

**BEUTH HOCHSCHULE FÜR TECHNIK BERLIN**  
University of Applied Sciences

Fachbereich VII Elektrotechnik – Mechatronik – Optometrie  
Studiengang Kommunikations- und Informationstechnik

# **Entwicklung eines Testframeworks basierend auf Docker Virtualisierung und Modultests**

**Masterarbeit**

eingereicht von:

**David Stier**

Matrikelnummer 821966  
02.05 2016

Betreuer:

**Prof. Dr. rer. nat. Dipl.-Inform Thomas Scheffler**

**Beuth Hochschule Berlin**  
Fachbereich VII

## **Kurzreferat**

Diese Masterarbeit beschreibt den Entwurf und die Realisierung eines Testframeworks. Dieses Testframework kann automatisiert eingelesene C-Quelltexte mittels Unit Tests und anderen Testverfahren auf die korrekte Funktionalität prüfen. Dafür werden die eingelesenen Quelltexte in einem Docker Container erstellt und mit vorgegebenen Unit Tests getestet. Ebenso können auch dynamische Tests mit Valgrind durchgeführt werden.

Im Verlauf dieser Arbeit wurde eine Laufzeitumgebung erstellt, die zwischen dem Unit Test Framework und dem zu prüfenden Programm interagiert. Dies ermöglicht es, das gesamte Programm als eine Einheit anzusehen und das zu prüfende Programm gegebenenfalls zu beenden.

## **Abstract**

This master thesis describes the design and implementation of a test framework. This test framework can test C-codes automatically. To perform this it uses Docker containers and unit tests to validate the functionality of the C-codes. Also this framework can perform dynamic tests with the Valgrind testing suite.

A special runtime environment was developed in order to apply the framework. The runtime interface interacts between the the program and the unit test framework. Additionally the runtime can stop a test program that crashes.

## **Danksagung**

Hiermit bedanke ich mich bei meinem Betreuer Professor Dr. rer. nat Thomas Scheffler dafür, dass er mir dieses Thema für meine Masterarbeit vorgeschlagen hat und mich bei der Realisierung der Arbeit unterstützte.

## **Erklärung**

Ich versichere, dass die vorliegende Masterarbeit selbstständig und unter Verwendung der angegebenen Quellen angefertigt wurde.

Berlin, der 02.05.2016

---

David Stier

# Inhaltsverzeichnis

1	Einleitung.....	1
1.1	Aufgabenbeschreibung.....	1
1.2	Herangehensweise.....	1
2	Grundlagen.....	2
2.1	Software Testing.....	2
2.1.1	Statisches Software Testing.....	3
2.1.1.1	Software Reviews.....	3
2.1.1.2	Statische Quellcodeanalyse.....	3
2.1.2	Dynamisches Software Testing.....	4
2.1.2.1	Whitebox Testing.....	4
2.1.2.2	Blackbox Testing.....	5
2.2	Das Unit Testsystem.....	6
2.2.1	Die xUnit Architektur.....	7
2.2.1.1	Die Klasse TestCase.....	8
2.2.1.2	Die Klasse TestRunner.....	9
2.2.1.3	Die Klasse TestFixture.....	11
2.2.1.4	Die Klasse TestSuite.....	12
2.2.1.5	Die Klasse TestResult.....	13
2.2.1.6	Beispiel.....	14
2.2.2	Das Google Test Framework.....	18
2.2.2.1	Google Test Framework Vergleiche.....	18
2.2.2.2	Erstellung von TestCases und TestFixtures.....	22
2.2.2.3	Ausführung der Test mit dem Google Test Framework.....	23
2.3	Die Valgrind Tool Suite.....	25
2.4	Docker.....	27
2.4.1	Einführung.....	27
2.4.2	Die Docker Architektur.....	28
2.4.2.1	Der Docker Client.....	29
2.4.2.2	Docker Daemon.....	30
2.4.2.3	Docker Images.....	30
2.4.2.4	Docker Container.....	30
2.4.3	Das Dockerfile.....	31
2.4.4	Docker Volumes.....	34
2.5	Linux Systemprogrammierung.....	36
2.5.1	Die elementaren E/A Funktionen.....	36
2.5.1.1	Die Low-Level Dateiebene.....	36
2.5.1.2	Die High-Level Dateiebene.....	37
2.5.2	Das Signal System.....	38
2.5.3	Prozesse.....	40
2.5.3.1	Prozesserzeugung.....	40
2.5.3.2	Prozessbeendigung.....	40
2.5.3.3	Prozesszustände.....	41
2.5.3.4	Prozesshierarchie.....	42
2.5.3.5	Threads.....	43
2.5.4	Interprozesskommunikation.....	44
2.5.4.1	Pipes.....	44
2.5.4.2	Benannte Pipes.....	46
2.5.4.3	Shared Memory.....	47
2.5.4.4	Semaphore.....	47
2.5.4.5	Sockets.....	49
2.5.4.6	Das Pseudoterminal.....	49
2.6	Weitere Verwendete Software und Tools.....	51

2.6.1 GCC.....	51
2.6.2 G++.....	51
2.6.3 Strip.....	51
2.6.4 Apache.....	51
2.6.5 Python.....	51
2.7 Fazit.....	52
3 Stand der Technik.....	53
3.1 Ähnliche Projekte.....	53
3.2 Literatur.....	54
4 Das Testframework.....	55
4.1 Ablauf des Testvorgangs.....	55
4.2 Programmablauf des Testframeworks.....	56
4.3 Architektur des Testframeworks.....	58
4.3.1 Die Erstellung des zu testenden Programms.....	59
4.3.2 Die Erstellung und Ausführung der Unit Tests.....	62
4.3.3 Löschen der temporären Dateien.....	65
4.4 Die Auswertung der Unit Tests.....	66
4.5 Die Laufzeitumgebung.....	69
4.5.1 Architektur der Laufzeitumgebung.....	69
4.5.1.1 Die Funktion runtime_setup().....	70
4.5.1.2 Die Funktion runtime_tearardown().....	70
4.5.1.3 Die Funktion readFromProgram().....	70
4.5.1.4 Die Funktion writeToProgram().....	70
4.5.1.5 Die Funktion do_valgrind_check().....	70
4.5.1.6 Synchronisation.....	71
4.5.1.7 Das Pseudoterminal.....	72
4.5.2 Die Aufgaben der Laufzeitumgebung.....	73
4.5.2.1 Weiterleitung der Standardstreams.....	73
4.5.2.2 Überwachung des Aufhängens des Testprogramms.....	74
4.5.2.3 Durchführung von Speichertests.....	75
4.6 Die Webseite.....	76
4.6.1 Aufbau der Webseite.....	76
4.6.2 Datenaustausch zwischen Webseite und Testcontainer.....	77
4.6.2.1 Das Volume testframework.....	78
4.6.2.2 Das Volume tfvolume.....	78
4.6.3 Ablauf eines Tests.....	79
4.6.4 Aufruf des Testcontainers.....	80
4.7 Einschränkungen.....	82
4.8 Open Issues.....	83
5 Zusammenfassung und Ausblick.....	84
5.1 Zusammenfassung der Arbeit.....	84
5.2 Ausblick.....	86
6 Abkürzungsverzeichnis.....	87
7 Abbildungsverzeichnis.....	88
8 Quellcodeverzeichnis.....	89
9 Tabellenverzeichnis.....	90
10 Quellenverzeichnis.....	91
11 Anhang.....	94
11.1 Installationsanleitung.....	94
11.1.1 Erstellung der Komponenten.....	94
11.1.2 Anlegen des Volumes tfvolume.....	95
11.1.3 Erstellen der Images.....	95
11.1.3.1 Erstellen des Testimages.....	95
11.1.3.2 Erstellen des Webserver Images.....	95
11.1.3.3 Einfügen von Unit Tests.....	96

11.1.3.4 Starten des Webservers.....	96
11.1.3.5 Anmerkungen.....	97
11.2 Erstellung von Unit Tests und Verifikation.....	98
11.2.1 Unit Tests und Ausgaben für Aufgabe 1.....	99
11.2.1.1 Lösung und Ausgabe für Student A.....	100
11.2.1.2 Lösung und Ausgabe für Student B.....	101
11.2.2 Unit Tests und Ausgaben für Aufgabe 2.....	102
11.2.2.1 Lösung und Ausgabe für Student A.....	103
11.2.2.2 Lösung und Ausgabe für Student B.....	104
11.2.3 Unit Tests und Ausgaben für Aufgabe 3.....	106
11.2.3.1 Lösung und Ausgabe für Student A.....	108
11.2.3.2 Lösung und Ausgabe für Student B.....	111
11.3 Anmerkungen.....	113

# 1 Einleitung

## 1.1 Aufgabenbeschreibung

Im Rahmen dieser Masterarbeit wird ein Testsystem basierend auf Docker Virtualisierung und Unit Tests entwickelt. Dieses Testframework soll in Lehrveranstaltungen dazu dienen, dass Studierende ihre Laborübungen automatisiert testen können. Das Framework soll dem Studenten eine Rückmeldung geben. Dafür werden die Quelltexte der Studierenden auf einem Server getestet. Zum Testen wird ein ausführbares Programm aus diesen Quelltexten erstellt. Anschließend kann das zu prüfende Programm mit Unit Tests getestet werden. Ebenso kann ein dynamischer Speichertest mit Valgrind durchgeführt werden. Die einzelnen Tests laufen in einem Docker Container ab. Somit wird sichergestellt, dass jeder Student die gleiche Umgebung erhält.

Ebenso soll dieses Testframework der Lehrkraft eine einfache Möglichkeit zur Kontrolle der einzelnen Programmieraufgaben der Studenten bieten.

## 1.2 Herangehensweise

Das Grundlagenkapitel behandelt die xUnit Architektur mit deren Verwendung an CPPUnit und dem Google Test Framework. Ebenso wird das Docker System erläutert. Zuletzt erfolgt eine kurze Erläuterung der Linux Systemprogrammierung und der Interprozesskommunikation.

Im Hauptteil wird dann die Erstellung des Testframeworks erläutert. Es wird beschrieben, wie die Unit Tests über die Laufzeitumgebung das zu prüfende Programm testen können. Ebenso wird erläutert wie die einzelnen Module funktionieren. Es wird dargestellt, wie zur Laufzeit das zu prüfende Programm und das Unit Test Programm erstellt werden.

Im Anhang wird die Bedienung des Testframeworks erläutert. Ebenso befindet sich im Anhang die Verifikation des Testframeworks.

## 2 Grundlagen

### 2.1 Software Testing

Software wird heutzutage in vielen Bereichen des Lebens und der Industrie eingesetzt. Um die Funktionalität von Software sicherzustellen existieren verschiedene Testmöglichkeiten, um die Verifikation der zu prüfenden Anforderungen zu ermöglichen. [Spillner 2014][Schäuffele 2013]

Software Testverfahren lassen sich in 2 Gruppen aufteilen

- **Statisches Software Testing**  
Statische Softwaretests werden ohne der Ausführung des zu testenden Programms ausgeführt. Sie werden ausschließlich auf Basis des Quelltextes der Software ausgeführt.
- **Dynamisches Software Testing**  
Dynamische Software Tests werden während der Ausführung des zu testenden Programms durchgeführt. Dazu gehören die Blackbox und die Whitebox Testverfahren.



## 2.1.1 Statisches Software Testing

Das statische Softwaretesting teilt sich in 2 Bereiche auf. Dies sind die Software Reviews und die statische Quellcodeanalyse. [Schäuffele 2013]

### 2.1.1.1 Software Reviews

Bei einem Software Review wird der Quelltext der zu prüfenden Software von den Entwicklern und Gutachtern geprüft. Es handelt sich hierbei um eine Durchsicht des Quellcodes. Damit lässt sich prüfen, ob Funktionen entsprechend der Spezifikation implementiert sind. Ebenso wird bei einem Software Review auf redundanten und obsoleten Quelltext geprüft. Zudem kann auf die Vollständigkeit der Dokumentation geprüft werden.

### 2.1.1.2 Statische Quellcodeanalyse

Ziel der statischen Quellcodeanalyse ist es, Fehler im Quelltext zu finden. Hierfür stehen diverse Werkzeuge zur Verfügung.

Dies sind unter anderem

- Lint  
Erstes Tool für statische Tests, wird im Projekt Splint weitergeführt, welches auch sicherheitskritische Fehler in C Programmcodes findet. [Evans 2002]
- PVS-Studio  
Ein kommerzielles Tool für die statische Quellcodeanalyse, es unterstützt die Programmiersprachen C/C++ und C#. [Viva 64 2016]
- Coverty Scan  
Ebenfalls ein kommerzielles Tool für die statische Quellcodeanalyse. [Coverty 2016]

Moderne Compiler führen heutzutage selbstständig eine statische Quellcodeanalyse aus. Dabei geben sie Fehlermeldungen und Warnungen aus. Dies hat dazu geführt das viele Tools nicht mehr weitergeführt werden.

## 2.1.2 Dynamisches Software Testing

Beim dynamischen Software Testing wird das zu prüfende Programm in einer Testumgebung ausgeführt. Das Ziel beim dynamischen Software Testing ist es, Fehler in den Programmfunktionen, sowie umgebungsabhängige Fehler zu finden. Zu dieser Umgebung gehören unter anderem: [Spillner 2014]

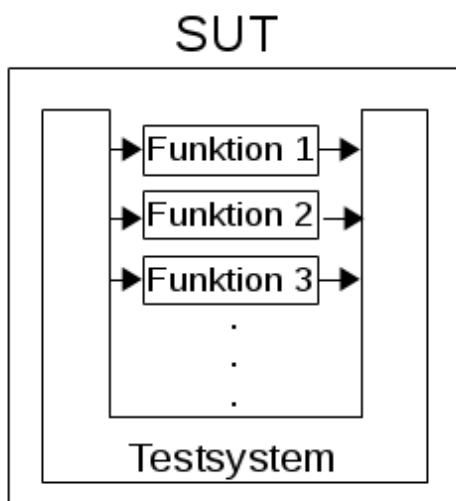
- Eingabeparameter
- Benutzerinteraktion
- Laufzeitumgebung

Es existieren eine Vielzahl an verschiedenen Testarten, die den dynamischen Tests zuzuordnen sind. Die wichtigste Testmethode stellen die Unit Tests dar. Ebenso sind Speichertests den dynamischen Testarten zuzuordnen.

Bei dynamischen Tests werden 2 Arten unterschieden, die Whitebox und die Blackbox Tests. Die Whitebox Tests laufen innerhalb des SUT, während die Blackbox Test das SUT als ein einzelnes Objekt ansehen.

### 2.1.2.1 Whitebox Testing

Das Prinzip des Whitebox Testings ist es, einzelne Funktionen oder Objekte zu testen.



In dieser Abbildung ist das Prinzip des Whitebox Testings dargestellt. Das Testsystem interagiert direkt mit den Einzelnen Funktionen und Objekten innerhalb des SUT<sup>1</sup>. Eine einfache und weit verbreitete Methode des Whitebox Testings ist es, Konsolenausgaben mit aktuellen Werten während der Entwicklungsphase auszugeben. Es werden jedoch auch Test Frameworks dafür verwendet. Die meistbenutzten Frameworks sind die Unit Test Frameworks. [Spillner 2014]

Abbildung 1: Visualisierung des Whitebox Testing [Spillner 2014]

1 SUT – System Under Test

### 2.1.2.2 Blackbox Testing

Beim Blackbox Testing wird das gesamte SUT als eine Einheit angesehen.[Spillner 2014]

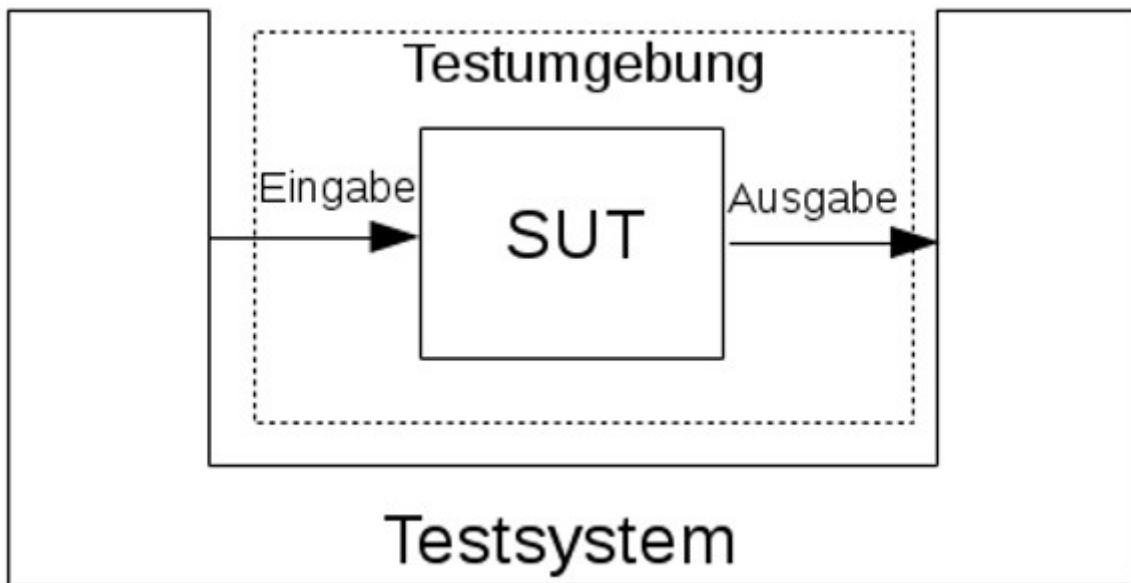


Abbildung 2: Visualisierung des Blackbox Testings [Spillner 2014]

Die obige Abbildung veranschaulicht den Blackbox Test. Dem SUT wird von einem Testsystem die Eingabe vorgegeben. Das Testsystem überprüft dabei die Ausgabe des SUT. Zudem kann das Testsystem eine Testumgebung bereitstellen. Damit können Speicherfehler gefunden werden. Ebenso können Testumgebungen die Kompatibilität zu verschiedenen Bibliotheken testen.

## 2.2 Das Unit Testsystem

Ein Unit Test ist ein Test, der eine einzelne Unit testet. Eine Unit ist hierbei eine Funktion oder eine Klasse. Somit ist ein Unit Test ein Whitebox Test. Da Unit Tests eine weit verbreitete und effiziente Methode für Software Tests darstellen, existieren heutzutage eine Vielzahl von Test Frameworks. Es existieren Unit Test Frameworks für nahezu jede Programmier- und Skriptsprache. Die meisten orientieren sich an der xUnit Architektur. [Hamill 2009]

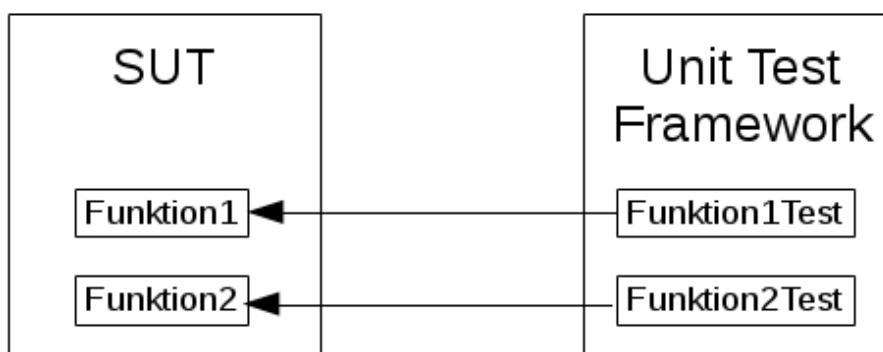


Abbildung 3: Visualisierung des Unit Test Prinzip [Hamill 2009]

In dieser Abbildung ist das Prinzip für Unit Tests veranschaulicht. In diesem Beispiel beinhaltet das SUT die Funktionen „Funktion1“ und „Funktion2“. Das Unit Test Framework stellt hier die Testfunktionen „Funktion1Test“ und „Funktion2Test“ bereit. Diese Funktionen führen einen oder mehrere Tests der jeweiligen Funktion durch.

Es ist grundsätzlich zu empfehlen die Testfunktionen wie die zu testende Funktion mit dem Zusatz „Test“ zu benennen.

## 2.2.1 Die xUnit Architektur

Viele Unit Test Frameworks basieren auf dieser Architektur. Da die unterschiedlichen Testframeworks für verschiedene Programmiersprachen angeboten werden, haben die einzelnen Frameworks weitere Sprachenabhängige Module oder Funktionen. Hier wird auf die allgemeine xUnit Architektur eingegangen und am Beispiel von CPPUNIT demonstriert. Die untere Abbildung stellt die xUnit Architektur mit ihren Grundklassen dar. [Hamill 2009]

Die xUnit Architektur stellt die folgenden Klassen bereit:

- Test  
Abstrakte Klasse die einen Test zur Verfügung stellt
- Testcase  
Dies ist die Grundklasse, von ihr erben alle Unit Tests
- Testrunner  
Diese Klasse ist die Hauptklasse, sie führt die Unit Tests aus
- Testfixture  
Diese Klasse stellt eine Testumgebung bereit, die von einen oder mehreren Tests verwendet werden kann
- Testsuite  
Diese Klasse stellt eine Sammlung von mehreren Unit Tests dar
- TestResult  
Sammelt die Ausgaben aller Tests

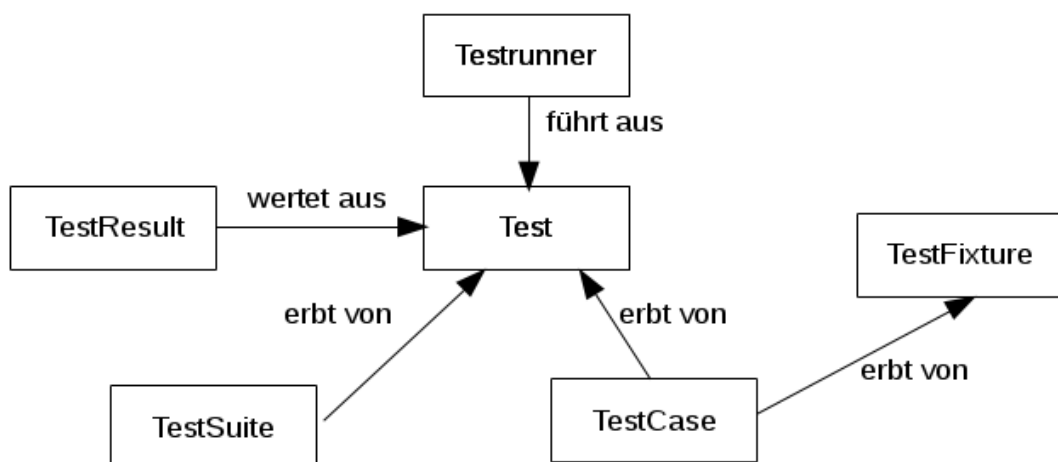


Abbildung 4: schematische Übersicht der xUnit Architektur [Hamill 2009]

### 2.2.1.1 Die Klasse TestCase

Die TestCase Klasse ist die Basisklasse für sämtliche Unit Tests. Sie stellt die Methode runTest() zur Verfügung, die einen oder mehrere Tests durchführt. [Hamill 2009]

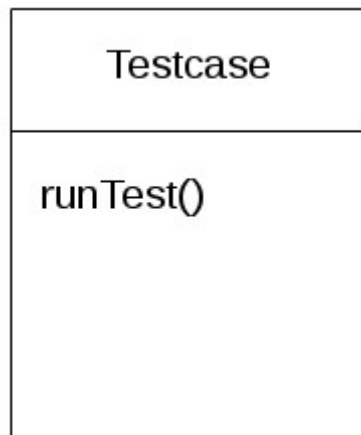


Abbildung 5: Visualisierung  
der Klasse TestCase in  
UML [Hamill 2009]

Die obige Abbildung veranschaulicht die Klasse TestCase in UML<sup>2</sup>. Für einen eigenen Unit Test wird von dieser Klasse geerbt.

```
8 #include <cppunit/TestCase.h>
9 #include "summe.h"
10
11 class SummeTest : public CppUnit::TestCase
12 {
13     public:
14
15     void runTest()
16     {
17         CPPUNIT_ASSERT( summe(1 ,2) == 3 );
18     }
19 };
20
--
```

Listing 1: Implementierung eines TestCases in CPPUnit

<sup>2</sup> UML – Unified Modeling Language

Listing 1 veranschaulicht die Verwendung der TestCase Klasse im CPPUnit Framework. Dies ist die xUnit Implementierung für C++.

In diesem Beispiel wird die Klasse SummeTest erzeugt. SummeTest erbt von der TestCase Klasse des CPPUnit Frameworks. Die Klasse SummeTest überschreibt die Methode runTest(). In dieser Methode befindet sich ein Unit Test. In diesem Beispiel wird die Funktion int Summe(int a, int b) getestet. Diese Funktion gibt die Summe von a und b zurück. Der Unit Test wird mit dem Makro CPPUNIT\_ASSERT realisiert. Damit wird hier überprüft ob der Rückgabewert der Funktion summe(1,2) mit dem erwarteten Wert 3 übereinstimmt. [Sommerlade 2016]

In der Methode runTest() können auch mehrere Unit Tests durchgeführt werden.

