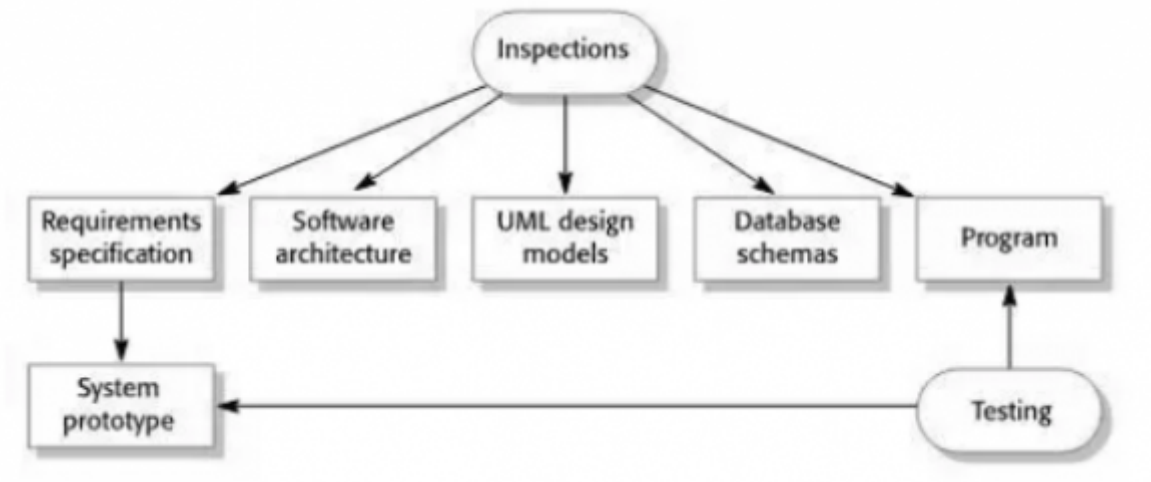
# Softwaretests und Unit Tests

Hierzu müssen Sie, wie schon Ian Sommerville in Software Engineering 8th Edition; Addison Wesley 2007 sagte, zwischen Validierung „Bauen wir das richtige Produkt?“ und der Verifikation „Bauen wir das Produkt richtig?“ unterscheiden.

Zwei sich ergänzende Ansätze zur Verifizierung und Validierung (V&V):

Softwareinspektionen oder Peer Reviews sind eine statische Technik, wobei eine Softwareinspektionen in jeder Phasen der Softwareentwicklung durchgeführt werden kann. Also Sie können jede einzelne Komponente auf Korrektheit überprüfen. Nehmen Sie an, Sie wollten ein autonomes Auto bauen, dann können Sie schon nach dem Einbau der Bremsen testen, ob die Bremsen Funktionieren. Aber ob Ihr autonomes Auto korrekt fahren kann, sehen Sie leider erst zum Schluss, wenn alle Komponenten verbaut sind und sie die Zusammenarbeiter der einzelnen Komponenten testen können.




Quelle: quoracdn.net

Bei der Softwareinspektionen überprüfen Sie die Korrespondenz zwischen einem Programm und seiner Spezifikation.

Programminspektionen: Ziel ist es, Programmfehler, Verstöße gegen Standards und schlechten Code zu finden, anstatt umfassendere Designfragen zu berücksichtigen. Programminspektionen wird in der Regel von einem Team durchgeführt deren Mitglieder den Code systematisch analysieren. Eine Inspektion wird in der Regel von Checklisten begleitet. Studien haben gezeigt, dass eine Inspektion von ungefähr 100 Zeilen Code ungefähr einen Personentag erfordern.

Automatisierte Code-Analyse Umfasst unter anderem Kontrollflussanalyse, Datennutzungs- / Flussanalyse, Informationsflussanalyse und Pfadanalyse. Statische Analysen machen auf Anomalien aufmerksam.

Eine formale Verifizierung kann das Fehlen bestimmter Fehler garantieren. Zum Beispiel, um zu gewährleisten, dass ein Programm keine Dead Locks, Race-Bedingungen oder Pufferüberläufe enthält. Das bedeutet aber auch, dass man nicht beweisen kann, dass eine Software fehlerfrei ist.

