

Inhaltsverzeichnis

1. Softwaretests.....	1
1.1. Einleitung.....	1
1.2. Grundlagen.....	2
1.2.1. Vom Unit-Test bis zum Abnahme-Test.....	4
1.2.2. Black-Box-Test (Funktionsorientiert) u. White-Box-Test (Strukturorientiert).....	5
1.2.3. Test ist nicht gleich Test.....	5
1.3. Systematischer-Test: Planung, Durchführung, Auswertung.....	6
1.3.1. Test-Planung: wer - was - wann - wie - wie lange.....	6
1.3.2. Test-Durchführung.....	6
1.3.3. Test-Auswertung.....	7
1.4. Black-Box-Test im Detail.....	7
1.4.1. Ziele.....	7
1.4.2. Auswahl der Testfälle.....	7
1.4.3. + Beispiel zum Black-Box-Test.....	8
1.5. White-Box-Test im Detail.....	9
1.5.1. Güte eines White-Box-Tests.....	11
1.6. Dokumentation: Vorschrift->Protokoll->Zusammenfassung.....	12
1.6.1. Aufbau einer Testvorschrift.....	12
1.6.2. Testzusammenfassung.....	13
1.7. Programmierte Testfälle (JUnit).....	15
1.8. Fragen.....	15

1. Softwaretests

Literatur:

<http://de.wikipedia.org/wiki/Softwaretest>

Martin Glinz: Software Engineering

IEEE Standard for Software Test Documentation (IEEE Std. 829-1998)

IEEE Standard for Software and System Test Documentation (IEEE Std. 829-2008)

1.1. Einleitung

☒ Validation:

Ist es die richtige/benötigte Software?

(SOLL/IST-Vergleich)

☒ Test:

Arbeitet die Software richtig?

☒ Debugging:

Warum arbeitet die Software nicht richtig?

☒ Verifikation:

Die Software ist richtig/korrekt !!!!!!!

(wird durch Tests festgestellt.)

- ☒ Das Problem der Individualverantwortung des Programmierers.

„... Seit der letzten Produktabnahme herrscht in der Softwareabteilung einer Firma äußerst schlechte Stimmung. Man spricht von Schuldzuweisungen. Zur Problemlösung bittet man Sie, ein Konzept zum Testen von Software vorzustellen....“
- ☒ Problem: Individualverantwortlichkeit (zB. Der Programmierer)
 - ☐ oftmals Schuldzuweisungen
 - ☐ oft Produktivität statt Qualität gefordert
 - ☐ mangelnde Fehleranalysen
- ☒ **Lösung: Teamverantwortung(Kooperation,Qualitätsbewußtsein,QS-Abteilung)**
 - ☐ **Trennung** von Produktverantwortung und Qualitätsverantwortung
 - ☐ Moderation durch unabhängige Berater!
 - ☐ **Qualitätsrunden**
 - Peer Review
 - aktive Beteiligung aller Mitwirkenden
 - Gleichberechtigung und Konsensbildung
 - ☐ Professioneller Einsatz von **Testwerkzeugen**
 - ☐ Einsatz von Entwicklungsmethoden, die das Testen stärker in der Vordergrund stellen.

1.2. Grundlagen

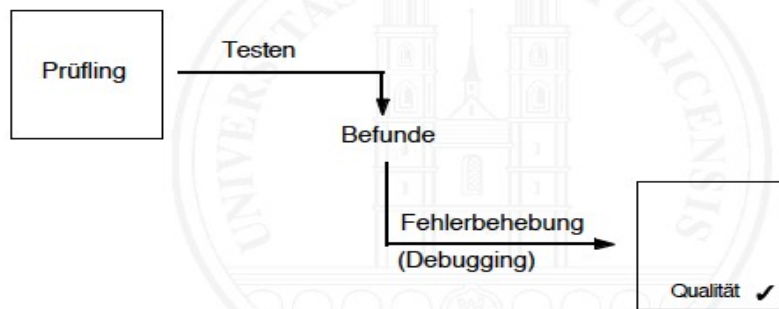
- ☒ Ziel des Softwaretests ist es, professionell **Fehler aufzudecken**.
- ☒ Unvorbereitete und undokumentierte Tests sind **sinnlos**.
- ☒ **Nachzuweisen**, dass keine **Fehler** vorhanden sind, **ist nicht Ziel** des Softwaretests.
- ☒ Werden bei einem Test keine Fehler gefunden, so beweist dies niemals, dass das Programm korrekt ist.

