**Test Driven Development**

**mit C++ (gtest / gmock)**

OOA

OOD

OOP

OOD

OOA

OOP

OOA

OOP

OOD

OOP

OOA

OOD

OOD

OOP

OOA

OOP

OOD

OOA

Johannes Nowak

**Inhalt**

[1 Einführung 6](#_Toc459298022)

[1.1 Einige Vorurteile 7](#_Toc459298023)

[1.2 Einige Begriffe 9](#_Toc459298024)

[1.3 Test-Driven Development 11](#_Toc459298025)

[1.4 Aufbau des Skripts 12](#_Toc459298026)

[1.5 Beispiele zum Thema Testgetriebene Entwicklung 13](#_Toc459298027)

[1.6 Beispiel zum Thema Refactoring 14](#_Toc459298028)

[1.7 Organisation der Beispiel-Projekte 16](#_Toc459298029)

[2 TDD – Ein einfaches Beispiel 18](#_Toc459298030)

[2.1 Start 19](#_Toc459298031)

[2.2 Adding 20](#_Toc459298032)

[2.3 Get / Set 22](#_Toc459298033)

[2.4 Index-Operator 24](#_Toc459298034)

[2.5 Refactoring 24](#_Toc459298035)

[2.6 Realloc 26](#_Toc459298036)

[2.7 Insert 27](#_Toc459298037)

[2.8 Refactoring 29](#_Toc459298038)

[2.9 Copy 30](#_Toc459298039)

[2.10 Assingment 31](#_Toc459298040)

[2.11 Ausgabe-Operator 33](#_Toc459298041)

[2.12 Resultate und Konsequenzen 34](#_Toc459298042)

[3 Testing – Das gtest Framework 35](#_Toc459298043)

[3.1 Start 36](#_Toc459298044)

[3.2 Exceptions 40](#_Toc459298045)

[3.3 Objects 42](#_Toc459298046)

[3.4 Complex Objects 44](#_Toc459298047)

[3.5 Fixtures 48](#_Toc459298048)

[3.6 Fixtures – Details 50](#_Toc459298049)

[3.7 Assertions 52](#_Toc459298050)

[3.8 Parameter 55](#_Toc459298051)

[3.9 Listeners 58](#_Toc459298052)

[3.10 Flags 60](#_Toc459298053)

[4 Testing – Übungen 61](#_Toc459298054)

[4.1 Stack 62](#_Toc459298055)

[4.2 Trimmer 65](#_Toc459298056)

[4.3 CSVReader 67](#_Toc459298057)

[4.4 Statemachine 69](#_Toc459298058)

[4.5 CommandHandling 73](#_Toc459298059)

[4.6 SharedPtr 76](#_Toc459298060)

[4.7 Resultate 80](#_Toc459298061)

[5 Mocking – Das gmock-Framework 84](#_Toc459298062)

[5.1 Assertions 85](#_Toc459298063)

[5.2 Arrangements 91](#_Toc459298064)

[6 Mocking – das Fakeit-Framework 95](#_Toc459298065)

[6.1 Assertions 96](#_Toc459298066)

[6.2 Arrangements 98](#_Toc459298067)

[7 Mocking - Übungen 99](#_Toc459298068)

[7.1 AcountService 100](#_Toc459298069)

[7.2 GroupChangeReader 104](#_Toc459298070)

[8 Test Driven Development: Parser 110](#_Toc459298071)

[9 Test Driven Development: logische Schaltungen 113](#_Toc459298072)

[10 Refactoring – Gruppenwechsel 117](#_Toc459298073)

[11 DbAccessTester 121](#_Toc459298074)

[12 Literatur 136](#_Toc459298075)

Zum wach- und warm-werden:

void **demo**() {

int tab[10];

tab[2] = 42;

cout << tab[2] << endl;

3[tab] = 77;

cout << \*(3 + tab) << endl;

}

Wir die Funktion überhaupt übesetzt? Wenn ja: was sind die Ausgaben?

void **demo**() {

int n = 2;

while(n --> 0) {

cout << n << endl;

}

}

Kennen Sie den Count-Down-Operator --> etwa noch nicht?

# Einführung

Die Einführung beginnt mit einigen grundsätzlichen Bemerkungen zum Thema "Test" und "testgetriebener Software-Einwicklung".

Ein kleines Beispiel wird dann in Thema TDD einführen: in mehreren aufeinander aufbauenden Schritten wird eine Datencontainer-Klasse entwickelt: eine Klasse namens IntArray.

Zunächst aber zwei Zitate von Kent Beck, dem großen Philosophen der testgetriebenen Software-Entwicklung (siehe den Literaturhinweis am Ende des Skripts).

Kent Beck beschreibt den "rhythm of Test-Driven Develepment":

1. *Quickly add a test.*
2. *Run all tests and see the new one fail.*
3. *Make a little change.*
4. *Run all tests and see them all succeed.*
5. *Refactor to remove duplication.*

Und ein etwas längeres Zitat:

*The goal is clean code that works…*

*Clean code that works is out of the reach of even the best programmers some of the time, and out of the reach of most programmers (like me) most of the time.*

*Divide and conquer…*

*First we'll solve the "that works" part of the problem. Then we'll solve the "clean code" part.*

*This is the opposite of architecture-driven development, where you solve "clean code" first, then scramble around trying to integrate into the design the things you learn as you solve the "that works" problem.*

## Einige Vorurteile

Das Thema "Testen von Software" ist mit einer Reihe von Vorurteilen verbunden (die folgende Aufzählung ist dem Buch von Johannes Link entnommen: "Softwaretests mit gmock"):

* Ich habe keine Zeit zum Testen.
* Testen von Software ist langweilig und stupide.
* Mein Code ist praktisch fehlerfrei, auf jeden Fall gut genug.
* Die Testabteilung testet. Die können das eh viel besser.

Hier einige einfache Klarstellungen:

Fast jeder Code enthält Fehler. Wer keine Zeit zum Testen hat, entdeckt diese Fehler nicht. Der Code ist unstabil. Es werden sich Fehlermeldungen häufen. Die Bearbeitung dieser Fehlermeldungen dauert (Debugging etc). Der Versuch, Zeit zu sparen, führt zum Gegenteil: dass mehr Zeit benötigt wird.

Testen – richtig verstanden – ist alles andere als langweilig und stupide. Testen ist ebenso anspruchsvoll wie die Erstellung des eigentlichen Programmcodes. Außerdem kann man besser schlafen, wenn die Software, die man erstellt hat, die Tests bestanden hat.

Jeder Code wird irgendwann einmal angefasst. Die Funktionalität von Software wird erweitert oder geändert. Wer garantiert, dass solche Änderungen und Erweiterungen keine unerwünschten Seiteneffekte haben?

Der Systemtest – und nur damit kann sich die Testabteilung befassen – kann keine Komponententests (Unit-Tests) ersetzen. Die Komponenten können nur von demjenigen getestet werden, der sie selbst geschrieben hat. Solche Komponenten müssen isoliert vom Gesamtsystem getestet werden können. Wird auf solche Unit-Tests verzichtet, können Fehler recht kostspielig werden – Fehler, die beim Unit-Test frühzeitig und viel billiger behoben werden könnten.

Resultate:

* Die Zeit, die zum Testen benötigt wird, ist keine verlorene Zeit. Testen hilft dabei, Zeit zu sparen.
* Testen von Software ist anspruchsvoll und kreativ.
* Code mag heute fehlerfrei sein; er sollte auch nach einer Änderung / Erweiterung weiterhin fehlerfrei sein.
* Units können nur vom Entwickler selbst getestet werden.

## Einige Begriffe

### Automatisierung von Tests

Ein Test soll zeigen, dass die tatsächlich von der Software produzierten Ergebnisse den erwarteten Ergebnissen entsprechen – oder eben: nicht entsprechen. Im ersten Falle sind die Tests erfolgreich, im letzteren Falle sind sie fehlgeschlagen (auch dies ist zunächst ein durchaus "positives" Resultat). Man kann die von der Software produzierten Ergebnisse natürlich bei jedem Test "manuell" mit den erwarteten Ergebnissen vergleichen. Es leuchtet aber sofort ein, dass solche Tests nicht praktikabel sind. U.a. muss man Tests wiederholt ausführen können. Sie sollten also derart geschrieben werden, dass sich automatisch ablaufen können – dass also der Vergleich zwischen den tatsächlichen und den erwarteten Ergebnissen automatisiert ist. Dann können solche Tests natürlich auch ohne großen Aufwand beliebig häufig wiederholt werden.

### Komponententest (Unit-Test)

Ein Komponententest bezieht sich auf eine isolierte Komponente. Was dabei als Komponente verstanden wird, ist natürlich Definitionssache. Es kann sich um eine einzelne Methode handeln, um eine Klasse, um ein Subsystem. Diese Einheiten sollten auf jeden Fall unabhängig voneinander getestet werden können.

Ein Interpreter z.B. enthält eine Scanner- und eine Parser-Komponente. Zunächst sollte der Scanner unabhängig von anderen Bestandteilen des Interpreters getestet werden können. Auch der Parser sollte unabhängig von weiteren Bestandteilen getestet werden können. Aber der Parser wird abhängig sein vom Scanner. Also wird der Parser-Test einen erfolgreichen Scanner-Test voraussetzen… (Oder sollte – zum Testen des Parsers – der Scanner nicht eher "simuliert", "gemockt" werden?) Komponenten sollten von dem Entwickler dieser Komponenten getestet werden.

### Integrations- / Interaktionstests

Integrationstest beziehen sich auf das Zusammenspiel aller Komponenten eines Systems. Sie setzen erfolgreiche Komponententests voraus. Auch solche Tests sollten natürlich automatisiert ablaufen. Die Abstände, in denen solche Tests laufen, sind zwar größer als beim Unit-Test (dort sollten die Intervalle möglichst klein sein), sollten aber dennoch relativ klein sein (z.B. jeden Tag ein Integrations-Test).

### Akzeptanz-Tests

Akzeptanz-Tests geben dem Kunden resp. dem Management das für die Überwachung des Projektfortschritts nötige Feedback. Sie werden vom Kunden spezifiziert – nur dieser weiß, was die Software "nach außen hin" leisten muss. Es geht hier also um die Funktionalität des Gesamtsystems aus Sicht des Benutzers. Auch solche Akzeptanz-Tests können teilweise automatisiert werden. Dabei ist es natürlich wieder der Entwickler, der die Spezifikation des Kunden in solche automatische Tests technisch umsetzen muss.

### White-Box / Black-Box-Tests

Beim White-Box-Test wird die Kenntnis der internen Implementierung eines Moduls (einer Unit) vorausgesetzt. Solche Tests werden u.a. dann verwendet, wenn es um das Mocken von Objekten geht, von welchen die zu testende Unit abhängig ist (siehe hierzu die Kapitel zum Thema Mocking). Beim Black-Box-Tests wird dagegen von der Implementierung einer Unit komplett absehen. Es wird nur das nach außen sichtbare Verhalten einer Unit getestet.

### Testabdeckung

Hier geht's um die Frage, wann ein Test "ausreichend" ist. Ein Test eines komplexen Systems kann – sofern seine Kosten vertretbar sein sollen - niemals komplett vollständig sein. Es geht dann um die Frage, was getestet werden soll. Z.B.: Welche möglichen Verzweigungen des Codes müssen getestet werden (alle können nicht getestet werden). Welche grenzwertigen Situationen müssen getestet werden?

## Test-Driven Development

Das Thema Test-Driven Development ist eng verknüpft mit den Konzepten "Extreme Programming (XP)" resp. "Agile Softwareentwicklung".

XP geht davon aus, dass ein komplexes System ohnehin nicht vollständig entworfen und spezifiziert werden kann. Solche Systeme können nur evolutionär entwickelt werden. Und dabei gewinnt die eigentliche Programmierung wieder einen zentralen Stellenwert: war sie bislang nur als "Hilfstätigkeit" begriffen worden, welche einfach nur einen gegebene Spezifikation umsetzt, so wird sie nun der eigentliche Dreh- und Angelpunkt der Entwicklung.

Das System wird in relative keine Einzelteile zerlegt, welche in einem überschaubaren Rahmen (1 bis 3) Wochen implementiert werden können. Diese Einheiten werden wiederum in kleinere Einheiten zerlegt, welche unmittelbar implementiert werden. Am Resultat dieser Implementierung kann unmittelbar ihr Erfolg oder ihr Misserfolg abgelesen werden. Und zu dieser Implementierung – das ist entscheidend – gehört immer auch das gleichzeige Schreiben (oder das Anpassen!) von Test-Code. Die Implementierung kann somit immer sofort verifiziert werden. "Ohne diesen Test gilt auch die Implementierung als nicht vorhanden" (Joh. Link).

Die Entwicklung der Komponenten geschieht in folgenden Schritten:

Zunächst wird Test-Code geschrieben. Dieser Code wird genau diejenigen Klassen benutzen, welche zum Schreiben des Test-Codes noch gar nicht existieren. Das erscheint zunächst merkwürdig: Wie kann ein Test einer Klasse geschrieben werden, welche noch nicht existiert? Natürlich kann er geschrieben werden! – er kann nur nicht kompiliert werden. Aber indem der Test-Code geschrieben wird, wird doch überhaupt erst klar, was die zu schreibende Klasse eigentlich leisten soll.

Dann wird die zu testende Klasse geschrieben – mit minimalen Aufwand. Es geht zunächst einmal nur darum, den Compiler zufriedenzustellen. Der Test-Code ist dann übersetzbar. Aber seine Ausführung wird scheitern. Im nächsten Schritt wird die zu testende Klasse derart erweitert (wiederum mit minimalem Aufwand), dass der Test gelingt. Dann wird der Test-Code erweitert – mit dem erklärten Ziel, seine Ausführung wieder scheitern zu lassen. Die zu testende Klasse wird erweitert, damit der Test wieder gelingt. Der Test-Code wird um neue Funktionalitäten erweitert. Die oben beschriebene Schrittfolge wird erneut durchlaufen.

## Aufbau des Skripts

* Im Kapitel 2 wird die Methode der testgetriebenen Entwicklung bereits in groben Zügen vorgestellt – anhand eines recht einfachen Beispiels.
* Im Kapitel 3 wird das gtest-Testframework vorgestellt.
* Kapitel 4 enthält einige Übungen, in denen es darum geht, Klassen "im nachhinein" zu testen. Anhand dieser Übungen sollte ein Gespür dafür entwickelt werden können, wie brauchbare Testklassen beschaffen sind.
* Kapitel 5 zeigt, was es mit Mocking auf sich hat. Dabei wird gmock genutzt.
* Im Kapitel 6 wird ein alternatives Mock-Werkzeug vorgestellt: das Werkzeug "fakeit".
* Kapitel 7 enthält einigen Übungen zum Thema Mocking.
* Im Kapitel 8 und 9 geht's schließlich um die "Philosophie" und Praxis der testgetriebenen Software-Entwicklung – also um das eigentliche Kernthema des Seminars. Diese Kapitel enthalten zwei Aufgaben, die mittels TDD schrittweise gelöst werden können.
* Im Kapitel 10 geht's um das Thema Refactoring.
* Im Kapitel 11 geht's um das Testen Datenbank-basierter Anwendungen. Dabei wird ein kleines, selbstentwickeltes Tool benutzt (DbAccessTester).
* Das Kapitel 12 schließlich enthält einige Literatur-Hinweise.

## Beispiele zum Thema Testgetriebene Entwicklung

Wie lernt man programmieren? Indem man programmiert.

Wie lernt man testgetriebene Softwareentwicklung? Indem man testgetrieben Software entwickelt.

Wir brauchen daher Beispiele. Solche Beispiele dürfen einerseits nicht trivial sein – andererseits aber müssen sie überschaubar sein (die Zeit ist knapp…)

Das prinzipielle Vorgehen bei der testgetriebenen Entwicklung ist bereits im Einleitung-Kapitel am Beispiel einer IntArray-Klasse beschrieben worden. Das Beispiel war im Gegensatz zu den Beispielen dieses Kapitels recht trivial – trotzdem sollte man sich an diesem Beispiel orientieren.

Im den Kapiteln zum Thema TDD werden zwei nicht triviale Beispiele beschrieben. Eines dieser Beispiele sollte im Seminar ausführlich behandelt werden – alle können nicht behandelt werden.

* Im ersten Beispiel geht's um die Entwicklung eines Scanners und eines Parsers für numerische Expressions.
* Im zweiten Beispiel geht's um die Entwicklung eines Systems zum Entwurf logischer Schaltungen.

Das erste Beispiel ist fokussiert auf das Konzept einzelner, kleiner Schritte. Im zweiten Beispiel steht eher der Spezifikations-Aspekt im Vordergrund.

Im folgenden werden die Aufgaben grob umrissen – konkrete Entwicklungsschritte werden aber nicht weiter vorgegeben.

Bei der Implementierung sollte natürlich immer auch sofort refaktoriert werden – sobald sich solche Refaktorierungen anbieten.

## Beispiel zum Thema Refactoring

Nicht immer startet der Entwickler auf der grünen Wiese – oft genug passiert es, dass er sich in existierenden, "gewachsenen" Quellcode einarbeiten muss, um seine Funktionalität zu erweitern oder einfach um Fehler zu beseitigen. Bei einer solchen Einarbeitung kann es sinnvoll sein, schrittweise zu refaktorieren – durch den Versuch einer Refaktorierung jeweils einzelner Teile werden diese Teile häufig überhaupt erst verständlich. Man könnte sagen: Verstehen heißt Refaktorieren. Refaktorieren ist risikolos, wenn bereits Tests vorliegen. Diese können dann entweder nach erfolgter Refaktorierung direkt genutzt werden (um zu zeigen, dass die Funktionalität durch die Refaktorierung nicht beeinträchtigt wurde) oder aber sukzessive erweitert werden.

Bei der Refaktorierung geht es zunächst natürlich einmal darum, Stellen in der Software zu entdecken, die "schlecht riechen". Hier eine kleine Liste von Kriterien für schlechten Code (die Reihenfolge der Punkte ist eher zufällig):

* Code-Duplikation (Copy & Paste)
* keine klaren Verantwortlichkeiten
* Eierlegende Wollmilchsau
* "langweiliger" Code (stattdessen vielleicht Iteration?)
* Exzessive dynamische Typabfragen mit downcasts (dynamic\_cast)
* Exzessiver Gebrauch von "switch"-Konstrukten (mit dynamic\_cast-Konstrukten)
* Exzessiver Gebrauch von static
* inkonsistente Namen (Finder beginnen mit find, get, read...)
* inkonsistente Groß/Kleinschreibung (class c, void M())
* fehlendes const
* Return-Werte statt Exceptions
* Definition lokaler Variablen ohne Initialisierung
* Definition lokaler Variablen, weit bevor sie benötigt werden
* Definition einer Instanzvariablen, wo auch eine lokale Variable ausreicht
* Änderung von Parametern
* Benutzung von vector o.ä., wenn ein einfacher Array angemessen ist
* Im Konstruktor: Aufruf von Methoden, die überschrieben werden können
* festverdrahtete Parameter
* Exzessiver Gebrauch von Klassenvererbung(=> Objektkomposition via Interfaces vs. klassischer Vererbung)
* Exzessiver Gebrauch von globalen Funktionen
* Instanzvariablen namens "typ" .....
* Exzessiver Gebrauch von void\*
* "Schalter"-Parameter
* tief geschachtelte Kontrollstrukturen
* zu große Methoden
* zu große Klassen
* zu lange Parameterlisten
* lange Message-Chains
* Benutzung von Gruppen von primitiven Daten statt expliziter Objekte
* Direkter Zugriff auf den internen Zustand von Objekten
* Unvollständige Library-Klassen
* Geschwätzige, überflüssige oder schlichtweg falsche Kommentare

Einige der obigen Kriterien sind von Martin Fowler übernommen worden (Fowler, Refactoring).

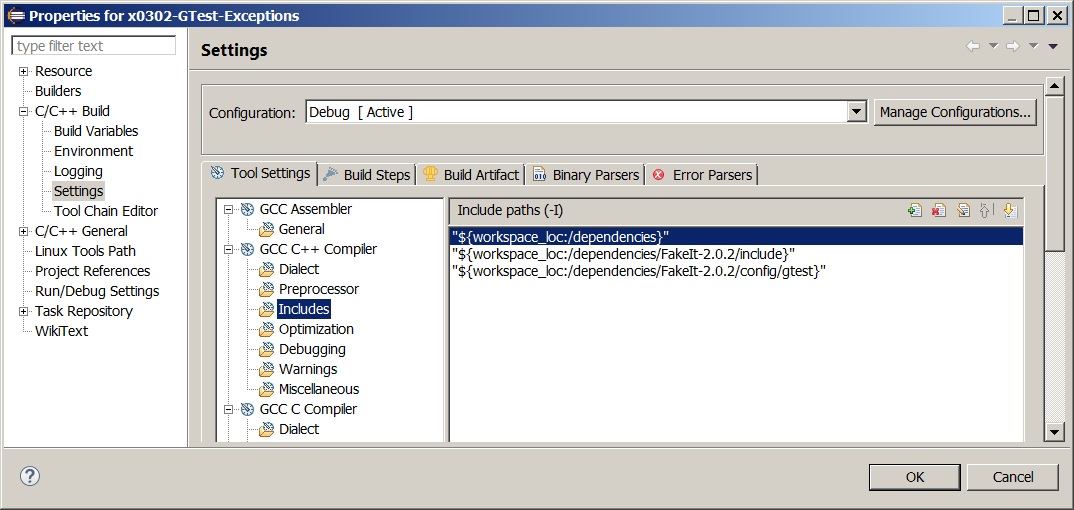
Im folgenden soll in Form einer größeren Übung ein einfaches, aber nicht triviales Programm refaktoriert werden. Das zu refaktorierende Programm ist vorgegeben – allerdings in einigermaßen grausamer Gestalt. In seiner anfänglichen Form ist das Programm auch kaum testbar. Deshalb soll es in mehreren Schritten refaktoriert werden – u.a. mit dem Ziel, dass seine Teile immer besser testbar werden. Für jede neu entwickelte Klasse soll also auch eine Testklasse erstellt werden. Ein weiteres Ziel ist natürlich ein verbessertes Design – womöglich auch die Erstellung wiederverwendbarer Komponenten.

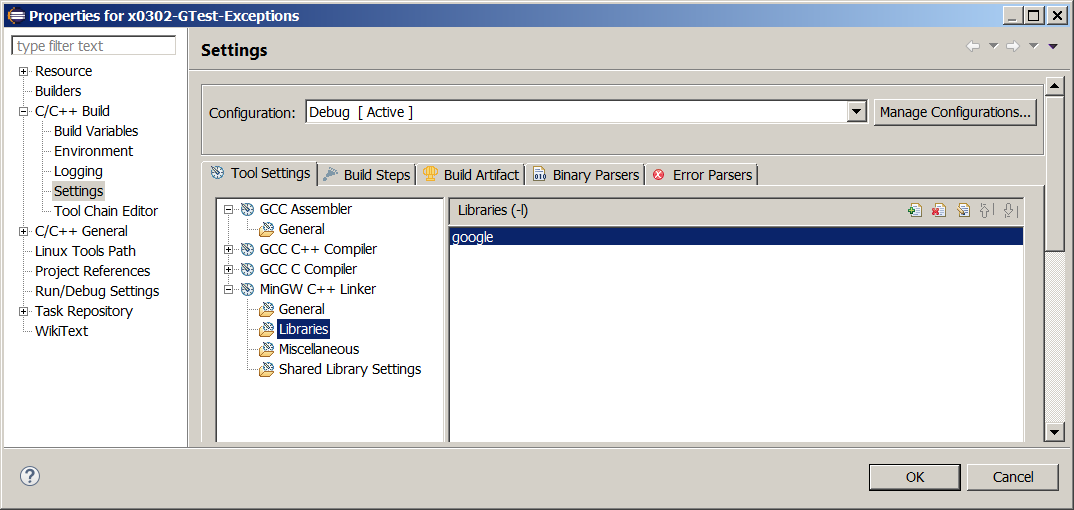
## Organisation der Beispiel-Projekte

Die Beispielprogramme befinden sich in der Datei CPP-TDD.zip. Diese Datei kann an beliebiger Stelle im Dateisystem extrahiert werden. Als Resultat entsteht ein Ordner mit dem Namen CPP-TDD. Dieser enthält ein Unterverzeichnis projects, welcher die Beispielprojekte enthält. Er enthält ein weiteres Unterverzeichnis namens dependencies, in welchem sich die verwendeten Tools befinden.