### 2.2.1.2 Die Klasse TestRunner

Die Testrunner Klasse führt die Unit Tests aus. Viele Unit Test Frameworks bieten unterschiedliche TestRunner an. Diese unterscheiden sich in der Art der Ausgabe. z.B. Ausgabe auf Konsole oder Speicherung in XML<sup>3</sup> Datei. Hier wird der einfachste TestRunner verwendet, der die Testergebnisse auf der Konsole ausgibt.[Hamill 2009]

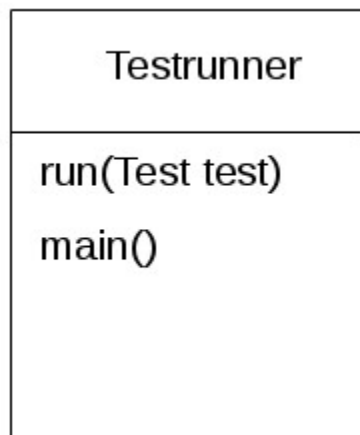


Abbildung 6: Visualisierung  
der Klasse TestRunner in  
UML [Hamill 2009]

---

3 XML – Extensible Markup Language

Die obige Abbildung veranschaulicht die Klasse TestRunner in UML. In xUnit ist lediglich die Methode run() definiert, die die Unit Tests ausführt. Zudem ist die TestRunner Klasse auch die Einstiegsklasse des Testprogramms (main).

```
8 #include <cppunit/ui/text/TestRunner.h>
9 #include "TestSum.h"
10
11 int main(int argc, char* argv[])
12 {
13     SummeTest summeTest;
14     CPPUNIT::TextUi::TestRunner runner;
15     runner.addTest(&summeTest);
16     runner.run();
17 }
18
```

Listing 2: Anwendung des Testrunners in CPPUnit

Dieses Listing veranschaulicht die Anwendung des Text Testrunners von CPPUnit. Hier wird der oben erstellte Testcase „SummeTest“ mit der Methode „addTest()“ aufgerufen. Diese Methode wird nicht von xUnit spezifiziert, sondern von CPPUnit so vorgegeben. Sie wird benötigt um dem Testframework diesen Testcase bekannt zu machen. Mit dem Aufruf der Methode run() werden die eingefügten Tests ausgeführt. [Sommerlade 2016]

```
david@david-ThinkPad-L440:~/workspace/cppunit$ ./summeTest
.
OK (1 tests)
```

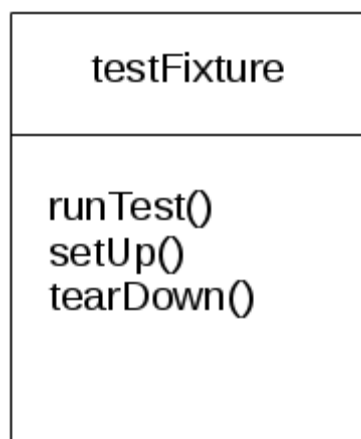
Abbildung 7: Ausgabe des Testrunners

In dieser Abbildung ist die Ausgabe des Tests dargestellt. Es wurde ein Test erfolgreich ausgeführt. Dies war der Testcase aus dem obigen Beispiel. Der Rückgabewert der Funktion Summe(1,2) entspricht dem erwarteten Wert 3.



### 2.2.1.3 Die Klasse TestFixture

Durch Instanzen der Klasse TestCase werden Funktionen direkt aufgerufen. Falls die korrekte Ausführung einer Funktion Variablen oder andere Objekte initialisiert werden müssen, muss dies manuell vor dem Unit Test geschehen. Dafür stellt die xUnit Architektur die Klasse TestFixture bereit. Diese Klasse ermöglicht verschiedene Tests unter der gleichen Umgebung. [Hamill 2009]



*Abbildung 8: Visualisierung der Klasse TestFixture in UML [Hamill 2009]*

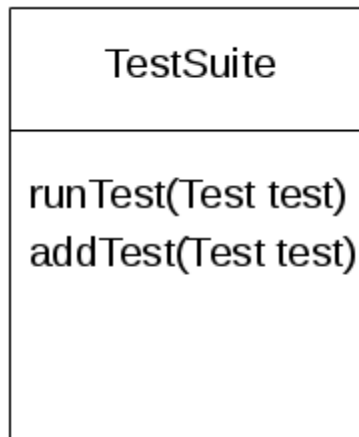
Die obige Abbildung stellt die Klasse TestFixture dar. Zusätzlich zu der Klasse TestCase besitzt sie die Methoden setUp() und tearDown().

Die Methode setUp() wird vor jedem Test aufgerufen. Die Methode tearDown() wird nach jedem Test aufgerufen. Die beiden Methoden werden benutzt um die Umgebung für die zu testenden Objekte zu initialisieren und zu bereinigen. Dieser Mechanismus ist notwendig, damit sämtliche Tests unabhängig von einander ausführbar sind.

In machen Testframeworks wie z.B. CPPUNIT ist testFixture die Elternklasse von TestCase. Dies ist so jedoch aus technischen Gründen implementiert. Der Grund ist, da die Methoden setUp() und tearDown() für die Testausführung intern benötigt werden. In der Testlogik ist Testfixture die Kindklasse von TestCase.

#### 2.2.1.4 Die Klasse TestSuite

Die Klasse TestSuite wird verwendet um mehrere zusammengehörende TestCases und TestFixtures zu bündeln. Dies erleichtert die Erstellung und Ausführung von einem Set von Unit Tests. [Hamill 2009]

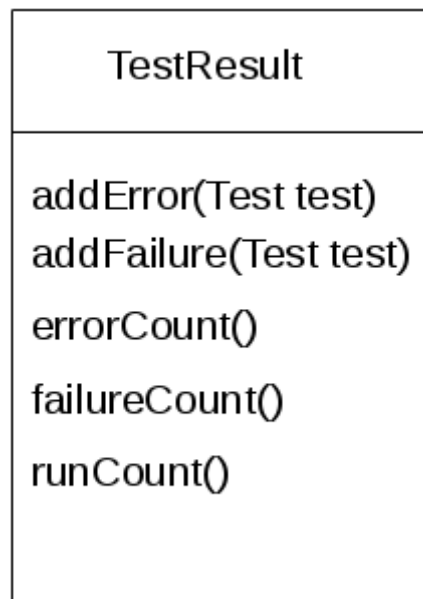


*Abbildung 9: Visualisierung  
der Klasse TestSuite in UML  
[Hamill 2009]*

Die oben stehende Abbildung veranschaulicht die Klasse TestSuite in UML. Die Methode runTest() wird von der abstrakten Klasse Test geerbt und führt die Tests aus. Die Methode addTest() fügt einen TestCase oder ein TestFixture zu der TestSuite hinzu.

### 2.2.1.5 Die Klasse TestResult

Dieses Element stellt das Ergebnis der Tests dar. TestResult beinhaltet Informationen, welche Tests erfolgreich ausgeführt wurden und beinhaltet Fehlerinformationen bei fehlgeschlagenen Tests. [Hamill 2009]



*Abbildung 10: Visualisierung  
der Klasse TestResult in UML  
[Hamill 2009]*

Die obige Abbildung veranschaulicht die Klasse TestResult. Wie jede andere Klasse kann sie je nach verwendetem Framework weitere Methoden beinhalten.

Sie beinhaltet in jedem Fall Methoden, die die Anzahl der ausgeführten Tests (runCount()), der fehlgeschlagenen Tests (failureCount()) sowie der fehlerhaften Tests (errorCount()) zurück liefern.

### 2.2.1.6 Beispiel

Dieses Beispiel veranschaulicht das Zusammenwirken und die Anwendung der Klassen TestFixture, TestSuite und TestResult am Beispiel von CPPUnit. Als zu testende Funktion wird wieder die Funktion `int summe(int a, int b)` verwendet. Für diese Funktion werden in diesem Beispiel mehrere Unit Tests erstellt und ausgeführt.

```
 8 #ifndef TESTSUMMEN_H_
 9 #define TESTSUMMEN_H_
10
11 #include <cppunit/TestFixture.h>
12 #include <cppunit/extensions/HelperMacros.h>
13 #include "summe.h"
14
15 class TestSummen : public CppUnit::TestFixture
16 {
17     CPPUNIT_TEST_SUITE(TestSummen);
18     CPPUNIT_TEST(testSumme);
19     CPPUNIT_TEST(testFehler);
20     CPPUNIT_TEST(testSummen);
21     CPPUNIT_TEST_SUITE_END();
22
23     public:
24     void setUp(void);
25     void tearDown(void);
26
27     protected:
28     void testSumme(void);
29     void testFehler(void);
30     void testSummen(void);
31
32     private:
33     int sum1, sum2, sum3;
34 };
35
36
37
38 #endif /* TESTSUMMEN_H_ */
```

Listing 3: Der Header `TestSummen.h` des Testfixtures

Für diesen Test wird das TestFixture „TestSummen“ angelegt. Der Header dieses Testfixtures ist in dem oberen Listing dargestellt. Die Implementierung dieses Testfixtures ist im folgenden Listing abgebildet.

```
8 #include <cppunit/extensions/TestFactoryRegistry.h>
9 #include "TestSummen.h"
10
11 CPPUNIT_TEST_SUITE_REGISTRATION(TestSummen);
12
13 void TestSummen::setUp(void)
14 {
15     this->sum1 = 0;
16     this->sum2 = 0;
17     this->sum3 = 0;
18 }
19
20 void TestSummen::tearDown()
21 {
22 }
23
24 void TestSummen::testSumme(void)
25 {
26     sum1 = summe(2, 3);
27     CPPUNIT_ASSERT(sum1 == 5);
28 }
29
30 void TestSummen::testFehler(void)
31 {
32     sum2 = summe(4, 5);
33     CPPUNIT_ASSERT(10 == sum2);
34 }
35
36 void TestSummen::testSummen(void)
37 {
38     CPPUNIT_ASSERT_EQUAL(0, sum1);
39     CPPUNIT_ASSERT_EQUAL(0, sum2);
40     sum1 = summe(6,7);
41     CPPUNIT_ASSERT_EQUAL(13, sum1);
42     sum2 = summe(8,9);
43     CPPUNIT_ASSERT_EQUAL(17, sum2);
44     sum3 = summe(sum1, sum2);
45     CPPUNIT_ASSERT_EQUAL(30, sum3);
46 }
47
```

Listing 4: Implementierung des TestFixtures TestSummen

Die Klasse TestSummen erbt von der TestFixture Klasse des CPPUnit Frameworks. Somit werden die Methoden setUp() und tearDown() geerbt und neu definiert. Die Methoden testSumme(), testFehler() und testSummen() stellen Unit Tests dar. Zudem beinhaltet das TestFixture 3 Parameter sum1, sum2 und sum3, die für die Tests verwendet werden.

Weiterhin wird auch eine TestSuite erstellt. Dies geschieht in CPPUnit am einfachsten mithilfe von Makros. Diese sind im Header in den Zeilen 17 -21 zu erkennen. Das Makro CPPUNIT\_TEST\_SUITE(name) legt die TestSuite mit dem Klassennamen „name“ an. Das Makro CPPUNIT\_TEST(test) deklariert die Methode „test“ als Testcase. Das Makro CPPUNIT\_TEST\_SUITE\_END() schließt die Deklaration der Testsuite ab. Das Makro CPPUNIT\_TEST\_SUITE\_REGISTRATION in Listing 4 Zeile 11 registriert die Testsuite. Dies wird für den Aufruf benötigt. [Sommerlade 2016]

In der Implementierung der Methode setUp() werden die 3 Parameter sum1, sum2 und sum3 mit dem Wert 0 initialisiert. Die Methode tearDown wird hier nicht benötigt da nichts freigegeben werden muss und ist deswegen nicht implementiert.

Die Methode testSumme() ruft die zu prüfende Funktion Summe auf und speichert deren Rückgabewert im Parameter sum1. Anschließend wird mit dem Makro CPPUNIT\_ASSERT ein Unit Test durchgeführt. Dabei wird das Ergebnis der Funktion summe() mit dem erwarteten Wert verglichen..

Die Methode testFehler() ist ähnlich wie testSumme(). Jedoch wird hier ein Vergleich mit einem falschen Wert durchgeführt. Dies ist **kein** sinnvoller Unit Test, sondern dient hier nur zur Veranschaulichung der Fehlerausgabe.

Die Methode testSummen() führt erst 2 Vergleiche auf die Parameter sum1 und sum2 aus. Anschließend wird 3 mal die zu prüfende Funktion summe() aufgerufen und auf den erwarteten Wert überprüft.

```
8 #include <cppunit/extensions/TestFactoryRegistry.h>
9 #include <cppunit/ui/text/TestRunner.h>
10 #include "TestSummen.h"
11
12
13 int main(int argc, char*argv[])
14 {
15 |
16     CppUnit::TextUi::TestRunner runner;
17     runner.addTest(CppUnit::TestFactoryRegistry::getRegistry().makeTest());
18     runner.run();
19 }
```

*Listing 5: Implementierung des Testrunners mit TestSuite*

Dieses Listing veranschaulicht nochmals die Implementierung des Testrunners. Jedoch werden hier die Tests mit der Methode makeTest() des CPPUnit Frameworks erzeugt. Dies ist möglich, da die Testsuite mit dem Makro CPPUNIT\_TEST\_SUITE\_REGISTRATION registriert wurde. [Sommerlade 2016]

Es empfiehlt sich diese Methode für die Testausführung zu verwenden, damit am Testrunner keine Veränderungen vorgenommen werden müssen. Dies ist sinnvoll damit nicht versehentlich eine neu erstellte Testsuite vergessen wird. Bei größeren Projekten mit einer Vielzahl an Testsuites kann dies passieren. Vor allem da im Erfolgsfall der Testausführung die Testinformationen gerne übersehen werden.

```
david@david-ThinkPad-L440:~/workspace/cppunit$ ./testsummen
..F.

!!!FAILURES!!!
Test Results:
Run: 3   Failures: 1   Errors: 0

1) test: TestSummen::testFehler (F) line: 33 ../TestSummen.cpp
assertion failed
- Expression: 10 == sum2

david@david-ThinkPad-L440:~/workspace/cppunit$ □
```

*Abbildung 11: Ausgabe des Testrunners mit Testsuite*

Diese Abbildung veranschaulicht die Ausgabe des Testrunners. Das CPPUnit Test Framework signalisiert, dass ein Fehler aufgetreten ist. Dies ist hier der Testcase `testFehler()`. Dieser TestCase hat in diesem Beispiel absichtlich einen falschen Vergleich ausgeführt um hier die Fehlerausgabe darzustellen. Die Ausgabe einer fehlerfreien Ausführung ist bereits in Abbildung 7 dargestellt.

Zu beachten ist auch, dass der Testcase „TestSummen“ erfolgreich war. In diesem Beispiel wurden zuerst 2 Vergleiche auf 0 auf die bereits verwendeten Parameter `sum1` und `sum2` ausgeführt. Damit wird gezeigt, dass die Methode `setUp()` vor jedem Testcase die Variablen neu initialisiert hat. Dies ist der Vorteil von Testfixtures, da dort alle Tests unabhängig voneinander ausgeführt werden können.

Die Klasse `TestResult` ist hier ebenfalls erkenntlich. Sie beinhaltet die 3 Zähler für die ausgeführten Tests, sowie die fehlgeschlagenen und fehlerhaften Tests. Ebenso beinhaltet `TestResult` die Fehlerinformationen wie z.B. Name und Position des fehlgeschlagenen Tests.

## 2.2.2 Das Google Test Framework

Das Google Test (Gtest) Framework ist ein Open Source C/C++ Unit Test Framework von Google. Es basiert auf der xUnit Architektur und ist aktuell in der Version 1.7.0 verfügbar. Im Gegensatz zu anderen Unit Test Frameworks unterscheidet das Gtest Framework in fatal und non-fatal Fehler. Ein fatal Fehler führt zu einem sofortigen Abbruch der Testausführung, während bei einem non-fatal Fehler die Testausführung fortgesetzt wird. Ein weiteres Merkmal des Google Test Framework ist es, dass es einfach zu handhaben ist. Mittels Makros müssen nur die Testcases und Testfixtures erstellt werden. Der Aufruf dieser Tests wird automatisch ausgeführt. Die xUnit Architektur wird durch diese Makros verborgen. Dennoch kann der Anwender mit der bereitgestellten API<sup>4</sup> auf die xUnit Klassen zugreifen und diese anpassen. Weiterhin unterstützt das Google Test Framework parametrisierte Tests (Ausführung des gleichen Tests mit unterschiedlichen Werten) und die automatisierte Testwiederholung (Dauerlaufstest). [Google 2015]

### 2.2.2.1 Google Test Framework Vergleiche

Das Google Test Framework stellt Vergleichmakros zur Verfügung, mit denen die Tests durchgeführt werden. Diese Makros teilen sich in fatal test (ASSERT) und non-fatal test (EXPECT). Falls ein fatal Vergleich fehlschlägt wird der gesamte Testvorgang abgebrochen. Bei einem non-fatal Vergleich wird die Ausführung fortgesetzt. In beiden Fällen wird jedoch eine Fehlermeldung ausgegeben.[Google 2015]

---

4 API – Application Programming Interface



Das Google Test Framework stellt die folgenden Vergleiche bereit. Zusätzlich ist es möglich, eigene Vergleiche zu erstellen.[Google 2015]

Basisvergleiche

Diese Vergleiche prüfen die Bedingung „bedingung“. Auf ihnen basieren sämtliche anderen Vergleiche. Diese Vergleiche sind in der folgenden Tabelle dargestellt:

fatal	non-fatal	prüft
ASSERT_TRUE(bedingung)	EXPECT_TRUE(bedingung)	„bedingung“ ist wahr (bool)
ASSERT_FALSE(bedingung)	EXPECT_FALSE(bedingung)	„bedingung“ ist falsch (bool)

*Tabelle 1: Übersicht über die Googletest Basisvergleiche[Google 2015]*

Binäre Vergleiche

Die binären Vergleiche werden verwendet um Integer Werte zu prüfen. Die folgende Tabelle beinhaltet die dafür vorgegeben Vergleiche

fatal	non-fatal	prüft
ASSERT_EQ(erwartet, aktuell)	EXPECT_EQ(erwartet, aktuell)	„erwartet“ entspricht „aktuell“
ASSERT_NE(wert1, wert2)	EXPECT_NE(wert1, wert2)	„wert1“ und „wert2“ sind verschieden
ASSERT_LT(wert1, wert2)	EXPECT_LT(wert1, wert2)	„wert1“ ist kleiner als „wert2“
ASSERT_LE(wert1, wert2)	EXPECT_LE(wert1, wert2)	„wert1“ ist kleiner oder gleich wie „wert2“
ASSERT_GT(wert1, wert2)	EXPECT_GT(wert1, wert2)	„wert1“ ist größer als „wert2“
ASSERT_GE(wert1, wert2)	EXPECT_GE(wert1, wert2)	„wert1“ ist größer oder gleich wie „wert2“

*Tabelle 2: Übersicht über die Googletest Binärvergleiche[Google 2015]*

### Gleitkomma Vergleiche

Gleitkomma Werte weisen immer eine Abweichung auf, da sie nur als Näherung dargestellt werden können. Deshalb ist ein direkter Vergleich schwer zu realisieren. Das Google Test Framework stellt für die Gleitkomma Vergleiche die Makros in der folgenden Tabelle bereit:

fatal	non-fatal	prüft
ASSERT_FLOAT_EQ(wert1,wert2)	EXPECT_FLOAT_EQ(wert1,wert2)	Die beiden Float Werte sind identisch*
ASSERT_DOUBLE_EQ(wert1, wert2)	EXPECT_DOUBLE_EQ(wert1, wert2)	Die beiden Double Werte sind identisch*
ASSERT_NEAR(wert1,wert2, diff)	EXPECT_NEAR(wert1,wert2, diff)	Die Differenz zwischen den beiden Werten beträgt maximal „diff“

*Tabelle 3: Übersicht über die Googletest Gleitkomma Vergleiche[Google 2015]*

\* identisch auf 4 Nachkommastellen.

### String Vergleiche

Zum Prüfen von C-Strings stellt das Google Test Framework die Vergleiche in dieser Tabelle zur Verfügung:

fatal	non-fatal	prüft
ASSERT_STREQ(str1, str2)	EXPECT_STREQ(str1, str2)	„str1“ und „str2“ sind identisch
ASSERT_STRNE(str1, str2)	EXPECT_STRNE(str1, str2)	„str1“ und „str2“ sind nicht identisch
ASSERT_STRCASEEQ(str1, str2)	EXPECT_STRCASEEQ(str1, str2)	„str1“ und „str2“ sind bis auf Groß- und Kleinschreibung identisch
ASSERT_STRCASENE(str1, str2)	EXPECT_STRCASENE(str1, str2)	„str1“ und „str2“ sind bis auf Groß- und Kleinschreibung nicht identisch

*Tabelle 4: Übersicht über die Googletest Stringvergleiche [Google 2015]*

Ausnahmen Vergleiche

fatal	non-fatal	prüft
ASSERT_THROW(ausdruck, exception)	EXPECT_THROW(ausdruck, exception)	Der Ausdruck „ausdruck“ löst die Ausnahme „exception“ aus
ASSERT_ANY_THROW(ausdruck)	EXPECT_ANY_THROW(ausdruck)	Der Ausdruck „ausdruck“ löst eine Ausnahme aus
ASSERT_NO_THROW(ausdruck)	EXPECT_NO_THROW(ausdruck)	Der Ausdruck „ausdruck“ löst keine Ausnahme aus

*Tabelle 5: Übersicht über die Googletest Ausnahmenvergleiche*

Die obere Tabelle stellt die Testmakros für die Ausnahmen Prüfung dar. Diese können nur in C++ Projekten verwendet werden, da C über keinen Ausnahmenmechanismus verfügt. Unter Windows existieren auch Vergleiche auf HRESULT Ausnahmen. Da dieses Projekt ausschließlich auf Linux basiert, werden diese hier nicht aufgeführt. [Google 2015]

### 2.2.2.2 Erstellung von TestCases und TestFixtures

Das Google Test Framework stellt Makros bereit, mit deren Hilfe die TestCases und TestFixtures erstellt werden können.

```
12 TEST(Rechner, DifferenzTest)
13 {
14     ASSERT_EQ(1, differenz(2,1));
15     ASSERT_EQ(3, differenz(8,5));
16     ASSERT_EQ(20, differenz(100,80));
17 }
```

*Listing 6: Implementierung eines Testcases im Gtest Framework*

Dieses Listing veranschaulicht die Implementierung eines Unit Tests für den TestCase „Rechner“. Der Unit Test hat den Namen „Differenztest“ und führt 3 Vergleiche auf die Funktion differenz(int, int) aus. Für einen Testcase können mehrere Tests mit unterschiedlichen Namen erstellt werden.

```
1 #include "gtest/gtest.h"
2 #include "../runtime/runtime.h"
3
4 class NameTest : public ::testing::Test
5 {
6
7     virtual void SetUp()
8     {
9         runtime_setup(NULL);
10    }
11    virtual void TearDown()
12    {
13        runtime_teardown();
14    }
15 };
16
17 TEST_F(NameTest, askName)
18 {
19     ASSERT_STREQ("Bitte Namen eingeben:\r\n", readFromProgram());
20     writeToProgram("David\n");
21     ASSERT_STREQ("Hallo David\r\n", readFromProgram());
22 }
23
```

*Listing 7: Implementierung eines TestFixtures im Gtest Framework*

In diesem Listing wird das TestFixture „NameTest“ erstellt. Dafür wird von der Gtest Klasse „Test“ geerbt. Dieses Testfixture beinhaltet den Test „askName“. Dieser Test führt 2 Stringvergleiche auf die Funktion readFromProgram() aus. Dazwischen wird die Funktion writeToProgram() aufgerufen. Ebenso werden für dieses Testfixture die virtuellen Methoden SetUp() und TearDown() überschrieben. Diese Methoden rufen hier die Funktionen runtime\_setup() und runtime\_teardown() auf. Diese Funktionen gehören zu der Laufzeitumgebung, die im Verlauf dieser Arbeit erstellt wurde.

Das Makro „TEST\_F“ ordnet den Unit Test einem Testfixture zu. Während das Makro „TEST“ den Unit Test einem TestCase zuordnet. [Google 2015]

### 2.2.2.3 Ausführung der Test mit dem Google Test Framework

Das Google Test Framework stellt eine Standard Main Funktion für den Testdurchlauf bereit. Dies ermöglicht es dem Anwender, dass dieser nur die TestCases und TestFixtures erstellen muss. Es kann jedoch auch eine eigene Main Funktion mit eigenem Testrunner verwendet werden.

```
1 // gtest_main.cpp
2 #include <stdio.h>
3 #include "gtest/gtest.h"
4
5 GTEST_API_ int main(int argc, char **argv) {
6     printf("Running main() from gtest_main.cc\n");
7     testing::InitGoogleTest(&argc, argv);
8     return RUN_ALL_TESTS();
9 }
```

*Listing 8: Darstellung der Standard Main Funktion mit Standard Testrunner des Gtest Frameworks [Google 2015]*

Dieses Listing veranschaulicht die Standard Main Funktion des Google Test Frameworks. Nachdem mit der Methode InitGoogleTest() das Test Framework initialisiert wurde, wird mit dem Makro RUN\_ALL\_TESTS() der Testdurchlauf gestartet. Dabei werden alle TestCases und TestFixtures des Projekts ausgeführt. Dieser Mechanismus verhindert, dass der Anwender versehentlich vergisst, die einzelnen Tests aufzurufen. [Google 2015]

Standardmäßig verwendet das Google Test Framework einen Testrunner mit Konsolenausgabe.

```
[=====] Running 1 test from 1 test case.  
[-----] Global test environment set-up.  
[-----] 1 test from HelloTest  
[  RUN   ] HelloTest.askName  
Bitte Namen eingeben:  
Hallo David  
[   OK   ] HelloTest.askName (603 ms)  
[-----] 1 test from HelloTest (603 ms total)  
  
[-----] Global test environment tear-down  
[=====] 1 test from 1 test case ran. (603 ms total)  
[  PASSED ] 1 test.  
david@david-ThinkPad-L440:~/workspace/UTest$ █
```

*Abbildung 12: Ausgabe des Gtest Testrunners*

Diese Abbildung stellt die Ausgabe des vorherigen TestFixtures dar. Der Testrunner markiert erfolgreiche Tests mit der Farbe Grün. Fehlgeschlagene Tests werden in der Farbe Rot dargestellt. Ebenfalls gibt der Testrunner weitere Informationen wie z.B. die Anzahl der Ausgeführten Tests, sowie die Ausführungszeit an.

