

4 Unit Testing

4.1 Übersicht

Automatisiertes Unit Testing ist eine Testmethodik, die im Lauf der letzten Jahre eine immer größere Popularität erlangt hat. Gründe hierfür sind unter anderem die zunehmenden Anforderungen an die Zuverlässigkeit von Software, die Wiederverwendbarkeit von Komponenten und die Effizienz von Teamentwicklungen.

Ziel des Unit Testing ist, die funktionale Korrektheit einer Softwareeinheit, eben einer Unit, sicherzustellen. Was ist eine Unit? In C versteht man darunter in der Regel eine einzelne Funktion oder eine Gruppe eng zusammengehöriger Funktionen. Klarer ist die Definition für C++: hier bezeichnet eine Unit eine einzelne Methode oder Klasse.

Unit-Definition

Vermutlich jeder Programmierer hat bereits Unit Tests geschrieben. Beispielsweise wird eine Funktion geschrieben, und man möchte wissen, ob sie funktioniert. Also wird ein kleines Testprogramm erstellt, die Funktion mit verschiedenen Eingangsparametern aufgerufen, die Ergebnisse auf den Bildschirm ausgegeben und geprüft, ob die Ausgabe mit den Erwartungen übereinstimmt. Funktioniert irgendwann einmal alles erwartungsgemäß, dann wandert das »Unit Testprogramm« in den elektronischen Papierkorb und ist verloren.

Und genau an dieser Stelle setzt das methodische Unit Testing an. Unit Tests werden als integraler Bestandteil des Software-Entwicklungsprozesses und nicht als Wegwerfprodukt betrachtet. Die Unit Tests werden gesammelt und während der Entwicklung kontinuierlich durchgeführt.

Besonders weit geht hier der Ansatz der testgetriebenen Entwicklung (Test Driven Development). Dabei entstehen Unit Tests parallel zum eigentlichen Programmcode. Die Vorgehensweise bei dieser Entwicklungsmethodik lässt sich mit drei wiederkehrenden Arbeitsschritten beschreiben:

*Testgetriebene
Entwicklung*

1. Vor der Erstellung eines neuen Stücks Code wird ein Test für diesen neuen Code geschrieben. Dabei ist sicherzustellen, dass der Test fehlschlägt.
2. Dann wird der Code auf möglichst einfache Weise geschrieben.

3. Ist der Test erfolgreich absolviert, dann kann der Code refaktoriert werden.

Dadurch, dass im ersten Schritt bereits ein Test geschrieben wird, besteht der Zwang, sich über die Funktionalität und die Testbarkeit des zu erstellenden Codes Gedanken zu machen. Dieser Schritt hat einen massiven Einfluss auf das Codedesign und führt zu einem sehr gut strukturierten Code. Der Aufwand, der für das Unit Testing getrieben werden muss, ist damit nicht nur ein reiner Testaufwand, sondern kommt auch dem Softwaredesign auf den untersten Ebenen zugute.

Im zweiten Schritt wird der Code erstellt. Diese Lösung muss nur den Test bestehen, sonst nichts. Sie muss nicht schön sein, sie muss nicht optimal sein. Es ist nicht mehr zu tun als unbedingt erforderlich.

Wenn im dritten Arbeitsschritt der Test erfolgreich verläuft, dann ist gewiß, dass der Code und der Test korrekt sind. Nun kann der Code refaktoriert werden. Codeduplikate und alle anderen Unschönheiten werden beseitigt, und alles wird so weit wie erforderlich optimiert. Anhand der Unit Tests ist jederzeit abprüfbar, dass trotz aller Umformungen die korrekte Funktion gegeben ist.

Nach Abschluss des dritten Schrittes kann die Entwicklung weitergehen und das nächste Stück Funktionalität realisiert werden. Und dies beginnt wieder bei Schritt 1 des Zyklus.

Die dreistufige Arbeitsweise empfiehlt sich auch für das Debugging. Tritt ein Fehler auf, der durch die Unit Tests nicht entdeckt wird, dann ist es eine schlechte Idee, sofort mit der Fehlerbeseitigung zu beginnen. Besser wird vielmehr zuerst ein Unit Test erstellt, der genau diesen Fehler aufdeckt. Dann kann der Fehler bereinigt werden. Mit dem Unit Test existiert das Werkzeug, um zu prüfen, dass der Fehler tatsächlich eliminiert ist. Der Fehler kann sich auch nicht nochmals auf irgendeinem anderen Weg einschleichen, da nun ein Unit Test existiert, der dies sofort entdecken würde (Regressionstest).

Die testgetriebene Entwicklung kann auch einen wesentlichen Beitrag für eine reibungslose Teamarbeit leisten. Änderungen eines Teammitgliedes an »seinem« Code, die vielleicht die Funktion des Gesamtwertes an einer ganz anderen Stelle beeinträchtigen, können bei konsequenter Umsetzung des Unit Testing sehr schnell entdeckt werden.

Was Unit Tests nicht leisten

Um keine Missverständnisse aufkommen zu lassen: Unit Tests und die testgetriebene Entwicklung sind kein Allheilmittel. Mit Unit Tests kann sehr fehlerfreier und sicherer Code produziert werden, der dennoch unbrauchbar und viel zu langsam sein kann. Akzeptanz-, Stress-, Performance-, Usabilitytests und wie sie alle heißen sind auch mit der testgetriebenen Entwicklungsmethodik nicht obsolet. Unit Tests sind auch nicht für alle Felder der Softwareentwicklung geeignet. Betriebssystem-

Code, verteilte Softwaresysteme oder grafische Bedienoberflächen sind eher weniger gute Kandidaten für Unit Tests, obwohl zumindest für die grafischen Oberflächen durchaus interessante Ansätze existieren [94].

Die Einführung einer testgetriebenen Entwicklung stößt bei Programmierern und Managern häufig auf eine gewisse Skepsis, denn es ist definitiv so, dass der Zeitaufwand für das Codieren ansteigt. Untersuchungen zeigen jedoch, dass die Produktivität im Sinne von »korrekter Code pro Zeiteinheit« ansteigt, in einigen Fällen sogar dramatisch [95].

Für den Einstieg in das Unit Testing gibt es eine ganze Reihe von Test-Frameworks. Einen Überblick gibt zum Beispiel [97]. Diese bieten Unterstützung bei der Organisation und Ausführung der Unit Tests und bei der Protokollierung der Testläufe und Testergebnisse. Quasi Urvater aller Testumgebungen ist *JUnit* [53, 52], das Framework für die Programmiersprache *Java*. Für die Sprachen *C* und *C++* sind verschiedene Testumgebungen verfügbar, von denen jedoch noch keine so ausgereift ist wie *JUnit*. Vielversprechende Kandidaten sind *CUnit* für die Programmiersprache *C*, die von *JUnit* abgeleitete Umgebung *CppUnit* für *C++* und die *Boost Test Library* ebenfalls für *C++*. Diese Frameworks werden in den folgenden Abschnitten vorgestellt.

Test-Frameworks

4.2 C Unit Testing mit CUnit

CUnit [28] ist ein schlankes Framework für Unit Tests in der Programmiersprache *C*. Der Code des Frameworks enthält keine Besonderheiten und stellt für gängige Compiler kein Problem dar. Die Distribution enthält Build-Dateien für *Jam* (Seite 188) und Projektdateien für Microsoft Visual Studio 2003 und 2005 (Visual C++ 7.0 und 8.0). Damit ist es ein Leichtes, die *CUnit*-Bibliothek zu erstellen und zusammen mit den entsprechenden Header-Dateien in ein geeignetes Verzeichnis zu installieren. Alternativ können auch fertig kompilierte Binaries verwendet werden, die für Windows und Linux (RPM) verfügbar sind. *CUnit* ist Open Source und unter der *GNU Library General Public Licence* (LGPL) verfügbar.

Die grundlegende Struktur des Frameworks ist in Abbildung 4.1 gezeigt. Alle Testfälle (Testcases) sind in Testsuites organisiert. Üblicherweise sind in einer Testsuite die Testfälle für eine bestimmte Funktionalität zusammengefasst. Jede Testsuite kann optional eine Initialisierungsfunktion und eine Aufräumfunktion enthalten. Die Initialisierungsfunktion kann verwendet werden, um vor Ausführung der Testfälle einer Suite eventuell erforderliche Voraussetzungen und Rahmenbedingungen herzustellen. Die Aufräumfunktion, die nach Ausführung der Testfälle aufgerufen wird, dient umgekehrt dazu, Artefakte, die bei der Initialisierung oder Testausführung entstanden sind, zu beseitigen.

