Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Seminararbeit

# Ausarbeitung im Seminar Softwareentwicklung in der Wissenschaft


vorgelegt von

Marvin Heuer

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Wirtschaftsinformatik
Matrikelnummer: 6995091

Betreuer: Jannek Squar


Hamburg, 2018-08-08

# Contents

# 1 Einleitung

Das Testen von Programmen gehört mit zu den wichtigsten Aufgaben in der Softwareentwicklung. Obwohl es eigene Experten für das Testen gibt, bleiben Bugs (= kleine Fehler in Programmen) unentdeckt. Es stellt sich die Frage, warum denn diese Bugs unentdeckt bleiben, eben obwohl ein gesamtes Team an Experten daran sitzt. Zudem lastet dem Testen der Duktus des schweren Nachvollziehens an. Dieser soll mittels dieser Arbeit ein wenig gelüftet werden. Als wichtigstes stellt sich jedoch die Frage, wie man vorgehen kann um gut zu testen. Dieser Frage widmet sich diese Arbeit, nicht außer Acht lassend das noch einmal Unterschiede zwischen professionellen Softwareentwicklern und dem Software entwickeln in der Wissenschaft existieren. Im Rahmen dieser Arbeit wird zunächst in die Begrifflichkeiten des Testens eingeführt. Dann werden die vier Phasen des Testens vorgestellt und jeweils mit praktischen Beispielen unterstützt. Abschließend wird noch ein kleines Beispiel aus der Wissenschaft vorgestellt.

# 2 Begriffe

*Zunächst einmal sollte man die Begrifflichkeiten mit denen in der Seminararbeit umgegangen wird sich einmal klar machen. Insbesondere da es viele verschiedene Arten von Tests gibt, die sich in wichtigen Kleinigkeiten unterscheiden.*

## 2.1 Definitionen

Da diese Arbeit vom „Software Testing“ handelt, wird als erstes eine Definition des Begriffs Softwaretest aufgezeigt. Als Softwaretest bezeichnet man die Ausführung eines Systems bzw. deren Komponenten unter bestimmten Bedingungen. Dabei wird der Ablauf überwacht und hinsichtlich einiger Aspekte ausgewertet.[1] Diese allgemeine Definition benutzt das Institute of Electrical and Electronics Engineers (IEEE) in seinem Verständnis von Softwaretests. Whittaker (2000) beschreibt in seinem Paper Softwaretests als den Prozess des Ausführens eines Softwaresystems, bei dem das Ergebnis mit der Spezifizierung abgeglichen wird.[2] Es wird besonders der Begriff der dynamischen Softwaretests zu dem Begriff der statischen Code Reviews abgegrenzt. Code Reviews stellen eine Form der manuellen Überprüfung von Quellcode dar. Dabei kann ein Code Review bei einem Programm auch schon in einem nicht ausführbaren Zustand erfolgen im Gegensatz zu Softwaretests. Insbesondere können Code Reviews innere Qualitätsmerkmale bewerten während Softwaretests lediglich Fehlverhalten und funktionale Probleme aufdecken.[3] Neben dem Softwaretest existieren auch noch Komponententests. Bei diesen werden individuelle Komponenten oder gleich eine ganze Sammlung von Komponenten getestet. Der Testinput ist dabei nur für die Komponenten, die getestet werden, vorgegeben. Es wird dabei häufig mit Debugger getestet.[4] Als eine Art der Weiterentwicklung existieren auch Integrationstests. Der Name verrät schon, dass hierbei der Fokus auf der Kommunikation zwischen den Komponenten liegt. Dabei werden verschiedene Komponenten durch eine Reihe von Einzeltests begutachtet. Vorher sollten dabei die einzelnen Komponenten isoliert fehlerfrei getestet worden sein.[5] Als fünfte Begrifflichkeit betrachten wir die Systemtests. Bei Systemtests geht es um eine Sammlung von Komponenten, die ein Produkt darstellen. Die Eingabedomäne dabei sind alle Komponenten.[6] Bezüglich der Testauswahl gibt es zwei Arten von Tests. Auf der einen Seite die funktionalen Tests und