## 2.3 Die Valgrind Tool Suite

Die Valgrind Tool Suite ist eine Open Source Toolsammlung für die dynamische Fehleranalyse von Linux Anwendungen. Diese Open Source Toolsammlung liegt aktuell in der Version 3.11.0 und beinhaltet die folgenden Tools: [Valgrind 2016]

### Memcheck

Memcheck wird verwendet um Speicherfehler zu finden. Zu diesen Speicherfehlern gehören:

- Zugriff auf nicht allokierten oder freigegebenen Speicher
- Pufferüberläufe
- fehlerhaftes Freigeben von Speicher
- Speicherlecks (allokiertes aber nicht freigegebener Speicher)

Memcheck ist das wichtigste Tool von Valgrind. [Valgrind 2016]

### Cachegrind

Cachegrind ist ein Cacheprofiler und überwacht die Verwendung der CPU<sup>5</sup>Caches. Dieses Tool wird für die Codeoptimierung verwendet.[Valgrind 2016]

### Callgrind

Callgrind ist eine Erweiterung zu Cachegrind und stellt eine dafür Visualisierung bereit. [Valgrind 2016]

### Massif

Massif ist ein Heap Profiler, dieser überwacht den Heap und gibt Informationen über Programmteile mit vielen Speicherallokationen. [Valgrind 2016]

---

5 CPU – Central Processing Unit

### Helgrind

Helgrind ist ein Tool um Fehler zwischen Race Conditions zwischen mehreren POSIX<sup>6</sup> Threads zu finden. [Valgrind 2016]

### DRD

Dieses Tool dient zur Fehlererkennung in Multithread C/C++ Programmen. [Valgrind 2016]

Für die Testausführung stellt Valgrind eine virtuelle CPU bereit. Das zu prüfende Programm wird auf dieser virtuellen CPU ausgeführt. Da diese virtuelle CPU überwacht werden kann, können die einzelnen Tools sämtliche Speicheraufrufe überwachen und Fehler feststellen. Da das gesamte Programm als eine Einheit angesehen wird, handelt es sich um einen Blackbox Test. In dieser Arbeit wird nur Memcheck verwendet, da die zu prüfenden Programme den Testumfang der restlichen Tools nicht benötigen. [Valgrind 2016]

---

6 POSIX – Portable Operating System Interface



## 2.4 Docker

Docker ist eine Open Source Lösung zur Isolation von Linux Programmen. Docker ist aktuell in Version 1.10.0 verfügbar. [Turnbull 2014][Docker 2016]

### 2.4.1 Einführung

Jede Anwendung verfügt über Abhängigkeiten von anderen Anwendungen oder Diensten. Diese können das Betriebssystem oder dynamische Bibliotheken sein. Ebenfalls können Sicherheitslücken einer Anwendung die Verfügbarkeit eines gesamten Systems mit unterschiedlichen Anwendungen gefährden. Um diese Probleme zu beheben, kann die Anwendung in einer virtuellen Maschine ausgeführt werden. Eine virtuelle Maschine simuliert einen gesamten Computer. Dies verursacht einen hohen Bedarf an Speicher und Rechenleistung. Eine Alternative zu virtuellen Maschinen sind Container. Diese Container teilen sich den Betriebssystemkernel, sind aber dennoch voneinander isoliert. Der Linux Kernel stellt mit LXC<sup>7</sup> dafür ein Verfahren bereit. Auf diesen Linux Containern basiert Docker. Im Gegensatz zu virtuellen Maschinen können mit Containern nur Linux Anwendungen ausgeführt werden, dafür sind Container ressourcensparender als virtuelle Maschinen. [Seidel 2016][Yanar 2016]

---

<sup>7</sup> LXC – Linux Container

## 2.4.2 Die Docker Architektur

Das Docker System beruht auf den folgenden Komponenten: [Turnbull 2014]

- Docker Clients
- Docker Engine (Docker Daemon)
- Docker Containern
- Docker Host

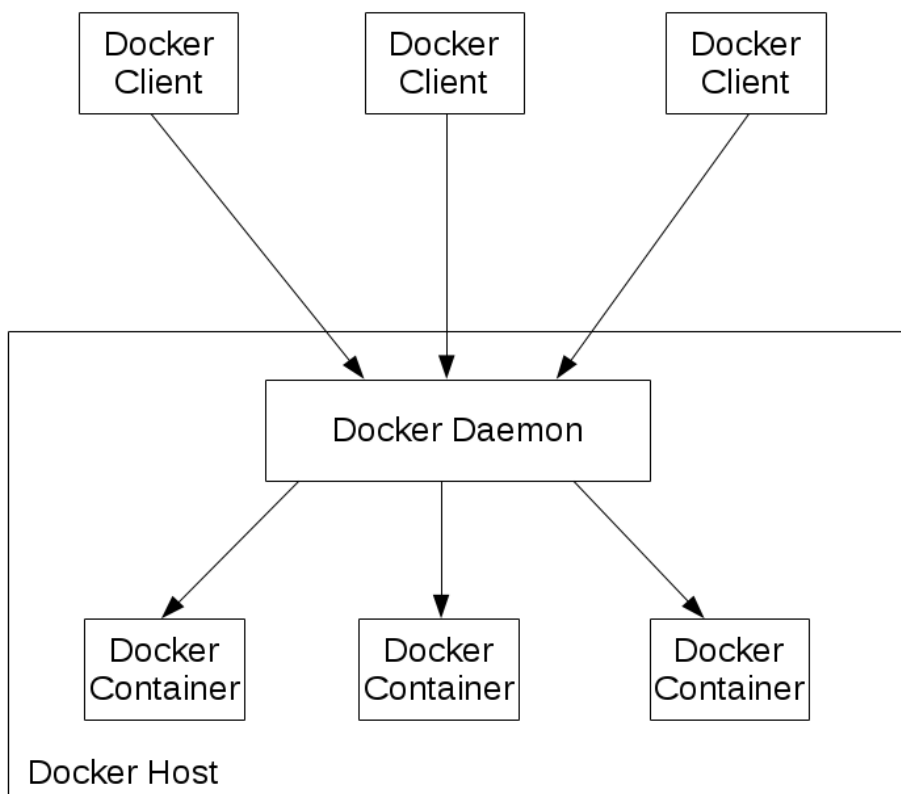


Abbildung 13: Visualisierung der Docker Architektur [Turnbull 2014]

Das Zusammenwirken dieser Komponenten wird durch die obere Abbildung veranschaulicht und im folgenden erläutert.

### 2.4.2.1 Der Docker Client

Der Docker Client stellt die Anwendungsschnittstelle zu Docker dar. Der Docker Client kann auf dem selben Rechner (Docker Host) wie der Docker Daemon ausgeführt werden, er kann aber auch von einem anderen Rechner ausgeführt werden. [Turnbull 2014]

```
david@david-ThinkPad-L440:~$ docker --help
Usage: docker [OPTIONS] COMMAND [arg...]
       docker daemon [ --help | ... ]
       docker [ --help | -v | --version ]

A self-sufficient runtime for containers.

Options:
  --config=~/.docker           Location of client config files
  -D, --debug                  Enable debug mode
  -H, --host=[]               Daemon socket(s) to connect to
  -h, --help                  Print usage
  -l, --log-level=info        Set the logging level
  --tls                        Use TLS; implied by --tlsverify
  --tlscacert=~/.docker/ca.pem Trust certs signed only by this CA
  --tlscert=~/.docker/cert.pem Path to TLS certificate file
  --tlskey=~/.docker/key.pem  Path to TLS key file
  --tlsverify                 Use TLS and verify the remote
  -v, --version               Print version information and quit
```

*Abbildung 14: Aufruf des Docker Clients in der Konsole*

Der Aufruf des Docker Clients wird durch den Befehl „docker“ ausgeführt. Die Syntax lautet: `docker [OPTIONS] COMMAND [arg...]` Dies wird in der oberen Abbildung veranschaulicht. Es stehen eine Vielzahl von Befehlen (Commands) zur Verfügung. Die wichtigsten sind: [Docker 2016]

- **build**  
Dieser Befehl erstellt ein neues Image, z.B. aus einem Dockerfile
- **start**  
Dieser Befehl startet einen gestoppten Container
- **run**  
Dieser Befehl startet einen neuen Container aus einem Image

Docker stellt zudem eine API zur Verfügung, mit deren Hilfe externe Anwendungen Zugriff auf den Docker Daemon erhalten. Wenn eine Anwendung diese API einbindet, wird diese Anwendung ebenfalls zu einem Docker Client. [Docker 2016]

### **2.4.2.2 Docker Daemon**

Der Docker Daemon wird auch Docker Server oder Docker Engine genannt. Der Docker Daemon ist für die gesamte Ausführung verantwortlich. Er verwaltet die Container und leiten Anfragen von den Clients an die Container weiter. [Turnbull 2014]

### **2.4.2.3 Docker Images**

Das Docker Image beinhaltet sämtliche Informationen und Abhängigkeiten die ein Docker Container benötigt. Das Docker Image ist die Basis für einen oder mehrere Docker Container. Diese Images sind vom Host unabhängig und können mit wenig Aufwand verteilt und weiterentwickelt werden. Ein Docker Image kann aus einem Dockerfile erstellt werden. Zudem stellt Docker mit Docker HUB einen Service bereit, von dem vorgefertigte Images bezogen werden können. [Turnbull 2014]

### **2.4.2.4 Docker Container**

Ein Docker Container bezeichnet eine oder mehrere Prozesse, die „gekapselt“ ausgeführt werden. Ein Docker Container ist eine laufende Instanz eines Docker Images. Der Docker Container beinhaltet ausschließlich die auszuführenden Prozesse, sowie deren Abhängigkeiten. Die Docker Container können auf jedem Linux Rechner ausgeführt werden. [Turnbull 2014]

## 2.4.3 Das Dockerfile

Mit Hilfe eines Dockerfiles kann ein Docker Image erstellt werden. Das Dockerfile beinhaltet Anweisungen aus denen das Image erstellt wird. [Turnbull 2014]

```
# This dockerfile creates the image for the testframework for a student
FROM gcc:5.2
MAINTAINER David Stier <s58387@beuth-hochschule.de>
RUN apt-get update && apt-get install -y valgrind
```

*Listing 9: Darstellung eines Dockerfiles*

Dieses Listing zeigt das Dockerfile für das Image „student“ Dieses Image dient in dieser Arbeit als Grundlage für den Container der die Tests ausführt.

Die oberste Zeile stellt einen Kommentar dar, dies ist anhand des '#' Zeichens erkenntlich.

Die erste Anweisung „FROM“ gibt an, welches Image als Grundlage für dieses Image dient. Hier ist dies ein Image, welches bereits den GCC<sup>8</sup> Compiler in der Version 5.2 beinhaltet. In jedem Dockerfile muss diese Anweisung als die erste Anweisung vorhanden sein.

Die Anweisung „MAINTAINER“ ist optional und enthält Information über den Ersteller dieses Images.

Die dritte Anweisung „RUN“ führt einen oder mehrere Befehle aus. Mehrere Befehle können durch den && Operator angegeben werden. APT<sup>9</sup> ist der Packmanager der Debian und Ubuntu Linux Distributionen. Hier werden zuerst die Paketlisten aktualisiert (apt-get update). Anschließend wird die Valgrind Tool Sammlung installiert. (apt-get install valgrind)

Die weiteren möglichen Befehle sind: [Docker 2016][Docker 2014]

- CMD

Mit diesem Befehl kann ein Kommando im Container ausgeführt werden, im Gegensatz zu RUN wird CMD erst beim starten des Containers aufgerufen. Es kann maximal ein CMD Befehl in einem Dockerfile vorhanden sein. Zudem kann dieser Befehl beim Starten des Docker Containers überschrieben werden.

---

8 GCC – GNU C Compiler

9 APT – Advanced Packaging Tool

- **EXPOSE**  
Mit diesem Befehl werden Ports für diesen Container freigegeben. Falls z.B. ein Webserver in diesem Container ausgeführt werden soll, muss mit EXPOSE 80 der Port für das HTTP<sup>10</sup> Protokoll freigegeben werden.
- **ADD**  
Mit diesem Befehl können Ordner und Dateien in ein Image eingefügt werden. Es können auch Inhalte von einer URL<sup>11</sup> (GIT und HTTP) direkt in das Image eingefügt werden. Auch Tarball Archive werden unterstützt.
- **COPY**  
Dieser Befehl ist ähnlich wie ADD, jedoch werden keine URLs oder Tarball unterstützt. Die Syntax ist identisch zu ADD und lautet:  
COPY /pfad/auf/host /pfad/im/image alternativ kann der Befehl auch in der Form COPY [„/pfad/auf/host“, „/pfad/im/image/“] angegeben werden. Diese Form wird benötigt falls Leerzeichen im Pfad oder dem Dateinamen vorhanden sind.
- **WORKDIR**  
Dieser Befehl ändert das aktuelle Arbeitsverzeichnis- Zu Beginn wird grundsätzlich vom root Verzeichnis ausgegangen. Jeder folgende ADD, COPY, RUN oder CMD Befehl wird nach dem WORKDIR Befehl vom neuen Arbeitsverzeichnis aus aufgerufen. Die Syntax lautet: WORKDIR /neues/arbeitsverzeichnis Es werden auch relative Pfade unterstützt.
- **ENV**  
Mit diesem Befehl können Umgebungsvariablen gesetzt werden.
- **USER**  
Mit diesem Befehl kann der Benutzer geändert werden. Standardmäßig werden alle Befehle als root ausgeführt. Alle folgenden Befehle werden unter diesem Benutzer ausgeführt.
- **ENTRYPOINT**  
Dieser Befehl ist ähnlich wie CMD. Damit kann ein Programm direkt beim starten des Containers ausgeführt werden. Es lässt sich beim Starten des Container auch überschreiben, dafür muss aber das entrypoint flag gesetzt werden. Damit wird ein versehentliches Überschreiben des auszuführenden Kommandos verhindert.

---

10 HTTP – Hypertext Transfer Protocol

11 URL – Uniform Resource Locator

- ARG  
Mit diesem Befehl kann eine Variable beim Erstellen des Images setzen. Dazu muss das arg Flag beim erstellen verwendet werden.
- STOPSIGNAL  
Mit diesem Befehl kann ein Signalhandler aktiviert werden. Damit kann der Container durch ein Signal beendet werden. (siehe Abschnitt Interprozesskommunikation)
- VOLUME  
Mit diesem Befehl kann ein Volume im Container verwendet werden. Volumes werden im folgenden Abschnitt genauer behandelt.

```
david@david-ThinkPad-L440:~/Docker$ docker build -t student .  
Sending build context to Docker daemon 412.7MB  
Step 1 : FROM gcc:5.2  
----> 7996a078ff49  
Step 2 : MAINTAINER David Stier <s58387@beuth-hochschule.de>  
----> Using cache  
----> f704b3d8878a  
Step 3 : RUN apt-get update && apt-get install -y valgrind  
----> Using cache  
----> e34e4a66ca5e  
Successfully built e34e4a66ca5e  
david@david-ThinkPad-L440:~/Docker$ █
```

*Abbildung 15: Visualisierung der Erstellung eines Docker Images*

Die obere Abbildung veranschaulicht den Erstellungsvorgang eines Images. Dafür wird das Kommando `docker build -t „imagename“` ausgeführt. Das `-t` Flag gibt an, wo das Basisimage des FROM Befehls zu finden ist. Falls kein weiterer Pfad angegeben wird, wird DockerHub als Quelle verwendet. Das zu erstellende Image hat hier den Namen „student“ Der Punkt am Ende ist der Pfad zum Dockerfile. Hier befindet sich das Dockerfile im Arbeitsverzeichnis. In dieser Abbildung waren das Basisimage und sämtliche Layer bereits im Cache. Falls dies nicht der Fall wäre, wird zuerst das Basisimage von DockerHub bezogen und anschließend die weiteren Installationsschritte durchgeführt.

Eine Layer ist eine Art „Zwischenimage“ dabei wird bei jedem Aufruf eines Befehl eine neue Layer mit den Änderungen angelegt. [Turnbull 2014]

## 2.4.4 Docker Volumes

Die einzelnen Docker Container sind vollständig voneinander isoliert. Somit kann kein Container auf das Dateisystem des Hosts oder von anderen Containern zugreifen. In vielen Fällen müssen verschiedene Container Daten untereinander oder mit dem Host austauschen. Für diesen Fall können Volumes verwendet werden. Ein Volume ist ein Pfad auf dem der Host und ein oder mehrere Container zugreifen können. Ein Image kann mit die folgenden Methoden erstellt werden: [Turnbull 2014][Docker 2016]

Der Pfad zu einem Hostverzeichnis kann beim Starten eines Docker Containers mit dem Flag `-v` übergeben werden:

```
david@david-ThinkPad-L440:~$ docker run -i -t -v/pfad/auf/host:/pfad/im/container/ student
```

*Abbildung 16: Starten eines Docker Containers mit einem Hostpfad als Volume*

In dieser Abbildung wird ein Container aus dem Image „student“ gestartet. Mit dem `-v` Flag wird hier der Pfad auf dem Host als Volume in den Container eingefügt. Änderungen innerhalb des Volumes vom Host oder vom Container werden unverzüglich durchgeführt und die Daten sind sofort verfügbar.

Ein Volume kann auch durch den Befehl „docker volume create“ erstellt werden.

```
|david@david-ThinkPad-L440:~$ docker volume create --driver local --name bspvolume
```

*Abbildung 17: Erstellung eines benannten Docker Volumes*

Diese Abbildung veranschaulicht die Erstellung eines benannten Volumes. Dieses Volume hat den Namen „bspvolume“. Es kann wie ein absoluter Pfad verwendet werden.

```
david@david-ThinkPad-L440:~$ docker run -i -t -v bspvolume:/pfad/im/container/
```

*Abbildung 18: Verwendung eines benannten Docker Volumes*

Die obere Abbildung veranschaulicht die Verwendung eines benannten Volumes. Der Container kann über den Pfad „/pfad/im/Container/“ direkt auf die Dateien im Volume zugreifen.



Mittels des Befehls „docker volume inspect <volume\_name>“ können weitere Informationen über ein benanntes Volume erhalten werden.

```
david@david-ThinkPad-L440:~$ docker volume inspect bspvolume
[
  {
    "Name": "bspvolume",
    "Driver": "local",
    "Mountpoint": "/var/lib/docker/volumes/bspvolume/_data",
    "Labels": {}
  }
]
david@david-ThinkPad-L440:~$ █
```

*Abbildung 19: Veranschaulichung der Informationen eines benannten Docker Volumes*

Diese Abbildung veranschaulicht sämtliche Informationen des Volumes bspvolume. Der Driver „local“ gibt an, dass dieses Volume lokal auf dem Host vorhanden ist. Der Mountpoint gibt den absoluten Pfad des Volumes auf dem Host an. In diesem Beispiel wird kein Label verwendet. Labels werden von Zugriffskontrollen wie z.B. SELinux verwendet.

Ein Volume kann auch im Dockerfile mit dem Befehl VOLUME angegeben werden. Dies können jedoch nur benannte Volumes sein. Ein direktes Einbinden eines Host Pfads ist nicht möglich, da ansonsten die Kompatibilität des Dockerfiles auf anderen Rechnern nicht mehr gewährleistet ist. [Docker 2016]

## 2.5 Linux Systemprogrammierung

Dieser Abschnitt beschreibt einige der Grundlagen der Linux Systemprogrammierung mit dem Zugriff auf Dateien und andere Kommunikationsschnittstellen. Sowie das Zusammenwirken von Prozessen und die Interprozesskommunikation.

### 2.5.1 Die elementaren E/A Funktionen

Die elementaren E/A <sup>12</sup>Funktionen spielen unter Unix/Linux eine wichtige Rolle. Sie sind die Grundlage für jede Kommunikation mit dem Anwender oder anderen Prozessen. Ein Begriff der Unix Philosophie lautet: „Everything is a file“ Unter Unix und Linux wird jede E/A Funktion zu jedem Kommunikationsmittel als Datei abstrahiert. Diese Abstraktion ermöglicht es, dass für alle Kommunikationsmöglichkeiten die gleichen Grundfunktionen verwendet werden können. Dafür sind unter Unix/Linux zwei Abstraktionsebenen definiert: [Dausmann 2007][Wolf 2006][Brown 2010]

#### 2.5.1.1 Die Low-Level Dateiebene

Auf der Low Level Dateiebene werden sämtliche „Dateien“ mit einem Filedeskriptor bezeichnet. Ein Filedeskriptor ist eine einfache Nummer die einen Kommunikationsweg symbolisiert. Es werden 3 Standard Filedeskriptoren definiert. Diese sind standardmäßig offen, diese sind in der unteren Tabelle dargestellt.[Dausmann 2007][Wolf 2006]

Filedeskriptor Nr.:	Symbolische Konstante	Bedeutung
0	STDIN_FILENO	Standardeingabe
1	STDOUT_FILENO	Standardausgabe
2	STDERR_FILENO	Standardfehlerausgabe

*Tabelle 6: Übersicht über die Standard Filedeskriptoren*[Wolf 2006]

Zum Zugriff auf einen Filedeskriptor werden die Funktionen `read(int fd, void* buf, size_t count)` zum lesen und `write(int fd, void* buf, size_t count)` zum schreiben verwendet. Diese Funktionen lesen/schreiben `count` Byte vom Filedeskriptor `fd` in/aus den Puffer `buf`. Eine Formatierung findet auf dieser Abstraktionsebene nicht statt. Die Daten werden Byteweise verarbeitet. Zudem stehen weitere Funktionen zur Einstellung und Verwendung der Filedeskriptoren bereit.

---

<sup>12</sup> E/A – Eingabe/Ausgabe

### 2.5.1.2 Die High-Level Dateiebene

Auf der High-Level Ebene werden Streams verwendet. Diese Streams sind in der Programmiersprache C vom Typ FILE\* Neben dem Filedeskriptor beinhaltet dieser Datentyp weitere Informationen wie z.B. die Puffergröße und die aktuelle Zeigeposition des Lese/Schreibzeigers. Für die Standard Filedeskriptoren existiert jeweils ein Standard Stream. Diese sind: [Dausmann 2007][Wolf 2006]

Stream	Beschreibung
FILE *stdin	Standard Eingabestream auf die Tastatur
FILE *stdout	Standard Ausgabestream auf den Bildschirm
FILE *stderr	Standard Fehlerausgabestream auf den Bildschirm

*Tabelle 7: Übersicht über die Standard Streams [Wolf 2006]*

Die obere Tabelle veranschaulicht die drei Standardstreams. Sie werden von vielen C-Programmen verwendet. Es können auch eigene Streams z.B. auf Dateien mit der Funktion fopen(„pfad“, „modus“) angelegt werden. Ein Stream kann einfacher verwendet werden als ein Filedeskriptor, da die formatierten Eingabe- und Ausgabefunktionen der printf und scanf Familien verwendet werden können. Diese Funktionen greifen intern auch mit den read() und write() Funktionen auf die Filedeskriptoren zu.[Wolf 2006]

## 2.5.2 Das Signal System

Signale sind Software Interrupts. Sie werden verwendet um asynchrone Ereignisse zu behandeln. Sie können vom Betriebssystem an die einzelnen Prozesse gesendet werden. Ebenso können sie vom Anwender oder von Prozessen an andere Prozesse gesendet werden.[Stevens 2005][Wolf 2006]

Signale können durch verschiedene Ereignisse erzeugt werden, z.B. falls

- Die Hardware einen Fehler feststellt, dies kann z.B. eine Division durch 0 sein.
- Der Betriebssystem Scheduler den aktiven Prozess wechselt.
- Der Anwender oder ein Prozess die „kill“ Funktion aufruft.

Da ein Signal ein Interrupt ist, kann ein Prozess einen Signalhandler bereit stellen. Je nach Signal kann der Prozess

- Das Signal ignorieren
- Eine eigene Funktion aufrufen (durch einen Signalhandler)
- Die Standardaktion ausführen (meistens Prozess beenden.)

Die möglichen Aktionen hängen vom Signal ab. Nicht jedes Signal kann ignoriert werden oder einen eigenen Signalhandler bereitstellen.

Je nach verwendetem Betriebssystem stehen ca. 30 Signale zur Verfügung. Jedes Signal besitzt eine eindeutige Identifikationsnummer und einen eindeutigen Namen.

Die folgende Tabelle veranschaulicht eine kleine Auswahl an wichtigen Signalen:

Signalnummer	Signalname	Beschreibung
2	SIGINT	Dieses Signal wird vom Anwender durch Strg+C ausgelöst und kann überschrieben werden
9	SIGKILL	Beendet einen Prozess unverzüglich, kann nicht ignoriert werden
10	SIGUSR1	Dieses Signal steht zur freien Verfügung
14	SIGALRM	Wird durch einen Timer ausgelöst, kann ignoriert und überschrieben werden
18	SIGCONT	Wird vom Scheduler gesendet wenn ein Prozess fortgesetzt wird
19	SIGSTOP	Wird vom Scheduler gesendet wenn ein Prozess unterbrochen wird

*Tabelle 8: Übersicht über einige Linux Signale [Stevens 2005][Wolf 2006]*

Unter Linux kann eine Übersicht über alle Signale mit dem Befehl „kill -l“ angezeigt werden.

Falls ein Prozess ein Signal erhält, wird der zum Signal zugehörige Signalhandler aufgerufen. Für einige Signale wird vom Betriebssystem ein Signalhandler bereitgestellt. (z.B. SIGKILL) Für einige Signale kann ein eigener Signalhandler verwendet werden.

Dieser Signalhandler hat immer die Form `static void <name_des_signalhandlers>(void)`. Dieser Signalhandler muss im Prozess mit der Funktion `signal` bekannt gemacht werden. Der Aufruf lautet: `signal(„signalnummer“, „<name_des_signalhandlers>“)`.

Da ein Signalhandler eine ISR<sup>13</sup> darstellt, sollte die Funktion möglichst zeitsparend sein um das Interrupt System nicht zu lange zu blockieren.

---

13 ISR – Interrupt Service Routine

## 2.5.3 Prozesse

Ein Prozess ist eine Abstraktion eines laufenden Programms. Ein Betriebssystem benötigt diese Abstraktionen um die Programme zu verwalten. Ein Prozess ist eine Instanz eines Programms in Ausführung. Der Prozess beinhaltet neben dem Programm u.A. den gesamten Inhalt der CPU-Register und Variablen, den aktuellen Programmzähler, sowie sämtliche offenen Filedeskriptoren und Signalhandler. Durch diese Abstraktion ist das Betriebssystem in der Lage sämtliche Prozesse zu starten, zu beenden und zu unterbrechen. Das Betriebssystem kann mit dem Scheduler Ressourcen wie die CPU und RAM, sowie die E/A Aufrufe unter den Prozessen aufteilen. Jeder Prozess erhält eine eindeutige PID.<sup>14</sup> Ein Prozess wird sequenziell ausgeführt. [Tanenbaum 2009]

### 2.5.3.1 Prozesserzeugung

Sobald durch einen Systemaufruf oder durch Benutzereingabe ein Programm gestartet wird, wird vom Betriebssystem ein neuer Prozess erzeugt. Das Betriebssystem teilt dem Prozess den Speicherbereich und die benötigten Ressourcen zu und beginnt die Prozessausführung. [Tanenbaum 2009]

### 2.5.3.2 Prozessbeendigung

Ein Prozess kann durch die folgenden Ereignisse beendet werden. [Tanenbaum 2009]

- Normales Beenden  
Der Prozess ist vollständig abgearbeitet. In diesem Fall beendet sich der Prozess selbst. Das Betriebssystem kann die verwendeten Ressourcen anschließend an andere Prozesse verteilen. Dieses Beenden kann unter Linux mit der Systemfunktion `exit()` durchgeführt werden. Falls die `main` Funktion eines Programms abgearbeitet ist, wird dies automatisch ausgeführt.
- Beenden aufgrund eines schwerwiegenden Fehlers  
Ein schwerwiegender Fehler ist meistens ein Programmierfehler. Dies ist z.B. der Zugriff auf einen `NULL` Zeiger oder eine unbehandelte Ausnahme. In so einem Fall wird der Prozess vom Betriebssystem beendet.