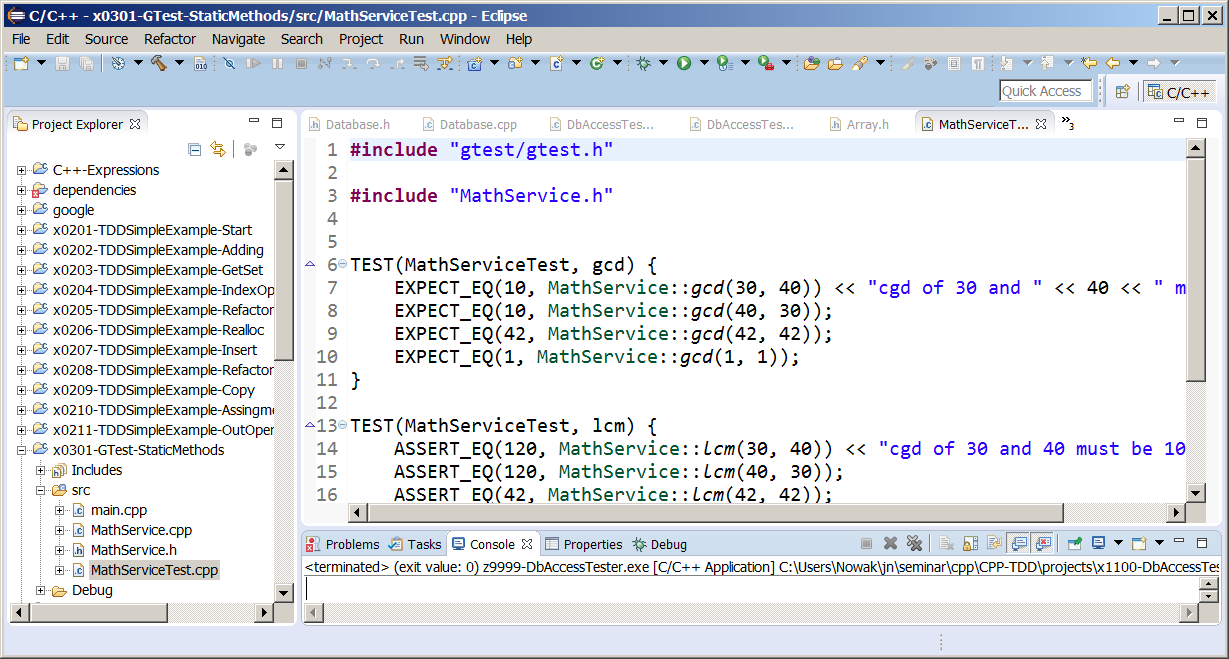
Grundlage ist die Datei googletest-master.zip, welche von Google heruntergeladen werden kann (diese befindet sich direkt in der Wurzel von CPP-TDD.zip). Diese Datei enthält zwei Unterverzeichnisse: googletest und googlemock. Beide Unterverzeichnisse enthalten jeweils die Unterverzeichnis inlucde und src. Diese include-Verzeichnisse enthalten die benötigten Header; die src-Verzeichnisse die benötigten cc-Dateien. Die include-Dateien wurden in den dependencies-Ordner kopiert; aus den cc-Dateien wurde die Bibliothek libgoogle.a erstellt (mittels des google-Projekts), welche ebenfalls nach dependencies kopiert wurde.

Alle weiteren Projekte beziehen sich auf diese Header und diese Bibliothek. Hier die Header- und die Bibliotheks-Settings eines beispielhaften Projekts:





Zu jedem Abschnitt eines jeden Kapitels dieses Skripts existiert im dem projects-Verzeichnis ein entsprechendes Projekt: z.B. x0301-GTest-StaticMethods, x0401-Testing-Stack etc. Die Namen (und die Nummerierung) dieser Projekte entsprechen der Kapitel-/ Abschnitts-Nummerierung in diesem Skript. Ein Ausschnitt aus dem Workspace:



# TDD – Ein einfaches Beispiel

Das Konzept der testgetriebenen Entwicklung sei im Folgenden an einem einfachen Beispiel erläutert (ein Beispiel demonstriert mehr als lange Romane!).

Es soll eine Klasse IntArray entwickelt werden. Ein IntArray soll beliebig viele int-Werte speichern können. Auf die Elemente soll indiziert zugegriffen werden können. Neue Elemente sollen ans Ende hinzugefügt oder an irgendeiner Stelle eingefügt werden können.

Ein IntArray kann somit als "Spezialausprägung" der template-Klasse vector angesehen werden.

Natürlich müssen bei der Entwicklung auch C++-Besonderheiten berücksichtigt werden (Copy-Construction, Assignment, Equality etc.).

## Start

Der erste Test-Code könnte wie folgt aussehen:

// IntArrayTest.cpp

TEST(**IntArrayTest**, **start**) {

IntArray array(3);

ASSERT\_EQ(0, array.size());

}

Einige technische Hinweise: TEST ist gtest-Makro, mittels derer eine Methode implementiert werden kann, welche dann vom gtest-Framework automatisch aufgerufen werden kann. ASSERT\_EQ ist ebenfalls ein gtest-Makro, dem zwei Parameter übergeben werden: das erwartete und das tatsächliche Ergebnis. Entspricht das tatsächliche Ergebnis dem erwarteten Ergebnis, kehrt ASSERT\_EQ lautlos zurück. Fall nicht, wird ASSERT\_EQ eine entsprechende Nachricht produzieren: wir wissen dann, dass der Test fehlgeschlagen ist.

Aufgrund des obigen Tests wissen wir nun, dass ein IntArray offenbar ein "Collection"-Objekt ist, welches beliebig viele int-Werte speichern können soll. Wir wissen weiterhin, dass dem Konstruktor der IntArray-Klasse eine Ganzzahl übergeben werden soll – die anfängliche Kapazität des IntArrays. Wir wissen schließlich, dass size die tatsächliche Anzahl der Elemente zurückliefern soll, die zum IntArray hinzugefügt wurden. Da der IntArray augenblicklich noch leer ist, muss size den Wert 0 liefern.

Der nächst Schritt besteht dann darin, den Test-Code kompilierbar zu machen. Es geht hier darum, die einfachste IntArray-Klasse zu schreiben, die sich denken lässt – deren Existenz dann aber den Test-Code kompilierbar macht. Also z.B.:

// ArrayTest.h

class **IntArray** {

public:

**IntArray**(unsigned int initialCapacity) {

}

unsigned int **size**() {

return 0;

}

};

Der Testcode (IntArrayTest) wird nun übersetzt. Und auch die Ausführung des Codes gelingt. Damit ist der erste Schritt getan.

## Adding

Mittels des Aufrufs einer add-Methode soll zu einem IntArray ein int-Wert hinzugefügt werden. Wir erweitern unseren Test um eine zweite Methode:

// IntArrayTest.cpp

TEST(**IntArrayTest**, **start**) { ... }

TEST(**IntArrayTest**, **adding**) {

IntArray array(3);

array.add(10);

array.add(20);

array.add(30);

ASSERT\_EQ(3, array.size());

}

Zunächst muss der Compiler zufriedengestellt werden. Wir erweitern die IntArray-Klasse durch die einfachste denkbare Implementierung der add-Methode:

// IntArray.h

class **IntArray** {

public:

**IntArray**(unsigned int initialCapacity) {

}

void **add**(int element) {

}

unsigned int **size**() {

return 3;

}

};

Die Ausführung der zweiten Test-Methode wird nun gelingen – aber die Ausführung der ersten wird nun scheitern. Wir müssen also mehr tun – aber nur minimal mehr:

// IntArray.h

class **IntArray** {

private:

unsigned int **size\_**;

public:

**IntArray**(unsigned int initialCapacity) : size\_(0) {

}

void **add**(int element) {

size\_++;

}

unsigned int **size**() {

return size\_;

}

};

Indem wir den Test-Code schreiben, spezifizieren(!) wird sukzessive die zu entwickelnde Klasse. Und diese Spezifikation ist ersten genauer als verbale "Romane". Und zweitens: sie ist nicht "abstrakt", sondern "ausführbar": die Korrektheit der Klasse IntArray kann immer sofort mittels dieses Test-Codes verifiziert werden. Der Test-Code dient also nicht nur zum Test bereits fertiger Software, sondern zugleich auch zu deren Spezifikation.

## Get / Set

Der nächste Schritt: Wir wollen lesend und schreibend auf die Elemente eines IntArrays zugreifen können. Wir spezifizieren dieses Feature in einer weiteren Testmethode:

// IntArrayTest.cpp

TEST(**IntArrayTest**, **start**) { ... }

TEST(**IntArrayTest**, **adding**) { ... }

TEST(**IntArrayTest**, **getSet**) {

IntArray array(10);

array.add(10);

array.add(20);

array.add(30);

ASSERT\_EQ(3, array.size());

ASSERT\_EQ(10, array.get(0));

ASSERT\_EQ(20, array.get(1));

ASSERT\_EQ(30, array.get(2));

ASSERT\_ANY\_THROW(array.get(3));

array.set(2, 300);

ASSERT\_EQ(300, array.get(2));

ASSERT\_ANY\_THROW(array.set(3, 300));

}

Mittels der get-Methode soll das index-te Element ermittelt werden können – der Index wird an get übergeben, und get liefert das Element zurück.

Mittels set soll ein bestehendes Element durch ein anderes Element ersetzt werden. An set wird zunächst der Index übergeben und dann der neue Wert.

Wird an get oder set ein "ungültiger" Index übergeben, soll get eine Exception werfen (das können wir mit dem gtest-Makro ASSERT\_ANY\_THROW zusichern).

// IntArray.h

class **IntArray** {

private:

// ...

public:

// ...

int **get**(unsigned int index) {

return 0;

}

void **set**(unsigned int index, int value) {

}

};

Der Compiler ist zufrieden – aber der Test wird natürlich scheitern (das Scheitern ist geplant!).

Damit ist der nächste Schritt vorgegeben: die "korrekte" Implementierung von get und set. Und hier sind wir an dem Punkt angelangt, wo wir etwas "mehr" tun müssen:

// IntArray.h

class **IntArray** {

private:

int\* **elements\_**;

unsigned int **capacity\_**;

unsigned int **size\_**;

public:

**IntArray**(unsigned int initialCapacity) :

elements\_(new int[initialCapacity]),

capacity\_(initialCapacity),

size\_(0) {

}

**~IntArray**() {

delete[] elements\_;

}

void **add**(int element) {

if(size\_ == capacity\_)

throw runtime\_error("this should never happen");

elements\_[size\_] = element;

size\_++;

}

unsigned int **size**() {

return size\_;

}

int **get**(unsigned int index) {

if (index >= size\_)

throw runtime\_error("index out of range");

return elements\_[index];

}

void **set**(unsigned int index, int value) {

if (index >= size\_)

throw runtime\_error("index out of range");

elements\_[index] = value;

}

};

Der Test gelingt.

## Index-Operator

Auf die Elemente eines IntArrays soll auch mit dem Index-Operator zugegriffen werden können – sowohl lesend als auch schreibend. Wir spezifizieren:

// IntArrayTest.cpp

TEST(**IntArrayTest**, **start**) { ... }

TEST(**IntArrayTest**, **adding**) { ... }

TEST(**IntArrayTest**, **getSet**) { ... }

TEST(**IntArrayTest**, **indexOperator**) {

IntArray array(10);

array.add(10);

array.add(20);

array.add(30);

ASSERT\_EQ(20, array[1]);

array[2] = 300;

ASSERT\_EQ(300, array[2]);

ASSERT\_ANY\_THROW(array[3]);

}

Wir implementieren:

// IntArray.h

class **IntArray** {

private:

// ...

public:

// ...

int& **operator[]**(unsigned int index) {

if (index >= size\_)

throw runtime\_error("index out of range");

return elements\_[index];

}

};

## Refactoring

Wir sehen uns den Code etwas genauer an – und erkennen, dass er Redundanzen enthält. Wir refaktorieren – indem wir eine Methode checkIndex definieren und an drei Stellen aufrufen:

// IntArray.h

class **IntArray** {

private:

// ...

public:

// ...

int **get**(unsigned int index) {

checkIndex(index);

return elements\_[index];

}

void **set**(unsigned int index, int value) {

checkIndex(index);

(\*this)[index] = value;

}

int& **operator[]**(unsigned int index) {

checkIndex(index);

return elements\_[index];

}

private:

void **checkIndex**(unsigned int index) {

if (index >= size\_)

throw runtime\_error("index out of range");

}

};

Alle bisherigen Tests werden auch weiterhin erfolgreich durchlaufen.

## Realloc

Ein IntArray hat eine bestimmte anfängliche Kapazität; die Kapazität soll aber je nach Bedarf dynamisch wachsen können. Wir spezifizieren folgendes Verhalten:

// IntArrayTest.cpp

// ...

TEST(**IntArrayTest**, **realloc**) {

IntArray array(1);

array.add(10);

array.add(20);

array.add(30);

ASSERT\_EQ(3, array.size());

ASSERT\_EQ(10, array[0]);

ASSERT\_EQ(20, array[1]);

ASSERT\_EQ(30, array[2]);

}

Die anfängliche Kapazität ist 1; es soll aber möglich sein, einem IntArray beliebig viele Elemente hinzuzufügen.

Der Test scheitert (wie geplant) – es wird die Exception "this should never happen" geworfen.

Wir erweitern die add-Methode der Klasse IntArray:

// IntArray.h

class **IntArray** {

// ...

void **add**(int element) {

if(size\_ == capacity\_) {

int\* newElements = new int [capacity\_ \* 2];

for (unsigned int i = 0; i < capacity\_; ++i)

newElements[i] = elements\_[i];

delete[] elements\_;

elements\_ = newElements;

capacity\_ \*= 2;

}

elements\_[size\_] = element;

size\_++;

}

// ...

};

Der Test verläuft nun erfolgreich.

## Insert

Die Funktionalität der IntArray-Klasse soll um eine insert-Methode erweitert werden, mittels derer ein Element an einer belieben Stelle der bereits existierenden Elemente eingefügt werden kann.

Wir spezifizieren das neue Feature:

// IntArrayTest.cpp

// ...

TEST(**IntArrayTest**, **insert**) {

IntArray array(1);

array.add(10);

array.add(20);

array.add(30);

array.insert(2, 25);

ASSERT\_EQ(4, array.size());

ASSERT\_EQ(10, array[0]);

ASSERT\_EQ(20, array[1]);

ASSERT\_EQ(25, array[2]);

ASSERT\_EQ(30, array[3]);

}

Wir stellen zunächst den Compiler zufrieden:

// IntArray.h

class **IntArray** {

// ...

void **insert**(unsigned int index, int element) {

}

// ...

};

Der Test misslingt natürlich. Wir haben uns somit die nächste Aufgabe gestellt. Hier die Lösung dieser Aufgabe:

// IntArray.h

class **IntArray** {

// ...

void **insert**(unsigned int index, int element) {

if(size\_ == capacity\_) {

int\* newElements = new int [capacity\_ \* 2];

for (unsigned int i = 0; i < capacity\_; ++i)

newElements[i] = elements\_[i];

delete[] elements\_;

elements\_ = newElements;

capacity\_ \*= 2;

}

for(unsigned int i = size\_; i > index; --i)

elements\_[i] = elements\_[i - 1];

elements\_[index] = element;

size\_++;

}

// ...

};

Der Test gelingt.

## Refactoring

Bei der Lösung der Aufgabe benutzten wir Copy&Paste – und haben uns somit eine Redundanz eingehandelt (der Reallokations-Mechanismus existiert nun sowohl in der add- als auch in der neuen insert-Methode). Wir refaktorieren – wir führen eine Hilfsmethode ensureCapacity ein:

// IntArray.h

class **IntArray** {

// ...

public:

// ...

void **add**(int element) {

ensureCapacity();

elements\_[size\_] = element;

size\_++;

}

void **insert**(unsigned int index, int element) {

ensureCapacity();

for(unsigned int i = size\_; i > index; --i)

elements\_[i] = elements\_[i - 1];

elements\_[index] = element;

size\_++;

}

private:

// ...

void **ensureCapacity**() {

if (size\_ == capacity\_) {

int\* newElements = new int[capacity\_ \* 2];

for (unsigned int i = 0; i < capacity\_; ++i)

newElements[i] = elements\_[i];

delete[] elements\_;

elements\_ = newElements;

capacity\_ \*= 2;

}

}

};

Der Test zeigt, dass die Refaktorierung erfolgreich war.

## Copy

Ein IntArray sollte vernünftig kopiert werden können. Wir spezifizieren:

TEST(**IntArrayTest**, **copy**) {

IntArray array1(1);

array1.add(10);

array1.add(20);

IntArray array2(array1); // oder: IntArray array2 = array1;

array1.add(30);

array2.add(40);

array2.add(50);

array2[0] = 1000;

ASSERT\_EQ(3, array1.size());

ASSERT\_EQ(10, array1[0]);

ASSERT\_EQ(20, array1[1]);

ASSERT\_EQ(30, array1[2]);

ASSERT\_EQ(4, array2.size());

ASSERT\_EQ(1000, array2[0]);

ASSERT\_EQ(20, array2[1]);

ASSERT\_EQ(40, array2[2]);

ASSERT\_EQ(50, array2[3]);

}

Der Compiler ist weiterhin zufrieden – der Test aber misslingt. Die Klasse IntArray benötigt einen vernünftigen Copy-Konstruktor:

// IntArray.h

class **IntArray** {

// ...

public:

// ...

**IntArray**(const IntArray& source) :

elements\_(new int[source.capacity\_]),

capacity\_(source.capacity\_),

size\_(source.size\_) {

for (unsigned int i = 0; i < size\_; i++)

elements\_[i] = source.elements\_[i];

}

// ...

};

Der Test verläuft nun erfolgreich.

## Assingment

Einem IntArray sollte auch ein anderer IntArray zugewiesen werden können. Wir spezifizieren:

// IntArrayTest.cpp

// ...

TEST(**IntArrayTest**, **assignment**) {

IntArray array1(1);

array1.add(10);

array1.add(20);

IntArray array2(10);

array2.add(77);

array2 = array1;

array1.add(30);

array2.add(40);

array2.add(50);

array2[0] = 1000;

ASSERT\_EQ(3, array1.size());

ASSERT\_EQ(10, array1[0]);

ASSERT\_EQ(20, array1[1]);

ASSERT\_EQ(30, array1[2]);

ASSERT\_EQ(4, array2.size());

ASSERT\_EQ(1000, array2[0]);

ASSERT\_EQ(20, array2[1]);

ASSERT\_EQ(40, array2[2]);

ASSERT\_EQ(50, array2[3]);

Der Test scheitert.

Wir erweitern die Klasse IntArray um einen Zuweisungs-Operator:

// IntArray.h

class **IntArray** {

// ...

public:

// ...

IntArray& **operator =** (const IntArray& source) {

delete[] elements\_;

elements\_ = new int[source.capacity\_];

capacity\_ = source.capacity\_;

size\_ = source.size\_;

copyElements(elements\_, source.elements\_, size\_);

return \*this;

}

// ...

};

Nun gelingt der Test.

## Ausgabe-Operator

Und schließlich sollte es möglich sein, einen IntArray via << auszugeben:

// IntArrayTest.cpp

// ...

TEST(**IntArrayTest**, **outOperator**) {

IntArray array(1);

array.add(10);

array.add(20);

array.add(30);

stringstream stream;

stream << array;

ASSERT\_EQ("IntArray [10, 20, 30]", stream.str());

}

Wir erweitern die Header-Datei um die Deklaration einer globalen Operator-Funktion:

// IntArray.h

class **IntArray** {

// ...

};

ostream& **operator <<** (ostream& out, const IntArray& array);

Und kommen dann natürlich um die Einführung einer cpp-Datei nicht mehr umhin:

// IntArray.cpp

// ...

ostream& **operator <<** (ostream& out, const IntArray& array) {

out << "IntArray [";

for (unsigned int i = 0; i < array.size(); i++) {

if (i > 0)

out << ", ";

out << array.get(i);

}

return out << "]";

}

Auch der Test gelingt.

Der letzte Schritt wäre eine erneute Refaktorierung: die Auslagerung der Methoden-Implementierungen aus der Header- in die cpp-Datei…

## Resultate und Konsequenzen

Die testgetriebene Entwicklung besteht aus einem Zyklus von Spezifikation (per "Test-Anwendung") und Implementierung. Dabei stellt sich der Entwickler abwechselnd auf zwei Positionen: auf die Position des Benutzers einer Klasse und auf die Position des Implementierens einer Klasse. Der Entwickler ist also kein "bornierter" Implementierter – er hat zunächst die Benutzung im Auge. Und eine Klasse ist nicht Selbstzweck – sie ist dazu da, benutzt zu werden.

Ein Entwickler, der mit der Implementierung beginnt und die Nutzung der Klasse aus den Augen verliert, wird möglicherweise eine Klasse entwickeln, die nur umständlich zu nutzen ist. Vielleicht ist die Klasse auch unvollständig. Solche Mängel können natürlich am besten vermieden werden, wenn der Entwickler zunächst an die Benutzung der Klasse denkt – und nicht nur an diese Benutzung denkt, sondern sie auch - für den Compiler verständlich und übersetzbar – aufschreibt.

Indem der Entwickler die Benutzung der Klasse demonstriert, spezifiziert er die Klasse. Diese Spezifikationen (also die Testmethoden) erzählen jeweils eine kleine "Story", aus welcher genau hervorgeht, wie die Klasse zu benutzen ist. Jemand, der die Klasse nutzen will, muss dann eigentlich nur die Testmethoden lesen. Anhand dieser Methoden kann er die Art und Weise der möglichen Nutzung exakt verstehen.

Jeder Schritt der Entwicklung einer Klasse ist abgesichert. Es kann also niemals Zweifel daran aufkommen, dass der aktuelle Implementierungsstand korrekt ist. (Was aber nicht notwendigerweise bedeutet, dass die Implementierung vielleicht später zu refaktorieren ist. Und auch nicht, dass die aktuelle Spezifikation auf ewig Bestand hat...)

Beim Wechsel von "rot" zu "grün" kann der Entwickler sich "zurücklehnen" – die zuletzt vorgenommenen Erweiterungen / Änderungen haben sich definitiv als zielführend herausgestellt. Der Entwickler wird daher das "grün" jeweils auch als Belohnung und Bestätigung seiner Arbeit auffassen können. Und gewöhnlich lieben es Menschen, bestätigt zu werden...

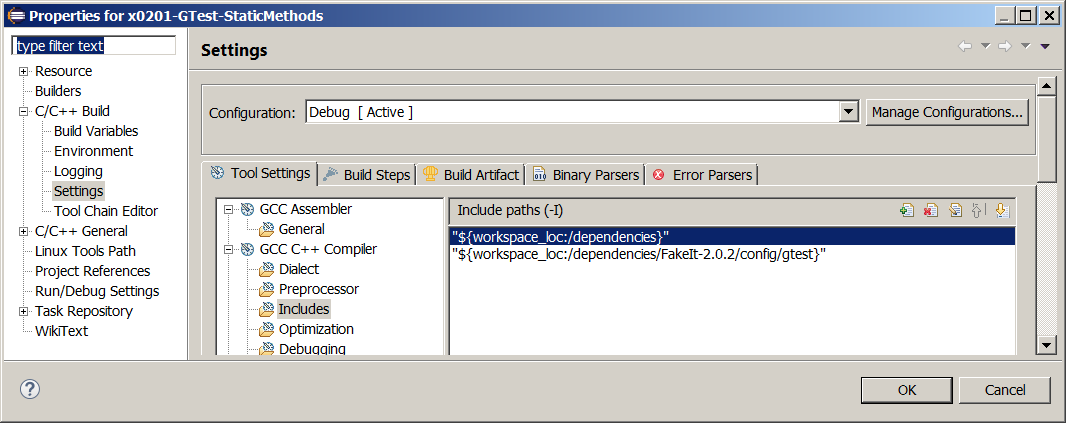
Der Vorteil von Testmethoden erweist sich insbesondere dann, wenn die Klassen später geändert oder erweitert werden müssen – der einfach refrakturiert werden sollen. Nach jedem Schritt einer solchen Erweiterung / Änderung / Refaktorierung können die Tests erneut ausgeführt werden. Ist das Ergebnis dann "grün", kann der Entwickler sicher sein, nichts "kaputt repariert" zu haben. Und nach der Auslieferung der neuen Software kann der Entwickler ruhig schlafen. (Psychologie ist nicht zu unterschätzen...)