[1] Vgl. IEEE 2014.
[2] Vgl. Whittaker 2000.
[3] Vgl. Arbeitsbereich Softwaretechnik und -architektur, SoSe 2018.
[4] Vgl. Whittaker 2000.
[5] Vgl. Whittaker 2000.
[6] Vgl. Whittaker 2000.

auf der anderen die strukturellen Tests.[7] Funktionale Tests sind sogenannte „black-box tests“, das heißt die Testauswahl und Testdatenadäquanz basiert alleine und nur auf der Spezifikation. Daher werden diese Tests auch „spezifikationsbasierte Tests“ genannt. Zudem gibt es noch die bereits erwähnten strukturellen Tests. Diese nennt man auch „white-box tests“ oder „code-based tests“. Letzteres vor allem, da die Eingaben bei den Tests auf der Struktur des Quellcodes und der Datenstrukturen basieren.

[7] Vgl. Whittaker 2000.

# 3 Die vier Phasen des Testens

*Zuerst muss die Softwareumgebung modelliert werden. Dies ist schon die erste Phase. In der zweiten Phase wird mit der Auswahl von Testszenarien begonnen. Bei der dritten Phase steht alles im Fokus der Ausführung und des Auswertens der Testszenarien. Als letztes, in der vierten Phase, wird auf das Messen des Testfortschritts eingegangen.*[1] *Diese vier Phasen werden nachfolgend Schritt für Schritt durchgegangen und erläutert.*

## 3.1 Phase 1: Modellieren der Softwareumgebung

Zur Softwareumgebung gehören alle Schnittstellen oder Programme, die die Software umgeben. Dies können eine Menge Fälle sein die man beachten muss. Betrachten wir beispielhaft die Schnittstellen zuerst. Diese müssen dabei nicht nur erkannt und sondern auch simuliert werden, um ein erfolgreiches Testen zu ermöglichen. Es gibt vier Arten von Schnittstellen. Als erstes betrachten wir die menschlichen Schnittstellen, darunter werden alle Eingabemöglichkeiten durch Menschen verstanden, z.B. die Benutzeroberfläche. Darüber hinaus existieren noch die Softwareschnittstellen., bestehend aus u.a. Application Programming Interfaces (APIs). Eine andere Schnittstelle zur Softwareumgebung ist das Dateisystem. Diese Schnittstelle sollte auch entsprechend simuliert werden können. Falls z.B. kein Speicherplatz mehr frei ist und das Programm etwas speichern möchte, muss getestet werden wie die Software damit umgeht. Als vierte und letzte Schnittstelle werden sämtliche Kommunikationsschnittstellen angesehen, also auch alle Kanäle über die das Programm in bestimmten Protokollen kommuniziert, Insbesondere wenn z.B. die Verbindung über einen Kanal abbricht, stellt sich die Frage, wie das Programm die Verbindung wieder aufbaut bzw. ob die Software überhaupt erkennt, dass die Verbindung unterbrochen wurde. Dazu kommen noch sämtliche Interaktionen außerhalb der Software, wie den soeben erwähnten potenziellen Absturz eines Kommunikationspartners. Aber auch der Umgang falls zwei Dateien parallel beschreiben werden durch zwei Programme, falls es dabei zu Widersprüchen kommt. Darüber hinaus lässt sich noch ein „non-repeatable read“ identifizieren, d.h. dass eine Software zweimal eine Lesevorgang durchführt auf dem selben Objekt und dabei jeweils unterschiedliche Ergebnisse erhält.[2] Meistens dann durch eine zweite Transaktion ausgelöst. Dieser Fall muss simuliert werden, um sicher gehen zu können, dass die Software damit umgehen kann. Als Beispiel wird in Whittaker (2000) ein Programm beschrieben, dass die Systemzeit und Datum beschreibt. In dem Programm gibt es verschiedene Eingabemöglichkeiten.

[1] Vgl. Whittaker 2000.
[2] Vgl. Logicalread.com, aufgerufen am 10.08.2018.