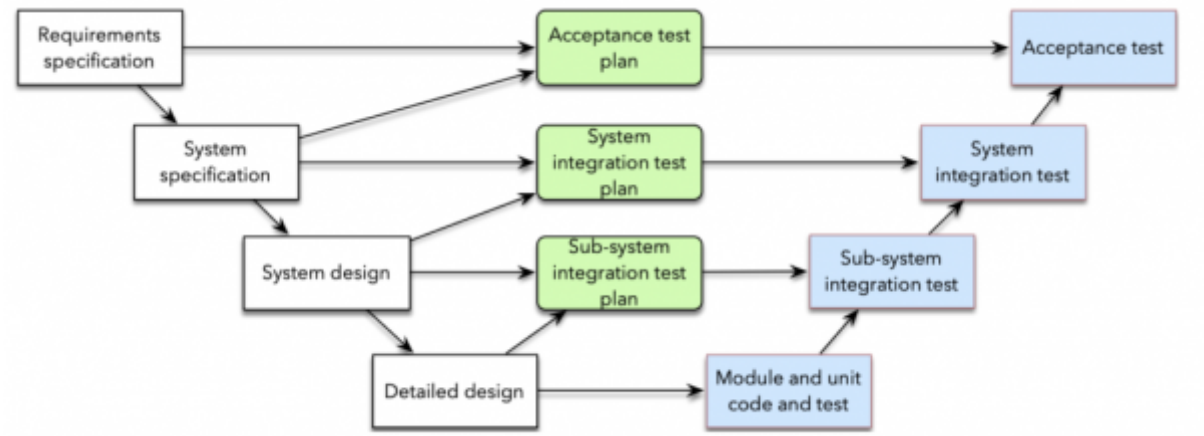
Software-Inspektionen belegt nicht, dass die Software auch nützlich ist.

Software Testing ist dagegen eine Dynamische Technik. Softwaretest bezieht sich auf das Ausführen einer Implementierung der Software mit Testdaten, um Programmfehler zu entdecken.

Die Validierungsprüfung soll zeigen, dass die Software den Vorgaben des Kunden entspricht, dabei sollte grundsätzlich für jede Anforderung ein Testfall vorliegen.

Die Fehlerüberprüfung soll Fehler aufdecken. D.H. gerichtete und beabsichtigte Fehler aufzudecken. Ebenso die Konformität überprüfen, welche bestimmt wird, wenn die Konformität mit den erforderlichen Fähigkeiten nachgewiesen werden soll. Beispiel, Ein Rechen-Programm soll nur Zahlen als Eingabe akzeptieren. Also Können Sie hier gezielt auf alle anderen Fehl-Eingaben überprüfen.

Testpläne legen den Zeitplan und die Verfahren der Tests fest. Sie setzen Standards für den Testprozess und Sie entwickeln während des gesamten Entwicklungsprozesses passende Tests.




Der Testumfang umfasst die Sammlung der zu überprüfenden Softwarekomponenten.

- **Unit-Tests** oder **Modultest**: sind relativ kleine ausführbare Testes, zum testen eines einzelnes Objekts oder einer Methode.
- **Integrationstest**: sind Komplettes für (Sub-)Systeme, hier weden die Schnittstellen zwischen Einheiten und deren Bedienbarkeit im ganzen überprüft.
- **Systemtest**: ist ein Test über die vollständig integrierte Anwendung. Kategorisiert nach Konformität: Funktion, Leistung, Stress oder Belastung

Aber auch hier kann nur das Vorhandensein von Fehlern angezeigt werden, nicht das Fehlen der Fehler. Das Design von Tests ist ein mehrstufiger Prozess.

- Identifizieren, modellieren und analysieren Sie die Verantwortlichkeiten des getesteten Systems (SUT System Under Test), verwenden Sie Vor- und Nachbedingungen, die in Anwendungsfällen als Eingabe identifiziert wurden.
- Entwerfen Sie Testfälle basierend auf dieser externen Perspektive
- Fügen Sie Testfälle hinzu, die auf Codeanalyse, und Verdacht auf Heuristiken basieren
- Entwickeln Sie erwartete Ergebnisse für jeden Testfall oder wählen Sie einen Ansatz, um den Status Bestanden / Nicht Bestehen jedes Testfalls zu bewerten

Ein automatisierte Testumgebung (TAS Test Automation System) wird zur Durchführung der Tests verwendet.start the implementation under test (IUT)

- Starten Sie die Implementierung unter Test (IUT)
- richten Sie eine (Test-) Umgebung ein

- erzwingen Sie einen für den Test erforderlichen Zustand
- verwenden Sie viele unterschiedliche Testeingaben
- bewerten Sie die resultierenden Ausgabe und den Zustand

Komplexe Systeme erfordern häufig eine erhebliche Anpassung der vorhandenen automatisierten Testumgebung.

