korrekt
  - Inkorrekte Werte werden geliefert
- Jede Änderung an der Speichernutzung (→ neuer Build / SW-Änderung) kann eine dieser Folgen haben. Vor allem dann kritisch, wenn Programm zunächst wie erwartet funktioniert obwohl es fehlerhafte Zeiger hat, und dann nach Update abstürzt
- Werkzeuge können fehlerhafte Zeiger identifizieren

3.3.4. Systemleistung analysieren

- Dynamische Analyse nicht nur zum Aufdecken von Fehlern, sondern auch um Programmleistung zu analysieren → z.B. dynaTrace → Info, dass bestimmte Module häufig aufgerufen werden → diese optimieren bringt viel
  → Info über das dynamische Verhalten hilft Module zu identifizieren, welche umfangreicher getestet werden sollten
- Dynamische Analyse wird häufig während des Systemtests durchgeführt

4. Qualitätsmerkmale bei technischen Tests

4.1. Einführung

- Fokus von TTA = wie funktioniert das Produkt, weniger auf funktionale Aspekte
- Beim Komponententest von Echtzeit- & eingebetteten Systemen → Performanz & Ressourcennutzung wichtig
- Beim Systemtest und betrieblichen Abnahmetest →Zuverlässigkeitsmerkmale (z.B. Wiederherstellbarkeit) wichtig
- Dazu gehören Server, Clients, Datenbanken, Netzwerke und andere Ressourcen
- Qualitätsmerkmalbeschreibung orientiert sich am ISO Standard 9126 → gilt als Leitfaden

Qualitätsmerkmal	Untermerkmale	Test Analyst	Techn. Test Analyst
Funktionalität	Richtigkeit, Angemessenheit, Interoperabilität, Einhaltung von Standards (Konformität)	X	
	Sicherheit		X
Zuverlässigkeit	SW-Reife (Robustheit), Fehlertoleranz, Wiederherstellbarkeit, Einhaltung von Standards (Konformität)		X
Benutzbarkeit	Verständlichkeit, Erlernbarkeit, Operabilität, Attraktivität, Einhaltung von Standards (Konformität)	X	
Effizienz	Performanz (Zeitverhalten), Ressourcennutzung, Einhaltung von Standards (Konformität)		X
Wartbarkeit	Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit, Einhaltung von Standards (Konformität)		X
Portabilität	Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit, Einhaltung von Standards (Konformität)		X

- TM zuständig für Erstellung von Berichten, (T)TA zuständig für Erhebung der Informationen zu den Metriken

4.2. Allgemeine Planungsaspekte

- Darauf ist zu achten, wenn nicht-funktionale Tests geplant werden:
  - Anforderungen der Stakeholder
  - Beschaffung benötigter Werkzeuge und Schulungen
  - Anforderungen bez. Testumgebung
  - Organisatorische Faktoren
  - Fragen der Datensicherheit

4.2.1. Anforderungen der Stakeholder

- Nicht-funktionale Anforderungen oft unbekannt / kaum spezifiziert → TTA muss das bekannt machen!
- Üblicher Ansatz: bestehendes System als Referenz anschauen

4.2.2. Beschaffung benötigter Werkzeuge und Schulungen

- Kommerzielle Werkzeuge / Simulatoren besonders relevant für Performanz- und Sicherheitstests
- TTA soll Kosten der Vorlaufzeiten für Beschaffung, Training, Einführung einschätzen können
- Komplexer Simulator entwickeln = eigenes Entwicklungsprojekt

4.2.3. Anforderungen bez. Testumgebung

- Für viele nicht-funktionale Tests wird produktionsähnliche Umgebung benötigt, um brauchbare Messungen zu erhalten. Grösse & Komplexität kann Planung & Finanzierung massgeblich beeinflussen
- Alternativen, die in Betracht gezogen werden sollten:
    - Verwendung der echten Produktionsumgebung
    - Verwendung einer reduzierten Version des Systems

4.2.4. Organisatorische Faktoren

- Bei nicht-funktionalen Tests oft Verhalten mehrerer Komponenten eines Gesamtsystems wichtig: Server, DB, Netzw.
- Bei Komponenten an verschiedenen Standorten, welche nur zu gewissen Uhrzeiten gebraucht werden können fürs Testen → gute Planung wichtig!