Testsuite

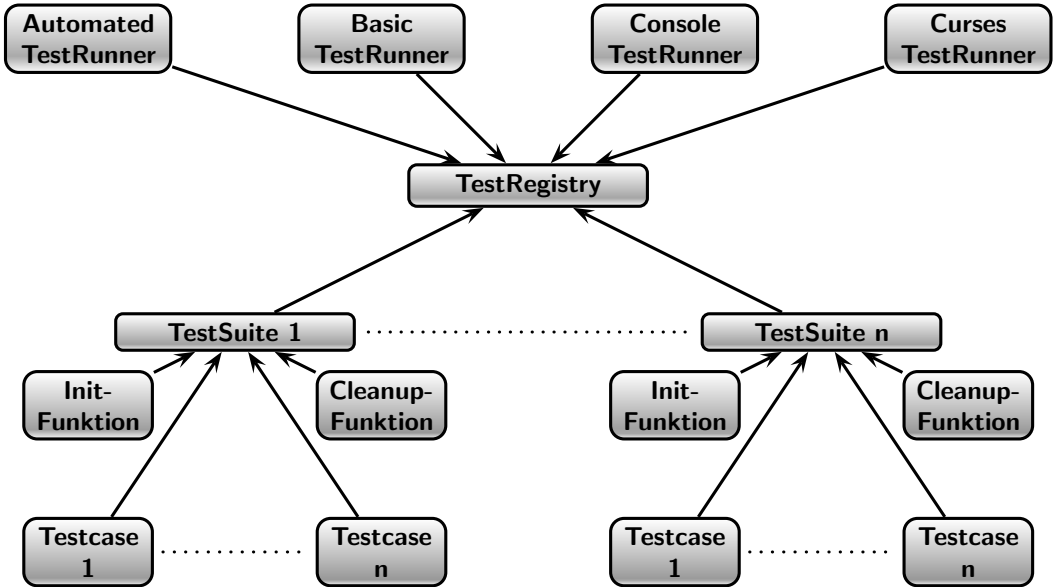


Abbildung 4.1
CUnit-Struktur

Testregistry

Alle Testsuites sind in einer Testregistry zusammengefasst. Auf diese zentrale Registrierung greifen verschiedene Testrunner zu, die die Tests ausführen und die Testergebnisse protokollieren. *CUnit* bietet zwei interaktive Testrunner mit einer Console- oder Curses-Bedienschnittstelle [29] und zwei automatische, nichtinteraktive Testrunner, die die Testergebnisse entweder auf die Console oder in eine XML-Datei schreiben.

Testrunner

*Einführendes
Beispiel*

Die konkrete Vorgehensweise für das Unit Testing mit *CUnit* kann am besten anhand eines Beispiels aufgezeigt werden. Hierzu realisieren wir für eine Bibliothek für das Lesen textbasierter Konfigurationsdateien eine Funktion, mit der die einzelnen Zeilen einer Textdatei gelesen werden können. Die Funktion ist in der Header-Datei `cfgreader.h` im Projekt-Unterverzeichnis `src` deklariert.

```

/* src/cfgreader.h */

#ifndef CFGREADER_H
#define CFGREADER_H
#include <stdio.h>

char *cfgReadline(FILE *fp);

#endif /* CFGREADER_H */

```

Die Funktion erhält als Parameter einen Dateizeiger auf die Konfigurationsdatei. Mit jedem Aufruf soll eine Zeile eingelesen und ein Zeiger auf die gelesene Zeichenkette zurückgeliefert werden.

Folgen wir der Methodik der testgetriebenen Entwicklung, dann muss zunächst ein Test für diese Funktion geschrieben werden. Doch vorher muss das Test-Framework aufgesetzt werden. Hierzu benötigen wir ein Programm, in dem das Framework initialisiert und die Testfunktionen definiert werden. Die Quelldatei `utest_cfgreader.c` für dieses Unit Testprogramm wird im Projekt-Unterverzeichnis `tests` gespeichert.

Erster Schritt

Hauptprogramm

```
1  /* tests/utest_cfgreader.c */
2
3  #include <CUnit/CUnit.h>
4  #include <CUnit/Basic.h>
5  #include <cfgreader.h>
6
7  int init_cfgReadline(void);
8  int cleanup_cfgReadline(void);
9  void test01_cfgReadline(void);
10
11 int main(void)
12 {
13     CU_pSuite suite;
14
15     CU_initialize_registry();
16     suite = CU_add_suite("cfgReadline",
17                         init_cfgReadline, cleanup_cfgReadline);
18     CU_ADD_TEST(suite, test01_cfgReadline);
19     CU_basic_run_tests();
20     CU_cleanup_registry();
21
22     return 0;
23 }
```

In Zeile 3 und 4 werden die *CUnit*-Header-Dateien eingebunden. Die Datei `CUnit/CUnit.h` enthält die grundlegenden Deklarationen, und `CUnit/Basic.h` ist für den im Beispiel verwendeten automatischen, nichtinteraktiven Testrunner erforderlich.

Mit der Include-Anweisung in Zeile 5 wird der Prototyp der zu testenden Funktion eingebunden. Die Initialisierungsfunktion, die Aufräumfunktion und die eigentliche Testfunktion sind in den Zeilen 7 bis 9 deklariert. In der Hauptroutine wird zunächst mit `CU_initialize_registry()` in Zeile 15 die Testregistry initialisiert und dann mit `CU_add_suite()` eine neue Testsuite zur Registry hinzugefügt. Beim Hinzufügen der Suite wer-

den die Initialisierungsfunktion und die Aufräumfunktion sowie der Name `cfgReadline` für die Suite angegeben.

Der Testfunktion wird in Zeile 18 mit Hilfe des Makros `CU_ADD_TEST()` zur Testsuite hinzugefügt. Anschließend wird mit `CU_basic_run_test()` der Testrunner aufgerufen, der alle Tests in allen registrierten Testsuites ausführt und die Testergebnisse auf die Console ausgibt. Am Ende bleibt noch, die Testregistry mit `CU_cleanup_registry()` aufzuräumen und das Testprogramm mit der `return`-Anweisung in Zeile 22 zu beenden.

Init-Funktion Die zu realisierende Funktion soll Zeilen aus einer Textdatei lesen. Für den Test der Funktion benötigen wir also eine Textdatei mit definiertem Inhalt. Es liegt nahe, diese Datei in der Init-Funktion `init_cfgReadline()` zu erstellen und einen Dateizeiger für die zu testende Funktion entsprechend zu initialisieren.

```

25 static FILE *fp;
26 const static char *FILENAME = "test_cfgreader.txt";
27
28 int init_cfgReadline(void)
29 {
30     if (NULL == (fp = fopen(FILENAME, "w"))) return -1;
31     if (EOF == fputs("First line\n", fp)) return -1;
32     if (EOF == fputs("Last line\n", fp)) return -1;
33     if (0 != fclose(fp)) return -1;
34     fp = fopen(FILENAME, "r");
35     if (NULL == fp) return -1;
36     return 0;
37 }
```

Diese Init-Funktion schreibt eine Textdatei mit zwei Zeilen und öffnet anschließend die Datei zum Lesen. Schlägt irgendeine dieser Aktionen fehl, dann wird dies dem Framework über den Rückgabewert `-1` signalisiert. Die Tests werden dann erst gar nicht durchgeführt.

Cleanup-Funktion Nach Ausführung der Tests werden die Textdatei und der Dateizeiger nicht mehr benötigt. In der Aufräumfunktion `cleanup_cfgReadline()` können wir deshalb die Datei schließen und löschen.

```

39 int cleanup_cfgReadline(void)
40 {
41     fclose(fp);
42     remove(FILENAME);
43     return 0;
44 }
```

Testfunktion Nun fehlt noch die eigentliche Testfunktion. Mit der zuvor erzeugten Textdatei erwarten wir beim ersten Aufruf unserer Lesefunktion die Zeichen-

kette »First line« und beim zweiten Aufruf den Wert »Last line«. Wir können unsere Erwartung mit dem Makro `CU_ASSERT_STRING_EQUAL` testen. *CUnit* stellt eine ganze Reihe solcher `CU_ASSERT`-Makros zur Prüfung verschiedener Bedingungen bereit. `CU_ASSERT_STRING_EQUAL` vergleicht zwei Zeichenketten. Bei Gleichheit der Zeichenketten wertet *CUnit* den Test als bestanden, bei Ungleichheit als fehlgeschlagen.

```
46 void test01_cfgReadline(void)
47 {
48     CU_ASSERT_STRING_EQUAL(cfgReadline(fp), "First line");
49     CU_ASSERT_STRING_EQUAL(cfgReadline(fp), "Last line");
50 }
```

Damit das Unit Testprogramm überhaupt kompiliert, benötigen wir eine minimale Implementierung der zu testenden Funktion.

```
/* src/cfgreader.c */

#include <stdio.h>
#include <cfgreader.h>

char *cfgReadline(FILE *fp)
{
    return "Bla";
}
```

Gemäß der Vorgehensweise bei der testgetriebenen Entwicklung muss sichergestellt sein, dass der Unit Test zunächst fehlschlägt. Die angegebene Funktion leistet das.

Das Erstellen des Testprogrammes erfolgt (natürlich) mit *make* und kann mit einer kleinen Make-Datei bewerkstelligt werden.