---

14 PID – Process Identifier

- Beendigung durch einen anderen Prozess  
Der Prozess wird durch ein Signal (u. a. SIGKILL) von einem anderen Prozess beendet. Dies kann z.B. mit dem Befehl Strg+C durchgeführt werden.

### 2.5.3.3 Prozesszustände

Obwohl ein Prozess eine unabhängige Einheit darstellt, tritt häufig der Fall auf, dass ein Prozess mit einem anderen Prozess kommunizieren muss. Ebenso kann das Betriebssystem nicht alle Prozesse gleichzeitig ausführen, da die Anzahl der CPU Kerne begrenzt ist. Ein Prozess kann nicht weiter arbeiten wenn er z.B. auf eine Eingabe wartet. In diesem Fall blockiert der Prozess so lange bis die Eingabe vorhanden ist. Es existieren 3 Prozesszustände: [Tanenbaum 2009]

- rechnend  
Der Prozess wird ausgeführt
- rechenbereit  
Der Prozess könnte rechnen, ist aber aktuell vom Scheduler unterbrochen
- blockiert  
Der Prozess wartet auf ein bestimmtes Ereigniss

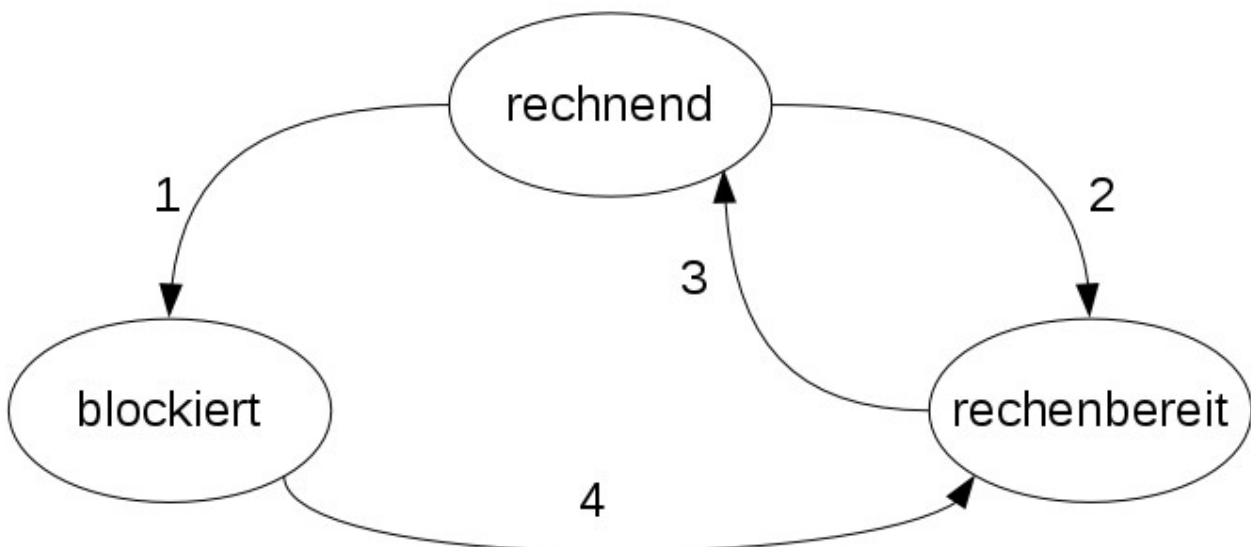


Abbildung 20: Visualisierung der 3 Prozesszustände [Tanenbaum 2009]

In dieser Abbildung sind die 3 Prozesszustände und deren Übergänge dargestellt. Diese Übergänge sind:

1. Der rechnende Prozess blockiert, weil er auf ein Ereignis wartet. z.B. Eingabe auf stdin
2. Der auszuführende Prozess wird vom Scheduler unterbrochen um die CPU einem anderen Prozess zuzuweisen
3. Der unterbrochene Prozess wird vom Scheduler fortgesetzt
4. Der blockierte Prozess hat sein Ereignis erhalten und kann fortgesetzt werden. Dazu muss ihm der Scheduler wieder die CPU zuweisen

#### 2.5.3.4 Prozesshierarchie

Die Prozesse unter Unix unterliegen einer Hierarchie. Die Erzeugung eines Prozesses durch einen anderen Prozess stellt eine gewisse Abhängigkeit her. Der neu erzeugte Prozess ist der Kindprozess des erzeugenden Prozesses.(Elternprozess) Ein Prozess stellt mit seinen Kind- und Kindeskind Prozessen eine Prozessfamilie dar. [Tanenbaum 2009]

Um einen Kindprozess zu erzeugen kann unter Linux der Systemaufruf `fork()` verwendet werden. Diese Funktion dupliziert den aktuellen Prozess. Dabei ist ein Prozess der Kindprozess des anderen. Durch die Duplizierung erbt der Kindprozess sämtliche Ressourcen wie Variablen, Filedeskriptoren und Signalhandler. Es ist eine gängige Praxis den Kindprozess durch einen Aufruf einer `exec()` Funktion mit einem anderen Programm zu überschreiben. Das neu ausgeführte Programm erbt die Filedeskriptoren, die Signalhandler werden jedoch überschrieben. [Stevens 2009]

Die Eltern-Kind Prozesshierarchie ist unter Unix/Linux systemrelevant. Beim Starten eines Linux Systems wird nur der Prozess `init` gestartet. Dieser `init` Prozess startet sämtliche anderen Prozesse. Somit ist jeder Prozess über mehrere Ebenen ein Nachfahre des `init` Prozesses.

Ein Prozess kann unter Linux auch durch die Systemfunktion `system()` gestartet werden. Dabei wird der aufrufende Prozess nicht überschrieben. Es wird ein neuer Prozess mit einer eigenen Shell gestartet. Dabei werden keine Ressourcen vererbt und es besteht kein direktes Verwandtschaftsverhältnis zwischen den Prozessen.



### 2.5.3.5 Threads

Jeder Prozess besitzt seinen eigenen Speicherbereich und wird sequenziell abgearbeitet. In manchen Fällen ist es sinnvoll, dass ein Prozess parallel abgearbeitet wird. Dies ist z.B. bei Multicore Prozessen der Fall. Ein Thread ist eine Art „Prozess im Prozess“. Mehrere Threads können auf einer Mehrkern CPU gleichzeitig abgearbeitet werden. Die Threads verwenden alle gemeinsam den Speicherbereich des Prozesses. Die Verwendung von Threads bietet sich bei rechenintensiven Prozessen auf Mehrkern CPUs an. Ebenfalls können verschiedene Aufgaben eines Prozesses in Threads aufgeteilt werden. Da mehrere Prozesse auf den gleichen Speicherbereich zugreifen, können Race Conditions entstehen. Unter Race Conditions versteht man den gleichzeitigen Zugriff auf die gleiche Ressource. Dies muss verhindert werden um die Konsistenz der Daten zu behalten. Dafür können z.B. Semaphore verwendet werden. [Tanenbaum 2009]

Threads werden z.B. vom Google Test Framework verwendet. [Google 2015]

## 2.5.4 Interprozesskommunikation

Prozesse müssen häufig untereinander kommunizieren. Dies wird als Interprozesskommunikation bezeichnet. Die IPC<sup>15</sup> beschäftigt sich mit den folgenden Kerngebieten:[Tanenbaum 2009][Stevens 2005][Wolf 2006]

- Weiterreichen von Informationen von einem Prozess zu einem anderen Prozess.
- Verhinderung des gleichzeitigen Zugriffs von zwei Prozessen auf eine gemeinsame Ressource (Race Conditions)
- Zeitliche Ablaufsteuerung bei Abhängigkeiten. Falls Prozess A Informationen von Prozess B benötigt, muss Prozess A so lange warten bis Prozess B fertig ist.
- Vermeidung von Deadlocks. Prozess A wartet auf Informationen von Prozess B. Gleichzeitig wartet Process B auf Informationen von Prozess A.

Im folgenden werden verschiedene Techniken der Interprozesskommunikation vorgestellt.

### 2.5.4.1 Pipes

Pipes stellen die älteste Form der Interprozesskommunikation dar. Pipes haben grundsätzlich die folgenden beiden Eigenschaften: [Stevens 2005][Wolf 2006][Holub 1988]

- Pipes sind unidirektional. Ein Prozess kann ein Pipe entweder nur schreiben oder nur lesen. Somit können keine Race Conditions auftreten. Für eine bidirektionale Kommunikation müssen zwei Pipes verwendet werden.
- Pipes können nur zwischen verwandten Prozessen verwendet werden. Dies ist häufig die Kommunikation zwischen einem Elternprozess und seinem Kindprozess. Eine Kommunikation zwischen zwei Kindprozessen ist auch möglich.

Ein Pipe kann unter Linux mit C mit der Funktion `int pipe(int fd[2])` erzeugt werden. Bei Erfolg gibt diese Funktion 0 zurück. Im Fehlerfall wird -1 zurückgegeben. Als Argument wird ein Array aus zwei Filedeskriptoren angegeben. Die Funktion `pipe` erzeugt zwei Filedeskriptoren für den Zugriff auf das Pipe. Der Filedeskriptor `fd[0]` wird zum Lesen aus der Pipe verwendet. `fd[1]` wird zum Schreiben in die Pipe verwendet.

---

15 IPC – Interprocess Communication

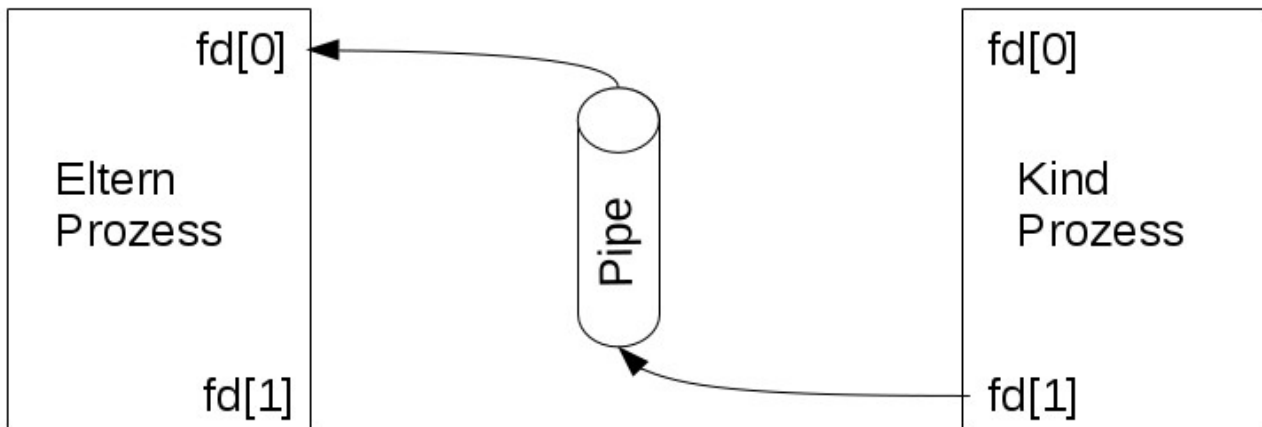


Abbildung 21: Darstellung eines Pipes zwischen 2 Prozessen [Wolf 2006]

Diese Abbildung veranschaulicht das Funktionsprinzip eines Pipes. In diesem Beispiel hat der Elternprozess den Schreibdeskriptor mit `close()` geschlossen. Der Kindprozess hat seinen Lesedeskriptor geschlossen. Mit diesem Pipe kann der Kindprozess nun Informationen an den Elternprozess senden. Dafür können die Funktionen `read()` und `write()` verwendet werden. Ebenfalls können mit der Funktion `fdopen()` Streams auf die Filedeskriptoren erstellt werden. Mit der Funktion `dup()` können Filedeskriptoren dubliziert werden, somit wäre es möglich die Streams `stdin`, `stdout` und `stderr` umzuleiten.

Die Funktion `FILE *popen(const char *cmd, const char type)` führt einen Fork und den Aufruf des Kommandos „cmd“ durch. Mit dieser Funktion wird ein Kindprozess erzeugt und direkt durch das Programm „cmd“ Überlagert. Dabei wird, je nach „type“ entweder `stdin` oder `stdout` direkt in ein Pipe umgeleitet. Die Ein- bzw. Ausgabe kann anschließend über den zurückgegeben Stream erfolgen. [Stevens 2005][Wolf 2006]

### 2.5.4.2 Benannte Pipes

Die benannten Pipes werden auch FIFO<sup>16</sup>s genannt. Im Gegensatz zu Pipes können FIFOs auch von Prozessen verwendet werden, die nicht miteinander verwandt sind. Dies ist möglich, da der Zugriff über das Dateisystem erfolgt. Unter Unix und Linux kann mit dem Systemaufruf `mkfifo()` ein FIFO erstellt werden. Dieser Aufruf ist aus dem Terminal und aus C-Programmen aus möglich und liefert einen Filedeskriptor auf den neu erstellten FIFO zurück. Mit diesem Filedeskriptor und der `open()` Funktion kann der FIFO zum lesen und schreiben geöffnet werden. Im Gegensatz zu Dateien können Informationen aus einem FIFO nur einmal gelesen werden. Ein FIFO kann zudem nur zum unidirektionalen Nachrichtenaustausch verwendet werden. FIFOs werden häufig von Client-Server Anwendungen verwendet. (Ein oder mehrere Clients schreiben Nachrichten, während ein Server Nachrichten liest) [Stevens 2005][Wolf 2006]

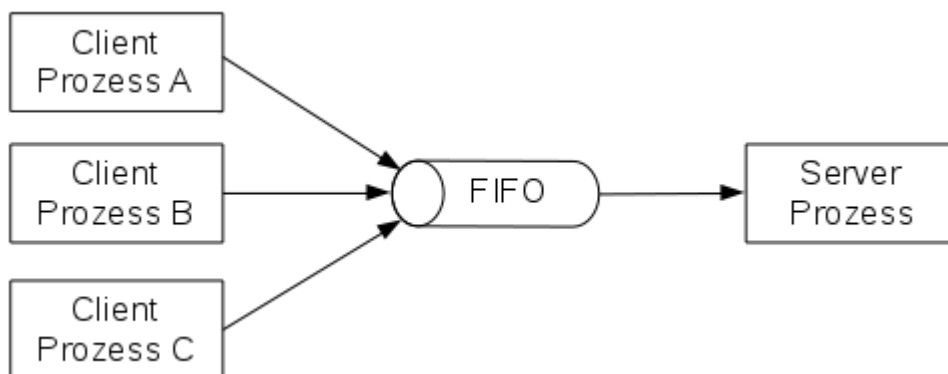


Abbildung 22: Darstellung eines FIFOs zwischen Clients und einem Server Prozess [Wolf 2006]

Diese Abbildung veranschaulicht den Client Server Anwendungsfall für einen FIFO. Drei Client Prozesse schreiben Nachrichten in einen FIFO. Diese Nachrichten werden vom Server Prozess ausgelesen und verarbeitet. Der Zugriff auf einen FIFO geschieht über den Filedeskriptor mit den Funktionen `read()` und `write()`. Standardmäßig ist ein FIFO blockierend, d.h. es kann maximal ein Teilnehmer gleichzeitig auf den FIFO zugreifen. In diesem Fall werden Race Conditions automatisch verhindert. Falls der FIFO mit der Funktion `fcntl()` und dem `O_NONBLOCK` Flag als nicht blockierend eingestellt wird, müssen die Teilnehmer die Race Conditions selbst verhindern.

---

16 FIFO – First In First Out

### 2.5.4.3 Shared Memory

Ein Shared Memory ist ein Speicherbereich, der direkt vom Unix/Linux Kernel verwaltet wird. Der Shared Memory kann von mehreren Prozessen zum Lesen und Schreiben verwendet werden. Dies ist die schnellste Methode der Interprozesskommunikation, da die Informationen nur geschrieben und gelesen werden müssen. Sie müssen nirgends zwischen gepuffert werden. Der Nachteil bei der Verwendung von Shared Memory besteht darin, dass keine automatische Zugriffskontrolle erfolgt. Die Prozesse müssen diese vollständig selbst durchführen. Dies kann mit Semaphoren geschehen. [Wolf 2006]

### 2.5.4.4 Semaphore

Ein Semaphor ist ein Zähler, der angibt, wie viele Instanzen einer Ressource vorhanden sind. (Im Regelfall 1). Bei der Initialisierung eines Semaphors muss dieser auf einen positiven Wert mit der Anzahl der möglichen Instanzen gesetzt werden. Für den Zugriff auf einen kritischen Bereich wird vor dem Zugriffs der Semaphor um 1 dekrementiert. Nach dem Zugriff wird der Semaphor wieder um 1 inkrementiert. Der Semaphor gibt somit immer an, wie viele Instanzen der Ressource aktuell verfügbar sind. Falls ein Prozess feststellt, dass ein Semaphor den Wert 0 hat, muss er so lange warten bis ein anderer Prozess den Semaphor wieder auf einen positiven Wert inkrementiert. [Stevens 2005] [Wolf 2006]

Die Operationen, die den Semaphor prüfen und inkrementieren bzw. dekrementiert müssen atomar sein. d.h. sie dürfen auf keinen Fall unterbrochen werden. Dies kann z.B. durch eine kurzzeitige Deaktivierung der Interrupts geschehen.

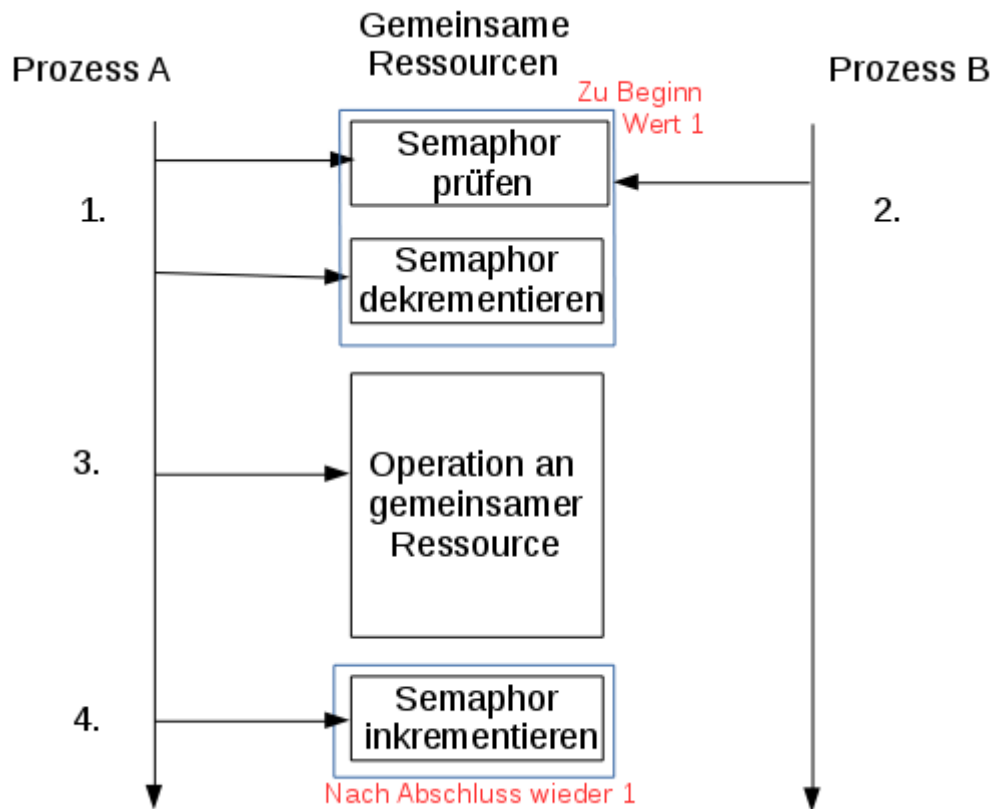


Abbildung 23: Veranschaulichung der Anwendung eines Semaphors

Diese Abbildung veranschaulicht die Funktionsweise eines Semaphors. Zwei Prozesse wollen gleichzeitig auf eine gemeinsame Ressource zugreifen. Die blauen Rechtecke symbolisieren atomare Funktionsbereiche. Die Anwendung verläuft wie folgt:

1. Prozess A will auf eine gemeinsame Ressource zugreifen. Dafür muss der Prozess zuerst den Semaphor prüfen. Da der Semaphor aktuell den Wert 1 besitzt, wird dieser Wert auf 0 dekrementiert und Prozess A kann die gemeinsame Ressource verwenden.
2. Prozess B versucht ebenfalls auf die gemeinsame Ressource zuzugreifen. Da der Semaphor von Prozess A auf 0 gesetzt wurde, muss Prozess B warten bis dieser Semaphor wieder einen positiven Wert besitzt.
3. Prozess A kann mit dem Zugriff auf die gemeinsame Ressource beginnen.
4. Nachdem Prozess A seine Operation abgeschlossen hat wird der Semaphor wieder um 1 inkrementiert. Der Semaphor hat jetzt wieder den Wert 1.

Anschließend könnte Prozess B auf die gemeinsame Ressource zugreifen, da der Semaphor wieder einen positiven Wert besitzt.

#### 2.5.4.5 Sockets

Die Socket Schnittstelle wird hauptsächlich in der Netzwerkprogrammierung verwendet. Diese Schnittstelle kann auch zur Interprozesskommunikation verwendet werden. (Der Netzwerktreiber ist auch ein Prozess) Für die Sockets sind sämtliche Anforderungen zur Vermeidung von Race Conditions und zur Synchronisation bereits implementiert. Zur Verwendung der Socket Schnittstelle steht die Funktion `socketpair()` bereit. Diese Funktion erzeugt ein lokales Socket Paar. Diese Funktion erzeugt zwei Socketdeskriptoren. Mit diesen beiden Deskriptoren kann anschließend ähnlich zum Pipe eine Verbindung zwischen zwei Prozessen hergestellt werden. Zum Nachrichtenaustausch können auch bei Sockets die Funktionen `read()` und `write` verwendet werden. Im Gegensatz zu Pipes können lokale Sockets auch bidirektional verwendet werden. [Stevens 2005][Wolf 2006]

#### 2.5.4.6 Das Pseudoterminal

Die Terminal Schnittstelle ist eine serielle Schnittstelle. Unter Unix/Linux wird diese TTY<sup>17</sup> Schnittstelle zum Zugriff auf serielle E/A Interfaces wie z.B. RS-232 verwendet. Ein Terminal kann für die Pufferung und Baudrate frei konfiguriert werden. Diese IPC-Schnittstelle besitzt die weitgehendsten Einstellungsmöglichkeiten. [Stevens 2005]

Unter Unix/Linux existiert auch eine Pseudoterminal Schnittstelle. Diese PTY<sup>18</sup> Schnittstelle ist eine reine Softwareschnittstelle. Für die beteiligten Prozesse sieht eine PTY Schnittstelle genau wie eine TTY Schnittstelle aus.

---

17 TTY – Teletype

18 PTY – Pseudo Teletype

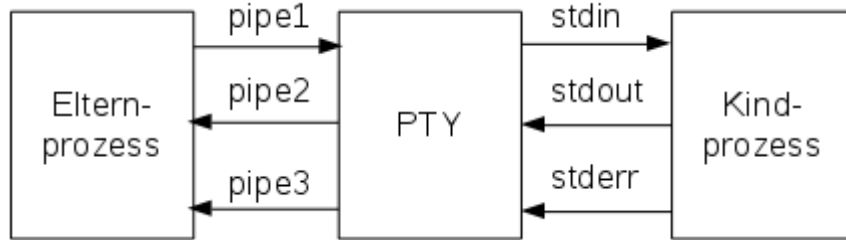


Abbildung 24: Visualisierung der PTY Schnittstelle [Stevens 2005]

Die obere Abbildung veranschaulicht die Anwendung eines Pseudoterminals. In dieser Abbildung wird das PTY zwischen einem Elternprozess und einem Kindprozess verwendet. Dabei werden die drei Standard Streams des Kindprozesses auf das Pseudoterminal umgeleitet. Der Elternprozess greift über Pipes auf das Pseudoterminal zu um die Ein- und Ausgabe des Kindprozesses abzugreifen.

Theoretisch können die Standardstreams direkt über Pipes an den Elternprozess weitergeleitet werden. In diesem Fall wird vom Betriebssystem die Pufferung von stdin und stdout auf Vollpufferung gesetzt. (stderr bleibt ungepuffert) Dies ist in vielen Fällen sinnvoll, wie z.B. zur Umleitung in eine Datei.

Für einige Fälle müssen jedoch abwechselnd Lese und Schreibprozesse an den Kindprozess weitergereicht werden. Dies ist bei Verwendung der Vollpufferung schwierig, da der Kindprozess die Puffer selbstständig leeren müsste. Mit einem Pseudoterminal kann die Pufferung vollständig auf die eigenen Bedürfnisse eingestellt werden.

Diese Methode der Umleitung wird z.B. von Telnet verwendet. Ebenso verwendet die Laufzeitumgebung dieser Arbeit die Pseudoterminal Schnittstelle, da diese Methode die bestmögliche Kontrolle über die Ein- und Ausgabe erlaubt. [Stevens 2005]



## 2.6 Weitere Verwendete Software und Tools

Die hier aufgelisteten Anwendungen und Tools werden in dieser Arbeit verwendet.

### 2.6.1 GCC

Der GCC Compiler ist der C-Compiler der GNU<sup>19</sup> Compiler Collection. Dieses Open Source Projekt stellt Compiler für viele Programmiersprachen und Plattformen bereit. Der Begriff GCC stand ursprünglich für den GNU C Compiler, heutzutage wird diese Bezeichnung für die GNU Compiler Collection verwendet. [Stallman 2003]

### 2.6.2 G++

Der G++ Compiler ist der C++ Compiler der GNU Compiler Collection. [Stallman 2003]

### 2.6.3 Strip

Das Open Source Tool Strip wird verwendet um Object Dateien zu bearbeiten. Mit diesem Tool ist es möglich Symbole aus Object Dateien zu entfernen. Ein Symbol ist z.B. ein Funktionsname. [Strip 2016]

### 2.6.4 Apache

Apache ist ein HTTP Webserver. Dieses Open Source Projekt ist der heute meist verwendete Webserver. In diesem Projekt wird er verwendet um eine PHP<sup>20</sup> Webseite darzustellen, die die Benutzerschnittstelle zu diesem Testframework darstellt. [Apache 2016]

### 2.6.5 Python

Python ist eine interpretierende, höhere Skriptsprache. Python ist eine einfach zu erlernende Sprache, da sie eine einfache, klare und übersichtliche Syntax besitzt. In diesem Projekt wird durch ein Python Skript die Docker Container für die Studenten angelegt und ausgeführt. [Python 2016]

---

19 GNU – GNU is not Unix

20 PHP – Hypertext Preprocessor

## 2.7 Fazit

Die xUnit Architektur ist eine universelle Architektur, die auf verschiedenen Systemen zur Anwendung kommen kann. Mithilfe von Unit Tests können mit wenig Aufwand sehr mächtige Tests durchgeführt werden. Diese Testausführung ist mit dem Google Test Framework einfach durchzuführen. Zudem werden Unit Tests als wichtiger Bestandteil der TDD<sup>21</sup> eingesetzt. Diese Methode wird bei der agilen Entwicklung verwendet. [Rupp 2014]

Das Docker System stellt eine einfache Schnittstelle für die automatisierte Containerverwendung bereit. Durch diese Betriebssystem Virtualisierung können viele Anwendungen mit wenig Aufwand auf verschiedenen Systemen ausgeführt werden.

