

SYP – PRE

Themenbereich 2

Testen von Software

Nico Kandut, Angelika Babin 5AHIFS

2018/2019

Inhaltsverzeichnis

[1 Einleitung 4](#_Toc535417111)

[1.1 ISO-Normen 4](#_Toc535417112)

[1.1.1 ISO 9126 4](#_Toc535417113)

[1.1.2 ISO 25010 5](#_Toc535417114)

[1.2 Funktionalität der Software 5](#_Toc535417115)

[2 Aufbau eines Tests 5](#_Toc535417116)

[3 Testprozess 6](#_Toc535417117)

[3.1 Planungsphase 6](#_Toc535417118)

[3.1.1 Metrik 7](#_Toc535417119)

[3.2 Vorbereitungsphase 7](#_Toc535417120)

[3.3 Durchführungsphase 7](#_Toc535417121)

[3.4 Auswertungsphase 7](#_Toc535417122)

[3.5 Abschlussphase 7](#_Toc535417123)

[4 Testmethoden 7](#_Toc535417124)

[4.1 Black-Box-Tests 8](#_Toc535417125)

[4.2 White-Box-Tests 9](#_Toc535417126)

[5 Arten von Tests 9](#_Toc535417127)

[5.1 Schichtenweise Teststrategie nach Stufen 9](#_Toc535417128)

[5.1.1 Unit-Tests (= Modul-Tests, Komponenten-Tests) 10](#_Toc535417129)

[5.1.2 Integration-Tests 10](#_Toc535417130)

[5.1.3 System-Tests 11](#_Toc535417131)

[5.1.4 Acceptance-Tests 11](#_Toc535417132)

[5.2 Zusätzliche Teststrategien nach weiteren wichtigen Kriterien 11](#_Toc535417133)

[5.2.1 Performance-Tests 12](#_Toc535417134)

[5.2.2 GUI – Tests 12](#_Toc535417135)

[6 Testen in verschiedenen Prozessmodellen 13](#_Toc535417136)

[6.1 V-Modell 14](#_Toc535417137)

[6.1.1 Validation 14](#_Toc535417138)

[6.1.2 Verifikation 14](#_Toc535417139)

[6.2 RUP - Rational Unified Process 14](#_Toc535417140)

[6.3 Scrum 15](#_Toc535417141)

[7 Automatisierung des Testens 15](#_Toc535417142)

[8 Best Practice 15](#_Toc535417143)

[9 Quellen 16](#_Toc535417144)

# Einleitung

Das Testen von Software ist ein essenzieller Teil des Qualitätsmanagements. Ziel des Testens ist es, Fehler in den einzelnen Softwareteilen zu erkennen, die Benutzeranforderung zu erfüllen und die Qualität zu messen. Aufgrund der Testergebnisse kann die Software gezielt repariert bzw. verbessert werden. Man kann jedoch nie sicher sein, dass man alle möglichen Fälle getestet hat. Daher können Tests nie garantieren, dass eine Software bug-frei ist, sondern nur zeigen, wenn Bugs vorhanden sind.   
Die Vergangenheit hat allerdings gezeigt, dass mit der richtigen Teststrategie die Fehlerwahrscheinlichkeit deutlich verringert werden kann.

Da das Testen, um Qualität zu garantieren, wichtig ist; wurden spezielle Regeln (ISO-Normen) aufgestellt, wie das Testen ablaufen muss. Werden diese Regeln von einem Betrieb befolgt, dann kann er sich zertifizieren. Dadurch wird garantiert, dass der Standard gehalten wird und spezielle Maßnahmen zur Qualitätssicherung getroffen werden.  
In der ISO 9126 und der ISO-Gruppe 25000, speziell 25010, wird genauestens spezifiziert wie das Qualitätsmanagement & Testen ablaufen muss.

## ISO-Normen

### ISO 9126

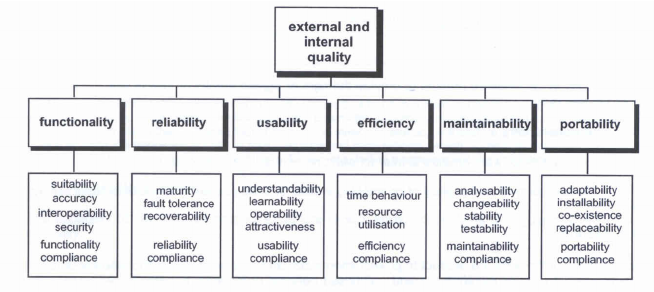
Ist der Vorgänger der späteren ISO 25010:2011.  


Abbildung 1: ISO 9126 - <https://www.researchgate.net/figure/ISO-9126-Internal-and-External-Quality-Characteristics-Level-5-Major-review-review-code_fig3_254916205>

### ISO 25010

Die ISO 25010 ist die aktuelle Zertifizierung für Qualitätsmanagement. Das Modell beinhaltet die in der ISO 9126 standardisierten Eigenschaften und zusätzlich welche für die Sicherheit und für die Kompatibilität des Systems. Es ist also eine Erweiterung der ISO 9126.