4.2.5. Fragen der Datensicherheit

- Testdaten zu anonymisieren kann schwierig sein → muss als Teil der Testrealisierung geplant werden

4.3. Sicherheitstests

4.3.1. Einführung

- Unterschied zu anderen Formen funktionaler Tests:
    - Standardverfahren zur Auswahl der Testeingangsdaten können wichtige Sicherheitsaspekte ausser Acht lassen
    - Die Symptome von Sicherheitsfehlern unterscheiden sich grundlegend von Symptomen, die bei anderen funktionalen Tests gefunden werden

- Sicherheitstests untersuchen die Verwundbarkeit eines Systems durch diverse Gefährdungen, indem sie versuchen, die Security Policy eines Sytems gezielt ausser Kraft zu setzen
- Mögliche Bedrohungen:
  - Nicht autorisiertes Kopieren von Anwendungen / Daten
  - Nicht autorisierter Zugriff → Zugriffsrechte, Benutzerrechte, Privilegien
  - Gezeigte Seiteneffekte, die nicht beabsichtigt waren
  - Cross-Site-Scripting (XSS) → Code der von aussen in eine Webseite eingebracht wird
  - Überlauf des Eingabebereichs (Speicherüberlauf) durch Eingabe extrem langer Zeichenketten
  - Denial of Service-Angriff (DoS) → Anwendung kann nicht mehr genutzt werden
  - Unbemerktes Abhören (Man-in-the-middle-Angriff), Nachahmen &/ Abändern von vertraulichen Daten und Weitergabe
  - Verschlüsselungscodes knacken, die vertrauliche Daten schützen
  - Logische Fallen (logic bombs / easter eggs) → in böser Absicht in den Code eingeschleuste Dinge, die nur unter bestimmten Bedingungen aktiviert werden

4.3.2. Sicherheitstests planen

- Folgende Themen besonders relevant:
  - Sicherheitstests können schon beim Systementwurf entstehen → für Komponenten-, Integrations- und Systemtestphase einplanen, regelmässig für die Zeit nach dem Produktivgang wiederholen
  - Auch Code-Reviews und statische Analyse durch Sicherheitswerkzeuge einbinden → Sicherheitsprobleme in Systemarchitektur, Entwurfsdokumenten und im Programmcode effektiv aufdeckbar
  - Sicherheitsangriffe planen und durchführen, auch Tests der Benutzerrechte, Zugriffsberechtigungen und Privilegien
  - Einholen von Genehmigungen wichtig bei Planung von Sicherheitstests
  - Beachten: Verbesserungen der Sicherheit = Auswirkung auf Systemleistung → Performanztests nötig?

4.3.3. Spezifikation von Sicherheitstests

- Sicherheitstests lassen sich je nach Ursprung des Sicherheitsrisikos unterscheiden:
  - Benutzerschnittstelle → nicht autorisierter Zugriff und bösartige Eingaben
  - Dateisystem → Zugriff auf vertrauliche Daten und Dateien / Repositories
  - Betriebssystem → Unverschlüsseltes Ablegen von sicherheitskritischen Informationen (PWs, etc.)
  - Externe SW → Interaktion mit externen Komponenten, die das System nutzt. Kann auf Netzwerkebene sein oder auf Ebene der SW-Komponenten
- Mögliche Verfahren beim Entwickeln von Sicherheitstests:
  - Nützliche Infos zur Spezifikation von Tests zusammentragen, z.B. Namen der MA's, physikalische Adressen, Details zum internen Netzwerk, IP-Nummern, Typ der verwendeten SW/HW, Version der Betriebs-SW
  - Schwachstellen des Systems mit gängigen Werkzeugen scannen → dienen nicht dazu, in System direkt einzudringen, sondern zunächst mal dazu jene

Schwachstellen zu identifizieren, die Sicherheitsvorkehrungen durchbrechen. Spezifische Schwachstellen lassen sich auch mit Hilfe von Checklisten identifizieren, z.B. mit Checklisten des National Institute of Standards and Technology (NIST)
    - Sicherheitsangriffe entwickeln → mehrere Eingaben über verschiedene Schnittstellen machen, um so die schwersten Sicherheitsfehler aufzudecken

