Software Tests

Florian Ehmke

21. Januar 2011

Gliederung

Einleitung

Motivation

> Though scientific software is an engine for scientific progress and provides data for critical decisions, the testing of scientific software is often anything but scientific.
>
> Diane Kelly and Rebecca Sanders

Einleitung

Fragen

- Warum überhaupt testen?
- Wie viel testen?
- Welche Tests sollten durchgeführt werden?
- Wer schreibt die Tests und wer führt sie durch?
- Wann findet das Testen statt?

Einleitung

Warum überhaupt testen?

- Fehler in der Software aufdecken
- Vertrauen in die Software erhöhen
- Softwarespezifikation verifizieren
- Benutzeranforderungen validieren

Einleitung

Verifikation / Validation

“Validation stellt sicher, dass man die richtige Software entwickelt. Verifikation stellt sicher, dass man die Software richtig entwickelt.”

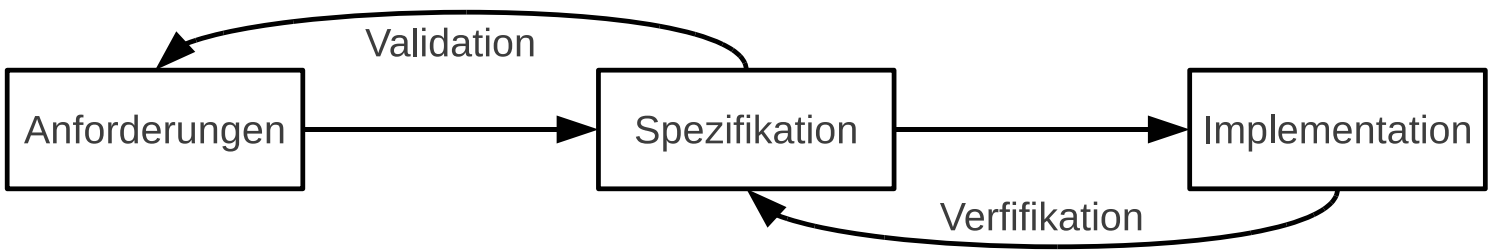

Abbildung: Verifikation und Validation

Einleitung

Definition Test

Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.

Pol, Koomen und Spillner

Einleitung

Test Spezifikation

- Testziel
  - Anforderung XY aus Spezifikation Z
- Vorbedingungen
  - Datenbank X muss Werte enthalten
- Beschreibung
  - Wie ist der Test durchzuführen?
- Erwartetes Ergebnis
  - z.B. bestimmter Datenbankzustand

Umfang der Tests

Ein-/Ausgabedaten Partitionierung

- Alle Eingabedaten zu testen ist typischerweise unmöglich

Listing 1: Summierer

```
1 class Trivial {
2   static int sum(int a, int b) {
3     return a + b;
4   }
5 }
```

Umfang der Tests

Ein-/Ausgabedaten Partitionierung - Äquivalenzklassen

- Werte aus Partitionen und Randwerte verwenden

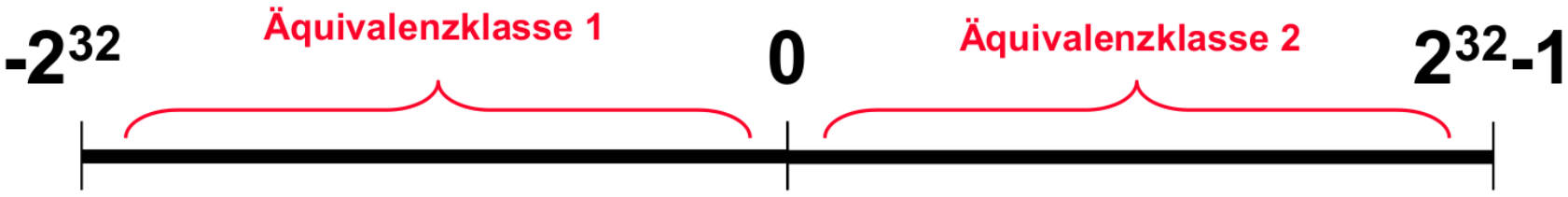

Abbildung: Äquivalenzklassen (Quelle: SE1 Skript)