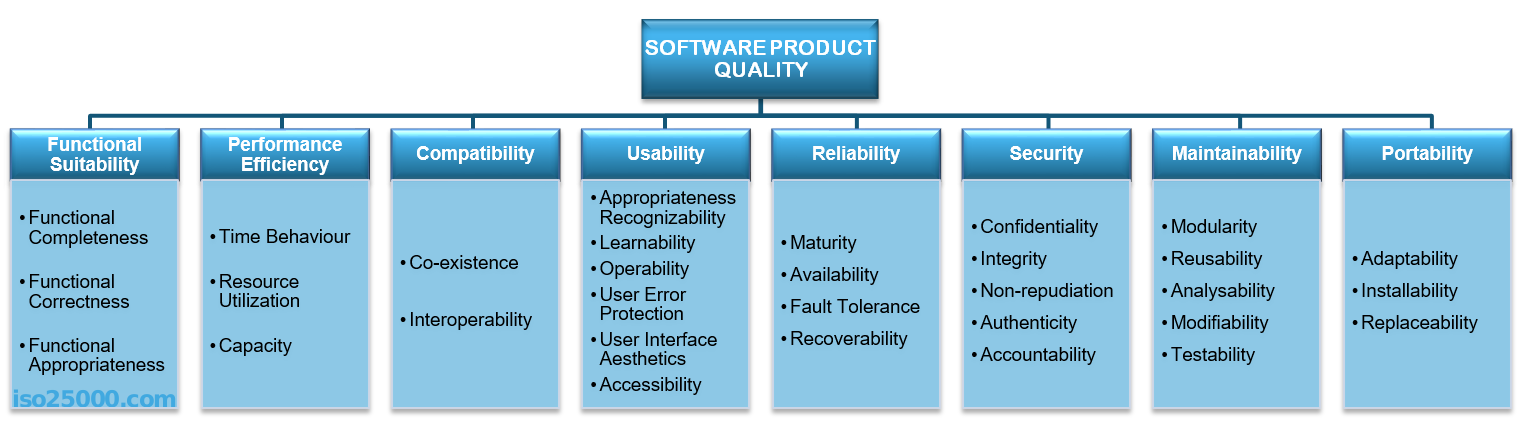


Abbildung 2 ISO 25010 - <https://iso25000.com/images/figures/en/iso25010.png>

## Funktionalität der Software

Durch Tests erhält man eine Übersicht über die Funktionalität der Software. Wichtig dabei ist die Unterscheidung zwischen Funktionalität aus Entwicklersicht und Funktionalität aus Anwendersicht, da es sonst zu Missverständnissen kommen kann.

Für den User bedeutet Funktionalität, dass das Programm sich genauso verhält wie er es sich vorgestellt hat. Dies kann mithilfe der Anforderungen im Pflichtenheft, mit sogenannten Akzeptanzkriterien oder bei Scrum über die User Stories gemessen werden.

Für den Entwickler bedeutet Funktionalität, einerseits dass eine Komponente ihren Zweck fehlerfrei erfüllt. Andererseits, dass die Komponenten mit den Anforderungen übereinstimmen.

# Aufbau eines Tests

Ein Testfall ist die Überprüfung von einem bestimmten Systemteil unter bestimmten Bedingungen, um überprüfen zu können, ob das Systemteil korrekt funktioniert.

Testfälle werden archiviert. Durch sie können Probleme im System gefunden werden und die benötigten Veränderungen zum Lösen des Problems herausgeleitet werden.

Ein Testfall kann diese Daten beinhalten:

* Test Suite ID
  + Die ID des Test Suites wohin dieser Testfall gehört
* Test Case ID
  + Die ID dieses Test Cases
* Test Case Summary
  + Kurze Zusammenfassung worüber der Testfall ist
* Related Requirement
  + Die IDs zu den Voraussetzungen die dieser Testfall benötigt
* Prerequisites
  + Alle Grundvoraussetzungen die vorhanden sein müssen, damit der Test ausgeführt werden kann
* Test Procedure
  + Step-by-Step alle Schritte des Tests aufgelistet
* Test Data
  + Die Daten die verwendet werden um den Test auszuführen (oder links zu ihnen)
* Expected Result
  + Das zu erwartende Ergebnis von dem Test
* Actual Result
  + Das Ergebnis nach dem Ausführen des Testes
* Status
  + Der Status nach dem Ausführen des Testfalles
  + Statusmöglichkeiten (Pass, Fail, Not Executed, Blocked)
* Remarks
  + Irgendwelche Kommentare zum Testfall oder dessen Ausführung
* Created by
  + Name des Erstellers des Testfalles
* Date of Creation
  + Datum der Erstellung des Testfalles
* Executed by
  + Name des Ausführers des Testfalles
* Date of Execution
  + Datum der Ausführung des Testfalles
* Test Environment
  + Die Umgebung (Hardware/Software/Network) in welchem der Testfall ausgeführt wurde

# Testprozess

Das Erstellen von Tests wird, egal auf welcher Ebene der Test/die Tests stattfinden, als Testprozess bezeichnet und in mehrere Phasen untergliedert.

## Planungsphase

In der Planungsphase der Tests wird entschieden, welche Teststrategie verwendet wird, um das Erreichen der Testziele in der Entwicklungsumgebung bestmöglich zu fördern. Zusätzlichen werden Risiken, Umfang, Organisation, Testmethoden, Prioritäten, Metriken, der zeitliche Ablauf der Testdurchführung und benötigte Ressourcen festgelegt.

### Metrik

Ist eine Funktion, die eine Eigenschaft von Software in einem Zahlenwert (Maßzahl) abbildet. Mit dieser Maßzahl können die Eigenschaften bewertet und verglichen werden.

## Vorbereitungsphase

Die Testinfrastruktur wird aufgebaut und überprüft ob die vorhandenen Daten genügen, um erste logische Testfälle zu erstellen, wenn die Daten Widersprüche oder Lücken aufweisen ist es nicht möglich gute Testfälle zu erstellen.

## Durchführungsphase

Die Testfälle werden in Gruppen oder Testsequenzen eingeteilt und bestimmten Szenarien zugeordnet.  
Zuerst werden Tests, die die Grundfunktionalitäten der Software nachweisen durchgeführt und in weiterer Folge werden Tests je nach Priorisierung durchgeführt. Die gesamte Testdurchführung wird genauestens Protokolliert.