<http://de.wikipedia.org/wiki/Programmfehler>:
„Statistische Erhebungen in der [Softwaretechnik](#) weisen im Mittel etwa 2 bis 3 Fehler je 1000 Zeilen Code aus.“
- ☒ Die **Korrektheit** eines Programms kann **durch Testen nicht bewiesen** werden.

Der Grund ist, dass alle Kombinationen aller möglichen Werte der Eingabedaten getestet werden müssten. Die Anzahl der Kombinationen ist bei realen Programmen zu groß, um alle zu testen.

Aus diesem Grund beschäftigen sich verschiedene Teststrategien/-konzepte mit der Frage, wie mit einer **möglichst geringen Anzahl von Testfällen eine große Testabdeckung** zu erreichen ist.

- ☑ Der Test findet meist **nur eine Auswirkung** des Fehlers und nicht seine eigentliche Ursache.



- ☑ Die Ergebnisse des Softwaretests tragen zur Beurteilung der realen **Qualität der Software** bei.
- ☑ **Testen** setzt voraus, dass die erwarteten Ergebnisse bekannt sind
 - ☐ Entweder muss **gegen eine Spezifikation**
 - ☐ oder **gegen vorhandene Testergebnisse** (z.B. bei der Wiederholung von Tests nach Programm-Modifikationen) getestet werden (so genannter **Regressionstest**)
- ☑ **Welche Funktionen** getestet werden, hängt von dem **Testanspruch** ab.

Bei einem Softwaresystem mit sensiblen Daten wird man z. B. eine höhere Priorität auf die Datensicherheit legen als bei einem Computerspiel, bei dem man eine höhere Priorität z. B. auf die Benutzbarkeit legen wird.
- ☑ Die Testplanung findet **parallel zur Softwareentwicklung** statt.

Siehe: [V-Modell](#)

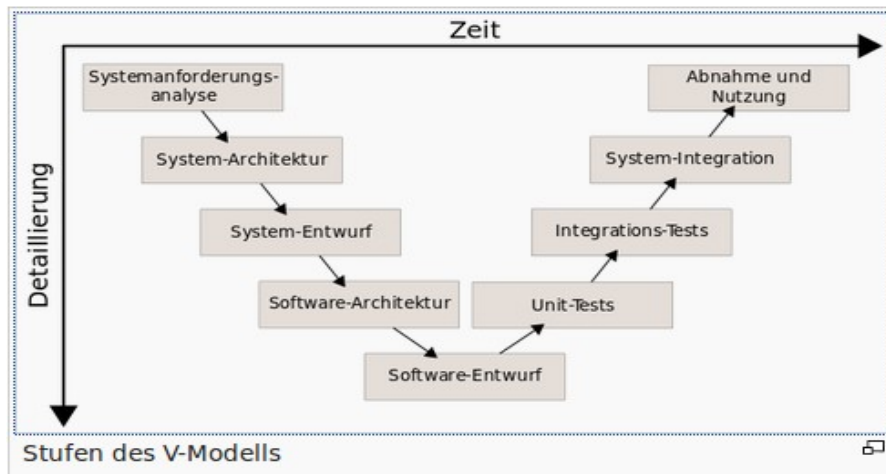


Abbildung 1: <https://de.wikipedia.org/wiki/Softwaretest>

1.2.1. Vom Unit-Test bis zum Abnahme-Test

Testgegenstand sind Komponenten, Teilsysteme oder Systeme

☑ **Unit-Test**, Modultest, Komponenten-Test

ist ein Test auf der tiefsten Ebene. Dabei werden einzelne Komponenten auf korrekte Funktionalität getestet. Häufig wird der Modultest durch den **Software-Entwickler** durchgeführt. Die Testobjekte sind einzelne abgegrenzte Module, also Unterprogramme, Units oder Klassen.

☑ **Integrations-Test**



testet die Zusammenarbeit voneinander abhängiger Komponenten.

☑ **System-Test** (System Integrations - Test)

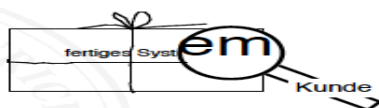


SOLL-IST Vergleich des gesamten Systems (funktionale und nicht funktionale Anforderungen). Gewöhnlich findet der Test auf einer **Testumgebung** statt und wird mit **Testdaten** durchgeführt. Die Testumgebung soll die Produktivumgebung des Kunden simulieren.