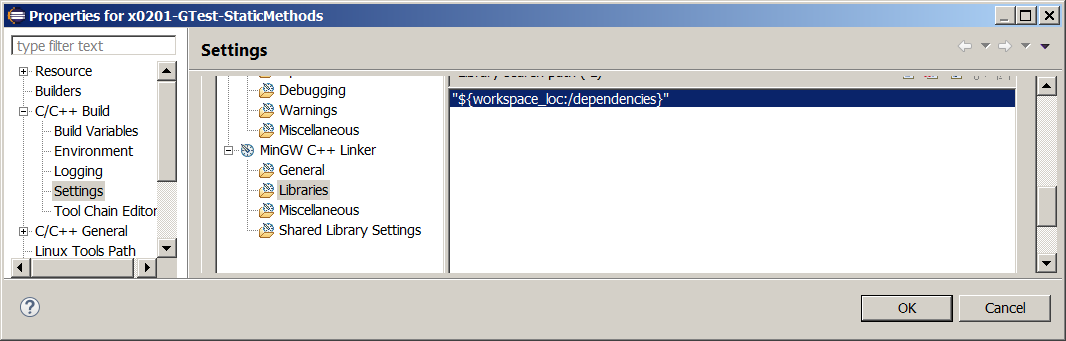
Auch ein Test-Client ist genau so ernst zu nehmen wie ein "richtiger" Client einer Klasse. Auch er braucht Pflege (immer dann, wenn die Spezifikation geändert oder erweitert wird).

# Testing – Das gtest Framework

Im folgenden wird das Test-Framework gtest näher vorgestellt.

Projekt-Einstellungen:





## Start

Hier die zu testende Klasse – eine Klasse mit ausschließlich statischen Methoden:

// MathService.h

class **MathService** {

public:

static int **gcd**(int x, int y);

static int **lcm**(int x, int y);

};

// MathService.cpp

#include "MathService.h"

int **MathService::gcd**(int x, int y) {

while (x != y) {

if (x > y)

x -= y;

else

y -= x;

}

return x;

}

int **MathService::lcm**(int x, int y) {

int a = x;

int b = y;

while (a != b) {

if (a < b)

a += x;

else

b += y;

}

return a;

}

Und hier die Testklasse:

// MathServiceTest.cpp

#include "gtest/gtest.h"

#include "MathService.h"

TEST(**MathServiceTest**, **gcd**) {

EXPECT\_EQ(10, MathService::gcd(30, 40))

<< "cgd of 30 and 40 must be 10";

EXPECT\_EQ(10, MathService::gcd(40, 30));

EXPECT\_EQ(42, MathService::gcd(42, 42));

EXPECT\_EQ(1, MathService::gcd(1, 1));

}

TEST(**MathServiceTest**, **lcm**) {

ASSERT\_EQ(120, MathService::lcm(30, 40))

<< "cgd of 30 and 40 must be 10";

ASSERT\_EQ(120, MathService::lcm(40, 30));

ASSERT\_EQ(42, MathService::lcm(42, 42));

ASSERT\_EQ(1, MathService::lcm(1, 1));

}

TEST(**MathServiceTest**, **DISABLED\_lcm**) {

// ...

}

In gtest werden alle Testmethoden mittels des Makros TEST definiert. Dem Makro werden zwei Parameter übergeben: Der Name des "Test-Cases" und der Name des konkreten Tests.

Assertions können in gtest u.a. mittels der Makros EXPECT\_EQ und ASSERT\_EQ definiert werden. Beide Makros verlangen zwei Argumente: den erwarteten Wert einer Berechnung und den tatsächlichen Wert dieser Berechnung (und zwar in genau dieser Reihenfolge: expected – actual).

Stimmt der erwartete mit dem tatsächlichen Wert überein, kehren die Makros lautlos zurück. Ansonsten wird die Abweichung der Erwartung vom tatsächlichen Resultat protokolliert (siehe weiter unten).

Während eine solche Abweichung beim ASSERT\_EQ-Makro zum sofortigen Abbruch der Test-Methode führt (die weiteren Assertions werden nicht mehr ausgeführt), werden bei EXPECT\_EQ auch noch die folgenden Assertions ausgeführt. ASSERT\_EQ ist als "hart", EXPCECT\_EQ ist "weich". I.d.R. sollte EXPECT\_EQ verwendet werden – um möglicherweise gleich mehrere Fehler mittels der Ausführung einer einzigen Testmethode ausfindig machen zu können. ASSERT\_EQ sollte nur dann verwendet werden, wenn es um die Entdeckung "fataler" Fehler geht – dergestalt, dass weitere Assertions sinnlos sind.

Beginnt der Name eines Tests mit DISABLED\_, dann wird die entsprechende Methode nicht ausgeführt – im Testprotokoll wird aber eben dieser Umstand auch vermerkt. (Man sieht es dem Protokoll also an, dass Methoden nicht ausgeführt werden.)

Wie wird nun ein gtest-Test ausgeführt?

Wir schreiben eine kleine main-Funktion:

// main.cpp

#include <iostream>

#include "gtest/gtest.h"

using namespace std;

int **main**(int argc, char\*\* argv) {

cout << "GTest-StaticMethods" << endl;

testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

Die main-Funktion kann mittels gtest-eigener Kommandozeilen-Parameter gestartet werden (siehe dazu später). Diese Parameter werden weitergereicht an die gtest-Funktion InitGoolgeTest. Mittels des Makros RUN\_ALL\_TESTS werden dann alle mit dem TEST-Makro definierten Testmethoden ausgeführt (egal, in welchen Klassen sie sich befinden).

Sofern alle Tests erfolgreich durchlaufen werden, produziert das Testprogramm die folgende Ausgabe:

[==========] Running 2 tests from 1 test case.

[----------] Global test environment set-up.

[----------] 2 tests from MathServiceTest

[ RUN ] MathServiceTest.gcd

[ OK ] MathServiceTest.gcd (0 ms)

[ RUN ] MathServiceTest.lcm

[ OK ] MathServiceTest.lcm (0 ms)

[----------] 2 tests from MathServiceTest (0 ms total)

[----------] Global test environment tear-down

[==========] 2 tests from 1 test case ran. (1 ms total)

[ PASSED ] 2 tests.

YOU HAVE 1 DISABLED TEST

Angenommen, die zu testende gcd-Methode ist falsch implementiert:

int **MathService::gcd**(int x, int y) {

while (x == y) { // hier liegt der Fehler!!!

if (x > y)

x -= y;

else

y -= x;

}

return x;

}

Dann produziert das Testprogramm die folgende Ausgabe:

GTest-StaticMethods

[==========] Running 2 tests from 1 test case.

[----------] Global test environment set-up.

[----------] 2 tests from MathServiceTest

[ RUN ] MathServiceTest.gcd

..\src\MathServiceTest.cpp:8: Failure

Expected: 10

To be equal to: MathService::gcd(30, 40)

Which is: 30

cgd of 30 and 40 must be 10

..\src\MathServiceTest.cpp:9: Failure

Expected: 10

To be equal to: MathService::gcd(40, 30)

Which is: 40

[ FAILED ] MathServiceTest.gcd (2 ms)

[ RUN ] MathServiceTest.lcm

[ OK ] MathServiceTest.lcm (0 ms)

[----------] 2 tests from MathServiceTest (2 ms total)

[----------] Global test environment tear-down

[==========] 2 tests from 1 test case ran. (2 ms total)

[ PASSED ] 1 test.

[ FAILED ] 1 test, listed below:

[ FAILED ] MathServiceTest.gcd

1 FAILED TEST

YOU HAVE 1 DISABLED TEST

## Exceptions

Im Folgenden wird gezeigt, wie getestet werden, dass Methoden unter bestimmten Bedingungen eine Exception werfen müssen. Die Erwartung besteht dann nicht darin, dass solche Methoden ein bestimmtes Resultat liefern, sondern eben darin, dass sie eine Exception werfen.

Die Klasse MathService wird um den Test von Preconditions erweitert:

// MathService.cpp

#include "MathService.h"

#include <stdexcept>

using namespace std;

int **MathService::gcd**(int x, int y) {

if (x <= 0 || y <= 0)

throw runtime\_error("x and y must be positve");

while (x != y) {

if (x > y)

x -= y;

else

y -= x;

}

return x;

}

int **MathService::lcm**(int x, int y) {

if (x <= 0 || y <= 0)

throw runtime\_error("x and y must be positve");

int a = x;

int b = y;

while (a != b) {

if (a < b)

a += x;

else

b += y;

}

return a;

}

Die Testklasse könnte nun wie folgt ausschauen:

// MathServiceTest.cpp

#include "gtest/gtest.h"

#include <stdexcept>

#include "MathService.h"

using namespace std;

TEST(**MathServiceTest**, **gcdException**) {

try {

MathService::gcd(1, 0);

ADD\_FAILURE();

}

catch(const runtime\_error& e) {

SUCCEED(); // comment only

}

EXPECT\_THROW(MathService::gcd(-1, 1), runtime\_error);

EXPECT\_ANY\_THROW(MathService::gcd(-1, 1));

EXPECT\_THROW({ MathService::gcd(-1, 1); }, runtime\_error);

// block possible

}

TEST(**MathServiceTest**, **lcmExceptions**) {

try {

MathService::gcd(1, 0);

FAIL();

}

catch(const runtime\_error& e) {

SUCCEED(); // comment only

}

ASSERT\_THROW(MathService::gcd(-1, 1), runtime\_error);

ASSERT\_ANY\_THROW(MathService::gcd(-1, 1));

}

Die erste Testmethode benutzt beim ersten Test einen try-catch, um die beim gcd-Aufruf erwartete Exception aufzufangen. Sofern die gcd-Methode wieder Erwarten normal zurückkehren würde, sorgt das Makro ADD\_FAILURE für eine entsprechende Fehler-Ausgabe. Sofern aber der gcd-Aufruf die erwartete Exception wirft, wird SUCCEED ausgeführt (SUCCEED ist allerdings nur ein Kommentar, der auch weggelassen werden könnte).

Mittels des Makros EXPECT\_THROW kann der Test eleganter und knapper formuliert werden. Dem Makro wird der Aufruf der zu testenden Methode und der Typ der erwarteten Ausnahme übergeben.

Falls der Typ der geworfenen Exception nicht interessant ist, kann auch das Makro EXPECT\_ANY\_THROW verwendet werden.

Während die erste Testmethode nur "weiche" Makros nutzt, werden in der zweiten Testmethode die entsprechenden "harten" Makros verwendet (statt EXPECT\_\* nun also ASSERT\_\*). Das FAIL-Makro führt zum sofortigen Abbruch der Testmethode.

## Objects

Bislang ging es darum, statische Methoden zu testen. Was müssen wir beachten, wenn wir nicht-statische Methoden testen wollen?

Die Klasse MathService wird umgebaut und erweitert:

// MathService.h

class **MathService** {

unsigned int **callCount\_**;

public:

**MathService()** : callCount\_(0) {

}

int **gcd**(int x, int y);

int **lcm**(int x, int y);

unsigned int **callCount**() {

return callCount\_;

}

};

Die gcd- und lcm-Methoden sind nun als Instanzmethoden deklariert. Das Instanz-Attribut callCount\_ soll mitzählen, wie häufig gcd resp. lcm aufgerufen werden. Die Instanz-Methode callCount liefert den Wert von callCount\_ zurück.

Hier die Implementierung der gcd- und der lcm-Methode:

// MathService.cpp

#include "MathService.h"

#include <stdexcept>

using namespace std;

int **MathService::gcd**(int x, int y) {

if (x <= 0 || y <= 0)

throw runtime\_error("x and y must be positve");

while (x != y) {

if (x > y)

x -= y;

else

y -= x;

}

callCount\_++;

return x;

}

int **MathService::lcm**(int x, int y) {

if (x <= 0 || y <= 0)

throw runtime\_error("x and y must be positve");

int a = x;

int b = y;

while (a != b) {

if (a < b)

a += x;

else

b += y;

}

callCount\_++;

return a;

}

Im Testprogramm werden wir am Anfang jeder Testmethode ein neues, "jungfräuliches" MathService-Objekt erzeugen müssen:

// MathServiceTest.cpp

// ...

TEST(**MathServiceTest**, **gcd**) {

MathService service;

EXPECT\_EQ(0, service.callCount());

EXPECT\_EQ(10, service.gcd(30, 40));

EXPECT\_EQ(42, service.gcd(42, 42));

EXPECT\_THROW(service.gcd(-1, 1), runtime\_error);

EXPECT\_EQ(2, service.callCount());

}

TEST(**MathServiceTest**, **lcm**) {

MathService service;

EXPECT\_EQ(0, service.callCount());

EXPECT\_EQ(120, service.lcm(30, 40));

EXPECT\_EQ(42, service.lcm(42, 42));

EXPECT\_THROW(service.lcm(-1, 1), runtime\_error);

EXPECT\_EQ(2, service.callCount());

}

TEST(**MathServiceTest**, gcdAndLcm) {

MathService service;

EXPECT\_EQ(0, service.callCount());

EXPECT\_EQ(10, service.gcd(30, 40));

EXPECT\_EQ(42, service.lcm(42, 42));

EXPECT\_EQ(2, service.callCount());

}

Im Unterschied zur alten MathService-Klasse, welche nur statische Methoden enthielt, haben wir es nun mit zustandsbehafteten Objekten zu tun. Und da die Testmethoden völlig unabhängig voneinander sein müssen, muss natürlich jeder Test ein eigenes Objekt besitzen. Ein Test darf nicht auf dem Resultat eines anderen Tests beruhen. (By the way: wenn die Testmethoden auf mehrere cpp-Dateien verteilt wären, wüsste man natürlich auch nicht, in welcher Reihenfolge sie ausgeführt würden).

## Complex Objects

Häufig müssen zunächst mehrere Objekte erzeugt und zusammengesteckt werden, um eines dieser Objekte dann überhaupt testen zu können. Hier ein Beispiel:

Mittels eines MathService kann der GGT und das KGV zweier Zahlen berechnet werden: die Klasse bietet die Methoden gcd und lcm. Ein MathService kann auch danach gefragt werden, wie häufig er bereits GGT- und KGV-Berechnungen durchgeführt hat – mittels der Methode callCount. Der Service ist im Interface IMathService spezifiziert und in MathService implementiert.

Ein MathService-Objekt benutzt nun zwei Hilfsobjekte: einen GGT-Service und einen KGV- Service. Ein Hilfsservice besitzt jeweils eine Berechnungs-Methode (gcd resp. lcm) und eine callCount-Methode. Jeder Hilfsservice speichert die Anzahl der bereits erfolgten gcd- resp. lcm-Aufrufe. Auch die Hilfsservices sind jeweils zunächst in einem Interface spezifiziert. Und der Hauptservice soll sich auf die Hilfsservices nur über diese Interfaces beziehen - damit u.a. die Implementierung der Hilfsservices jederzeit durch eine andere Implementierung ersetzt werden kann.

Also müssen dem Hauptservice Referenzen auf die beiden benötigten Hilfsservices per Dependency Injection übergeben werden. Dies geschieht mittels des Konstruktors.

: MathService

*IMathService*

gcd (x, y)

callCount ()

lcm (x, y)

: GcdMathService

*IGcdMathService*

gcd (x, y)

callCount ()

2

: LcmMathService

*ILcmMathService*

lcm (x, y)

callCount ()

3

Hier die im folgenden Test benutzten Interfaces – das Interface des Hauptservice und das der beiden Hilfsservices:

// IMathService.h

class **IMathService** {

public:

virtual **~IMathService**() { }

virtual int **gcd**(int x, int y) = 0;

virtual int **lcm**(int x, int y) = 0;

virtual unsigned int **callCount**() = 0;

};

// IGcdMathService.h

class **IGcdMathService** {

public:

virtual **~IGcdMathService**() { }

virtual int **gcd**(int x, int y) = 0;

virtual unsigned int **callCount**() = 0;

};

// ILcmMathService.h

class **ILcmMathService** {

public:

virtual **~ILcmMathService**() { }

virtual int **lcm**(int x, int y) = 0;

virtual unsigned int **callCount**() = 0;

};

Hier die Implementierung dieser Interfaces:

// MathService.h

#include "IMathService.h"

#include "IGcdMathService.h"

#include "ILcmMathService.h"

class **MathService**: public IMathService {

IGcdMathService& **gcdService\_**;

ILcmMathService& **lcmService\_**;

public:

**MathService**(IGcdMathService& gcdService, ILcmMathService& lcmService) :

gcdService\_(gcdService),

lcmService\_(lcmService) {

}

int **gcd**(int x, int y) {

return gcdService\_.gcd(x, y);

}

int **lcm**(int x, int y) {

return lcmService\_.lcm(x, y);

}

unsigned int **callCount**() {

return gcdService\_.callCount() + lcmService\_.callCount();

}

};

// GcdMathService.h

#include "IGcdMathService.h"

// ...

class **GcdMathService** : public IGcdMathService {

private:

unsigned int **callCount\_**;

public:

**GcdMathService**() : callCount\_(0) {

}

int **gcd**(int x, int y) {

if (x <= 0 || y <= 0)

throw runtime\_error("x and y must be positve");

while (x != y) { if (x > y) x -= y; else y -= x; }

callCount\_++;

return x;

}

unsigned int **callCount**() {

return callCount\_;

}

};

// LcmMathService.h

#include "ILcmMathService.h"

// ...

class **LcmMathService** : public ILcmMathService {

private:

unsigned int **callCount\_**;

public:

**LcmMathService**() : callCount\_(0) {

}

int **lcm**(int x, int y) {

if (x <= 0 || y <= 0)

throw runtime\_error("x and y must be positve");

int a = x;

int b = y;

while (a != b) { if (a < b) a += x; else b += y; }

callCount\_++;

return a;

}

unsigned int **callCount**() {

return callCount\_;

}

};

Soll nun die Hauptservice-Klasse getestet werden, so müssen zunächst zwei Hilfsservice-Objekte erzeugt werden, welche dann dem Konstruktor von MathService übergeben werden müssen:

// MathServiceTest.cpp

// ...

TEST(**MathServiceTest**, **gcd**) {

GcdMathService gcdService;

LcmMathService lcmService;

MathService service(gcdService, lcmService);

EXPECT\_EQ(0, service.callCount());

EXPECT\_EQ(10, service.gcd(30, 40));

EXPECT\_EQ(42, service.gcd(42, 42));

EXPECT\_THROW(service.gcd(-1, 1), runtime\_error);

EXPECT\_EQ(2, service.callCount());

}

TEST(**MathServiceTest**, **lcm**) {

GcdMathService gcdService;

LcmMathService lcmService;

MathService service(gcdService, lcmService);

EXPECT\_EQ(0, service.callCount());

EXPECT\_EQ(120, service.lcm(30, 40));

EXPECT\_EQ(42, service.lcm(42, 42));

EXPECT\_THROW(service.lcm(-1, 1), runtime\_error);

EXPECT\_EQ(2, service.callCount());

}

TEST(**MathServiceTest**, **gcdAndLcm**) {

GcdMathService gcdService;

LcmMathService lcmService;

MathService service(gcdService, lcmService);

EXPECT\_EQ(0, service.callCount());

EXPECT\_EQ(10, service.gcd(30, 40));

EXPECT\_EQ(42, service.lcm(42, 42));

EXPECT\_EQ(2, service.callCount());

}

Jede der drei Testmethoden beginnt jeweils mit denselben Zeilen: der Erzeugung und Verbindung dreier Objekte. Andere zu testende Objekte mögen noch komplexer sein…

## Fixtures

Um vermeiden zu können, das Erzeugen und Zusammenstecken der Objekte in jeder de Testmethoden immer wieder in identischer Weise implementieren zu müssen, kann der Aufbau des Objektgraphen in einer einzigen Methode implementiert werden, welche zu Beginn jeder Testmethode automatisch aufgerufen wird.

Zu diesem Zweck muss eine explizite Testklasse geschrieben werden, welche von der gtest-Klasse testing::Test abgeleitet ist. In dieser Klasse können dann zwei Methoden der Basisklasse überschrieben werden: SetUp und TearDown. Das Testframe sorgt dann dafür, dass vor jedem Aufruf einer Testmethode die SetUp-Methode aufgerufen wird – und nach Ausführung der Testmethode die inverse Methode TearDown.

Dann müssen wir zur Definition einer Testmethode allerdings das Makro TEST\_F verwenden – wobei der Wert des ersten an TEST\_F übergebenen Parameters der Namen der Testklasse sein muss.

In SetUp können dann die beteiligen Objekte (dynamisch) erzeugt und miteinander verbunden werden. Das Ergebnis dieser Operationen wird an Instanzvariablen zugewiesen (auf welche dann die Testmethoden zugreifen können). In TearDown können die dynamisch erzeugten Ressourcen dann wieder freigegeben werden.

// MathServiceTest.cpp

// ...

class **MathServiceTest** : public testing::Test {

protected:

IGcdMathService\* **gcdService**;

ILcmMathService\* **lcmService**;

IMathService\* **service**;

public:

virtual void **SetUp**() override {

gcdService = new GcdMathService();

lcmService = new LcmMathService();

service = new MathService(\*gcdService, \*lcmService);

}

virtual void **TearDown**() override {

delete gcdService;

delete lcmService;

delete service;

}

};

TEST\_F(**MathServiceTest**, **gcd**) {

EXPECT\_EQ(0, service->callCount());

EXPECT\_EQ(10, service->gcd(30, 40));

EXPECT\_EQ(42, service->gcd(42, 42));

EXPECT\_THROW(service->gcd(-1, 1), runtime\_error);

EXPECT\_EQ(2, service->callCount());

}

TEST\_F(**MathServiceTest**, **lcm**) {

EXPECT\_EQ(0, service->callCount());

EXPECT\_EQ(120, service->lcm(30, 40));

EXPECT\_EQ(42, service->lcm(42, 42));

EXPECT\_THROW(service->lcm(-1, 1), runtime\_error);

EXPECT\_EQ(2, service->callCount());

}

TEST\_F(**MathServiceTest**, **gcdAndLcm**) {

EXPECT\_EQ(0, service->callCount());

EXPECT\_EQ(10, service->gcd(30, 40));

EXPECT\_EQ(42, service->lcm(42, 42));

EXPECT\_EQ(2, service->callCount());

}

## Fixtures – Details

Neben dem Überschrieben der SetUp- resp. der TearDown-Methode können in einer Testklasse auch zwei weitere Methoden definiert werden: SetUpTestCase und TearDownTestCase. Diese Methoden müssen allerdings als static definiert sein. Die SetUpTestCase-Methode wird ein einziges Mal aufgerufen – beim Start der Testanwendung. Auch TearDownTestCase wird nur ein einziges Mal aufgerufen – kurz bevor die Testanwendung terminiert. Da diese beiden Methoden als static definiert sein müssen, können sie also auch nur static-Ressourcen anfordern resp. freigeben.

In der folgenden Testanwendung soll gezeigt werden, welche Methoden wann aufgerufen werden. Zu diesem Zweck ist in der Testklasse auch ein Konstruktor und der Destruktor implementiert. Alle Methoden diagnostizieren nur ihren Aufruf:

// DemoTest.cpp

// ...

class **DemoTest** : public testing::Test {

public:

static void **SetUpTestCase**() {

cerr << "SetUpTestCase" << endl;

}

static void **TearDownTestCase**() {

cerr << "TearDownTestCase" << endl;

}

**DemoTest**() {

cerr << "\tCTOR" << endl;

}

virtual **~DemoTest**() {

cerr << "\tDTOR" << endl;

}

virtual void **SetUp**() override {

cerr << "\t\tSetUp" << endl;

}

virtual void **TearDown**() override {

cerr << "\t\tTearDown" << endl;

}

};

TEST\_F(**DemoTest**, **foo**) {

cerr << "\t\t\tfoo" << endl;

EXPECT\_EQ(10, 10);

}

TEST\_F(**DemoTest**, **bar**) {

cerr << "\t\t\tbar" << endl;

EXPECT\_EQ(10, 10);

}

Die Diagnose-Ausgaben erfolgen auf stderr (damit man sie besser von den gtest-Ausgaben unterscheiden kann). Man beachte auch die Einrückungen der Ausgaben.