Umfang der Tests

Codeabdeckung

- Funktionsabdeckung (`foo(x,x)`)
- Zeilenabdeckung (`foo(1,1)`)
- Entscheidungsabdeckung (`foo(1,1), foo(1,0)`)
- Bedingungsabdeckung (`foo(1,1), foo(1,0), foo(0,0)`)

Listing 2: Beispiel Codeabdeckung

```
1  int foo(int x, int y)
2  {
3      int z = 0;
4      if ((x > 0) && (y > 0)) {
5          z = x;
6      }
7      return z;
8  }
```

Umfang der Tests

Error Seeding

Listing 3: Original Code

```
1  if (a && b)
2    c = 1;
3  else
4    c = 0;
```

Listing 4: Mutierter Code

```
1  if (a || b)
2    c = 1;
3  else
4    c = 0;
```

Umfang der Tests

Blackbox / Whitebox

Abbildung: Blackbox / Whitebox

- Blackbox
  - Tests ohne Kenntnisse über die innere Funktionsweise
  - Testet Übereinstimmung mit Spezifikation
- Whitebox
  - Tests mit Kenntnissen über die innere Funktionsweise
  - Gängiges Kriterium ist die Codeabdeckung

Umfang der Tests

Positiv / Negativ

- Positiv: Nur erwartete Eingaben testen
- Negativ: Unerwartete/Ungültige Eingaben testen
- Positivtests erhöhen Vertrauen in Korrektheit
- Negativtests erhöhen Vertrauen in Robustheit