☑ **Abnahme-Test** (acceptance test): eine besondere Form des Tests:

☐ nicht: Fehler finden

☐ sondern: zeigen, dass das System die gestellten Anforderungen (s. Pflichtenheft) erfüllt,



d.h. In allen getesteten Fällen fehlerfrei arbeitet.

1.2.2. Black-Box-Test (Funktionsorientiert) u. White-Box-Test (Strukturorientiert)

- ☑ **Funktionsorientierter Test (Black-Box-Test)**
 - ☐ **Testfall-Auswahl aufgrund der Spezifikationen im Pflichtenheft**
 - ☐ Programmstruktur selbst kann unbekannt sein

werden von Programmierern und Testern entwickelt, die **KEINE** Kenntnisse über den inneren Aufbau des zu testenden Systems haben. In der Praxis werden Black-Box-Tests meist von speziellen **Test-Abteilungen** oder Test-Teams entwickelt.

- ☑ **Strukturorientierter Test (White-Box-Test)**
 - ☐ **Testfall-Auswahl aufgrund der Programmstruktur**
 - ☐ Spezifikation muss ebenfalls bekannt sein (wegen der erwarteten Resultate)

werden oft **von den gleichen Programmierern** entwickelt, die den Testling entwickelt haben.

1.2.3. Test ist nicht gleich Test

- ☑ **Laufversuch:** Der Entwickler „testet“
 - ☐ Entwickler übersetzt, bindet und startet sein Programm
 - ☐ Läuft das Programm nicht oder sind Ergebnisse offensichtlich falsch, werden die Defekte gesucht und behoben („Debugging“)
 - ☐ Der „Test“ ist beendet, wenn das Programm läuft und die Ergebnisse vernünftig aussehen
- ☑ **Wegwerf-Test:** Jemand testet, aber ohne System
 - ☐ Jemand führt das Programm aus und gibt dabei Daten vor
 - ☐ Werden Fehler erkannt, so werden die Defekte gesucht und behoben
 - ☐ Der Test endet, wenn der Tester findet, es sei genug getestet
- ☑ **Systematischer Test:**
 - ☐ **Spezialisten testen**
 - Es gibt
 - ☐ **TEST-PLANUNG:**
 - ☐ **TEST-DURCHFÜHRUNG:**
 - ☐ **TEST-ABNAHME/AUSWERTUNG:**

1.3. Systematischer-Test: Planung, Durchführung, Auswertung

1.3.1. Test-Planung: **wer - was - wann - wie - wie lange**

☒ **Vorbereitung**

- ☐ Einbettung des Testens in die **Entwicklungsplanung**:
 - ☐ Termine und Kosten für das Testen
 - ☐ welche Dokumente sind zu erstellen
 - ☐ V-Modell: **In jeder Phase der Systementwicklung werden auch die entsprechenden Tests geplant**
- ☐ Bereitstellen von Test-Spezialisten-**Team**
- ☐ Bereitstellen d. **Testumgebung**:
 - https://de.wikipedia.org/wiki/Kontinuierliche_Integration
 - ☐ Allg. Dokumentationsverwaltung
 - ☐ Programmverwaltung (Versionsmanagement: git, ...)
 - ☐ Testsysteme (Buld-Systeme, Coverage-Systeme, jenkins, ...)

☒ **Testvorschriften festlegen**

Die Testvorschrift wird zu Beginn des Projektes festgelegt, und sie beinhaltet folgende Elemente:

- ☒ **Test-Art (Unit-Test od. Integrations-test, System-Test, Abnahme-Test, ...)**
- ☒ **Test-Umgebung definieren**
- ☒ **Test-Daten festlegen**
- ☒ **Test-Abnahmekriterien**
- ☒ **einzelne Testfälle**

1.3.2. Test-Durchführung

- ☐ **Testumgebung nutzen**
- ☐ **Testfälle** nach Testvorschrift **ausführen**
- ☐ **Fehlersuche und -behebung erfolgen separat**
Fehlerbehebung (ist nicht Bestandteil des Tests!)
- ☐ Ergebnisse notieren
- ☐ Ist-Resultate werden mit Soll-Resultaten **verglichen**
- ☐ Prüfling während des Tests nicht verändern
- ☐ Nicht bestandene Tests werden **wiederholt**
- ☐ Testergebnisse werden **dokumentiert** (s. Testvorschrift)

