**Test Driven Development**

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

**Solutions**

OOA

OOD

OOP

OOD

OOA

OOP

OOA

OOP

OOD

OOP

OOA

OOD

OOD

OOP

OOA

OOP

OOD

OOA

Johannes Nowak

Johannes Nowak

e-mail: johannes.nowak@t-online.de

Juli / August 2016

**Inhalt**

[1 Einführung 5](#_Toc458342469)

[2 TDD – Ein einfaches Beispiel 6](#_Toc458342470)

[3 Testing - Das gtest-Framework 7](#_Toc458342471)

[4 Testing - Übungen 8](#_Toc458342472)

[4.1 Stack 8](#_Toc458342473)

[4.2 Trimmer 10](#_Toc458342474)

[4.3 CSVReader 11](#_Toc458342475)

[4.4 Statemachine 12](#_Toc458342476)

[4.5 CommandHandling 13](#_Toc458342477)

[4.6 SharedPtr 15](#_Toc458342478)

[5 Mocking – Das gmock-Framework 17](#_Toc458342479)

[6 Mocking – Das fakeit-Framework 18](#_Toc458342480)

[7 Mocking - Übungen 19](#_Toc458342481)

[7.1 AccountService 19](#_Toc458342482)

[7.2 GroupChangeReader 22](#_Toc458342483)

[8 Test Driven Development - Parser 24](#_Toc458342484)

[8.1 Start 24](#_Toc458342485)

[8.2 Specials 25](#_Toc458342486)

[8.3 Whitespace 27](#_Toc458342487)

[8.4 Numbers 28](#_Toc458342488)

[8.5 Values 30](#_Toc458342489)

[8.6 Interface 32](#_Toc458342490)

[8.7 ParseEmptyInput 33](#_Toc458342491)

[8.8 ParseNumber 34](#_Toc458342492)

[8.9 ParseTimes 35](#_Toc458342493)

[8.10 ParseMultiplicative 37](#_Toc458342494)

[8.11 ParseAdditive 39](#_Toc458342495)

[8.12 ParseNested 41](#_Toc458342496)

[8.13 ParseUnaries 44](#_Toc458342497)

[8.14 Expressions 46](#_Toc458342498)

[8.15 ExpressionParser 49](#_Toc458342499)

[8.16 GenericParser 52](#_Toc458342500)

[9 Test Driven Development – Logische Schaltungen 55](#_Toc458342501)

[9.1 Start 55](#_Toc458342502)

[9.2 Refactoring 57](#_Toc458342503)

[9.3 States 59](#_Toc458342504)

[9.4 Toggle 61](#_Toc458342505)

[9.5 Refactoring 63](#_Toc458342506)

[9.6 Inputs und Outputs 65](#_Toc458342507)

[9.7 Refactoring 68](#_Toc458342508)

[9.8 Connect 70](#_Toc458342509)

[9.9 Preconditions 72](#_Toc458342510)

[9.10 Logic 74](#_Toc458342511)

[10 Refactoring – Gruppenwechsel 77](#_Toc458342512)

[10.1 Start 77](#_Toc458342513)

[10.2 Orders, Customer, Products 80](#_Toc458342514)

[10.3 Printer 84](#_Toc458342515)

[10.4 OrderReader 87](#_Toc458342516)

[10.5 Repository 89](#_Toc458342517)

[10.6 OrderGroupChanger 92](#_Toc458342518)

[10.7 Repository Test 94](#_Toc458342519)

[10.8 Proxy 95](#_Toc458342520)

# Einführung

Keine Aufgaben, also auch keine Lösungen…

# TDD – Ein einfaches Beispiel

Keine Aufgaben, also auch keine Lösungen…

# Testing - Das gtest-Framework

Keine Aufgaben, also auch keine Lösungen…

# Testing - Übungen

## Stack

// 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**) {

Stack<int> stack(2);

stack.push(10);

EXPECT\_FALSE(stack.isEmpty());

EXPECT\_FALSE(stack.isFull());

stack.push(20);

EXPECT\_FALSE(stack.isEmpty());

EXPECT\_TRUE(stack.isFull());

}

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

Stack<int> stack(2);

stack.push(10);

stack.push(20);

EXPECT\_THROW(stack.push(30), runtime\_error);

EXPECT\_TRUE(stack.isFull());

}

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

Stack<int> stack(2);

stack.push(10);

stack.push(20);

EXPECT\_EQ(20, stack.top());

EXPECT\_EQ(20, stack.top());

}

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

Stack<int> stack(2);

EXPECT\_THROW(stack.top(), runtime\_error);

}

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

Stack<int> stack(2);

stack.push(10);

stack.push(20);

EXPECT\_TRUE(stack.isFull());

EXPECT\_EQ(20, stack.pop());

EXPECT\_FALSE(stack.isFull());

EXPECT\_EQ(10, stack.pop());

EXPECT\_TRUE(stack.isEmpty());

}

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

Stack<int> stack(2);

EXPECT\_THROW(stack.pop(), runtime\_error);

}

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

Stack<int> stack1(2);

stack1.push(10);

Stack<int> stack2(2);

stack2 = stack1;

stack1.push(20);

EXPECT\_TRUE(stack1.isFull());

EXPECT\_EQ(20, stack1.top());

EXPECT\_EQ(10, stack2.pop());

EXPECT\_TRUE(stack2.isEmpty());

}

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

Stack<int> stack1(2);

stack1.push(10);