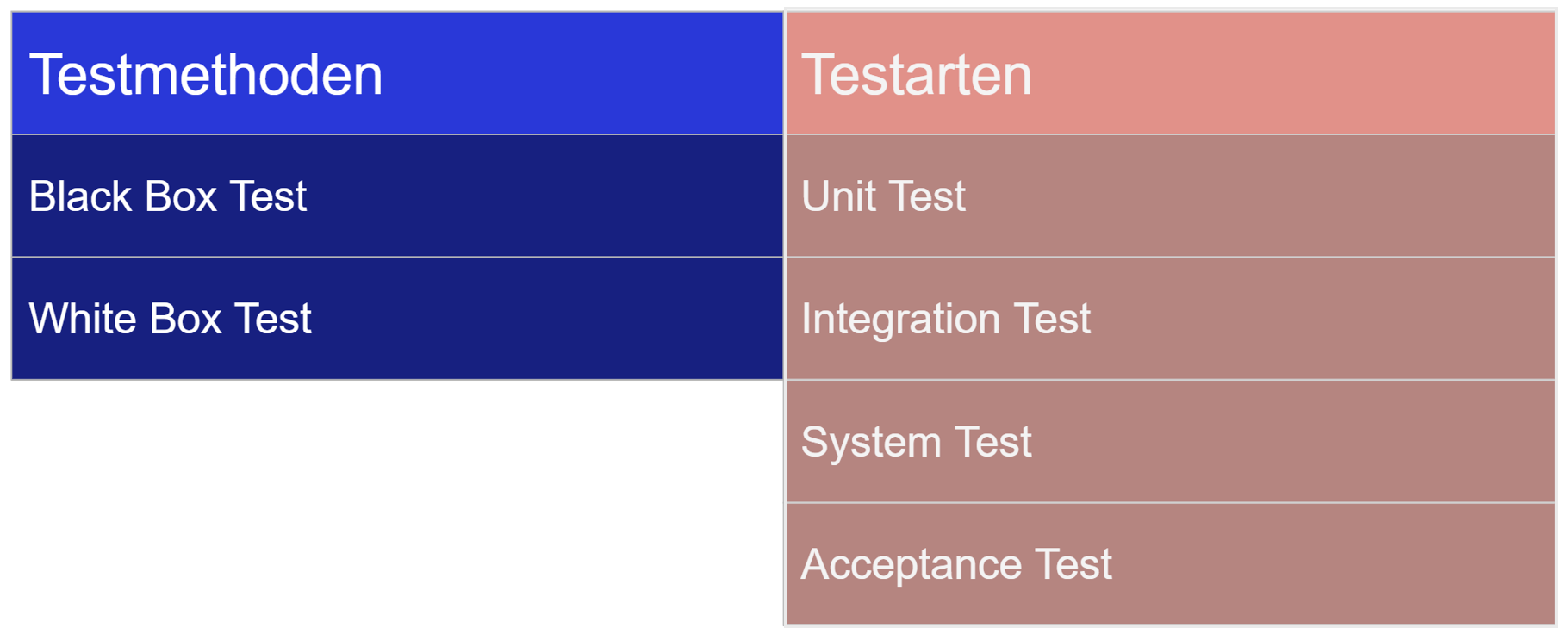
## Auswertungsphase

Durch die festgelegten Testendekriterien und die gesetzten Ziele in der Planungsphase wird entschieden ob weitere Tests durchgeführt werden müssen oder ob die Ergebnisse ausreichend für einen Abschlussbericht sind.

## Abschlussphase

In der letzten Phase werden alle Dokumentationen rund um die Software und die Tests zusammengetragen, um die gesammelten Informationen für zukünftige Wartungsarbeiten zur Verfügung zu haben.

# Testmethoden

Grundsätzlich gibt es gibt es zwei Testmethoden, die mehr oder weniger bei jedem Testverfahren verwendet werden. Die sogenannten Black-Box-Tests überprüfen die Prinzipien eines Systems, wohingegen die White-Box-Tests die Korrektheit der internen Struktur des Systems begutachten.  
Das getrennte Anwenden der Black-Box- und White-Box-Tests ist nicht ausreichend, um eine gut getestete Software zu garantieren. Daher wird angeraten sie parallel zu verwenden.

## Black-Box-Tests

Bei Black-Box-Test sind nur die Eingangs- und Ausgangsparameter bekannt, welche sich nach den Spezifikationen/Anforderungen richten. Der Code wird zum Erstellen und Ausführen der Tests nicht angesehen, was nur die Existenz von Fehlern beweisen kann und nicht die Abwesenheit.

Verschiedene Testverfahren sind dazu da Testfälle, welche mit hoher Wahrscheinlichkeit Fehler aufdecken, schnell herauszufiltern.

Unterschieden wird zwischen mehreren Blackbox-Testverfahren:

* **Fehlerbasiertes Testen**
* **Äquivalenzklassenbasiertes Testen**Hierbei werden Wertebereiche, in welchen sich die Software äquivalent verhält, betrachtet.

Zum Beispiel soll das Geburtsdatum eines Kunden überprüft werden. Welche Geburtsdaten im Endeffekt gültig oder ungültig sind, wird aus den Anforderungen abgeleitet. Zum Testen wird ein Intervall von 100 Jahren angenommen. Nun wird mit ungültigen Geburtsdaten in der Zukunft und in der Vergangenheit getestet.

* **Zustandsbasiertes Testen**

Hierbei werden die Test-Ergebnisse auf die richtigen Zustände aus dem, in der Softwareentwicklung entstandenen, Zustandsdiagramm geprüft, denn welcher Zustand der Richtig ist hängt von der Historie der Aktionen ab. Pro Zustand gibt es einen Test, der ihn überprüft. Zusätzlich werden die Zustandsübergänge, welche alle Zustände beinhalten, geprüft, um zu bestimmen ob der Zustand durch die richtige Abfolge an Ereignissen auftritt.

Das Testobjekt kann ein komplettes System mit unterschiedlichen Zuständen, aber auch eine Klasse mit verschiedenen Zuständen in einem objektorientierten System sein.