Make-Datei

```
1 # tests/Makefile
2
3 CFLAGS += -I/home/project/cunit/include -I../src
4 CFLAGS += -Wall
5 LDFLAGS += -L/home/project/cunit/lib
6 LDLIBS = -lcunit
7 VPATH=../src ../bin
8
9 ../bin/utest_cfgreader.exe: utest_cfgreader.o cfgreader.o
10 →$(CC) -o @$^ $(LDFLAGS) $(LOADLIBES) $(LDLIBS)
11
12 ../bin/utest_cfgreader.o: utest_cfgreader.c cfgreader.h
13 ../bin/cfgreader.o: cfgreader.c cfgreader.h
14
15 ../bin/%.o: %.c
16 →$(CC) $(CFLAGS) -c -o @$<
```

Wichtig ist die Angabe der Suchpfade für *CUnit*-Header-Dateien (Zeile 2) und für die *CUnit*-Bibliothek (Zeile 4) sowie das Einbinden der *CUnit*-Bibliothek in Zeile 6.

Nach dem Übersetzen kann das Unit Testprogramm zum ersten Mal ausgeführt werden:

```
$utest_cfgreader
```

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

```
Suite cfgReadline, Test test01_cfgReadline had failures:
```

1. utest_cfgreader.c:47 - →
CU_ASSERT_STRING_EQUAL(cfgReadline(fp),"First line")
2. utest_cfgreader.c:48 - →
CU_ASSERT_STRING_EQUAL(cfgReadline(fp),"Last line")

```
--Run Summary: Type      Total    Ran  Passed  Failed
                 suites      1      1    n/a     0
                 tests      1      1      0     1
                 asserts    2      2      0     2
```

Damit ist das erste Etappenziel der testgetriebenen Entwicklung erreicht. Ein Test ist geschrieben, und der Test schlägt fehl.

Zweiter Schritt

Im nächsten Schritt geht es nun darum, den Code für die Lesefunktion zu schreiben, und zwar (testgetrieben!) auf die einfachstmögliche Weise. Zur Beschleunigung des Codier- und Testzyklus modifizieren wir jedoch zuerst die Make-Datei, so dass nach der Übersetzung des Programmes der Unit Test gleich ausgeführt wird.

```
1 # tests/Makefile
2 CFLAGS += -Ic:/home/project/cunit/include -I../src
3 CFLAGS += -Wall
4 LDFLAGS += -Lc:/home/project/cunit/lib
5 LDLIBS = -lcunit
6 VPATH = ../src ../bin
7
8 all: ../bin/utest_cfgreader.exe utest
9 ../bin/utest_cfgreader.exe: utest_cfgreader.o cfgreader.o
10 →$(CC) -o $@ $^ $(LDFLAGS) $(LOADLIBES) $(LDLIBS)
11
12 ../bin/utest_cfgreader.o: utest_cfgreader.c cfgreader.h
13 ../bin/cfgreader.o: cfgreader.c cfgreader.h
14
```

```

15 utest:
16 → ../bin/utest_cfgreader.exe
17
18 ../bin/%.o: %.c
19 → $(CC) $(CFLAGS) -c -o $@ $<
20
21 .PHONY: all utest

```

Für die Lesefunktion könnte nach etwas Überlegung die folgende Lösung herauskommen.

```

/* src/cfgreader.c */

#include <stdio.h>
#include <string.h>
#include <cfgreader.h>

char *cfgReadline(FILE *fp)
{
    static char buffer[256];
    int length;

    fgets(buffer, 256, fp);
    length = strlen(buffer);
    if ((length > 0) && (buffer[length-1] == '\n'))
    {
        buffer[length-1] = 0;
    }
    return buffer;
}

```

Es ist nicht sicher, dass dies die einfachstmögliche Weise ist, um die Aufgabe zu erfüllen. Auf jeden Fall ist die Funktion recht simpel, und sie besteht den Unit Test:

```

--Run Summary: Type      Total   Ran  Passed  Failed
                suites      1     1     n/a     0
                tests       1     1     1       0
                asserts     2     2     2       0

```

Was nun? Es ist offensichtlich, dass der Code an einigen Stellen verbessert werden muss. Beispielsweise muss die Funktion das Dateiende erkennen, sie soll dann einen Null-Zeiger zurückliefern. Das ist eine neue Funktion, also brauchen wir zunächst einen geeigneten Test. Da die einzelnen Tests voneinander unabhängig sein sollen, kommen wir nicht umhin, die bestehende Testfunktion `test01_cfgReadline` so abzuändern, dass am Ende des

Dritter Schritt

Tests die gleichen Bedingungen wie am Anfang des Tests gelten. In diesem Fall reicht es, den Dateizeiger auf den Dateianfang zurückzusetzen.

```
void test01_cfgReadline(void)
{
    CU_ASSERT_STRING_EQUAL(cfgReadline(fp), "First line");
    CU_ASSERT_STRING_EQUAL(cfgReadline(fp), "Last line");
    fseek(fp, 0L, SEEK_SET);
}
```

Jetzt kann der Test für die neue Funktionalität geschrieben werden. Wir erwarten, dass der dritte Aufruf der Funktion `cfgReadline()` einen Nullzeiger zurückgibt. Dies kann mit dem Makro `CU_ASSERT_PTR_NULL` aus dem *CUnit*-Framework geprüft werden:

```
void test02_cfgReadline(void)
{
    CU_ASSERT_STRING_EQUAL(cfgReadline(fp), "First line");
    CU_ASSERT_STRING_EQUAL(cfgReadline(fp), "Last line");
    CU_ASSERT_PTR_NULL(cfgReadline(fp));
    fseek(fp, 0L, SEEK_SET);
}
```

Diese Testfunktion muss zur Testsuite hinzugefügt werden. Gleichzeitig erweitern wir die `main()`-Funktion des Testprogrammes. Der Rückgabewert soll gleich der Anzahl der fehlgeschlagenen Tests sein. Diesen Wert liefert die Funktion `CU_get_number_of_tests_failed()`:

```
12 int main(void)
13 {
14     CU_pSuite suite;
15     int retval;
16
17     CU_initialize_registry();
18     suite = CU_add_suite("cfgReadline",
19                         init_cfgReadline, cleanup_cfgReadline);
20     CU_ADD_TEST(suite, test01_cfgReadline);
21     CU_ADD_TEST(suite, test02_cfgReadline);
22     CU_basic_run_tests();
23     retval = CU_get_number_of_tests_failed();
24     CU_cleanup_registry();
25
26     return retval;
27 }
```

Da der Exit-Code des Testprogrammes nun gleich der Anzahl der fehlgeschlagenen Tests ist, erhalten wir im Build-Vorgang eine Rückmeldung, wenn die Unit Tests nicht alle bestanden werden.

```
$make
```

```
utest_cfgreader.exe
```

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

```
Suite cfgReadline, Test test02_cfgReadline had failures:
```

1. utest_cfgreader.c:60 - →
CU_ASSERT_PTR_NULL(cfgReadline(fp))

```
--Run Summary: Type      Total   Ran  Passed  Failed
                 suites     1     1    n/a     0
                 tests     2     2     1     1
                 asserts   5     5     4     1
```

```
make: *** [utest] Error 1
```

Jetzt haben wir wieder einen Test, der wie gewünscht fehlschlägt, und wir können nun beginnen, die neue Funktionalität zu implementieren, bis der Test erfolgreich ist.

Das Beispiel bis hier erläuterte sehr detailliert den Zyklus der testgetriebenen Entwicklung und die prinzipielle Vorgehensweise des Unit Testing mit dem *CUnit*-Framework.

In den folgenden Abschnitten werden die einzelnen Funktionen des Frameworks genauer betrachtet.

4.2.1 CU_ASSERT-Makros

Bei der Realisierung einer Softwarefunktion gibt der Programmierer eine Zusicherung (englisch: Assertion), dass die Funktion auf eine spezifizierte Art und Weise arbeitet. In den Unit Tests soll geprüft werden, ob diese Zusicherungen auch tatsächlich eingehalten werden. Konkret wird die korrekte Funktion der »Units« geprüft, indem die Ausgabewerte oder Zustände der zu prüfenden Einheit mit erwarteten Werten oder Zuständen verglichen werden. Das *CUnit*-Framework stellt für die Prüfung ein Bündel von ASSERT-Makros zur Verfügung, um verschiedene Vergleiche zwischen tatsächlichen und erwarteten Werten oder Zuständen durchführen zu können. Das Framework führt genau Buch über die Gesamtzahl der aufgerufenen ASSERT-Makros, die Zahl der erfolgreichen und die Zahl der fehlgeschlagenen Prüfungen. Der Name der Quelldatei und die Zeilen-

nummer des Makroaufrufes können im Fall einer fehlgeschlagenen Prüfung protokolliert werden.

Die 35 verfügbaren Makros werden im Folgenden beschrieben.

`CU_ASSERT(<Ausdruck>)`

`CU_ASSERT_FATAL(<Ausdruck>)`

`CU_TEST(<Ausdruck>)`

`CU_TEST_FATAL(<Ausdruck>)`

Die Prüfung gilt als bestanden, wenn die Auswertung von `<Ausdruck>` den Wert wahr liefert, das heißt ungleich Null ist. Der übergebene Ausdruck muss in einen int-Ausdruck expandieren.