Stack<int> stack2(stack1);

stack1.push(20);

stack2.push(30);

EXPECT\_TRUE(stack1.isFull());

EXPECT\_TRUE(stack2.isFull());

EXPECT\_EQ(20, stack1.top());

EXPECT\_EQ(30, stack2.pop());

}

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

Stack<int> stack1(2);

Stack<int> stack2(2);

Stack<int> stack3(200);

EXPECT\_EQ(stack1, stack2);

EXPECT\_NE(stack1, stack3);

stack1.push(10);

EXPECT\_NE(stack1, stack2);

stack2.push(10);

EXPECT\_EQ(stack1, stack2);

stack2.push(20);

EXPECT\_NE(stack1, stack2);

stack1.push(20);

EXPECT\_EQ(stack1, stack2);

}

## Trimmer

// 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**) {

stringstream in("hello world\n");

stringstream out;

Trimmer::trim(in, out);

EXPECT\_EQ("hello world\n", out.str());

}

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

stringstream in(" hello world \n");

stringstream out;

Trimmer::trim(in, out);

EXPECT\_EQ("hello world\n", out.str());

}

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

stringstream in(" hello \n\n \n world \n ");

stringstream out;

Trimmer::trim(in, out);

EXPECT\_EQ("hello\nworld\n", out.str());

}

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

stringstream in(" \n\n \n \t\t \n ");

stringstream out;

Trimmer::trim(in, out);

EXPECT\_EQ("", out.str());

}

## CSVReader

// 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");

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

string\* ptr = reader.readLine();

assertTokens(ptr, "1111", "Pascal", "30.0", "1970");

ptr = reader.readLine();

assertTokens(ptr, "2222", "Modula", "40.0", "1980");

ptr = reader.readLine();

ASSERT\_TRUE(ptr == 0);

}

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

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

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

string\* ptr = reader.readLine();

assertTokens(ptr, "1111", "Pascal", "30.0", "1970");

ptr = reader.readLine();

assertTokens(ptr, "2222", "Modula", "40.0", "1980");

ptr = reader.readLine();

ASSERT\_TRUE(ptr == 0);

}

## Statemachine

// 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());

}

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

MyStateMachine machine;

machine.dispatch(new EventA(42));

machine.dispatch(new EventB("Hello"));

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

}

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

MyStateMachine machine;

machine.dispatch(new EventA(42));

machine.dispatch(new EventB("Hello"));

machine.dispatch(new EventA(77));

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

}

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

MyStateMachine machine;

machine.dispatch(new EventA(42));

machine.dispatch(new EventB("Hello"));

machine.dispatch(new EventB("World"));

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

}

## CommandHandling

// 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**) {

Handlers handlers;

XYHandler\* handler = new XYHandler;

handlers << (XCommand::Handler\*)handler;

handlers.dispatch(new XCommand(42));

handlers.dispatch(new YCommand(3.14));

handlers.dispatch(new XCommand(77));

handlers.dispatch(new YCommand(2.71));

EXPECT\_EQ(2, handler->xValues.size());

EXPECT\_EQ(42, handler->xValues[0]);

EXPECT\_EQ(77, handler->xValues[1]);

EXPECT\_EQ(2, handler->yValues.size());

EXPECT\_DOUBLE\_EQ(3.14, handler->yValues[0]);

EXPECT\_DOUBLE\_EQ(2.71, handler->yValues[1]);

}

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**, **twoCommandTypesTwoHandlerType**) {

Handlers handlers;

XHandler\* xHandler = new XHandler;

YHandler\* yHandler = new YHandler;

handlers << xHandler << yHandler;

handlers.dispatch(new XCommand(42));

handlers.dispatch(new YCommand(3.14));

handlers.dispatch(new XCommand(77));

handlers.dispatch(new YCommand(2.71));

EXPECT\_EQ(2, xHandler->xValues.size());

EXPECT\_EQ(42, xHandler->xValues[0]);

EXPECT\_EQ(77, xHandler->xValues[1]);

EXPECT\_EQ(2, yHandler->yValues.size());

EXPECT\_DOUBLE\_EQ(3.14, yHandler->yValues[0]);

EXPECT\_DOUBLE\_EQ(2.71, yHandler->yValues[1]);

}

## SharedPtr

// 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());

}

# Mocking – Das gmock-Framework

Keine Aufgaben, also auch keine Lösungen…

# Mocking – Das fakeit-Framework

Keine Aufgaben, also auch keine Lösungen…

# Mocking - Übungen

## AccountService

// 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) {

Account account = dao\_.findAccount(nr);

account.balance += amount;

dao\_.updateAccount(account);

dao\_.insertMovement(Movement(nr, amount));

}

void **withdraw**(int nr, int amount) {

Account account = findAccount(nr);

if (amount > account.balance + account.credit)

throw runtime\_error("das gehe gar nicht");

account.balance -= amount;

dao\_.updateAccount(account);

dao\_.insertMovement(Movement(nr, -amount));

}

void **transfer**(int fromNr, int toNr, int amount) {

withdraw(fromNr, amount);

deposit(toNr, amount);

}

};

// AccountServiceTest.cpp

// ...