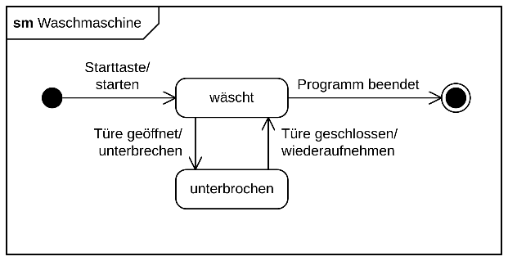
Im [Zustandsdiagramms](https://de.wikipedia.org/wiki/Zustandsdiagramm) wird das Testobjekt von einem Startzustand bis zu einem Endzustand modelliert. Vom Startzustand ausgehend, kann das Testobjekt verschiedene Zustände einnehmen. Auslöser für Zustandsänderungen können Ereignisse oder erfüllte Bedingungen sein.

Abbildung 3 Zustandsdiagramm https://de.wikipedia.org/wiki/Zustandsdiagramm (UML)

* **Testen mit Entscheidungstabellen**

Die Eingangsbedingungen und Aktionen, welche beide aus den Spezifikationen entnommen werden, des Testobjekts werden in Entscheidungstabellen logisch dargestellt und Testfällen zugeordnet. Sie werden durch das exponentielle Wachstum bei einer neuen Bedingung schnell unübersichtlich, deswegen wird die Anzahl der nötigen Tests begrenzt. Dies kann durch funktional abhängige Bedingungen, welche in der Praxis nicht auftreten passieren.

Sie sind keine Garantie für die Fehlerfreiheit der Software, da sie beim Softwareentwurf spätere Detail- und Implementationsentscheidungen nicht abdecken.

## White-Box-Tests

Tests werden mit Zugriff auf den Quellcode entwickelt. Es wird die Korrektheit eines Systems geprüft und nicht ob es eine Semantik erfüllt. Interne Sicherheitslöcher, gebrochene oder schlecht strukturierte Pfade in den Codierungsprozessen können aufgedeckt werden und erwartete Ergebnisse oder die Funktionalität von Schleifen werden getestet.

Vorteile des White Box Testens sind:

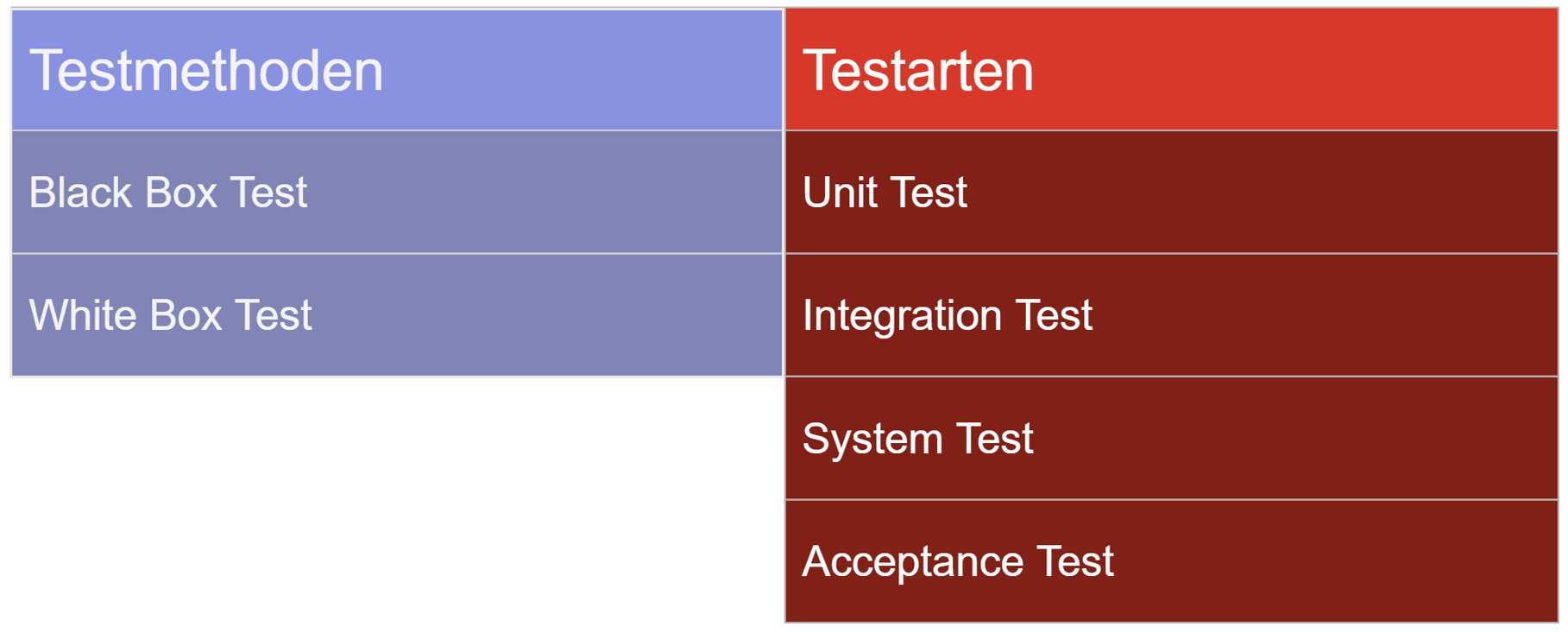
* Code-Optimierung
* Einfache Automatisierung
* Gründlicher durch das Überprüfen aller Codepfade
* Können früh im Projekt beginnen

Doch es gibt auch wesentliche Nachteile:

* Sie können teuer und komplex sein
* Für die Automatisation werden professionelle Tool und fundierte Kenntnisse vom Programm und Implementierung benötigt.
* Sie sind zeitaufwändig, besonders bei größeren Anwendungen kostet es Zeit alles vollständig zu testen

Das Lokalisieren fehlerverursachender Komponenten wird mit diesen Tests vereinfacht.