Testebenen

V-Modell

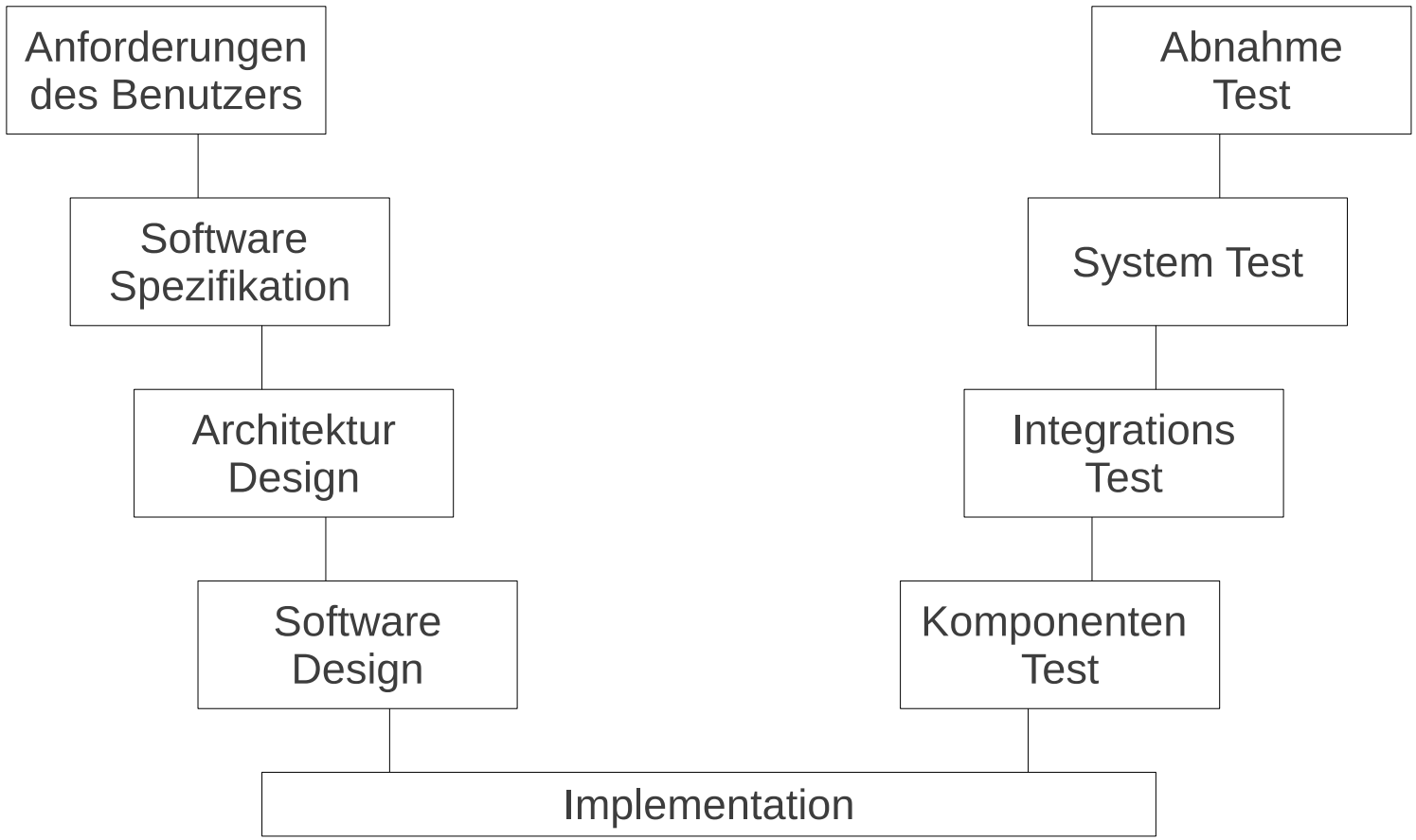

Abbildung: Das V-Modell

Testebenen

V-Modell

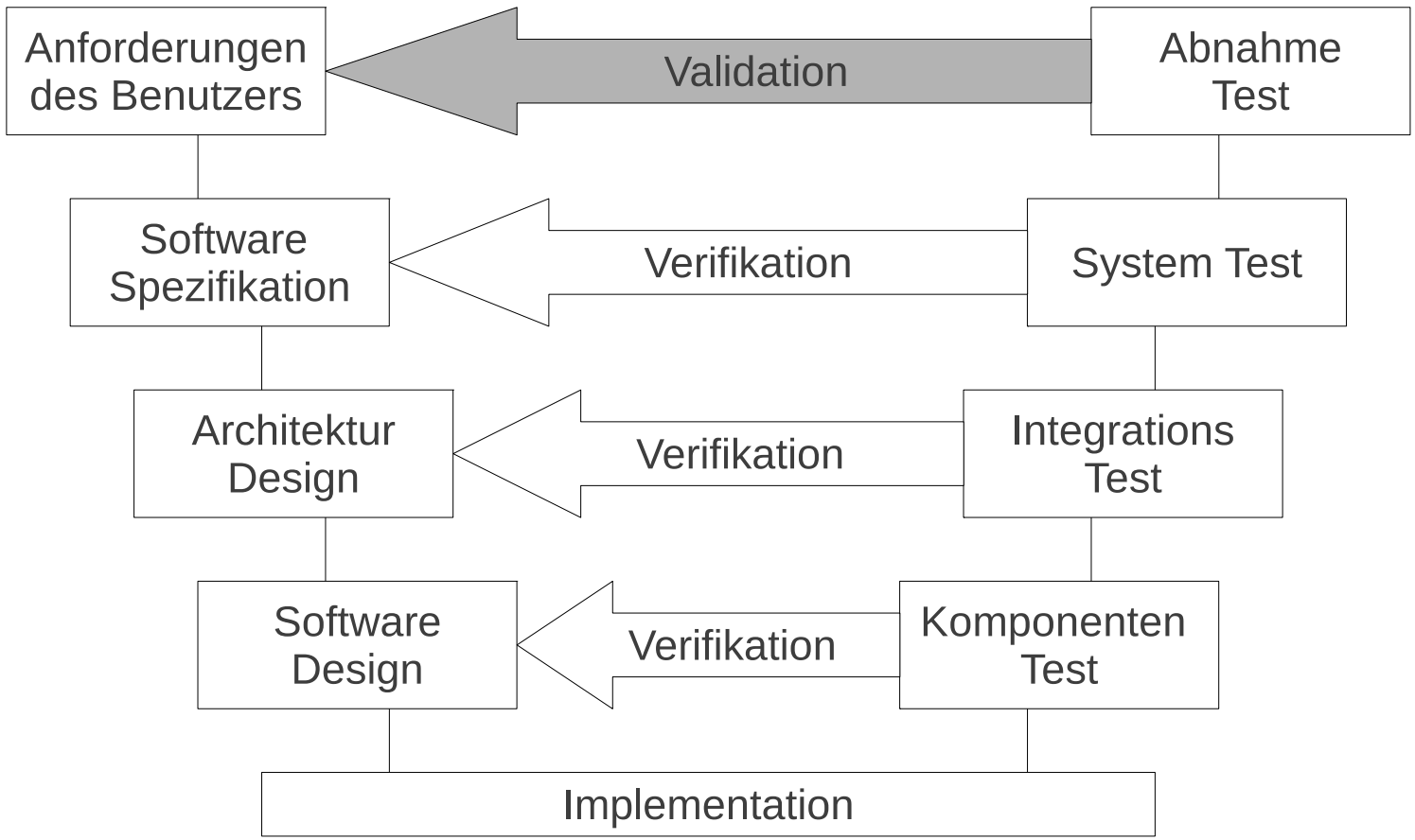

Abbildung: Das V-Modell (Verifikation und Validation)

Testebenen

Komponenten Test

- Testet eine Komponente in Isolation
- Komponente ist kleinster testbarer Teil einer Software
- Im Idealfall unabhängig von anderen Komponenten
- Test einer Funktion / Klasse
- Test First / Testgetriebene Entwicklung

Testebenen

Integrations Test

- Testet Interaktion zwischen Modulen
- Testet Semantik von Schnittstellen
- Basieren auf dem Architektur Design
- Top-Down
  - Anwender-/Betriebssystemnahe Komponenten zuerst
- Bottom-Up
  - Zuletzt aufgerufene Komponenten zuerst

Testebenen

Integrations Test - Beispiel Fehler

Spezifikation nicht Eindeutig bezüglich verwendeter Einheit

Länge	Bezeichnung	Ort
$1.609,344$	Landmeile	USA
1.852	Nautische Meile	international

Abbildung: Meilen - Unterschiede

Resourcen Verwendung geht nicht aus Interface hervor

- Modul A erstellt temporäre Datei `tmp`
- Modul B erstellt temporäre Datei `tmp`

Testebenen

System Test

Abbildung: Abnahmetest

- Testet die integrierten Komponenten
- Basiert auf der Software Spezifikation
- Benötigt keine Einblicke in den Code (Blackbox)
- Testumgebung sollte Produktivumgebung entsprechen

Testebenen

Abnahmetest

Abbildung: Abnahmetest

- Eng verwandt mit dem System Test
- Deckt Fehler in Software Spezifikation auf
- Wird mit dem Benutzer zusammen durchgeführt
- Validation

Testebenen

Weitere Testarten

- Performance Test
  - Testet Geschwindigkeit und Effizienz
- Stress Test
  - Testet Verhalten unter (extremer) Last (z.B Antwortzeiten)
- Wiederinbetriebnahmetests
  - Testet wie gut sich die Software nach Hardwarefehler oder Stromausfall wieder in Betrieb nehmen lässt
- Regressionstest
  - Wiederholung aller / einer Teilmenge aller Testfälle um Nebenwirkungen von Modifikationen auszuschließen