Hier das Resultat:

SetUpTestCase

CTOR

SetUp

foo

TearDown

DTOR

CTOR

SetUp

bar

TearDown

DTOR

TearDownTestCase

Wie man sieht, wird jede der Testmethoden auf einem jeweils neuen(!) Objekt der Testklasse ausgeführt. Und das ist gut so - denn dann können Testmethoden, die voneinander abhängig sind, erst gar nicht formuliert werden. Der Unsinn einer solchen Abhängigkeit würde sofort offenkundig.

## Assertions

Im folgenden werden einige weitere ASSERT\_\*- resp. EXPECT\_\*-Makros vorgestellt. Zudem wird gezeigt, was beim Vergleich von Objekten zu beachten ist.

Die folgende einfache Point-Klasse dient zur Demonstration. Man beachte, dass in dieser Klasse einige Vergleichsoperatoren überladen sind. Man beachte schließlich, dass der globale Ausgabeoperator überladen wurde.

// Point.h

// ...

class **Point** {

const int **x\_**;

const int **y\_**;

public:

**Point**(int x, int y);

virtual **~Point**();

int **x**() const;

int **y**() const;

double **size**() const;

bool **operator ==** (const Point& other) const;

bool **operator !=** (const Point& other) const;

bool **operator >** (const Point& other) const;

bool **operator >=** (const Point& other) const;

};

ostream& **operator <<** (ostream& out, const Point& point);

// Point.cpp

// ...

**Point::Point**(int x, int y) : x\_(x), y\_(y) {

}

**Point::~Point**() {

}

int **Point::x**() const {

return this->x\_;

}

int **Point::y**() const {

return this->y\_;

}

double **Point::size**() const {

return ::sqrt(this->x\_ \* this->x\_ + this->y\_ \* this->y\_);

}

bool **Point::operator ==** (const Point& other) const {

return this->x\_ == other.x\_ && this->y\_ == other.y\_;

}

bool **Point::operator !=** (const Point& other) const {

return !(\*this == other);

}

bool **Point::operator >** (const Point& other) const {

return this->size() > other.size();

}

bool **Point::operator >=** (const Point& other) const {

return this->size() >= other.size();

}

ostream& **operator <<** (ostream& out, const Point& point) {

return out << "Point [" << point.x() << ", " << point.y() << "]" ;

}

Hier eine Testanwendung (statt EXPECT\_\* könnte hier auch stets ASSERT\_\* verwendet werden):

// DemoTest.cpp

// ...

TEST(**PointTest**, **equal**) {

Point p1(10, 20);

Point p2(10, 20);

EXPECT\_EQ(p1, p2); // requires operator ==

}

TEST(**PointTest**, **unequal**) {

Point p1(10, 20);

Point p2(11, 22);

EXPECT\_NE(p1, p2); // requires operator !=

}

TEST(**PointTest**, **greaterThan**) {

Point p1(10, 20);

Point p2(11, 22);

EXPECT\_GT(p2, p1); // requires operator >

// EXPECT\_GT(p1, p2); // bad test!! (but see the output!)

}

TEST(**PointTest**, **greaterOrEqual**) {

Point p1(10, 20);

Point p2(11, 22);

Point p3(10, 20);

EXPECT\_GE(p2, p1); // requires operator >=

EXPECT\_GE(p2, p1); // requires operator >=

}

TEST(**FloatDoubleTest**, **equal**) {

float f = 3.14;

double d = 3.14;

EXPECT\_FLOAT\_EQ(3.14, f);

EXPECT\_DOUBLE\_EQ(3.14, d);

EXPECT\_NEAR(3.14, f, 0.001);

EXPECT\_NEAR(3.14, d, 0.001);

}

TEST(**CStringTest**, **equal**) {

const char\* cs1 = "Hello";

EXPECT\_EQ("Hello", cs1);

char\* cs2 = new char[5 + 1];

strcpy(cs2, "Hello");

EXPECT\_NE("Hello", cs2); // NE !!!

EXPECT\_STREQ("Hello", cs2);

EXPECT\_STRCASEEQ("HELLO", cs2);

}

TEST(**StringTest**, **equal**) {

string cs1 = "Hello";

string cs2 = "Hello";

EXPECT\_EQ("Hello", cs1);

EXPECT\_EQ(cs2, cs1);

// EXPECT\_STREQ(cs2, cs1); // will not compile

}

TEST(**BooleanTest**, **trueFalse**) {

EXPECT\_TRUE(1 == 1) << ".....";

EXPECT\_FALSE(1 == 0);

}

Für den Vergleich von float resp. double-Werten existieren die speziellen Makros EXPECT\_FLOAT, EXPECT\_DOUBLE und EXPECT\_NEAR. Bei der Verwendung des letzten Makros muss als dritter Parameter die verlangte Genauigkeit übergeben werden. (float- resp. double-Berechnungen sind naturgemäß ungenau…)

Für die (Zustands-)Vergleich zweier C-Strings wird das Makro EXPECT\_STREQ verwendet (resp. EXPECT\_CASEEQ). Das EXPECT\_EQ-Makro vergleicht im Falle von C-Strings nur Pointer!.

Für string-Objekte kann allerdings wieder das gewöhnliche EXPECT\_EQ-Makro verwendet werden.

Die globale operator <<-Funktion wird von gtest aufgerufen, wenn eine Assertion fehlschlägt – sie erlaubt also ein schönes Fehlerprotokoll (ohne die Existenz dieses Operators würden einfach die Bytes des jeweiligen Objekts hexadezimal ausgegeben).

## Parameter

Dieselbe(n) Testmethoden können im Verlauf eines Tests mehrfach aufgerufen werden – und zwar mit unterschiedlichen "Parametern".

Wir testen wieder unseren MathService:

// MathServiceTest.cpp

class **Param** {

public:

const int **x**;

const int **y**;

const int **gcd**;

const int **lcm**;

**Param**(int x, int y, int gcd, int lcm)

: x(x), y(y), gcd(gcd), lcm(lcm) {

}

};

ostream& **operator <<** (ostream& out, const Param& p) {

return out << "Param [" << p.x << ", "

<< p.y << ", " << p.gcd << ", " << p.lcm << "]";

}

class **MathServiceTest** : public testing::TestWithParam<Param> {

public:

//virtual void **SetUp**() override {

// const Param& p = this->GetParam();

// cout << p << endl; // only for demo...

//}

};

Param **params**[] = {

Param(30, 40, 10, 120),

Param(40, 30, 10, 120),

Param(5, 5, 5, 5),

Param(1, 1, 1, 1),

};

**INSTANTIATE\_TEST\_CASE**\_P(params, MathServiceTest, testing::ValuesIn(params));

TEST\_P(**MathServiceTest**, **gcdNormal**) {

MathService service;

const Param& p = this->GetParam();

cout << "gcdNorml : " << p << endl;

EXPECT\_EQ(p.gcd, service.gcd(p.x, p.y));

}

TEST\_P(**MathServiceTest**, **lcmNormal**) {

MathService service;

const Param& p = this->GetParam();

cout << "lcmNorml : " << p << endl;

EXPECT\_EQ(p.lcm, service.lcm(p.x, p.y));

}

Wir definieren eine Klasse mit dem beispielhaften Namen Param, welche die Inputs und die erwarteten Outputs von GGT- und KGV-Berechnungen enthält.

Wir benötigen dann eine explizite Testklasse, welche abgeleitet ist von testing::TestWithParam<Param>.

Dann können wir einen Array mit beispielhaften Param-Objekten definieren (params).

Mittels des Makros INITIATE\_TEST\_CASE kann die Testklasse dann mit den beispielhaften Param-Objekten verbunden werden.

Die Testmethoden müssen dann mit dem Makro TEST\_P eingeleitet werden.

Für jedes Param-Objekt des params-Arrays werden jeweils beide Testmethoden aufgerufen. Diese können dann das jeweilige Param-Objekt (mit den Inputs und den erwarteten Outputs) mittels der Methode GetParam ermitteln.

Die Diagnose-Ausgaben zeigen, wie der Mechanismus funktioniert:

GTest-Parameterized

[==========] Running 8 tests from 1 test case.

[----------] Global test environment set-up.

[----------] 8 tests from params/MathServiceTest

[ RUN ] params/MathServiceTest.gcdNormal/0

gcdNorml : Param [30, 40, 10, 120]

[ OK ] params/MathServiceTest.gcdNormal/0 (0 ms)

[ RUN ] params/MathServiceTest.gcdNormal/1

gcdNorml : Param [40, 30, 10, 120]

[ OK ] params/MathServiceTest.gcdNormal/1 (0 ms)

[ RUN ] params/MathServiceTest.gcdNormal/2

gcdNorml : Param [5, 5, 5, 5]

[ OK ] params/MathServiceTest.gcdNormal/2 (1 ms)

[ RUN ] params/MathServiceTest.gcdNormal/3

gcdNorml : Param [1, 1, 1, 1]

[ OK ] params/MathServiceTest.gcdNormal/3 (0 ms)

[ RUN ] params/MathServiceTest.lcmNormal/0

lcmNorml : Param [30, 40, 10, 120]

[ OK ] params/MathServiceTest.lcmNormal/0 (0 ms)

[ RUN ] params/MathServiceTest.lcmNormal/1

lcmNorml : Param [40, 30, 10, 120]

[ OK ] params/MathServiceTest.lcmNormal/1 (0 ms)

[ RUN ] params/MathServiceTest.lcmNormal/2

lcmNorml : Param [5, 5, 5, 5]

[ OK ] params/MathServiceTest.lcmNormal/2 (0 ms)

[ RUN ] params/MathServiceTest.lcmNormal/3

lcmNorml : Param [1, 1, 1, 1]

[ OK ] params/MathServiceTest.lcmNormal/3 (0 ms)

[----------] 8 tests from params/MathServiceTest (1 ms total)

[----------] Global test environment tear-down

[==========] 8 tests from 1 test case ran. (1 ms total)

[ PASSED ] 8 tests.

## Listeners

Um die Funktionsweise von gtest eingreifen zu können, können wir eine Listener-Klasse bauen, die von testing::EmptyTestEventListener abgeleitet ist. Dabei können wir die Methoden OnTestStart, OnTestPartResult und OnTestEnd überschreiben. Die Methoden unserer Listener-Klasse werden dann bei der Ausführung jeder Testmethode aufgerufen. Dabei wird jeweils eine testing::TestInfo- resp. eine testing::TestPartResult-Referenz übegeben:

// ResultPrinter.h

class **ResultPrinter** : public testing::EmptyTestEventListener {

virtual void **OnTestStart**(const testing::TestInfo& testInfo) {

printf(">>> Test %s.%s starting.\n",

testInfo.test\_case\_name(), testInfo.name());

}

virtual void **OnTestPartResult**(

const testing::TestPartResult& testPartResult) {

printf("%s in %s:%d\n%s\n",

testPartResult.failed() ? "Failure" : "Success",

testPartResult.file\_name(),

testPartResult.line\_number(),

testPartResult.summary());

}

virtual void **OnTestEnd**(const testing::TestInfo& testInfo) {

printf("<<< Test %s.%s ending.\n",

testInfo.test\_case\_name(), testInfo.name());

}

};

Hier eine einfache Testklasse:

// DemoTest.cpp

TEST(**DemoTest**, **foo**) {

EXPECT\_EQ(77, 77);

}

TEST(**DemoTest**, **bar**) {

EXPECT\_EQ(42, 11);

}

Und hier der Runner – die main-Funktion, die unseren Listener registriert und den default-Listener deregistriert:

// main.cpp

int **main**(int argc, char\*\* argv) {

cout << "GTest-Listener" << endl;

testing::InitGoogleTest(&argc, argv);

testing::TestEventListeners& listeners =

testing::UnitTest::GetInstance()->listeners();

delete listeners.Release(listeners.default\_result\_printer());

listeners.Append(new ResultPrinter);

return RUN\_ALL\_TESTS();

}

Die Ausgaben sind nun viel "kompakter":

GTest-Listener

>>> Test DemoTest.foo starting.

<<< Test DemoTest.foo ending.

>>> Test DemoTest.bar starting.

Failure in ..\src\DemoTest.cpp:14

Expected: 42

To be equal to: 11

<<< Test DemoTest.bar ending.

## Flags

Das Verhalten des gtest-Runners kann mittels zusätzlicher Flags beim Aufruf über die Kommandozeile beeinflusst werden. Solche Flags können auch in der main-Funktion gesetzt werden.

Hier einige Beispiel:

// main.cpp

int **main**(int argc, char\*\* argv) {

cout << "GTest-Flags" << endl;

testing::InitGoogleTest(&argc, argv);

// testing::GTEST\_FLAG(list\_tests) = true;

// ... --gtest\_print\_time=false

testing::GTEST\_FLAG(print\_time) = false;

testing::GTEST\_FLAG(filter)="DemoTest.foo\*:DemoTest.bar";

// ... --gtest\_repeat=2

testing::GTEST\_FLAG(repeat)=2;

int failures = RUN\_ALL\_TESTS();

cout << "failures = " << failures << endl;

return failures;

}

Anhand der Kommentare wird deutlich, wie die Parameter jeweils als Kommandozeilen-Flags übergeben werden könnten.

Die Bedeutung der Flags sollte klar sein.

# Testing – Übungen

Dieses Kapitel enthält einige Übungen, in denen die zu testenden Klassen bereits vorgegeben sind. Die Aufgabe besteht jeweils darin, diese Klassen nun "im nachhinein" zu testen.

* In der ersten Übung geht's um den Test einer Stack-Klasse.
* In der zweiten Übung wird eine Trimmer-Klasse getestet. Die Trimmer-Klasse liest eine Textdatei ein und gibt diese in komprimierter Form (in "getrimmter" Form) wieder aus.
* In der dritten Übung schließlich geht's um das Testen einer vorgegeben CSVReader-Klasse.
* In der vierten Übung geht's schließlich um den Test einer StateMachine-Klasse.
* In der fünften Übung geht's um den Test eines CommandHandlings-Konzepts.
* In der sechsten Übung geht's schließlich um den Test einer selbstentwickelten SharedPtr-Klasse. Der Test dieser Klasse ist nicht ganz trivial…

Natürlich müssen nicht alle Aufgaben gelöst werden – es handelt sich nur um Angebote. "Musterlösungen" finden sich in einer separaten Workspace.

## Stack

Die zu testende Klasse:

// Stack.h

// ...

template <typename T>

class **Stack** {

private:

int **size\_**;

const int **capacity\_**;

T\* **elements\_**;

void **requireNotEmpty**() const {

if (size\_ == 0)

throw runtime\_error("stack mustn't be empty");

}

void **requireNotFull**() const {

if (size\_ == capacity\_)

throw runtime\_error("stack mustn't be full");

}

void **copyElements**(const Stack<T>& source) {

for(unsigned int i = 0; i < source.size\_; i++)

elements\_[i] = source.elements\_[i];

}

public:

**Stack**(unsigned int capacity)

: size\_(0), capacity\_(capacity), elements\_(new T[capacity]) {

}

virtual **~Stack**() {

delete[] elements\_;

}

**Stack**(const Stack<T>& source)

: size\_(source.size\_),

capacity\_(source.capacity\_),

elements\_(new T[source.capacity\_]) {

copyElements(source);

}

Stack<T>& **operator =** (const Stack<T>& source);

bool isEmpty() const {

return size\_ == 0;

}

bool **isFull**() const {

return size\_ == capacity\_;

}

void **push**(const T& element) {

requireNotFull();

elements\_[size\_] = element;

size\_++;

}

const T& **top**() const {

requireNotEmpty();

return elements\_[size\_ - 1];

}

const T& **pop**() {

requireNotEmpty();

const T& element = elements\_[size\_ - 1];

size\_--;

return element;

}

bool **operator==**(const Stack& other) const;

bool **operator!=**(const Stack& other) const {

return !(\*this == other);

}

};

template <typename T>

Stack<T>& **Stack<T>::operator =** (const Stack<T>& source) {

if (capacity\_ != source.capacity\_)

throw runtime\_error("capacities must be equal");

size\_ = source.size\_;

copyElements(source);

return \*this;

}

template <typename T>

bool **Stack<T>::operator==**(const Stack& other) const {

if (capacity\_ != other.capacity\_)

return false;

if (size\_ != other.size\_)

return false;

for (unsigned int i = 0; i < size\_; i++)

if (elements\_[i] != other.elements\_[i])

return false;

return true;

}

Der bereits vorgegebene Testrahmen:

// StackTest.cpp

// ...

class **StackTest** : public testing::Test {

};

TEST\_F(**StackTest**, **constructor1**) {

Stack<int> stack(2);

EXPECT\_TRUE(stack.isEmpty());

}

TEST\_F(**StackTest**, **constructor2**) {

Stack<int> stack(2);

EXPECT\_FALSE(stack.isFull());

}

TEST\_F(**StackTest**, **push**) {

// TODO

}

TEST\_F(**StackTest**, **pushWhenFull**) {

// TODO

}

TEST\_F(**StackTest**, **top**) {

// TODO

}

TEST\_F(**StackTest**, **topWhenEmpty**) {

// TODO

}

TEST\_F(**StackTest**, **pop**) {

// TODO

}

TEST\_F(**StackTest**, **popWhenEmpty**) {

// TODO

}

TEST\_F(**StackTest**, **assingment**) {

// TODO

}

TEST\_F(**StackTest**, **copyConstructor**) {

// TODO

}

TEST\_F(**StackTest**, **equal**) {

// TODO

}

## Trimmer

Die zu testende Klasse:

// Trimmer.h

class **Trimmer** {

public:

static void **trim**(const char\* inputFileName, const char\* outputFileName);

static void **trim**(istream& inputStream, ostream& outputStream);

};

// Trimmer.cpp

// ...

void **Trimmer::trim**(const char\* inputFileName, const char\* outputFileName) {

ifstream in(inputFileName);

ofstream out(outputFileName);

trim(in, out);

in.close();

out.close();

}

void **Trimmer::trim**(istream& in, ostream& out) {

int ch = in.get();

while (ch != -1) {

bool isEmpty = true;

int nTrailingBlanks = 0;

while(ch != -1 && ch != '\n') {

if (ch != ' ' && ch != '\t') {

isEmpty = false;

}

if (! isEmpty) {

if (ch != ' ' && ch != '\t') {

for(int i = 0; i < nTrailingBlanks; i++)

out << " ";

nTrailingBlanks = 0;

out << (char)ch;

}

else

nTrailingBlanks++;

}

ch = in.get();

}

if (! isEmpty)

out << endl;

ch = in.get();

}

}

Der Testrahmen:

// TrimmerTest.cpp

// ...

class **TrimmerTest** : public testing::Test {

};

TEST\_F(**TrimmerTest**, **emptyInput**) {

stringstream in;

stringstream out;

Trimmer::trim(in, out);

EXPECT\_EQ(0, out.str().length());

}

TEST\_F(**TrimmerTest**, **singleFullLine**) {

// TODO

}

TEST\_F(**TrimmerTest**, **singleLineWithLeadingAndTrailingBlanks**) {

// TODO

}

TEST\_F(**TrimmerTest**, **emptyLines**) {

// TODO

}

TEST\_F(**TrimmerTest**, **onlyEmptyLines**) {

// TODO

}

## CSVReader

Die zu testende Klasse:

// CSVReader.h

// ...

class **CSVReader** {

private:

istream& **in\_**;

const char **seperator\_**;

const unsigned int **tokenCount\_**;

int **current\_**;

void **next**() {

current\_ = in\_.get();

}

void **skipWhitespace**() {

while (current\_ == ' ' || current\_ == '\t')

next();

}

public:

**CSVReader**(istream& in, char seperator, unsigned int tokenCount) :

in\_(in), seperator\_(seperator), tokenCount\_(tokenCount) {

current\_ = in\_.get();

}

virtual **~CSVReader**() {

}

string\* **readLine**();

char **seperator**() {

return seperator\_;

}

unsigned int **tokenCount**() {

return tokenCount\_;

}

};

// CSVReader.cpp

// ...

string\* **CSVReader::readLine**() {

string\* tokens = new string[tokenCount\_];

unsigned int index = 0;

if (current\_ == -1)

return 0;

while (index == 0) {

while (current\_ != -1 && current\_ != '\n') {

skipWhitespace();

stringstream stream;

while (current\_ != -1 &&

current\_ != '\n' && current\_ != seperator\_) {

stream << (char) current\_;

next();

}

string token = stream.str();

token.erase(token.find\_last\_not\_of(" \t") + 1);

tokens[index] = token;

index++;

next();

}

next();

}

if (index != tokenCount\_)

throw runtime\_error("illegal line");

return tokens;

}

Der Testrahmen:

// **CSVReaderTest**.cpp

// ...

class **CSVReaderTest** : public testing::Test {

protected:

void **assertTokens**(string\* tokens,

string s0, string s1, string s2, string s3) {

ASSERT\_TRUE(tokens != 0);

EXPECT\_EQ(s0, tokens[0]);

EXPECT\_EQ(s1, tokens[1]);

EXPECT\_EQ(s2, tokens[2]);

EXPECT\_EQ(s3, tokens[3]);

delete[] tokens;

}

};

TEST\_F(**CSVReaderTest**, **emptyInput**) {

stringstream in("");

CSVReader reader(in, ';', 4);

EXPECT\_EQ(0, reader.readLine());

}

TEST\_F(**CSVReaderTest**, **moreLines**) {

stringstream in(

" 1111 ; Pascal; 30.0; 1970\n\n 2222 ; Modula; 40.0; 1980\n");

// TODO

}

TEST\_F(**CSVReaderTest**, **moreLinesWithoutWhitespace**) {

stringstream in(

"1111;Pascal;30.0;1970\n2222;Modula;40.0;1980\n");

// TODO

}

## Statemachine

Die zu testenden Klasse:

// Statemachine.h

// ...

class **Event**;

class **StateMachine**;

class **State** {

public:

const StateMachine\* **machine**;

const int **id**;

**State**(StateMachine\* machine, int id) :

machine(machine), id(id) {

}

virtual **~State**() {

}

virtual void **onError**(Event\* e) {

cout << "\*\*\*\*\* ERROR \*\*\*\*" << endl;

}

};

class **Event** {

public:

virtual **~Event**() {

}

virtual int **dispatch**(State\* state) = 0;

};

class **StateMachine** {

private:

vector<State\*> **states\_;**

State\* **current\_;**

protected:

**StateMachine**(State\* initialState) :

current\_(initialState) {

}

void **add**(State\* state) {

states\_.push\_back(state);

}

public:

virtual **~StateMachine**() {

}

void **dispatch**(Event\* event) {

State\* old = this->current\_;

int id = event->dispatch(this->current\_);

if (id >= 0)

this->current\_ = states\_[id];

cout << old->id << " ==> " << this->current\_->id << endl;

delete event;

}

State\* current() {

return current\_;

}

protected:

virtual void **onUnknownEvent**(Event\* e) {

cout << "\*\*\*\*\* unknown event \*\*\*\*\*" << endl;

}

};

#define **DECLARE\_EVENT**(eventClass) \

class Handler { \

public: \

virtual ~Handler () { } \

virtual int handle (eventClass\* e) = 0; \

}; \

virtual int dispatch (State\* state) { \

Handler\* handler = dynamic\_cast <Handler\*> (state); \

if (handler != 0) \

return handler->handle (this); \

state->onError (this); \

return -1; \

}

Für's Testen können folgende Event-Klassen verwendet werden:

// Events.h

// ...

struct **EventA**: public Event {

const int **i**;

**EventA**(int i) :

i(i) {

}

**DECLARE\_EVENT**(EventA)

};

struct **EventB**: public Event {

const string **s**;

**EventB**(const string s) :

s(s) {

}

**DECLARE\_EVENT**(EventB)

};

Und folgende State-Klassen:

// States.h

typedef enum {

**X**, **Y**, **Z**

} **StateIds**;

struct **StateX**: public State, public EventA::Handler {

**StateX**(StateMachine\* machine) :

State(machine, StateIds::X) {

}

virtual int **handle**(EventA\* e) {

cout << "StateX - EventA: " << e->i << endl;

return StateIds::Y;

}

};

struct **StateY**: public State, public EventB::Handler {

**StateY**(StateMachine\* machine) :

State(machine, StateIds::Y) {

}

virtual int **handle**(EventB\* e) {

cout << "StateY - EventB: " << e->s << endl;

return StateIds::Z;

}

};

struct **StateZ**: public State, public EventA::Handler, public EventB::Handler {

**StateZ**(StateMachine\* machine) :

State(machine, StateIds::Z) {

}

virtual int **handle**(EventA\* e) {

cout << "StateZ - EventA: " << e->i << endl;

return StateIds::Y;

}

virtual int **handle**(EventB\* e) {

cout << "StateZ - EventB: " << e->s << endl;

return StateIds::X;

}

};

class **MyStateMachine**: public StateMachine {

public:

StateX **sx**;

StateY **sy**;

StateZ **sz**;

**MyStateMachine**() :

StateMachine(&sx), sx(this), sy(this), sz(this) {

add(&sx);

add(&sy);

add(&sz);

}

};

Der Testrahmen:

// StatemachineTest.cpp

// ...

class **StateMachineTest** : public testing::Test {

};

TEST\_F(**StateMachineTest**, **InitialState**) {

MyStateMachine machine;

EXPECT\_EQ(&machine.sx, machine.current());

}

TEST\_F(**StateMachineTest**, **EventA**) {

MyStateMachine machine;

machine.dispatch(new EventA(42));

EXPECT\_EQ(&machine.sy, machine.current());

}

// TODO

## CommandHandling

Die zu testenden Klassen:

// CommandHandling.h

#include <vector>

using namespace std;

class **IHandler**;

class **ICommand**;

class **Handlers**;

class **IHandler** {

public:

virtual **~IHandler**() {

}

};

class **ICommand** {

public:

virtual **~ICommand**() {

}

virtual void **dispatch**(Handlers& handlers) = 0;

};

class **Handlers** {

private:

vector<IHandler\*> **handlers\_**;

bool **deleteHandlers\_**;

bool **deleteCommands\_**;

public:

**Handlers**(bool deleteHandlers = true, bool deleteCommands = true) :

deleteHandlers\_(deleteHandlers), deleteCommands\_(deleteCommands) {

}

Handlers& **operator <<** (IHandler\* handler) {

handlers\_.push\_back(handler);

return \*this;

}

virtual **~Handlers**() {

if (deleteHandlers\_) {

for (unsigned int i = 0; i < handlers\_.size(); ++i)

delete handlers\_.at(i);

}

}

void **dispatch**(ICommand\* cmd) {

cmd->dispatch(\*this);

if (deleteCommands\_)

delete cmd;

}

unsigned int **size**() const {

return handlers\_.size();

}

IHandler\* **at**(unsigned int index) const {

return handlers\_.at(index);

}

};

#define **COMMAND**(className) \

class **Handler**: public IHandler { \

public: \

virtual void **handle**(className& cmd) = 0; \

}; \

void **dispatch**(Handlers& handlers) { \

for (unsigned int i = 0; i < handlers.size(); i++) { \

Handler\* hh = dynamic\_cast<Handler\*>(handlers.at(i)); \

if (hh != 0) \

hh->handle(\*this); \

} \

}

Der Beispielklassen, die im Test verwendet werden sollten, und der Testrahmen existieren bereits:

// CommandHandlingTest.cpp

// ...

class **XCommand** : public ICommand {

public:

**COMMAND**(XCommand)

int **i**;

**XCommand**(int i) : i (i) { }

};

class **YCommand** : public ICommand {

public:

**COMMAND**(YCommand)

double **d**;

**YCommand**(double d) : d (d) { }

};

class **XYHandler** : public XCommand::Handler, public YCommand::Handler {

public:

vector<int> xValues;

vector<double> yValues;

void **handle**(XCommand& x) {

xValues.push\_back(x.i);

}

void **handle**(YCommand& y) {

yValues.push\_back(y.d);

}

};

TEST(**ComCommandHandlingTest**, **twoCommandTypesOneHandlerType**) {

// TODO...

}

class **XHandler** : public XCommand::Handler {

public:

vector<int> **xValues**;

void **handle**(XCommand& x) {

xValues.push\_back(x.i);

}

};

class **YHandler** : public YCommand::Handler {

public:

vector<double> **yValues**;

void **handle**(YCommand& y) {

yValues.push\_back(y.d);

}

};

TEST(**ComCommandHandlingTest**, **twoCommandTypesTwoHandlerTypes**) {

// TODO...

}

## SharedPtr

Die zu testende Klasse:

// SharedPrt.h

// ...

template<typename T>

class **SharedPtr** {

private:

class **Counter** {

friend SharedPtr;

private:

int **count\_**;

public:

**Counter**() :

count\_(1) {

}

void **inc**() {

count\_++;

}

void **dec**() {

count\_--;

}

bool **isZero**() {

return count\_ == 0;

}

};

T\* **ptr\_**;

Counter\* **counter\_**;

public:

**SharedPtr**() :

ptr\_(0), counter\_(0) {

}

**SharedPtr**(T\* ptr) :

ptr\_(ptr), counter\_(0) {

if (ptr != 0)

counter\_ = new Counter();

}

**SharedPtr**(const SharedPtr<T>& other) :

ptr\_(other.ptr\_), counter\_(other.counter\_) {

counter\_->inc();

}

**~SharedPtr**() {

if (counter\_) {

counter\_->dec();

if (counter\_->isZero()) {

delete counter\_;

delete ptr\_;

}

}

}

SharedPtr<T>& **operator =**(const SharedPtr<T>& other) {

if (counter\_) {

counter\_->dec();

if (counter\_->isZero()) {

delete counter\_;

delete ptr\_;

}

}

counter\_ = other.counter\_;

ptr\_ = other.ptr\_;

if (other.counter\_)

counter\_->inc();

return \*this;

}

T& **operator\*()** const {

return \*ptr\_;

}

T\* **operator->()** const {

return ptr\_;

}

operator **bool()** {

return ptr\_ != 0;

}

int **count()** {

return counter\_ ? counter\_->count\_ : 0;

}

};

template<typename T>

ostream& **operator <<**(ostream& out, SharedPtr<T>& ptr) {

return out << \*ptr;

}

Wir können folgende Testklasse verwenden:

// foo.h

// ...

class **Foo** {

private:

**Foo**(const string& s) : s(s) {

count\_++;

}

static int **count\_**;

public:

static Foo\* **create**(const string& s) {

return new Foo(s);

}

string **s**;

virtual **~Foo**() {

count\_--;

}

static int **count**() {

return count\_;

}

};

ostream& **operator <<** (ostream& out, const Foo& foo);

// foo.cpp

// ...

int **Foo::count\_** = 0;

ostream& **operator <<** (ostream& out, const Foo& foo) {

return out << "Foo [" << foo.s << "]";

}

Der Testrahmen:

// SharedPtrTest.cpp

// ...

class **SharedPtrTest**: public testing::Test {

};

TEST\_F(**SharedPtrTest**, **simple**) {

{

SharedPtr<Foo> p(Foo::create("spring"));

EXPECT\_EQ(1, Foo::count());

EXPECT\_EQ(1, p.count());

Foo& f = \*p;

EXPECT\_EQ("spring", f.s);

EXPECT\_EQ("spring", p->s);

}

EXPECT\_EQ(0, Foo::count());

}

TEST\_F(**SharedPtrTest**, **copy**) {

{

SharedPtr<Foo> p1(Foo::create("spring"));

SharedPtr<Foo> p2(p1);

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

TEST\_F(**SharedPtrTest**, **assign**) {

{

SharedPtr<Foo> p1(Foo::create("spring"));

SharedPtr<Foo> p2(Foo::create("summer"));

p2 = p1;

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

TEST\_F(**SharedPtrTest**, **assignToEmpty**) {

{

SharedPtr<Foo> p1(Foo::create("spring"));

SharedPtr<Foo> p2;

p2 = p1;

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

TEST\_F(**SharedPtrTest**, **assignFromEmpty**) {

{

SharedPtr<Foo> p1;

SharedPtr<Foo> p2(Foo::create("spring"));

p2 = p1;

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

TEST\_F(**SharedPtrTest**, **assignFromEmptyToEmpty**) {

{

SharedPtr<Foo> p1;

SharedPtr<Foo> p2;

p2 = p1;

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

SharedPtr<Foo> **returnSharedPtr**() {

SharedPtr<Foo> f(Foo::create("spring"));

return f;

}

TEST\_F(**SharedPtrTest**, **returnSharedPtr**) {

{

SharedPtr<Foo> p = returnSharedPtr();

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

string **passSharedPtr**(SharedPtr<Foo> p) {

EXPECT\_EQ(2, p.count());

return p->s;

}

TEST\_F(**SharedPtrTest**, **passSharedPtr**) {

{

SharedPtr<Foo> p (Foo::create("spring"));

// TODO

}

EXPECT\_EQ(0, Foo::count());

}

## Resultate

Bei den bisherigen Tests sind bereits einige wichtige Eigenschaften von Testklassen deutlich geworden:

### Sprechende Namen

Testmethoden sollten – wie alle anderen Methoden natürlich auch - sprechende Namen haben (auch wenn das zuweilen dazu führt, dass recht lange Namen erforderlich sind). Testmethoden sollten also nicht aufwendig kommentiert werden müssen.

### Unabhängigkeit

Testmethoden müssen voneinander vollständig unabhängig sein. Eine Testmethode darf sich insbesondere nicht auf ein Ergebnis verlassen, welches von einer anderen Testmethode berechnet wurde. Instanzvariablen haben also in Testklassen nichts zu suchen (es sei denn, man benutzt SetUp/TearDown-Methoden). Testmethoden sollten also in sich abgeschlossen sein.

Hier ein katastrophales Beispiel:

// Ein katastrophales Beispiel!!!

Stack<String>\* **stack;**

TEST(**StackTest**, **testCreate**) {

stack = new Stack<String>();

ASSERT\_EQ(0, stack->size());

}

TEST(**StackTest**, **testPush**) {

stack->push("one");

stack->push("two");

ASSERT\_EQ(2, stack->size());

}

TEST(**StackTest**, **testException**) {

try {

stack->push("three");

FAIL();

}

catch(runtime\_error& ) {

}

}

Der Entwickler hat sich offenbar gedacht, dass die Testmethoden in exakt derjenigen Reihenfolge ausgeführt werden, in welcher sie textuell hinterlegt sind. Genau darauf aber ist kein Verlass! Würde z.B. die zweite der obigen Testmethode als erste ausgeführt werden, würde das Programm abstürzen.

### Strukturelle Einfachheit

Testmethoden sollten keine komplexen Kontrollstrukturen enthalten. Beim Testen des push-Verhaltens eines Stacks hätte man z.B. folgende Methode schreiben können:

TEST(**StackTest**, **testPush**() {

Stack<String>\* stack = new Stack<String>(2);

String elements[] = { "one", "two", "three" };

int i = 0;

try {

while (i < 3) {

stack->push(elements[i]);

i++;

}

FAIL();

}

catch(runtime\_error&) {

if (i != 2)

FAIL();

}

delete stack;

}

Diese Testmethode ist wesentlich komplexer als die zu testende Methode – sie enthält wahrscheinlich eher Fehler als die push-Methode. Folgende Testmethode ist wesentlich verständlicher:

TEST(**StackTest**, **testPush**() {

Stack<String> stack(2);

stack.push("one");

stack.push("two");

try {

stack.push("three");

FAIL();

}

catch(runtime\_error&) {

}

}

### Vollständigkeit

Im strengen Wortsinn kann ein Test natürlich niemals "vollständig" sein. Angenommen, wir wollen eine Methode testen, welche die Wurzel einer Gleitkommazahl berechnet. Leider gibt's unendlich viele solcher Zahlen – die Wurzel-Methode kann also niemals mit allen Zahlen getestet werden.

In der Regel bildet man sog. Äquivalenzklassen. Eine Äquivalenzklasse ist ein Bereich von Eingabe-Werten für eine zu testende Methode. Diese Bereiche werden derart festgelegt, dass man annehmen kann, dass eine Methode für alle Werte eines jeweiligen Bereichs sich gleichartig verhalten wird. Bei der Wurzel-Methode wird man folgende Bereiche festlegen: den Bereich der negativen Zahlen (hier muss die Methode eine Exception werfen; den Bereich mit dem einzigen Wert 0 – hier muss 0 zurückgeliefert werden; den Bereich der Werte größer als 0 aber kleiner als 1 – für alle Zahlen dieses Bereichs muss ein Wert zurückgeliefert werden, der größer ist als der Eingabe-Wert; der Bereich mit dem Wert 1 – hier muss 1 geliefert werden; und schließlich den Bereich aller Werte größer als 1 – hier muss ein Wert zurückgeliefert werden, welcher kleiner als der Eingabewert ist. Für alle diese Bereiche wird dann ein Stellvertreter-Wert ausgewählt, mit welchem der Test ausgeführt wird.

Hat eine Klasse eine Vielzahl von Methoden, die auf bestimmte Weise zusammenspielen (wie z.B. im Stack-Beispiel), muss natürlich dieses Zusammenspiel getestet werden (pop nimmt die Wirkung von push zurück etc.).

Insbesondere sollte das Verhalten bei illegalen Eingaben (nicht positive Werte für MathService.gcd) resp. illegalen Sequenzen von Methoden-Aufrufen (Stack.pop im Zustand Stack.isEmpty etc.) getestet werden.

Triviale Tests sollten nach Meinung des Autors vermieden werden. Sei etwa folgende Klasse gegeben:

struct **Point** {

int **x;**

int **y**;

}

Folgender Test ist überflüssig:

TEST(**PointTest**, **blahblah**) {

Point p;

p.x = 42;

ASSERT\_EQ(42, p.x);

}

Solche Tests sind langweilig – sie verleiden einem den Spaß am Testen. Und man sollte sich schließlich nicht dümmer stellen als man ist...

### Unabhängigkeit von problematischen Ressourcen

Ein Test, der z.B. von externen Dateien abhängt, ist problematisch. Die Eingabedatei, aus welcher die zu testende Methode liest, mag versehentlich verändert worden sein – oder gelöscht worden sein. Dann wird die Methode natürlich alles andere liefern als das erwartete Ergebnis. Sofern das Resultat der Methode eine Ausgabe ist, muss die Ausgabedatei im Testprogramm wieder eingelesen werden...

In solchen Fällen sollte man sich überlegen, ob die Eingaben nicht direkt im Testprogramm hinterlegt werden können – siehe das Trimmer- und das CSVReader-Beispiel.

Man möchte beim Testen auch gern von Datenbanken unabhängig sein – auch dies sind problematische externe Ressourcen. Oder von Socket-Verbindungen etc. Eine solche Unabhängigkeit kann aber erst dann realisiert werden, wenn Mocking-Werkzeuge eingesetzt werden.

### Test als Spezifikation und Dokumentation

Eine Testklasse sollte man als Spezifikation der in ihr getesteten Klasse(n) ansehen können – und damit zugleich auch als deren Dokumentation...

# Mocking – Das gmock-Framework

Im Folgenden geht's um die Benutzung eines Mocking-Werkzeugs. Wir verwenden das gmock-Werkzeug von Google.

Worum geht's beim Mocking?

Seien zwei Klassen A und B gegeben. Die Klasse A benötigt ein Objekt, dessen Klasse ein Interface IB implementiert. Aus irgendwelchen Gründen steht beim Testen aber eine solche Klasse (z.B. B) nicht zur Verfügung (sie ist noch nicht fertig; oder sie benötigt den Zugriff auf eine Datenbank, deren Inhalt dem Wechsel unterliegt und also zur wiederholten Produktion von Testergebnissen nicht brauchbar ist; oder B beruht auf Socket-Verbindungen, deren Problematik aber beim Test der Klasse A nicht gewünscht ist...)

Dann stellt sich die Frage, wie die Klasse A getestet werden kann, ohne dass eine "richtige" Implementierung von IB genutzt wird. Es soll getestet werden, ob A mit einem IB-Objekt korrekt zusammenspielt (Interaktionstest). Es muss also das "Innere" von A getestet werden: ruft A Methoden von IB in erwarteter Reihenfolge und mit den erwarteten Argumenten auf? (Endoskopischer Test). Die Klasse A wird bei solchen Test somit als White-Box betrachtet.

Hier das Objektdiagramm zur oben beschriebenen Problematik:

: A

: B

*IB*

Was tun, wenn B fehlt?:

: A

*IB*

Man benötigt einen Mock für IB.

## Assertions

Mittels eines XMLWriters kann auf bequeme Weise ein XML-Text erzeugt werden. Der XMLWriter benutzt einen Ausgabe-Schreiber, welcher über das Interface IPrinter spezifiziert ist. Dieser Ausgabe-Schreiber wird gemockt.

: XMLWriter

: ????

begin(name)

text(value)

end()

println(s))

*IPrinter*

*IXMLWriter*

Ein XMLWriter kann z.B. wie folgt verwendet werden:

IPrinter\* printer = ...

XMLWriter xmlWriter(\*printer)

xmlWriter.begin("a");

xmlWriter.begin("b");

xmlWriter.text("hello");

xmlWriter.end();

xmlWriter.begin("c");

xmlWriter.text("world");

xmlWriter.end();

xmlWriter.end();

Er muss dann folgende Ausgabe erzeugen:

<a>

<b>

hello

</b>

<c>

world

</c>

</a>

Dem Konstruktor von XMLWriter wird ein IPrinter übergeben.

Der Nutzen eines XMLWriters besteht offenbar darin, dass man weder an die spitzen Klammern noch an die Einrückungen denken muss. Und beim Abschluss eines Elements muss nurmehr end aufgerufen werden – parameterlos.

In diesem Beispiel geht es nur um die Verifizierung erwarteter Aufrufe.

Die Interfaces:

// IXMLWriter.h

// ...

class **IXMLWriter** {

public:

virtual **~IXMLWriter**() { }

virtual void **begin**(const string& name) = 0;

virtual void **text**(const string& value) = 0;

virtual void **end**() = 0;

};

// IPrinter.h

// ...

class **IPrinter** {

public:

virtual **~IPrinter()** { }

virtual void **print**(unsigned int indent, const string& line) = 0;

};

Die Implementierung von XMLWriter benutzt einen Stack. Beim Aufruf von begin wird der an begin übergebene Element-Name auf den Stack gelegt; beim Aufruf von end wird der oberste Name vom Stack entfernt und zur Ausgabe des Ende-Tags verwendet. Aufgrund der Anzahl der auf dem Stack liegenden Namen kann die jeweils erforderliche Einrückung berechnet werden.

// XMLWriter.h

// ...

// #include "IPrinter.h"

class **XMLWriter**: public IXMLWriter {

private:

IPrinter& **printer\_**;

vector<string> **stack\_**;

public:

**XMLWriter**(IPrinter& printer) :

printer\_(printer) {

}

void **begin**(const string& name) {

stringstream stream;

stream << "<" << name << ">";

printer\_.print(stack\_.size(), stream.str());

stack\_.push\_back(name);

}

void **text**(const string& value) {

printer\_.print(stack\_.size(), value);

}

void **end**() {

if (stack\_.size() == 0)

throw runtime\_error("illegal call of end()");

stringstream stream;

stream << "</" << stack\_[stack\_.size() - 1] << ">";

stack\_.erase(stack\_.end());

printer\_.print(stack\_.size(), stream.str());

}

};

Hier die Testklasse:

// XMLWriterTest.cpp

// ...  
#include "gmock/gmock.h"

class **MockedPrinter** : public IPrinter {

public:

**MOCK\_METHOD2**(print, void(unsigned int, const string&));

};

TEST(**XMLWriter**, **emptyElement**) {

MockedPrinter p;

XMLWriter w(p);

EXPECT\_CALL(p, print(0, "<a>"));

EXPECT\_CALL(p, print(0, "</a>"));

w.begin("a");

w.end();

}

TEST(**XMLWriter**, **elementWithText**) {

MockedPrinter p;

XMLWriter w(p);

EXPECT\_CALL(p, print(0, "<a>"));

EXPECT\_CALL(p, print(1, "Hello"));

EXPECT\_CALL(p, print(0, "</a>"));

w.begin("a");

w.text("Hello");

w.end();

}

TEST(**XMLWriter**, **nestedElements**) {

MockedPrinter p;

XMLWriter w(p);

EXPECT\_CALL(p, print(0, "<a>"));

EXPECT\_CALL(p, print(1, "<b>"));

EXPECT\_CALL(p, print(2, "Hello"));

EXPECT\_CALL(p, print(1, "</b>"));

EXPECT\_CALL(p, print(0, "</a>"));

w.begin("a");

w.begin("b");

w.text("Hello");

w.end();

w.end();

}

TEST(**XMLWriter**, **repeatedElements**) {

MockedPrinter p;

XMLWriter w(p);

EXPECT\_CALL(p, print(0, "<a>")).Times(1);

EXPECT\_CALL(p, print(1, "<b>")).Times(2);

EXPECT\_CALL(p, print(2, "Hello"));

EXPECT\_CALL(p, print(1, "</b>")).Times(2);

EXPECT\_CALL(p, print(2, "World"));

EXPECT\_CALL(p, print(0, "</a>"));

w.begin("a");

w.begin("b");

w.text("Hello");

w.end();

w.begin("b");

w.text("World");

w.end();

w.end();

}

TEST(**XMLWriter**, **illegalCallOfEnd**) {

MockedPrinter p;

XMLWriter w(p);

EXPECT\_CALL(p, print(0, "<a>"));

EXPECT\_CALL(p, print(0, "</a>"));

w.begin("a");

w.end();

EXPECT\_THROW(w.end(), runtime\_error);

}

In der Testklasse wird zunächst eine neue Klasse definiert:

class **MockedPrinter** : public IPrinter {

public:

**MOCK\_METHOD2**(print, void(unsigned int, const string&));

};

Die Klasse MockedPrinter implementiert das Interface IPrinter. Dem XMLWriter wird also ein MockedPrinter übergeben werden können. Die print-Methode des Interfaces werden auf recht eigentümliche Weise definiert: mittels des gmock-Makros MOCK\_METHOD2. Diesem Makro werden zwei Parameter übergeben: der Name der zu implementierenden Methode (print) und die Signatur dieser Methode: der Returntyp (void) und die beiden Parametertypen (unsigned int und const string&). Das Makro wird in einer gültigen Implantierung der print-Methode expandieren – wie auch immer diese Implementierung ausschauen mag…

(Neben dem Makro MOCK\_METHOD2 gibt's weitere Makros: MOCK\_METHOD0, MOCK\_METHOD1, MOCK\_METHOD3 etc. für Methoden mit keinem, einem resp. mit drei (etc.) Parametern.

Die erste TEST-Methode soll nun genauer untersucht werden:

TEST(**XMLWriter**, **emptyElement**) {

MockedPrinter p;

XMLWriter w(p);

EXPECT\_CALL(p, print(0, "<a>"));

EXPECT\_CALL(p, print(0, "</a>"));

w.begin("a");

w.end();

}

Wir definieren ein MockedPrinter- und ein XMLWriter-Objekt – wobei der MockedPrinter als Parameter an den Konstruktor von XMLWriter übergeben wird.

Wir überspringen nun zunächst die beiden EXPECT\_CALL-Zeilen. Auf den XMLWriter werden zwei Methoden aufgerufen: begin und end. Wir stellen uns nun die Frage, was diese beiden Methoden-Aufrufe bewirken müssten. Genauer: welche MockedPrinter-Methoden müssten beim Aufruf dieser beiden XMLWriter-Methoden aufgerufen werden – sofern denn diese beiden XMLWriter-Methoden korrekt implementiert sind?

Der Aufruf von begin("a") müsste dazu führen, dass die MockedPrinter-Methode print aufgerufen werden müsste – mit den Parametern 0 (Einrückungstiefe) und "<a>". Der folgende end()-Aufruf müsste dazu führen, dass ebenfalls die print-Methode von MockedPrinter aufgerufen werden müsste – mit den Parametern 0 und "</a>".

