18 Unit-Test

Dieses Kapitel behandelt die folgenden Themen:

- Unit-Test
- Werkzeuge
- Test Driven Development
- Boost Unit Test Framework

Das Testen von Software ist aufwendig: Es nimmt typischerweise etwa ein Drittel des gesamten Software-Entwicklungsaufwandes ein. Der Anteil ist bei sicherheitskritischen Systemen noch erheblich höher.Es gibt verschiedene Arten von Tests: Modul(Unit)-Test, Integrationstest, Systemtest, Abnahmetest. So dient der Abnahmetest dazu, die Funktionalität des Systems bzw. der Software dem Auftraggeber gegenüber nachzuweisen. Ein Bestehen dieses Tests ist in der Regel vertragliche Voraussetzung für die Bezahlung des Werks. Testen hat zwei Aufgaben:

- **Fehler finden:** Je früher ein Fehler erkannt und beseitigt wird, desto besser. Wenn Sie testen, legen Sie sich eine sehr kritische Haltung gegenüber dem zu testenden Prüfling zu! Seien Sie geradezu »sadistisch« und versuchen Sie, das Programm mit Ihren Testfällen möglichst zum Absturz und zu falschen Ergebnissen zu bringen. Unbewusst oder bewusst Schwächen eines Programms auszublenden, spart zwar kurzfristig Zeit, gibt aber langfristig Ärger.

- **Qualität nachweisen:** Qualität ist der Grad, in dem ein System bzw. eine Software Anforderungen erfüllt. Bei dem oben genannten Abnahmetest liegen die Anforderungen in schriftlich festgelegter Form vor (Spezifikation/Pflichtenheft). Es geht dabei nicht nur um funktionale Anforderungen (etwa korrekte Berechnung), sondern auch um nicht-funktionale, zum Beispiel Berechnung innerhalb einer maximalen Dauer, Robustheit gegen fehlerhafte Daten usw.

Eine Übersicht über Softwaretests geben [SL]. Dem Softwareentwickler am nächsten ist der *Unit-Test*. Dabei testet jeder Entwickler selbst die von ihm programmierte Einheit (englisch *unit*), bevor er sie weitergibt. Der Vorteil: Ein Entwickler kennt seine Software genau – und kann deshalb leichter als andere Fehler finden. Der Nachteil: Ein Entwickler kennt seine Software genau – und ist deswegen »betriebsblind«, das heißt, er sieht sein eigenes Produkt zu unkritisch. Es mag unbewusste emotionale Widerstände geben, weswegen Tests auf höherer Ebene nicht von denselben Personen geplant und ausgeführt werden sollen, die die Software entwickelt haben. Erst nach bestandenem Unit-Test ist der Programmteil bereit zu Integration mit anderen Komponenten. Der Integrationstest dient zur Prüfung, ob alle integrierten Komponenten wie beabsichtigt zusammenspielen.

18.1 Werkzeuge

Ein Programm mal eben auszuprobieren, ist kein Testen. Testfälle müssen systematisch mithilfe von Testverfahren spezifiziert werden, zum Beispiel Äquivalenzklassenbildung. Darunter versteht man die Aufteilung von Tests in Klassen, die äquivalent sind, also Gemeinsamkeiten haben. Ein Beispiel für den Test eines Sortierprogramms: Die Testfolgen »1, 3, 7, 4, 5, 8« und »9, 2, 22, 6, 1« gehören zur Äquivalenzklasse »alle Zahlen sind unterschiedlich«, im Gegensatz zur Folge »100, 100, 4, 40, 9, 7«, die mindestens zwei gleiche Zahlen enthält. Zum Finden des Fehlers im Sortierprogramm der Aufgabe von Seite 144 ist gerade dieser Unterschied wichtig. Äquivalenzklassen reduzieren die Anzahl von Tests in den Fällen, in denen die Zahl *aller möglichen* Testfälle einfach zu groß ist, um noch vernünftig handhabbar zu sein. Schon zwanzig voneinander unabhängige `if`-Anweisungen können sich zu über einer Million (genau: 2^{20}) verschiedener Möglichkeiten der Ausführung eines Programms addieren. Oder, um das Beispiel des Sortierprogramms wieder aufzugreifen: Wenn *alle* möglichen Kombinationen einer Folge von N Werten getestet werden sollen, ist der Test bereits bei einem kleinen N (zum Beispiel 20) nicht mehr durchführbar.

Die notwendige Anzahl von Testfällen kann also recht groß werden, schon bei Software mittlerer Größe. Weil nach einer Änderung der Software möglicherweise auch entfernte Softwareteile betroffen sind, müssen alle Tests wiederholt werden (Regressionstest). Das lässt sich nur dann ökonomisch bewerkstelligen, wenn der Testprozess automatisiert abläuft.

Der Test selbst ist ebenfalls Software, die programmiert werden muss. Da wesentliche Elemente des Testens wiederkehren, ist es sehr empfehlenswert, Frameworks für die Programmierung der Tests einzusetzen und auf eine rein individuelle Lösung zu verzichten.