Beispiel:

```
51 void test01(void)
52 {
53     CU_ASSERT(1 + 2 == 3); /* pass */
54     CU_ASSERT(1 + 2 == 4); /* fail */
55     CU_TEST(1 + 2 == 3);   /* pass */
56     CU_TEST(1 + 2 == 4);   /* fail */
57 }
```

Suite asserttest, Test test01 had failures:

1. utest.c:54 - 1 + 2 == 4
2. utest.c:56 - 1 + 2 == 4

Im Gegensatz zu den Makros `CU_ASSERT()` und `CU_TEST()` wird bei den Makros `CU_ASSERT_FATAL()` und `CU_TEST_FATAL()` der aktuelle Testfall abgebrochen, wenn die Prüfung fehlschlägt.

`CU_ASSERT_TRUE(<Ausdruck>)`

`CU_ASSERT_TRUE_FATAL(<Ausdruck>)`

Die Prüfung ist bestanden, wenn der expandierte `<Ausdruck>` wahr, das heißt ungleich Null ist. Ein Fehlschlag bei der Prüfung mit `CU_ASSERT_TRUE_FATAL()` bewirkt einen Abbruch des Testfalles.

`CU_ASSERT_FALSE(<Ausdruck>)`

`CU_ASSERT_FALSE_FATAL(<Ausdruck>)`

Die Prüfung ist bestanden, wenn der expandierte `<Ausdruck>` nicht wahr, das heißt gleich Null ist. `CU_ASSERT_FALSE_FATAL()` bricht bei fehlgeschlagener Prüfung den Testfall ab.

Beispiel:

```
59 void test02(void)
60 {
61     CU_ASSERT_TRUE(2 == 3);
```

```

62     CU_ASSERT_FALSE(2 == 2);
63 }

```

Suite assertttest, Test test02 had failures:

1. utest.c:61 - CU_ASSERT_TRUE(2 == 3)
2. utest.c:62 - CU_ASSERT_FALSE(2 == 2)

CU_PASS(*Meldung*)

Diese Prüfung zählt als immer bestanden. Das Makro wird eigentlich nur benötigt, um die Anzahl der durchgeführten Prüfungen korrekt zu zählen, und ist nur in Zusammenhang mit CU_FAIL() von Interesse. Die *Meldung* wird nie ausgegeben.

CU_FAIL(*Meldung*)

CU_FAIL_FATAL(*Meldung*)

Als Gegenstück zu CU_PASS() zählt diese Prüfung als nie bestanden. Die *Meldung* wird im Fehlerprotokoll ausgegeben. Das Makro ist zusammen mit CU_PASS() geeignet, um aufwändigere benutzerdefinierte Prüfungen zu realisieren.

Hier ein Beispiel (aus [28]):

```

65 void test03(void)
66 {
67     jmp_buf buf;
68     int i;
69
70     i = setjmp(buf);
71     if (i == 0)
72     {
73         run_other_func(&buf);
74         CU_PASS("run_other_func() succeeded.");
75     }
76     else
77     {
78         CU_FAIL("run_other_func() issued longjmp.");
79     }
80 }

```

Suite assertttest, Test test03 had failures:

1. utest.c:78 - CU_FAIL("run_other_func() issued longjmp.")

Beim Aufruf des Makros CU_FAIL_FATAL() wird der Testfall abgebrochen.

CU_ASSERT_EQUAL(*⟨Wert⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_EQUAL_FATAL(*⟨Wert⟩*, *⟨Erwartungswert⟩*)

Die Makros prüfen, ob der *⟨Erwartungswert⟩* und der übergebene *⟨Wert⟩* gleich sind. Für den Vergleich wird der Operator `==` verwendet. Stimmen die Werte nicht überein, dann gilt der Test als nicht bestanden. `CU_ASSERT_EQUAL_FATAL()` bricht den Testfall bei nicht bestandenem Test ab.

CU_ASSERT_NOT_EQUAL(*⟨Wert⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_NOT_EQUAL_FATAL(*⟨Wert⟩*, *⟨Erwartungswert⟩*)

Diese Makros sind quasi das Gegenstück zu `CU_ASSERT_EQUAL()` und `CU_ASSERT_EQUAL_FATAL()`. Der Test ist bestanden, wenn *⟨Wert⟩* und *⟨Erwartungswert⟩* nicht übereinstimmen. Für den Vergleich der Werte wird der Operator `!=` verwendet. `CU_ASSERT_NOT_EQUAL_FATAL()` bricht bei nicht bestandenem Test den Testfall ab.

CU_ASSERT_PTR_EQUAL(*⟨Zeiger⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_PTR_EQUAL_FATAL(*⟨Zeiger⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_PTR_NOT_EQUAL(*⟨Zeiger⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_PTR_NOT_EQUAL_FATAL(*⟨Zeiger⟩*, *⟨Erwartungswert⟩*)

Die Makros prüfen auf Gleichheit bzw. Ungleichheit zweier Zeiger. *⟨Zeiger⟩* und *⟨Erwartungswert⟩* werden in einen `void*` umgewandelt und mit dem Operator `==` beziehungsweise `!=` verglichen. Mit den FATAL-Versionen der Makros wird der Testfall bei nicht bestandener Prüfung abgebrochen.

CU_ASSERT_PTR_NULL(*⟨Zeiger⟩*)

CU_ASSERT_PTR_NULL_FATAL(*⟨Zeiger⟩*)

CU_ASSERT_PTR_NOT_NULL(*⟨Zeiger⟩*)

CU_ASSERT_PTR_NOT_NULL_FATAL(*⟨Zeiger⟩*)

Der *⟨Zeiger⟩* wird nach einer Typumwandlung auf `void*` mit dem Nullzeiger `NULL` verglichen. Bei Gleichheit gilt für `CU_ASSERT_PTR_NULL()` der Test als bestanden, für `CU_ASSERT_PTR_NOT_NULL()` als nicht bestanden. Mit den FATAL-Versionen wird der Testfall bei nicht bestandenem Test abgebrochen.

CU_ASSERT_STRING_EQUAL(*⟨Zeichenkette⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_STRING_EQUAL_FATAL(*⟨Zeichenkette⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_STRING_NOT_EQUAL(*⟨Zeichenkette⟩*, *⟨Erwartungswert⟩*)

CU_ASSERT_STRING_NOT_EQUAL_FATAL(*⟨Zeichenkette⟩*, *⟨Erwartungswert⟩*)

Gleichheit und Ungleichheit von Zeichenketten können mit diesen Makros geprüft werden. Der Vergleich erfolgt mit der Funktion `strcmp()` aus der C-Standardbibliothek. Vor dem Vergleich wird

eine Typumwandlung von $\langle \text{Zeichenkette} \rangle$ und $\langle \text{Erwartungswert} \rangle$ auf `const char*` durchgeführt. Ein Beispiel:

```
char *strtoupper(char *s)
{
    int i;
    for (i = strlen(s); i > 0; i--)
    {
        s[i] = (char)toupper((int)s[i]);
    }
    return s;
}

100 void test07(void)
101 {
102     char s[] = "blub";
103
104     CU_ASSERT_STRING_EQUAL(strtoupper(s), "BLUB");
105     CU_ASSERT_STRING_NOT_EQUAL(s, "BLUB");
106 }
```

Suite assertttest, Test test07 had failures:

1. utest.c:104 - CU_ASSERT_STRING_EQUAL(strtoupper(s),"BLUB")
2. utest.c:105 - CU_ASSERT_STRING_NOT_EQUAL(s,"BLUB")

Bei Verwendung der FATAL-Versionen der Makros wird der Testfall bei nicht bestandener Prüfung abgebrochen.

```
CU_ASSERT_NSTRING_EQUAL( $\langle \text{Zeichenkette} \rangle$ ,  $\langle \text{Erwartungswert} \rangle$ ,  $\langle n \rangle$ )
CU_ASSERT_NSTRING_EQUAL_FATAL( $\langle \text{Zeichenkette} \rangle$ ,  $\langle \text{Erwartungswert} \rangle$ ,  $\langle n \rangle$ )
CU_ASSERT_NSTRING_NOT_EQUAL( $\langle \text{Zeichenkette} \rangle$ ,  $\langle \text{Erwartungswert} \rangle$ ,  $\langle n \rangle$ )
CU_ASSERT_NSTRING_NOT_EQUAL_FATAL( $\langle \text{Zeichenkette} \rangle$ ,  $\langle \text{Erwartungswert} \rangle$ ,  $\langle n \rangle$ )
```

Diese Makros prüfen auf Gleichheit beziehungsweise Ungleichheit der ersten $\langle n \rangle$ Zeichen der $\langle \text{Zeichenkette} \rangle$ und des Erwartungswertes. Der Vergleich erfolgt mit der Funktion `strncmp()` aus der C-Standardbibliothek. $\langle \text{Zeichenkette} \rangle$ und $\langle \text{Erwartungswert} \rangle$ werden vor dem Vergleich in `const char*` umgewandelt, für $\langle n \rangle$ wird eine Typumwandlung in `size_t` durchgeführt.