Zum einen kann man mit Alt und F4 das Programm beenden. Zum anderen kann man mit der Tabtaste zwischen den Feldern wechseln. (Siehe **??**)

Wenn wir uns zu diesem Beispielprogramm die Umgebung anschauen können wir zunächst ganz logisch überlegen. Es gibt zwei Eingabequellen, wie man gerade eben gesehen hat. Dabei existieren drei verschiedene Arten von Eingaben. Offensichtlich können gültige Werte eingegeben werden, in denen der reguläre Fall ausgeführt wird. Die Korrektheit und das Vertrauen in das Verhalten bei gültigen Werten kann mittels Positivtests bestätigt werden.[3] Daneben gibt es ungültige und unerwartete Werte. Dies sind Werte, die nicht erlaubt sind bzw. bei denen nicht das Programm sich nicht normal weiterverhalten darf, sondern diese Fälle gesondert behandeln muss. Darüber hinaus sollten die Rahmenbedingungen überprüft werden. Das umfasst den Arbeitsspeicher (RAM), andere Programme die laufen etc. Wenn das getan ist, kommt die zweite Phase.

## 3.2 Phase 2: Auswahl von Testszenarien

Das Hauptziel bei der Auswahl von Testszenarien ist es, eine hohe Abdeckung zu erreichen.[4] Im Optimalfall wird jede Codezeile mindestens einmal ausgeführt. Dazu wird versucht jedes extern generiertes Ereignis zu nutzen. Der generelle Fokus sollte dabei auf „input sequences“ liegen. „Input sequences“ sind keine einzelnen Ein-gaben, sondern Folgen von Eingaben. Ein Beispiel dazu; eine Textverarbeitungs-soft¬ware kann schlecht getestet werden, wenn nur ein Buchstabe zurzeit geprüft wird. Erst eine Kette von Zeichen zeigt, ob das Programm etwas taugt für unsere Zwecke. Dieses Beispiel wird auch generell auf Software übertragen und angewandt. Als Kriterien bei der Auswahl von Testszenarien werden auf verschiedene Kriterien geachtet. Whittaker (2000) beschreibt diese als „Execution path test criteria“, als, frei übersetzt, Ausführungspfadtestkriterien. Da dieses Wort im Deutschen sehr unpraktisch und klobig klingt, hat es sich hierzu lande nicht durchgesetzt. Was darunter verstanden wird ist, dass

- jede Anweisung einmal ausgeführt werden muss.
- jede Verzweigung ausgewertet werden muss.
- jede Datenstruktur genutzt werden muss

Ad 1) Eine Anweisung stellt eine in der Syntax einer Programmiersprache formulierte einzelne Vorschrift dar, die im Rahmen der Abarbeitung des Programms auszuführen ist.[5] Jede sollte ausgeführt werden. Damit wird sichergestellt, dass jeder Teil des Programms einmal ausgeführt wurde. Insbesondere verweist der erste Punkt auf Punkt 2. In dem wird noch einmal genauer spezifiziert, dass jede Verzweigung ausgewertet werden muss. Eine Verzweigung legt fest, welcher von zwei (oder mehr) Programmabschnitten, abhängig von einer (oder mehreren) Bedingungen, ausgeführt wird.[6] Es handelt sich

[3] Vgl. Arbeitsbereich Softwaretechnik und -architektur, SoSe 2017.
[4] Vgl. Whittaker 2000.
[5] Vgl. Wikipedia.org, aufgerufen am 02.07.2018.
[6] Vgl. Wikipedia.org, aufgerufen am 02.07.2018.

dabei um eine Kontrollstruktur. Wenn die Kriterien nur den zweiten Punkt umfassten, so könnte eine der Bedingungen als falsch ausgewertet werden und es würde nicht jede Anweisung zwingend ausgeführt werden. Anders herum ist durch die Anforderung, dass jede Anweisung mindestens einmal ausgeführt wird sichergestellt, dass auch jede Verzweigung einmal ausgewertet wurde. Im dritten Punkt wird definiert, dass jede Datenstruktur genutzt werden muss. Als Datenstruktur bezeichnet man das Konstrukt in einem Programm (bzw. Im Speicher [...]), das Daten auf eine gewisse Weise speichert.[7] Nicht nur wird damit sichergegangen, dass jede Datenstruktur gültig definiert wurde, sondern dass sie auch ausführbar ist.