# Arten von Tests



## Schichtenweise Teststrategie nach Stufen

Grundsätzlich wird das Testen eines Systems in vier Schichten unterteilt (siehe Punkt 7.1 V-Modell). Sie beginnen dabei die kleinesten Teile der Software, sogenannte **Units**, zu testen, wenn hierbei keine Fehler gefunden werden, dann werden sie zu größeren Teilen zusammengefasst und ihr Zusammenspiel getestet (**Integration Tests**). Am Schluss wird die Software noch zwischen verschiedenen Rechnern getestet, um die volle Funktionstüchtigkeit des Systems zu garantieren (**System-Test**). Der Abnahmetest, auch **Acceptance-Test**, ist dann da, damit der Kunde sein Ok gibt.

### Unit-Tests (= Modul-Tests, Komponenten-Tests)

Ein Modul, also die kleinste Software-Einheiten, werden isoliert voneinander getestet. Alle Features (Methoden/Funktionen) und Klassen werden Tests unterzogen.

Es wird überprüft ob alle Inputwerte einer Funktion die richtigen Output-Werte liefern, dazu werden zahlreiche Testfälle definiert, die alle mögliche Input-Gruppen darstellen sollen. Eine Input-Gruppe ist eine Menge gleichartiger Testfälle zB Eine Funktion, die zwei natürliche Zahlen addiert, soll getestet werden. Um alle Input-Gruppen abzudecken, muss man die Funktion nicht mit allen Zahlen von -1.000.000 bis +1.000.000 ausführen, sondern nur sechsmal. Jeder Inputparameter soll einmal positiv sein, einmal negativ und einmal 0. Sollte der Code kritisch sein, muss genauer getestet werden, zB positiv + negativ mit positivem/negativem Ergebnis, Null als Ergebnis, etc.

Die Vorteile:

* Einmal Schreiben immer wieder ausführen
* Man weiß welcher Programmteil Fehler verursacht
* Zeitersparnis da man nicht mehr manuell testen muss

Nachteile:

* Hoher Aufwand bei der Erstellung
* Hoher Aufwand bei Veränderungen im zu testenden Code

Es gibt für jede Programmiersprache mindestens ein Unit-Testing-Framework, das die entsprechende Funktionalität zur Verfügung stellt. Zum Beispiel ist xUnit die Bezeichnung für verschiedene Frameworks für Modultests. Der bekannteste Vertreter ist JUnit für die Programmiersprache Java. Weiters verbreitet sind NUnit für das .Net-Framework, PHPUnit und QUnit für JavaScript.

### Integration-Tests

Erst wenn ein Modul alle Unit-Tests erfolgreich absolviert, wird es in der zweiten Stufe weiter getestet. Beim Integration-Testing wird die Interaktion, zB der Austausch von Daten durch Nachrichten, und die Kompatibilität des Moduls mit anderen Komponenten des Systems, zB die Nutzung gleicher Ressourcen, getestet. Zusätzlich werden die Schnittstellen zwischen den Modulen getestet. Die Reihenfolge der Implementation bestimmt auch die Reihenfolge der Integrations-Tests.  
Beginnt man am UI mit der Programmierung (**TOP DOWN**), so beginnt man auch mit dem Integration-Testing am UI, um die Verzögerungen zu minimieren.

Genauso verhält es sich, wenn man mit dem Backend beginnt (**BOTTOM UP**).

Werden alle Programmteile zur ungefähr gleichen Zeit fertiggestellt, so kann man auch alle auf einmal testen (**BIG BANG**), praktisch ein abgespeckter Systemtest.

Kombiniert man Top-Down und Bottom-Up, spricht man von **MIXED/SANDWICH**-Testing.

Wichtig beim Integrations-Testing ist es, die Anforderungen möglichst genau zu erfassen, da nicht definierte Fälle nicht getestet werden.

### System-Tests

Im Rahmen der Vorbereitung auf den Projektabschluss oder einen Termin wird das gesamte System in einer möglichst realitätsnahen Umgebung getestet. Der Systemtest ist im Grunde genommen einfach der komplexeste Integrationstest. Es werden keine Komponenten mehr durch Mockups ersetzt, stattdessen sind alle produzierten Module am Test beteiligt.

Außerdem werden in diesem Schritt Subsysteme zusammengebracht, was oft Inkonsistenzen und Missverständnisse aufgrund falscher Annahmen ans Tageslicht bringt. Hierbei ist das Risiko eines Fehlschlages beim „Scharfschalten“ der Systeme besonders groß und auch das Zusammenspiel verschiedener Technologien zB verschiedene Webbrowser birgt Gefahren.

Systemtests sind erst gegen Ende des Projekts sinnvoll, wenn von jedem Programmteil zumindest eine funktionierende Version existiert.   
Fertige Produkte, die alle Systemtests bestanden haben, nennt man Release Candidate, da sie dem Kunden präsentiert werden.

### Acceptance-Tests

Oder auch Abnahmetests werden vom Kunden oder eng mit dem Kunden zusammen durchgeführt. Anhand der Tests muss der Kunde entscheiden, ob die gelieferte Software den Anforderungen, welche im Pflichtenheft, in den Use Cases des Softwaresystems oder im Scrum sind es die User Stories, definiert wurden, entspricht. Akzeptanztests sind Black-Box-Tests.

Oft erfolgten Akzeptanztests durch die praktische Nutzung der Software. Hierbei kann der Auftraggeber/Kunde die Software nutzen und Feedback zur Funktionalität geben.

Grundsätzlich wird zwischen Alpha- und Betatests unterschieden:

#### Alphatests

Alphaversionen enthalten meist nur die grundlegenden Funktionen und sind meist an vielen Stellen noch unfertig.