1.3.3. Test-Auswertung

- ☐ **Testbefunde** / Test-Dokumentation erstellen
- ☐ gefundene Fehler(symptome) analysieren
- ☐ Fehlerursachen bestimmen lassen (Debugging)
- ☐ Fehler beheben
- ☐ **TESTENDE:**
 - Test **endet**, wenn vorher definierte **Test-Fälle fehlerfrei** sind

1.4. Black-Box-Test im Detail

1.4.1. Ziele

Ziele dieser Methode sind:

- ☒ **Funktionsüberdeckung:**
 - ☐ jede **spezifizierte Funktion (Pflichtenheft) mindestens einmal** aktiviert
- ☒ **Ausgabeüberdeckung:**
 - ☐ jede **spezifizierte Ausgabe mindestens einmal** erzeugt
- ☒ **Ausnahmeüberdeckung:**
 - ☐ jede **spezifizierte Ausnahme- bzw. Fehlersituation mindestens einmal** erzeugt

1.4.2. Auswahl der Testfälle

Techniken **um möglichst wenig, möglichst gute Testfälle** auszuwählen sind

- ☒ **Grenzwertanalyse**
 - ☐ Beispiel: Multiplikation von ganzen Zahlen
 - Mögliche Grenzfälle
 - Testfall 1:** x ist null
 - Testfall 2:** y ist null
 - Testfall 3:** x und y sind beide null
 - Testfall 4:** Produkt läuft positiv über
 - Testfall 5:** Produkt läuft negativ über
- ☒ **Äquivalenzklassenbildung**
 - ☐ **ähnliche Eingaben** werden zu sog. Äquivalenzklassen gebildet
 - ☐ pro Klasse wird ein Repräsentant Testkandidat
 - ☐ Beispiel: Multiplikation von ganzen Zahlen
 - Mögliche Äquivalenzklassen
 - Testfall 1:** x und y positiv
 - Testfall 2:** x positiv, y negativ
 - Testfall 3:** x negativ, y positiv
 - Testfall 4:** x und y negativ

1.4.3. +Beispiel zum Black-Box-Test

☑ 1. Spezifikation erstellen

Gegeben sei ein Programm, das folgende Spezifikation erfüllen soll:

- ☐ Das Programm fordert zur Eingabe von drei nicht negativen reellen Zahlen auf und liest die eingegebenen Werte.
- ☐ Das Programm interpretiert die eingegebenen Zahlen als Strecken a , b und c .
- ☐ Es untersucht, ob die drei Strecken ein Dreieck bilden und klassifiziert gültige Dreiecke.

Das Programm liefert folgende Ausgaben:

- ☐ kein Dreieck wenn $a+b \leq c$ oder $a+c \leq b$ oder $b+c \leq a$
 - ☐ gleichseitiges Dreieck, wenn $a=b=c$
 - ☐ gleichschenkliges Dreieck, wenn $a=b$ oder $b=c$ oder $a=c$
 - ☐ unregelmäßiges Dreieck sonst
- ☑ Das Programm zeichnet ferner alle gültigen Dreiecke winkeltreu und auf maximal darstellbare Größe skaliert in einem Fenster der Größe 10x14 cm.
Die Seite c liegt unten parallel zur Horizontalen. Alle Eckpunkte haben einen Minimalabstand von 0,5 cm vom Fensterrand.
- ☑ Das Programm liefert eine Fehlermeldung, wenn andere Daten als drei nicht negative reelle Zahlen eingegeben werden. Anschließend wird mit einer neuen Eingabeaufforderung versucht, gültige Werte einzulesen.

☑ 2. Testüberdeckungskriterien festlegen

- ☐ Aktivierung aller **Funktionen**
Pruefen() und Klassifizieren()
Skalieren() und Zeichnen()
- ☐ Erzeugen aller **Ausgaben**
kein Dreieck
gleichseitiges Dreieck
gleichschenkliges Dreieck
unregelmäßiges Dreieck
- ☐ Erzeugung aller **Ausnahmesituationen**
ungültige Eingabe

☑ 3a. Testfälle festlegen (Beispiel Äquivalenzklassen)