Ein weiteres Konzept, welches in der Wissenschaft und Forschung interessant ist, ist das sogenannte „fault seeding“. Beim „fault seeding“ werden künstlich Fehler in ein Programm eingebaut und daraufgesetzt, dass diese entdeckt werden. Die Idee bzw. der grundlegende Gedankengang dahinter ist, dass somit ähnliche Fehler aufgedeckt werden können bzw. verhindert werden in der Zukunft. Auch kann man so sichergehen, dass jede Zeile einmal ausgeführt und geprüft wurde. Falls dies nicht so wäre, dann hätte der gepflanzte Fehler entdeckt werden müssen.

Dabei wird auf verschiedene Testdomaingrößen geachtet. Am Ende soll damit jeder physischer Input (Eingabe) sowie jede Interfacekontrollstruktur umfasst werden. Zusätzlich kann noch ein „discrimination criterion“ eingebaut werden. Bei einem „discrimination criterion“ geht es ähnlich wie beim „fault seeding“ um ein abstraktes wissenschaftliches Konzept, indem die Testdomaingröße variabel gestaltet wird. Konkret werden immer weiter zufällige Eingaben aus der potenziellen eingabemenge eingefügt in unsere Testdomain, sodass am Ende eine faire Zusammenstellung aller möglichen Eingaben zusammengestellt wurde.

Im Allgemeinen geht es darum, dass typisches ausführbares Verhalten nachgeahmt wird. Dazu werden unter anderem Grenzwerte und Äquivalenzklassen verwendet. Grenzwerte sind dabei die ersten und letzten Werte in einer Äquivalenzklasse. Durch Äquivalenzklassen wird der Eingangsraum einer Methode bzw. hier eines Programms partitioniert. Das heißt: Werte aus einer Äquivalenzklasse verursachen identisches funktionales Verhalten und testen identische spezifizierte Funktionen. Es werden also mit einer verringerten Anzahl von Testfällen die gleiche Testabdeckung erreicht.[8] Zu Beginn dieser Phase wurde geschrieben, dass das Erreichen einer hohen Abdeckung das Hauptziel in Phase 2 ist. Äquivalenzklassen helfen dabei, dieses Ziel zu erreichen.

Das Beispiel, welches zu erst in Phase 1 vorgestellt wurde, wollen wir nun weiter betrachten. In Section 3.2 ist ein Pseudoquellcode des Programms zu sehen. Zunächst sollten die Pfade aufgedeckt werden und dann alle Pfade geprüft werden. In dem Beispielcode hier wären es 28 Testfälle. Da wir 3 Verzweigungen mit jeweils zwei möglichen Szenarien betrachten (entweder Wahr oder Falsch), lässt sich dafür gut eine Wahrheitstabelle aufzeigen (siehe Table 3.1).

Als nächstes stellt sich die Frage, welche Schnittstellen sich finden lassen. Es könnten zum Beispiel erstmal die möglichen Eingaben nach Zeit und Datum aufgeteilt werden.

[7] Vgl. Computerlexikon.com, aufgerufen am 09.08.2018.
[8] Vgl. Arbeitsbereich Softwaretechnik und -architektur, SoSe 2018.

| Mögliche Fälle | While | Case 1 | If 1 | Case 2 | If 2 | Case 3 | If 3 |
|---|---|---|---|---|---|---|---|
| 1 | F | - | - | - | - | - | - |
| 2 | T | T | T | - | - | - | - |
| 3 | T | T | F | - | - | - | - |
| 4 | T | F | - | T | T | - | - |
| 5 | T | F | - | T | F | - | - |
| 6 | T | F | - | F | - | T | T |
| 7 | T | F | - | F | - | T | F |
| 8 | T | F | - | F | - | F | - |