#### Betatests

Betaversionen (Release Candidate) hingegen sind bereits vollständig ausgestattet. Sie haben allerdings noch keine Abnahmetests bestanden.

## Zusätzliche Teststrategien nach weiteren wichtigen Kriterien

Zusätzlich zu diesen vier großen Teststrategien können noch andere Tests wie **Performance-Tests** oder **GUI-Tests** eingebaut werden, die die Software zusätzlich von anderen Blickwinkeln heraus testen.

### Performance-Tests

Dies sind Tests, welche die Effizienz, wie Geschwindigkeit, Antwortzeit und Stabilität des Programms prüfen. Eigenschaften, sogenannte Metriken (siehe 3.1 Planungsphase von Tests), die etwas über die Performance des Systems Aussagen sind:

* Durchsatz: Wie viele Einheiten an Informationen das System über einen gewissen Zeitraum verarbeiten kann.
* Speicher: wie viel Arbeitsspeicher benötigt wird.
* Antwortzeit: Die Zeit, welche das System benötigt um auf eine Anfrage, request, zu reagieren.

Generell unterscheidet man zwischen zwei Haupt-Methoden bei Performance-Tests:

#### Load testing

Hierbei wird die Anzahl an erwarteten Benutzer über einen gewissen Zeitraum simuliert, um herauszufinden wie lange die Antwortzeit des Systems ist und um mögliche Engpässe, sogenannte „bottlenecks“, zu lokalisieren bevor das System in Betrieb genommen wird.

#### Stress testing

Um herauszufinden was die maximale Kapazität und die Skalierbarkeit des Systems ist, wird die Anzahl der Benutzer über die erwartete Benutzeranzahl gesetzt. Dabei werden auch die Hardware Ressourcen komplett ausgelastet, um zu Wissen ab wann das System abstürzt.

Ein gutes Bespiel für den Einsatzbereich von Performance Tests sind Online-Portale für den Kauf von Konzertkarten, die sich in den letzten Jahren stark in ihrer Performance gebessert haben. Am Anfang stürzten sie beim Verkaufsstart ab, denn es wurde nicht damit gerechnet, dass so viele Leute gleich beim Start darauf zugreifen.

### GUI – Tests

In der Softwareentwicklung „grafical user interface testing“ ist der Prozess des Testens eines Produkts ob es die Grafischen Nutzer Interface Spezifikationen auch erfüllt.

Test Case Generation

Um ein Set von Test Cases zu generieren versuchen die Test Designer alle Funktionalitäten der GUI auszutesten.

Es gibt drei Probleme:

* 1. Nicht wie ein CLI (Command Line Interface) System, hat eine GUI mehrere Operationen, die getestet werden müssen. Als Beispiel WordPad hat 325 mögliche GUI Operationen.
  2. Manche Events können nur ausgelöst bzw. überprüft werden durch eine Sequenz von GUI Operationen, wie z.B. das Speichern, was die Schwierigkeit Exponentiell steigert.
  3. Regressions Testing kann auch ein Problem werden beim GUI Testen. Es ist möglich, dass die GUI sich signifikant ändert jedoch das darunter nicht.

Es werden heutzutage schon Künstliche Intelligenzen (KIs) benutzt, um GUIs intelligent zu testen, um diese Probleme zu umgehen.

Es gibt 2 Methoden die GUIs zu testen, entweder:

1. Mouse position capture

Diese häufig gebrauchte Methode benutzt das Capture and Playback Verfahren. Man speichert zuerst alle Ausgänge mit Bitmap Grafiken und speichert das Testverfahren, um es wiederholen zu können und anschließend die Grafiken zu vergleichen.

1. Event Capture

Benutzt die GUI interaction data um die Events abzufangen und in logs zu speichern welche gefiltert werden.

# Testen in verschiedenen Prozessmodellen

Da Tests für das Garantieren der Qualität wichtig sind, werden sie in der Praxis auch in die verschiedenen Prozessmodelle eingebaut. Zu unterscheiden ist, wann getestet wird und wie oft. Modelle, wo erst am Schluss das komplette System getestet wird, werden heutzutage fast gar nicht mehr verwendet, wie zum Beispiel das Wasserfallmodell.

Je später ein Fehler entdeckt wird, desto teurer und zeitaufwändiger wird es, ihn zu beheben oder im schlimmsten Fall ist er gar nicht mehr zu beheben. Wie sich ein Fehler auf die Kosten auswirkt, wird in der Fehlerbehebungskostenkurve dargestellt.

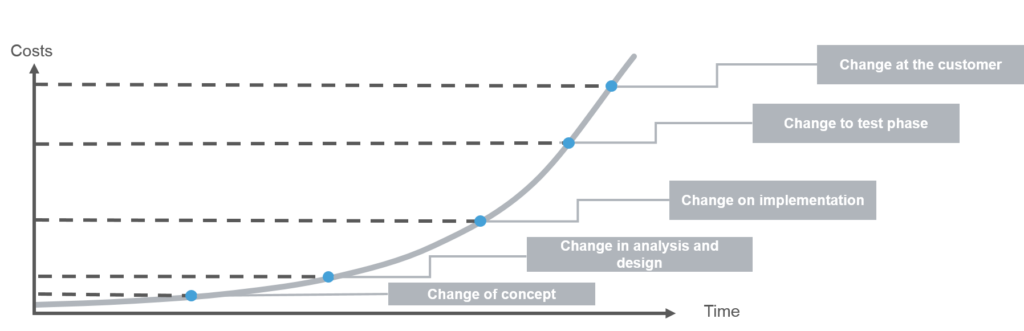


Abbildung 2 Fehlerkostenkurve- <https://blog.exxeta.com/energy/2017/04/11/requirements-engineering-in-der-energiewirtschaft/?lang=de>

Folgende Prozessmodelle werden als Beispiel aufgezählt, um zu zeigen wie das Testen in den Entwicklungsprozess integriert wird.