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**) {

EXPECT\_CALL(\*dao, findAccount(4711))

.WillOnce(Return(Account(4711, 5000, 0)));

EXPECT\_CALL(\*dao, updateAccount(Account(4711, 6000, 0)))

.Times(1);

EXPECT\_CALL(\*dao, insertMovement(Movement(4711, 1000)));

service->deposit(4711, 1000);

}

TEST\_F(**AccountServiceTest**, **withdraw**) {

EXPECT\_CALL(\*dao, findAccount(4711))

.WillOnce(Return(Account(4711, 5000, 0)));

EXPECT\_CALL(\*dao, updateAccount(Account(4711, 4000, 0)))

.Times(1);

EXPECT\_CALL(\*dao, insertMovement(Movement(4711, -1000)));

service->withdraw(4711, 1000);

}

TEST\_F(**AccountServiceTest**, **illegalWithdraw**) {

EXPECT\_CALL(\*dao, findAccount(4711))

.WillOnce(Return(Account(4711, 500, 0)));

EXPECT\_CALL(\*dao, updateAccount(Account(4711, 4000, 0))).Times(0);

EXPECT\_CALL(\*dao, insertMovement(Movement(4711, -1000))).Times(0);

try {

service->withdraw(4711, 1000);

FAIL();

}

catch(const runtime\_error& e) {

// okay

}

}

TEST\_F(**AccountServiceTest**, **transfer**) {

EXPECT\_CALL(\*dao, findAccount(4711))

.WillOnce(Return(Account(4711, 5000, 0)));

EXPECT\_CALL(\*dao, findAccount(4712))

.WillOnce(Return(Account(4712, 5000, 0)));

EXPECT\_CALL(\*dao, updateAccount(Account(4711, 4000, 0))).Times(1);

EXPECT\_CALL(\*dao, updateAccount(Account(4712, 6000, 0))).Times(1);

EXPECT\_CALL(\*dao, insertMovement(Movement(4711, -1000)));

EXPECT\_CALL(\*dao, insertMovement(Movement(4712, 1000)));

service->transfer(4711, 4712, 1000);

}

## GroupChangeReader

// GroupChangeReaderTest.cpp

// ...

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);

EXPECT\_CALL(p, processBegin()).Times(1);

EXPECT\_CALL(p, processGroupBegin(1000)).Times(1);

EXPECT\_CALL(p, processPosition(orders[0])).Times(1);

EXPECT\_CALL(p, processGroupEnd(1000)).Times(1);

EXPECT\_CALL(p, processEnd()).Times(1);

r.run();

}

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);

EXPECT\_CALL(p, processBegin()).Times(1);

EXPECT\_CALL(p, processGroupBegin(1000)).Times(1);

EXPECT\_CALL(p, processPosition(Order(1000, 100, 10))).Times(1);

EXPECT\_CALL(p, processPosition(Order(1000, 200, 20))).Times(1);

EXPECT\_CALL(p, processGroupEnd(1000)).Times(1);

EXPECT\_CALL(p, processEnd()).Times(1);

r.run();

}

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);

EXPECT\_CALL(p, processBegin()).Times(1);

EXPECT\_CALL(p, processGroupBegin(1000)).Times(1);

EXPECT\_CALL(p, processPosition(Order(1000, 100, 10))).Times(1);

EXPECT\_CALL(p, processPosition(Order(1000, 200, 20))).Times(1);

EXPECT\_CALL(p, processGroupEnd(1000)).Times(1);

EXPECT\_CALL(p, processGroupBegin(2000)).Times(1);

EXPECT\_CALL(p, processPosition(Order(2000, 300, 30))).Times(1);

EXPECT\_CALL(p, processGroupEnd(2000)).Times(1);

EXPECT\_CALL(p, processEnd()).Times(1);

r.run();

}

# Test Driven Development - Parser

## Start

Die Aufgabenbeschreibung befindet sich im Skript.

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **emptyInput**) {

stringstream stream("");

Scanner scanner(stream);

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

// Scanner.h

// ...

typedef enum { **END** } **Symbol**;

class **Scanner** {

public:

**Scanner**(istream& in) {

}

void **next**() {

}

Symbol **current**() {

return END;

}

};

## Specials

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **onlySpecials**) {

stringstream stream("+-()\*/");

Scanner scanner(stream);

scanner.next();

EXPECT\_EQ(PLUS, scanner.current());

scanner.next();

EXPECT\_EQ(MINUS, scanner.current());

scanner.next();

EXPECT\_EQ(OPEN, scanner.current());

scanner.next();

EXPECT\_EQ(CLOSE, scanner.current());

scanner.next();

EXPECT\_EQ(TIMES, scanner.current());

scanner.next();

EXPECT\_EQ(DIV, scanner.current());

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

// Scanner.h

typedef enum {

**UNDEFINED**, **END**, **PLUS**, **MINUS**, **TIMES**, **DIV**, **OPEN**, **CLOSE**

} **Symbol**;

class **Scanner** {

private:

istream& **in\_**;

map<int, Symbol> **specials\_**;

int **ch\_**;

Symbol **current\_**;

public:

**Scanner**(istream& in) : in\_(in) {

specials\_[-1] = END;

specials\_['+'] = PLUS;

specials\_['-'] = MINUS;

specials\_['\*'] = TIMES;

specials\_['/'] = DIV;

specials\_['('] = OPEN;

specials\_[')'] = CLOSE;

current\_ = UNDEFINED;

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

}

void **next**() {

map<int,Symbol>::iterator it = specials\_.find(ch\_);

if (it != specials\_.end()) {

current\_ = it->second;

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

return;

}

throw runtime\_error("illegal char");

}

Symbol **current**() {

if (current\_ == UNDEFINED)

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

return current\_;

}

};

## Whitespace

Der Scanner soll Spezial-Zeichen natürlich auch dann erkennen, wenn diese in Whitespace-Zeichen eingebettet sind. Alle bisherigen Methoden des ScannerTests sollten bestehen bleiben. Die Testklasse wird durch eine neue Methode specialsWithWhitespace erweitert, in welcher dem Scanner ein "komplizierterer" String übergeben wird: " + -( ".