Wer testet / Wie testen

Testumgebung

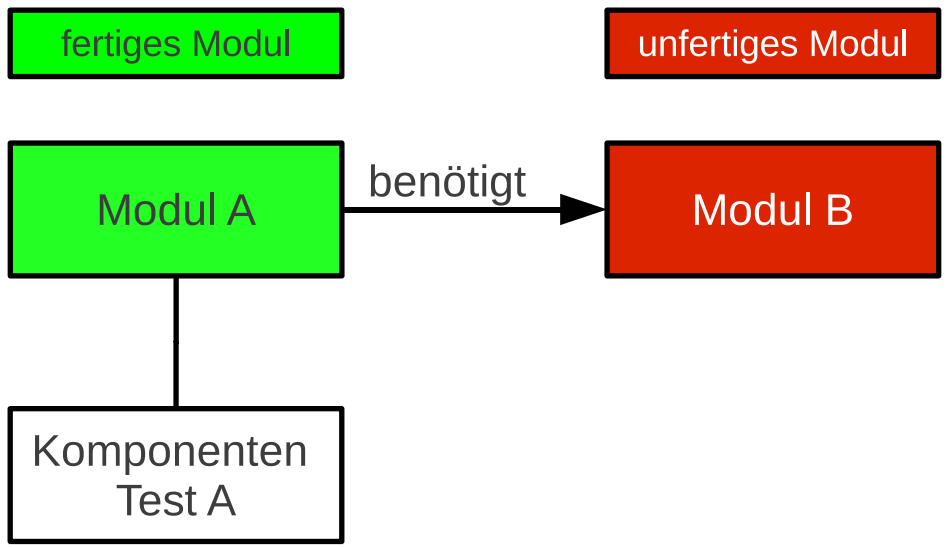

Abbildung: Komponenten Test

Wer testet / Wie testen

Testumgebung

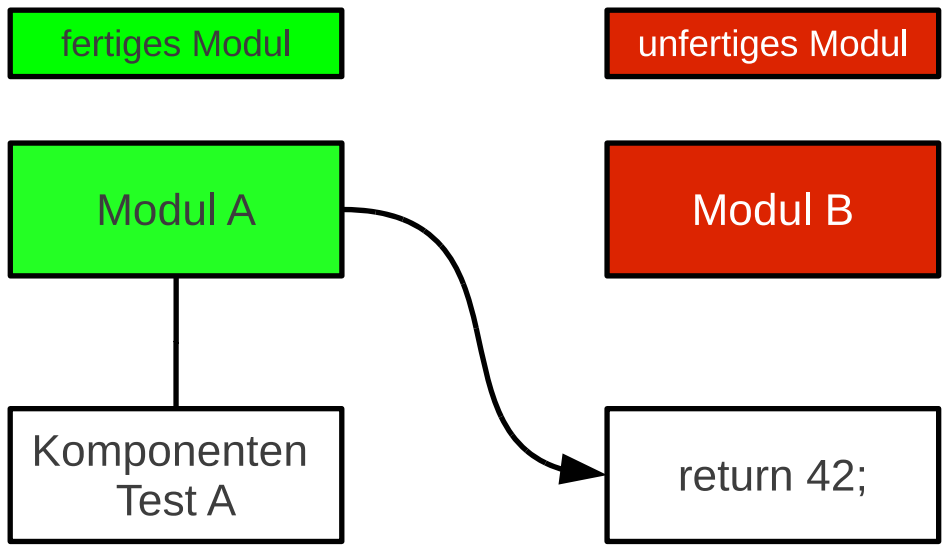

Abbildung: Scaffolding

Wer testet / Wie testen

Testumgebung

- Externe Komponenten müssen simuliert werden
  - Datenbank
  - Dateien
  - Nicht implementierte Komponenten
- Interface für jede externe Komponente
- Mock-Objekte implementierteren dieses Interface