## https://lh5.googleusercontent.com/q3rg7PeKGcDCx-RP7z6lbwm4Ha4fEgorhfpTisI26lJJLRJlXq5PxWzMItKYZ7AIt6k6FzexZIv23MAPZpy9JukELO8jaSYHWaXCrEr-Tt_zJuErM60Fwx-6F_vKDZGnGox8eYQV-Modell

Das V-Modell wurde 1979 von Barry Boehm aus dem Wasserfallmodell weiterentwickelt und wurde zuerst für staatliche Projekte eingesetzt.  
Die linke Flanke beinhaltet die Analyse und das Design des Systems. Das Ziel dieser Flanke ist es das komplette System zu programmieren, wenn alles programmiert wurde, beginnen die Testphasen.

Je implementierungsnäher sich etwas ändert desto weniger Auswirkungen auf das System bestehen. Die Testfälle bekommen Informationen, was zu testen ist aus ihren vorangegangenen Phasen bzw. können die Tests auch schon fix und fertig festgelegt werden. Es wird von der detailliertesten Einheit zur ungenauesten getestet. Bei den Tests unterscheidet man zwei wichtige Bestandteile:

### https://lh3.googleusercontent.com/Biyhb9z8wEdTcoGOWTtrZN23hG4GOkiAndsZ8N7mJuQ7tZIe8iexpgeboSUi6Q8giMIrm-6r9BXX3o2Or_1C8inPUGF592fx2P8wA1b2onSNWTKz7LrqlpmBoVjZmy7c-RCcXtEValidation

Die Eignung eines Produktes bezogen auf seinen Einsatzzweck.  
“Wird das richtige Produkt entwickelt?”  
Die Validation fällt unter die Methode der Black-Box-Tests, denn das Ergebnis wird angeschaut.

### Verifikation

Die Überprüfung der Übereinstimmung zwischen einem Softwareprodukt und seiner Spezifikation. “Wird ein korrektes Produkt entwickelt?”

Die Verifikation fällt in die Methode des White-Box-Testens, denn es wird der Code auf die Korrekte Programmierung hin untersucht.

## http://www.infforum.de/images/rup-modell.jpgRUP - Rational Unified Process

Im RUP wird das Testen auf der senkrechten Achse als eigener Workflow aufgeführt. Die Ausschläge für die Tests sind nach jeder Iteration und/oder jeder Phase (auf der waagrechten Achse).

## Scrum

Im Scrum erfolgt das Testen für jede fertige Story oder am Ende des Sprintes aber vor dem Sprint Review. Das Qualitätsabnahme-Team testet die vom Product Owner freigegebenen Stories, wenn ein Fehler gefunden wird, dann entscheidet der Product Owner je nachdem ob noch Zeit ist, ob der Fehler noch in diesem Sprint oder erst im nächsten behoben werden muss. Zusätzlich wird im Sprint Review, das fertige Artefakt von den Stakeholdern getestet, wenn diese Fehler beanstanden, dann wird dies in den nächsten Sprint aufgenommen.

## https://www.it-agile.de/fileadmin/images/tdd-zyklus.jpgTest-Driven-Development (TDD)

Ist eine Technik, welche inkrementell vorgeht und bei der zuerst die Tests und darauf erst die Software geschrieben wird. Die Software beinhaltet genau so viel Code, um erfolgreich durch den Test zu kommen. Wird zu viel Code geschrieben, entstehen Stellen die nicht getestet sind, was beim Refactoring Probleme bereiten kann. Test und Code werden refaktorisiert. Also aufgeräumt und um neue Fähigkeiten erweitert. Jeder Zyklusabschnitt soll nicht länger als ein paar Minuten in Anspruch nehmen.

Abbildung TTD-Zyklus: https://www.it-agile.de/wissen/agiles-engineering/testgetriebene-entwicklung-tdd/

Der große Vorteil davon ist, dass zuerst die Requirements und Designs genauestens durchdacht werden müssen.

# Automatisierung des Testens

Heute gewinnen Continuous-Integration/Continuous-Delivery-Systeme wie Scrum, bei denen nicht die gesamte Software am Anfang geplant wird, sondern immer wieder kleine, voll funktionsfähige Programmteile ausgeliefert werden, immer mehr an Bedeutung. Besonders bei solchen Prozessmodellen ist es wichtig, dass der Testaufwand möglichst gering ist, da sehr viele Tests anfallen und immer wieder wiederholt werden müssen (jede Iteration). Diese Herausforderung kann mithilfe von automatischen Testsystemen (npm, Jenkins, Github, ...) bewältigt werden.

Automatisierbare Bereiche reichen von der Testfallerstellung, über die Testausführung, bis zur Testdokumentation. Meistens werden die Testfälle per Hand geschrieben und automatisiert ausgeführt.

Beispiele

* **Unit-Testing**

Einzelne Funktionen werden mit verschiedenen Inputs ausgeführt und auf ihre Richtigkeit überprüft.

* **GUI-Testing**

Es werden Inputs wie Mausklicks simuliert, und die Veränderungen am User-Interface werden als Output erfasst. (Siehe Punkt 4.6 GUI-Tests)

# Best Practice

* So gut wie es geht, sollen Test-Cases nur eine Sache überprüfen. Man sollte sie nicht mit anderen Test-Cases überlappen oder umsonst komplizierter machen.
* Probiere sicher zu gehen, dass alle positiven, als auch negativen Szenarien gedeckt sind.
* Sprache:
  + Schreib in einfacher und leicht zu verstehender Sprache
  + Verwende aktive Sprache: Tu dies, Tu das.
  + Verwende für Felder und Forms die exakten Namen und umschreib diese nicht.
* Eigenschaften eines guten Test-Cases:
  + Akkurat: genau zum Zweck
  + Sparsam: Keine unnötigen Wörter.
  + Zurückverfolgbarkeit: Kann zu den Anforderungen zurückverfolgt werden.
  + Wiederholbarkeit: Der Test-Case kann immer wieder und wieder ausgeführt werden
  + Wiederverwendbarkeit: Kann falls nötig wiederverwendet werden