Diese beiden Erwartungen bezüglich des Zusammenspiels von XMLWriter und MockedPrinter formulieren wir nun ein zwei EXPECT\_CALL-Zeilen. Diese beiden Zeilen müssen den Aufrufen der XMLWriter-Methoden vorausgehen. An EXPECT\_CALL übergeben wir zwei Parameter: den MockedPrinter und den Aufruf der jeweiligen Methode, den wir erwarten.

gmock kann nun feststellen, ob tatsächlich die beiden erwarteten Aufrufe stattgefunden haben.

Wir könnten in die zu testende Klasse (XMLWriter) einen Fehler einbauen: wir vergessen in der begin-Methoden den Aufruf der print-Methode. Oder wir rufen diese Methode zwar auf, übergeben aber falsche Parameter. Oder wir vergessen in der end-Methode den Aufruf der print-Methode…: all diese Fehler werden erkannt und protokolliert.

Wie funktioniert gmock – ganz grob besehen? Die Ausführung der EXPECT\_CALL Makros wird dazu führen, dass für jeden der erwarteten Aufrufe ein wie auch immer geartetes "Call"-Objekt erzeugt wird, welches diesen erwarteten Aufruf repräsentiert. Jedes dieser Call-Objekte wird dann zu einer Liste der erwarteten Aufrufe hinzugefügt.

Wenn dann im Kontext der Aufrufe der XMLWriter-Methoden die print-Methoden des MockedPrinters ausgeführt werden, wird für jeden dieser tatsächlichen Aufrufe ebenfalls ein "Call"-Objekt erzeugt, welches nun einer Liste der aktuellen Aufrufe hinzugefügt wird. Anschließend (beim Ende des Tests) werden diese beiden Listen (die Liste der erwarteten und die Liste der tatsächlichen Aufrufe) dann miteinander verglichen. Werden beim diesem Vergleich dann Unstimmigkeiten festgestellt, werden diese ins Fehlerprotokoll aufgenommen.

Die folgenden Tests sollten müssen nun nicht weiter erläutert werden – bis auf einen letzten Hinweis: EXPECT\_CALL kann mittels Times "fortgesetzt" werden. Als Parameter wird an Times dann die Anzahl der erwarteten Aufrufe übergeben. Dies ist insbesondere im Test repeatedElements erforderlich.

## Arrangements

Im folgenden zeigen wir, wie ein Mock-Objekt Werte an den Aufrufer zurückliefern kann, welcher dieser zur weiteren Berechnungen benötigt.

Ein Phytagoras kann aufgrund zweier Katheten die Hypotenuse eines rechtwinkligen Dreiecks berechnen – und aufgrund einer Hypotenuse und einer Kathete die zweite Kathete. Bei diesen Berechnungen müssen immer wieder Quadrate und Wurzeln berechnet werden. Zur Erledigung dieser Trivial-Aufgaben benutzt der Phytagoras ein Objekt, dessen Klasse das Interface ICalculator implementiert:

: Pythagoras

: ????

c(a, b)

a(c, b)

sqr(x)

sqrt(x)

*ICalculator*

*IPythagoras*

Die Interfaces:

// IPythagoras.h

class **IPythagoras** {

public:

virtual **~IPythagoras**() { }

virtual double **c**(double a, double b) = 0;

virtual double **a**(double c, double b) = 0;

};

// ICalculator.h

class **ICalculator** {

public:

virtual **~ICalculator**() { }

virtual double **sqr**(double value) = 0;

virtual double **sqrt**(double value) = 0;

};

Die Implementierung des Pythagoras:

// Pythagoras.h

// ...

class **Pythagoras**: public IPythagoras {

private:

ICalculator& **calculator\_**;

public:

**Pythagoras**(ICalculator& calculator) :

calculator\_(calculator) {

}

double **c**(double a, double b) {

return calculator\_.sqrt(calculator\_.sqr(b) + calculator\_.sqr(a));

}

double **a**(double c, double b) {

return calculator\_.sqrt(calculator\_.sqr(c) - calculator\_.sqr(b));

}

};

Ein Pythagoras könnte wie folgt genutzt werden:

ICalculator\* calculator = ...

Pythagoras pythagors(&calculator);

cout << pythagors.c(3.0, 4.0) << endl;

cout << pythagors.a(5.0, 4.0) << endl;

Die berechnete Hypotenuse hat die Größe 5.0, die berechnete Kathete die Größe 3.0.

Die Testklasse:

// Pythagoras.h

// ...

class **MockedCalculator**: public ICalculator {

public:

**MOCK\_METHOD1**(sqr, double(double));

**MOCK\_METHOD1**(sqrt, double(double));

};

using testing::**Return**;

using testing::**Throw**;

TEST(**PythagorasTest**, **c**) {

MockedCalculator c;

Pythagoras p(c);

EXPECT\_CALL(c, sqr(3.0)).WillOnce(Return(9.0));

EXPECT\_CALL(c, sqr(4.0)).WillOnce(Return(16.0));

EXPECT\_CALL(c, sqrt(25.0)).WillOnce(Return(5.0));

ASSERT\_EQ(5.0, p.c(3.0, 4.0));

}

TEST(**PythagorasTest**, **a**) {

MockedCalculator c;

Pythagoras p(c);

EXPECT\_CALL(c, sqr(5.0)).WillOnce(Return(25.0));

EXPECT\_CALL(c, sqr(4.0)).WillOnce(Return(16.0));

EXPECT\_CALL(c, sqrt(9.0)).WillOnce(Return(3.0));

ASSERT\_EQ(3.0, p.a(5.0, 4.0));

}

TEST(**PythagorasTest**, **exceptions**) {

MockedCalculator c;

Pythagoras p(c);

EXPECT\_CALL(c, sqr(3.0)).WillOnce(Return(9.0));

EXPECT\_CALL(c, sqr(4.0)).WillOnce(Return(16.0));

EXPECT\_CALL(c, sqrt(-7.0)).WillOnce(Throw(runtime\_error("")));

ASSERT\_ANY\_THROW(p.a(3.0, 4.0));

}

Wir definieren zunächst wieder eine Mock-Klasse, die das zu mockende Interface (ICalculator) implementiert:

class **MockedCalculator**: public ICalculator {

public:

**MOCK\_METHOD1**(sqr, double(double));

**MOCK\_METHOD1**(sqrt, double(double));

};

Man beachte, dass hier das Makro MOCK\_METHOD1 benutzt wird (beide zu mockende Methoden haben genau einen Parameter). Der Parameter und beider Methoden ist vom Typ double – und auch der Ergebnistyp beider Methoden ist double.

Am Anfang jeder Testmethode wird ein MockedCalculator und ein Pythagoras definiert – wobei der MockedCalculator dem Pythagoras via Konstruktor injiziert wird.

Die EXPECT\_CALL-Aufrufe werden in allen Testmethoden nun "ergänzt" durch die Klausel WillOnce(Return(...)). Der Return-Klausel wird derjenige Wert übergeben, welcher von der gemockten Methode zurückgeliefert werden soll – wenn denn der Aufruf dieser Methode wie erwartet stattfindet. Mit diesen Werten kann der Aufrufer der Mock-Methode dann weiterrechnen:

Soll also eine Aufruf einer Methode "arrangiert" werden, die einen Wert liefert, wird Return verwendet. Neben Return kann auch Throw verwendet werden, wenn arrangiert werden soll, dass die aufgerufene Methode eine Exception wirft – siehe hierzu den letzten der obigen drei Tests.

Was testen wir in der letzten Testmethode? Wir setzen voraus, dass ein Calculator die ihm übergebenen Parameter auf Plausibilität prüft. Wird der sqrt-Methode z.B. ein negativer Wert übergeben, wird der Calculator eine Exception liefern. Wir unterstellen weiterhin, dass Pythagoras keinerlei Prüfungen des an sqrt übergebenen Parameter vornimmt. Dann wird Pythagoras die sqrt-Methode mit einem negativen Wert aufrufen – und der Calculator also eine Exception werfen. Diese Exception sollte als solche an die "Anwendung" weitergereicht werden…

# Mocking – das Fakeit-Framework

Fakeit ist ein weiteres Mock-Werkzeug – ein Werkzeug, welches allerdings C++11 voraussetzt.

Fakeit erscheint mächtiger als gmock – und gleichzeitig aber auch einfacher zu nutzen. So muss z.B. keine explizite Mock-Klasse für das zu mockende Interface spezifiziert werden – all das geschieht hinter unserem Rücken.

Im folgenden benutzen wir wieder die beiden im letzten Kapitel diskutierten Beispiele: XMLWriter / IPrinter und Pythagoras / ICalculator.

Wir werden diese Beispiel aber nicht weiter kommentieren…

## Assertions

// XMLWriterTest.cpp

// ...

#include "fakeit.hpp"

using namespace fakeit;

TEST(**XMLWriter**, **emptyElement**) {

Mock<IPrinter> mock;

//When(Method(mock,print)).AlwaysReturn();

Fake(Method(mock,print));

IPrinter& p = mock.get();

XMLWriter w(p);

w.begin("a");

w.end();

Verify(Method(mock,print).Using("<a>")).Once();

Verify(Method(mock,print).Using("</a>")).Once();

}

TEST(**XMLWriter**, **elementWithText**) {

Mock<IPrinter> mock;

Fake(Method(mock,print));

IPrinter& p = mock.get();

XMLWriter w(p);

w.begin("a");

w.text("Hello");

w.end();

Verify(Method(mock,print).Using("<a>")).Once();

Verify(Method(mock,print).Using("\tHello")).Once();

Verify(Method(mock,print).Using("</a>")).Once();

}

TEST(**XMLWriter**, **nestedElements**) {

Mock<IPrinter> mock;

Fake(Method(mock,print));

IPrinter& p = mock.get();

XMLWriter w(p);

w.begin("a");

w.begin("b");

w.text("Hello");

w.end();

w.end();

Verify(Method(mock,print).Using("<a>")).Once();

Verify(Method(mock,print).Using("\t<b>")).Once();

Verify(Method(mock,print).Using("\t\tHello")).Once();

Verify(Method(mock,print).Using("\t</b>")).Once();

Verify(Method(mock,print).Using("</a>")).Once();

}

TEST(**XMLWriter**, **repeatedElements**) {

Mock<IPrinter> mock;

Fake(Method(mock,print));

IPrinter& p = mock.get();

XMLWriter w(p);

w.begin("a");

w.begin("b");

w.text("Hello");

w.end();

w.begin("b");

w.text("World");

w.end();

w.end();

Verify(Method(mock,print).Using("<a>")).Once();

Verify(Method(mock,print).Using("\t<b>")).Twice();

Verify(Method(mock,print).Using("\t\tHello")).Once();

Verify(Method(mock,print).Using("\t\tWorld")).Once();

Verify(Method(mock,print).Using("\t</b>")).Twice();

Verify(Method(mock,print).Using("</a>")).Once();

}

Damit fakeit mit diesem Beispiel klarkommt, muss der Parameter von IPrinter.print geändert werden: const string& muss durch string ersetzt werden (fakeit hat offenbar Probleme mit Referenz-Parametern…)

## Arrangements

// PythagorasTest.cpp

// ...

#include "fakeit.hpp"

using namespace fakeit;

TEST(**PythagorasTest**, **c**) {

Mock<ICalculator> mock;

When(Method(mock,sqr)(3.0)).Return(9.0);

When(Method(mock,sqr)(4.0)).Return(16.0);

When(Method(mock,sqrt)(25.0)).Return(5.0);

ICalculator& c = mock.get();

Pythagoras p(c);

ASSERT\_EQ(5.0, p.c(3.0, 4.0));

}

TEST(**PythagorasTest**, **a**) {

Mock<ICalculator> mock;

When(Method(mock,sqr)(5.0)).Return(25.0);

When(Method(mock,sqr)(4.0)).Return(16.0);

When(Method(mock,sqrt)(9.0)).Return(3.0);

ICalculator& c = mock.get();

Pythagoras p(c);

ASSERT\_EQ(3.0, p.a(5.0, 4.0));

}

TEST(**PythagorasTest**, **exceptions**) {

Mock<ICalculator> mock;

When(Method(mock,sqr)(3.0)).Return(9.0);

When(Method(mock,sqr)(4.0)).Return(16.0);

When(Method(mock,sqrt)(-7.0)).Throw(runtime\_error(""));

ICalculator& c = mock.get();

Pythagoras p(c);

ASSERT\_ANY\_THROW(p.a(3.0, 4.0));

}

# Mocking - Übungen

Dieses Kapitel enthält zwei Übungen zum Thema Mocking:

* In der ersten Übung geht's um einen AccountService, der auf ein AccountDAO angewiesen ist. Der AccountService enthält die Fachlogik der Konto-Anwendung, der AccountDAO enthält die Persistenzlogik. Der AccountService soll getestet werden – ohne das reale Datenbankzugriffe erfolgen. Das AccountDAO-Objekt muss also gemockt werden.
* In der zweiten Anwendung geht's um den Test eines Gruppenwechsel-Algorithmus, der in einer Klasse GroupChangeReader implementiert ist. Die Ausgaben des GroupChangerReaders erfolgen über ein Interface. Dieses Ausgabe-Interface wird gemockt werden.

(Da die Zeit knapp ist, wird es i.d.R. reichen müssen, eine der beiden Übungen durchzuführen. By the way: die erste Übung ist komplex, die zweite relativ einfach…)

## AcountService

Die im folgenden zu implementierende Kontoverwaltung besteht hauptsächlich aus zwei Komponenten. Eine Klasse AccountServiceImpl enthält die Fachlogik für das Verwalten von Konten. Die Persistenzlogik der Kontoverwaltung wird in einer zweiten Klasse implementiert werden, welche das Interface AccountDAO implementiert. Die Fachlogik-Klasse benutzt diese Persistenz-Klasse.

Die Fachlogik (AccountService) stellt z.B. eine Methode withdraw zur Verfügung. Dieser Methode wird die Nummer eines Kontos und der abzuhebende Betrag übergeben. Diese Methode wird dann zunächst auf die DAO-Methode findAccount aufrufen. Diese wird ein Objekt der Klasse Account zurückliefern (oder eine Exception werfen…). Dann wird geprüft, ob der Bestand des Kontos noch ausreicht, um den gewünschten Betrag auszahlen zu können. Wenn nicht, wird eine Exception geworfen. Ansonsten wird der gewünschte Betrag vom Bestand des Kontos abgezogen und dieses über den Aufruf der DAO-Methode updateAccount wieder persistiert werden. Zudem wird die DAO-Methode insertMovement aufgerufen, um ein Movement-Objekt zu persistieren (eine Objekt, welches die aktuelle Kontobewegung repräsentiert.)

Der (zu testende!) AccountService hängt also einem DAO-Objekt ab, das gemockt werden sollen: von einem Objekt, dessen Klasse das Interface IAccounDAO implementiert:

: AccountService

Account findAccount (nr)

createAccount (nr)

deposit (nr, amount)

withdraw (nr, amount)

transfer(fromNr, toNr,

amount)

*IAccountService*

: ????

Account findAccount (nr)

insertAccount (account)

updateAccount (account)

insertMovement (movement)

*IAccountDAO*

Die Klassen der persistenten Objekte (Account und Movement) sind bereits vorgegeben:

// Account.h

// ...

class **Account** {

public:

int **nr**;

int **balance**;

int **credit**;

**Account**(int nr, int balance, int credit)

: nr(nr), balance(balance), credit(credit) {

}

bool **operator ==** (const Account& other) const {

return nr == other.nr &&

balance == other.balance &&

credit == other.credit;

}

};

ostream& **operator <<** (ostream& out, const Account& account) {

return out << "Account [" <<

account.nr << ", " <<

account.balance << ", " <<

account.credit << "]";

}

// Movement.h

// ...

class **Movement** {

public:

int **accountNr**;

int **amount**;

**Movement**(int accountNr, int amount)

: accountNr(accountNr), amount(amount) {

}

bool **operator ==** (const Movement& other) const {

return accountNr == other.accountNr && amount == other.amount;

}

};

ostream& **operator <<** (ostream& out, const Movement& movement) {

return out << "Movement [" << movement.accountNr << ", " << movement.amount << "]";

}

Man beachte, dass beide Klassen den ==-Operator überladen. Dies wird für die folgende Verwendung der Klassen entscheidend sein...

Das Interface IAccountService spezifiziert die fachliche Logik der Kontoverwaltung:

// IAccountService.h

// ...

class **IAccountService** {

public:

virtual **~IAccountService**() { }

virtual void **createAccount**(int nr, int credit) = 0;

virtual Account **findAccount**(int nr) = 0;

virtual void **deposit**(int nr, int amount) = 0;

virtual void **withdraw**(int nr, int amount) = 0;

virtual void **transfer**(int fromNr, int toNr, int amount) = 0;

};

Das (zu mockende) Interface IAccountDAO spezifiziert die Persistenzlogik:

// IAccountDao.h

// ...

class **IAccountDao** {

public:

virtual **~IAccountDao**() { }

virtual void **insertAccount**(const Account& account) = 0;

virtual void **updateAccount**(const Account& account) = 0;

virtual Account **findAccount**(int nr) = 0;

virtual void **insertMovement**(const Movement& movement) = 0;

};

Das grobe Gerüst der AccountService-Klasse ist bereits vorgegeben:

// AccountService.h

// ...

class **AccountService** : public IAccountService {

private:

IAccountDao& **dao\_**;

public:

**AccountService**(IAccountDao& dao) :

dao\_(dao) {

}

void **createAccount**(int nr, int credit) {

dao\_.insertAccount(Account(nr, 0, credit));

}

Account **findAccount**(int nr) {

return dao\_.findAccount(nr);

}

void **deposit**(int nr, int amount) {

// TODO

}

void **withdraw**(int nr, int amount) {

// TODO

}

void **transfer**(int fromNr, int toNr, int amount) {

// TODO

}

};

Auch der Rahmen der Testklasse ist bereits vorgegeben:

// AccountServiceTest.h

// ...

using testing::Return;

class **MockedAccountDao**: public IAccountDao {

public:

MOCK\_METHOD1(**insertAccount**, void(const Account& account));

MOCK\_METHOD1(**updateAccount**, void(const Account& account));

MOCK\_METHOD1(**findAccount**, Account(int nr));

MOCK\_METHOD1(**insertMovement**, void(const Movement& movement));

};

using testing::Return;

class **AccountServiceTest** : public testing::Test {

protected:

MockedAccountDao\* **dao**;

IAccountService\* **service**;

public:

virtual void **SetUp**() override {

dao = new MockedAccountDao();

service = new AccountService(\*dao);

}

virtual void **TearDown**() override {

delete service;

delete dao;

}

};

TEST\_F(**AccountServiceTest**, **createAccount**) {

EXPECT\_CALL(\*dao, insertAccount(Account(4711, 0, 1000))).Times(1);

service->createAccount(4711, 1000);

}

TEST\_F(**AccountServiceTest**, **findAccount**) {

EXPECT\_CALL(\*dao, findAccount(4711))

.WillOnce(Return(Account(4711, 1000, 2000)));

Account account = service->findAccount(4711);

ASSERT\_EQ(Account(4711, 1000, 2000), account);

}

TEST\_F(**AccountServiceTest**, **deposit**) {

// TODO

}

TEST\_F(**AccountServiceTest**, **withdraw**) {

// TODO

}

TEST\_F(**AccountServiceTest**, **illegalWithdraw**) {

// TODO

}

TEST\_F(**AccountServiceTest**, **transfer**) {

// TODO

}

Gehen Sie bei der Implementierung in kleinen Schritten vor. Implementieren Sie jeweils zuerst eine Testmethode. Der folgende Testlauf sollte dann rot sein. Dann implementieren Sie die im Test aufgerufene AccountService-Methode. Dann sollte der folgende Testlauf grün sein. Wiederholen Sie diese Abfolge, bis alle Test-Methoden und alle AccountService-Methoden implementiert sind.

## GroupChangeReader

Ein Gruppenwechsel-Prozessor transformiert eine Eingabe, welche implizit gruppenförmig strukturiert ist, in eine Ausgabe, in welcher diese Gruppenstruktur explizit wird.

Ein recht abstrakte Definition…

Angenommen, die Eingabe beinhalt Order-Objekte:

// Order.h

// ...

class **Order** {

public:

const int **customerNr**;

const int **productNr**;

const int **amount**;

**Order**(int customerNr, int productNr, int amount)

: customerNr(customerNr), productNr(productNr), amount(amount) {

}

bool **operator ==** (const Order& other) const {

return customerNr == other.customerNr &&

productNr == other.productNr &&

amount == other.amount;

}

};

ostream& **operator <<** (ostream& out, const Order& order);

Angenommen, die Eingabe bestehe aus folgenden Objekten:

vector<Order> orders;

orders.push\_back(Order(1000, 100, 10));

orders.push\_back(Order(1000, 200, 10));

orders.push\_back(Order(2000, 100, 20));

orders.push\_back(Order(3000, 300, 30));

orders.push\_back(Order(3000, 400, 20));

orders.push\_back(Order(3000, 100, 10));

Diese Eingabe enthält drei Gruppen: die erste Gruppe enthält die Aufträge für den Kunden 1000, die zweite Gruppe enthält die Aufträge für den Kunden 2000 und die dritte Gruppe enthält die Aufträge für den Kunden 3000. Die erste Gruppe enthält 2 Objekte, die zweite Gruppe enthält ein einzige Objekt und die dritte Gruppe umfasst drei Objekte.

Aufgrund dieser Eingabe könnte z.B. folgende Druckausgabe erzeugt werden müssen:

Begin

GroupBegin 1000

Position Order[1000, 100, 10]

Position Order[1000, 200, 10]

GroupEnd 1000

GroupBegin 2000

Position Order[2000, 100, 20]

GroupEnd 2000

GroupBegin 3000

Position Order[2000, 300, 30]

Position Order[2000, 200, 20]

Position Order[2000, 100, 10]

GroupEnd 3000

End

Der zu testende Gruppenwechsel-Algorithmus wird die eigentliche Ausgabe an ein Objekt delegieren, dessen Klasse das folgende Interface implementiert:

// IProcessor.h

// ...

template <typename T, typename K>

class **IProcessor** {

public:

virtual **~IProcessor**() { }

virtual void **processBegin**() = 0;

virtual void **processGroupBegin**(const K& key) = 0;

virtual void **processPosition**(const T& obj) = 0;

virtual void **processGroupEnd**(const K& key) = 0;

virtual void **processEnd**() = 0;

};

T steht für den Typ der Objekte, die prozessiert werden (hier: Order); K steht für den Schlüssel (K bezeichnet den Typ des in T enthaltenen Gruppenschlüssels (hier: den Typ von customerNr, also int).

Eine konkrete Klasse, welche dieses Interfaces implementiert und die obige Ausgabe erzeugt, könnte wie folgt ausschauen (man beachte, dass T hier durch Order und K durch int ersetzt sind):

// PrintProcessor

// ...

class **PrintProcessor** : public IProcessor<Order, int> {

public:

void **processBegin**() {

cout << "Begin" << endl;

}

void **processGroupBegin**(const int& key) {

cout << "\tGroupBegin " << key << endl;

}

void **processPosition**(const Order& obj) {

cout << "\t\tPosition " << obj << endl;

}

void **processGroupEnd**(const int& key) {

cout << "\tGroupEnd " << key << endl;

}

void **processEnd**() {

cout << "End" << endl;

}

};

Hier nun die zu testende Klasse, welche den eigentlichen Gruppenwechsel-Algorithmus implementiert:

// GroupChangeReader.h

// ...

template<typename T, typename K>

class **GroupChangeReader** {

private:

IProcessor<T, K>& **processor\_**;

vector<T>& **input\_**;

unsigned int **index\_**;

public:

**GroupChangeReader**(IProcessor<T, K>& processor, vector<T>& input)

: processor\_(processor), input\_(input), index\_(0) {

}

virtual **~GroupChangeReader**() {

}

void **run**() {

T\* current = next();

processor\_.processBegin();

while (current != 0) {

K key = getKey(\*current);

processor\_.processGroupBegin(key);

while (current != 0 && getKey(\*current) == key) {

processor\_.processPosition(\*current);

current = next();

}

processor\_.processGroupEnd(key);

}

processor\_.processEnd();

}

protected:

virtual K **getKey**(const T& obj) = 0;

private:

T\* **next**() {

if (index\_ == input\_.size())

return 0;

return &input\_[index\_++];

}

};

Die Klasse hat zwei Typ-Parameter: T und K. T steht für den Typ der Eingabe-Objekte (hier also für Order), K steht für den "Schlüssel", der die Gruppenstruktur definiert (hier also für int – den Typ von customerNr).