Beispiel:

```
108 void test08(void)
109 {
110     CU_ASSERT_NSTRING_EQUAL("blub", "blup", 3);
111     CU_ASSERT_NSTRING_NOT_EQUAL("blub", "blup", 3);
112 }
```

Suite asserttest, Test test08 had failures:

1. utest.c:111 - CU_ASSERT_NSTRING_NOT_EQUAL("blub", "blup", 3)

Bei einer nicht bestandenen Prüfung wird der Testfall bei Verwendung der Makros `CU_ASSERT_NSTRING_EQUAL_FATAL()` und `CU_ASSERT_NSTRING_NOT_EQUAL_FATAL()` abgebrochen.

`CU_ASSERT_DOUBLE_EQUAL(<Wert>, <Erwartungswert>, <ε>)`

`CU_ASSERT_DOUBLE_EQUAL_FATAL(<Wert>, <Erwartungswert>, <ε>)`

`CU_ASSERT_DOUBLE_NOT_EQUAL(<Wert>, <Erwartungswert>, <ε>)`

`CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL(<Wert>, <Erwartungswert>, <ε>)`

Die `double`-Zahlen `<Wert>` und `<Erwartungswert>` werden unter Berücksichtigung eines Toleranzwertes `<ε>` miteinander verglichen. Die Prüfung mit `CU_ASSERT_DOUBLE_EQUAL()` gilt als bestanden, wenn die Bedingung

$$|\langle \text{Wert} \rangle - \langle \text{Erwartungswert} \rangle| \leq |\epsilon|$$

erfüllt ist. Analog gilt `CU_ASSERT_DOUBLE_NOT_EQUAL()` für

$$|\langle \text{Wert} \rangle - \langle \text{Erwartungswert} \rangle| > |\epsilon|$$

als bestanden. `<Wert>` und `<ε>` werden für den Vergleich in den Datentyp `double` gewandelt.

4.2.2 Funktionen der Testregistry

Die Testregistry ist, wie in Abbildung 4.1 gezeigt, die zentrale Komponente des *CUnit*-Frameworks. Die wichtigsten Funktionen der Registry sind die Initialisierungsfunktion und die Aufräumfunktion:

```
CU_ErrorCode CU_initialize_registry(void)
```

```
void CU_cleanup_registry(void)
```

Initialisierung

Finalisierung

Die Initialisierungsfunktion reserviert Speicher für die interne Datenstruktur der Testregistry, die Aufräumfunktion gibt diesen Speicher wieder frei. Der Rückgabewert der Initialisierungsfunktion ist entsprechend `CUE_SUCCESS` im Erfolgsfall oder `CUE_NOMEMORY`, wenn die Speicherreservierung fehlgeschlagen ist. Ist die Testregistry beim Aufruf der Initialisierungsfunktion bereits initialisiert, dann wird intern die Aufräumfunktion aufgerufen und die Registry neu initialisiert.

Mit der Funktion

```
CU_BOOL CU_registry_initialized(void)
```

kann geprüft werden, ob die Registry initialisiert ist. Falls ja, so liefert die Funktion den Wert `CU_TRUE` zurück, sonst `CU_FALSE`.

Das Framework stellt weiterhin die grundlegenden Funktionen für die Verwaltung mehrerer Registrys zur Verfügung. Dies kann bei umfangreichen Projekten nützlich sein, beispielsweise wenn eine einzelne Testregistry zu groß wird und dadurch Tests von Teilkomponenten zu langsam ausgeführt werden. Der folgende Code demonstriert die grundsätzliche Vorgehensweise am Beispiel von zwei Registrys.

Mehrere Registrys

```
1  /* utest.c */
2
3  #include <CUnit/CUnit.h>
4  #include <CUnit/Basic.h>
5
6  int main(void)
7  {
8      CU_pTestRegistry test_registry[2];
9
10     test_registry[0] = CU_create_new_registry();
11     test_registry[1] = CU_create_new_registry();
12
13     if ((NULL == test_registry[0]) || (NULL == test_registry[1]))
14     {
15         /* Fehler bei der Initialisierung */
16     }
17
18     CU_set_registry(test_registry[0]);
19     {
20         /* Testsuites zur ersten Registry hinzufügen */
21     }
22
23     CU_set_registry(test_registry[1]);
24     {
25         /* Testsuites zur zweiten Registry hinzufügen */
26     }
27
28     CU_set_registry(test_registry[0]);
29     CU_basic_run_tests();
30
31     CU_set_registry(test_registry[1]);
32     CU_basic_run_tests();
33
34     CU_destroy_existing_registry(&test_registry[0]);
35     CU_destroy_existing_registry(&test_registry[1]);
36
37     return 0;
38 }
```

Das *CUnit*-Framework arbeitet mit einer einzigen aktiven Registry. Viele Funktionen, etwa für das Hinzufügen von Testsuites oder das Ausführen der Tests, beziehen sich auf die aktive Registry. Deshalb wird im Beispiel nach dem Erzeugen der Registry mit `CU_create_new_registry()` in Zeile 10 und 11 und der Erfolgsprüfung in Zeile 13 die aktive Registry mit dem Funktionsaufruf `CU_set_registry()` in Zeile 18 beziehungsweise 23 umgeschaltet. Anschließend können Testsuites zu der jeweils aktiven Registry hinzugefügt werden. Bei Bedarf kann die aktive Registry mit dem Funktionsaufruf `CU_get_registry()` ermittelt werden. Die Funktion liefert einen Zeiger vom Typ `CU_pTestRegistry` auf die aktive Registry zurück. Auch das Ausführen der Tests mit `CU_basic_run_test()` in Zeile 29 und 32 bezieht sich auf die aktive Registry, die in den jeweiligen Programmzeilen unmittelbar vor dem Funktionsaufruf eingestellt wird. Zum Abschluss müssen am Ende des Unit Testprogrammes die erzeugten Registrys mit `CU_destroy_existing_registry()` wieder freigegeben werden.

4.2.3 Rund um Testsuites und Testcases

Im *CUnit*-Framework sind alle Testcases in einer oder mehreren Testsuites organisiert. Diese Testsuites sind Bestandteil der Testregistry. Mit der Funktion

```
CU_pSuite CU_add_suite(const char* strSuiteName,
                      CU_initializeFunc pInit,
                      CU_cleanupFunc pClean);
```

Testsuite erzeugen

wird eine Testsuite erzeugt und zur aktiven Registry hinzugefügt. Die Suite erhält den Namen `strSuiteName`, der eindeutig sein muss. Eine Registry kann nicht zwei Suites mit dem gleichen Namen enthalten.

Initialisierungs- und Aufräumfunktion

Mit `pInit` und `pClean` können eine Initialisierungsfunktion und eine Aufräumfunktion für die Suite spezifiziert werden. Die Initialisierungsfunktion wird vor dem Ausführen der Tests in der Suite aufgerufen, die Aufräumfunktion am Ende der Tests. Die Initialisierungsfunktion ist geeignet, wenn für die Tests der Suite bestimmte Voraussetzungen geschaffen werden müssen. In der Aufräumfunktion können alle Hinterlassenschaften der Tests beseitigt werden. Wird eine der Funktionen nicht benötigt, dann kann anstelle dessen der Wert `NULL` übergeben werden.

Beide Funktionen werden ohne Parameter aufgerufen und müssen einen `int`-Wert zurückgeben. Der Rückgabewert 0 signalisiert dem Framework eine erfolgreiche Initialisierung oder Finalisierung. Jeder andere Rückgabewert wird als Fehler gewertet und protokolliert. Schlägt die Initialisierungsfunktion fehl, dann werden die Tests der Suite nicht ausgeführt.

Die Funktion `CU_add_suite()` liefert im Erfolgsfall einen Zeiger auf die interne Datenstruktur der Testsuite zurück. Dieser Zeiger wird für das

Hinzufügen von Tests zur Testsuite benötigt. Im Fehlerfall ist der Rückgabewert der Funktion NULL. Die Ursache des Fehlers kann wie in Abschnitt 4.2.5 beschrieben über die Funktionen `CU_get_error()` und `CU_get_error_msg()` ermittelt werden.

Für das Hinzufügen einzelner Tests zu einer Suite ist die Funktion

Tests hinzufügen

```
CU_pTest CU_add_test(CU_pSuite pSuite, const char* strTestName,
                    CU_TestFunc pTestFunc)
```

zuständig. `pSuite` ist ein Zeiger auf die interne Datenstruktur der Suite, wie er von `CU_add_suite()` zurückgeliefert wird. Der eindeutige Name des Tests wird über `strTestName` festgelegt. `pTestFunc` ist der Zeiger auf die Testfunktion, die über keine Aufrufparameter verfügt und keinen Wert zurückliefern darf. Die Funktion liefert im Normalfall einen Zeiger auf eine interne Datenstruktur zurück. Tritt beim Hinzufügen des Tests ein Fehler auf, dann ist der Rückgabewert NULL. Genauere Informationen über den Fehler liefern die Funktionen `CU_get_error()` und `CU_get_error_msg()`.

Das Makro

*Tests per Makro
hinzufügen*

```
CU_ADD_TEST(pSuite, pTestFunc)
```

erlaubt eine verkürzte Schreibweise für die Aufnahme einer Testfunktion `pTestFunc` in eine Testsuite `pSuite`. Dabei wird der eindeutige Name des Tests automatisch aus dem Namen der Testfunktion generiert.