Für diese Frameworks hat sich der Name XUnit eingebürgert, in Anlehnung an JUnit, ein Unit-Test-Framework für die Programmiersprache Java. Dabei steht X für den Kontext, zum Beispiel DB für ein Framework zum Testen von Datenbankanwendungen. Es gibt mehrere Frameworks für Unit-Tests in C++. Ich stelle Ihnen in Auszügen das Boost Unit Test Framework vor, nicht nur, weil die Boost-Libraries den C++-Standard beeinflusst haben, sondern auch, weil sie portabel und für ihre Qualität bekannt sind. Außerdem verwende ich in diesem Buch an mehreren Stellen die Boost Libraries (Threads, reguläre Ausdrücke, Internet-Anbindung) und sehe keinen Anlass zu wechseln.

18.2 Test Driven Development

Mit dem Aufkommen der agilen Softwareentwicklung (*http://agilemanifesto.org/*), besonders befördert durch das Extreme Programming (XP)-Buch [Beck], nimmt die Bedeutung der testgetriebenen Entwicklung (englisch *TDD = Test Driven Development*) zu. Dabei wird zuerst ein Testfall spezifiziert, also *bevor* der zu prüfende Code überhaupt existiert. Der Testfall wird mit einem XUnit-Werkzeug ausgeführt und schlägt natürlich fehl. Anschließend entsteht nach und nach der Programmcode, wobei der Test wiederholt ausgeführt wird – solange, bis er bestanden wird. Im nachfolgenden Schritt wird der nächste Testfall spezifiziert und es wird der Ablauf wiederholt. Dabei werden auch alle vorherigen Testfälle ausgeführt, um sicher zu sein, dass eine Änderung nicht andere Programmteile ungünstig beeinflusst. Dieses Vorgehen hat einige Vorteile:

- Die Planung der Testfälle setzt eine gründliche Auseinandersetzung mit der Anforderungsdefinition und dem Design voraus. Dabei entstehende Unklarheiten und Widersprüche werden noch vor der Codierung beseitigt.
- Die Testfälle dienen als Spezifikation für die zu erstellende Software. Damit reduziert sich der nachträgliche Testaufwand.
- Es wird keine Software geschrieben, die nicht gebraucht wird. Ich habe gelegentlich beobachtet, dass bei dem Schreiben einer Klasse in guter Absicht möglicherweise nützliche Methoden gleich mitprogrammiert werden, ohne dass klar ist, ob sie jemals gebraucht werden. In der industriellen Wirklichkeit führt dies zu unnötigen Kosten – jede Methode muss zusätzlich getestet und dokumentiert werden.
- Die regelmäßigen Regressionstest garantieren bei Erfolg, dass die Software ein (durch die Testfälle definiertes) Mindestmaß an Qualität besitzt.
- Der in manchen Projekten am Ende zu beobachtende verstärkte Zeitdruck führt zu Aussagen wie »Zum ausführlichen Testen haben wir keine Zeit mehr!«, verbunden mit teuren Änderungen nach Auslieferung der Software. Die Wahrscheinlichkeit für solche Probleme ist bei der testgetriebenen Entwicklung wegen der ständig mitlaufenden Tests gering.

Ein testgetrieben entwickeltes System ist normalerweise von höherer Qualität als ein auf herkömmliche Art entwickeltes (Test nach Abschluss der Programmierung). Daraus folgt jedoch nicht unbedingt, dass es ausreichend getestet ist, nämlich dann, wenn die

Testfälle nicht systematisch auf Basis der Anforderungsdefinition und unter Verwendung guter Testverfahren entwickelt wurden. Das Qualitätsproblem verlagert sich von der Programmierung auf die Testfallspezifikation. Ein weiterer negativer Aspekt ist die isolierte Betrachtung des aktuellen Testfalls. Bei sofortiger Kenntnis aller Anforderungen an eine Komponente ist möglicherweise ein besseres Design möglich, und es müssten in eine Klasse nicht nachträglich viele Änderungen vorgenommen werden. Diese Argumente sprechen nicht gegen die testgetriebene Entwicklung, wenn man sie berücksichtigt. In Abschnitt 18.3.1 unten finden Sie ein ausführliches Beispiel für die testgetriebene Entwicklung einer Operatorfunktion.

18.3 Boost Unit Test Framework

Das Boost Unit Test Framework stellt Komponenten zur Verfügung, die das Schreiben von Tests, die Organisation von Tests und den kontrollierten Ablauf erlauben. Obwohl das Framework aus vielen Klassen besteht, ist die einfachste Schnittstelle zur Benutzung ein Satz von Makros. Mit wenigen Ausnahmen werde ich mich im Folgenden darauf beschränken, weil mit den Makros die wichtigsten Bereiche abgedeckt werden. Allen, die mehr wissen möchten, empfehle ich [Roz].

Installation

Das Boost Unit Test Framework wird automatisch mitinstalliert, wenn Sie die Boost Libraries entsprechend den Hinweisen auf der DVD einrichten.

Compilations- und Link-Optionen

Es wird davon ausgegangen, dass die Boost-Libraries installiert sind. Die Optionen sind:

- Statisches Linken[1] der Boost-Test-Library.
- Dynamisches Linken der Boost-Test-Library.
- »Header-only«: Damit ist gemeint, dass an einer Stelle die Quelltexte aller benötigten Funktionen mit einer `#include`-Anweisung eingebunden werden. Der Nachteil: Die Compilation dauert deutlich länger. Der Vorteil: Beim Linken muss keine Bibliothek angegeben werden. Alle Funktionen werden statisch zur ausführbaren Datei gebunden.
- »Auto-Linking«: Eine spezielle Möglichkeit des Frameworks für Microsoft Compiler, die zu linkenden Teile automatisch zu ermitteln.