Dem Konstruktor werden zwei Parameter übergeben: ein IProcessor und ein vector, der die Eingabeobjekte enthält.

Die run-Methode implementiert den eigentlichen Algorithmus. (Das genaue Studium der run-Methode sei dem Leser / der Leserin überlassen.)

Um festzustellen, wann ein Gruppenwechsel stattfindet, muss der Gruppenschlüssel ermittelt werden können. Hierzu dient die abstrakte getKey-Methode. Dieser wird ein Eingabeobjekt vom Typ T (hier: Order) übergeben; sie muss den Schlüssel dieses Objekts ermitteln (hier: die customerNr). Die GroupChangeReader-Klasse ist somit abstrakt.

Eine konkrete GroupChangeReader-Klasse kann nun wie folgt abgeleitet werden:

// OrderGroupChangeReader.h

// ...

class **OrderGroupChangeReader**: public GroupChangeReader<Order, int> {

public:

**OrderGroupChangeReader**(

IProcessor<Order, int>& processor,

vector<Order>& input)

: GroupChangeReader(processor, input) {

}

protected:

int **getKey**(const Order& obj) {

return obj.customerNr;

}

};

Man beachte: T ist hier durch den konkreten Typ Order ersetzt worden und K durch int.

Der obige PrintProcessor könnte nun wie folgt genutzt werden:

vector<Order> orders;

orders.push\_back(Order(1000, 100, 10));

orders.push\_back(Order(1000, 200, 10));

orders.push\_back(Order(2000, 100, 20));

orders.push\_back(Order(3000, 300, 30));

orders.push\_back(Order(3000, 400, 20));

orders.push\_back(Order(3000, 100, 10));

PrintProcessor p;

OrderGroupChangeReader r(p, orders);

r.run();

Dieser Aufruf der run-Methode würde genau diejenige Ausgabe produzieren, die am Anfang dieses Abschnitts skizziert wurde.

Zum Zwecke des Tests der GroupChangeReader-Klasse (also für den Test der run-Methode dieser Klasse) wird das Interface IProcessor gemockt werden.

Hier der bereits vorgegebene Rahmen der Testklasse:

// GroupChangeReaderTest.h

// ...

class **MockedProcessor**: public IProcessor<Order, int> {

public:

MOCK\_METHOD0(**processBegin**, void());

MOCK\_METHOD1(**processGroupBegin**, void(const int& key));

MOCK\_METHOD1(**processPosition**, void(const Order& obj));

MOCK\_METHOD1(**processGroupEnd**, void(const int& key));

MOCK\_METHOD0(**processEnd**, void());

};

TEST(**GroupChangeReader**, **emptyElement**) {

MockedProcessor p;

vector<Order> orders;

OrderGroupChangeReader r(p, orders);

EXPECT\_CALL(p, processBegin()).Times(1);

EXPECT\_CALL(p, processEnd()).Times(1);

r.run();

}

TEST(**GroupChangeReader**, **oneGroupOneElement**) {

MockedProcessor p;

vector<Order> orders;

orders.push\_back(Order(1000, 100, 10));

OrderGroupChangeReader r(p, orders);

// TODO

}

TEST(**GroupChangeReader**, **oneGroupMoreElement**) {

MockedProcessor p;

vector<Order> orders;

orders.push\_back(Order(1000, 100, 10));

orders.push\_back(Order(1000, 200, 20));

OrderGroupChangeReader r(p, orders);

// TODO

}

TEST(**GroupChangeReader**, **twoGroups**) {

MockedProcessor p;

vector<Order> orders;

orders.push\_back(Order(1000, 100, 10));

orders.push\_back(Order(1000, 200, 20));

orders.push\_back(Order(2000, 300, 30));

OrderGroupChangeReader r(p, orders);

// TODO

}

# Test Driven Development: Parser

Eine String-förmige Eingabe enthält numerische Ausdrücke. Die numerischen Werte solcher Ausdrücke sollen berechnet werden.

Sei etwa folgende Eingabe gegeben:

"10 \* (200 – 150) + 3 \* (5 - (3 - 1))"

Der Wert dieses Ausdrucks ist 59 (natürlich unter der Berücksichtigung der Vorrang-Regel: Punkt- geht vor Strich-Rechnung).

Zunächst geht es darum, die Folge der in der Eingabe enthaltenen Zeichen in eine Folge von "Symbolen" zu transformieren – dies ist die Aufgabe des Scanners. Diese vom Scanner erzeugte Symbolfolge wird dann vom eigentlichen Parser weiterverarbeitet werden.

Die obige Zeichenfolge z.B. enthält folgende Symbole

10

\*

(

200

-

150

)

+

3

etc….

Der erste Test und der erste Implementierungsschritt des Scanners ist bereits vorgegeben:

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **emptyInput**) {

stringstream stream("");

Scanner scanner(stream);

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

Wir spezifizieren hier, dass dem Konstruktor der Scanner-Klasse die zu scannende Zeichenfolge übergeben wird. Im obigen Test wird die Eingabe in Form eines stringstreams übergeben. Der Konstruktor wird aber mit jedem istream zufrieden sein – so dass ggf. auch ein ifstream übergeben werden kann. Im obigen Falle wird dem Scanner eine leere Eingabe übergeben.

Mittels der next-Methode wird der Scanner jeweils zum nächsten Symbol der Eingabe weitergetrieben. Der Aufruf dieser Methode verändert also den Zustand des Scanners. Die next-Methode liefert void.

Mittels der current-Methode kann das aktuelle Symbol, zu dem der Scanner mittels des letzten next-Aufrufs weitergetrieben wurde, ermittelt werden. Der Aufruf dieser Methode ändert nichts am Zustand des Scanners – sie wird also const sein können.

Erkennt der Scanner das Ende der Eingabe, wird ein Spezial-Symbol zurückgeliefert: END. Dies wird eine Konstant eines Aufzählungstyps sein, welcher später noch weitere Konstanten enthalten wird: PLUS, MINUS etc. Bei einer leeren Eingabe muss natürlich bereits das erste Symbol, das der Scanner erkennt, END sein.

Hier die minimale Scanner-Klasse, die sowohl den Compiler zufriedenstellt als auch den Test erfolgreich bestehen wird:

// Scanner.h

// ...

typedef enum { **END** } **Symbol**;

class **Scanner** {

public:

**Scanner**(istream& in) {

}

void **next**() {

}

Symbol **current**() {

return END;

}

};

Der Aufzählungstyp Symbol enthält zunächst nur eine einzige Konstante: END. Dieser Typ wird natürlich später erweitert werden. Dem Konstruktor wird eine istream-Referenz übergeben – er muss aber noch nicht weiter implementiert werden. Auch die next-Methode ist noch leer. Die current-Methode schließlich liefert END.

Damit ist der Anfang getan.

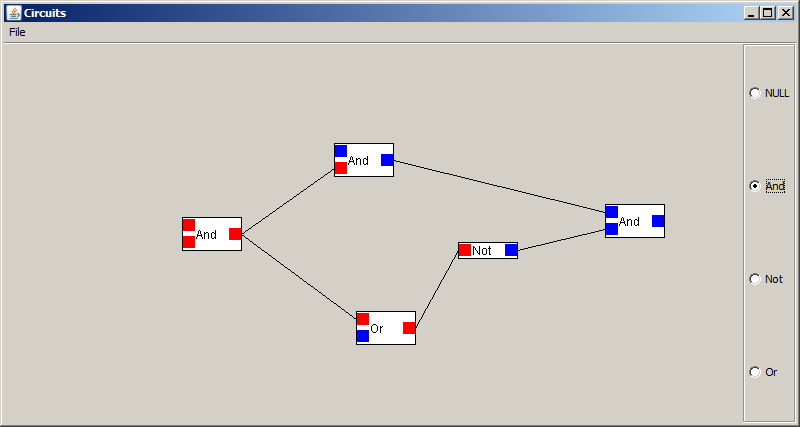
Hier einige Hinweise für mögliche Entwicklungsschritte:

* Der Scanner soll folgende Eingabe korrekt scannen: "+-/\*"
* Der Scanner soll Whitespaces überlesen: " + - /\* "
* Der Scanner soll ganze Zahlen erkennen: " 123 + 44 \*/+ 77 "
* Der Scanner soll Gleitkommazahlen erkennen: " 12.3 + 44 \*/+ 7.7 "
* Der Parser sollte eine Zahl parsen können: " 12.3 "
* Der Parser sollte eine multiplikative Verknüpfung parsen können: " 2 \* 30/10"
* Der Parser sollte eine additive Verknüpfung parsen können (welche ihrerseits multiplikative Verknüpfungen enthält) : " 2 \* 30 / 10 + 20 – 10 \* 5 "
* Der Parser sollte geklammerte Expressions parsen können: " (1 + 2) \* 3 "
* Der Parser sollte statt eines numerischen Ergebnisses eine Baum von Expression-Objekten liefern, welche den Ausdruck repräsentieren. Solche Expressions sollten berechnet werden können.

Viel Spaß && Erfolg bei der Entwicklung!!

# Test Driven Development: logische Schaltungen

Es geht um die Entwicklung eines Systems, welches den Entwurf beliebig komplexer logischer Schaltungen ermöglicht.



Hier soll allerdings keine(!) grafische Oberfläche entwickelt werden – sondern "nur" der "Kern", der sich hinter einer solchen GUI verbirgt.

Eine logisiche Schaltung (ein Circuit) kann beliebig viele Eingänge und Ausgänge enthalten. And- und Or-Schaltungen haben jeweils zwei Eingänge (Inputs) und einen Ausgang (Output); eine Not-Schaltung hat einen Eingang und einen Ausgang; ein Halbaddierer hätte drei Eingänge und zwei Ausgänge.

Welche wichtigen Use-Cases können unterschieden werden?

* Eine Schaltung muss erzeugt werden können. Nach der Erzeugung einer And- und einer Not-Schalung sind alle Eingänge und der Ausgang jeweils "blau" (also false). Nach der Erzeugung einer Not-Schaltung ist auch deren Eingang blau – der Ausgang aber "rot" (also true).
* Sofern ein Eingang nicht mit einem Ausgang einer anderen Schaltung verbunden ist – sofern er also "frei" ist - , kann dieser Eingang "getogglet" werden: aus "rot" wird "blau" und umgekehrt. Das hat zur Folge, dass möglicherweise auch der Ausgang (die Ausgänge) dieser Schaltung ihren Zustand ändern müssen – und auch die Eingänge, die von diesen Augängen "versorgt" werden etc.
* Ein Ausgang kann mit dem Eingang einer anderen Schaltung verbunden werden – aber nur dann, wenn dieser Eingang "frei" ist (anschließend ist dieser Eingang dann nicht mehr "frei"). Ein Ausgang kann mit mehreren Eingängen verbunden werden (welche er dann "versorgt") – aber ein Eingang kann nur von einem einzigen Ausgang versorgt werden.
* Man möchte den Typ einer Schaltung ändern können – aus einer And-Schaltung z.B. eine Or-Schaltung machen. (Das setzt voraus, dass die neue Schaltung dieselbe Anzahl Eingänge und Ausgänge hat wie die alte Schaltung.)

An diesen Use-Cases könnte sich die Entwicklung orientieren.

Welche Klassen werden benötigt? Im obigen Text wurden bereits folgende Begriffe genannt: Circuit, Input, Output. Vielleicht kann sich die Entwicklung an diesen "Substantiven" orientieren.

Hier eine mögliche Schrittfolge:

* Wir sollten Schaltungen nach der Anzahl ihrer Eingänge resp. ihrer Ausgänge fragen können.
* Schaltungen sollten nach ihrer Erzeugung einen korrekten Zustand haben.
* Die Eingänge von Schaltungen sollten getogglet werden können.
* Ausgänge sollten mit Eingängen verbunden werden können (sofern die Eingänge frei sind).
* Eingänge sollten nicht getogglet werden können, wenn sie bereits von einem Ausgang versorgt werden.
* Sind Schaltungen bereits mit anderen Schaltungen verbunden, so hat das Togglen möglicherweise eine "weitreichende" Wirkung.
* Angenommen, eine And-Schaltung ist bereits mit einer Vielzahl anderer Schaltungen verbunden. Dann bemerkt der Benutzer, dass die And-Schaltung eigentlich eine Or-Schaltung sein sollte. Wie kann der Typ einer Schaltung geändert werden?

Hier ein möglicher Einstieg. Zunächst die Testklasse:

// CircuitTest.cpp

// ...

class **CircuitTest** : public testing::Test {

protected:

Circuit\* **c1**;

Circuit\* **c2**;

Circuit\* **c3**;

public:

virtual void **SetUp**() override {

c1 = new AndCircuit();

c2 = new OrCircuit();

c3 = new NotCircuit();

}

virtual void **TearDown**() override {

delete c1;

delete c2;

delete c3;

}

};

TEST\_F(**CircuitTest**, **start**) {

EXPECT\_STREQ("and", c1->name());

EXPECT\_EQ(2, c1->inputCount());

EXPECT\_EQ(1, c1->outputCount());

EXPECT\_STREQ("or", c2->name());

EXPECT\_EQ(2, c2->inputCount());

EXPECT\_EQ(1, c2->outputCount());

EXPECT\_STREQ("not", c3->name());

EXPECT\_EQ(1, c3->inputCount());

EXPECT\_EQ(1, c3->outputCount());

}

In allen Testmethoden können drei Schaltungen verwendet werden, welche jeweils beim Aufruf der SetUp-Methode erzeugt werden.

Anhand der SetUp-Methode wird deutlich, dass es drei Klassen geben muss: AndCircuit, OrCircuit und NotCircuit. Alle Schaltungsklassen müssen von Circuit abgeleitet sein (denn sonst könnten wir mit einem Circuit\*-Zeiger nicht auf diese Schaltungsobjekte verweisen).

Damit der Compiler den start-Test übersetzt, müssen auf jedes Circuit-Objekt die Methoden name(), inputCount() und outputCount() aufrufbar sein. Diese müssen bereits in der Basisklasse Circuit spezifiziert sein – denn sonst könnten sich über Circuit\*-Zeiger nicht aufgerufen werden. Ihre Implementierung wird in den abgeleiteten Klassen erfolgen.

Hier eine minimale Implementierung, die sowohl den Compiler zufriedenstellt als auch den obigen Test besteht (die Basisklasse ist in Circuit.h implementiert, die abgeleiteten Klassen in Circuits.h (Plural)):

// Circuit.h

class **Circuit** {

public:

virtual **~Circuit**() { }

virtual const char\* **name**() = 0;

virtual unsigned int **inputCount**() = 0;

virtual unsigned int **outputCount**() = 0;

};

// Circuits.h

// ...

class **AndCircuit** : public Circuit {

public:

virtual const char\* **name**() {

return "and";

}

virtual unsigned int **inputCount**() {

return 2;

}

virtual unsigned int **outputCount**() {

return 1;

}

};

class **OrCircuit** : public Circuit {

public:

virtual const char\* **name**() {

return "or";

}

virtual unsigned int **inputCount**() {

return 2;

}

virtual unsigned int **outputCount**() {

return 1;

}

};

class **NotCircuit** : public Circuit {

public:

virtual const char\* **name**() {

return "not";

}

virtual unsigned int **inputCount**() {

return 1;

}

virtual unsigned int **outputCount**() {

return 1;

}

};

# Refactoring – Gruppenwechsel

Im folgenden wird ein "katastrophales" Programm refaktoriert werden.

Es handelt sich bei dem Programm um einen einfachen Gruppenwechsel. Eine Eingabedatei enthält nach Gruppen sortierte Auftrags-Sätze. Diese in der Eingabe implizite Gruppenstruktur wird transformiert in eine Druckliste, welche die Gruppenstruktur explizit ausweist.

Die Eingabe-Dateien:

customers.txt enthält für jeden Kunden dessen Nummer und dessen Namen:

1000,Nowak

2000,Rueschenpoehler

3000,Montjoe

products.txt enthält für jedes Produkt dessen Nummer, Namen und dessen Einzelpreis:

100,Jever,11

200,Veltins,12

300,Krombacher,13

400,Bitburger,14

orders.txt enthält die Aufträge. Jeder Auftrag hat eine Kundennummer, eine Produktnummer und eine Menge. Die Aufträge eines Kunden sind jeweils zu einer Gruppe zusammengefasst.

1000;100;10

1000;200;20

2000;200;20

2000;300;30

2000;400;40

3000;100;10

Das folgende Programm erstellt eine Liste aller Aufträge. Sie hat einen Kopf und einen Fuß (im Fuß wird der Gesamtwert aller Aufträge ausgegeben). Für jede Gruppe von Aufträgen wird ein Gruppenkopf (mit der Nummer und dem Namen des Kunden) und ein Gruppenfuß ausgegeben (mit der Summe aller Auftragswerte dieses Kunden). Und für jede Auftragsposition wird eine Zeile mit Produkt-Nummer, Produkt-Namen, Einzelpreis, Menge und Positionswert ausgegeben.

Hier die aufgrund der obigen Eingaben produzierte Ausgabe (result.txt):

Orders

1000 Nowak

100 Jever 10 11 110

200 Veltins 20 12 240

---------

350

2000 Rueschenpoehler

200 Veltins 20 12 240

300 Krombacher 30 13 390

400 Bitburger 40 14 560

---------

1190

3000 Montjoe

100 Jever 10 11 110

---------

110

Total: 1650

Das Programm benutzt die im Testing-Kapitel vorgestellte CSVReader-Klasse:

// CSVReader.h

// ...

class **CSVReader** {

private:

// ...

public:

**CSVReader**(istream& in, char seperator, unsigned int tokenCount) :

in\_(in), seperator\_(seperator), tokenCount\_(tokenCount);

string\* **readLine**(); // Result (new string[]) must be deleted

// ...

};

Und hier schließlich das "vorgefundene" Programm:

// main.cpp

// ...

string\* **getCustomer**(string number, vector<string\*> customers) {

for (unsigned int i = 0; i < customers.size(); i++) {

if (customers[i][0] == number)

return customers[i];

}

throw runtime\_error("customer not found");

}

string\* **getProduct**(string number, vector<string\*> products) {

for (unsigned int i = 0; i < products.size(); i++) {

if (products[i][0] == number)

return products[i];

}

throw runtime\_error("product not found");

}

void **demo**() {

vector<string\*> customers = vector<string\*>();

ifstream customersStream("customers.txt");

CSVReader customersReader(customersStream, ',', 2);

string\* customerTokens = customersReader.readLine();

while (customerTokens != 0) {

customers.push\_back(customerTokens);

customerTokens = customersReader.readLine();

}

customersStream.close();

vector<string\*> products = vector<string\*>();

ifstream productsStream("products.txt");

CSVReader productsReader(productsStream, ',', 3);

string\* productTokens = productsReader.readLine();

while (productTokens != 0) {

products.push\_back(productTokens);

productTokens = productsReader.readLine();

}

productsStream.close();

ofstream out("result.txt");

out << "Orders" << endl << endl;

ifstream ordersStream("orders.txt");

CSVReader ordersReader(ordersStream, ';', 3);

string\* order = ordersReader.readLine();

int sum = 0;

while (order != 0) {

string customerNumber = order[0];

string\* customer = getCustomer(customerNumber, customers);

out << customerNumber << " " << customer[1] << endl;

int groupSum = 0;

while (order != 0 && order[0] == customerNumber) {

string productNumber = order[1];

string\* product = getProduct(productNumber, products);

string productName = product[1];

int productPrice = atoi(product[2].c\_str());

int positionPrice =

productPrice \* atoi(order[2].c\_str());

out << "\t" <<

productNumber << " " <<

productName << " " +

order[2] << " " <<

productPrice << " " <<

positionPrice << endl;

groupSum += positionPrice;

order = ordersReader.readLine();

}

sum += groupSum;

out << "\t---------" << endl;

out << "\t" << groupSum << endl;

out << endl;

}

out << "Total:\t" << sum << endl;

ordersStream.close();

out.close();

cout << "Done. See results.txt" << endl;

}

int **main**(int argc, char\*\* argv) {

cout << "Refactoring-Start" << endl;

try {

demo();

}

catch(const runtime\_error& e) {

cout << "Exception: " << e.what() << endl;

}

//testing::InitGoogleTest(&argc, argv);

//return RUN\_ALL\_TESTS();

}

# DbAccessTester

DbAccessTester ist ein kleines selbstentwickeltes Tool zum Testen Datenbank-basierter Anwendungen.

Bekanntlich kann die DAO-Zugriffsschicht einer Anwendung gemockt werden – so dass die Service-Schicht unabhängig von einer realen Datenbank getestet werden kann. Bleibt aber immer noch die DAO-Schicht…

Im folgenden wird eine Anwendung getestet werden, welche auf die Schichtenstruktur verzichtet – und sowohl die Fachlogik als auch die Persistenzlogik in ein und derselben Klasse implementiert. In diesem Falle benötigt man natürlich auch zum Testen der Fachlogik eine reale Datenbank.

Wir bauen einen AccountService, welche folgende Value-Klasse benutzt:

// Account.h

// ...

struct **Account** {

unsigned int **nr**;

int **balance**;

unsigned int **credit**;

string **customer**;

**Account**(unsigned int nr, int balance, unsigned int credit,

const string& customer) :

nr(nr), balance(balance), credit(credit), customer(customer) {

}

bool **operator ==** (const Account& other) const {

return nr == other.nr && balance == other.balance &&

credit == other.credit && customer == other.customer;

}

};

Man beachte den equals-Operator!

Der AccountService, der im folgenden vorgestellt wird, benutzt als Datenbank SQLite, auf welche mittels einer C++-Schnittstelle zugegriffen wird.

Siehe hierzu:

<http://www.codeproject.com/Articles/6343/CppSQLite-C-Wrapper-for-SQLite>

Als Datenbank kann auch eine beliebige andere Datenbank verwendet werden (für jeden Datenbank-Typ verlangt das DbAccessTester-Tool allerdings einen kleinen "Adapter" – siehe hierzu weiter unten).

Die benutzte Datenbank enthält folgende Tabelle:

create table **account**(

**nr** integer,

**balance** integer,

**credit** integer,

**customer** varchar(64),

primary key(nr)

)

Hier nun der AccountService, der im folgenden Methode für Methode vorgestellt wird:

// AccountService.h

#include <string>

#include "Account.h"

#include "CppSQLite3.h"

using namespace std;

class **AccountService** {

private:

CppSQLite3DB\* **db\_**;

public:

**AccountService**(CppSQLite3DB\* db) :

db\_(db) {

}

Dem Konstruktor von AccountService wird ein Pointer auf SQLite übergeben, der in der Instanzvariablen db\_ gespeichert wird.

void **createAccount**(unsigned int nr, unsigned int credit,

const string& customer) {

try {

CppSQLite3Statement stmt = db\_->compileStatement(

"insert into account values(?, 0, ?, ?)");

stmt.bind(1, (int) nr);

stmt.bind(2, (int) credit);

stmt.bind(3, customer.c\_str());

stmt.execDML();

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(e.errorMessage());

}

}

Mittels createAccount kann eine neue account-Zeile angelegt werden. Damit eine Anwendung nicht allzu sehr an SQLite gebunden ist, wird eine evtl. von SQLite geworfene CppSQLite3Exception wird in einen runtime\_error konvertiert (dies gilt auch für alle weiteren Methoden).