4.4. Zuverlässigkeitstests

- Qualitätsmerkmal der Zuverlässigkeit: Reife | Fehlertoleranz | Wiederherstellbarkeit
- SW-Reife messen → statistisches Mass der SW-Reife über gewisse Zeitspanne überwachen und mit gewünschter Zuverlässigkeit vergleichen:
    - Mittlere Betriebsdauer zwischen Ausfällen (Mean Time Between Failures, MTBF)
    - Mittlere Reparaturzeit nach einem Systemausfall (Mean Time to Repair, MTTR)
    - Messung der Fehlerdichte, z.B. wöchentliche Anzahl Fehlerwirkungen mit einem bestimmten Schweregrad
    - Diese Metriken können auch als Endekriterien dienen
- Fehlertoleranz testen (→ Robustheit) → Neben Negativtest bei funkt. Testing, auch Fehler bewerten, die ausserhalb der zu testenden Anwendung auftreten
- Werden normalerweise vom Betriebssystem gemeldet (Disk voll, Prozess / Dienst nicht verfügbar, Datei nicht gefunden, Speicher nicht verfügbar, etc.)
- Können auf Systemebene durch spezifische Werkzeuge unterstützt werden
- Wiederherstellbarkeitstests → Wie sieht es nach einem HW- / SW-Ausfall aus? Wiederherstellung? Zu Wiederherstellbarkeitstests gehören: Failover (Ausfallsicherheit), Backup & Restore
- Desaster Revovery Test → Test auf Wiederherstellbarkeit nach einem Totalausfall
- Präventive Massnahmen für HW Versagen → Lastverteilung auf mehrere Prozessoren, Servercluster, Festplatten, sodass bei Ausfall eine andere Komponente übernehmen kann → redundante Systeme
- Präventive Massnahmen für SW Versagen → Implementierung mehrere Instanzen, dissimilar redundante Systeme
- Redundante Systeme = Kombination von SW & HW
- Redundante Lösungen → zwei oder mehr Entwicklerteam setzen dieselben SW-Anforderungen um → gleiche Services mit unterschiedlichen Lösungsansätzen
- Ausfallsicherheitstests → Durch Herbeiführung von Fehlerwirkungen in kontrollierter Umgebung oder Simulationen der Fehlerwirkung testen. Nach Fehlerwirkung wird Ausfallsicherheitsmechanismus getestet → Datenverlust? Datenkorruption? SSL eingehalten?
- Backup- und Wiederherstellbarkeitstests → Bewertung der meist in Handbüchern dokumentierten Verfahrensweisen für Backups und Datenwiederherstellungsprozeduren, falls Datenverlust oder korrupte Daten auftreten
- Mögliche Metriken, die bei Backup- und Wiederherstellungstests gemessen werden:
    - Benötigte Zeit für die verschiedenen Backup-Arten (vollständiges, inkrementelles)
    - Benötigte Zeit für Wiederherstellung der Daten
    - Definierte Ebenen von garantierten Daten-Backups (Wiederherstellung aller Daten, welche nicht älter als 24h sind, etc.)

4.4.1. Zuverlässigkeitstests planen

- Folgende Aspekte besonders relevant:
  - Zuverlässigkeit kann auch überwacht werden nachdem die SW in Produktion gegangen ist
  - TTA kann Zuverlässigkeitswachstumsmodell (Reliability Growth Model) auswählen, welches die zu erwartenden Zuverlässigkeitswerte über eine Zeitspanne zeigt
  - Zuverlässigkeitstests sollen in produktionsähnlichen Umgebungen durchgeführt werden. Die Umgebung sollte so stabil wie möglich bleiben, damit Trends beobachtet werden können
  - Häufig Gesamtsystem nötig → Oft als Teil des Systemtests. Es können aber auch einzelne Komponenten oder integrierte Komponenten auf Zuverlässigkeit überprüft werden. Auch Reviews verwendbar
  - Lange Durchlaufzeiten nötig, sodass Zuverlässigkeitstests statistisch auswertbar werden

4.4.2. Spezifikation von Zuverlässigkeitstests

- Wiederholbares, vordefiniertes Set an Testfällen können verwendet werden
- Können zufällig aus einem Pool gewählt werden, oder es werden Testfälle aus einem statistischen Modell mit pseudo-zufälligen Methoden generiert
- Können auch auf Anwendungsmustern / Nutzerprofielen basieren

4.5. Performanztests

4.5.1. Einführung

- Performanz = Zeit- & Ressourcenverhalten, Teilmerkmal des Qualitätsmerkmals «Effizienz»

4.5.2. Arten von Performanztest

- Lasttests → ansteigende Grade erwarteter, realistischer Systemlasten
- Stresstests → steigende Überbelastung → Abnahme der Systemleistung, allmählich, vorhersehbar, ohne Ausfall
- Skalierbarkeitstests → Ziel: Kann das System wachsen? → Schwellenwerte können definiert werden

4.5.3. Performanztests planen

- Aspekte bei Planung von Performanztests:
  - Es kann erforderlich sein, dass das gesamte System implementiert ist, um brauchbare Ergebnisse zu erhalten
    → Performanztest meist auf Systemstufe