Wegen der schnellen Compilation und der Verwendung des GNU C++-Compilers habe ich in den Beispielen die zweite Möglichkeit gewählt und die Makefiles entsprechend konfiguriert. Es genügt, in einem Shell-Fenster `make` einzugeben, um alles zu übersetzen und zu linken. In den Quellprogrammen sind die passenden Flags und `#include`-Anweisungen zu setzen, was allerdings sehr einfach ist, wie hier zu sehen:

[1] Zum Begriff »Linken« siehe Glossar Seite 953.

```
// Header-only (compiliert langsamer, statisches Linken)
#include<boost/test/included/unit_test.hpp>
// ... Rest der Datei
```

```
// dynamisches Linken erwünscht:
#define BOOST_TEST_DYN_LINK
#include<boost/test/unit_test.hpp>
// ... Rest der Datei
```

Testaufbau

Eine *Test-Suite* besteht aus einer Menge von *Testfällen*. Das Makro `BOOST_AUTO_TEST_SUITE( suite_name )` startet eine Test-Suite, mit `BOOST_AUTO_TEST_SUITE_END()` wird sie beendet. Jedes `BOOST_AUTO_TEST_CASE`-Makro fügt einen Testfall hinzu. Wenn `BOOST_TEST_MAIN` definiert ist, wird automatisch eine `main()`-Funktion erzeugt. Das folgende Beispiel zeigt eine Test-Suite mit zwei einfachen Testfällen. Im ersten wird die Funktion `length()` der Klasse `string` geprüft, im zweiten ihr Gleichheitsoperator. `BOOST_CHECK(bedingung)` prüft die Bedingung und gibt eine Fehlermeldung aus, wenn die Bedingung nicht erfüllt ist.

Listing 18.1: Einfache Test-Suite

```
#define BOOST_TEST_MAIN
// dynamisches Linken erwünscht:
#define BOOST_TEST_DYN_LINK
#include<boost/test/unit_test.hpp>

BOOST_AUTO_TEST_SUITE( einfacher_stringtest )

BOOST_AUTO_TEST_CASE( laenge )
{
   std::string s("xyz");
   BOOST_CHECK( s.length() == 3 );
}

BOOST_AUTO_TEST_CASE( gleichheits_operator )
{
   std::string s("abc");
   BOOST_CHECK( s == "abc" );
}

BOOST_AUTO_TEST_SUITE_END()
```

Weil `main()` automatisch erzeugt wird, ist die Test-Suite nach Compilation lauffähig. Die Ausgabe des Programms ist

```
Running 2 test cases...
*** No errors detected
```

Falls nun fälschlicherweise 4 statt 3 im ersten Test eingetragen wird, ist die Ausgabe

```
Running 2 test cases...
main.cpp(11): error in "laenge": check s.length() == 4 failed
*** 1 failure detected in test suite "Master Test Suite"
```

Die Zahl in Klammern gibt die Zeilennummer der Testdatei an, in der der Fehler auftrat. Wie Sie sehen, bedeutet ein Fehlschlagen des Tests nicht unbedingt, dass der Prüfling einen Fehler hat – es kann auch der Test falsch sein. Die Wurzel des Testfallbaums ist die »Master Test Suite«, die mehrere Test-Suiten enthalten kann. Der Name kann geändert werden, wenn statt `BOOST_TEST_MAIN` zum Beispiel `BOOST_TEST_MODULE neuer_name` geschrieben wird. Die Prüfung einer einfachen Bedingung gibt es in drei Varianten:

- `BOOST_WARN(Bedingung)`: Die Nichterfüllung der Bedingung wird nicht als Fehler gesehen und auch nicht als solcher gezählt. Bei der Standardeinstellung gibt es keine Meldung, erst wenn die ausführbare Datei mit einem passenden Log-Level aufgerufen wird, zum Beispiel `testprog.exe --log_level=warning`. Das Testprogramm läuft weiter.
- `BOOST_CHECK(Bedingung)`: Die Nichterfüllung der Bedingung wird als Fehler gesehen und gemeldet. Das Testprogramm läuft weiter.
- `BOOST_REQUIRE(Bedingung)`: Die Nichterfüllung der Bedingung wird als so kritischer Fehler gesehen, dass eine Fortsetzung des Tests nicht sinnvoll ist. Das Testprogramm bricht ab.

Die Informationen dieses Abschnitts genügen, um die Methode der testgetriebenen Entwicklung am Beispiel zu zeigen, wie Sie im nächsten Abschnitt sehen. Im Anschluss daran werden weitere Möglichkeiten des Frameworks vorgestellt und demonstriert.