Klasse, Subklasse	Repräsentant		
kein Dreieck		unregelmäßiges Dreieck	
a größte Seite	4.25, 2, 1.3	α spitz, β spitz	3, 5, 6
b größte Seite	1.3, 4.25, 2	α spitz β rechtwinklig	3, 5, 4
c größte Seite	2, 1.3, 4.25	α spitz β stumpf	3, 6, 4
gleichseitiges Dreieck	4.2, 4.2, 4.2	β spitz, γ spitz	6, 3, 5
gleichschenkliges Dreieck		β spitz γ rechtwinklig	4, 3, 5
a=b	4.71, 4.71, 2	β spitz γ stumpf	4, 3, 6
b=c	3, 5.6, 5.6	γ spitz, α spitz	5, 6, 3
a=c	11, 6, 11	γ spitz α rechtwinklig	5, 4, 3
		γ spitz α stumpf	6, 4, 3

☑ 3b. Testfälle festlegen (Beispiel Grenzwertanalyse)

Grenzfall	Testwerte
kein Dreieck	
$a=b=c=0$	0, 0, 0
$a=b+c$	6, 2, 4
$b=a+c$	2, 6, 4
$c=a+b$	2, 4, 6
sehr flaches Dreieck	
$c=a+b - \epsilon$, ϵ sehr klein	3, 4, 6.999999999999
$b=a+c - \epsilon$, ϵ sehr klein	3, 6.999999999999, 4
sehr steiles Dreieck	
c klein, $a=b$ sehr groß	10^7 , 10^7 , 5

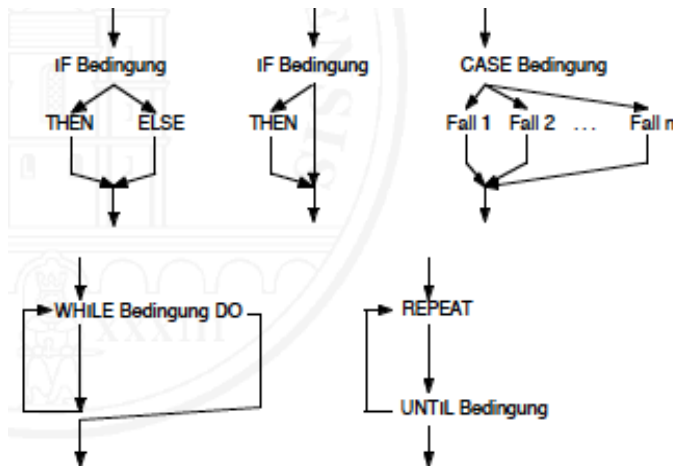
1.5. White-Box-Test im Detail

- ☑ Im Gegensatz zum Black-Box-Test ist für diesen Test ein Blick in den **Quellcode** gestattet, d.h. es wird am Code geprüft.
- ☑ Testfälle werden so gewählt, dass das Programm systematisch durchlaufen wird
- ☑ Spezielle Test-Abteilungen werden für White Box Tests in der Regel nicht eingesetzt, da der Nutzen speziell für diese Aufgabe abgestellter Tester meist durch den Aufwand der Einarbeitung in das System eliminiert wird.
- ☑ Überdeckungen
 - ☐ **Anweisungsüberdeckung:**
Jede Anweisung des Programms wird mindestens einmal ausgeführt. Auch If, while, ... werden als eine Anweisung angesehen. D.h. Bei if-else wird nur ein Teil getestet. Bei while vielleicht keiner.
 - ☐ **Zweigüberdeckung:**
jeder Programmzweig wird mindestens einmal durchlaufen. D.h. Man braucht mehrere Testfälle, sodass zB. Einmal der if-Teil und ein andermal der else-Teil getestet wird.
 - ☐ **Pfadüberdeckung:**
jede mögliche Kombination von Programmzweigen wird mindestens einmal durchlaufen. Hier braucht man die meisten Testfälle.

☑ Pfade:

IF und SCHLEIFEN haben 2 Pfade

SWITCH/CASE haben mehrere Pfade



☑ Während die Anweisungsabdeckung einfach zu bewerkstelligen ist, bereitet die Pfadabdeckung Probleme, da die Anzahl der Pfade exponentiell mit der Anzahl der Verzweigungen im Programm wächst. D.h. Man braucht sehr viele Testfälle.