Wer testet / Wie testen

Rollenverteilung

- Optimal: Tester und Entwickler 2 verschiedene Personen
- Sonst oft psychologische Barriere

Wer testet / Wie testen

Automatisieren

- Regelmäßige Wiederholung der Regressionstests
  - Nur durch Automatisierung wirtschaftlich
- Entlastung der Tester von eintönigen Tätigkeiten
- Langfristiger Echtzeitbetrieb kann simuliert werden
- Einmalig hoher Aufwand, danach Pflege/Wartung

Wer testet / Wie testen

Tools

- JUnit (Java)
- CUnit (C)
- Check (C)
- googletest (C/C++)
- FUnit (Fortran)

Wer testet / Wie testen

Tools - CUnit

Listing 5: CUnit Header

```
1 #include <CUnit/CUnit.h>
2 #include <CUnit/Basic.h>
3 #include <CUnit/Automated.h>
4
5 #include "bankFunctions.h"
```

Listing 6: CUnit Test

```
1 void testDeposit(void) {
2   float amount = 5000;
3   deposit(amount);
4   CU_ASSERT_EQUAL(getBalance(), amount + userBalance);
5 }
```

Wer testet / Wie testen

Tools - CUnit

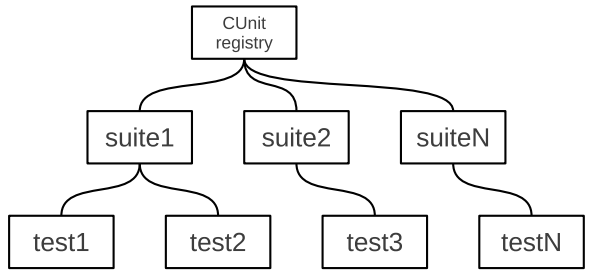

Abbildung: CUnit Hierarchie

Listing 7: CUnit Beispiel

```
1  int main()
2  {
3      CU_pSuite pSuite = NULL;
4      CU_initialize_registry())
5      pSuite = CU_add_suite("Suite_1", setup, NULL);
6      CU_add_test(pSuite, "test Deposit", testDeposit)) {
7      CU_basic_set_mode(CU_BRM_VERBOSE);
8      CU_basic_run_tests();
9  }
```

Wer testet / Wie testen

Tools - CUnit

- `CU_ASSERT`
- `CU_ASSERT_TRUE`
- `CU_ASSERT_FALSE`
- `CU_ASSERT_EQUAL`
- `CU_ASSERT_PTR_EQUAL`
- `CU_ASSERT_PTR_NULL`
- `CU_ASSERT_STRING_EQUAL`
- `CU_ASSERT_DOUBLE_EQUAL`

Wer testet / Wie testen

Tools - CUnit

```
    CUnit - A Unit testing framework for C - Version 2.1-0
    http://cunit.sourceforge.net/


Suite: Suite_1
  Test: test Deposit ... passed
  Test: test Withdraw ... passed
  Test: test set balance ... passed
  Test: test get balance ... FAILED
    1. testBank.c:37  - CU_ASSERT_EQUAL(getBalance(),5000)

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         4       4       3       1
               asserts       4       4       3       1
```

Abbildung: CUnit Ausgabe Basic-Mode

Wer testet / Wie testen

Tools - CUnit

CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Total Number of Suites	1
Total Number of Test Cases	4

Listing of All Tests

Suite	Suite_1	Initialize Function?	Yes	Cleanup Function?	No	Test Count	4
Test Cases	test Deposit test Withdraw test set balance test get balance						

File Generated By CUnit v2.1-0 at Mon Jan 17 13:48:08 2011

Abbildung: CUnit Automated listing.xml

CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Running Suite Suite_1

Running test test Deposit ...	Passed
Running test test Withdraw ...	Passed
Running test test set balance ...	Passed
Running test test get balance ...	Failed

File Name	testBank.c	**Line Number**	37
Condition	CU_ASSERT_EQUAL(getBalance(),5000)		