Das Ziel der Testausführung besteht darin, festzustellen, dass die getestete Implementierung (IUT Implementation Under Test) durch Ausführen der Schnittstellen zwischen ihren Teilen minimal funktionsfähig ist.

- Führen Sie dazu die Testsuite aus. Das Ergebnis jedes Tests wird als bestanden oder nicht bestanden bewertet
- Nutzen Sie Tools um die Testabdeckung auszuwerten
- Falls erforderlich, entwickeln Sie zusätzliche Tests, um aufgedeckten Code auszuführen
- Stoppen Sie den Test, wenn das Testziel erreicht ist

# Test Point oder Testdatum, Testpunkt

Ein Testpunkt ist ein bestimmter Wert für …

- Testfall-Eingabe
- eine Zustandsvariable
- Der Testpunkt wird aus einer Domäne ausgewählt. Die Domäne ist die Menge von Werten, die Eingabe- oder Zustandsvariablen annehmen können.

## Heuristiken

Heuristiken zur Testpunktauswahl:

- Äquivalenzklassen
- Grenzwertanalyse
- Sonderwertprüfung

# Test Case oder Testfall

Testfälle spezifizieren:

- Vorprüfungsstatus der getesteten Implementierung (IUT implementation under test)
- Testeingaben / Bedingungen
- erwartete Ergebnisse

# Test Suite

Eine Testsuite ist eine Sammlung von Testfällen

# Test Run oder Testlauf

- Ein Testlauf ist die Ausführung (mit Ergebnissen) einer Testsuite
- Das IUT liefert tatsächliche Ergebnisse, wenn ein Testfall darauf angewendet wird. Ein Test, dessen tatsächliche Ergebnisse den erwarteten Ergebnissen entsprechen, gilt als bestanden

# Test Driver und automatisierte Test-Frameworks

- Test driver ist eine Klasse- oder Hilfsprogramm, das Testfälle auf ein IUT anwendet
- Test automatisierte Test-Frameworks sind ganze Systeme von Testtreibern und anderen Tools zur Unterstützung der Testausführung

# Failures, Errors & Bugs

- Failure = Defekt oder Fehlschlag
- Fault = Mangel
- Error = Fehler

Ein Fehler ist der manifestierter Fehlschlag eines Systems oder einer Komponente, die eine erforderliche Funktion innerhalb der angegebenen Grenzen nicht erfüllt. Dies kann auf Grund eines Softwarefehler, fehlendem Code oder falscher Code liegen. Ein Fehler ist eine menschliche Handlung, die einen Softwarefehler verursacht

- Bug: error or fault

# Test Plan

Ein Testplan ist ein Dokument für den menschlichen Gebrauch, das einen Testansatz erläutert, darunter den Arbeitsplan, die allgemeinen Verfahren und die Erläuterung des Testdesigns.