Die Linux Systementwicklung ist ein sehr umfangreiches Thema, die Systementwicklung ist für die Entwicklung von vielen Anwendungen nötig. Die Verwendung von Linux Programmen und deren Entwicklung wird in den nächsten Jahren durch die immer weiter voranschreitende Verbreitung von Docker immer wichtiger. [Loschwitz 2015]

---

21 TDD – Test Driven Development

## 3 Stand der Technik

### 3.1 Ähnliche Projekte

Das Testframework TravisCI ist ein online basiertes Testframework. TravisCI verwendet virtuelle Maschinen um verschiedene Testumgebungen bereitzustellen. TravisCI wird vornehmlich für Open Source Projekte verwendet und ermöglicht es Anwendungen unter verschiedenen Umgebungen zu prüfen. Für diese Anwendungen werden eine Vielzahl an Programmiersprachen und Testumgebungen unterstützt. [Travis 2016]

Mustafa Akin, Tutor an der Bilkent Universität hat die auf Docker basierende Webanwendung PAGES<sup>22</sup> entwickelt. Mit dieser Testumgebung können Studierende online Programmierübungen in unterschiedlichen Programmiersprachen durchführen. PAGES unterscheidet sich von diesem Testframework darin, dass dieses Testframework lokal geschriebenen Quellcode prüft. PAGES verhindert, dass unterschiedliche Umgebungen zu fehlerhaften Ergebnissen führen. Dieses Testframework prüft ausschließlich Quellcode. Dieser wird in einem Docker Container erstellt. Da der gesamte Quellcode unter der gleichen Umgebung erstellt wird, sollten keine umgebungsabhängigen Fehler auftreten. Zudem verwenden Programmierübungen aus Einsteigerkursen meist ausschließlich die C-Standardbibliothek. [Akin 2016]

---

<sup>22</sup> PAGES – Program Assignment Grading System

## 3.2 Literatur

Das Buch „Advanced Programming in the Unix Environment“ von W. Richard Stevens und Steven A. Rago behandelt die komplette Systementwicklung unter Unix/Linux. Das Buch beinhaltet Erläuterungen zu allen relevanten Aspekten der Systemprogrammierung. Diese sind mit Codebeispielen verständlich erläutert.

Das Buch „Moderne Betriebssysteme“ von Andrew S. Tanenbaum beinhaltet eine detaillierte Beschreibung sämtlicher Komponenten und Vorgängen, die ein modernes Betriebssystem verwendet.

Paul Hamill beschreibt in seinem Buch „Unit Test Frameworks“ das Prinzip der xUnit Architektur. Ebenso erläutert er die Verwendung von unterschiedlichen Unit Test Frameworks. Diese sind u.a. CppUnit und JUnit.

„The Docker Book“ von James Turnbull veranschaulicht das Funktionsprinzip von Docker. Dieses Buch beinhaltet sämtliche relevanten Informationen die für das Verständnis und für die Verwendung von Docker benötigt werden.

## 4 Das Testframework

Dieses Kapitel beinhaltet den Aufbau und die Entwicklung des Testframeworks.

### 4.1 Ablauf des Testvorgangs

Zum Testen der Quellcodes werden die folgenden Tests in dieser Reihenfolge durchgeführt:

1. Statischer Quellcodetest

Der Statische Quellcodetest wird durch den GCC Compiler durchgeführt. Da für die folgenden Unit- und Speichertests ein ausführbares Programm benötigt wird, muss das zu prüfende Programm erstellt werden. Der GCC Compiler gibt bei gefährlichen Quellcodepassagen Warnungen aus. Bei Fehlerhaften Quellcodepassagen wird eine Fehlermeldung ausgegeben. Die Warnungen und Fehlermeldungen beinhalten Hinweise und Position des Fehlers. Die gesamte Ausgabe des GCC Compilers ist somit eine Quellcodeanalyse. Diese wird direkt verwendet.

2. Unit Tests und Speichertest

Im zweiten Schritt werden Unit Tests und ein optionaler Speichertest mit dem Valgrind Tool „memcheck“ durchgeführt. Mit den Unit Tests können sowohl die Ein- und Ausgabe des zu testenden Programms, als auch einzelne Funktionen getestet werden. Anschließend kann ein optionaler Speichertest mit dem Valgrind Tool durchgeführt werden. Für diesen Speichertest werden die gleichen Eingaben wie bei den Unit Tests verwendet.

Sämtliche Testergebnisse werden in der Datei „result.txt“ gespeichert. Diese Datei wird zu Beginn eines Testdurchlaufs angelegt und beinhaltet sämtliche Informationen wie die Matrikelnummer, die Aufgabe, die zu prüfenden Quellcodes und die Ausgabe aller Tests.

## 4.2 Programmablauf des Testframeworks

Als Schnittstelle zwischen Anwender und dem Testframework dient eine PHP Webseite. Auf dieser Webseite kann jeder Student seine Quellcodes hochladen. Diese Webseite wird von einem Apache Webserver zur Verfügung gestellt. Der Apache Webserver wird in einem eigenen Docker Container ausgeführt.

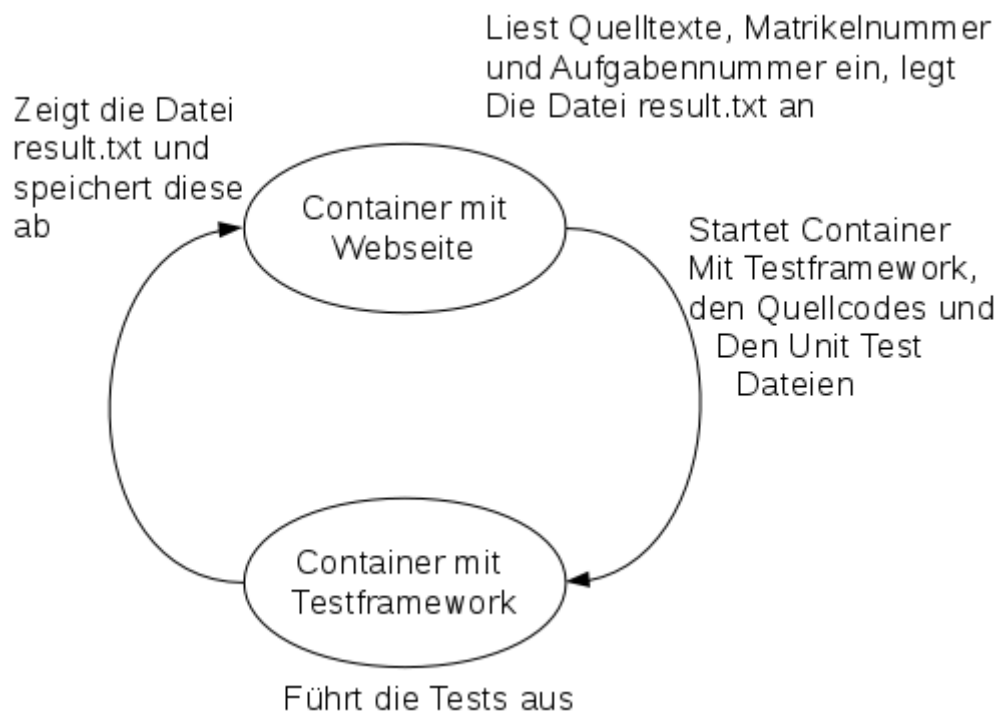


Abbildung 25: Darstellung des Programmablaufs mit Docker Containern

Diese Abbildung veranschaulicht den Ablauf des Testprogramms. Die Webseite liest neben den Quelltexten auch die Matrikelnummer und die zu prüfende Aufgabennummer ein. Dabei wird von der PHP Seite die Datei result.txt angelegt. In dieser Datei befinden sich sämtliche Eingelesenen Informationen, sowie eine Liste mit allen eingegebenen Dateien.

Über ein Python Skript wird ein Container aus dem Image „student“ gestartet. Als Namen erhält der Container die Matrikelnummer des zu prüfenden Studenten. Über ein Volume erhält dieser Container die zu prüfenden Quellcodes, die für diese Aufgabe hinterlegten Unit Tests Dateien, sowie das ausführbare Testframework. Anschließend wird das Testframework gestartet.

Anschließend an die Testausführung wird die Datei „result.txt“ auf der Webseite dargestellt. Dies ermöglicht es dem Studenten das Testergebnis zu sehen. Die Datei result.txt wird für die Kontrolle in einem weiteren Volume abgespeichert. Dies ermöglicht es dem Dozenten zu überwachen welcher Student die Aufgaben korrekt bearbeitet hat. Jeder Student hat die Möglichkeit die Aufgaben unbegrenzt oft zu wiederholen. Die result.txt Datei wird bei jedem Aufruf überschrieben.

### 4.3 Architektur des Testframeworks

Das Testframework besteht aus drei Modulen. Jeweils ein Modul ist für eine Aufgabe zuständig. Dies ermöglicht eine einfache Erweiterung um weitere Module.

Jedes Modul führt einen Schritt des Tests durch. Insgesamt werden 3 Schritte durchgeführt:

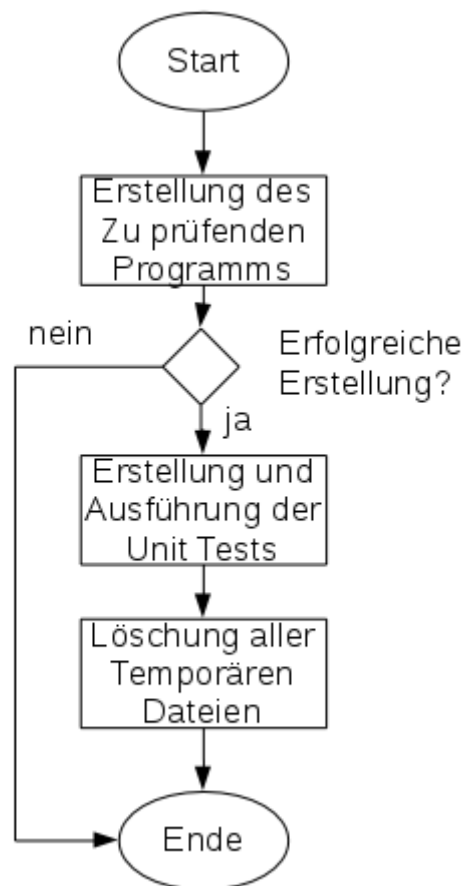


Abbildung 26: Darstellung des Ablaufs eines Testdurchgangs

Diese Abbildung stellt den Testablauf dar. Dieser wird im folgenden erläutert.

Zuerst wird das zu prüfende Programm erstellt. Falls dies nicht erfolgreich sein sollte wird der Testablauf abgebrochen. Falls die Erstellung erfolgreich ist werden anschließend die Unit Tests und der Speichertest durchgeführt. Der Speichertest ist als Unit Test implementiert um ihn bei Bedarf zu deaktivieren. Anschließend werden sämtliche temporären Dateien gelöscht, die während eines Testdurchlaufs angelegt wurden. Falls die Testausführung abgebrochen wird, existieren auch keine temporären Dateien.



### 4.3.1 Die Erstellung des zu testenden Programms

Dieses Modul erstellt das zu prüfende Programm. Dies ist gleichzeitig der statische Quellcode Test. Dies wird durch die Funktion `create_test_program()` durchgeführt.

Diese Funktion verfügt über die Hilfsfunktion `parse_file()`. Diese Hilfsfunktion liest aus der zweiten Zeile der „result.txt“ Datei die einzelnen Source und Header Dateien ein. Die Header Dateien werden in einer Liste gespeichert und für die Unit Tests benötigt und die Source Dateien werden für den Aufruf des GCC Compilers in einem Zeigerarray gespeichert. In diesem Zeigerarray befinden sich nach Aufruf der `parse_file()` Funktion sämtliche Argumente für den GCC Compiler. Diese sind:

- `gcc` Der Name des GCC Compilers
- `-Wall` Dieses Flag signalisiert, dass alle Warnungen ausgegeben werden
- `-save-temps=obj` Dieses Flag teilt dem GCC Compiler mit, dass die Object Dateien nicht gelöscht werden sollen. Diese werden für die Unit Tests benötigt
- Alle eingelesenen Dateien mit der Endung `.c`

Anschließend erzeugt die Funktion `create_test_program()` ein Pipe. Dieses wird für die Umleitung der GCC Compilerausgabe benötigt.

Daraufhin wird mit der `fork()` Funktion ein Kindprozess erzeugt. Im Kindprozess werden zuerst `stdout` und `stderr` in das Pipe umgeleitet. Danach wird mit der `execvp()` Funktion der GCC Compiler mit allen Argumenten aufgerufen.

Der Elternprozess liest alle Ausgaben des GCC Compilers aus und speichert diese in der Datei `result.txt`.

Nachdem der Kindprozess beendet ist. (GCC Ausführung ist abgeschlossen) wird geprüft ob die Programmerstellung erfolgreich war. Dafür wird mit der `access()` Funktion überprüft ob die Datei „a.out“ vorhanden ist. Falls diese Datei vorhanden ist, war die Erstellung erfolgreich. In diesem Fall wird der Vermerk „Creation of test program finished“ in die `result.txt` Datei geschrieben. Falls a.out nicht vorhanden ist, ist die Erstellung fehlgeschlagen. In diesem Fall wird der Vermerk „Program could not build“ in die `result.txt` Datei geschrieben.

Die folgende Abbildung stellt vereinfacht den Ablauf der `create_test_program()` Funktion dar.

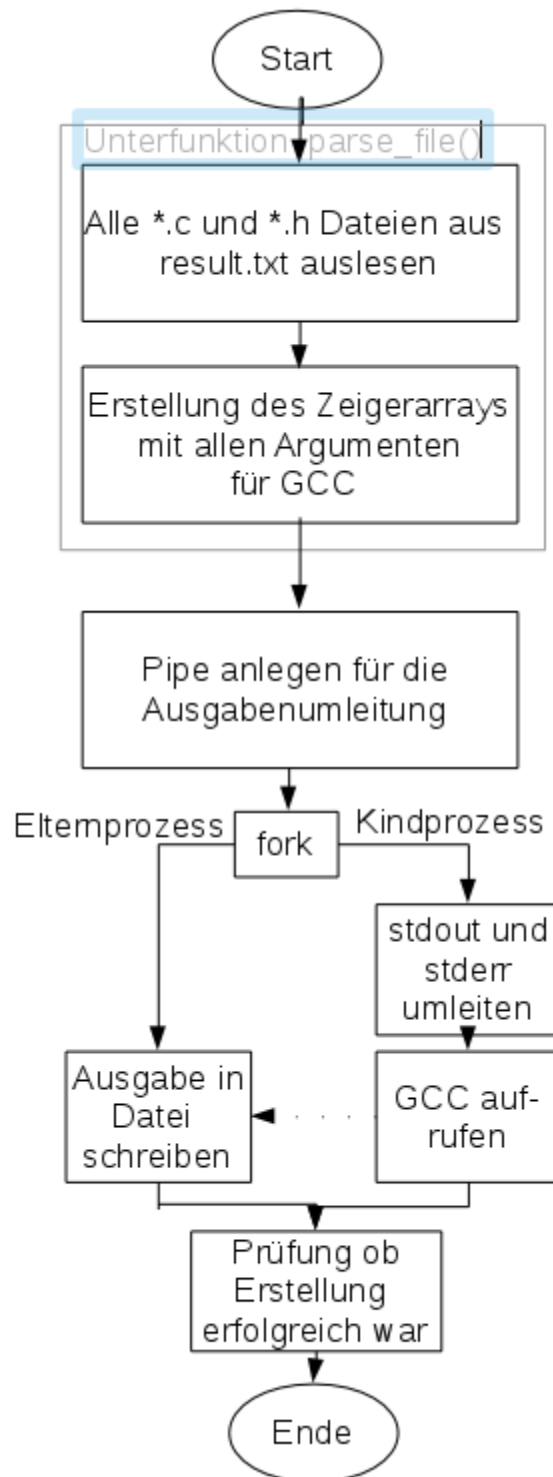


Abbildung 27: Vereinfachte Darstellung der create\_test\_program() Funktion

Das zu prüfende Programm erhält hier aus Einfachheit den Namen a.out. Dies ist der Standard Name des GCC Compilers.[Stallman 2003] Da dieses Programm nur vom Testframework verwendet wird, ist ein aussagekräftiger Name nicht notwendig.

Der Aufruf des GCC Compilers mit den Funktionen fork() und execvp() kann hier nicht durch die popen() Funktion vereinfacht werden. Dies liegt daran, dass die popen() Funktion entweder stdin oder stdout umleiten kann. Der GCC Compiler gibt die Warnungen und Fehlermeldungen über stderr aus.

Der GCC Compiler muss hier mit allen Quellcodenamen direkt aufgerufen werden. Ein vereinfachter Aufruf durch gcc -Wall \*.c ist mit einer exec() Funktion nicht möglich. Dies liegt daran, dass die Substitution von \*.c in alle c Dateien von der Shell durchgeführt wird. Der Aufruf von exec() verwendet keine Shell und kann die Substitution nicht selbst durchführen. Wenn der GCC Compiler über die system() Funktion mit system("gcc -Wall \*.c") aufgerufen wird, dann ist das möglich. Durch System wird eine Shell angelegt, die die Substitution durchführt und den GCC Compiler mit allen c Dateien aufruft. In diesem Fall wäre keine Umleitung durch Pipes möglich, da kein Verwandtschaftsverhältnis zwischen dem aufrufenden Prozess und dem GCC Prozess bestehen würde.

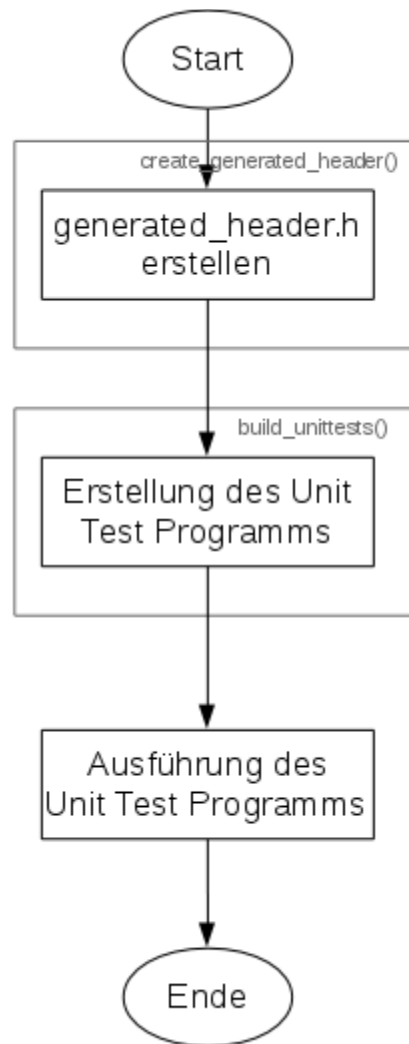
### 4.3.2 Die Erstellung und Ausführung der Unit Tests

Der zweite Prüfungsschritt sind die Unit Tests und der Speichertest. Der Speichertest ist in den Unit Tests integriert. Im Hauptprogramm werden die Unit Tests von der Funktion `execute_Unittests()` erstellt und ausgeführt. Die Speichertests werden über die die Unit Tests aufgerufen. Ausgeführt werden die Speichertests über die Laufzeitumgebung. Mit dieser Laufzeitumgebung können auch die Eingabe und Ausgabe des Testprogramms über Unit Tests geprüft werden.

Die Unit Tests müssen zur Laufzeit erstellt werden. Dies ist nötig, da die Ausführung von Unit Tests in einer eigenen Anwendung durchgeführt wird. Da die zu prüfenden Programme im Quelltext vorliegen, müssen auch die zugehörigen Unit Tests zur Laufzeit erstellt werden. Für die Einbindung der zu prüfenden Quellcodes wird während der Laufzeit der Header `generated_header.h` angelegt. Dieser Header stellt die Schnittstelle zwischen dem zu prüfenden Studenten C Programm und dem prüfenden C++ Unit Test Programm dar. Ebenfalls ermöglicht der generierte Header das Einbinden von Headern, die erst zur Laufzeit bekannt werden. Einzig die Funktionsnamen müssen eindeutig bekannt sein.

Im Rahmen dieser Arbeit wurde die Funktion `execute_unittests()` erstellt. Diese Funktion führt drei Schritte aus:

1. Erstellung des generierten Headers
2. Erstellung der Unit Test Anwendung
3. Ausführung dieser Unit Test Anwendung



*Abbildung 28: Vereinfachte Darstellung der Funktion `execute_unittests()`*

Diese Abbildung stellt vereinfacht die Funktion `execute_unittests()` dar. Die grauen Blöcke stellen die Unterfunktionen `create_generated_header()` und `build_unittests()` dar.

Die Unterfunktion `create_generated_header()` erzeugt den generierten Header `generated_header.h`. Dafür wird die Liste mit den Headernamen aus der Funktion `parse_file()` verwendet.

Für diesen wurde die folgende Form definiert:

```
#ifndef GENERATED_HEADER_H
#define GENERATED_HEADER_H
#ifdef __cplusplus
    extern "C"{
#endif
#include "headername.h"
#ifdef __cplusplus
    }
#endif
#endif
```

Der Schritt `#include "headername.h"` wird für jeden eingelesenen Header durchgeführt. Mit den Präprozessor Flags `#ifdef __cplusplus` wird es dem G++ Compiler ermöglicht C Code einzubinden.

Die neu erstellte Funktion `build_unittests()` erstellt im Anschluss das ausführbare Programm „UnitTests“. Dieses Programm führt die Unittests aus.

Im ersten Schritt wird mit dem Tool Strip das Symbol „main“ aus den Object Dateien des Studentencodes entfernt. Dies ist notwendig, da der Linker keine zwei main Funktionen verarbeiten kann. (Eine Main Funktion im Studenten Code und eine Main Funktion für das Testprogramm) Der Aufruf geschieht mit der `system()` Funktion:

```
system("strip --strip-symbol=main *.o")
```

Im Gegensatz zum GCC Compiler kann hier die Substitution `*.o` verwendet werden. Dies wird hier angewandt, da der Name der Object Datei mit der main Funktion unbekannt ist. Da nur eine main Funktion existieren kann wird auch nur aus der entsprechenden Object Datei das Symbol entfernt. Die Ausgabe von Strip ist hier nicht relevant, somit kann auf eine Umleitung verzichtet werden.

Anschließend werden die `*.cpp` Dateien der Unittests mit dem G++ Compiler erstellt, aber nicht gelinkt. Beim Verlinken müssen die Object Dateien des Studentencodes berücksichtigt werden.

Anschließend wird mit dem G++ Compiler die Verlinkung der Object Files der Unit Tests, des zu prüfenden Programms, sowie Bibliotheken für das Google Testframework und der Laufzeitumgebung.

Dafür wird wieder die `system()` Funktion verwendet. Bei beiden G++ Aufrufen werden außer die Header Dateien des Google Testframeworks (für die Unit Test Codes benötigt) sowie die `pthread` Bibliothek eingebunden. (Vom Google Test Framework benötigt)

Anschließend wird das erstellte Programm `UnitTests` ausgeführt. Eine Umleitung der Ausgabe ist nicht notwendig. Die Ausgabe in die Datei `result.txt` wird vom Testrunner durchgeführt.

### 4.3.3 Löschen der temporären Dateien

Im letzten Schritt werden sämtliche temporären Dateien gelöscht. Dafür wird die Funktion `cleanup()` aufgerufen.

Die Funktion `cleanup()` ruft mit der `system()` Funktion das Kommando „`rm -rf`“ auf um die temporären Dateien zu löschen. Diese sind:

- alle `*.o`, `*.s`, `*.i`  
Dateien dies sind die temporären Dateien die der GCC und der G++ Compiler erzeugt haben
- der generierte Header
- die beiden Programme `a.out` und `UnitTest`
- die Datei `inputs.txt`  
diese Datei wird von der Laufzeitumgebung angelegt und verwendet

## 4.4 Die Auswertung der Unit Tests

Für die Unit Tests wird das Google Test Framework verwendet. Dieses Test Framework stellt eine API bereit. Mit dieser API können einzelne Module des Google Test Frameworks angepasst werden.

Das Google Test Framework stellt die Klasse TestEventListener bereit. Diese Klasse wird vom Testrunner für die Testausgabe verwendet.

Für dieses Projekt wird ein eigener TestEventListener verwendet. Dieser TestEventListener speichert alle Ausgaben in der Datei result.txt.