Cumulative Summary for Run

Type	Total	Run	Succeeded	Failed
Suites	1	1	- NA -	0
Test Cases	4	4	3	1
Assertions	4	4	3	1

File Generated By CUnit v2.1-0 at Mon Jan 17 13:48:08 2011

Abbildung: CUnit Automated result.xml

Testen wissenschaftlicher Software

Ist-Zustand

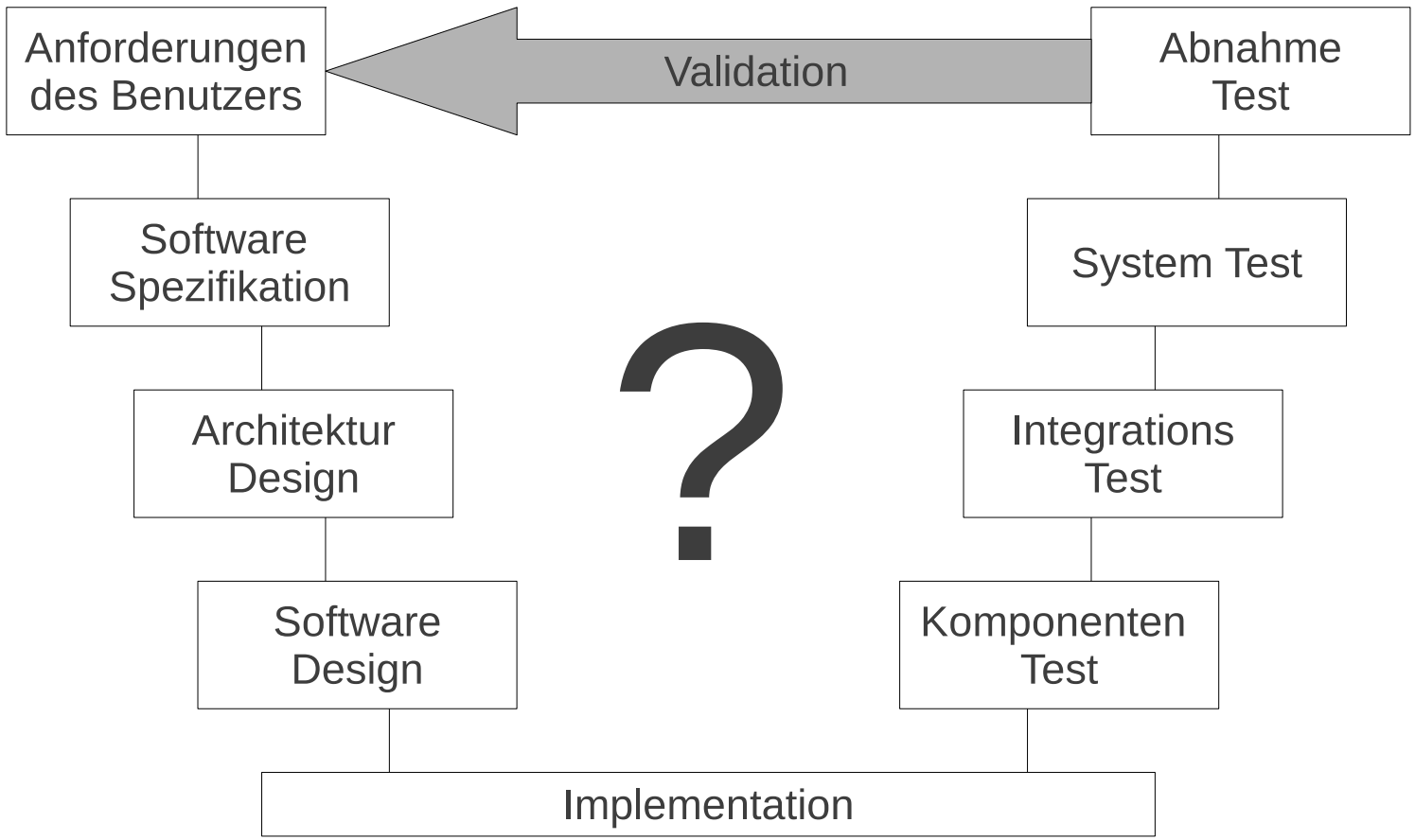

Abbildung: V-Modell Ist-Zustand in der Wissenschaft

Testen wissenschaftlicher Software

Probleme

- Die Software ist lediglich Mittel zum Zweck
- Die Wissenschaftler sind nicht ausgebildet im Testen
- Wissenschaftliche Modelle häufig sehr komplex
  - Erschwert Kooperation zw. Tester u. Wissenschaftler
  - Erfordern hohe Genauigkeit
  - Testdaten oft nicht zuverlässig
- Komplexität steigt zusammen mit der Rechenleistung

Testen wissenschaftlicher Software

Wie viel testen?

- Abhängig von Größe des Projekts
- “So viel wie nötig, so wenig wie möglich”
- Wissenschaftler = Entwickler und Benutzer
  - $\rightarrow$ Abnahmetest nicht relevant

Listing 8: (Relativ) unwichtiger Code

```
1 FILE *fp = fopen("logfile","w");
2 fprintf(fp, "%d:%d = %d\n", hour, minute, value);
3 fclose(fp);
```

Testen wissenschaftlicher Software