Table 3.1: Wahrheitstabelle

Es gibt am Tag 24 Stunden mit jeweils 60 Minuten. Jede Minute enthält wiederum 60 Sekunden, sodass man auf 86.400 verschiedene zulässige eingaben kommt. Dabei stellen das nur die zulässigen Eingaben dar. Darüber hinaus sollten noch die unzulässigen Eingaben geprüft werden. Auf der Datumsseite wird ein ähnliches Vorgehen betrieben. Es zeigt sich, dass z.B. eine Kombination wie Mitternacht 1999, beides, Zeit und Datum, testen kann. Spannend wäre auch die Frage, wie das Programm mit mehreren Eingaben hintereinander umgeht. Dies könnte man prüfen. Dabei stehen bei der Anzahl der Testfälle nach oben hin keine Grenzen. Es sind unendlich viele Testfälle möglich.

```
1  Input = GetInput()
2  While (Input != Alt-F4) do
3      Case (Input = Time)
4          If ValidHour(Time.Hour) and ValidMin(Time.Minute)
               ↪ and
5              ValidSec(Time.Second) and ValidAP(Time.AmPm)
6          Then
7              UpdateSystemTime(Time)
8          Else
9              DisplayError("Invalid Time.")
10         EndIf
11     Case (Input = Date)
12         If ValidDay(Date.Day) and ValidMonth(Date.Month) and
13             ValidYear(Date.Year)
14         Then
15             UpdateSystemDate(Date)
16         Else
17             DisplayError("Invalid Date.")
18         EndIf
19     Case (Input = Tab)
20         If TabLocation = 1
21         Then
22             MoveCursor(2)
```

```
23              TabLocation = 2
24          Else
25              MoveCursor(1)
26              TabLocation = 1
27          EndIf
28      EndCase
29      Input = GetInput()
30  Enddo
```

## 3.3 Phase 3: Ausführen und Auswerten der Testszenarien

In der dritten Phase geht es um das Ausführen und Auswerten der Testszenarien. Zunächst wird ein kleiner Exkurs zum Thema Vollautomatisierung gehalten. Dann geht es über in die Szenario-Auswertung sowie in die Ansätze zum Auswerten. Bei der Vollautomatisierung muss jede Eingabequelle und Ausgabe der gesamten Simulationsumgebung simuliert werden. Dazu werden auch häufig Daten-sammlungs¬methoden in den Code eingefügt. Die Szenario-Auswertung ist einfach zu sagen, aber leider schwerer umzusetzen, da man dabei weniger automatisieren kann. Bei einer Auswertung kommt es darauf an, die Ausgabe des Programms (aus den Testszenarios) mit der erwarteten Ausgabe zu vergleichen. Zum Auswerten gibt es zwei Ansätze in der Wissenschaft, zum einen den Formalismus. Dabei werden die Spezifikationen formalisiert aufgeschrieben und aus den Formalien abgeleitet. Formale Spezifikation erleichtern dabei ungemein den Prozess. Die andere Variante der eingebettete Testcode. Damit kann schon gemeint sein, dass das Programm uns Methoden bietet um an die internen Datenstrukturen heranzukommen. Es kann aber auch selbsttestende Programme umfassen. Diese können dabei Methoden umfassen, die sich gegenseitig kontrollieren oder auch Umkehrmethoden. Eine Art von Tests, die im Begriffsteil nicht vorgestellt wurde, ist das Regressionstesten. Dabei geht es um die Frage, wieviel „Retesting“ nötig ist. Man nehme an, dass ein Problem in einer Software behoben wird durch einen Fix. Dann gibt es vier Möglichkeiten für diesen Fix. Der Fix kann nur das gemeldete Problem lösen. Der Fix könnte aber auch das gemeldete Problem nicht lösen. Vielleicht löst der Fix auch das Problem, macht dabei aber etwas anderes „kaputt“. Das „Worst-Case-Szenario“ wäre, dass der Fix das gemeldete Problem nicht lösen kann, und dabei noch etwas anderes „kaputt“ macht. „Kaputt“ heißt in diesem Kontext, dass die Software nicht so funktioniert, wie sie sollte. Unter einem Regressionstest versteht man in der Softwaretechnik die Wiederholung von Testfällen, um sicherzustellen, dass Modifikationen in bereits getesteten Teilen der Software keine neuen Fehler („Regressionen“) verursachen. Solche Modifikationen entstehen regelmäßig z. B. aufgrund der Pflege, Änderung und Korrektur von Software.[9] Um es noch deutlicher zu formulieren: Es geht um die Bestimmung, ob neue Probleme als Ergebnis von