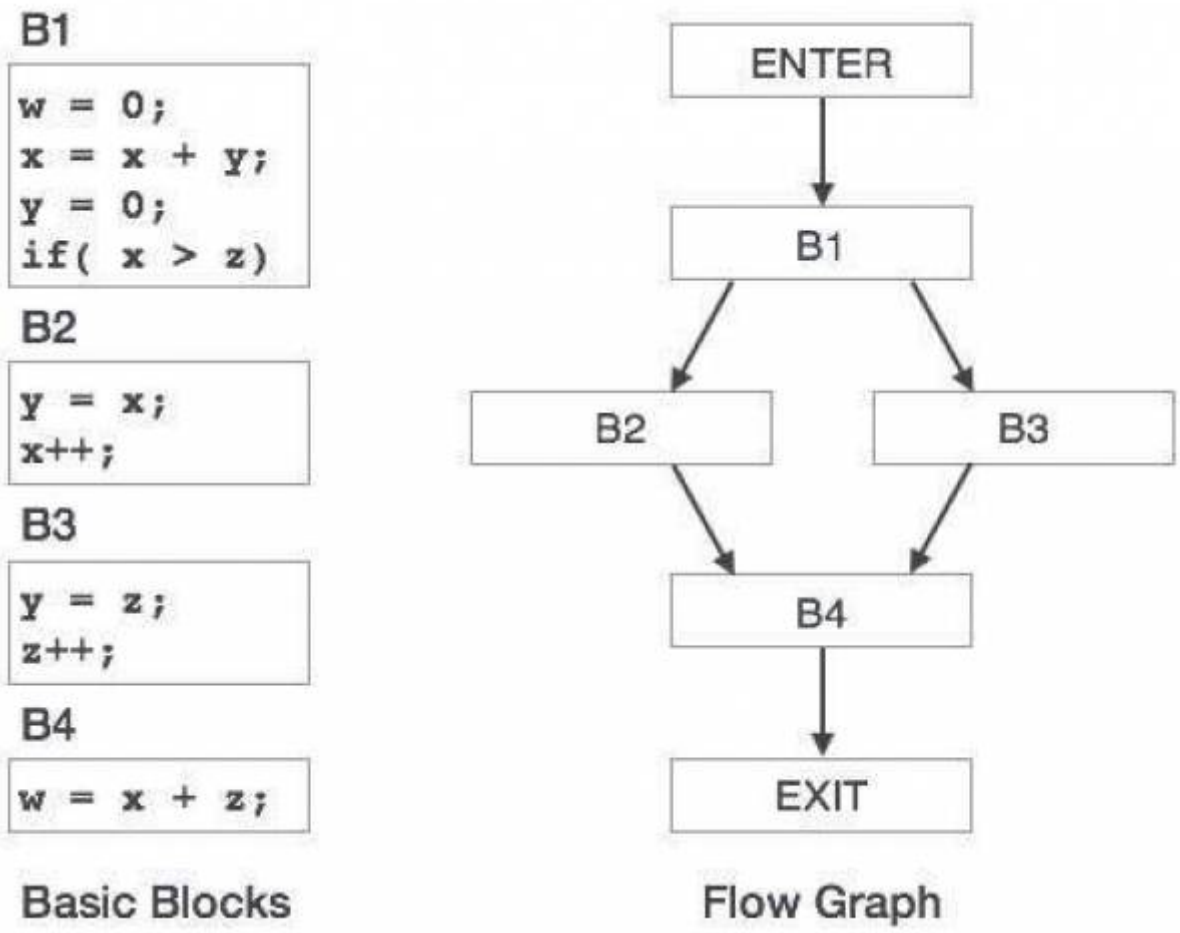
Das Testen muss auf einem Fehlermodell basieren. Da die Anzahl der möglichen Tests unendlich sind, darum müssen Sie aus praktischen Gründen eine Annahme machen, wo welche Fehler am wahrscheinlichsten zu finden sind!

Es gibt zwei allgemeine Fehlermodelle und entsprechende Teststrategien:

- Konformitätsgesteuertes Testen
- Fehlergesteuertes Testen

## Control-flow Graph

Eine Darstellung aller Pfade durch ein Programm.




Quelle: tutorialspoint.com

- Statement Coverage wird erreicht, wenn alle Anweisungen in einer Methode mindestens einmal ausgeführt wurden
- Branch Coverage wird erreicht, wenn jeder Pfad von einem Knoten mindestens einmal von einer Testsuite ausgeführt wird. zusammengesetzte Prädikate werden als eine einzige Anweisung behandelt
- Einfache Zustandsabdeckung erfordert, dass jede einfache Bedingung mindestens einmal als wahr und falsch bewertet wird (Hierbei müssen nicht alle möglichen Zweige getestet werden.)
- Zustandsüberdeckung = Einfache Zustandsüberdeckung + Zweigabdeckung
- Die Abdeckung mehrerer Bedingungen erfordert, dass alle wahr / falsch-Kombinationen einfacher Bedingungen mindestens einmal ausgeführt werden
- Modifizierte Bedingungs- / Entscheidungsdeckung (empfohlen für z. B. SIL 4-Software)

In Java beispielsweise, werden einfache atomare Bedingungen durch die Operatoren „&&“ oder „||“ getrennt.

Ein Basisblock ist eine Sequenz aufeinanderfolgender Anweisungen, bei denen der Steuerungsfluss am Anfang eintritt ,und am Ende ohne Anhalten oder einer Verzweigungsmöglichkeit wieder verlässt. Eine Basisblockabdeckung wird erreicht, wenn alle Basisblöcke einer Methode ausgeführt werden. Manchmal wird „Anweisungsabdeckung“ als Synonym für „Basisblockabdeckung“ verwendet.

Auf Bytecode-Ebene (oder niedriger) gibt es keine explizite Unterstützung für bedingte boolesche Operatoren und wird daher unter Verwendung entsprechender „if“ s kompiliert.