Dafür wird die Klasse TestframeworkListener verwendet. Diese Klasse erbt von der Google Test Framework Klasse EmptyTestEventListener. (Ein leerer TestEventListener) [Google 2015]

Die Klasse TestframeworkListener überschreibt die folgenden Methoden:

- OnTestProgramStart()  
Diese Methode wird vom Google Test Framework vor der Ausführung der Tests aufgerufen. Hier wird diese Funktion verwendet um die Datei result.txt zu öffnen. [Google 2015]
- OnTestStart()  
Diese Methode wird vor der Ausführung eines Unit Tests aufgerufen. Der TestEventListener schreibt in die result.txt Datei die Meldung, dass der Test mit dem Namen <TestCaseName>.<UnitTestName> gestartet wird. [Google 2015]
- OnTestEnd()  
Diese Methode wird nach der Ausführung eines Unit Tests ausgeführt. Es wird die Meldung <TestCaseName>.<UnitTestName> finished in die result.txt Datei geschrieben. [Google 2015]
- OnTestProgramEnd()  
Diese Methode wird nach Beendigung sämtlicher Tests aufgerufen. Hier wird die Meldung, dass die Testausführung abgeschlossen ist, sowie eine Übersicht über die Anzahl der ausgeführten, erfolgreichen und fehlgeschlagenen Tests in die Datei result.txt geschrieben Diese Datei wird anschließend geschlossen. [Google 2015]



- OnTestPartResult()

Diese Methode wird nur bei einem fehlgeschlagenen Vergleich aufgerufen. Der TestframeworkListener schreibt in diesem Fall eine Meldung mit dem fehlerhaften Vergleich in die Datei result.txt [Google 2015]

```
Start Tests
Test: InputTest.testAddition started
Test: InputTest.testAddition finished

Test: InputTest.testSubtraktion started
Test: InputTest.testSubtraktion finished

Test: Rechner.SummeTest started
Failure in file: Unittests/test_rechner.cppLine: 9
  Expected: 3
  To be equal to: summe(1,1)
  Which is: 2
Test: Rechner.SummeTest finished

Test: Rechner.DifferenzTest started
Test: Rechner.DifferenzTest finished

Test: Rechner.Produkttest started
Test: Rechner.Produkttest finished

Test: Rechner.QuotientTest started
Test: Rechner.QuotientTest finished

Unittests finished
6 tests in: 2 TestCases
Tests succeeded: 5
Tests failed: 1
```

*Abbildung 29: Darstellung der Ausgabe des  
TestframeworkListener*

Diese Abbildung veranschaulicht sämtliche Ausgaben des TestEventListeners. In der ersten Zeile ist die Ausgabe der Methode OnTestProgramStart() dargestellt. Es werden sechs Unit Tests ausgeführt. Somit sind die Ausgaben der Methoden OnTestStart() und OnTestEnd() insgesamt sechs mal dargestellt. Diese Methoden veranschaulichen, dass der einzelne Unit Test gestartet und beendet wurde. Der dritte Unit Test ist fehlgeschlagen. Somit wird die Methode OnTestPartResult() aufgerufen. Die Ausgabe dieser Methode ist im dritten Unit Test dargestellt. Es wird die Datei und Zeile des fehlgeschlagenen Unit Tests ausgegeben. Ebenso werden der Erwartungswert und der aktuelle Ausgabewert der fehlerhaften Funktion ausgegeben. (Hier wurde für die Demonstration ein falscher Vergleich durchgeführt) Zuletzt wird die Ausgabe der OnTestProgramEnd() Methode ausgegeben, schreibt eine Übersicht über die Anzahl der ausgeführten, der erfolgreichen, sowie der fehlgeschlagen Tests in die Datei result.txt

Das Google Testframework inklusive des selbsterstellten TestEventListeners werden in der Bibliothek Gtest.a bereitgestellt. Dies erleichtert das Erstellen der Unit Tests während der Ausführung im Docker Container.

## 4.5 Die Laufzeitumgebung

Dieses Testframework wird im Rahmen einer Lehrveranstaltung verwendet. Ein Großteil der zu prüfenden Studentencodes in Lehrveranstaltungen verwenden die Standard-ein- und Ausgabe als Benutzerschnittstelle. Zum Erlernen der Programmiersprache C sind die ersten Programme meist ausschließlich Programme die nur aus der Main Funktion und Standard-ein- und Ausgabe bestehen, z.B. das Hello World Programm. Diese sind zu simpel um sie mit herkömmlichen Unit Test Frameworks zu testen. Deshalb wird eine Laufzeitumgebung verwendet um die Standard-ein- und Ausgabe mit Stringvergleichen über Unit Tests zu prüfen. Die Laufzeitumgebung kann ebenfalls ein aufgehängtes Programm beenden. Die Laufzeitumgebung führt auch Speichertests mit dem Valgrind Tool memcheck durch.

### 4.5.1 Architektur der Laufzeitumgebung

Die Hauptaufgabe der Laufzeitumgebung ist es, die Standard-ein- und Ausgabe an das Unit Test Framework durchzureichen. Dafür wird ein Pseudoterminal verwendet.

Die Laufzeitumgebung stellt die folgenden Funktionen bereit:

- `runtime_setup()`
- `runtime_tearardown()`
- `readFromProgram()`
- `writeToProgram()`
- `do_valgrind_check()`

Der Signalhandler `sig_alarm()` ist ebenfalls ein Bestandteil der Laufzeitumgebung.

Die Funktionen werden im folgendem erläutert:

#### **4.5.1.1 Die Funktion runtime\_setup()**

Diese Funktion initialisiert die Laufzeitumgebung. Dabei wird das Pseudoterminal geöffnet und das zu prüfende Programm mittels Fork als Kindprozess gestartet. Ebenso wird der Signalhandler sig\_alarm() mit dem Signal SIGALRM verknüpft und der Timer für die Beendigung gestartet. Diese Funktion muss in der setUp() Methode des Testfixtures aufgerufen werden.

#### **4.5.1.2 Die Funktion runtime\_teardown()**

Diese Funktion blockiert die Ausführung so lange bis das zu prüfende Programm beendet ist. Somit wird sichergestellt, dass die Unit Test Ausführung korrekt weitergeführt werden kann. Diese Funktion muss in der tearDown() Methode des Testfixtures aufgerufen werden.

#### **4.5.1.3 Die Funktion readFromProgram()**

Diese Funktion liest von der Standardaus- und Standardfehlerausgabe des zu Prüfenden Programms. Für den Leseprozess wird der Kindprozess des zu prüfenden Programms kurzzeitig mit den Signalen SIGSTOP und SIGCONT unterbrochen.

#### **4.5.1.4 Die Funktion writeToProgram()**

Diese Funktion schreibt einen String an die Standardeingabe des zu prüfenden Programms. Nach dem Schreibvorgang wird der geschriebene String mit der read() Funktion wieder eingelesen. Dies ist notwendig um den Positionszeiger des Filedeskriptors anzupassen. Zudem werden sämtliche eingegebenen Strings in der Datei inputs.txt gespeichert. Diese werden für den Speichertest benötigt. Der Kindprozess mit dem zu prüfenden Programms wird während dem Schreib und Leseprozess kurzzeitig mit den Signalen SIGSTOP und SIGCONT unterbrochen

#### **4.5.1.5 Die Funktion do\_valgrind\_check()**

Diese Funktion führt den Speichertest mit dem Valgrind Tool memcheck durch. Falls das zu prüfende Programm durch die Laufzeitumgebung beendet wurde, wird dieser Test ausgelassen. Ansonsten könnte sich der ausführende Server aufhängen.

#### 4.5.1.6 Synchronisation

Das Pseudoterminal arbeitet mit zwei File Deskriptoren, jeweils einer für die Laufzeitumgebung und das zu prüfende Programm. Diese Filedeskriptoren können beide zum Lesen und zum Schreiben verwendet werden. Das Pseudoterminal verhindert Race Conditions automatisch. Jedoch gibt es hier einen Bug im Linuxkernel. [Vogelsberger 2014] Durch diesen Bug können Race Conditions entstehen. Ebenso wird in der Funktion `writeToProgram()` ein Schreib- und eine Lesevorgang durchgeführt. Dazwischen könnten Race Conditions entstehen.

Diese Race Conditions lassen sich nicht durch ein Semaphor beheben. Dazu müsste das zu prüfende Programm den Semaphor selbst setzen. Da das zu prüfende Programm über keine Kenntnisse über das PTY verfügt muss eine andere Technik verwendet werden.

Die einzige Möglichkeit Race Conditions durch einen Prozess (die Laufzeitumgebung) zu verhindern besteht darin, den Kindprozess (das zu Prüfende Programm) kurzzeitig zu unterbrechen. Dafür werden die Signale SIGSTOP vor dem Zugriff auf das PTY und SIGCONT nach dem Zugriff auf das PTY an den Kindprozess gesendet. Diese Signale werden auch vom Scheduler verwendet um einen Prozesswechsel durchzuführen. Dies benötigt zwar eine höhere Ausführzeit als Semaphore, da aber nur wenige Zugriffe durchgeführt werden kann dies vernachlässigt werden.

Ein weiteres Problem stellen Deadlocks dar. Es besteht die Möglichkeit, dass der Kindprozess blockiert weil er auf eine Eingabe wartet. Gleichzeitig wartet die Laufzeitumgebung auf eine Ausgabe des Kindprozess. Um diese Deadlock zu verhindern wird in den Funktionen `readFromProgram()` und `writeToProgram()` nach der Nachrichtenübermittlung die Systemfunktion `sleep()` aufgerufen. Es werden insgesamt 500 Mikrosekunden gewartet. Somit wird der Scheduler den Kindprozess ausführen. Dieses Problem besteht insbesondere zu Beginn der Programmausführung. Es ist undefiniert ob zuerst der Elternprozess oder der Kindprozess aufgerufen wird. Der Elternprozess wartet zu Beginn grundsätzlich 500 Millisekunden um den Kindprozess den Vortritt zu überlassen. In dieser Zeitspanne wird der Scheduler mit an Sicherheit grenzender Wahrscheinlichkeit den Kindprozess aufrufen. Die Zeitspanne ist hier größer, da der Kindprozess zu Beginn mehr Zeit benötigt. Der Kindprozess muss erst die Standardstreams umleiten und mit der `execvp()` Funktion den Prozess zu überlagern. Die Zeitspanne ist dafür ausreichend groß gewählt,

#### 4.5.1.7 Das Pseudoterminal

Die Weiterleitung der Standard Streams ist ein wichtiger Teil der Laufzeitumgebung. Die Weiterleitung kann weder mit Pipes oder mit FIFOs durchgeführt werden. Der Grund dafür ist die Pufferung. Sobald ein stdin oder stdout umgeleitet werden, werden diese vom Linux Kernel automatisch auf Vollpufferung umgestellt; stderr bleibt ungepuffert. Diese Pufferungen können nicht durch die Funktion `setbuf()` verändert werden. Da nicht sichergestellt werden kann, dass die zu testenden Quellcodes nach jeder E/A Funktion die Puffer durch `fflush()` leeren, muss ein Pseudoterminal verwendet werden. Ein PTY ist die einzige Möglichkeit um die Standardstreams mit Zeilenpufferung zu verwenden.

Ein Pseudoterminal ist ein virtuelles Gerät. Es verwendet ein Master und ein Slave Interface. Master und Slave verfügen jeweils über einen eigenen Filedeskriptor.

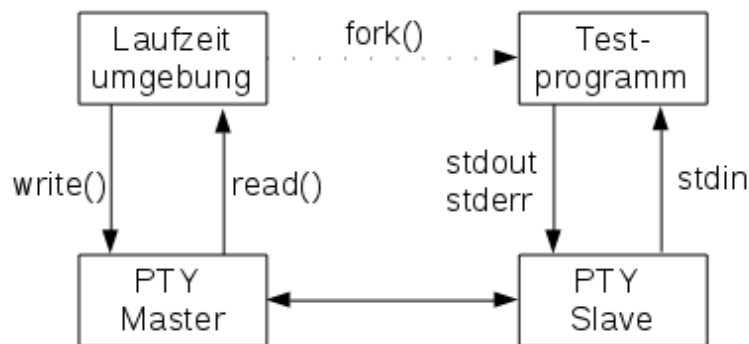


Abbildung 30: Visualisierung des Pseudoterminals

Diese Abbildung veranschaulicht die Funktionsweise des Pseudoterminals. Das Master Interface ist unter Linux unter `/dev/ptmx` erreichbar. Das Slave Interface ist unter `/dev/pts/„nummer“` zu erreichen. (Die „nummer“ muss beim Anlegen des PTYs angegeben werden) Diese beiden „Dateien“ stellen die Schnittstelle zum Pseudoterminal dar. Bei der Anwendung eines Pseudoterminals müssen für diese Interfaces die Lese- und Schreibe Rechte mit der `chmod` Funktion zugewiesen werden. Die Weiterleitung der Nachrichten zwischen Interfaces wird vom Kernel durchgeführt.

Das Pseudoterminal kann mit den Strukturen `termios` für die Terminal Optionen (z.B. Baudrate) und `winsize` (Fenstergröße) frei konfiguriert werden. Dies ist bei einer reinen Umleitung der Standard Streams jedoch nicht notwendig.

Für die Speichertests kann das Pseudoterminal nicht verwendet werden, da `memcheck` eine eigene Laufzeitumgebung verwendet. Diese kann nicht über das PTY angesprochen werden. Dafür wird ein Pipe verwendet.

## 4.5.2 Die Aufgaben der Laufzeitumgebung

Die Aufgaben der Laufzeitumgebung werden hier erläutert. Dies sind die Nachrichtenweiterleitung, das Beenden beim Aufhängen und der Speichertest.

### 4.5.2.1 Weiterleitung der Standardstreams

Die Laufzeitumgebung stellt die Funktionen `readFromProgram()` und `writeFromProgram()` zur Verfügung.

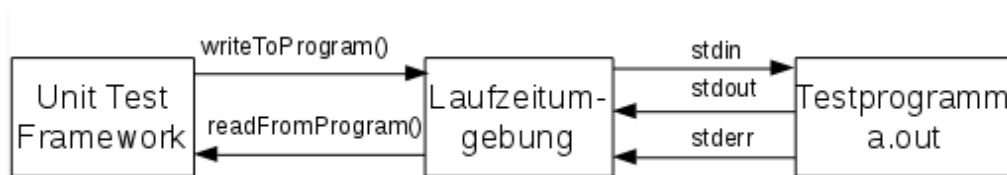


Abbildung 31: Visualisierung der Weiterleitung der Standardstreams

Diese Abbildung veranschaulicht die Weiterleitung der Standardstreams. Mit der Funktion `writeToProgram()` kann das Unit Test Framework Strings an die Laufzeitumgebung senden. Die Laufzeitumgebung gibt diese Strings an die Standardeingabe des zu testenden Programms durch.

Die Standardausgabe sowie die Standardfehlerausgabe des zu prüfenden Programms können über die Funktion `readFromProgram()` über die Laufzeitumgebung gelesen werden. Da das PTY nur einen Filedeskriptor bereitstellt, ist eine Unterscheidung dieser Streams nicht möglich.

#### 4.5.2.2 Überwachung des Aufhängens des Testprogramms

Da für die Tests das Programm ausgeführt werden muss, besteht die Möglichkeit, dass sich das zu prüfende Programm aufhängt. Da das Testframework auf einem Server läuft, könnte dies dazu führen, dass sich der Server aufhängt. Zudem ist dies ein Programmfehler, der von dem Testframework festgestellt werden muss. Dafür wird ein Timer verwendet. Dieser Timer wird in der Funktion `runtime_setup()` aktiviert. Die maximale Ausführzeit beträgt fünf Sekunden. Nach Ablauf dieser fünf Sekunden wird das Signal `SIGALRM` ausgelöst. Sobald dieses Signal eintrifft, wird der Kindprozess durch den Signalhandler `sig_alarm()` beendet. Dafür wird das Signal `SIGKILL` verwendet. Dieses Signal führt zuverlässig zum Beenden des Prozesses.

Die `runtime_tearardown()` Funktion wartet mit der `waitpid()` Funktion auf das Beenden des Kindprozesses. Mit den Makros `WIFEXITED` und `WTERMSIG` wird überprüft, ob das Programm normal beendet ist (`WIFEXITED`) oder durch ein Signal beendet wurde. (`WTERMSIG`)

Falls das Testprogramm korrekt beendet ist, wird die Meldung „Program finished successfully“ in die `result.txt` Datei geschrieben. Falls das Testprogramm durch das Signal beendet wurde, wird die Meldung „Program freezes and was aborted“ in die Datei `result.txt` geschrieben.



### 4.5.2.3 Durchführung von Speichertests

Die Speichertests werden mit dem Valgrind Tool memcheck durchgeführt. Die Laufzeitumgebung stellt dafür die Funktion `do_valgrind_check()` zur Verfügung. Diese Funktion kann nur nach der Ausführung der Tests auf die Standardein- und Ausgabe durchgeführt werden. Dies ist notwendig, da die Eingaben erst von der Funktion `writeToProgram()` in die Datei `inputs.txt` geschrieben werden müssen.

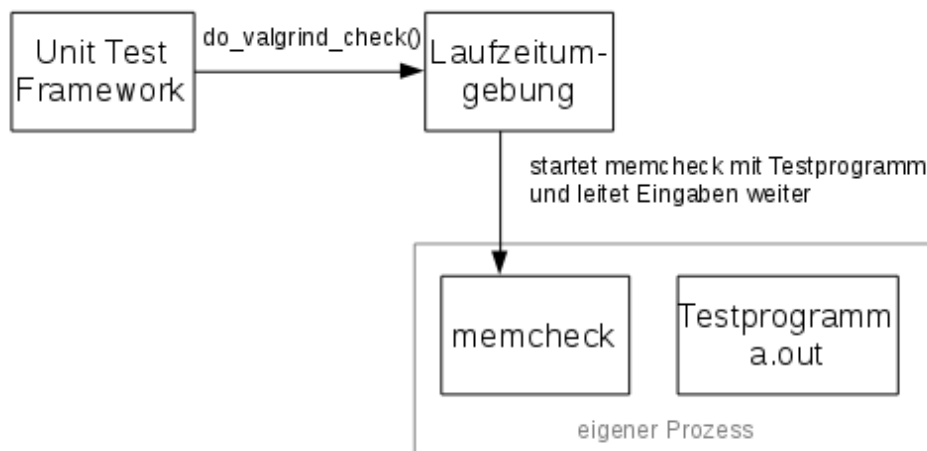


Abbildung 32: Visualisierung der Speichertest Durchführung

Diese Abbildung veranschaulicht den Speichertest mit memcheck. Das Unit Test Framework ruft mit der Funktion `do_valgrind_check()` das Programm memcheck über die Laufzeitumgebung auf. Die Laufzeitumgebung gibt automatisch die eingelesenen Strings an Memcheck weiter. Diese Nachrichten werden von Memcheck an das zu prüfende Programm weitergeleitet. Die Ausgabe des Testprogramms wird vernachlässigt. Diese wurde schon von den Unit Tests geprüft. Es müssen die gleichen Eingabestrings verwendet werden. Da die Ausführung des zu Prüfenden Programms von der Eingabe abhängig sein kann. Ebenso kann damit geprüft werden ob ein zu großer Eingabestring einen Pufferüberlauf auslöst.

Der Aufruf von Valgrind geschieht durch die `popen()` Funktion. Dabei wird die Eingabe über ein Pipe an memcheck geleitet. Memcheck ermöglicht die Ausgabe über einen Fildeskriptor. (Das Flag `--log-fd`). Über diesen Fildeskriptor wird die `result.txt` übergeben. Ebenfalls wird der quiet Modus (Flag `--quiet`) verwendet. Somit gibt Memcheck nur Fehlermeldungen aus. Dies erleichtert die Interpretation der `result.txt` Datei.

## 4.6 Die Webseite

Dieser Abschnitt behandelt den Webseite. Diese stellt die Schnittstelle zwischen den Studenten und dem Testframework dar.

### 4.6.1 Aufbau der Webseite

Eine Webseite liest die Quelltexte, die Matrikelnummer und die Aufgabennummer ein. Diese Webseite wird von einem Apache Webserver in einem Docker Container bereitgestellt. Die Webseite ist in PHP erstellt.



← ⓘ | 127.0.0.1

# Docker Unit Test Framework

Matrikelnummer:  Aufgabe 1 ▾

Dateien:   Keine Dateien ausgewählt.

Abbildung 33: Darstellung der Webseite des Testframeworks

Diese Abbildung veranschaulicht die Webseite. Es existiert ein Eingabefeld für die Matrikelnummer, eine Auswahlliste für die Aufgabennummer und ein Uploadfeld für die zu prüfenden Quelltexte.

Der Button „Absenden“ startet den Test. Dafür startet die Webseite einen neuen Docker Container der die Tests durchführt. Nach Beendigung der Tests wird auf der Webseite das Ergebnis dargestellt.

### 4.6.2 Datenaustausch zwischen Webseite und Testcontainer

Die Webseite muss mit dem Testcontainer Daten austauschen. Dafür werden Docker Volumes verwendet.

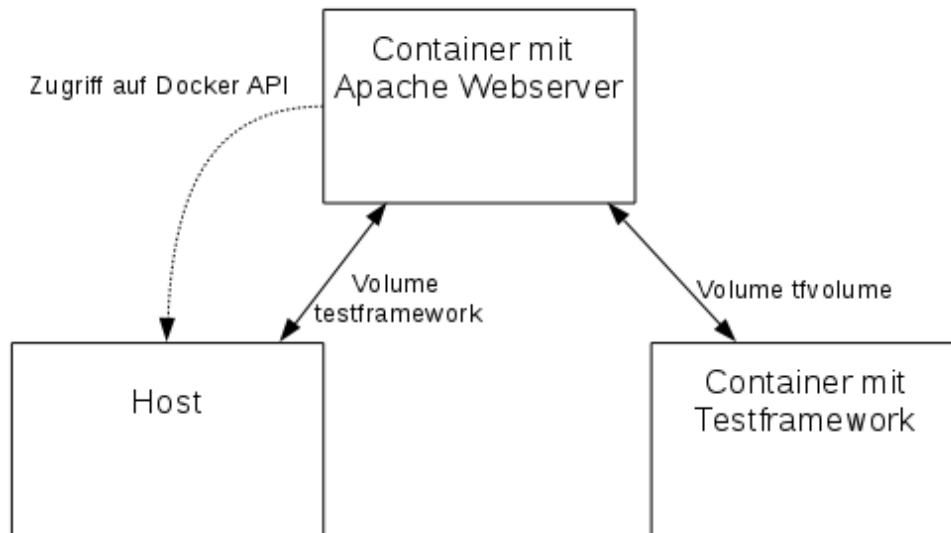


Abbildung 34: Darstellung der Docker Container und Docker Volumes

Diese Abbildung veranschaulicht die Docker Container und die Verwendung der Volumes. Es werden zwei Volumes verwendet. Das Volume „testframework“ wird zwischen dem Host und dem Webserver Container verwendet. Das volume „tfvolume“ wird zwischen dem Webserver Container und dem Testcontainer verwendet.

#### 4.6.2.1 Das Volume testframework

Dieses Volume stellt das Interface für die Lehrkraft dar. In diesem Volume können auf dem Host Rechner die Unit Test Dateien eingegeben werden und die Ergebnisse der Studenten ausgelesen werden. Dieses Volume beinhaltet die folgenden Unterordner:

- test  
Dieser Unterordner beinhaltet sämtliche Dateien, die das Testframework zur Ausführung benötigt. Dies sind das Testframework als ausführbares Programm, die Laufzeitumgebung als Bibliothek und das Google Test Framework als Bibliothek. Die Bibliotheken befinden sich in den Unterordnern „runtime“ und „Unittests“
- Unittests  
Dieser Unterordner beinhaltet in Unterordnern die TestCases und TestFixtures. Für jede Aufgabe wird ein eigener Unterordner verwendet. Diese Unterordner verwenden die Aufgabennummer als Namen (0, 1, 2, ...)
- Results  
Dieser Unterordner wird verwendet um die ausgegeben result.txt Dateien zu speichern. Die Webseite legt für jede Matrikelnummer einen Unterordner an. In dieser Unterordner wird die Datei result.txt als „aufgabe\_<Aufgabennummer>“.txt gespeichert.
- include  
Dieser Ordner ist das include Verzeichnis des Google Testframeworks, dieser wird für die Erstellung der Unit Tests benötigt.

#### 4.6.2.2 Das Volume tfvolume

Dieses Volume wird ausschließlich zwischen den Containern der Webseite und dem Testframework verwendet. Es ist standardmäßig leer und wird nur für den kurzzeitigen Datenaustausch des Testvorgangs verwendet.

### 4.6.3 Ablauf eines Tests

Zu Beginn eines Tests wird der Ordner „test“ aus dem Volume „testframework“ in das Volume „tfvolume“ kopiert. Ebenfalls werden die hinterlegten Unit TestCases und Testfixtures vom Volume „testframework“ in das Volume „tfvolume“ kopiert. Die Webseite legt die Datei result.txt an. In diese Datei schreibt sie die Matrikelnummer, die Aufgabennummer, sowie die Namen aller zu prüfenden Quellcodes. Diese Datei wird im Volume „tfvolume“ gespeichert. Die eingegebenen Quellcodes werden ebenfalls im Volume „tfvolume“ gespeichert. Anschließend wird der Testcontainer erstellt. In diesem Container wird das ausführbare Programm „Testframework“ aufgerufen und somit die Tests ausgeführt. Nach Beendigung der Tests wird der Testcontainer gelöscht. Die Datei result.txt wird mit den Testergebnissen auf der Webseite angezeigt und im Unterordner „Results“ des Volumes „testframework“ gespeichert.

Ursprünglich war die Idee, dass für jeden Studenten ein eigener Container angelegt wird, dies ist jedoch nicht notwendig, da die Informationen nicht dauerhaft abrufbar sein müssen, bzw. dauerhaft gespeichert werden müssen. Das Testframework wird erst zur Laufzeit in den Testcontainer kopiert. Dies hätte auch im Image des Studentencontainers durchgeführt werden. Falls dieses Testframework jedoch weiterentwickelt wird, müsste für jede Änderung das Image neu erstellt werden. Das selbe gilt für die TestCases und TestFixtures. Änderungen an Komponenten des Testsystems können ohne Zeitverlust durchgeführt werden. Dies ist mit etwas Mehraufwand in der Testausführung verbunden, jedoch überwiegt der Vorteil der dynamischen Änderung.

## 4.6.4 Aufruf des Testcontainers

Die Webseite muss einen weiteren Container aufrufen. Dafür wird die Docker API verwendet. Es stehen mehrere inoffizielle Docker API Bibliotheken für verschiedene Programmiersprachen auf der Docker Homepage zur Verfügung.<sup>23</sup>

Da die Webseite auf PHP basiert, war die ursprüngliche Idee, dass eine PHP API verwendet wird. Sämtliche PHP APIs sind jedoch mangelhaft dokumentiert, nicht mehr verfügbar und/oder benötigen eine umständliche Installation. Deshalb wird die Python Bibliothek Docker-Py verwendet. Diese Bibliothek ist gut dokumentiert und einfach zu Verwenden. [Docker-Py 2016]

Die Webseite ruft ein Python Skript für die Containererstellung auf.

```
from docker import Client
import sys

#get an instance to the Docker API
cli = Client(base_url='unix:///var/run/docker.sock')

#matrikelnummer is the name of the container
matnr = sys.argv[1]

#create a new container from the image "student" using the volume "tfvolume" and execute the Testframework
container = cli.create_container(image='student', volumes='/tfvolume',
host_config=cli.create_host_config(binds={'/var/lib/docker/volumes/tfvolume/_data':{'bind':'/tfvolume', 'mode':
'rw'}}), working_dir='/tfvolume/test/',command='./Testframework' ,name=matnr)

#start container
response=cli.start(container.get('Id'))

#Wait until container is finished
cli.wait(container.get('Id'))

#delete the container
cli.remove_container(matnr)
```

Listing 10: Darstellung des Python Skripts container.py

Dieses Listing stellt das verwendete Python Skript „container.py“ dar.