[9] Vgl. Wikipedia.org, aufgerufen am 09.08.2018.

Änderungen in der Software auftauchen.

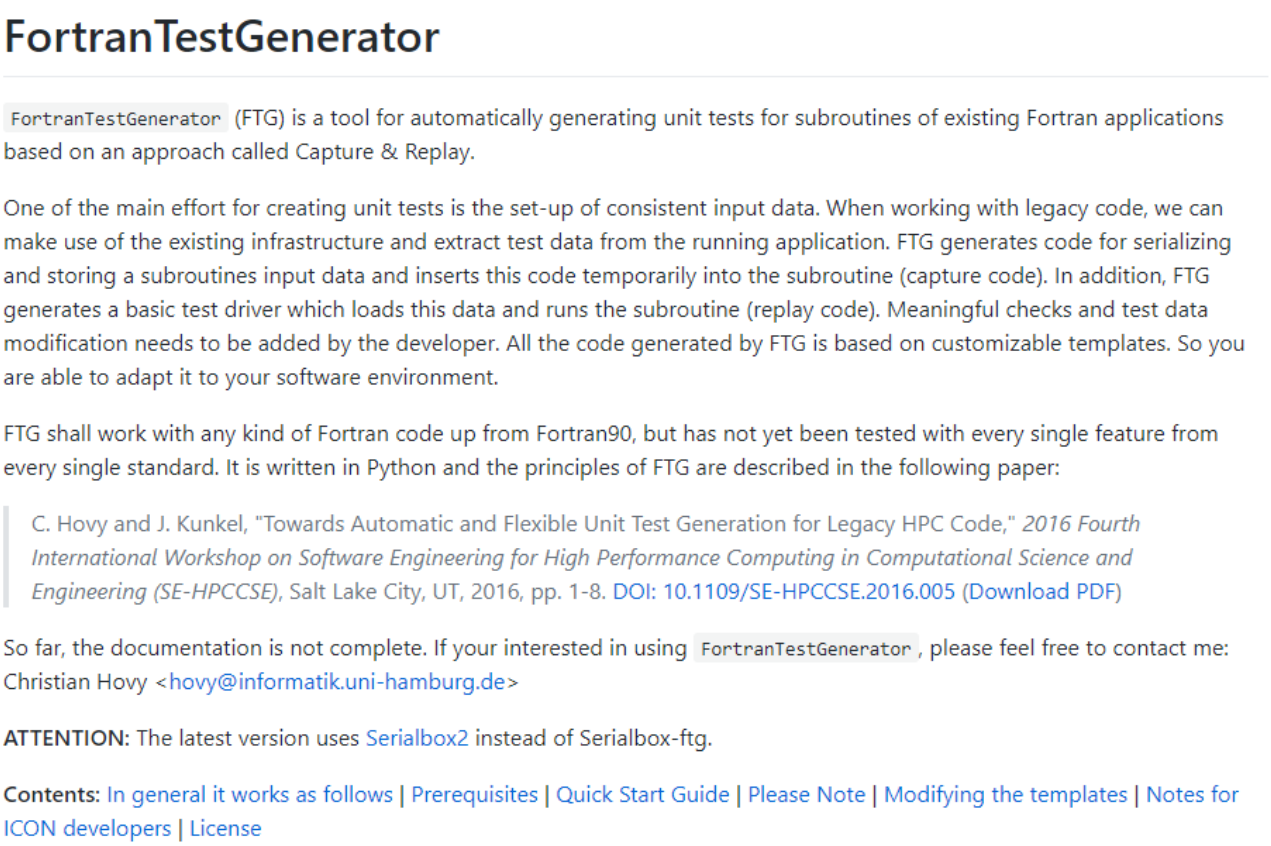
## 3.4 Phase 4: Testfortschritt messen

Diese Phase ist schwer zu erklären. Hier geht es um Testfortschritt und wie man ihn messen kann. Offensichtlich ist der Fortschritt des Testens nicht in Zahlen zu messen bzw. nur sehr schwer zu messen. Dies ist gerade in anderen quantitativen wissenschaftlichen Fachdisziplinen schwer zu vermitteln, dass man hier keine konkrete Zahl angeben kann. Es bietet sich eher in den Kategorien zu denken, ob die Tests strukturell und funktional vollständig sind. Unter strukturelle Vollständigkeit stellen sich die Fragen, ob klassische Programmierfehler existieren[10] oder ob der komplette Quellcode ausgeführt wurde.[11] Eine weitere wichtige Frage zu strukturellen Vollständigkeit ist die Frage, ob sämtliche interne Daten initialisiert und benutzt wurden.[12] Wenn das Konstrukt der „seeded errors“ genutzt wird, so müssen auch diese überprüft werden.[13] Bei der funktionalen Vollständigkeit stellt sich die Frage, ob sämtliches Verhalten abgedeckt wurde. Also zum Beispiel, ob alle Eingaben getestet wurden[14] oder ob alle Ausführungspfade durch Tests abgedeckt sind.[15] Darüber hinaus ist es wichtig, dass durch die Eingaben der gesamte Zustandsraum entdeckt wurde[16] und dass alle erwarteten Szenarien ausgeführt werden.[17] Eine Möglichkeit die Komplexität des Testens eines Programms zu bestimmen könnte die „Testability“ sein.