Die Erweiterung des Tests:

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **specialsWithWhitespace**) {

stringstream stream(" + -( ");

Scanner scanner(stream);

scanner.next();

EXPECT\_EQ(PLUS, scanner.current());

scanner.next();

EXPECT\_EQ(MINUS, scanner.current());

scanner.next();

EXPECT\_EQ(OPEN, scanner.current());

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

Die Erweiterung der Implementierung:

// Scanner.h

// ...

class **Scanner** {

private:

// ...

public:

// ...

void **next**() {

while(isspace(ch\_))

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

map<int,Symbol>::iterator it = specials\_.find(ch\_);

if (it != specials\_.end()) {

current\_ = it->second;

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

return;

}

throw runtime\_error("illegal char");

}

// ...

};

## Numbers

Der Scanner soll Zahlen erkennen können: sowohl Ganzzahlen als auch Gleitkomma-Zahlen.

Ihm wird z.B. folgender String übergeben: " 42 +3.14- ". (Natürlich ist die Reihenfolge dieser Symbolfolge "unsinnig" – aber über Sinn resp. Unsinn urteilt der Scanner nicht…)

Die Erweiterung des ScannerTests (wir führen eine neues Symbol ein: NUMBER):

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **numbers**) {

stringstream stream(" 42 +3.14- ");

Scanner scanner(stream);

scanner.next();

EXPECT\_EQ(NUMBER, scanner.current());

scanner.next();

EXPECT\_EQ(PLUS, scanner.current());

scanner.next();

EXPECT\_EQ(NUMBER, scanner.current());

scanner.next();

EXPECT\_EQ(MINUS, scanner.current());

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

Die Erweiterung des Scanners:

// Scanner.h

// ...

typedef enum { ... **NUMBER** } **Symbol**;

class **Scanner** {

private:

// ...

public:

// ...

void **next**() {

while(isspace(ch\_))

get();

if (isdigit(ch\_)) {

get();

while (isdigit(ch\_)) {

get();

}

if (ch\_ == '.') {

get();

while (isdigit(ch\_)) {

get();

}

}

current\_ = NUMBER;

return;

}

map<int,Symbol>::iterator it = specials\_.find(ch\_);

if (it != specials\_.end()) {

current\_ = it->second;

get();

return;

}

throw runtime\_error("illegal char");

}

// ...

};

## Values

Die Zahlenwerte sollen nun auch berechnet werden können – und mittels einer neu einzuführenden Scanner-Methode namens number() zurückgeliefert werden.

Die Erweiterung des Tests:

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **values**) {

stringstream stream(" 42 +3.14- ");

Scanner scanner(stream);

scanner.next();

EXPECT\_EQ(NUMBER, scanner.current());

EXPECT\_DOUBLE\_EQ(42.0, scanner.number());

scanner.next();

EXPECT\_EQ(PLUS, scanner.current());

scanner.next();

EXPECT\_EQ(NUMBER, scanner.current());

EXPECT\_DOUBLE\_EQ(3.14, scanner.number());

scanner.next();

EXPECT\_EQ(MINUS, scanner.current());

EXPECT\_ANY\_THROW(scanner.number());

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

Die Erweiterung des Scanners:

// Scanner.h

// ...

class **Scanner** {

private:

// ...

double **number\_**;

public:

// ...

void **next**() {

while(isspace(ch\_))

get();

if (isdigit(ch\_)) {

stringstream stream;

stream << (char)ch\_;

get();

while (isdigit(ch\_)) {

stream << (char)ch\_;

get();

}

if (ch\_ == '.') {

stream << (char)ch\_;

get();

while (isdigit(ch\_)) {

stream << (char)ch\_;

get();

}

}

stream >> number\_;

current\_ = NUMBER;

return;

}

map<int,Symbol>::iterator it = specials\_.find(ch\_);

if (it != specials\_.end()) {

current\_ = it->second;

get();

return;

}

throw runtime\_error("illegal char");

}

// ...

double **number**() {

if (current\_ != NUMBER)

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

return number\_;

}

// ...

};

## Interface

Um alternative Scanner-Implementierungen zuzulassen, spezifizieren wir nun ein Interfaces IScanner, welches von der Scanner-Klasse implementiert wird.

Wir zeigen im Test, dass die Klasse Scanner das Interface IScanner implementiert (eigentlich ist das kein neuer Test, sondern nur eine Demonstration):

// ScannerTest.cpp

// ...

TEST(**ScannerTest**, **interface**) {

stringstream stream(" 42 ");

Scanner s(stream);

IScanner& scanner = s;

scanner.next();

EXPECT\_EQ(NUMBER, scanner.current());

EXPECT\_DOUBLE\_EQ(42.0, scanner.number());

scanner.next();

EXPECT\_EQ(END, scanner.current());

}

Das neue Interface (in welchem natürlich auch der Symbol-Typ definiert sein muss):

// IScanner.h

typedef enum { ... } **Symbol**;

class **IScanner** {

public:

virtual **~IScanner**() { }

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

virtual Symbol **current**() = 0;

virtual double **number**() = 0;

};

Hier der modifizierte Scanner:

// Scanner.h

class **Scanner** : public IScanner {

// ...