- So früh wie möglich durchführen, auch wenn noch keine produktionsähnliche Umgebung vorhanden ist
- Code-Reviews können Performanzprobleme identifizieren, v.a. «Wait/Retry-Logik» Probleme
- HW, SW, Netzwerkbandbreite müssen geplant und budgetiert werden
- Kosten für Beschaffung von HW und Werkzeugen
- Testinfrastruktur mieten?
- Sind Performanzwerkeuge mit Kommunikationsprotokollen kompatibel?
- Fehlerzustände haben oft erhebliche Auswirkungen auf SW → Wenn Anforderungen an Performanz des Systems sehr wichtig → sinnvoll die Performanz der kritischen Komponenten zu testen, nicht bis Systemtest warten

4.5.4. Spezifikation von Performanztests

- Anzahl Nutzer pro Nutzungsprofil lässt sich mit Monitorwerkzeugen bestimmen oder durch Vorhersagen
- Vorhersagen können auf Algorithmen basieren oder vom Fachbereich stammen → besonders wichtig für Spezifizierung der Nutzungsprofile bei Skalierbarkeitstests

4.6. Ressourcennutzung

- Bei eingebetteten Echtzeitsystemen spielt die Speichernutzung (Memory Footprint) eine wichtige Rolle bei Performanztests
- Auch die dynamische Analyse kann Ressourcennutzung untersuchen um Leistungsengpässe zu identifizieren

4.7. Wartbarkeitstests

- Grösserer Teil des Lebenszyklus einer SW = Wartung der SW
- Typische Wartbarkeitsziele:
  - Minimierung der Besitz-oder Betriebskosten der SW
  - Minimierung der Ausfallzeiten für SW-Wartung
- Wenn folgende Faktoren zutreffen, sollen Wartbarkeitstests in Teststrategie enthalten sein:
  - Änderungen der SW sind wahrscheinlich, nachdem diese in Produktion gegangen ist
    → Fehlerbehebungen oder geplante Aktualisierungen
  - Nach Ansicht der Stakeholder überwiegt der Nutzen, der sich aus dem Erreichen der oben erwähnten Ziele für den SW-Lebenszyklus ergibt
  - Das Risiko einer schlechten Wartbarkeit der SW rechtfertigt Wartbarkeitstests
- Geeignete Verfahren für Wartbarkeitstests = statische Analyse und Review
- Mit Wartbarkeitstest soll begonnen werden, sobald die Entwurfsdokumente zur Verfügung stehen
- Wartbarkeit soll frühzeitig im Lebenszyklus bewertet werden
- Dynamische Wartbarkeitstests untersuchen vorranging die dokumentierten Verfahren, die für die Wartung einer Anwendung entwickelt wurden → SW Update
- Testfälle = Wartungsszenarien

4.7.1. Analysierbarkeit, Modifizierbarkeit, Stabilität und Testbarkeit

- Wartbarkeit lässt sich auf unterschiedliche Arten messen:
  - Analysierbarkeit → Aufwand für die Diagnose von Problemen, die im System identifiziert wurden
  - Modifizierbarkeit → Aufwand für Programmänderungen
  - Testbarkeit → Aufwand für das Testen von Änderungen
  - Stabilität → Reaktion des Systems auf Änderungen. Systeme mit niedriger Stabilität zeigen nach jeder Änderung eine grosse Anzahl an Folgeproblemen
- Der benötigte Aufwand für Wartungsaufgaben hängt von mehreren Faktoren ab: verwendete SW-Design-Methodik, Programmierstandards
- Stabilität in diesem Kontext != Robustheit / Fehlertoleranz

4.8. Portabilitätstest

- Wie einfach ist es, eine SW in ihre vorgesehene Umgebung zu übertragen?
- Installierbarkeit, Koexistenz/Kompatibilität, Anpassbarkeit, Austauschbarkeit
- Portabilitätstests können schon mit einzelnen Komponenten beginnen (→ Austauschbarkeit einer bestimmten Komponente, z.B. Wechseln eines DB-Managementsystems auf ein anderes)

4.8.1. Installationstest

- Testen der Installation der SW und die dokumentierten Installationsanleitungen
- Typische Ziele:
  - Validieren, dass die SW erfolgreich installiert werden kann
  - Testen, ob die Installationssoftware richtig mit Fehlerwirkungen umgeht, die bei der Installation auftreten
  - Testen, ob sich eine nur teilweise Installation/Deinstallation der SW abschliessen lässt
  - Testen, ob sich ein Installations-Wizard eine ungültige HW-Plattform oder Betriebssystem-Konfigurationen erfolgreich identifizieren kann
  - Validieren, dass eine frühere Version installierbar ist («Downgrade») oder die SW sich deinstallieren lässt
- Nach Installationstest → Testen der Funktionalität
- Parallel zu den Installationstests laufen Benutzbarkeitstests
  → Validierung, dass die Anwender bei der Installation verständliche Anweisungen und Feedback/Fehlermeldungen erhalten