**Leitfaden**

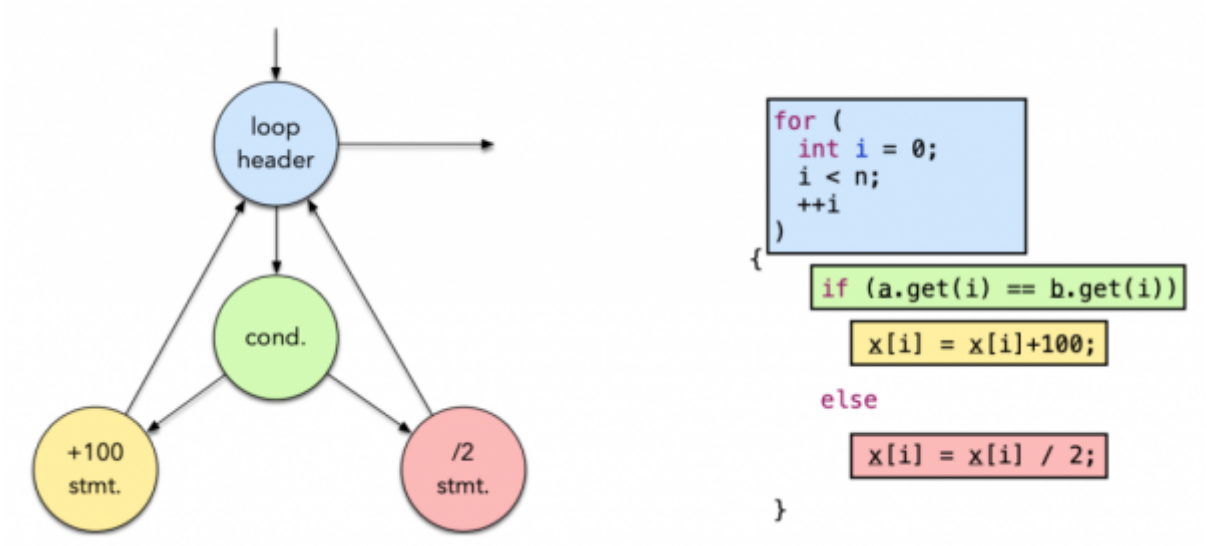
Einen sehr guten Leitfaden finden sie unter: Analyse der Codeabdeckung

# Grenzen für Tests

Die Anzahl der Eingabe- und Ausgabekombinationen für triviale Programme ist bereits sehr groß, wenn nicht sogar unbeschränkt.

Angenommen, wir möchten eine Linie zeichnen und beschränken die Eingabe der Punkte auf positive Ganzzahlen zwischen 1 und 10. Dann gibt es allein schon 10*10 Möglichkeiten für den ersten Punkt und nochmal 10*10 Möglichkeiten für den zweiten Punkt, also insgesamt 10*10*10*10 = 10.000 Möglichkeiten, eine einzige Linie zu zeichnen. Da ein Dreieck drei Punkte hat, haben wir dann 10.000 x 10.000 x 10.000 mögliche Eingaben für die drei Zeilen, einschließlich der ungültigen Kombinationen. Wir können niemals alle Eingänge, Zustände oder Ausgänge über handgeschriebene Unittests testen.

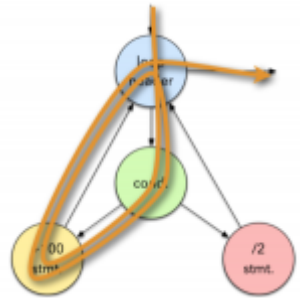
Verzweigung und dynamisches Binden (dynamic binding) führen zu einer sehr großen Anzahl eindeutiger Ausführungssequenzen. Eine einfache Iteration erhöht die Anzahl möglicher Sequenzen auf astronomische Proportionen.




Quelle: Software Technology Group and Reactive Programming, TU Darmstadt VL WS18

Wenn wir die Entry-Exit-Pfade ohne die Iteration zählen, gibt es nur drei Pfade, der erste ist der Loop-Header der direkt terminiert (blau, Ende). Der zweite Pfad ist der Loop Header über eine Condition und +100 (blau, grün, gelb, blau, Ende) und den letzten Pfad, über den Loop Header, eine Condition und dann / 2 (blau, grün, rot, blau, Ende).

Fängt man nun noch an zu iterieren, werden die Pfade exponentiell groß:




1 Itteration besteht aus 3 Pfaden (siehe oben) 2 Itterationen besteht dann schon aus 5 Pfaden 3 Itterationen besteht dann aus 9 Pfaden 10 Itterationen besteht dann aus 1025 Pfaden 20 Itterationen besteht dann aus 1.048.577 Pfaden

## Fehlerempfindlichkeit

Die Fähigkeit des Codes, Fehler aus einer Testsuite auszublenden, wird als Fehlerempfindlichkeit bezeichnet.