[18] Dabei ist nicht der Begriff nach Popper gemeint, sondern der Grad der Testbarkeit einer Software. Dieser Begriff wurde im Informatikkontext unter anderem von Voas (1992) geprägt und stellt einen interessanten Ansatz dar zur Vergleichbarkeit des Testens bei verschiedenen Programmen. Darüber hinaus wurde im Vortrag über Zuverlässigkeitsmodelle gesprochen. Zuverlässigkeitsmodelle sind mathematische Modelle von Testszenarios. Diese sollen das Verhalten von Software voraussagen können. Diese setzen auch ein sogenanntes „operational profile“ voraus. Als „operational profile“ wird eine Beschreibung verstanden, die spezifiziert wie die Benutzer erwartet werden ihre Eingaben zu machen. Diese Zuverlässigkeitsmodelle berechnen die Wahrscheinlichkeit von Ereignissen, wobei sich hier die Frage stellt, ob diese Modelle überhaupt wirklich signifikant gebaut werden können.

[10] Vgl. Beizer 1990.
[11] Vgl. Myers 1976.
[12] Vgl. Rapps/Weyuker 1985.
[13] Vgl. Myers 1976.
[14] Vgl. Myers 1976.
[15] Vgl. Goodenough/Gerhart 1975.
[16] Vgl. Whittaker/Thomason 1994.
[17] Vgl. Musa 1996.
[18] Vgl. Voas 1992.

# 4 Beispiel aus der Wissenschaft

Als Beispiel aus der Wissenschaft kann man den FortranTestGenerator betrachten. Dieser wurde in Python geschrieben und wird unter anderem von Christian Hovy aus der Arbeitsgruppe Wissenschaftliches Rechnen entwickelt. Dieser Testgenerator erstellt automatisch Komponententests für Subroutinen von existierenden Fortran-Anwendungen.[1] Der ganze Generator basiert auf dem „Capture & Replay"-Ansatz. (siehe Figure 4.1)



## FortranTestGenerator

`FortranTestGenerator` (FTG) is a tool for automatically generating unit tests for subroutines of existing Fortran applications based on an approach called Capture & Replay.