4.8.2. Koexistenz-/Kompatibilitätstest

- Computersysteme, die nicht miteinander interagieren, sind dann kompatibel, wenn sie in derselben Umgebung ausgeführt werden können, ohne sich gegenseitig zu beeinflussen
- Kompatibilitätstests werden dann ausgeführt, wenn neue SW oder eine Aktualisierung in einer neuen Umgebung installiert wird, in der bereits Anwendungen installiert sind
- Typische Ziele von Kompatibilitätstests:

- Bewertung möglicher negativer Auswirkungen auf die Funktionalität, wenn Anwendungen in derselben Umgebung geladen werden
- Bewertung der Auswirkungen auf jede Anwendung, die sich aus Modifikationen oder dem Installieren einer aktuelleren Betriebssystemversion ergeben

4.8.3. Anpassbarkeitstests

- Zeigt, ob eine bestimmte Anwendung in allen geplanten Zielumgebungen korrekt funktioniert
- Anpassungsfähiges System = offenes System, das sein Verhalten an Änderungen in der Umgebung anpassen kann
- Anpassbarkeit kann sich darauf beziehen, dass die SW durch Ausführung eines definierten Verfahrens auf verschiedene spezifizierte Umgebungen portieren lässt
- Anpassbarkeitstests können zusammen mit Installationstests durchgeführt werden. Normalerweise finden anschliessend funktionale Tests statt, um Fehlerzustände aufzudecken, die durch das Anpassen der SW an die andere Umgebung entstanden sein können

4.8.4. Austauschbarkeitstests

- Sind darauf fokussiert, die Austauschbarkeit von SW-Komponenten eines Systems gegen andere Komponenten zu testen
- Besonders relevant bei Systemen, die kommerzielle Standardsoftware für bestimmte Systemkomponenten verwenden
- Austauschbarkeitstests können parallel zu funktionalen Integrationstests durchgeführt werden, wenn alternative Komponenten für die Integration in das Gesamtsystem verfügbar sind
- Austauschbarkeit lässt sich in technischen Reviews oder Inspektionen der Systemarchitektur oder des Systementwurfs bewerten, bei denen Wert auf klare Definition der Schnittstellen zu möglichen Austauschkomponenten gelegt wird

5. Reviews

5.1. Einführung

- Reviews leisten den grössten einzelnen sowie den kosteneffektivsten Beitrag zur gelieferten Qualität
- TTA nehmen normalerweise an technischen Reviews und Inspektionen teil → Inputs zum Systemverhalten
- TTA spielen wichtige Rolle bei Definition, Anwendung und Pflege von Review-Checklisten und bei Bereitstellung von Informationen über den Schweregrad von Fehlerzuständen
- Zum guten Review gehört: Inhalte verstehen | zu bestimmen was fehlt | zu verifizieren, dass das beschriebene Produkt mit anderen, bereits entwickelten / in Entwicklung befindlichen Produkten konsistent ist
- Review soll sich am Review-Gegenstand orientieren, nicht am Autor

- Aufgabe des TTA beim Review → Sicherstellen, dass die im Arbeitsergebnis gelieferten Infos ausreichend sind, um das Testen zu unterstützen
- Wenn Info nicht vorhanden / nicht klar und eindeutig / nicht detailliert genug → Fehler

5.2. Checklisten in Reviews verwenden

- Nützlichsten Checklisten = die welche über einen Zeitraum von einem einzelnen Unternehmen entwickelt wurden, da diese folgendes wiedergeben:
  - Art des Produkts
  - Örtliches Entwicklungsumfeld
    - Personal | Werkzeuge | Prioritäten
  - Historie früherer Erfolge und Fehlerzustände
  - Spezifische Themen (z.B. Performanz, Sicherheit)
- Manche Unternehmen erweitern SW-Checklisten um sogenannte Anti-Patterns = Antimuster, die sich auf häufige Fehler, schlechte Verfahren, ineffektive Praktiken beziehen

5.2.1. Architekturreviews

- Inhaltspunkte für Checkliste für Architekturreviews:
  - Verbindungspooling → Den für die Ausführung benötigten Zeitaufwand, der mit dem Aufbau von Datenbankverbindungen zusammenhängt, reduzieren - durch einen gemeinsamen Pool von Verbindungen
  - Lastverteilung → Gleichmässige Verteilung der Last zwischen einer Menge von Ressourcen
  - Verteilte Verarbeitung
  - Caching → Lokale Kopie von Daten verwenden, um Zugriffszeiten zu reduzieren
  - Verzögerte Instantiierung (lazy instantiation)
  - Parallelität von Transaktionen
  - Prozesstrennung zwischen OLTP (Online Transactional Processing) und OLAP (Online Analytical Processing)
  - Replikation von Daten