18.3.1 Beispiel: Testgetriebene Entwicklung einer Operatorfunktion

In diesem Beispiel gehe ich von der Klasse `Datum` des Abschnitts 9.3 ab Seite 334 aus und nehme an, dass der `operator++()` geschrieben werden soll, also im Gegensatz zum genannten Abschnitt noch nicht existiert. Der erste Testfall ist die Prüfung, ob das Hochzählen des Tages funktioniert:

Listing 18.2: operator++()-Test

```
// cppbuch/k18/tdd/main.cpp
#define BOOST_TEST_MAIN
#define BOOST_TEST_DYN_LINK
#include<boost/test/unit_test.hpp>
#include"datum.h"

BOOST_AUTO_TEST_SUITE( datum_operator_plus_plus )

BOOST_AUTO_TEST_CASE( tag )
{
   Datum d(1, 1, 2011);
   for(int tag = 2; tag <= 31; ++tag) {
      ++d;
      BOOST_CHECK( d.tag() == tag );
   }
}

BOOST_AUTO_TEST_SUITE_END()
```

Dieser Testfall kann leicht mit der Funktion

Listing 18.3: Unvollständiger operator++(), Version 0

```
Datum& Datum::operator++() { // Datum um 1 Tag erhöhen
   ++tag_;
   return *this;
}
```

erfüllt werden. Der nächste Testfall prüft, ob das Weiterschalten des Monats funktioniert:

```
BOOST_AUTO_TEST_CASE( einfacher_monatswechsel )
{
   Datum d(31, 1, 2011);
   ++d;
   BOOST_CHECK( d.tag() == 1 );
   BOOST_CHECK( d.monat() == 2 );
}
```

Der Testfall scheitert natürlich. Die Ausgabe des Testprogramms ist:

```
  Running 2 test cases...
  main.cpp(25): error in "monatswechsel": check d.tag() == 1 failed
  main.cpp(26): error in "monatswechsel": check d.monat() == 2 failed
  *** 2 failures detected in test suite "master_testsuite"
```

Die Zahlen nach `main.cpp` geben die Zeilennummern des Testprogramms an, in denen der Fehler aufgetreten ist. Die geänderte Funktion löst scheinbar das Problem:

Listing 18.4: Unvollständiger operator++(), Version 1

```
Datum& Datum::operator++() {
    int tmp[] = {31,28,31,30,31,30,31,31,30,31,30,31};
    if(tag_ == tmp[monat_-1]) {
       tag_ = 1;
       ++monat_;
    }
    else
      ++tag_;
   return *this;
}
```

Funktioniert das für alle Monate? Das offensichtliche Schaltjahrproblem hebe ich für später auf. Der folgende Testfall gibt Auskunft:

```
BOOST_AUTO_TEST_CASE( alle_monatswechsel )
{
   int monate[] = {31,28,31,30,31,30,31,31,30,31,30,31};
   for(int monat = 1; monat <= 12; ++monat) {
      Datum d(monate[monat-1], monat, 2011);
      ++d;
      BOOST_CHECK( d.tag() == 1 );
      if(monat < 12) {
         BOOST_CHECK( d.monat() == monat+1 );
      }
      else {
         BOOST_CHECK( d.monat() == 1 );
      }
```

```
   }
}
```

Dieser Test bringt an den Tag, dass die letzte Lösung noch nicht in Ordnung ist. Ein kleiner Einschub direkt nach Inkrementieren des Monats beseitigt das Problem:

```
      if(monat_ == 13) {
        monat_ = 1;
        ++jahr_;
      }
```

Auf den Test, ob das Jahr korrekt hochgezählt wird, verzichte ich aus Platzgründen. Nun bleibt noch das Schaltjahr. Dazu hilft der folgende Testfall, in dem `BOOST_CHECK` durch `BOOST_REQUIRE` ersetzt wird, damit der Test bei der großen Anzahl getesteter Jahre nach dem ersten Fehler abbricht.

```
BOOST_AUTO_TEST_CASE( schaltjahr )
{
   for(int jahr = 1800; jahr < 2300; ++jahr) {
      bool schaltjahr = // durch 4 teilbar, aber nicht durch 100, es sei denn, durch 400
         ((jahr % 4 == 0) && (jahr % 100 != 0)) || (jahr % 400 == 0);
      Datum d(28, 2, jahr);
      ++d;
      if(schaltjahr) {
         BOOST_REQUIRE( d.tag() == 29 );
         BOOST_REQUIRE( d.monat() == 2 );
      }
      else {
         BOOST_REQUIRE( d.tag() == 1 );
         BOOST_REQUIRE( d.monat() == 3 );
      }
   }
}
```

Nun könnte man im `operator++()` das Schaltjahr explizit berücksichtigen, ähnlich wie im Testfall. Geschickter ist es jedoch, die Prüfung auf die Eigenschaft »Schaltjahr« durch eine eigene, unabhängig aufrufbare Funktion `istSchaltjahr()` zu realisieren, die direkt nach der Definition des Feldes `tmp` aufgerufen wird:

Listing 18.5: operator++()

```
Datum& Datum::operator++() {
    int tmp[] = {31,28,31,30,31,30,31,31,30,31,30,31};
    if(istSchaltjahr()) {
       tmp[1] = 29;
    }
    if(tag_ == tmp[monat_-1]) {
       tag_ = 1;
       ++monat_;
       if(monat_ == 13) {
          monat_ = 1;
          ++jahr_;
       }
    }
```

```
    else {
       ++tag_;
    }
   return *this;
}
```