Für umfangreiche Testszenarien bietet das *CUnit*-Framework mit den Funktionen

*Testsuites und
Testcases zentral
verwalten*

```
CU_ErrorCode CU_register_suites(CU_SuiteInfo suite_info[])
CU_ErrorCode CU_register_nsuites(int suite_count, ...)
```

eine alternative Möglichkeit, um Testsuites und Testcases zu verwalten. Das einleitende Beispiel von Seite 200 lässt sich mit diesen Funktionen so formulieren, dass alle Testsuites und Tests an einer Stelle erfasst werden:

```
12 int main(void)
13 {
14     int retval;
15     CU_TestInfo test_array[] =
16     {
17         { "test01/cfgReadline", test01_cfgReadline },
18         { "test02/cfgReadline", test02_cfgReadline },
19         CU_TEST_INFO_NULL
20     };
21     CU_SuiteInfo suites[] =
22     {
23         { "cfgReadline", init_cfgReadline,
24           cleanup_cfgReadline, test_array },
25         CU_SUITE_INFO_NULL
26     };
```

```

27     CU_initialize_registry();
28     CU_register_suites(suites);
29     CU_basic_run_tests();
30     retval = CU_get_number_of_tests_failed();
31     CU_cleanup_registry();
32     return retval;
33 }

```

In den Zeilen 15–20 werden die Namen und Funktionszeiger der Testcases für eine einzelne Suite in einem Array vom Typ `CU_TestInfo` gespeichert. Der letzte Wert `CU_TEST_INFO_NULL` kennzeichnet das Ende des Arrays. Analog wird in den Zeilen 21–26 ein Array vom Typ `CU_SuiteInfo` definiert, das alle Testsuites mit Name, Initialisierungsfunktion und Aufräumfunktion sowie den Testcases der Suite enthält. Für die Testcases wird das zuvor definierte `CU_TestInfo`-Array angegeben. Das Ende des `CU_SuiteInfo`-Arrays wird mit dem Wert `CU_SUITE_INFO_NULL` markiert.

Anschließend können nach der Initialisierung der Testregistry alle in diesem Array definierten Testsuites zusammen mit ihren Testcases mit dem Funktionsaufruf `CU_register_suites()` in Zeile 28 in einem Rutsch registriert werden.

Es besteht auch die Möglichkeit, mehrere `CU_SuiteInfo`-Arrays zu definieren und dann einzelne oder auch alle Arrays mit der Funktion `CU_register_nsuites()` zu registrieren. Als erstes Argument ist die Anzahl der Arrays anzugeben, dann folgen in der variablen Argumentliste die einzelnen `CU_SuiteInfo`-Arrays. Im Beispiel wäre demnach in Zeile 28 auch der Aufruf

```
CU_register_nsuites(1, suites);
```

möglich. Der Rückgabewert der Funktionen `CU_register_suites()` und `CU_register_nsuites()` ist `CUE_SUCCESS`, wenn bei der Registrierung der Testsuites und Testcases kein Fehler aufgetreten ist. Sonst wird einer der in Abschnitt 4.2.5 aufgeführten Fehlercodes zurückgegeben.

4.2.4 Testrunner

Für das Ausführen und Protokollieren der Unit Tests stellt das *CUnit*-Framework verschiedene Testrunner zur Verfügung.

Basic-Testrunner

Der *Basic*-Testrunner, der in den bisherigen Beispielen stets verwendet wurde, führt die Tests automatisch ohne Interaktion mit dem Benutzer aus. Für diesen Testrunner muss die Include-Datei `CUnit/Basic.h` eingebunden werden. Die Testergebnisse werden auf die Standardausgabe geschrieben. Der Umfang der Ausgabe kann mit der Funktion

```
void CU_basic_set_mode(CU_BasicRunMode mode)
```

eingestellt werden. Für den Parameter `mode` sind folgende Werte möglich:

CU_BRM_SILENT

Weder Testergebnisse noch eine Zusammenfassung werden angezeigt. Nur Fehlermeldungen werden ausgegeben.

CU_BRM_NORMAL

Ausgegeben werden Informationen über fehlgeschlagene Tests und eine kurze Zusammenfassung am Ende. Dies ist der voreingestellte Ausgabeumfang, der auch in den bisherigen Beispielen eingestellt war.

CU_BRM_VERBOSE

Alle Informationen über bestandene und fehlgeschlagene Tests und eine Zusammenfassung am Ende werden angezeigt.

Die Funktion `CU_basic_get_mode()` liefert den aktuell eingestellten Ausgabemodus zurück.

Für den Aufruf des *Basic*-Testrunners gibt es drei Funktionen. Die Funktion

Basic-Testrunner aufrufen

```
CU_ErrorCode CU_basic_run_tests(void)
```

startet den Testrunner und führt alle Tests in allen Testsuites der aktiven Testregistry aus. Die Testergebnisse werden entsprechend dem aktuellen Ausgabemodus angezeigt.

Sollen nur die Tests einer bestimmten Testsuite ausgeführt werden, dann ist die Funktion

```
CU_ErrorCode CU_basic_run_suite(CU_pSuite pSuite)
```

zu verwenden. Und wenn nur ein einzelner Test in einer bestimmten Suite ausgeführt werden soll, dann kann dies mit der Funktion

```
CU_ErrorCode CU_basic_run_test(CU_pSuite pSuite, CU_pTest pTest)
```

bewerkstelligt werden. Auch in diesem Fall wird vor Ausführung des Tests die Initialisierungsfunktion und nach dem Test die Aufräumfunktion der spezifizierten Suite aufgerufen.

Der zweite nicht interaktive Testrunner ist der *Automated*-Testrunner. Dieser Testrunner schreibt die Testergebnisse nicht auf die Standardausgabe, sondern erzeugt eine Datei im XML-Format. Die Funktionen dieses Testrunners sind in der Include-Datei `CUnit/Automated.h` deklariert.

Automated-Testrunner

Die Ausführung aller Tests in allen Testsuites der aktiven Registry wird mit der Funktion

```
void CU_automated_run_tests(void)
```

gestartet. Funktionen wie beim *Basic*-Testrunner, die nur Tests einer einzelnen Suite oder nur einzelne Tests ausführen, stehen nicht zur Verfügung. Eine XML-Ergebnisdatei, die der Ausgabe des *Basic*-Testrunner von Seite 199 entspricht, ist in Abbildung 4.2 gezeigt.

```
1 <?xml version="1.0" ?>
2 <?xml-stylesheet type="text/xsl" href="CUnit-Run.xsl" ?>
3 <!DOCTYPE CUNIT_TEST_RUN_REPORT SYSTEM "CUnit-Run.dtd">
4 <CUNIT_TEST_RUN_REPORT>
5   <CUNIT_HEADER/>
6   <CUNIT_RESULT_LISTING>
7     <CUNIT_RUN_SUITE>
8       <CUNIT_RUN_SUITE_SUCCESS>
9         <SUITE_NAME> cfgReadline </SUITE_NAME>
10        <CUNIT_RUN_TEST_RECORD>
11          <CUNIT_RUN_TEST_SUCCESS>
12            <TEST_NAME> test01_cfgReadline </TEST_NAME>
13          </CUNIT_RUN_TEST_SUCCESS>
14        </CUNIT_RUN_TEST_RECORD>
15      </CUNIT_RUN_SUITE_SUCCESS>
16    </CUNIT_RUN_SUITE>
17  </CUNIT_RESULT_LISTING>
18  <CUNIT_RUN_SUMMARY>
19    :
20    :
26    <CUNIT_RUN_SUMMARY_RECORD>
27      <TYPE> Test Cases </TYPE>
28      <TOTAL> 1 </TOTAL>
29      <RUN> 1 </RUN>
30      <SUCCEEDED> 1 </SUCCEEDED>
31      <FAILED> 0 </FAILED>
32    </CUNIT_RUN_SUMMARY_RECORD>
33    <CUNIT_RUN_SUMMARY_RECORD>
34      <TYPE> Assertions </TYPE>
35      <TOTAL> 2 </TOTAL>
36      <RUN> 2 </RUN>
37      <SUCCEEDED> 2 </SUCCEEDED>
38      <FAILED> 0 </FAILED>
39    </CUNIT_RUN_SUMMARY_RECORD>
40  </CUNIT_RUN_SUMMARY>
41  <CUNIT_FOOTER> File Generated By CUnit v2.1-0 at Sat Nov 25 20:12:14 2006
42 </CUNIT_FOOTER>
43 </CUNIT_TEST_RUN_REPORT>
```

Abbildung 4.2
XML-Ausgabe des
Automated-
Testrunner

Die in der XML-Datei referenzierte DTD-Datei `CUnit-Run.dtd` ist Bestandteil der *CUnit*-Distribution. Dort findet sich auch eine Vorlage für die XSL-Stildatei `CUnit-Run.xsl`, auf die in Zeile 2 der XML-Ausgabe verwiesen wird. Diese Stildatei transformiert die XML-Ausgabe in HTML. Der XML-Code aus Abbildung 4.2 wird damit in einem Internet-Browser, ähnlich wie in Abbildung 4.3 gezeigt, dargestellt.