5.2.2. Code-Reviews

- 6 Bereiche, die von Code-Review-Checklisten abgedeckt werden können:
  - 1. Struktur
    - Design im Code vollständig?
    - Entspricht Code den einschlägigen Programmierkonventionen?
    - Code gut strukturiert, konsistent im Stil und einheitlich formatiert?
    - Gibt es Prozeduren, die nicht aufgerufen / nicht benötigt werden, gibt es unerreichbaren Code?
    - Gibt es im Code Überbleibsel vom Testen (Platzhalter, Testroutinen)?
    - Kann Code durch Aufrufe externer Komponenten / Bibliotheksfunktionen ersetzt werden?
    - Gibt es Codeblöcke die sich wiederholen, die in einer Prozedur zusammengefasst werden können?
    - Ist die Speichernutzung effizient?

        - Wird Symbolik benutzt anstatt «magische» Konstanten / String-Konstanten?
        - Gibt es übermässig komplexe Module die umstrukturiert / aufgeteilt werden sollten?
    - 2. Dokumentation
        - Ist Code eindeutig, ausreichend dokumentiert & in wartbaren Stil kommentiert?
        - Passen alle Kommentare zum Code?
        - Entspricht die Dokumentation den geltenden Standards?
    - 3. Variablen
        - Sind alle Variablen richtig mit sinnvollen, konsistenten & eindeutigen Namen definiert?
        - Gibt es redundante oder ungenutzte Variablen?
    - 4. Arithmetische Operationen
        - Wird im Code vermieden, dass Gleitkommazahlen auf Gleichheit geprüft werden?
        - Verhindert der Code systematisch Rundungsfehler?
        - Vermeidet der Code Additionen / Subtraktionen mit sehr unterschiedlich grossen Zahlen?
        - Werden Teiler (Divisoren) auf Null / Rauschen getestet?
    - 5. Schleifen und Zweige
        - Sind alle Schleifen, Verzweigungen & Logikkonstrukte vollständig, korrekt und richtig verschachtelt?
        - Werden in IF-ELSEIF Ketten die häufigsten Fälle zuerst getestet?
        - Werden alle Fälle in einem IF-ELSEIF oder CASE Block behandelt, inkl. ELSE- / DEFAULT-Klauseln?
        - Gibt es für jede Case-Anweisung einen Standardwert?
        - Sind die Abbruchbedingungen von Schleifen offensichtlich und immer erreichbar?
        - Sind die Index-Variablen oder Teilskripte unmittelbar vor der Schleife richtig initialisiert?
        - Können Anweisungen innerhalb der Schleife auch ausserhalb platziert werden?
        - Wird vermieden, dass der Code in der Schleife die Index-Variable manipuliert / diese nach Beendigung der Schleife verwendet?
    - 6. Defensive Programmierung
        - Werden Indizes, Zeiger, Teilskripte mit Bezug auf Arrays, Datensätze / Dateigrenzen getestet?
        - Wird getestet, ob importierte Daten und Eingabeparameter gültig und vollständig sind?
        - Sind alle Ausgangsvariablen zugewiesen?
        - Wird in jeder Anweisung das richtige Datenelement verarbeitet?
        - Wird jeder zugewiesene Speicher freigegeben?
        - Wird Code zur Behandlung von Zeitüberläufen / Fehlerbedingungen für Zugriffe auf externe Geräte verwendet?
        - Wird vor einem Zugriff auf Dateien geprüft, ob diese existieren?
        - Sind alle Dateien & Geräte in einem korrekten Zustand, wenn das Programm beendet wird?