Zufällige Korrektheit wird erreicht, wenn fehlerhafter Code für einige Eingaben korrekte Ergebnisse liefern kann. Z.B. vorausgesetzt, der korrekte Code wäre: y = x + x, doch bei der Implementierung wurde: y = x * x geschrieben. Testen Sie nun mit den Werten x = 2 oder x = 0 so erhalten Sie für y den erwarteten Wert. Man spricht dann von Code der den Fehler versteckt. Es wird ein fehlerhaftes Ergebnis aus fehlerhaftem Code berechnet, doch der Fehler ist ersichtlich und nur schwer zu entdecken.

Beispiel: Sie schreiben ein Rechenprogramm, welches die umgekehrte polnische Notation (UPN) oder auch Postfixnotation verwendet um folgende Rechenausdrücke lösen zu können: „4,5,+“ oder „1,2,*,3,+,2,3,*,*“. Nun müssen Sie Ihr Programm auf Korrektheit überprüfen. Welche Eingaben sind valide und welche nicht. Welche Fehler müssen geworfen werden, wenn eine Eingabe ungültig ist. wir schreiben Test, für die validen Eingaben: „4,5,+“ gleich 9 unf „1,2,*,3,+,2,3,*,*“ gleich 30. nun müssen wir Eingaben wie: „4,5“, „4,5,+,+“, „“, „2,+,3″,“3hx5“ und „+4“ abfangen bzw. mittels Fehlerbeschreibung behandeln. Herzu unterstützt die Code Coverage Visualisierung eclemma ungemein, mit der wir sehen können, welche Codeteil unsere Tests abdecken.

## Verhaltensorientierte Entwicklung

Weitere sehr nützliche Programme und Methoden sind: TestNG, ScalaTest oder hamcrest. Und dann wären wir wieder bei der Verhaltensorientierte Entwicklung bei der die Entwickler die Verhaltensabsicht des Systems definieren und dann dieses entwickeln. http://behaviour-driven.org/

## (Methoden-) Stub

Ein Stub ist eine temporäre Implementierung einer Komponente vergleichbar mit einem Platzhalter für eine noch unvollständige Komponente. Stubs werden häufig benötigt, um komplexe Systeme zu simulieren und Teile komplexer Systeme isoliert nachprüfbar zu machen.

# Mockup

Eine Alternative ist die Verwendung eines Mock-Objekts, welches das ursprüngliche Objekt in seinem Verhalten nachahmt und das Testen erleichtert.

Testen umfasst die Bemühungen, Fehler zu finden. Beim Debuggen werden Fehler gefunden und korrigiert. Das Debuggen ist daher kein Test und das Testen ebenso kein Debugging!

Die Tests müssen systematisch durchgeführt werden. Tests die alle Möglichkeiten abdecken sind nicht möglich. Test-Coverage-Modelle helfen Ihnen dabei die Qualität Ihrer Testsuite einzuschätzen. DasErreichen eines Testabdeckungsziels ist in der Regel keinesfalls ausreichend. Nehmen Sie eine externe Perspektive ein, wenn Sie Ihre Testsuite entwickeln und prüfen Sie darum auch für Sie selbst „unlogisch“ erscheinende Eingaben ab. Gehen Sie immer von der schlimmst möglichen Eingabe aus.

# Testabdeckung

„Der Geschäftsführer eines Softwareunternehmens, welche eine Flugsoftware für eine Firma entwickelte, verkündete voller Stolz, dass die entwickelte Software nun in einem Flugzeug installiert wurde und die Fluggesellschaft den Mitarbeitern des Softwareunternehmens kostenlose Erstflüge anbietet. Der Geschäftsführer fragt seine Mitabriter, „Wer Interesse an einem kostenlosen Erstflug hätte?“. Niemand von den Mitarbeitern zeigte Interesse an einem kostenlosen Erstflug. Nach einer Weile meldete sich ein tapfere Softwaretester freiwillig und erklärt: „Ich werde es tun. Denn ich weiß ganz genau, dass das Flugzeug garnicht abheben kann.“ Unbekannter Autor von auf softwaretestingfundamentals.com

From:
https://wiki.haberland.it/ - **haberland.it**

Permanent link:
**https://wiki.haberland.it/doku.php?id=projekte.haberland.it:software-engineering:software-tests**

Last update: **2020/05/12 11:45**