SharedPtr<Account> **findAccount**(unsigned int nr) {

try {

CppSQLite3Statement stmt = db\_->compileStatement(

"select balance, credit, customer "

"from account where nr = ?");

stmt.bind(1, (int) nr);

CppSQLite3Query query = stmt.execQuery();

if (query.eof())

return SharedPtr<Account>();

int balance = query.getIntField(0, 0);

int credit = query.getIntField(1, 0);

string customer = query.getStringField(2, 0);

query.nextRow();

if (!query.eof())

throw runtime\_error("result is not unique");

return SharedPtr<Account>(

new Account(nr, balance, credit, customer));

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(e.errorMessage());

}

}

Mittels findAccount kann aufgrund einer Kontonummer die entsprechende account-Zeile gefunden werden, deren Daten dann in ein dynamisch erzeugtes Account-Objekt übertragen werden. Der Account-Pointer wird in Form eines SharedPtr zurückgeliefert (SharedPtr ist eine Klasse, die weiter unten vorgestellt wird). Sofern zur übergebenen Kontonummer kein Konto existiert, wird ein "leerer" SharedPtr geliefert. Sofern der SQL-Select mehr als eine einzige Zeile liefert, wird ein runtime\_error geworfen (dann wird etwas oberfaul sein – denn die Kontonummer ist der Primärschlüssel der Tabelle).

void **deposit**(unsigned int nr, unsigned int amount) {

try {

SharedPtr<Account> account = findAccount(nr);

if (!account)

throw runtime\_error("account not found");

account->balance += amount;

CppSQLite3Statement stmt = db\_->compileStatement(

"update account set balance = ? where nr = ?");

stmt.bind(1, account->balance);

stmt.bind(2, (int) nr);

stmt.execDML();

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(e.errorMessage());

}

}

deposit ermöglicht eine Einzahlung. Der Methode wird die Nummer des Kontos und der einzuzahlende Betrag übergeben. Sofern zur übergebenen Kontonummer kein Konto existiert, wird ein runtime\_error geworfen.

void **withdraw**(unsigned int nr, unsigned int amount) {

try {

SharedPtr<Account> account = findAccount(nr);

if (!account)

throw runtime\_error("account not found");

if (amount > account->balance + account->credit)

throw runtime\_error("amount too large");

account->balance -= amount;

CppSQLite3Statement stmt = db\_->compileStatement(

"update account set balance = ? where nr = ?");

stmt.bind(1, account->balance);

stmt.bind(2, (int) nr);

stmt.execDML();

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(e.errorMessage());

}

}

withdraw ist invers zu deposit – ermöglicht also eine Auszahlung. Man beachte, dass die Methode einen runtime\_error wirft, wenn der gewünschte Auszahlungsbetrag nicht verfügbar ist.

void **transfer**(unsigned int fromNr, unsigned int toNr,

unsigned int amount) {

// the sequence is silly, but correct

// (in the context of an transaction)...

deposit(toNr, amount);

withdraw(fromNr, amount);

}

};

transfer schließlich ermöglicht eine Überweisung. Man beachte den Kommentar: Natürlich werden alle AccountService-Methoden im Kontext einer Transaktion aufgerufen werden müssen, deren Grenzen von der Anwendung bestimmt werden. Die Transaktionssteuerung ist nicht die Angelegenheit des AccountServices, sondern obliegt dem Aufrufer dieses Services.

Wie kann die AccountService-Klasse (oder ähnliche Klassen) nun getestet werden?

Vor dem Aufruf einer jeden Test-Methode muss dafür gesorgt werden, dass sich die Datenbank immer in einem bestimmten Initialzustand befinden. In Falle des AccountServices sollte eine leere account-Tabelle existieren. Die Datenbank muss also vor jeder Testmethode neu vorbereitet werden – wir bezeichnen diesen Schritt als "prepare"-Schritt.

In jeder Testmethode können dann auf den AccountService bestimmte Operationen aufgerufen werden. Diese Aufrufe verändern i.d.R. den Zustand der Datenbank. Wir bezeichnen diesen Schritt als "play" – der AccountService "spielt" mit der Datenbank.

Nach dieser play-Phase befindet sich die Datenbank in einem bestimmten Zustand. Angenommen, wir "fotografieren" diesen Zustand (wir lesen alle Tabellen der Datenbank aus und speichern die Werte der Tabellen in einer geeigneten programm-internen Datenstruktur). Wir nennen diese Fotografie "actual database".

Was erwarten wir? Wir erwarten, das dieses Foto das erwartete Bild enthält. Wie können wir das erwartete Bild spezifizieren? Wir präparieren die Datenbank erneut (s.o.). Dann setzen wir "per Hand" einige SQL-Statements ab, welche die Datenbank wieder verändern. Wir bezeichnen diesen Schritt als "replay".

Wir fotografieren die Datenbank erneut - und bezeichnen diese zweite Fotografie dann als "expected database". Wir besitzen dann zwei Fotos ("actual database" und "expected database").

Das einzige, was wir dann noch benötigen, ist ein Werkzeug, welches diese beiden Fotos vergleicht. Sie die beiden Fotos gleich, so entspricht das Resultat der Ausführung der AccountService-Methoden unseren Erwartungen (und der Test sollte grün liefern); sind sie ungleich, so sollte der Test scheitern. Dann allerdings sollten die Unterschiede der beiden Fotos auch exakt diagnostiziert werden.

Diese Überlegungen können direkt in den folgenden Test "übersetzt" werden:

// AccountServiceTest.cpp

#include <gtest/gtest.h>

#include "DbAccessTester.h"

#include "SQLiteAdapter.h"

#include "AccountService.h"

using namespace std;

class **AccountServiceTest**: public testing::Test {

protected:

DbAccessTester\* **tester**;

IDatabaseAdapter\* **adapter**;

CppSQLite3DB\* **db**;

AccountService\* **accountService**;

const char\* **filename** = "demo.db";

public:

virtual void **SetUp**() {

db = new CppSQLite3DB;

accountService = new AccountService(db);

adapter = new SQLiteAdapter(\*db, filename);

Array<string>::Builder prepareStrings;

prepareStrings

<< "create table account("

" nr integer, "

" balance integer, "

" credit integer, "

" customer varchar(64), "

" primary key(nr)"

");";

prepareStrings

<< "create table movement("

" nr integer, "

" movementdate date, "

" amount integer, "

" primary key(nr, movementdate)"

");";

tester = new DbAccessTester(\*adapter, prepareStrings);

tester->open();

}

virtual void **TearDown**() {

tester->close();

delete tester;

delete adapter;

delete accountService;

delete db;

}

};

In der SetUp-Methode werden vier Pointer initialisiert:

Ein Pointer auf die SQLite-Datenbank

Ein Pointer auf ein AccountService-Objekt

Ein Pointer auf einen SQLite-Adapter

Ein Pointer auf einen DbAccessTester

Dem AccountService und dem SQLite-Adapter wird der SQLite-Pointer übergeben. Sowohl der AccountService als auch der SQL-Adapter können also SQLite-Datenbank-Operationen ausführen.

Dem DbAccessTester wird als erster Parameter der SQLite-Adapter übergeben (man könnte dem Konstruktor auch einen Adapter für eine andere Datenbank übergeben – die Adapter sind spezifiziert über ein Interface namens IDatabaseAdapter). Als zweiter Parameter wird eine Folge von SQL-Strings übergeben: die prepareStrings (diese Strings werden in Form eines Arrays übergeben – zur Array-Klasse siehe weiter unten). Der DbAccessTester wird also jederzeit diese prepareStrings ausführen können (er wird sie jeweils zweimal ausführen: vor der play- und vor der replay-Phase). Schließlich wird die Datenbank geöffnet (Herstellung der Connection).

(Nur spaßeshalber wird neben der account-Tabelle eine zweite Tabelle angelegt: movement. Diese Tabelle wird in der vorliegenden Lösung nicht weiter benutzt.)

In TearDown wird die Datenbank geschlossen und die allokierten Ressourcen werden wieder freigegeben.

Der erste Test testet die createAccount-Methode:

TEST\_F(**AccountServiceTest**, **createAccount**) {

tester->play();

accountService->createAccount(4711, 1000, "Nowak");

tester->replay();

tester->execute(

"insert into account values(4711, 0, 1000, 'Nowak')");

bool ok = tester->compare();

EXPECT\_TRUE(ok);

if (!ok)

tester->printDiffs();

}

Zunächst wird die play-Phase gestartet – mittels der DbAccessTester-Methode play. Der AccountService wird beauftragt, ein neues Konto anzulegen. Damit ist auch die play-Phase bereits beendet.

Dann wird mittels des replay-Aufrufs die replay-Phase gestartet. Beim replay-Aufruf wird der aktuelle Zustand der Datenbank fotografiert ("actual database") und es werden erneut die perpareStrings ausgeführt. Dann wird mittels des execute-Methode des DbAccessTesters ein SQL-Insert ausgeführt. Wir erwarten, dass der createAccount-Aufruf genau diejenige Wirkung auf die Datenbank hat, die wir hier direkt mittels eines SQL-insert-Befehls erzeugen.

Schließlich wird compare aufgerufen. compare fotografiert die Datenbank erneut ("expected database") – und vergleicht dann die beiden Fotos. Sind die Fotos gleich, wird true geliefert, ansonsten false. Wir erwarten natürlich true (EXPECT\_TRUE(true)). Sofern der Test nicht erfolgreich war, sollten die Unterschiede der beiden Fotos ausgegeben werden. Dies geschieht mittels der DbAccessTester-Methode printDiffs.

Der Test wird "grün" ergeben.

Aber einmal angenommen, die createAccount-Methode der AccountService-Klasse wäre wie folgt (nämlich fehlerhaft) implementiert:

void **createAccount**(unsigned int nr, unsigned int credit,

const string& customer) {

try {

CppSQLite3Statement stmt = db\_->compileStatement(

"insert into account values(?, 0, ?, ?)");

stmt.bind(1, (int) credit);

stmt.bind(2, (int) nr);

stmt.bind(3, customer.c\_str());

stmt.execDML();

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(e.errorMessage());

}

}

Hier liegt offensichtlich ein Fehler in der Parameter-Ersetzung von: nr und credit sind vertauscht.

Der Test würde dann folgendes Resultat liefern:

[ RUN ] AccountServiceTest.createAccount

..\src\AccountServiceTest.cpp:49: Failure

Value of: ok

Actual: false

Expected: true

Table account

Expected, but not actual

Row { { 4711, 0, 1000, Nowak } }

Actual, but not expected

Row { { 1000, 0, 4711, Nowak } }

[ FAILED ] AccountServiceTest.createAccount

Der Fehler in der AccountService-Implementierung wäre somit erkannt worden.

Der zweite Test testet die deposit-Methode:

TEST\_F(**AccountServiceTest**, **deposit**) {

tester->play();

accountService->createAccount(4711, 1000, "Nowak");

accountService->deposit(4711, 4000);

tester->replay();

tester->execute(

"insert into account values(4711, 0, 1000, 'Nowak')");

tester->execute(

"update account set balance = 4000 where nr = 4711");

bool ok = tester->compare();

EXPECT\_TRUE(ok);

if (!ok)

tester->printDiffs();

}

Die Ausführung der beiden AccountService-Methoden createAccount und deposit sollte dasselbe Resultat erzeugen wie die Ausführung der beiden SQL-Befehle. Der Test wird "grün" ergeben.

Denselben Test hätten wir auch kürzer formulieren können:

TEST\_F(**AccountServiceTest**, **deposit2**) {

tester->play();

accountService->createAccount(4711, 1000, "Nowak");

accountService->deposit(4711, 4000);

tester->replay();

tester->execute(

"insert into account values(4711, 4000, 1000, 'Nowak')");

bool ok = tester->compare();

EXPECT\_TRUE(ok);

if (!ok)

tester->printDiffs();

}

Wir wollen ja nicht testen, ob die AccountService-Methoden exakt dieselben SQL-Anweisungen ausgeführt haben, die wir in der replay-Phase formulieren – es geht einzig und allein um das endgültige Resultat (um dass, was hinten herauskommt (H.Kohl)).

Trotzdem ist der erste deposit-Test wahrscheinlich lesbarer…

Wir wollen die transfer-Methode testen:

TEST\_F(**AccountServiceTest**, **transfer**) {

tester->play();

accountService->createAccount(4711, 1000, "Nowak");

accountService->createAccount(4712, 2000, "Ruepoe");

accountService->deposit(4711, 4000);

accountService->transfer(4711, 4712, 2500);

tester->replay();

tester->execute(

"insert into account values(4711, 0, 1000, 'Nowak')");

tester->execute(

"insert into account values(4712, 0, 2000, 'Ruepoe')");

tester->execute(

"update account set balance = 1500 where nr = 4711");

tester->execute(

"update account set balance = 2500 where nr = 4712");

bool ok = tester->compare();

EXPECT\_TRUE(ok);

if (!ok)

tester->printDiffs();

}

Hierzu ist nun nichts weiter mehr zu sagen.

Angenommen, die transfer-Methode wird mit einem "falschen" Überweisungsbetrag aufgerufen. Dann müsste eine Exception geliefert werden. Im Falle einer Exception würden wird die Transaktion zurücksetzen (die wir natürlich zuvor explizit starten müssen):

TEST\_F(**AccountServiceTest**, **transferRollback**) {

tester->play();

accountService->createAccount(4711, 1000, "Nowak");

accountService->createAccount(4712, 2000, "Ruepoe");

accountService->deposit(4711, 4000);

tester->beginTransaction();

try {

accountService->transfer(4711, 4712, 6000);

tester->commitTransaction();

FAIL();

}

catch (...) {

SUCCEED();

tester->rollbackTransaction();

}

tester->replay();

tester->execute(

"insert into account values(4711, 0, 1000, 'Nowak')");

tester->execute(

"insert into account values(4712, 0, 2000, 'Ruepoe')");

tester->execute(

"update account set balance = 4000 where nr = 4711");

bool ok = tester->compare();

EXPECT\_TRUE(ok);

if (!ok)

tester->printDiffs();

}

Der Test liefert grün. Man sieht: deposit hat funktioniert, die transfer-Transaktion ist zurückgesetzt worden (man beachte hier noch einmal die "verrückte" Implementierung von transfer!).

Natürlich kann der DbAccessTester auch dann genutzt werden, wenn einfache Lesezugriffe getestet werden sollen. Dann benötigt man weder eine play- noch eine replay-Phase – sondern nur den Aufruf von prepare:

TEST\_F(**AccountServiceTest**, **withoutPlayAndReplay**) {

vector<string> v;

tester->prepare();

accountService->createAccount(4711, 1000, "Nowak");

SharedPtr<Account> pAccount = accountService->findAccount(4711);

ASSERT\_TRUE(pAccount); // ASSERT – not EXPECT

Account account = \*pAccount;

EXPECT\_EQ(Account(4711, 0, 1000, "Nowak"), account);

}

Abschließend einige Hinweise zur Implementierung der DbAccessTester-Klasse.

Die Implementierung benutzt exzessiv eine Klasse SharedPtr, die das "shared pointer"-Konzept implementiert. Diese Klasse ist entweder von der C++-11-Klasse std::shared\_ptr abgeleitet (wenn CPP11 definiert ist – siehe SharedPtr.h) oder aber from scratch implementiert ist. Man beachte, dass zudem eine Klasse existiert, die es ermöglicht, einen Pointer automatisch in einen SharedPtr zu konvertieren (SharedPtrConvertable – ebenfalls in SharedPtr.h). Auch von dieser automatischen Konvertierung wird exzessiv Gebrauch gemacht (siehe die Klassen in Database.h/.cpp, die allesamt u.a. von SharedPtrConvertable abgeleitet sind).

Weiterhin wird durchgängig eine Template-Klasse Array verwendet. Ein Array<T> repräsentiert einen Array von T's, der immutable ist. Zum Konstruktion eines Array<T>-Objekts wird ein Array<T>::Bilder genutzt. Solche Arrays werden u.a. zur internen Repräsentation der Datenbank-Daten genutzt. Siehe Array.h.

Die Daten und die Metadaten der Datenbank werden in Objekten der folgenden Klassen gehalten (s. Database.h/.cpp):

ColumnInfo

TableInfo

DatabaseInfo

Row

Table

Database

Objekte der …Info-Klassen repräsentieren die Metadaten der Datenbank, Objekte der letzten drei Klassen die eigentlichen Daten. Objekte all dieser Klassen sind ebenfalls immutable.

Um die aktuellen Inhalte von Tabellen mit den erwarteten Inhalten zu vergleichen, existiert die Klasse Diff<T>. Ein Diff-Objekt (welches ebenfalls immutable ist) repräsentiert den Unterschied zweier Tabellen. Siehe Diff.h.

Die eigentliche DbAccessTester-Klasse (die einzige Klasse, die mutable ist) besitzt folgenden Header:

// DbAccessTester.h

// ...

#include "Array.h"

#include "SharedPtr.h"

#include "Diff.h"

#include "Database.h"

using namespace std;

class **IDatabaseAdapter** {

public:

virtual **~IDatabaseAdapter**() {

}

virtual void **open**() = 0;

virtual void **close**() = 0;

virtual void **beginTransaction**() = 0;

virtual void **commitTransaction**() = 0;

virtual void **rollbackTransaction**() = 0;

virtual int **execute**(const string& sql) = 0;

virtual SharedPtr<DatabaseInfo> **getDatabaseInfo**() = 0;

virtual SharedPtr<Database> **getDatabase**() = 0;

};

typedef Diff<SharedPtr<Row>> **TableDiff**;

class **DbAccessTester** {

private:

// ...

public:

**DbAccessTester**(IDatabaseAdapter& adapter,

Array<string>::Builder& prepareStringsBuilder,

bool trace = false);

void **open**() const;

void **close**() const;

const SharedPtr<Database> **getDatabase**() const;

void **drop**() const;

void **prepare**() const;

void **execute**(const string& sql) const;

void **beginTransaction**() const;

void **commitTransaction**() const;

void **rollbackTransaction**() const;

void **play**();

void **replay**();

bool **compare**();

const SharedPtr<TableDiff> **diffs**(const string& tableName) const;

const SharedPtr<Array<SharedPtr<TableDiff>>> **diffs**() const;

void **printDiffs**(ostream& out = cerr) const;

};

Das Interface IDatabaseAdapter muss jeweils für einen konkreten Datenbank-Typ implementiert werden. Die Implementierung für SQLite sieht wie folgt aus:

// SQLiteAdapter.h

#include <string>

#include "CppSQLite3.h"

#include "DbAccessTester.h"

using namespace std;

class **SQLiteAdapter**: public IDatabaseAdapter {

private:

CppSQLite3DB& **db\_**;

const char\* **filename\_**;

string **errorMsg**(const string& method, const CppSQLite3Exception& e);

public:

**SQLiteAdapter**(CppSQLite3DB& db, const char\* filename);

void **open**();

void **close**();

int **execute**(const string& sql);

void **beginTransaction**();

void **commitTransaction**();

void **rollbackTransaction**();

SharedPtr<DatabaseInfo> **getDatabaseInfo**();

SharedPtr<Database> **getDatabase**();

};

// SQLiteAdapter.cpp

#include "SQLiteAdapter.h"

**SQLiteAdapter::SQLiteAdapter**(CppSQLite3DB& db, const char\* filename) :

db\_(db), filename\_(filename) {

}

string **SQLiteAdapter::errorMsg**(const string& method,

const CppSQLite3Exception& e) {

stringstream s;

s << "SQLiteAdapter-Error (" << method << ") : " << e.errorMessage();

return s.str();

}

void **SQLiteAdapter::open**() {

try {

db\_.open(filename\_);

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("open", e));

}

}

void **SQLiteAdapter::close**() {

try {

db\_.close();

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("close", e));

}

}

int **SQLiteAdapter::execute**(const string& sql) {

try {

return db\_.execDML(sql.c\_str());

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("execute", e));

}

}

void **SQLiteAdapter::beginTransaction**() {

try {

db\_.execDML("begin transaction");

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("beginTransaction", e));

}

}

void **SQLiteAdapter::commitTransaction**() {

try {

db\_.execDML("commit transaction");

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("commitTransaction", e));

}

}

void **SQLiteAdapter::rollbackTransaction**() {

try {

db\_.execDML("rollback transaction");

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("rollbackTransaction", e));

}

}

SharedPtr<DatabaseInfo> **SQLiteAdapter::getDatabaseInfo**() {

try {

Array<SharedPtr<TableInfo>>::Builder tableInfosBuilder;

const char\* metaSql = "SELECT name FROM sqlite\_master WHERE "

"type='table' ORDER BY name;";

for (CppSQLite3Query metaQuery = db\_.execQuery(metaSql);

!metaQuery.eof();

metaQuery.nextRow()) {

string tableName = metaQuery.fieldValue(0);

stringstream sql;

sql << "SELECT \* FROM " << tableName << " where 1 = 0";

CppSQLite3Query query = db\_.execQuery(sql.str().c\_str());

Array<SharedPtr<ColumnInfo>>::Builder columnInfosBuilder;

for (int i = 0; i < query.numFields(); i++)

columnInfosBuilder <<

new ColumnInfo(query.fieldName(i));

TableInfo\* tableInfo =

new TableInfo(tableName, columnInfosBuilder);

tableInfosBuilder << tableInfo;

}

return new DatabaseInfo(tableInfosBuilder);

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("getDatabaseInfo", e));

}

}

SharedPtr<Database> **SQLiteAdapter::getDatabase**() {

try {

SharedPtr<DatabaseInfo> databaseInfo = getDatabaseInfo();

Array<SharedPtr<Table>>::Builder tablesBuilder;

for (unsigned int i = 0;

i < databaseInfo->tableInfos.size(); i++) {

SharedPtr<TableInfo> tableInfo =

databaseInfo->tableInfos[i];

stringstream sql;

sql << "SELECT \* FROM " << tableInfo->name;

Array<SharedPtr<Row>>::Builder rowsBuilder;

for (CppSQLite3Query q = db\_.execQuery(sql.str().c\_str());

!q.eof();

q.nextRow()) {

Array<string>::Builder rowBuilder;

for (int i = 0; i < q.numFields(); i++)

rowBuilder << q.fieldValue(i);

rowsBuilder << new Row(rowBuilder);

}

tablesBuilder << new Table(tableInfo, rowsBuilder);

}

return new Database(databaseInfo, tablesBuilder);

}

catch (const CppSQLite3Exception& e) {

throw runtime\_error(errorMsg("getDatabase", e));

}

}

Die Klassen sind sorgfältig getestet. Es existieren die folgenden Testklassen:

SharedPtrTest

ArrayTest

DiffTest

DbAccessTesterTest

SQLiteAdapterTest

Die Datei main.cpp enthält das Hauptprogramm.

Zusätzlich existiert eine Klasse Counter, die als Basisklasse vieler der oben beschriebenen Klassen verwendet wird. Diese Klasse wird verwendet, um die korrekte Freigabe aller allokierten Objekte zu demonstrieren. (By the way: die Verwendung der SharedPtr-Klasse führt dazu, dass nirgendwo ein delete erforderlich ist (mit einer einzigen Ausnahme…)).

Die Klasse NoCopyNoAssign schließlich ist eine Basisklasse aller Klassen (der meisten hier diskutierten Klassen), welche die Copy-Konstruktion und das Assignment verbietet. Objekte werden also nirgendwo kopiert (was i.d.R. aufwendig wäre…).

# Literatur

Johannes Link: Softwaretests mit JUnit (DPunkt 2005)

Das Buch enthält u.a. eine kleine Einführung in JUnit (ein Testwerkzeug für Java). Der weitaus größere Teil des Buches befasst sich aber mit den Konzepten der Testgetriebenen Softwareentwicklung. Das Buch ist also auf jeden Fall auch interessant für C++-Entwickler.

Kent Beck: Test-Driven Development by Example (Addison Wesley 2005)

Beck ist der TDD-Papst … Das Buch enthält hauptsächlich zwei ausführliche Beispiele: das Money-Beispiel (in Java) und ein xUnit-Beispiel (das wird in Python vorgestellt)

Martin Fowler: Refactoring (Addison-Wesley 2005)

Fowler schreibt nur vernünftige Bücher (sehr empfehlenswert auch: UML Destilled – für Leute, die UML einfach nur verwenden wollen, ohne zuvor ein achtsemestriges UML-Studium machen zu wollen – ohne also das offizielle Handbuch zu UML lesen wollen…)

Zu gtest:

<https://github.com/google/googletest/blob/master/googletest/docs/Primer.md>

<https://github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md>

<https://github.com/google/googletest/blob/master/googletest/docs/FAQ.md>

Zu gmock

// <https://github.com/google/googletest/blob/master/googlemock/docs/v1_7/ForDummies.md>

// <https://github.com/google/googletest/blob/master/googlemock/docs/v1_7/CheatSheet.md>

// <https://github.com/google/googletest/blob/master/googlemock/docs/v1_7/FrequentlyAskedQuestions.md>

// <https://github.com/google/googletest/blob/master/googlemock/docs/v1_7/CookBook.md>