6. Testwerkzeuge und Automatisierung

6.1. Integration und Informationsaustausch zwischen Werkzeugen

- Ein ideales Toolset verhindert, dass Informationen über die einzelnen Werkzeuge hinweg dupliziert werden

6.2. Ein Testautomatisierungsprojekt definieren

- Aktivitäten des TTA im Zusammenhang mit Testautomatisierung:
  - Bestimmen, wer für die Testausführung verantwortlich sein wird
  - Das am besten geeignete Werkzeug für das Unternehmen, den Zeitplan, die Fähigkeiten im Team, die Wartungsanforderungen usw. auswählen
    → kann auch bedeuten ein eigenes Werkzeug selber zu entwickeln anstatt eins zu beschaffen
  - Die Anforderungen für die Schnittstellen zwischen dem Automatisierungswerkzeug und anderen Werkzeugen (Anforderungs-, Test-, Fehlermanagementwerkzeug) definieren
  - Vorgehensweise für Automatisierung auswählen: Schlüsselwort- oder Datengetriebene Vorgehensweise?
  - Mit TM zusammen die Kosten für Implementierung (inkl. Schulung) schätzen
  - Zeitplanung für das Automatisierungsprojekt erstellen & Zeit für die Wartung einplanen
  - TA und Business Analysten schulen, wie die Daten für Automatisierung zu verwenden & zu liefern sind
  - Bestimmen, wie die automatisierten Tests ausgeführt werden
  - Bestimmen, wie die Testergebnisse der autom. Tests mit den manuellen Tests kombiniert werden

6.2.1. Die Vorgehensweise für die Automatisierung auswählen

- Eine der ersten Entscheidungen des TTA ist, welche die effektivste Schnittstelle für die Automatisierung ist
  → GUI oder auf API-Ebene durch CLI (Command Line Interface)? Sonstige Schnittstelle?
- GUI kann sich öfters ändern → höherer Wartungsaufwand
- Datengetriebene Vorgehensweise: Trennung der autom. Testschritte von den Daten (In/Out)
  Z.B.: Daten für Eingabefeld → Daten für Positivtest != Daten für Negativtest, aber Testschritte die gleichen
  TA: Datenerstellung mit Wissen über Geschäftslogik & Eigentümer der Skripte & Verantwortlich für dessen Ausführung
  TTA: Entwicklung der Testschritte in den Skripten
- Schlüsselwort- / Aktionswortgetriebene Vorgehensweise: Trennung der Aktion, die mit den gelieferten Daten durchgeführt werden soll, vom Testskript
  TAs erstellen abstrakte Metasprache, die eher die Aktionen beschreibt als direkt ausführbar zu sein

Z.B.: «Anmelden», «Benutzer_anlegen», «Benutzer_löschen»
TTA erstellt die Skriptmodule dazu
TA stellt die Schritte aus den Metasprachaktionen zu Testfällen zusammen
→ Durch diese Trennung steigt die Wartbarkeit des Codes & Rentabilität für Automatisierung

- Zustand des Systems zu Beginn und am Ende der Tests muss berücksichtigt werden!

6.2.2. Geschäftsprozesse für die Automatisierung modellieren

- Folgende Punkte sind dabei zu berücksichtigen:
  - Je höher die Granularität der Schlüsselwörter, desto spezifischer sind die Szenarien, die abgedeckt werden können. Die Wartung der Aktionsreihenfolge wird jedoch durch die abstrakte Sprache komplexer
  - Wenn TA auch die konkreten Aktionen spezifizieren («Schaltfläche betätigen», etc.), können die schlüsselwortgetriebenen Tests besser mit verschiedenen Situationen umgehen. Da jedoch GUI getrieben, steigt der Wartungsaufwand der Aktionen selber (Ersatz von Aktionen, etc.)
  - Die Verwendung von Sammelbegriffen als Schlüsselwörter kann die Entwicklung einfacher, die Wartung aber komplizierter machen. Es kann z.B. sechs verschiedene Schlüsselwörter geben für «Datensatz erstellen». Sollte ev. ein Schlüsselwort erstellt werden, das alle sechs Schlüsselwörter der Reihe nach aufruft, um die Aktion zu vereinfachen?
  - Es wird immer wieder dazu kommen, dass neue oder andere Schlüsselwörter benötigt werden. Schlüsselwörter haben 2 Aspekte: Geschäftslogik & Automatisierungsfunktionalität
    → Es muss Prozess gefunden werden der beide Aspekte berücksichtigt
- Schlüsselwortgetriebene Testautomatisierung kann die Wartungskosten verringern, ist aber kostspieliger und schwieriger in der Entwicklung

6.3. Spezifische Testwerkzeuge

6.3.1. Werkzeuge zur Fehlereinpflanzung und zum Einfügen von Fehler

- Fehlereinpflanzungswerkzeuge → v.a. auf Code-Ebene. Um systematisch einzelne oder bestimmte Arten von Fehlerzuständen zu generieren
- Diese Werkzeuge fügen absichtlich Fehlerzustände in das Testobjekt ein, um so eine Bewertung der Qualität der Testsuiten zu ermöglichen (z.B. deren Eigenschaft, Fehler zu finden)
- Werkzeuge zum Fehlereinfügen werden eingesetzt, um die Fehlerbehandlungsmechanismen des Testobjekts bei anormalen Betriebsbedingungen zu testen
- Generierung absichtlich falscher Eingabewerte, um zu testen, dass die SW damit umgehen kann
- Wird v.a. vom TTA verwendet, jedoch auch von Entwicklern, um neu entwickelten Code zu testen