Damit ist `operator++()` vollständig. Wie sinnvoll ein Regressionstest ist, zeigt sich an dem folgenden Optimierungsversuch. Bei jedem Aufruf wird das Array `tmp` neu auf dem Laufzeit-Stack abgelegt. Um das zu vermeiden, wird es als `static` deklariert.

```
Datum& Datum::operator++() {
    // static vermeidet Neuinitialisierung:
    static int tmp[] = {31,28,31,30,31,30,31,31,30,31,30,31}; // ???
    if(istSchaltjahr()) {
       tmp[1] = 29;
    }
// usw.
```

Überlegen Sie, ob das wirklich eine gute Idee ist, bevor Sie weiterlesen. Was geschieht? Der Test scheitert! Der Grund liegt in der `if`-Anweisung, die `tmp` dauerhaft verändert. Wenn beim nächsten Test kein Schaltjahr vorliegt, wird dennoch 29 angenommen. Der wiederholte Testfall `schaltjahr` hat sehr schnell gezeigt, dass die mal eben eingeführte »Verbesserung« fehlerhaft ist. Das `static`-Feld ist zudem nicht thread-sicher, weil auf das Feld von zwei Programmen gleichzeitig zugegriffen werden kann. Korrekturen: Feld als `const` anlegen, um es thread-sicher zu machen, und dann (natürlich) nur lesend darauf zugreifen. Ergebnis:

Listing 18.6: operator++() mit static-Feld

```
Datum& Datum::operator++() {
    static const int tmp[] = {31,28,31,30,31,30,31,31,30,31,30,31};
    int letzterTagImMonat = tmp[monat_-1];
    if(monat_ == 2 && istSchaltjahr()) {
       letzterTagImMonat = 29;
    }
    if(tag_ == letzterTagImMonat) {
       tag_ = 1;
       ++monat_;
       if(monat_ == 13) {
          monat_ = 1;
          ++jahr_;
       }
    }
    else {
       ++tag_;
    }
   return *this;
}
```

Die isolierte Sicht auf einen Testfall versperrt möglicherweise den Blick auf eine andere Möglichkeit. Wenn es die Anforderung gibt (und sie schon bekannt ist), dass der Konstruktor der Klasse `Datum` und die Methode `set(int t, int m, int j` eine Exception

werfen sollen, wenn ein ungültiges Datum verwendet wird, liegt es nahe, eine Prüfmethode `istGueltigesDatum(int t, int m, int j)` zu schreiben, die von allen anderen Methoden aufrufbar ist. Sie finden sie auf Seite 336. Davon profitiert auch `operator++()`, der nun deutlich kürzer geschrieben werden kann. Es wird ausgenutzt, dass es sich um ein Monatsende handeln muss, wenn das Datum nach dem Hochzählen des Tages ungültig wird.

Listing 18.7: Kürzerer operator++()

```
Datum& Datum::operator++() {
   ++tag_;
   if(!istGueltigesDatum(tag_, monat_, jahr_)) { // Monatsende erreicht?
     tag_ = 1;
     if (++monat_ > 12) {
        monat_ = 1;
        ++jahr_;
     }
   }
   return *this;
}
```

18.3.2 Fixture

Im JUnit-Test-Sprachgebrauch bezeichnet Fixture durchzuführende vorbereitende Maßnahmen, mit `setUp()` aufgerufen, und nachbereitende Maßnahmen (Aufräumarbeiten), mit dem Namen `tearDown()` verbunden. Diese Maßnahmen können zum Beispiel dafür sorgen, dass vor jedem Testfall dieselben Bedingungen herrschen. Die Durchführung eines Testfalls besteht aus vier Phasen:

- Vorbereitende Maßnahmen (setUp)
- Test durchführen
- Ergebnis prüfen und protokollieren
- Aufräumarbeiten (tearDown)

Der Vorteil der Verwendung von Fixtures besteht in der Trennung der vor- und nachbereitenden Maßnahmen vom eigentlichen Test. Das C++-Prinzip »Resource Acquisition Is Initialization« (RAII, siehe Glossar) erlaubt es, auf die Methoden `setUp()` und `tearDown()` zu verzichten. An ihre Stelle treten Konstruktor und Destruktor. Ein einfaches Beispiel: Wenn ein `Datum`-Objekt dynamisch angelegt werden soll, lässt sich das leicht mit einem `shared_ptr<Datum>` als Fixture realisieren:

```
BOOST_AUTO_TEST_CASE( konstruktor_mit_fixture )
{
   // shared_ptr<Datum> als Fixture
   std::shared_ptr<Datum> p(new Datum(1, 1, 2011)); // Konstruktor (setUp)
   BOOST_CHECK( p->jahr() == 2011 );
}  // delete automatisch durch shared_ptr-Destruktor (tearDown)
```

Im folgenden Abschnitt wird ein Fixture auf globaler Ebene zum Öffnen und Schließen einer Log-Datei eingesetzt.

18.3.3 Testprotokoll und Log-Level

Es gibt zwei Möglichkeiten, eine Datei als Testprotokoll zu erzeugen. Die erste ist, die Bildschirmausgabe in eine Datei umzuleiten. Dabei kann der Log-Level als Parameter übergeben werden. Beispiel:

```
testprog.exe --log_level=error
```

Anstelle der Übergabe in der Kommandozeile kann dasselbe mit einer entsprechenden Definition der Umgebungsvariablen `BOOST_TEST_LOG_LEVEL` erreicht werden. Die Spalte eins der Tabelle 18.1 enthält weitere Werte für den Log-Level.