---

23 [https://docs.docker.com/engine/reference/api/remote\\_api\\_client\\_libraries/](https://docs.docker.com/engine/reference/api/remote_api_client_libraries/)

Der Ablauf des Skriptes ist wie folgt:

Zuerst wird mit der Bibliotheksfunktion Client("pfad/zur/docker/api") eine Schnittstelle zur API eingerichtet. [Docker-Py 2016]

Die Methode create\_container() erstellt einen neuen Container. Dieser Container verwendet das Volume „tfvolume“. Von diesem Volume muss der absolute Pfad auf dem Host bekannt sein. (Befehl: docker volume inspect tfvolume) Der Container verwendet die eingegeben Matrikelnummer als Namen. Es wird das Programm „Testframework“ im Pfad „tfvolume/test/“ ausgeführt. [Docker-Py 2016]

Mit der Methode start() wird der Container gestartet. [Docker-Py 2016]

Anschließend wird mit der Methode wait() auf die Beendigung des Containers gewartet. Die Skriptausführung wird in dieser Zeit blockiert. [Docker-Py 2016]

Mit der Methode remove\_container() wird der Container nach der Ausführung gelöscht. [Docker-Py 2016]

## 4.7 Einschränkungen

Das Testframework ermöglicht es, sowohl die Ein-Ausgabe des zu prüfenden Programms, als auch einzelne Funktionen zu testen. Falls einzelne Funktionen geprüft werden sollen, muss der Name der Funktion eindeutig bekannt sein. Zudem müssen die Deklarationen dieser Funktionen in Headern erfolgen. Die Namen der Header und Source Files ist jedoch egal.

Falls auf die Ein- und Ausgabe eines Programms geprüft werden soll, müssen die Ausgabestrings eindeutig bekannt sein. z.B. sind die Strings „Hallo Welt“ und „Hello World“ nicht identisch. Unit Test Frameworks führen genaue Vergleiche durch. Ein Vergleich auf Ähnlichkeit ist nicht vorgesehen. Durch Verwendung der `strstr()` Funktion könnte auf das Vorhandensein von Teilstrings geprüft werden, dies widerspricht jedoch dem Prinzip der Unit Tests.

Unter Windows schließt sich ein Konsolenfenster direkt nach der Ausführung des Programms. Aus diesem Grund wird häufig in Internetquellen und Literatur empfohlen am Ende des Programms `getchar()` aufzurufen um das Fenster offen zu lassen. Dies würde hier dazu führen, dass die Laufzeitumgebung das Programm als aufgehängt erkennen würde, falls keine Eingabe durch die Unit Tests erfolgt. Somit muss dies Unterlassen werden.

Das Testframework beinhaltet aktuell nur die Standardbibliothek von C. Falls Programme geprüft werden sollen, die weitere Bibliotheken verwenden, reicht es unter Umständen nicht aus, diese in das Image des Testcontainers zu installieren, da der Linkeraufruf aktuell nicht verändert werden kann.



## 4.8 Open Issues

Der Aufruf des Studentencontainers wird von einem Python Skript ausgeführt. Der Apache Webserver verwendet den User www-data. Dieser User hat nicht die nötigen Rechte um das Skript mit Docker Py aufzurufen.

Im Interaktiven Modus des Apache Containers kann das Python Skript manuell mit root Rechten aufgerufen werden. In diesem Fall funktioniert die Ausführung. Die Zeit hat nicht gereicht um die nötigen Zugriffsrechte für die einzelnen Dateien anzupassen. Ein Webserver sollte aus Sicherheitsgründen nicht mit root Rechten ausgeführt werden.

## 5 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein Testframework zu entwickeln, welches mit Hilfe von Unit Tests und Docker Virtualisierung C-Codes von Studenten prüft. Dies ist abgesehen von dem User Rechte Problem erfüllt.

### 5.1 Zusammenfassung der Arbeit

Der gesamte Entwicklungsprozess lässt sich in 3 Schritte einteilen

#### Formulierung der Anforderungen

Zuerst wurden die Anforderung an das Testframework formuliert. Dieses Testframework soll für einen C-Grundkurs verwendet werden. Einstiegsprogramme sind in der Regel simpel gehalten. Mit üblichen Testsystemen wie den Unit Tests lassen sie sich nicht richtig testen, da sie zu simpel für diese Testmethoden sind. Der erste Schritt eine Software zu prüfen ist es, diese zu erstellen. Dies wird durch den GCC Compiler realisiert. Dies ist zugleich ein statischer Softwaretest. Der GCC Compiler liefert eine detaillierte Ausgabe über eventuelle Fehler und gefährliche Codepassagen.

Da Einsteiger C Programme in der Regel nur aus der Main Funktion und Ein- und Ausgabe bestehen, war das Ziel diese in die Tests mit einzubeziehen. Dafür werden Unit Tests verwendet. Um die Standardstreams umzuleiten muss eine Laufzeitumgebung verwendet werden. Somit kann das Google Test Framework die Standardausgabe mit Unit Tests zu prüfen. Ebenfalls besteht bei Programmen immer das Risiko, dass sie in einer Endlosschleife hängen bleiben. Ein Anwender an einem PC kann die Ausführung beenden. Ein automatisiertes Testframework muss dies automatisch durchführen. Ebenso sind Speicherfehler ein häufig auftretender Fehler. Mit der Valgrind Toolsammlung steht dafür ein Testframework bereit. Um auf Speicherfehler prüfen zu können, wurde diese ebenfalls in das Testframework integriert.

### **Implementierung der Laufzeitumgebung**

Die Entwicklung der Laufzeitumgebung hat viel Zeit in Anspruch genommen. Da die Einarbeit in die Systemprogrammierung aufwändig war. Die Versuche die Umleitung der Standard Streams mit Pipes durchzuführen ist an der Pufferung gescheitert. Ebenso die Umleitung durch FIFOs. Somit musste ein Pseudoterminal verwendet werden. Durch Fachliteratur war dafür eine Einarbeitung nötig bis die Anwendung des Pseudoterminals erfolgreich war. Die Realisierung der Überwachung für den Aufhängeschutz hat ebenfalls eine weitere Vertiefung der Betriebssystemkenntnisse erfordert. Die Integration von Valgrind hat ebenfalls viel Zeit in Anspruch genommen. Da ausschließlich interaktive Programme getestet werden, ist es notwendig, dass sämtliche Eingaben für den Speichertest durchgeführt werden. Die Ursprüngliche Idee war, den Speichertest simultan zu den Eingabetests durchzuführen. d.h. die Unit Tests führen die Eingaben durch und prüfen die Ausgaben, gleichzeitig führt memcheck einen Speichertest durch. Dies ist nicht möglich, da Valgrind eine eigene Laufzeitumgebung verwendet. Diese ist nicht kompatibel zum Pseudoterminal. Dies hat dazu geführt, das das Programm für den Speichertest erneut ausgeführt wird und mit den gleichen Eingaben erneut ausgeführt wird.

### **Implementierung der Unit Tests**

Das Google Test Framework ist einfach zu verwenden. Für dieses Projekt wurde lediglich die Testausgabe angepasst, da diese Ausgaben in eine Datei geleitet werden müssen.

### **Ausführung aller Tests**

Im nächsten Schritt wurden alle Testmethoden kombiniert, so das eine ausführbare Anwendung sämtliche Tests durchführt. Die Schwierigkeit lag dadrin 2 Programme zu erstellen und Auszuführen. Im Fall der Unit Tests kommt hinzu, das das Programm mit Programmteilen erstellt werden müssen, die erst zur Laufzeit bekannt werden. Dafür wird der generated Header verwendet um Header einzubinden die erst noch nicht bekannt sind. Bei den Unit Tests bestand zudem das Problem wie man Object Dateien mit dem Linker verbindet in denen sich 2 Main Funktionen befinden. Eine dieser Main Funktionen befindet sich zudem in einer unbekanntem Object Datei. Dieses Problem wird mit dem Programm Strip behoben.

## Erstellung der Webseite

Die Erstellung der Webseite ist nicht komplett fertig geworden. Dies liegt an einem User Rechte Problem. Für Docker stehen mehrere inoffizielle API Bibliotheken zur Verfügung, die jedoch teilweise nicht richtig funktionieren oder nur unzureichend dokumentiert sind. Zuerst wurde es versucht eine der 3 PHP Bibliotheken zu verwenden. Diese benötigen weitere Tools zur Installation und Ausführung. Es hat viel Zeit beansprucht diese zu testen. Mit keiner dieser Bibliotheken konnte ein zufriedenstellendes Ergebnis erreicht werden. Die Wartbarkeit eines Programmes hängt von den verwendeten Bibliotheken ab. Da die Wartbarkeit bei fremden Bibliotheken nicht gewährleistet werden kann, wurde als Alternative die Python Bibliothek Docker-Py verwendet. Diese Bibliothek ist gut dokumentiert und einfach anzuwenden.

## 5.2 Ausblick

Falls dieser Webserver weiterverwendet werden soll, muss das Rechte Problem behoben werden. Dies liegt vermutlich an den Ausführrechten von einer oder mehreren Dateien. Ebenso könnte eine Authentifizierung durchgeführt werden.

Diese Probleme könnten auch dadurch gelöst werden, dass das Testframework direkt an Moodle angebunden wird. Moodle bietet eine API Schnittstelle für eigene Module<sup>24</sup>

Eine weitere Idee wäre eine schönere Interpretation der Ausgabe. Aktuell schreiben der Compiler, der TestEventListener des Testrunners und Valgrind in eine Datei. Diese Ausgaben könnten durch ein weiteres Modul im Testframework interpretiert werden und in ein einheitliches Darstellungsformat umgewandelt werden. Dieser Interpreter könnte vor der cleanup() Funktion des Testframeworks aufgerufen werden.

---

<sup>24</sup> [https://docs.moodle.org/dev/Core\\_APIs](https://docs.moodle.org/dev/Core_APIs)

## 6 Abkürzungsverzeichnis

API – Application Programming Interface  
APT – Advanced Packaging Tool  
CPU – Central Processing Unit  
E/A – Eingabe/Ausgabe  
FIFO – First In First Out  
GCC – GNU C Compiler  
GNU – GNU is not Unix  
HTTP – Hypertext Transfer Protocol  
IEEE – Institute of Electrical and Electronics Engineers  
IPC – Interprocess Communication  
ISR – Interrupt Service Routine  
LTS – Long Term Support  
LXC – Linux Container  
PAGS – Program Assignment Grading System  
PHP – Hypertext Preprocessor  
PID – Process Identifier  
POSIX – Portable Operating System Interface  
PTY – Pseudo Teletype  
SUT – System Under Test  
TDD – Test Driven Development  
TTY – Teletype  
UML – Unified Modeling Language  
URL – Uniform Resource Locator  
XML – Extensible Markup Language

## 7 Abbildungsverzeichnis

Abbildung 1: Visualisierung des Whitebox Testing [Spillner 2014].....	4
Abbildung 2: Visualisierung des Blackbox Testings [Spillner 2014].....	5
Abbildung 3: Visualisierung des Unit Test Prinzip[Hamill 2009].....	6
Abbildung 4: schematische Übersicht der xUnit Architektur [Hamill 2009].....	7
Abbildung 5: Visualisierung der Klasse TestCase in UML [Hamill 2009].....	8
Abbildung 6: Visualisierung der Klasse TestRunner in UML [Hamill 2009].....	9
Abbildung 7: Ausgabe des Testrunners.....	10
Abbildung 8: Visualisierung der Klasse TestFixture in UML [Hamill 2009].....	11
Abbildung 9: Visualisierung der Klasse TestSuite in UML [Hamill 2009].....	12
Abbildung 10: Visualisierung der Klasse TestResult in UML [Hamill 2009].....	13
Abbildung 11: Ausgabe des Testrunners mit Testsuite.....	17
Abbildung 12: Ausgabe des Gtest Testrunners.....	24
Abbildung 13: Visualisierung der Docker Architektur [Turnbull 2014].....	28
Abbildung 14: Aufruf des Docker Clients in der Konsole.....	29
Abbildung 15: Visualisierung der Erstellung eines Docker Images.....	33
Abbildung 16: Starten eines Docker Containers mit einem Hostpfad als Volume.....	34
Abbildung 17: Erstellung eines benannten Docker Volumes.....	34
Abbildung 18: Verwendung eines benannten Docker Volumes.....	34
Abbildung 19: Veranschaulichung der Informationen eines benannten Docker Volumes.....	35
Abbildung 20: Visualisierung der 3 Prozesszustände [Tanenbaum 2009].....	41
Abbildung 21: Darstellung eines Pipes zwischen 2 Prozessen [Wolf 2006].....	45
Abbildung 22: Darstellung eines FIFOs zwischen Clients und einem Server Prozess [Wolf 2006].	46
Abbildung 23: Veranschaulichung der Anwendung eines Semaphors.....	48
Abbildung 24: Visualisierung der PTY Schnittstelle [Stevens 2005].....	50
Abbildung 25: Darstellung des Programmablaufs mit Docker Containern.....	56
Abbildung 26: Darstellung des Ablaufs eines Testdurchgangs.....	58
Abbildung 27: Vereinfachte Darstellung der create_test_program() Funktion.....	60
Abbildung 28: Vereinfachte Darstellung der Funktion execute_unittests().....	63
Abbildung 29: Darstellung der Ausgabe des TestframeworkListener.....	67
Abbildung 30: Visualisierung des Pseudoterminals.....	72
Abbildung 31: Visualisierung der Weiterleitung der Standardstreams.....	73
Abbildung 32: Visualisierung der Speichertest Durchführung.....	75
Abbildung 33: Darstellung der Webseite des Testframeworks.....	76
Abbildung 34: Darstellung der Docker Container und Docker Volumes.....	77
Abbildung 35: Darstellung der Lösung von Student A Aufgabe 1.....	100
Abbildung 36: Darstellung der Ausgabe von Student B Aufgabe 1.....	101
Abbildung 37: Darstellung der Ausgabe von Student A Aufgabe 2.....	103
Abbildung 38: Darstellung der Ausgabe von Student B Aufgabe 2.....	105
Abbildung 39: Darstellung der Ausgabe von Student A Aufgabe 3.....	110
Abbildung 40: Darstellung der Ausgabe von Student B Aufgabe 3.....	112

## 8 Quellcodeverzeichnis

Listing 1: Implementierung eines TestCases in CPPUnit.....	8
Listing 2: Anwendung des Testrunners in CPPUnit.....	10
Listing 3: Der Header TestSummen.h des Testfixtures.....	14
Listing 4: Implementierung des TestFixtures TestSummen.....	15
Listing 5: Implementierung des Testrunners mit TestSuite.....	16
Listing 6: Implementierung eines Testcases im Gtest Framework.....	22
Listing 7: Implementierung eines TestFixtures im Gtest Framework.....	22
Listing 8: Darstellung der Standard Main Funktion mit Standard Testrunner des Gtest Frameworks	23
Listing 9: Darstellung eines Dockerfiles.....	31
Listing 10: Darstellung des Python Skripts container.py.....	80
Listing 11: Darstellung von test_helloworld.cpp.....	99
Listing 12: Darstellung von helloworld.c von Student A.....	100
Listing 13: Darstellung des Hello World Programm von Student B.....	101
Listing 14: Darstellung von test_fragenamen.cpp.....	102
Listing 15: Darstellung der Lösung von Student A.....	103
Listing 16: Darstellung der Lösung von Student B.....	104
Listing 17: Darstellung von test_input.cpp.....	106
Listing 18: Darstellung von test_rechner.cpp.....	107
Listing 19: Header und Source File der Funktion summe(int int).....	108
Listing 20: Darstellung von rechner.c von Student A.....	109
Listing 21: Header und Source Datei mit sämtlichen Funktionen des Taschenrechners.....	111
Listing 22: Veranschaulichung des Taschenrechners von Student B.....	112

## 9 Tabellenverzeichnis

Tabelle 1: Übersicht über die Googletest Basisvergleiche[Google 2015].....	19
Tabelle 2: Übersicht über die Googletest Binärvergleiche[Google 2015].....	19
Tabelle 3: Übersicht über die Googletest Gleitkomma Vergleiche[Google 2015].....	20
Tabelle 4: Übersicht über die Googletest Stringvergleiche [Google 2015].....	20
Tabelle 5: Übersicht über die Googletest Ausnahmenvergleiche.....	21
Tabelle 6: Übersicht über die Standard Filedeskriptoren[Wolf 2006].....	36
Tabelle 7: Übersicht über die Standard Streams[Wolf 2006].....	37
Tabelle 8: Übersicht über einige Linux Signale [Stevens 2005][Wolf 2006].....	38



## 10 Quellenverzeichnis

Andreas Spillner, Thomas Roßner, Mario Winter, Tilo Linz (2014)  
Praxiswissen Softwaretest – Testmanagement  
dpunkt Verlag, 4. Auflage

Jörg Schäuffele, Thomas Zurawka (2013)  
Automotive Software Engineering Grundlagen, Prozesse, Methoden und Werkzeuge  
Vieweg Verlag, 1. Auflage

Paul Hamill (2009)  
Unit Test Frameworks A language Independent Overview  
O'Reilly Verlag, 1. Auflage

Manfred Dausmann, Ulrich Bröckl, Joachim Goll (2007)  
C als erste Programmiersprache – Vom Einsteiger zum Profi  
Teubner Verlag, 6. Auflage

Andrew S. Tanenbaum (2009)  
Moderne Betriebssysteme  
Pearson Studium Verlag 3. Auflage

W. Richard Stevens, Stephen A. Rago (2005)  
Advanced Programming in the Unix Environment  
Addison-Wesley Longman Verlag, 2. Auflage

Jürgen Wolf (2006)  
Linux-Unix Programmierung Das umfassende Handbuch  
Rheinwerk Verlag 2. Auflage

Chris Rupp & die SOPHISTen (2014)  
Requierelements-Engineering und -Management  
Hanser Verlag 6. Auflage

Allen Holub (1988)  
C Für Kenner Ausgesuchte C-Softwaretools  
Markt und Technik Verlag AG (Auflage unbekannt)

David Evans, David Larochelle (2002)  
Improving Security Using Extensible Lightweight Static Analysis  
IEEE<sup>25</sup> Software, Paper University of Virginia

25 IEEE – Institute of Electrical and Electronics Engineers

Mark Vogelsberger (2014) abgerufen am 27.05.2016

Linux Magazin Security Blog

<http://www.linux-magazin.de/Blogs/Insecurity-Bulletin/Linux-Kernel-Race-Condition-laesst-Befehle-ausfuehren>

Viva64 PVS Studio Documentation (2016) Abgerufen am 27.05.2016

<http://www.viva64.com/en/d/>

Covertly Scan Static Analysis (2016) Abgerufen am 27.05.2016

<https://scan.coverity.com/>

Eric Sommerlade, Michael Feathers, Jerome Lacoste, Baptiste Lepillieur et al. (2016)

CppUnit Documentation Version 1.11.0 Abgerufen am 27.05.2016

<http://cppunit.sourceforge.net/doc/cvs/index.html>

James Turnbull (2014)

The Docker Book (Ebook Verlag und Auflage nicht vorhanden)

Google Test Framework (2015) Abgerufen am 27.05.2016

Dokumentation des Google Test Framework Google Inc.

<https://github.com/google/googletest>

Valgrind Organisation (2016) Abgerufen am 27.05.2016

<http://valgrind.org/>

Docker Documentation (2016) Abgerufen am 27.05.2016

<https://docs.docker.com/>

Docker Quellcode (2014) Abgerufen am 27.05.2016

<https://github.com/docker/docker/blob/670c8696a29825b23208496bd4d8e88b5faa7773/builders/dispatchers.go#L77>

Apache Foundation Abgerufen am 27.05.2016

<https://httpd.apache.org/>

Richard M. Stallman and the GCC Developer Community (2003)

Using the GNU Compiler Collection For GCC version 5.3.0 GNU Press,  
Free Software Foundation

Strip Documentation – GNU Binary Utilities Abgerufen am 27.05.2016

<https://sourceware.org/binutils/docs-2.23/binutils/strip.html>

Python 3.5.1 Documentation Abgerufen am 27.05.2016

<https://docs.python.org/3/>

Neil Brown (2010) Abgerufen am 27.05.2016  
Ghost of Unix Past- Artikel auf lwn.net  
<http://lwn.net/Articles/411845/>

Erkan Yanar (2016)  
Ship IT  
i'X Magazin 03/2016 heise Verlag

Udo Seidel (2016)  
Build, Ship, Run - Das Docker-Universum im dritten Lebensjahr  
i'X Magazin 04/2016 heise Verlag

Martin Loschwitz (2014)  
Alteingesessen - Die Etablierten der Linux Containerlandschaft  
Linux Magazin 11/2015 Computec Media GmbH

Travis CI GmbH (2016) Abgerufen am 30.04.2016  
Dokumentation TravisCI  
<https://docs.travis-ci.com/>

Mustafa Akin (2016) Abgerufen am 30.04.2016  
PAGS Homepage  
<http://pags.cs.bilkent.edu.tr/>

Docker-Py (2016) Abgerufen am 30.04.2016  
Documentation of docker-dy  
<https://docker-py.readthedocs.io/en/stable/>

Ubuntu (2016) Abgerufen am 01.05.2016  
<http://www.ubuntu.com/>

## 11 Anhang

In diesem Abschnitt werden die Installation und die Verwendung des Testframeworks erläutert. Die Beschreibung der Anwendungen erfolgt an Beispielen. Dies ist gleichzeitig auch die Verifikation. Für sämtliche Kommandos wird davon ausgegangen, dass diese im für die Datei verwendeten Verzeichnis aus aufgerufen werden.

### 11.1 Installationsanleitung

#### 11.1.1 Erstellung der Komponenten

Für den Betrieb dieses Testframeworks muss Docker installiert sein. Die Erstellung und Verwendung dieser Programme ist nur unter Linux (Kubuntu 15.10) getestet. Bei Verwendung von MacOS oder Windows können diese Programme wegen den Systemaufrufen vermutlich nicht ausgeführt werden.

Das Testframework besteht aus drei Projekten, dem eigentlichen Testframework, dem Googletest Framework und der Laufzeitumgebung. Dafür steht jeweils ein eigenes Makefile zur Verfügung.

Die drei Projekte befinden sich auf der CD im Ordner src. Die Targets der Projekte lauten:

- Testframework für das eigentliche Testframework
- Gtest.a für das Google Testframework mit eigenem TestEventListener
- runtime für die Laufzeitumgebung.

Diese 3 Projekte können durch den Aufruf von make mit dem entsprechenden Target erstellt werden.

Nach der Erstellung müssen die folgenden Dateien in das Volume testframework kopiert werden. Dafür steht auf der CD der Ordner Volume bereit.

- Das Programm Testframework in den Ordner Volume/test
- Die Bibliothek gtest.a in den Ordner Volume/test/Unittests
- Die Bibliothek libruntime.a mit allen Headern in den Ordner Volume/test/runtime

In dem beiliegenden Ordner Volume auf der CD sind diese Dateien bereits eingefügt.

## 11.1.2 Anlegen des Volumes tfvolume

Das Volume tfvolume wird ausschließlich zwischen dem Container der Webseite und dem Testcontainer verwendet. Das Volume muss unter Docker zuerst erstellt werden.

**Befehl:** `docker volume create --driver local --name tfvolume`

Anschließend muss der absolute Pfad dieses Volumes ermittelt werden.

**Befehl:** `docker volume inspect tfvolume`

Der absolute Pfad erscheint unter dem Eintrag „Mountpoint“. Dieser muss in das Python Skript `container.py` in der Funktion `create_host_config(binds={pfad/des/volumes = { ...` gesetzt werden. Dieses Python Skript befindet sich auf der CD im Pfad `Webserver/src`.

## 11.1.3 Erstellen der Images

Die Images für den Webserver müssen erstellt werden. Dafür stehen Dockerfiles bereit. Für die Testausführung wird ein Container gestartet. Das Image muss vor dem Aufruf auf dem System vorhanden sein.

### 11.1.3.1 Erstellen des Testimages

Dieses Image ist die Basis für die Tests. Das Dockerfile befindet sich auf der CD im Ordner `Testimage`. Aus diesem Dockerfile kann das Image erstellt werden.

**Befehl:** `docker build -t student .`

Der Name dieses Images muss `student` sein, damit das Python Script dieses Image verwenden kann.

### 11.1.3.2 Erstellen des Webserver Images

Das Dockerfile für den Webserver befindet sich auf der CD im Ordner `Webserver`.

**Befehl:** `docker build -t webserver .`

Der Name dieses Images kann frei gewählt werden. Jedoch muss dieser dann auch beim Starten des Webservers verwendet werden.

### 11.1.3.3 Einfügen von Unit Tests

Unit Tests müssen in cpp Dateien vorliegen. Diese müssen in dem Ordner Volume/Unittets/<Aufgabennummer> gespeichert werden. Aktuell ist die Anzahl der Aufgaben zu Testzwecken auf drei festgelegt. Um weitere Aufgaben hinzuzufügen, muss unter Volume/Unittests ein neuer Ordner mit der Aufgabennummer angelegt werden. Zudem muss die Aufgabe in der Auswahlliste in der Datei index.php im Verzeichnis Webserver/src angepasst werden. Dazu muss in dem „form“ Element ein weiterer Eintrag analog zu den vorhandenen angelegt werden.

### 11.1.3.4 Starten des Webservers

Der Webserver kann aus dem Image „webserver“ gestartet werden.

**Befehl:** docker run -i -t -p 80:80  
-v /var/run/docker.sock : /var/run/docker.sock  
-v /pfad/zu/CD/Volume : /testframework  
-v tfvolume: /tfvolume webserver

Durch die Flags -i und -t wird der Container interaktiv gestartet. Dies ist aktuell notwendig um das Python Skript aufzurufen.

Das Flag -p 80:80 binden den Port 80 des Hosts an den Port 80 des Containers. Dies ist notwendig um die Webseite über http im Browser abzurufen.

-v /var/run/docker.sock : /var/run/docker.sock wird benötigt um die Docker API im Container verfügbar zu machen. Dies ist der Standard Pfad der Docker API.

-v /pfad/zu/CD/Volume : /testframework bindet das Volume im Container unter dem Ordner testframework ein. Dieser Pfad muss absolut sein und an den jeweiligen Pfad des Volumes angepasst werden.

-v tfvolume: /tfvolume bindet das Volume tfvolume in den Container ein.

Nach erfolgreichem Aufruf ist man als User root im Container eingeloggt. Nun muss der Apache Webserver gestartet werden.