};

## ParseEmptyInput

Wir beginnen nun mit dem Parser.

Der Parser benötigt einen Scanner. Also wird dem Konstruktor von Parser der Scanner übergeben (der Parser wird natürlich mit jedem IScanner zufrieden sein).

Der Parser besitzt eine parse-Methode, die den Wert des geparsten Ausdrucks zurückliefert.

Sofern die Eingabe leer ist, soll die parse-Methode eine Exception werfen.

Hier die Spezifikation:

// ParserTest.cpp

TEST(**ParserTest**, **emptyInput**) {

stringstream stream("");

Scanner s(stream);

IScanner& scanner = s;

Parser parser(scanner);

EXPECT\_ANY\_THROW({ double v = parser.parse(); cout << v; });

}

Hier die anfängliche Implementierung der Parser-Klasse:

// Parser.h

// ...

class **Parser** {

private:

IScanner& **scanner\_**;

Symbol **current\_**;

public:

**Parser**(IScanner& scanner) : scanner\_(scanner), current\_(UNDEFINED) {

}

double **parse**() {

next();

if (current\_ != NUMBER)

throw runtime\_error("Number expected");

return 0;

}

private:

void **next**() {

scanner\_.next();

current\_ = scanner\_.current();

}

};

## ParseNumber

Der einfachste Ausdruck, dessen Wert berechnet werden kann, ist eine Zahl.

Expression:

Number

Zunächst auch hier wieder die Spezifikation des Parsers:

// ParserTest.cpp

// ...

TEST(**ParserTest**, **oneNumber**) {

stringstream stream(" 3.14");

Scanner scanner(stream);

Parser parser(scanner);

EXPECT\_NEAR(3.14, parser.parse(), 0.01);

}

Die Erweiterung des Parsers:

// Parser.h

// ...

class **Parser** {

private:

IScanner& **scanner\_**;

Symbol **current\_**;

public:

**Parser**(IScanner& scanner) : scanner\_(scanner), current\_(UNDEFINED) {

}

double **parse**() {

next();

if (current\_ != NUMBER)

throw runtime\_error("Number expected");

double value = scanner\_.number();

next();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

// ...

};

## ParseTimes

Im nächsten Schritt soll eine beliebige Multiplikation geparst werden können:

Number

Number

\*

Die Erweiterung der Testklasse:

// ParserTest.cpp

TEST(**ParserTest**, **times**) {

stringstream stream(" 2 \* 3 \* 4 ");

Scanner scanner(stream);

Parser parser(scanner);

EXPECT\_NEAR(24.0, parser.parse(), 0.01);

}

Die Erweiterung des Parsers (wir führen eine neue Methode parseNumber ein – denn an zwei Stellen des neuen Algorithmus muss jeweils eine Zahl gelesen werden):

// Parser.h

// ...

class **Parser** {

// ...

public:

// ...

double **parse**() {

next();

double value = parseNumber();

while (current\_ == TIMES) {

next();

double v = parseNumber();

value = value \* v;

}

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

double **parseNumber**() {

if (current\_ != NUMBER)

throw runtime\_error("Number expected");

double value = scanner\_.number();

next();

return value;

}

// ...

};

## ParseMultiplicative

Ein multiplikativer Ausdruck ist eine Folge von Zahlen, welche mittels \* oder / miteinander verbunden sind:

Multiplicative:

Number

Number

\*

/

Ein neuer Testfall:

// ParserTest.cpp

TEST(**ParserTest**, **multiplicative**) {

stringstream stream(" 2 \* 3 / 4 \* 6 ");

Scanner scanner(stream);

Parser parser(scanner);

EXPECT\_NEAR(9.0, parser.parse(), 0.01);

}

Wir erweitern die Implementierung durch die Einführung einer neuen Methode: parseMultiplicative. Diese implementiert das obige Railroad-Diagramm: sie ruft an zwei Stelle parseNumber auf. In der parse-Methode wird dann statt parseNumber die neue Methode parseMultiplicative aufgerufen:

// Parser.h

// ...

class **Parser** {

private:

// ...

public:

// ...

double **parse**() {

next();

double value = parseMultiplicative();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

double **parseMultiplicative**() {

double value = parseNumber();

while (current\_ == TIMES || current\_ == DIV) {

bool times = current\_ == TIMES;

next();

double v = parseNumber();

if (times)

value = value \* v;

else

value = value / v;

}

return value;

}

double **parseNumber**() { ... }

// ...

};

## ParseAdditive

Ein Ausdruck kann ein additiver Ausdruck sein: ein multiplikativer Ausdruck gefolgt von einem additiven Operator (+, -) gefolgt von einem weiteren multiplikativen Ausdruck gefolgt von….

Additive:

Multiplicative

Multiplicative

+

-

Multiplicative:

Number

Number

\*

/

Die Erweiterung der Testklasse:

// ParserTest.cpp

// ...