Die zweite Möglichkeit ist, per Programm eine Log-Datei anzulegen und den Log-Level zu definieren. Dazu wird eine Klasse mit den entsprechenden Einstellungen definiert:

```
class LogKonfiguration {
public:
   LogKonfiguration()
      : test_log("test.log") {                          // Dateiname
      boost::unit_test::unit_test_log.set_stream(test_log);
      boost::unit_test::unit_test_log.set_threshold_level(
                     boost::unit_test::log_test_units); // Log-Level
   }
   ~LogKonfiguration() { // schließen und zurücksetzen
      test_log.close();
      boost::unit_test::unit_test_log.set_stream(std::cout);
   }
private:
   std::ofstream test_log;
};
```

Ein Objekt dieser Klasse wird mit

```
BOOST_GLOBAL_FIXTURE( LogKonfiguration );
```

in der Testdatei als globales Fixture erzeugt. Die Spalte zwei der Tabelle 18.1 enthält die möglichen, per Programm setzbaren Log-Level. Ein Log-Level schließt alle Ausgaben der Log-Level der darunter liegenden Tabellenzeilen ein.

Tabelle 18.1: Log-Level

Kommandozeile bzw. BOOST_TEST_LOG_LEVEL	per Programm festlegbar	protokolliert wird
success	log_successful_tests	alles
all		alles
test_suite	log_test_units	Suites und Testfälle
message	log_messages	BOOST_TEST_MESSAGE-Nachrichten
warning	log_warnings	Warnungen
error	log_all_errors	Fehler
cpp_exception	log_cpp_exception_errors	nicht gefangene Exceptions
system_error	log_system_errors	Systemfehler
fatal_error	log_fatal_errors	von Testmakros erkannte kritische Fehler oder fatale Systemfehler
nothing	log_nothing	nichts

Wenn eine Log-Datei mit dem Fixture angelegt wird, werden Log-Level-Einstellungen per Kommandozeilen-Option ignoriert. Die per Programm einstellbaren Log-Level sind im Namespace `boost::unit_test`.

18.3.4 Prüf-Makros

Sie haben schon verschiedene Prüf-Makros kennen gelernt, es gibt aber noch mehr für verschiedene Zwecke. Vielen der Makros ist gemeinsam, dass es sie für die drei Level `WARN`, `CHECK` und `REQUIRE` gibt, die oben auf Seite 530 beschrieben werden. In diesen Fällen wird im Folgenden einfach `<level>` als Platzhalter für eine der drei Varianten geschrieben, ohne weiter auf sie einzugehen.

Hinweis

Für alle Beispiele wird die Klasse `Datum` aus Abschnitt 9.3 zugrundegelegt, einschließlich der Lösungen der Übungsaufgaben. Die Klasse `Datum` und Beispiele befinden sich im Verzeichnis *cppbuch/k18/datum*. Die Testdatei ist *main.cpp*.

BOOST_<level>(Bedingung)

prüft die Bedingung (Beispiel siehe oben, Seite 529).

BOOST_<level>_MESSAGE(Bedingung, Text)

liefert bei ungültiger Bedingung den übergebenen Text.

BOOST_ERROR(Text) und BOOST_FAIL(Text)

verhalten sich wie `BOOST_CHECK_MESSAGE(false, Text)` und `BOOST_REQUIRE_MESSAGE(false, Text)`.

BOOST_<level>_PREDICATE(Prädikat, Argumente)

prüft das Prädikat. Im Unterschied zu `BOOST_<level>()` wird eine Funktion oder ein Funktor, gegebenenfalls mit Parametern, übergeben. Jeder Parameter muss von runden Klammern umschlossen sein. Beispiel für die Funktion `istSchaltjahr(int)`:

```
BOOST_AUTO_TEST_CASE( schaltjahrtest )
{
   BOOST_CHECK_PREDICATE( istSchaltjahr, (2012) );
}
```

BOOST_TEST_MESSAGE(Text)

ist eine Dokumentationshilfe. Das Makro gibt den übergebenen Text aus. Ein Beispiel sehen Sie unten bei `BOOST_<level>_THROW`.

BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES(testfall, Anzahl)

definiert die Anzahl der zu erwartenden, das heißt, absichtlich hervorgerufenen Fehler für den Testfall. Die ausgegebene Fehleranzahl eines Testfalls wird um diese Zahl reduziert. Ein Beispiel sehen Sie unten bei `BOOST_<level>_THROW`.