One of the main effort for creating unit tests is the set-up of consistent input data. When working with legacy code, we can make use of the existing infrastructure and extract test data from the running application. FTG generates code for serializing and storing a subroutines input data and inserts this code temporarily into the subroutine (capture code). In addition, FTG generates a basic test driver which loads this data and runs the subroutine (replay code). Meaningful checks and test data modification needs to be added by the developer. All the code generated by FTG is based on customizable templates. So you are able to adapt it to your software environment.

FTG shall work with any kind of Fortran code up from Fortran90, but has not yet been tested with every single feature from every single standard. It is written in Python and the principles of FTG are described in the following paper:

> C. Hovy and J. Kunkel, "Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code," *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*, Salt Lake City, UT, 2016, pp. 1-8. DOI: 10.1109/SE-HPCCSE.2016.005 (Download PDF)

So far, the documentation is not complete. If your interested in using `FortranTestGenerator`, please feel free to contact me: Christian Hovy <hovy@informatik.uni-hamburg.de>

**ATTENTION:** The latest version uses Serialbox2 instead of Serialbox-ftg.

**Contents:** In general it works as follows | Prerequisites | Quick Start Guide | Please Note | Modifying the templates | Notes for ICON developers | License

Figure 4.1: Bild vom FortranTestGenerator

[1] Vgl. FortranTestGenerator auf Github, aufgerufen am 15.7.2018.

# Bibliography


[1] Arbeitsbereich Softwaretechnik und -architektur, Skript zur Vorlesung „Softwareentwicklung II", SoSe 2017.

[2] Arbeitsbereich Softwaretechnik und -architektur, Skript zur Vorlesung „Softwaretechnik", SoSe 2018.

[3] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, New York, 1990.

[4] Computerlexikon.com: Datenstruktur, URL: `https://www.computerlexikon.com/was-ist-datenstruktur`. Aufgerufen am 09.08.2018.

[5] Github: FortranTestGenerator, URL: `https://github.com/fortesg/fortrantestgenerator`. Aufgerufen am 15.07.2018.

[6] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Trans. Software Eng., Vol. 2, No. 2, June 1975, pp. 156–173.

[7] IEEE 730-2014 IEEE Standard for Software Quality Assurance Processes, 3.2.

[8] Logicalread.com, SQL Server Concurrency–Non-repeatable Reads. URL: `https://logicalread.com/sql-server-concurrency-nonrepeatable-reads-w01/#.W2zHBtIzaUk`. Aufgerufen am 10.08.2018.

[9] J.D. Musa, "Software Reliability Engineered Testing," Computer, Vol. 29, No. 11, Nov. 1996, pp. 61–68.

[10] G.J. Myers, The Art of Software Testing, John Wiley Sons, New York, 1976.

[11] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Dataflow Information," IEEE Trans. Software Eng., Vol. 11, No. 4, Apr. 1985, pp. 367–375.

[12] J.M. Voas, "PIE: A Dynamic Failure-Based Technique," IEEE Trans. Software Eng., Vol. 18, No. 8, Aug. 1992, pp. 717–727.

[13] J.A. Whittaker and M.G. Thomason, "A Markov Chain Model for Statistical Software Testing," IEEE Trans. Software Eng., Vol. 20, No. 10, Oct. 1994, pp. 812–824.

[14] J.A. Whittaker, "What Is Software Testing? And Why Is It So Hard?", IEEE Software 0740-7459, Vol. 17, No. 1, Jan/Feb 2000.

[15] Wikipedia.org: Anweisung (Programmierung). URL: https://de.wikipedia.org/wiki/Anweisung_(Programmierung). Aufgerufen am 02.07.2018.

[16] Wikipedia.org: Bedingte Anweisung und Verzweigung. URL: https://de.wikipedia.org/wiki/Bedingte_Anweisung_und_Verzweigung. Aufgerufen am 02.07.2018.

[17] Wikipedia.org: Regressionstest. URL: https://en.wikipedia.org/wiki/Regression_testing. Aufgerufen am 09.08.2018.

## Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Wirtschaftsinformatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

| | |
|---|---|
| Ort, Datum | Unterschrift |