TEST(**ParserTest**, **additive**) {

stringstream stream(" 2 + 2 \* 10 / 4 - 6 \* 2");

Scanner scanner(stream);

Parser parser(scanner);

EXPECT\_NEAR(2.0 + 2 \* 10 / 4 - 6 \* 2, parser.parse(), 0.01);

}

Wir erweitern die Implementierung um eine Methode parseAdditive. Diese ruft parseMultiplicative auf. Und parse ruft nun parseAdditive auf:

// Parser.h

// ...

class **Parser** {

private:

// ...

public:

// ...

double **parse**() {

next();

double value = parseAdditive();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

double **parseAdditive**() {

double value = parseMultiplicative();

while (current\_ == PLUS || current\_ == MINUS) {

bool plus = current\_ == PLUS;

next();

double v = parseMultiplicative();

if (plus)

value = value + v;

else

value = value - v;

}

return value;

}

double **parseMultiplicative**() {

double value = parseNumber();

while (current\_ == TIMES || current\_ == DIV) {

bool times = current\_ == TIMES;

next();

double v = parseNumber();

if (times)

value = value \* v;

else

value = value / v;

}

return value;

}

double **parseNumber**() { ... }

// ...

};

## ParseNested

Ein multiplikativer Ausdruck war bislang definiert als eine multiplikative Verknüpfung von Zahlen. Dort aber, wo eine Zahl stehen kann, kann natürlich auch ein weiterer, geklammerter Ausdruck stehen.

Wir korrigieren (bzw. erweitern) also: ein multiplikativer Ausdruck ist eine multiplikative Verknüpfung von Simple-Expression. Und eine Simple Expression ist entweder eine Zahl oder aber ein geklammerter additiver Ausdruck. Die ganze Definition ist somit rekursiv geworden.

Additive:

Multiplicative

Multiplicative

+

-

Multiplicative:

Simple

Simple

\*

/

Simple:

Number

Additive

(

)

Die Erweiterung des Tests:

// ParserTest.cpp

// ...

TEST(**ParserTest**, **nested**) {

stringstream stream(" (2.0 + 2) \* 10 / (4 - (6 \* 2))");

Scanner scanner(stream);

Parser parser(scanner);

EXPECT\_NEAR( (2.0 + 2) \* 10 / (4 - (6 \* 2)), parser.parse(), 0.01);

}

Wir erweitern die Implementierung um eine Methode parseSimple. Diese ruft ggf. parseNumber oder aber parseAdditive auf. Und parseMultiplicative ruft nicht mehr parseNumber auf, sondern parseSimple:

// Parser.h

// ...

class **Parser** {

private:

// ...

public:

// ...

double **parse**() {

next();

double value = parseAdditive();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

double **parseAdditive**() { ... }

double **parseMultiplicative**() {

double value = parseSimple();

while (current\_ == TIMES || current\_ == DIV) {

bool times = current\_ == TIMES;

next();

double v = parseSimple();

if (times)

value = value \* v;

else

value = value / v;

}

return value;

}

double **parseSimple**() {

double value;

if (current\_ == NUMBER) {

value = parseNumber();

}

else if (current\_ == OPEN) {

next();

value = parseAdditive();

if (current\_ != CLOSE)

throw runtime\_error(") expected");

next();

}

else {

throw runtime\_error("Number or ( expected");

}

return value;

}

double **parseNumber**() { ... }

// ...

};

## ParseUnaries

Einer Zahl oder einem geklammerten Ausdruck kann ein unärer Operator vorausgehen (ein unärer Plus- oder Minus-Operator). Wir erweitern das Railroad-Diagramm für einen Simple-Ausdruck:

Simple:

Number

Additive

(

)

+

-

Wir erweitern die Testklasse:

// ParserTest.cpp

// ...

TEST(**ParserTest**, **unaries**) {

stringstream stream(" (-2.0 + +2) \* 10 / (4 - (6 \* 2))");

Scanner scanner(stream);

Parser parser(scanner);

EXPECT\_NEAR( (-2.0 + +2) \* 10 / (4 - (6 \* 2)), parser.parse(), 0.01);

}

In der Implementierung muss parseSimple ein wenig erweitert werden:

// Parser.h

// ...

class **Parser** {

private:

// ...

public:

// ...

double **parse**() {

next();

double value = parseAdditive();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

double **parseAdditive**() { ... }

double **parseMultiplicative**() { ... }

double **parseSimple**() {

double value;

Symbol unary = UNDEFINED;

if (current\_ == PLUS || current\_ == MINUS) {

unary = current\_;

next();

}

if (current\_ == NUMBER) {

value = parseNumber();

} else if (current\_ == OPEN) {

next();

value = parseAdditive();

if (current\_ != CLOSE)

throw runtime\_error(") expected");

next();

} else {

throw runtime\_error("Number or ( expected");

}

if (unary == PLUS)

value = +value;

else if (unary == MINUS)

value = -value;

return value;

}

double **parseNumber**() { ... }

// ...

};

## Expressions

Der Parser soll keinen Wert berechnen, sondern einen Baum von Expressions liefern. Zuvor wurde immer von Expression gesprochen, ohne aber dieses Konzept in Klassen zu implementieren.

Eine Expression ist entweder eine NumberExpression, eine BinaryExpression oder eine UnaryExpression. Eine BinaryExpression verweist auf ihre beiden Seiten (left und right), die wiederum eine Expression sind. Eine BinaryExpression besitzt zudem einen Pointer auf eine Berechnungsfunktion (Binary::plus, Binary::minus, etc.). Eine UnaryExpression verweist auf eine innere Expression (inner) und besitzt ebenfalls eine Berechnungsfunktion (Unary::plus, Unary::minus).

Expression spezifiziert eine Methode double evaluate(), welche von den abgeleiteten Klassen natürlich implementiert werden muss.