BOOST_<level>_THROW (Ausdruck, Exceptiontyp)

prüft, ob der Ausdruck eine Exception des Typs `Exceptiontyp` oder einer davon abgeleiteten Klasse wirft. Falls keine Exception geworfen wird, gibt es einen Fehler. Beispiel:

```
// in der letzten Zeile provozierten Fehler nicht mitzählen:
BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES( konstruktor_ungueltiges_Datum, 1 )

BOOST_AUTO_TEST_CASE( konstruktor_ungueltiges_Datum )
{
   BOOST_CHECK_THROW( Datum(30, 2, 2011), UngueltigesDatumException );
   BOOST_TEST_MESSAGE( "Gegenprobe: " ); // soll fehlschlagen, kein Fehler
   BOOST_CHECK_THROW( Datum(1, 2, 2011), UngueltigesDatumException );
}
```

BOOST_<level>_EXCEPTION (Ausdruck, Exceptiontyp, Prüffunktion)

wirkt wie `BOOST_<level>_THROW` mit dem Unterschied, dass die geworfene Exception der Prüffunktion übergeben wird. Gibt diese Funktion `false` zurück, ist der Test fehlgeschlagen; gibt sie `true` zurück, ist er bestanden. Die Prüffunktion kann zum Beispiel prüfen, ob der Text der Exception korrekt ist oder ob sie überhaupt vom richtigen Typ ist. Um Letztes zu zeigen, seien die folgende Klasse und Funktion vorausgesetzt:

```
class FalscheException : public std::exception {};

bool checkException(const std::exception& e) {
   return typeid(e) == typeid(UngueltigesDatumException);
}
```

Der Test

```
BOOST_CHECK_EXCEPTION( Datum(30, 2, 2011), UngueltigesDatumException,
                                          checkException);
```

wird bestanden. Wenn aber das Makro wie gezeigt verändert wird, schlägt der Test fehl.

```
BOOST_CHECK_EXCEPTION( Datum(30, 2, 2011), FalscheException, checkException);
```

BOOST_<level>_NO_THROW(Ausdruck)

schlägt fehl, wenn der Ausdruck eine Exception wirft. Zum Beispiel schlägt

```
BOOST_CHECK_NO_THROW( Datum(30, 2, 2011));
```

fehl. Es ist möglich, mehrere Anweisungen anstelle des Ausdrucks auszuführen, wenn sie in einen `do-while(0)`-Block gepackt werden:

```
BOOST_CHECK_NO_THROW( do {
                        int tag = 30;
                        Datum(tag, 2, 2011);
                      } while(0));
```

BOOST_TEST_CHECKPOINT(Nachricht)

Dieses Makro ist sehr hilfreich zum Aufspüren von Fehlern. Die Nachricht kann so gewählt werden, dass ein direkt zu einem Fehler führender Wert ausgegeben wird. In allen anderen Fällen tut das Makro nichts. Ein Beispiel:

```
BOOST_AUTO_TEST_CASE( konstruktor_ungueltiges_Datum_checkpoint )
{
   for(int tag = 1; tag < 30; ++tag) {
     BOOST_TEST_CHECKPOINT( "Datum mit Tag = " << tag );
     Datum d(tag, 2, 2011); // Exception bei tag == 29: Fehler
   }
}
```

Am Ende der Schleife, wenn `tag` den Wert 29 annimmt, gibt es eine Exception, weil das zu erzeugende Datum ungültig ist. Das Testprogramm erzeugt folgende Ausgabe:

```
unknown location(0): fatal error in "konstruktor_ungueltiges_Datum_check-
point": std::runtime_error: 29.02.2011 ist ein ungueltiges Datum!
main.cpp(132): last checkpoint: Datum mit Tag = 29
```

Die letzte Zeile weist auf den fehlerhaften Wert hin. Ohne das Makro `BOOST_TEST_CHECKPOINT` würde diese Zeile fehlen. In diesem Beispiel ist der Fehler leicht zu sehen. In anderen Fällen, zum Beispiel wenn sehr viele Daten in einer Schleife eingelesen und verarbeitet werden, ist es hilfreich, wenn ein fehlererzeugender Datensatz dokumentiert wird.

Bei meinen Tests unter Linux und Windows XP funktionierte `BOOST_TEST_CHECKPOINT` wie erwartet. Windows Vista, Business Edition, schmeckte das Programm nicht: Es tat sich ein Fenster auf mit der Information »testdatum.exe funktioniert nicht mehr. Es wird nach einer Lösung gesucht.« Natürlich wird keine gefunden, und das Programm kann durch Button-Klick beendet werden.

Relationale Makros

Diese Makros vergleichen zwei Werte `links` und `rechts` mit relationalen Operatoren. Der Unterschied zu `BOOST_<level>` werden im Falle eines `false`-Ergebnisses die verglichenen Werte ausgegeben. Die Tabelle 18.2 zeigt die relationalen Makros.

Tabelle 18.2: Relationale Makros

Makro	Wirkung wie
`BOOST_<level>_EQUAL(links, rechts)`	`BOOST_<level>(links == rechts)`
`BOOST_<level>_NE(links, rechts)`	`BOOST_<level>(links != rechts)`
`BOOST_<level>_GE(links, rechts)`	`BOOST_<level>(links >= rechts)`
`BOOST_<level>_GT(links, rechts)`	`BOOST_<level>(links > rechts)`
`BOOST_<level>_LE(links, rechts)`	`BOOST_<level>(links <= rechts)`
`BOOST_<level>_LT(links, rechts)`	`BOOST_<level>(links < rechts)`

Die Makros `BOOST_<level>_EQUAL` und `BOOST_<level>_NE` sollen *nicht* für `float`- und `double`-Werte genommen werden, weil bitweise verglichen wird. Weitere Informationen zum Vergleich von `float`- und `double`-Werten finden Sie auf Seite 561. Dazu passende Makros finden Sie unten.