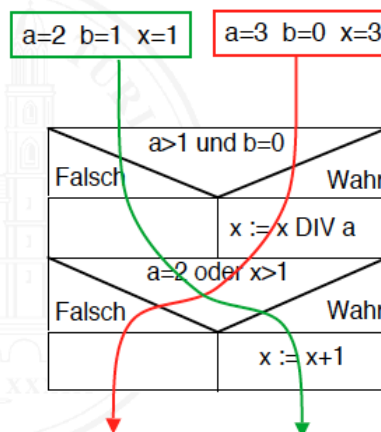
☑ +Beispiel

(nach Myers, 1979)

```

...
VAR
  a,b,x: INTEGER;
...
BEGIN
...
  IF (a>1) AND (b=0)
    THEN x := x DIV a;
  IF (a=2) OR (x>1)
    THEN x := x+1;
...

```



Beispiel: Notwendige Testfälle

☑ Anweisungsüberdeckung z.B. mit dem Testfall:

a=2 b=0 x=1

☑ Zweigüberdeckung mit den Testfällen:

a=3 b=0 x=3

a=2 b=1 x=1

☑ Pfadüberdeckung mit den Testfällen:

a=1 b=1 x=2

a=3 b=0 x=3

a=2 b=0 x=4

a=1 b=1 x=1

1.5.1. Güte eines White-Box-Tests

Die Testgüte hängt von gewählter Überdeckung und erreichtem **Überdeckungsgrad** ab

- ☑ **Überdeckungsgrad**
Prozentuales Verhältnis der Anzahl überdeckter Elemente zur Anzahl vorhandener Elemente
- ☑ Beispiel: Der Testfall **a=3 b=0 x=3** erreicht 50% Zweigüberdeckung
- ☑ **Anweisungsüberdeckung** ist ein **schwaches** Kriterium.
Fehlende Anweisungen werden beispielsweise nicht entdeckt
- ☑ **Zweigüberdeckung** wird in der Praxis angestrebt.
Dennoch: falsch formulierte Bedingungsterme (z.B. $x > 1$ statt $x < 1$) werden nicht entdeckt
- ☑ **Pfadüberdeckung** ist in fast allen Programmen, die **Schleifen mit Verzweigungen** enthalten, **nicht testbar**

1.6. Dokumentation: Vorschrift->Protokoll->Zusammenfassung

Ein ingenieurmäßiges Vorgehen beim Testen von Software, bedeutet auch, dass die Testdokumentation eine wichtige Aufgabe darstellt.

1. Das wichtigste Dokument für Testvorbereitung und -durchführung ist die **Testvorschrift**.
2. **Testprotokolle** notieren zu jedem Testfall das Testergebnis.
3. Eine **Testzusammenfassung** bildet den Nachweis über die Durchführung und das Gesamtergebnis eines Tests

Es gibt Normen mit sehr umfangreichen Vorschriften für Testplanung und -dokumentation (IEEE 1987, 1988, 1998a, 1998b)
für den Test kritischer Software sollten diese verwendet werden
für gewöhnliche Software genügen die hier genannten Dokumente

1.6.1. Aufbau einer Testvorschrift

Die Testvorschrift wird zu Beginn des Projektes festgelegt, und sie beinhaltet folgende Elemente:

- 1. Teststart (Unittest od. Komponententest, ...)**
- 2. Testumgebung, Testdaten**
- 3. Abnahmekriterien**

4. einzelne Testfälle

Hier im Detail:

1. Testart

1.1 Art und Zweck des Tests

Art und Zweck des im Dokument beschriebenen Tests (Unit-Test, Integrationstest, ...)

1.2 Testumfang

Welche Einheiten (Unit, Subsystem, ...) des Systems getestet werden

1.3 Referenzierte Unterlagen

Verzeichnis aller Unterlagen, auf die im Dokument Bezug genommen wird

2. Testumgebung und Testdaten

2.1 Überblick

Testgüte, Annahmen und Hinweise

2.2 Testwerkzeuge

Test-Software und -Hardware, Betriebssystem, Werkzeuge

2.3 Testdaten, Testdatenbank

Wo die für den Test benötigten Daten bereit liegen oder bereitzustellen sind

2.4 Personalbedarf

wieviele Personen zur Testdurchführung benötigt werden

3. Annahmekriterien

Kriterien für

erfolgreichen Test-Abschluss

Test-Abbruch

Unterbrechung und Wiederaufnahme des Tests

4. Testfälle

Testabschnitt 1: Korrekte Eingaben

Zweck:

- testet die Klassifikationsfunktion
- testet bei echten Dreiecken die Zeichne-Funktion