*Expression*

NumberExpression

*BinaryExpression*

double(

double, double)

right

left

*UnaryExpression*

double(

double)

inner

Ein Test:

// ParserTest.cpp

// ...

TEST(**ExpressionTest**, **binary**) {

Expression\* e1 = new NumberExpression(10);

Expression\* e2 = new NumberExpression(32);

Expression\* e3 = new BinaryExpression(e1, Binary::plus, e2);

Expression\* e4 = new UnaryExpression(Unary::minus, e3);

EXPECT\_EQ(-42, e4->evaluate());

delete e4;

}

Die Implementierung:

// Operator.h

class **Binary** {

public:

typedef double (\***op**)(double, double);

static double **plus**(double x, double y) {

return x + y;

}

static double **minus**(double x, double y) {

return x - y;

}

static double **times**(double x, double y) {

return x \* y;

}

static double **div**(double x, double y) {

return x / y;

}

};

class **Unary** {

public:

typedef double (\***op**)(double);

static double **plus**(double x) {

return +x;

}

static double **minus**(double x) {

return -x;

}

};

// Expression.h

// ...

class **Expression** {

public:

virtual **~Expression**() {

}

virtual double **evaluate**() const = 0;

};

class **NumberExpression**: public Expression {

public:

const double **value**;

**NumberExpression**(double value) :

value(value) {

}

double **evaluate**() const {

return value;

}

};

class **BinaryExpression**: public Expression {

public:

const Expression\* **left**;

const Binary::op **op**;

const Expression\* **right**;

**BinaryExpression**(Expression\* left, Binary::op op, Expression\* right) :

left(left), op(op), right(right) {

}

virtual **~BinaryExpression**() {

delete left;

delete right;

}

double **evaluate**() const {

return op(left->evaluate(), right->evaluate());

}

};

class **UnaryExpression**: public Expression {

public:

const Unary::op **op**;

const Expression\* **inner**;

**UnaryExpression**(Unary::op op, Expression\* inner) :

op(op), inner(inner) {

}

virtual **~UnaryExpression**() {

delete inner;

}

double **evaluate**() const {

return op(inner->evaluate());

}

};

## ExpressionParser

Die Implementierung des neuen Parsers übernehmen wir zum großen Teil von der Implementierung des bisherigen Parses (den wir im übrigen zu ValueParser umtaufen).

Wir ersetzen double durch Expression – dort, wo bislang aufgrund jeweils eines binären oder unären Operators ein neuer double berechnet wurde, wird nun eine neue Expression erzeugt; und dort, wo der Scanner eine Zahl liefert, wird diese Zahl in einer NumberExpression verpackt.

Die neue Testklasse (überall dort, wo bislang das Resultat von parser.parse() verwendet wurde, wird nun das Resultat von parser.parse()->evaluate() verwendet):

// ExpressionParserTest.cpp

// ...

TEST(**ExpressionParserTest**, **times**) {

stringstream stream(" 2 \* 3 \* 4 ");

Scanner scanner(stream);

ExpressionParser parser(scanner);

EXPECT\_NEAR(2 \* 3 \* 4 , parser.parse()->evaluate(), 0.01);

}

// ...

Die Implementierung des neuen Parsers:

// Parser.h

// ...

class **ExpressionParser** {

private:

IScanner& **scanner\_**;

Symbol **current\_**;

public:

**ExpressionParser**(IScanner& scanner) :

scanner\_(scanner), current\_(UNDEFINED) {

}

Expression\* **parse**() {

next();

Expression\* value = parseAdditive();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

Expression\* **parseAdditive**() {

Expression\* value = parseMultiplicative();

while (current\_ == PLUS || current\_ == MINUS) {

bool plus = current\_ == PLUS;

next();

Expression\* v = parseMultiplicative();

if (plus)

value = new BinaryExpression(value, Binary::plus, v);

else

value = new BinaryExpression(value, Binary::minus, v);

}

return value;

}

Expression\* **parseMultiplicative**() {

Expression\* value = parseSimple();

while (current\_ == TIMES || current\_ == DIV) {

bool times = current\_ == TIMES;

next();

Expression\* v = parseSimple();

if (times)

value = new BinaryExpression(value, Binary::times, v);

else

value = new BinaryExpression(value, Binary::div, v);

}

return value;

}

Expression\* **parseSimple**() {

Expression\* value;

Symbol unary = UNDEFINED;

if (current\_ == PLUS || current\_ == MINUS) {

unary = current\_;

next();

}

if (current\_ == NUMBER) {

value = parseNumber();

} else if (current\_ == OPEN) {

next();

value = parseAdditive();

if (current\_ != CLOSE)

throw runtime\_error(") expected");

next();

} else {

throw runtime\_error("Number or ( expected");

}

if (unary == PLUS)

value = new UnaryExpression(Unary::plus, value);

else if (unary == MINUS)

value = new UnaryExpression(Unary::minus, value);

return value;

}

Expression\* **parseNumber**() {

if (current\_ != NUMBER)

throw runtime\_error("Number expected");

Expression\* value = new NumberExpression(scanner\_.number());

next();

return value;

}

void **next**() {

scanner\_.next();

current\_ = scanner\_.current();

}

};

## GenericParser

Das Studium der letzten Erweiterung sei dem Leser / der Leserin überlassen:

// Parser.h

// ...

template<typename T>

class **Parser** {

private:

IScanner& **scanner\_**;

Symbol **current\_**;

public:

**Parser**(IScanner& scanner) :

scanner\_(scanner), current\_(UNDEFINED) {

}

virtual **~Parser()** {

}

T **parse**() {

next();