XML-Ausgabe

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

Running Suite `cfgReadline`

Running test `test01_cfgReadline ...` Passed

Cumulative Summary for Run				
Type	Total	Run	Succeeded	Failed
Suites	1	1	- NA -	0
Test Cases	1	1	1	0
Assertions	2	2	2	0

File Generated By CUnit v2.1.0 at Sat Nov 25 18:41:01 2006

Abbildung 4.3
Internet-Browser-Darstellung eines XML-Testreports

Neben der Ausgabe der Testergebnisse in eine XML-Datei bietet der Automated-Testrunner auch die Möglichkeit, eine Übersicht aller Testsuites und Testcases der aktiven Registry in eine XML-Datei zu schreiben. Für das Erstellen der Übersicht ist die Funktion

```
CU_ErrorCode CU_list_tests_to_file(void)
```

zuständig. DTD-Datei und eine XSL-Stildatei zur übersichtlichen Darstellung der XML-Ausgabe in einem Internet-Browser sind ebenfalls Bestandteil der Distribution.

Die voreingestellten Namen der Ausgabedateien für das Testprotokoll und für die Testübersicht lauten `CUnitAutomated-Results.xml` beziehungsweise `CUnitAutomated-Listing.xml`. Ist ein anderes Namenspräfix als `CUnitAutomated` für die Ausgabedateien gewünscht, dann kann dieses mit der Funktion

```
void CU_set_output_filename(const char* szFilenamePrefix)
```

eingestellt werden.

Als interaktive Testrunner bietet das *CUnit*-Framework den *Console*-Testrunner und den *Curses*-Testrunner an.

Die Funktionen des Console-Testrunners stehen nach Einbindung der Header-Datei `CUnit/Console.h` zur Verfügung. Nun, genaugenommen handelt es sich nur um die eine Funktion

```
void CU_console_run_tests(void)
```

Console-Testrunner

Der Funktionsaufruf erlaubt mit dem Charme typischer Console-Dialoge die Anzeige und Auswahl von Testsuites und das Durchführen der Tests. Ein Beispiel ist in Abbildung 4.4 gezeigt.

Abbildung 4.4
Dialog des
Console-Testrunners

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

```
***** CUNIT CONSOLE - MAIN MENU *****
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
Enter Command : r
```

```
Running Suite : asserttest
  Running test : test01
  Running test : test02
  Running test : test03
  Running test : test04
  Running test : test05
  Running test : test06
  Running test : test07
  Running test : test08
  Running test : test09
  Running test : test10
```

```
--Run Summary: Type      Total   Ran  Passed  Failed
                suites      1      1    n/a     0
                tests      10     10     1     9
                asserts    18     18     3    15
```

```
***** CUNIT CONSOLE - MAIN MENU *****
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
```

```
Enter Command : _
```

Curses-Testrunner

Der Curses-Testrunner verwendet die Curses-Bibliothek [29] für die Realisierung einer interaktiven Benutzerschnittstelle. Der Funktionsumfang entspricht dem des Console-Testrunners. Aufgerufen wird der Testrunner mit der Funktion

```
void CU_curses_run_tests(void) ,
```

die in der Include-Datei CUnit/CUCurses.h deklariert ist.

4.2.5 Fehlerbehandlung

Im Fehlerfall liefern verschiedene Funktionen des *CUnit*-Frameworks einen Fehlercode vom Typ `CU_ErrorCode` zurück. Dabei handelt es sich um einen Aufzählungstyp (enum), der die Fehlerursache beschreibt. Die verschiedenen Fehlercodes sind in Tabelle 4.1 beschrieben. Grundsätzlich kann der Fehlerstatus mit der Funktion

```
CU_ErrorCode CU_get_error(void)
```

jederzeit ermittelt werden. Die Funktion

```
const char* CU_get_error_msg(void)
```

liefert zu jedem Fehlerstatus die in Tabelle 4.1 aufgeführte kurze Statusmeldung.

Fehlercode	Statusmeldung / Erläuterung
CUE_SUCCESS	No Error Es ist kein Fehler aufgetreten.
CUE_NOMEMORY	Memory allocation failed. Die Anforderung von Arbeitsspeicher ist fehlgeschlagen.
CUE_NOREGISTRY	Test registry does not exist. Die Testregistry ist nicht initialisiert.
CUE_REGISTRY_EXISTS	Registry already exists. Eine existierende Registry darf nicht überschrieben werden, etwa durch einen Aufruf von <code>CU_set_registry()</code> , ohne zuvor <code>CU_cleanup_registry()</code> aufzurufen.
CUE_NOSUITE	NULL suite not allowed. Ein Zeiger auf eine Testsuite-Struktur ist NULL.
CUE_NO_SUITENAME	Suite name cannot be NULL. NULL-Zeiger für Suite-Name ist nicht erlaubt.
CUE_SINIT_FAILED	Suite initialization function failed. Die Initialisierungsfunktion einer Testsuite liefert einen Fehlercode zurück.
CUE_SCLEAN_FAILED	Suite cleanup function failed. Die Aufräumfunktion einer Testsuite liefert einen Fehlercode zurück.
CUE_DUP_SUITE	Suite having name already registered. Eine Testsuite mit dem spezifizierten Namen wurde bereits registriert.
CUE_NOTEST	NULL test not allowed. Ein Zeiger auf eine Testcase-Struktur ist NULL.

Tabelle 4.1
*Fehlercodes und
Meldungen des
CUnit-Frameworks*

CUE_NO_TESTNAME	Test name cannot be NULL. NULL-Zeiger für Testcase-Name ist nicht erlaubt.
CUE_DUP_TEST	Test having this name already in suite. Ein Testcase mit dem spezifizierten Namen wurde bereits registriert.
CUE_TEST_NOT_IN_SUITE	Test not registered in specified suite. Ein angegebener Testfall befindet sich nicht in der spezifizierten Suite.
CUE_FOPEN_FAILED	Error opening file. Fehler beim Öffnen einer Datei.
CUE_FCLOSE_FAILED	Error closing file. Fehler beim Schließen einer Datei.
CUE_BAD_FILENAME	Bad file name. Ungültiger Dateiname.
CUE_WRITE_ERROR	Error during write to file. Fehler beim Schreiben in eine Datei.

Das Verhalten des *CUnit*-Frameworks für den Fehlerfall, das heißt, wenn `CU_get_error()` einen anderen Wert als `CUE_SUCCESS` zurückliefert, kann durch Aufruf von

```
void CU_set_error_action(CU_ErrorAction action)
```

gesteuert werden. Für den Parameter `action` sind die Werte `CUEA_IGNORE`, `CUEA_FAIL` und `CUEA_ABORT` erlaubt. Diese Werte besitzen folgende Bedeutung:

`CUEA_IGNORE`

Fehler werden ignoriert, der Testlauf wird fortgesetzt.

`CUEA_FAIL`

Der Testlauf wird im Fehlerfall beendet.

`CUEA_ABORT`

Die gesamte Testapplikation wird bei Auftreten eines Fehlers beendet.

Voreingestellt ist der Wert `CUEA_IGNORE`. Die aktuelle Einstellung kann mit der Funktion

```
CU_ErrorAction CU_get_error_action(void)
```

ermittelt werden.

4.2.6 Eigene Testrunner schreiben

Wenn die Testrunner des *CUnit*-Frameworks nicht genügen, beispielsweise weil die Ausgabe nicht gefällt oder weil ein spezielles Ausgabeformat benötigt wird, dann ist es nicht allzu schwierig, einen angepassten Testrunner zu

erstellen. Betrachtet wird die prinzipielle Vorgehensweise am Beispiel eines nicht interaktiven Testrunners, der die Meldungen für fehlschlagene Tests in gleicher Weise formatiert wie Fehlermeldungen des GNU-C-Compilers. Mit diesem Format kann ein Editor wie *Emacs* oder die Entwicklungsumgebung *Eclipse CDT* [14] besser umgehen als mit dem Ausgabeformat des Basic-Testrunners. Auf die für praktische Anwendungen sicherlich erforderlichen Fehlerprüfungen wird zugunsten der Übersichtlichkeit verzichtet.

Das Interface des Testrunners umfasst eine einzige Funktion für das Ausführen der Tests. Diese Funktion ist in der Header-Datei `custom.h` deklariert.

Interface

```

1  /* custom.h */
2  #ifndef CUSTOM_H_
3  #define CUSTOM_H_
4  #include <CUnit/CUError.h>
5
6  CU_ErrorCode CCU_custom_run_tests(void);
7
8  #endif /*CUSTOM_H_*/

```

Die Funktion `CCU_custom_run_tests()` liefert einen Fehlercode vom Typ `CU_ErrorCode` zurück, der in der Datei `CUnit/CUError.h` definiert ist.

Für die Implementierung des Testrunners in der Quelldatei `custom.c` muss die Standard-Header-Datei `stdio.h` und die Include-Datei `TestRun.h` des *CUnit*-Frameworks eingebunden werden.

```

1  /* custom.c */
2  #include <stdio.h>
3  #include <CUnit/TestRun.h>
4  #include <custom/custom.h>

```

Nun werden die wichtigen Funktionen deklariert. Dabei handelt es sich um fünf Callback-Funktionen, die dem Framework übergeben und im Verlauf der Testausführung aufgerufen werden.