**Befehl:** etc/init.d/apache2 start

Unter Umständen müssen die Lese- und Schreib- und Ausführrechte für die Volumes auf dem Host und im Container mittels chmod a+rwx gesetzt werden.

Auf dem Host kann die Webseite nun mit dem Browser unter 127.0.0.1 aufgerufen werden.

### 11.1.3.5 Anmerkungen

Für die Ausführung des python Skripts fehlen dem User www-data die nötigen Rechte. In der Datei student.php sind deswegen die Befehle `rename($filename, $newname)` zum umbenennen und kopieren von result.txt nach `testframework/Results/<matrikelnummer>/aufgabe_<aufgabenummer>.txt` und `system(rm -rf /tfvolume/*)` zum löschen sämtlicher Dateien im Volume tfvolume nach der Ausführung. Diese Schritte müssen manuell durchgeführt werden.

Das Python Skript container.py befindet sich im Ordner /var/www/html des Webseitencontainers. Dieses Skript muss manuell gestartet werden.

**Befehl:** `python container.py <matrikelnummer>`

Die Datei result.txt befindet sich nach dem Aufruf im Verzeichnis /tfvolume/test und beinhaltet nach dem Aufruf des Skripts die Testausgabe.

## 11.2 Erstellung von Unit Tests und Verifikation

In diesem Abschnitt wird die Erstellung von Unit Tests erläutert. Dies dient zudem als Verifikation des Testframeworks. Es werden drei Programmieraufgaben verwendet, die von zwei Studenten durchgeführt wurden.

Student A besitzt die Matrikelnummer 12345

Student B besitzt die Matrikelnummer 98765

Die drei Programmieraufgaben lauten:

1. Das Hello World Programm
2. Ein Programm, das den Anwender nach seinem Namen fragt. Nach der Eingabe des Namens soll die Ausgabe „Hallo <Name>“ erfolgen.
3. Ein Taschenrechner Programm. Dieses Programm soll über die Funktionen `summe(int, int)`, `differenz(int, int)`, `produkt(int, int)` und `quotient(int, int)` verfügen. Dieses Programm soll den Anwender nach zwei Zahlen a und b und der auszuführenden Operation fragen. Anschließend soll die Lösung in der Form  $a+b=ergebnis$  erfolgen.



## 11.2.1 Unit Tests und Ausgaben für Aufgabe 1

Für das Hello World Programm wird das folgende TestFixture verwendet:

```
1 #include "gtest/gtest.h"
2 #include "../runtime/runtime.h"
3 #include "../runtime/valgrind_check.h"
4
5 class HelloWorldTest : public ::testing::Test
6 {
7
8     virtual void SetUp()
9     {
10         runtime_setup("");
11     }
12     virtual void TearDown()
13     {
14         runtime_tearardown();
15     }
16 };
17
18 TEST_F(HelloWorldTest, testHelloWorld)
19 {
20     EXPECT_STREQ("Hello World\r\n", readFromProgram());
21     do_valgrind_check();
22 }
```

*Listing 11: Darstellung von test\_helloworld.cpp*

Dieses Listing stellt das TestFixture für Aufgabe 1 dar. Dafür werden die Header der Laufzeitumgebung und für den Speichercheck eingebunden.

Das TestFixture hat den Namen HelloWorldTest und erbt von der Google Testframework Klasse Test. Die Methoden SetUp() und TearDown() werden für die Initialisierung bzw. Beendigung der Laufzeitumgebung verwendet.

Für dieses TestFixture wird der TestCase testHelloWorld definiert. Dies wird durch das Makro TEST\_F(HelloWorldTest, testHelloWorld) realisiert.

In diesem TestFixture wird die Ausgabe des zu prüfenden Programms mit dem String „Hello World\r\n“ verglichen. Anschließend wird mit der Funktion do\_valgrind\_check() der Speichertest durchgeführt.

### 11.2.1.1 Lösung und Ausgabe für Student A

Student A hat die folgende Lösung für Aufgabe 1 über die Webseite hochgeladen:

```
1 #include <stdio.h>
2
3 void main(void)
4 {
5     printf("Hello World\n");
6 }
7
```

*Listing 12: Darstellung von helloworld.c von Student A*

```
Test for 12345 Aufgabe 1 01.05.2016 - 12.03
tested files : helloworld.c

../helloworld.c:3:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(void)
    ^
Creation of testprogram finished

Start Tests
Test: HelloWorldTest.testHelloWorld started
Valgrind check started

Valgrind check stopped

Programm finished successfully

Test: HelloWorldTest.testHelloWorld finished

Unittests finished
1 tests in: 1 TestCases ran
Tests succeeded: 1
Tests failed: 0
```

*Abbildung 35: Darstellung der Lösung von Student A Aufgabe 1*

Dieses Listing zeigt das Hello World Programm von Student A.

Diese Abbildung veranschaulicht die Ausgabe des Testframeworks für Student A. Der Compiler hat die Warnung ausgegeben, dass die Main Funktion nicht den Rückgabewert int verwendet. Das Programm wurde jedoch erstellt. Zu Erkennen ist auch, dass der Unit Test erfolgreich durchgeführt wurde. Ebenfalls wurde der Speichertest erfolgreich ausgeführt. (Keine Ausgabe zwischen Valgrind check started/stopped)

### 11.2.1.2 Lösung und Ausgabe für Student B

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World");
6
7     //endless loop, program will hang up
8     for(;;);
9 }
```

*Listing 13: Darstellung des Hello World Programm von Student B*

Dieses Listing zeigt die Lösung von Student B. Die Ausgabe von Hello World erfolgt ohne Zeilenumbruch. Zudem wird sich dieses Programm durch eine Endlosschleife aufhängen.

```
Test for 98765 Aufgabe 1 01.05.2016 - 12.40
tested files : helloworld_absturz.c

Creation of testprogram finished

Start Tests
Test: HelloWorldTest.testHelloWolrd started
Failure in file: Unittests/test_helloworld.cpp Line: 20
    Expected: "Hello World\r\n"
To be equal to: readFromProgram()
    Which is: NULL

Programm freezes and was abortet

Test: HelloWorldTest.testHelloWorld finished

Unittests finished
1 tests in: 1 TestCases ran
Tests succeeded: 0
Tests failed: 1
```

*Abbildung 36: Darstellung der Ausgabe von Student B Aufgabe 1*

Die obige Abbildung stellt die Ausgabe von Student B dar. Der Vergleich auf die Ausgabe ist fehlgeschlagen, da keine Ausgabe erfolgt ist. (Die Puffer werden bei Beendigung eines Programms automatisch geleert, dies tritt hier aber nie ein) Zudem erfolgt die Meldung der Laufzeitumgebung, dass die Programmausführung abgebrochen wurde. Es sind keine Meldungen des Speichertests erfolgt, da dieser nicht aufgerufen wurde.

## 11.2.2 Unit Tests und Ausgaben für Aufgabe 2

Für Aufgabe 3 wird die Klasse FrageNamenTest verwendet.

```
1 #include "gtest/gtest.h"
2 #include "../runtime/runtime.h"
3 #include "../runtime/valgrind_check.h"
4
5 class FrageNamenTest : public ::testing::Test
6 {
7
8     virtual void SetUp()
9     {
10         runtime_setup("");
11     }
12     virtual void TearDown()
13     {
14         runtime_teardown();
15     }
16 };
17
18 TEST_F(FrageNamenTest, testName)
19 {
20     EXPECT_STREQ("Bitte geben sie ihren Namen ein\r\n", readFromProgram());
21     writeToProgram("David\n");
22     EXPECT_STREQ("\nHallo David\r\n", readFromProgram());
23
24     do_valgrind_check();
25 }
26
27 TEST_F(FrageNamenTest, testNameLang)
28 {
29     EXPECT_STREQ("Bitte geben sie ihren Namen ein\r\n", readFromProgram());
30     writeToProgram("Daaaaaviiiiid\n");
31     EXPECT_STREQ("\nHallo Daaaaaviiiiid\r\n", readFromProgram());
32
33     do_valgrind_check();
34 }
35
```

Listing 14: Darstellung von test\_fragenamen.cpp

Dieses Listing veranschaulicht die Unit Tests für Aufgabe 2. Es werden 2 TestCases definiert (testName und testNameLang). Diese prüfen zwei mal die Ausgabe des zu testenden Programms. Zwischendurch wird durch die Funktion writeToProgram() eine Eingabe an das zu prüfende Programm übermittelt.

### 11.2.2.1 Lösung und Ausgabe für Student A

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char buf[80];
6     printf("Bitte geben sie ihren Namen ein\n");
7     scanf("%s", &buf);
8     printf("Hallo %s\n", buf);
9 }
10
```

Listing 15: Darstellung der Lösung von Student A

Dieses Listing veranschaulicht die Lösung von Student A Aufgabe 2. Dieses Programm gibt einen String aus, liest einen String ein und gibt wieder einen String aus.

```
[test for 12345 Aufgabe 2 01.05.2016 - 13.33
tested files : fragenamen.c

../fragenamen.c: In function 'main':
../fragenamen.c:7:11: warning: format '%s' expects argument of type 'char *', but argument 2 has type 'char (*)
[80]' [-Wformat=]
    scanf("%s", &buf);
          ^
Creation of testprogram finished

Start Tests
Test: FrageNamenTest.testName started

Programm finished successfully
Test: FrageNamenTest.testName finished

Test: FrageNamenTest.testNameLang started
Valgrind check started

Valgrind check stopped

Programm finished successfully
Test: FrageNamenTest.testNameLang finished

Unittests finished
2 tests in: 1 TestCases ran
Tests succeeded: 2
Tests failed: 0
```

Abbildung 37: Darstellung der Ausgabe von Student A Aufgabe 2

Die obige Abbildung stellt die Ausgabe des Tests für Student A Aufgabe 2 dar. Es ist erkenntlich, dass alle Test erfolgreich ausgeführt wurden.

### 11.2.2.2 Lösung und Ausgabe für Student B

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char *buf;
6     buf = malloc(8);
7     printf("Bitte geben sie ihren Namen ein\n");
8     scanf("%s", buf);
9     printf("Hallo %s\n", buf);
10 }
11
```

*Listing 16: Darstellung der Lösung von Student B*

Dieses Listing veranschaulicht die Lösung von Student B für Aufgabe 2. Hier wird ein Puffer mit `malloc()` allokiert. Dieser ist für die Eingabe zu klein und wird nicht mehr freigegeben.

```

Test for 98765 Aufgabe 2 01.05.2016 - 13.39
tested files : fragenamen.c

../fragenamen.c: In function 'main':
../fragenamen.c:6:11: warning: implicit declaration of function 'malloc' [-Wimplicit-function-declaration]
    buf = malloc(8);
          ^
../fragenamen.c:6:11: warning: incompatible implicit declaration of built-in function 'malloc'
../fragenamen.c:6:11: note: include '<stdlib.h>' or provide a declaration of 'malloc'
Creation of testprogram finished

Start Tests
Test: FrageNamenTest.testName started

Programm finished successfully

Test: FrageNamenTest.testName finished

Test: FrageNamenTest.testNameLang started
Valgrind check started

==27== Invalid write of size 1
==27==    at 0x4E8D774: _IO_vfscanf (vfscanf.c:1073)
==27==    by 0x4E9C9A8: __isoc99_scanf (isoc99_scanf.c:37)
==27==    by 0x40062B: main (in /tfvolume/test/a.out)
==27== Address 0x51e0048 is 0 bytes after a block of size 8 alloc'd
==27==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==27==    by 0x400607: main (in /tfvolume/test/a.out)
==27==
==27== Invalid write of size 1
==27==    at 0x4E8D81A: _IO_vfscanf (vfscanf.c:1158)
==27==    by 0x4E9C9A8: __isoc99_scanf (isoc99_scanf.c:37)
==27==    by 0x40062B: main (in /tfvolume/test/a.out)
==27== Address 0x51e004d is 5 bytes after a block of size 8 alloc'd
==27==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==27==    by 0x400607: main (in /tfvolume/test/a.out)
==27==
==27== Invalid read of size 1
==27==    at 0x4E7FDCC: vfprintf (vfprintf.c:1642)
==27==    by 0x4E85D88: printf (printf.c:33)
==27==    by 0x400641: main (in /tfvolume/test/a.out)
==27== Address 0x51e0048 is 0 bytes after a block of size 8 alloc'd
==27==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==27==    by 0x400607: main (in /tfvolume/test/a.out)
==27==
==27== Invalid read of size 8
==27==    at 0x4EBA7AF: __GI_memcpy (memcpy.S:107)
==27==    by 0x4EA9906: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1312)
==27==    by 0x4E7FD86: vfprintf (vfprintf.c:1642)
==27==    by 0x4E85D88: printf (printf.c:33)
==27==    by 0x400641: main (in /tfvolume/test/a.out)
==27== Address 0x51e0045 is 5 bytes inside a block of size 8 alloc'd
==27==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==27==    by 0x400607: main (in /tfvolume/test/a.out)
==27==
==27== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==27==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==27==    by 0x400607: main (in /tfvolume/test/a.out)
==27==
Valgrind check stopped

Programm finished successfully
    
```

Abbildung 38: Darstellung der Ausgabe von Student B Aufgabe 2

Diese Abbildung zeigt die Ausgabe von Student Bs Lösung für Aufgabe 2. Diese Abbildung ist nicht komplett, da die Valgrind Ausgabe trotz quiet Flag sehr umfangreich ist. (Auf der CD ist die Datei vollständig vorhanden) Der Speichertest hat festgestellt, dass die Eingabe vom zweiten Unit Test zu einem Pufferüberlauf geführt hat. Ebenso hat memcheck festgestellt, dass der Speicher nicht mehr freigegeben wird.

### 11.2.3 Unit Tests und Ausgaben für Aufgabe 3

Für Aufgabe 3 werden zwei Unit Test Dateien verwendet.

```
1 #include "gtest/gtest.h"
2 #include "../runtime/runtime.h"
3 #include "../runtime/valgrind_check.h"
4
5 class InputTest : public ::testing::Test
6 {
7
8     virtual void SetUp()
9     {
10         runtime_setup("");
11     }
12     virtual void TearDown()
13     {
14         runtime_teardown();
15     }
16 };
17
18 TEST_F(InputTest, testAddition)
19 {
20     EXPECT_STREQ("Bitte 2 Zahlen eingeben\r\n", readFromProgram());
21     writeToProgram("1 2\n");
22     EXPECT_STREQ("\nBitte auswählen: +-*/", readFromProgram());
23     writeToProgram("+\n");
24     EXPECT_STREQ("\n1+2 = 3\r\n", readFromProgram());
25 }
26
27 TEST_F(InputTest, testSubtraktion)
28 {
29     EXPECT_STREQ("Bitte 2 Zahlen eingeben\r\n", readFromProgram());
30     writeToProgram("8 5\n");
31     EXPECT_STREQ("\nBitte auswählen: +-*/", readFromProgram());
32     writeToProgram("-\n");
33     EXPECT_STREQ("\n8-5 = 3\r\n", readFromProgram());
34
35     do_valgrind_check();
36 }
```

Listing 17: Darstellung von test\_input.cpp

Dieses Listing stellt das TestFixture für Aufgabe 3 dar. Dieses Testfixture führt zwei Programmdurchläufe mit unterschiedlichen Eingaben durch. Dafür werden wieder die Funktionen readFromProgram() und writeToProgram() verwendet.



```
1 #include "gtest/gtest.h"
2 #include "../generated_header.h"
3
4 TEST(Rechner, SummeTest)
5 {
6     ASSERT_EQ(3, summe(1,2));
7     ASSERT_EQ(9, summe(4,5));
8     ASSERT_EQ(10, summe(3,7));
9     ASSERT_EQ(100, summe(50,50));
10 }
11
12 TEST(Rechner, DifferenzTest)
13 {
14     ASSERT_EQ(1, differenz(2,1));
15     ASSERT_EQ(3, differenz(8,5));
16     ASSERT_EQ(20, differenz(100,80));
17 }
18
19 TEST(Rechner, Produkttest)
20 {
21     ASSERT_EQ(1, produkt(1,1));
22     ASSERT_EQ(0, produkt(0,4711));
23     ASSERT_EQ(10, produkt(2,5));
24 }
25 }
26
27 TEST(Rechner, QuotientTest)
28 {
29     ASSERT_EQ(1, quotient(1,1));
30     ASSERT_EQ(5, quotient(20,4));
31     ASSERT_EQ(7, quotient(49,7));
32 }
--
```

*Listing 18: Darstellung von test\_rechner.cpp*

Dieses Listing zeigt den TestCase Rechner. Für diesen Testcase werden vier Tests definiert. Diese führen jeweils drei Vergleiche auf die Funktionen `summe(int, int)`, `differenz(int,int)`, `produkt(int,int)` sowie `quotient(int, int)` durch. Zu beachten ist hierbei, dass der generated Header eingebunden wird, damit die TestCases Zugriff auf die unbekannt Header erhalten.

### 11.2.3.1 Lösung und Ausgabe für Student A

Student A hat eine Datei rechner.c mit der Main Funktion und der Programmlogik, sowie vier Header – Source Paare, die jeweils eine Funktion des Taschenrechners definieren bzw. deklarieren über die Webseite hochgeladen.

```
 /
 8 #ifndef SUMME_H_
 9 #define SUMME_H_
10
11 int summe(int a, int b);
12
13
14 #endif /* SUMME_H_ */
15
```

```
 7 #include "summe.h"
 8
 9 int summe(int a, int b)
10 {
11     return a+b;
12 }
13
```

*Listing 19: Header und Source  
File der Funktion summe(int int)*

Dieses Listing veranschaulicht den Header und die Source Datei der Funktion summe(int,int). Die Header und Source Dateien der anderen drei Funktionen sind ähnlich wie für diese Funktion und werden wegen der Übersichtlichkeit hier nicht dargestellt.

```
8 #include <stdio.h>
9 #include "summe.h"
0 #include "differenz.h"
1 #include "produkt.h"
2 #include "quotient.h"
3
4 void main(void)
5 {
6     int a,b;
7     char c, tmp;
8
9     printf("Bitte 2 Zahlen eingeben\n");
0     if(!scanf("%d %d", &a, &b))
1     {
2         printf("Keine korrekte Eingabe\n");
3         return;
4     }
5     fflush(stdin);
6     printf("Bitte auswählen: +-*/");
7     tmp = getchar();
8     c = getchar();
9
0     switch(c)
1     {
2         case '+':
3             printf("%d+%d = %d\n", a,b, summe(a,b));
4             break;
5
6         case '-':
7             printf("%d-%d = %d\n", a,b, differenz(a,b));
8             break;
9
0         case '*':
1             printf("%d*%d = %d\n", a,b, produkt(a,b));
2             break;
3
4         case '/':
5             printf("%d/%d = %d\n", a,b, quotient(a,b));
6             break;
7
8         default:
9             printf("Keine Operation gewählt\n");
0     }
1 }
```

Listing 20: Darstellung von rechner.c von Student A

Dieses Listing veranschaulicht die Datei rechner.c mit der Main Funktion und der Programmlogik des Taschenrechners.

```
Test for 12345 Aufgabe 3 01.05.2016 - 14.01
tested files : differenz.c, differenz.h, produkt.c, produkt.h, quotient.c, quotient.h, rechner.c, summe.c, summe.h
../rechner.c:14:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(void)
   ^
../rechner.c: In function 'main':
../rechner.c:18:10: warning: variable 'tmp' set but not used [-Wunused-but-set-variable]
  char c, tmp;
         ^
Creation of testprogram finished

Start Tests
Test: InputTest.testAddition started

Programm finished successfully

Test: InputTest.testAddition finished

Test: InputTest.testSubtraktion started
Valgrind check started

Valgrind check stopped

Programm finished successfully

Test: InputTest.testSubtraktion finished

Test: Rechner.SummeTest started
Test: Rechner.SummeTest finished

Test: Rechner.DifferenzTest started
Test: Rechner.DifferenzTest finished

Test: Rechner.Produkttest started
Test: Rechner.Produkttest finished

Test: Rechner.QuotientTest started
Test: Rechner.QuotientTest finished

Unittests finished
6 tests in: 2 TestCases ran
Tests succeeded: 6
Tests failed: 0
```

*Abbildung 39: Darstellung der Ausgabe von Student A Aufgabe 3*

Die obige Abbildung veranschaulicht die Testausgabe. Es ist zu sehen, dass alle Tests erfolgreich durchgeführt wurden.

### 11.2.3.2 Lösung und Ausgabe für Student B

Die Lösung von Student B ist ähnlich zu der Lösung von Student A. Der einzige Unterschied ist, dass Student B alle Funktionen in einem Header-Source Paar angelegt hat.

```
1 #ifndef FUNKTIONEN_H
2 #define FUNKTIONEN_H
3
4 int summe(int,int);
5 int differenz(int,int);
6 int produkt(int,int);
7 int quotient(int,int);
8
9 #endif
10
11 #include "funktionen.h"
12
13 int summe(int a, int b)
14 {
15     return a+b;
16 }
17
18 int differenz(int a, int b)
19 {
20     return a-b;
21 }
22
23 int produkt(int a, int b)
24 {
25     return a*b;
26 }
27
28 int quotient(int a, int b)
29 {
30     return a/b;
31 }
```

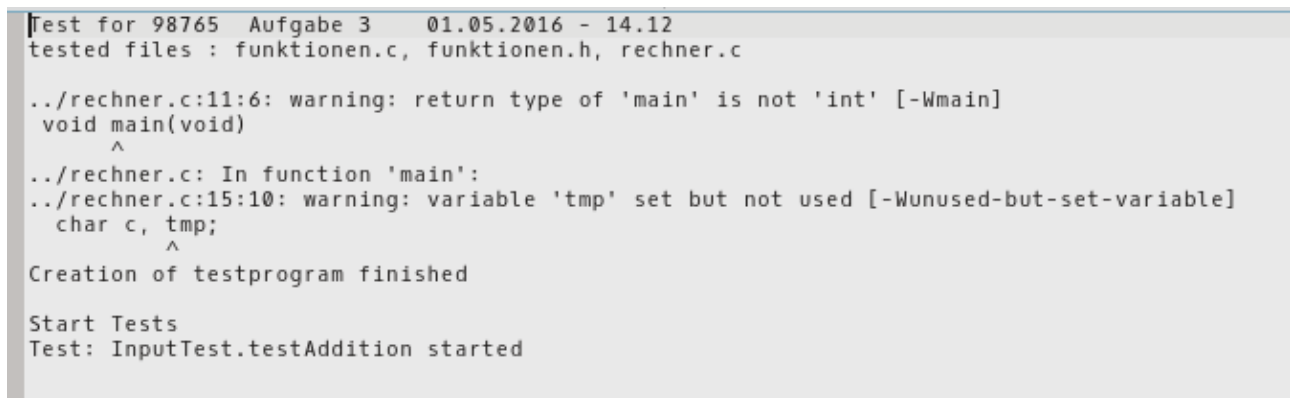
*Listing 21: Header und Source Datei  
mit sämtlichen Funktionen des  
Taschenrechners*

Dieses Listing veranschaulicht den Header `funktionen.h` und die Source Datei `funktionen.c`. Im Gegensatz zu Student A verwendet Student B nur zwei Dateien um alle Funktionen zu definieren bzw. deklarieren.

```
1  
2  
3  
4  
5  
6  
7  
8 #include <stdio.h>  
9 #include "funktionen.h"  
10  
11 void main(void)  
12 {  
13     int a,b;  
14     char c, tmp;  
15  
16     printf("Bitte 2 Zahlen eingeben\n");  
17     if(!scanf("%d %d" &a, &b))
```

Listing 22: Veranschaulichung des Taschenrechners von  
Student B

Das obige Listing veranschaulicht die Main Funktion von Student B. Hier ist zu sehen, dass nur ein Header mit allen Funktionen eingebunden wird. Diese Datei wird wegen der Übersichtlichkeit nicht komplett dargestellt. Der ausgeblendete Teil ist identisch wie bei der Lösung von Student A.



```
|Test for 98765 Aufgabe 3 01.05.2016 - 14.12  
tested files : funktionen.c, funktionen.h, rechner.c  
  
../rechner.c:11:6: warning: return type of 'main' is not 'int' [-Wmain]  
void main(void)  
  ^  
../rechner.c: In function 'main':  
../rechner.c:15:10: warning: variable 'tmp' set but not used [-Wunused-but-set-variable]  
char c, tmp;  
  ^  
  
Creation of testprogram finished  
  
Start Tests  
Test: InputTest.testAddition started
```

Abbildung 40: Darstellung der Ausgabe von Student B Aufgabe 3

Diese Abbildung veranschaulicht die Testausgabe für Student B für Aufgabe 3. Diese wurde wegen der Übersichtlichkeit abgeschnitten, da sie fast identisch zu der Testausgabe von Student A für Aufgabe 3 ist. Der Einzige Unterschied ist oben in der zweiten Zeile zu sehen. Dort sind drei Dateien aufgelistet. Die Lösung von Student A verfügt über neun getestete Dateien.

Dieses Beispiel soll die Vorteile des generated Headers verdeutlichen. Mit der Hilfe des generated Header können Funktionen getestet werden ohne Kenntnis über die verwendeten Dateien zu haben. Lediglich der Funktionsname muss eindeutig definiert sein.

## 11.3 Anmerkungen

Bei der Prüfung auf Strings ist auf die Zeilenenden- und Wagenrücklaufzeichen (`\n` und `\r`) zu achten. Diese sind abhängig von der verwendeten Linux Distribution. (unterschiedliche Kernel Versionen) Diese beinhaltet vermutlich Änderungen an der PTY Schnittstelle (wegen des Bugs) Für das Basisimage des Testcontainers wird das `ubuntu:latest` Image verwendet. Dieses stellt sicher, dass immer die aktuellste Version dieses Ubuntu Images verwendet wird. Kurz vor Ende dieser Arbeit wurde dieses Basisimage von Version 15.10 auf Version 16.04 geändert. Dies hat dazu geführt, dass die `/r` und `/n` Zeichen auch beim Einlesen eines Strings durch die `readFromProgram()` Funktion berücksichtigt werden müssen. Obwohl zu Ende dieser Arbeit die Zeit knapp geworden ist, wurden für diese Beispiele die Ein- und Ausgaben angepasst. Somit wird in nachfolgenden Projekten anhand der Beispiele der Einstieg in die Verwendung der Laufzeitumgebung einfacher. Mit Ubuntu 16.04 steht eine LTS<sup>26</sup> Version bereit, die eine längere Unterstützungsdauer bietet. Bei Verwendung einer anderen Linux Distribution oder anderem Kernel müssten diese Zeichen unter Umständen anders verwendet werden. [Ubuntu 2016]

---

26 LTS – Long Term Support