# Beispiele

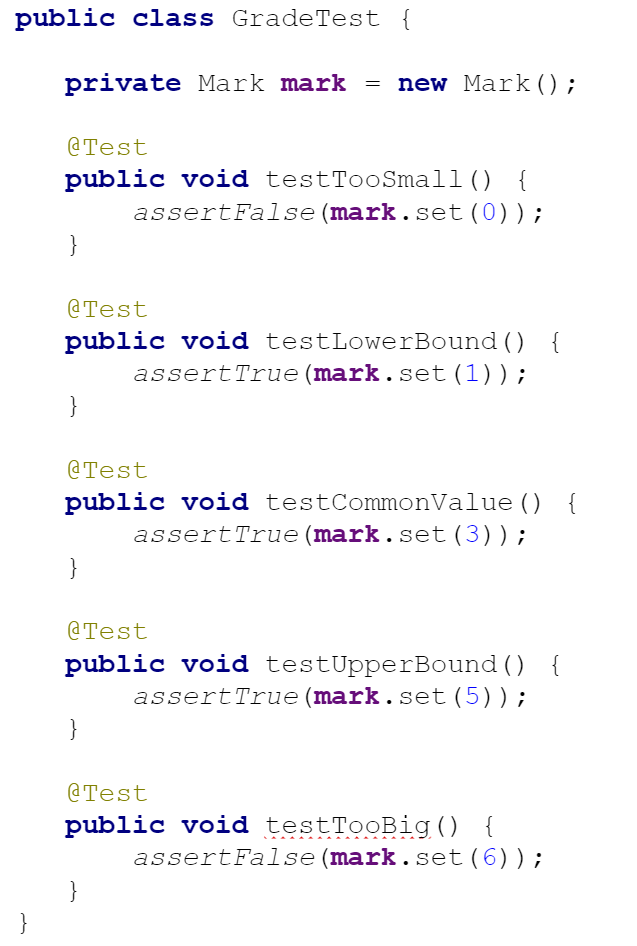
USER STORY: EINGABE EINER ZEUGNISNOTE

In dieser Userstory soll der User, ein Lehrer, die Zeugnisnote eines Schülers in das System der Schule eintragen.

Akzeptanzkriterien:

* Die Note darf nicht kleiner als 1 sein.
* Die Note darf nicht größer als 5 sein.

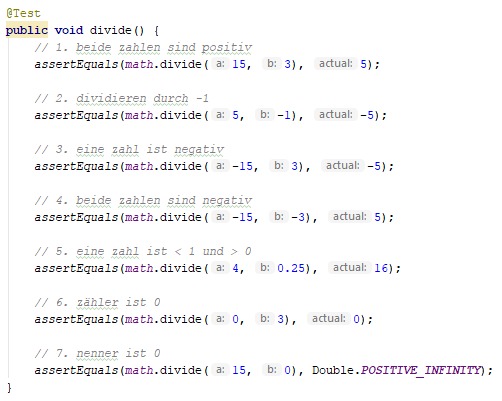
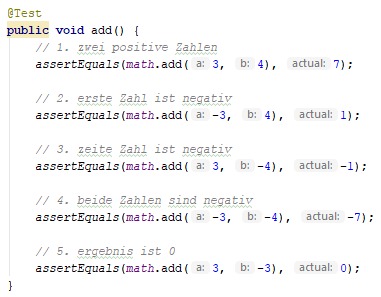
Testfälle:

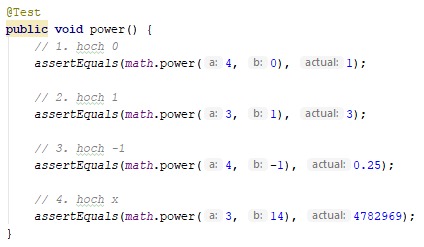


MATURA BEISPIEL

In der 1. Klasse der Informatikabteilung lernen die Schüler wie man einen Taschenrechner programmiert. Um die Programme der Schüler mittels der Black-Box Testmethode überprüfen zu können, sollen für Funktionen des Taschenrechners JUnit Tests von den Höheren Klassen programmiert werden.

Erklären sie was eine Black-Box-Testmethode ist, welche Methoden darunterfallen und welche andere Testsmethode es gibt.

Erstellen sie JUnit Tests für Addition, Division und Potenzieren. 



# Quellen

* <http://softwaretestingfundamentals.com>
* <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Hauptaktivitaten-der-Systementwicklung/Software-Implementierung/Testen-von-Software/Akzeptanztest>
* <https://de.wikipedia.org/wiki/Testautomatisierung>
* https://www.springer-campus-it-onlinestudium.de/w3lmedia/W3L/Medium186039/9783868340129\_Leseprobe.pdf
* <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Hauptaktivitaten-der-Systementwicklung/Software-Implementierung/Testen-von-Software/Modultest>
* <https://www.johner-institut.de/blog/iec-62304-medizinische-software/blackbox-testing/>
* https://www.itpedia.nl/de/2018/02/05/white-box-testing-onder-de-loep/
* <https://de.wikipedia.org/wiki/Zustandsdiagramm_(UML)>
* <http://softwaretestingfundamentals.com/white-box-testing/>
* <http://www.gm.fh-koeln.de/~winter/tav/html/tav30/TAV30P07_ANECON.pdf>
* <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Hauptaktivitaten-der-Systementwicklung/Software-Implementierung/Testen-von-Software/Integrationstest>
* <http://softwaretestingfundamentals.com/test-case/>
* <http://agiledata.org/essays/tdd.html>
* <https://www.it-agile.de/wissen/agiles-engineering/testgetriebene-entwicklung-tdd/>