Vorbereitungsarbeiten: keine

Aufräumarbeiten: keine

Hinweis: alle Zahlen sind als Dezimalzahlen einzugeben




Testsequenz 1-1: Kein Dreieck

TestfallNr.	Eingabe	erwartetes Resultat	Befund
1-1-1	4.25, 2, 1.3	kein Dreieck	
1-1-2	1.3, 4.25, 2	kein Dreieck	
1-1-3	2, 1.3, 4.25	kein Dreieck	

Testsequenz 1-2: regelmäßiges Dreieck

TestfallNr.	Eingabe	erwartetes Resultat	Befund
1-2-1	4.2, 4.2, 4.2	gleichseitig 	
1-2-2	4.71, 4.71, 2	gleichschenkelig 	
1-2-3	3, 5.6, 5.6	gleichschenkelig 	
1-2-4	11, 6, 11	gleichschenkelig 	

Testsequenz 1-3: unregelmäßiges Dreieck

TestfallNr.	Eingabe	erwartetes Resultat	Befund
1-3-1	3, 5, 6	unregelmäßig 	
1-3-2	3, 5, 4	unregelmäßig 	
1-3-3	3, 6, 4	unregelmäßig 	
...	

1.6.2. Testzusammenfassung

Dokumentiert

☒ **Wer** hat getestet

☒ **Was:** Testgegenstand

☒ **Wie:** Verwendete Testvorschrift

☒ **Gesamtbefund**

☒ Wichtig für die Archivierung von Testergebnissen

TESTZUSAMMENFASSUNG			
Test Nr.:		Arbeitspaket Nr.:	
Testbeginn (Datum und Zeit):		Test Dauer:	
Testende (Datum und Zeit):			
GEGENSTAND UND ZWECK DES TESTS			
Projekt/Produkt:		Release Nr.:	
Geliefert von:			
<input type="radio"/> Einzeltest	<input type="radio"/> Integrationstest	<input type="radio"/> Systemtest	<input type="radio"/> Abnahmetest
TESTVORSCHRIFT			
Nummer/Version	Titel		
EMPFEHLUNG			
<input type="radio"/> akzeptieren (keine Wiederholung des Tests)		<input type="radio"/> wie es ist	
<input type="radio"/> nicht akzeptieren (Wiederholung des Tests)		<input type="radio"/> kleine Fehler	
<input type="radio"/> Test nicht beendet		<input type="radio"/> einige grössere Fehler	
		<input type="radio"/> einige fatale Fehler	
ZUSAMMENFASSUNG			
BEILAGEN			
<input type="radio"/> Liste der getesteten Software-Einheiten			
<input type="radio"/> Liste der Problemmeldungen			
<input type="radio"/> andere:			
TESTTEAM			
Name	(Leiter)	Datum	Visum

1.7. Programmierte Testfälle (JUnit)

An Stelle textuell beschriebener Testfälle können auch **programmierte Testfälle** verwendet werden:

- ☒ Jeder Testfall ist ein Objekt
- ☒ Enthält Testdaten und erwartetes Resultat
- ☒ Ruft den Testling auf
- ☒ Vergleicht erwartetes und geliefertes Resultat
- ☒ Eingebettet in ein passendes Laufzeitsystem sind teilautomatisierte Tests möglich
- ☒ Geeignet vor allem für Komponenten- und Integrationstest
- ☒ Nützlich als kontinuierlicher Regressionstest bei inkrementeller Entwicklung
- ☒ In Java-Umgebungen: JUnit

1.8. Fragen

- ☒ Güte des White-Box-Tests?
 - ☐ Überdeckungsgrad
 - ☐ Zweig- und Pfadabdeckung
 - ☐ 2 if-else-Zweige ergeben 4 Pfade
 - ☐ Pfadabdeckung bei Schleifen mit if-Verzweigungen kaum möglich.
- ☒ Debuggen
 - ☐ White-Box-Test oder Black-Box-Test?
- ☒ Regressionstest
 - ☐ Test gegen die Spezifikation oder gegen vorherige Testergebnisse
- ☒ Unterschied Systemtest zu Abnahmetest
 - ☐ Abnahmetest zeigt das Fehlen von Fehlern
 - ☐ Systemtest läuft noch auf der Testumgebung
- ☒ Finde Testfälle für: 2 Zahlen multiplizieren
 - ☐ Äquivalenzklassen
 - a, b pos
 - a, b neg
 - a neg, b pos
 - ...
 - ☐ Grenzwertanalyse
 - a, b null
 - a * b liefert Überlauf pos od. neg