Callback-Funktionen

```

6  static void custom_test_start_message_handler(
7      const CU_pTest pTest, const CU_pSuite pSuite);
8  static void custom_test_complete_message_handler(
9      const CU_pTest pTest, const CU_pSuite pSuite,
10     const CU_pFailureRecord pFailure);
11 static void custom_all_tests_complete_message_handler(
12     const CU_pFailureRecord pFailure);
13 static void custom_suite_init_failure_message_handler(
14     const CU_pSuite pSuite);
15 static void custom_suite_cleanup_failure_message_handler(
16     const CU_pSuite pSuite);

```

Die Namen der Callback-Funktionen sind selbstredend:

`custom_test_start_message_handler()` wird vom Framework unmittelbar vor dem Aufruf jeder Testfunktion aufgerufen. Die Argumente `pTest` und `pSuite` sind Zeiger auf Datenstrukturen des Testfalles und der Testsuite, über die beispielsweise der Name des Tests und der Suite ermittelt werden kann.

`custom_test_complete_message_handler()` wird im Anschluss an jeden Test aufgerufen. Für den Fall eines fehlgeschlagenen Tests verweist der Zeiger `pFailure` auf eine verkettete Liste mit Informationen über den Fehlschlag. Dies sind unter anderem Zeilennummer, Dateiname und Beschreibung der fehlgeschlagenen `CU_ASSERT`-Makros.

`custom_all_tests_complete_message_handler()` wird nach dem Abschluss aller Tests aufgerufen. `pFailure` zeigt auf eine verkettete Liste mit Informationen über die fehlgeschlagenen Tests.

`custom_suite_init_failure_message_handler()` ist eine Funktion, die aufgerufen wird, wenn die Initialisierungsfunktion einer Testsuite einen Fehlercode zurückliefert.

`custom_suite_init_failure_message_handler()` ist für die Behandlung eines Fehlers in der Aufräumfunktion einer Testsuite zuständig.

Die Callback-Funktionen werden in der Funktion `CCU_custom_run_tests()`, mit der der Testrunner gestartet wird, registriert. Dann wird mit `CU_run_all_tests()` die Ausführung aller Tests in allen Testsuites der aktiven Registry angestoßen.

```
18 CU_ErrorCode CCU_custom_run_tests(void)
19 {
20     CU_set_test_start_handler(
21         custom_test_start_message_handler);
22     CU_set_test_complete_handler(
23         custom_test_complete_message_handler);
24     CU_set_all_test_complete_handler(
25         custom_all_tests_complete_message_handler);
26     CU_set_suite_init_failure_handler(
27         custom_suite_init_failure_message_handler);
28     CU_set_suite_cleanup_failure_handler(
29         custom_suite_cleanup_failure_message_handler);
30
31     return CU_run_all_tests();
32 }
```

Die Funktion `custom_test_start_message_handler()` wird wie beschrieben unmittelbar vor dem Aufruf eines Tests ausgeführt. Der auszuführende Test und die zugehörige Suite werden über Zeiger an die Funktion übergeben. Zunächst wird geprüft, ob der Test zu einer neuen Testsuite gehört.

Falls ja, wird der Name der Suite ausgegeben. Anschließend wird der Name des Tests ausgegeben.

```
34 static void custom_test_start_message_handler(  
35     const CU_pTest pTest, const CU_pSuite pSuite)  
36 {  
37     static char *suiteName = NULL;  
38  
39     if (pSuite->pName != suiteName)  
40     {  
41         suiteName = pSuite->pName;  
42         printf("Suite %s\n", suiteName);  
43     }  
44     printf("Test %s\n", pTest->pName);  
45 }
```

Nach Abschluss des Tests wird `custom_test_complete_message_handler()` aufgerufen. Zusätzlich zu den Zeigern auf die Testsuite und den Testfall wird auch ein Zeiger `pFailure` auf eine verkettete Liste übergeben. Ist der Test nicht fehlgeschlagen, dann hat `pFailure` den Wert `NULL`. Sonst enthält die Liste Informationen über alle fehlgeschlagenen `CU_ASSERT`-Makros. Dateiname, Zeilennummer und eine Beschreibung sind verfügbar. Die Funktion durchläuft diese Liste und gibt die Informationen im gewünschten Format aus.

```
47 static void custom_test_complete_message_handler(  
48     const CU_pTest pTest, const CU_pSuite pSuite,  
49     const CU_pFailureRecord pFailure)  
50 {  
51     CU_pFailureRecord p;  
52  
53     if (pFailure == NULL)  
54     {  
55         printf("passed\n");  
56     }  
57     else  
58     {  
59         p = pFailure;  
60         while (p != NULL)  
61         {  
62             printf("%s: %d: failure: %s\n",  
63                 p->strFileName,  
64                 p->uiLineNumber,  
65                 p->strCondition);  
66             p = p->pNext;  
67         }  
68     }  
69 }
```

Liefert die Initialisierungsfunktion oder die Aufräumfunktion einer Suite einen Fehler zurück, dann wird die Callback-Funktion `custom_suite_init_failure_message_handler()` beziehungsweise `custom_suite_cleanup_failure_message_handler()` aufgerufen. Diese Funktionen tun nichts weiter, als eine entsprechende Meldung auszugeben.

```

71 static void custom_suite_init_failure_message_handler(
72     const CU_pSuite pSuite)
73 {
74     printf("Suite %s: Initialization failed\n", pSuite->pName);
75 }

77 static void custom_suite_cleanup_failure_message_handler(
78     const CU_pSuite pSuite)
79 {
80     printf("Suite %s: Denitailization failed\n", pSuite->pName);
81 }

```

Interessanter wird es wieder beim Abschluss aller Tests, wenn nämlich die Funktion `custom_all_tests_complete_message_handler()` aufgerufen wird.

```

83 static void custom_all_tests_complete_message_handler(
84     const CU_pFailureRecord pFailure)
85 {
86     CU_pTestRegistry r = CU_get_registry();
87
88     printf("\n==RUN SUMMARY==\nType\tTotal\tRan\tFailed\n");
89     printf("Suites\t%d\t%d\t%d\n",
90         r->uiNumberOfSuites,
91         CU_get_number_of_suites_run(),
92         CU_get_number_of_suites_failed());
93     printf("Tests\t%d\t%d\t%d\n",
94         r->uiNumberOfTests,
95         CU_get_number_of_tests_run(),
96         CU_get_number_of_tests_failed());
97     printf("Asserts\t%d\t%d\t%d\n",
98         CU_get_number_of_asserts(),
99         CU_get_number_of_asserts(),
100        CU_get_number_of_failures());
101 }

```

Als Parameter erhält die Funktion mit `pFailure` einen Zeiger auf die verkettete Liste mit den Informationen über alle fehlgeschlagenen Prüfungen. Für die Zusammenfassung der Testergebnisse wird diese Liste jedoch nicht benötigt.

Zuerst wird die Gesamtzahl der Testsuites, die Anzahl ausgeführter und die Anzahl fehlgeschlagener Suites ausgegeben. Die Gesamtzahl der Suites ist nur über das Feld `uiNumberOfSuites` der internen Datenstruktur der aktiven Registry zugänglich, die in Zeile 86 mit `CU_get_registry()` gelesen wird. Lesefunktionen zur Kapselung der Registry-Daten sind hier leider nicht vorhanden. Die Anzahl der ausgeführten und fehlgeschlagenen Testsuites kann hingegen bequem mit den Funktionen `CU_get_number_of_suites_run()` und `CU_get_number_of_suites_failed()` ermittelt werden.

Die Ausgabe der Gesamtzahl der Testfälle und der Zahl ausgeführter und fehlgeschlagener Testfälle erfolgt analog. Das Feld `uiNumberOfTest` der Registry-Datenstruktur liefert die Gesamtzahl, die Funktionsaufrufe `CU_get_number_of_tests_run()` und `CU_get_number_of_tests_failed()` liefern die anderen zwei Werte.

Und fast genauso können die Zahlen für alle `CU_ASSERT`-Makros bestimmt werden: `CU_get_number_of_asserts()` gibt die Gesamtzahl von `CU_ASSERT`-Makros zurück und `CU_get_number_of_failures()` ergibt die Anzahl fehlgeschlagener Makros. Die Anzahl nicht fehlgeschlagener Makros, die im Beispiel nicht benötigt wird, kann mit `CU_get_number_of_successes()` bestimmt werden.

Das Framework bietet noch eine ganze Reihe weiterer Funktionen, um auf die verschiedensten Daten der Testregistry und der Testergebnisse zuzugreifen. Damit dürften sich die meisten Wünsche für benutzerdefinierte Testrunner erfüllen lassen. Einen guten Überblick über diese Funktionen verschafft die Dokumentation des Frameworks, die mit dem Programm *doxygen* (!) erzeugt werden kann.

*Zusammenfassung
der Testergebnisse*