BOOST_<level>_CLOSE(Links, Rechts, Toleranz)

Dieses Makro prüft, ob die ersten zwei Werte relativ (nicht absolut) dicht beieinanderliegen. Der dritte Wert gibt die Toleranz in *Prozent* an.

```
BOOST_CHECK_CLOSE( 1.25, 1.27, 1.0); // schlägt fehl
BOOST_CHECK_CLOSE( 1.25, 1.27, 2.0); // ok, Differenz ist kleiner als 2 %
```

Nach *boost/test/floating_point_comparison.hpp* ist der Test dann erfolgreich, wenn

```
(D/fabs(links) <= fabs(toleranz)) && (D/fabs(rechts) <= fabs(toleranz))
```

gilt, wobei `D` der Absolutbetrag der Differenz `links - rechts` ist. `links` und `rechts` müssen vom selben Typ sein. Das in Abschnitt 20.2.1 angesprochene Overflow-/Underflow-Problem wird durch Einsetzen des Maximalswerts für den Datentyp bzw. 0 gelöst.

BOOST_<level>_CLOSE_FRACTION(Links, Rechts, Toleranz)

arbeitet wie `BOOST_<level>_CLOSE`, nur dass die Toleranz anders definiert wird. Der Test ist erfolgreich, wenn `fabs(toleranz) > (fabs(rechts/links) - 1))` ist.

BOOST_<level>_SMALL(Wert, Toleranz)

ist erfolgreich, wenn der Betrag des Werts > als der Betrag der Toleranz ist.

BOOST_<level>_EQUAL_COLLECTIONS

vergleicht zwei Container (Collections) auf gleiche Inhalte, ähnlich wie der Algorithmus `mismatch` von Seite 692. Im Unterschied zu Letzterem muss das Ende des zweiten Bereichs angegeben werden. Unterschiedliche Elemente werden gemeldet. Dabei können Container der Standardbibliothek und einfache Arrays verwendet werden, sogar gemischt, wie das Beispiel zeigt:

```
BOOST_AUTO_TEST_CASE( eq_collection )
{
   std::vector<int> v;
   for(int i=1; i <= 8; ++i)
      v.push_back(i);
   int arr [] = { 1, 2, 3, 5, 5, 6, 7, 9};
    BOOST_CHECK_EQUAL_COLLECTIONS(v.begin(), v.end(), arr, arr+8 );
}
```

BOOST_<level>_BITWISE_EQUAL(a, b)

prüft, ob alle Bits der zwei Werte übereinstimmen. Falls nicht, schlägt der Test fehl, und es werden alle nicht übereinstimmenden Positionen ausgegeben.

18.3.5 Kommandozeilen-Optionen

Die Steuerung des Log-Levels mit einer Kommandozeilen-Option wird oben auf Seite 535 beschrieben. Daneben gibt es weitere Optionen, von denen die wichtigsten in der Tabelle 18.3 genannt werden. Alle Optionen sind auch durch entsprechende Umgebungsvariablen einstellbar. Die Spalte zwei zeigt die möglichen Werte. Der voreingestellte Wert ist jeweils zuerst genannt.

Tabelle 18.3: Kommandozeilen-Optionen

Option	Werte	Bedeutung
`auto_start_dbg`	no, yes	Bei Systemfehler Debugger starten
`build_info`	no, yes	Anzeige der Compiler-Version
`catch_system_errors`	yes, no	no: Systemfehler werden nicht aufgefangen. Sie können damit von einem übergeordneten Programm (GUI) analysiert werden.
`detect_memory_leak`	1, 0, > 1	nur für MS-Compiler (Details siehe [Roz])
`detect_fp_exceptions`	no, yes	Floating Point-Exceptions fangen (falls vom System unterstützt)
`log_format`	HRF XML	HRF = human readable format (ASCII-Text) für weitere automatisierte Verarbeitung
`report_format`	dasselbe wie `log_format`	
`output_format`	dasselbe wie `log_format`, hat aber ggf. Vorrang vor den beiden anderen	
`random`	0 1 z >1	Tests der Reihe nach ausführen zufällige Reihenfolge, basierend auf der aktuellen Zeit zufällige Reihenfolge mit z als Initialisierungswert (random seed)
`result_code`	yes, no	no: es wird stets 0 zurückgegeben (bei Einbindung in GUI von Interesse)

`run_test`	durchzuführende Testfälle und -suiten. Wildcards sind erlaubt. Beispiele:
	`testprog --runtest=test_a`
	`testprog --runtest=test_suite,test_c,xtest*`	

`show_progress`	no, yes	Anzeige eines Fortschrittsbalkens
`log_level`	siehe Seite 535	

Die entsprechende Umgebungsvariable ergibt sich aus der Option, indem die Option in Großschreibung an die Zeichenkette `BOOST_TEST_` gehängt wird. So gehört zu der Option `show_progress` die Umgebungsvariable `BOOST_TEST_SHOW_PROGRESS`. Die einzige Ausnahme ist nach [Roz] die Umgebungsvariable `BOOST_TESTS_TO_RUN` zur Option `run_test`.