Genauigkeit

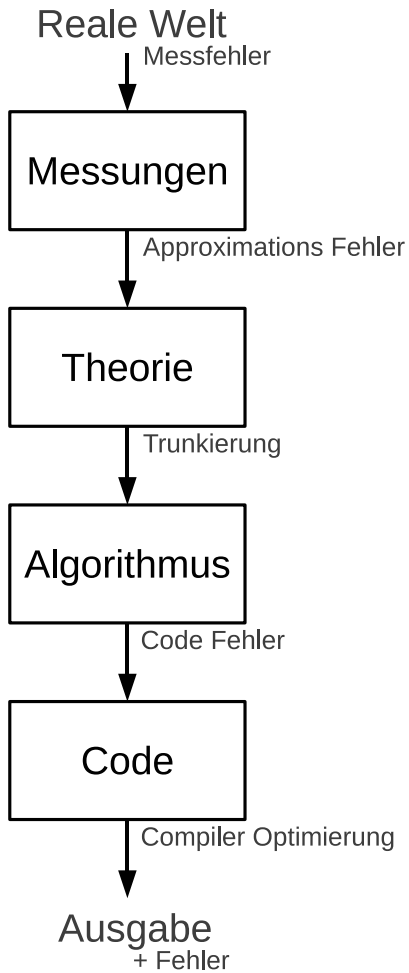

Abbildung: Genauigkeits Fehler

Zusammenfassung

- Softwaretests sind ein umfangreiches Thema
- Gutes Testen verringert Fehlerzahl, erhöht Vertrauen in Software und erleichtert das Erweitern
- Testen wissenschaftlicher Software ist schwer(-er)
- Die Modelle / Themen werden komplexer
- Kooperation zw. Experten in Software und Wissenschaft kostet Zeit die man oft nicht hat

Literatur I


John Paul Elfriede Dustin, Jeff Rashka.
Automated Software Testing - Introduction, Management and Performance.
Addison Wesley Longman, Inc., Massachusetts, 2008.


C. Greenough L.S. Chin, D.J. Worth.
A survey of software testing tools for computational science.
Technical report, CCLRC ePublication Archive [http://epubs.cclrc.ac.uk/oai] (United Kingdom), 2007.


Rebecca Sanders und Diane Kelly.
The challenge of testing scientific software.
In *Proceedings of the 3rd annual Conference of the Association for Software Testing (CAST 2008)*, 2008.

Literatur II


Mauro Pezze und Michal Young.
Software testing and analysis - process, principles and techniques.
John Wiley & Sons, Inc, 2008.