6.3.2. Performanztestwerkzeuge

- 2 Hauptfunktionen: Lastgenerierung & Messung und Analyse des Systemverhaltens unter bestimmter Last
- Hauptantwort: «Bei 500 gleichzeitigen Usern und 50% CPU Auslastung ist die durchschnittliche Antwortzeit von Aktion X 1.8s»
- Typische Metriken und Berichte dieser Testwerkzeuge:
    - Anzahl simulierter Anwender im Testverlauf
    - Anzahl & Art der von den simulierten Anwendern erzeugten Transaktionen & deren Eingangsrate
    - Antwortzeiten für bestimmte, von den Anwendern angeforderte Transaktionen
    - Berichte und Graphen, die Systemlast und Antwortzeiten gegenüberstellen
    - Berichte über Ressourcennutzung (z.B. Auslastung im Zeitverlauf mit Mindest- und Maximalwerten)
- Wichtige Faktoren beim Implementieren von Performanztestwerkzeugen sind:
    - Die für die Lastgenerierung benötigte HW und Netzwerkbandbreiten
    - Kompatibilität des Werkzeugs mit dem Kommunikationsprotokoll des zu testenden Systems
    - Ausreichende Flexibilität des Werkzeugs für einfache Implementierung unterschiedlicher Nutzungsprofile
    - Benötigte Überwachungs-, Analyse- und Berichtsfunktionen

6.3.3. Werkzeuge für den webbasierten Test

- Verwendungszwecke einiger gebräuchlicher webbasierter Testwerkzeuge:
    - Hyperlink-Werkzeuge → Scanning von Webseiten, ob keine Hyperlinks ungültig sind oder fehlen
    - HTML- & XML-Werkzeuge → Überprüfen ob die HTML- & XML-Standards in den Seiten eingehalten werden
    - Lastsimulatoren testen das Serververhalten, wenn viele Anwender eine Serververbindung herstellen
    - Einfach Testausführungswerkzeuge, die mit unterschiedlichen Browsern funktionieren
    - Werkzeuge zum Scannen des Servers, um nicht verlinkte Dateien zu identifizieren
    - HTML-Prüfprogramme
    - Cascading Style Sheet (CSS)-Prüfprogramme
    - Werkzeuge zur Prüfung von Standardverletzungen
    - Werkzeuge, die eine Reihe von Sicherheitsproblemen identifizieren
- TA & TTA verwenden diese Werkzeuge vorwiegend beim Systemtest

6.3.4. Werkzeugunterstützung für modellbasiertes Testen

- Modellbasiertes Testen (MBT) → Testverfahren, bei dem ein formales Modell (z.B. endliche Zustandsmaschine) eingesetzt wird, um das erwartete Verhalten eines SW-gesteuerten Systems während der Ausführung zu beschreiben
- Auch andere Modelle wie z.B. Petrinetze und State Charts unterstützen MBT

- MBT-Modelle und -Werkzeuge können eingesetzt werden um grosse Mengen von unterschiedlichen Ausführungssequenzen zu generieren
- MBT-Werkzeuge → Reduktion der grossen Anzahl möglicher Pfade, die in einem Modell generiert werden können
- Damit lässt sich neue Sicht auf SW werfen → Aufdecken von bisher übersehenen Fehlerzuständen

6.3.5. Komponententest- und Build-Werkzeuge

- Komponententest- und Build-Automatisierungswerkzeuge → oft von Entwicklern verwendet, von TTA v.a. in agilen Entwicklungen eingesetzt und gewartet
- Häufig auf Programmiersprache zugeschnitten → Java = JUnit für automatisierte Modultests
- Andere Sprachen → Zusammengefasst in Oberbegriff «xUnit-Test-Framework» → generieren Testobjekte für jede erzeugte Klasse
- Debugging-Werkzeuge → erleichtern den manuellen Komponententest auf tiefster Ebene, erlauben dem TTA Programmvariablen während der Ausführung zu ändern und die SW Zeile für Zeile auszuführen
- Build-Automatisierungswerkzeuge → erlauben es nach jeder Änderung einer SW-Komponente einen Build zu erstellen. Danach führen andere Werkzeuge automatisch Komponententests durch
  → normalerweise Bestandteil kontinuierlicher Integrationsumgebungen