T value = parseAdditive();

if (current\_ != END)

throw runtime\_error("End of input exprected");

return value;

}

private:

T **parseAdditive**() {

T value = parseMultiplicative();

while (current\_ == PLUS || current\_ == MINUS) {

bool plus = current\_ == PLUS;

next();

T v = parseMultiplicative();

if (plus)

value = create(value, Binary::plus, v);

else

value = create(value, Binary::minus, v);

}

return value;

}

T **parseMultiplicative**() {

T value = parseSimple();

while (current\_ == TIMES || current\_ == DIV) {

bool times = current\_ == TIMES;

next();

T v = parseSimple();

if (times)

value = create(value, Binary::times, v);

else

value = create(value, Binary::div, v);

}

return value;

}

T **parseSimple**() {

T value;

Symbol unary = UNDEFINED;

if (current\_ == PLUS || current\_ == MINUS) {

unary = current\_;

next();

}

if (current\_ == NUMBER) {

value = parseNumber();

} else if (current\_ == OPEN) {

next();

value = parseAdditive();

if (current\_ != CLOSE)

throw runtime\_error(") expected");

next();

} else {

throw runtime\_error("Number or ( expected");

}

if (unary == PLUS)

value = create(Unary::plus, value);

else if (unary == MINUS)

value = create(Unary::minus, value);

return value;

}

T **parseNumber**() {

if (current\_ != NUMBER)

throw runtime\_error("Number expected");

T value = create(scanner\_.number());

next();

return value;

}

void **next**() {

scanner\_.next();

current\_ = scanner\_.current();

}

protected:

virtual T **create**(T left, Binary::op op, T right) = 0;

virtual T **create**(Unary::op op, T inner) = 0;

virtual T **create**(double value) = 0;

};

// ValueParser.h

// ...

class **ValueParser** : public Parser<double> {

public:

**ValueParser**(IScanner& scanner) :

Parser(scanner) {

}

protected:

virtual double **create**(double left, Binary::op op, double right) {

return op(left, right);

}

virtual double **create**(Unary::op op, double inner) {

return op(inner);

}

virtual double **create**(double value) {

return value;

}

};

// ExpressionParser.h

// ...

class **ExpressionParser** : public Parser<Expression\*> {

public:

**ExpressionParser**(IScanner& scanner) :

Parser(scanner) {

}

protected:

virtual Expression\* **create**(

Expression\* left, Binary::op op, Expression\* right) {

return new BinaryExpression(left, op, right);

}

virtual Expression\* **create**(Unary::op op, Expression\* inner) {

return new UnaryExpression(op, inner);

}

virtual Expression\* **create**(double value) {

return new NumberExpression(value);

}

};

// ValueParserTest.cpp

// ...

TEST(**ValueParserTest**, **nested**) {

stringstream stream(" (2.0 + 2) \* 10 / (4 - (6 \* 2))");

Scanner scanner(stream);

ValueParser parser(scanner);

EXPECT\_NEAR( (2.0 + 2) \* 10 / (4 - (6 \* 2)), parser.parse(), 0.01);

}

// ...

// ExpressionParserTest.cpp

// ...

TEST(**ExpressionParserTest**, **nested**) {

stringstream stream(" (2.0 + 2) \* 10 / (4 - (6 \* 2))");

Scanner scanner(stream);

ExpressionParser parser(scanner);

EXPECT\_NEAR( (2.0 + 2) \* 10 / (4 - (6 \* 2)),

parser.parse()->evaluate(), 0.01);

}

// ...

# Test Driven Development – Logische Schaltungen

## Start

Die Aufgabenbeschreibung befindet sich im Skript.

// 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());

}

// 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

Die Implementierung der drei abstrakten Methoden der Basisklasse in den abgeleiteten Klassen ist ziemlich "dumm".

Die Testklasse bleibt unverändert.

Wir definieren in der Basisklasse drei Instanzvariaben: name\_, inputCount\_ und outputCount\_. Dann können die entsprechenden Methoden bereits in der Basisklasse implementiert werden.

// Circuit.h

class **Circuit** {

protected:

const char\* **name**\_;

const unsigned int **inputCount**\_;

const unsigned int **outputCount**\_;

public:

**Circuit**(const char\* name,

unsigned int inputCount, unsigned int outputCount) :

name\_(name), inputCount\_(inputCount), outputCount\_(outputCount) {

}

virtual **~Circuit**() {

}

const char\* **name**() {

return name\_;

}

unsigned int **inputCount**() {

return inputCount\_;

}

unsigned int **outputCount**() {

return outputCount\_;

}

};

// Circutis.h

// ...

class **AndCircuit**: public Circuit {

public:

**AndCircuit**() :

Circuit("and", 2, 1) {

}

};

class **OrCircuit**: public Circuit {

public:

**OrCircuit**() :

Circuit("or", 2, 1) {

}

};

class **NotCircuit**: public Circuit {

public:

**NotCircuit**() :

Circuit("not", 1, 1) {

}

};

Die abgeleiteten Klassen sind nun wesentlich schlanker…

Nachteil: Jede And-Schaltung hat Instanzvariablen, deren Werte mit den Werten anderer And-Schaltungen identisch sind (und das gilt natürlich auch für alle anderen Schaltungstypen). Das ist redundant. Wir können hier aber auch schlecht auf static-Elemente ausweichen. Vielleicht wird sich dieses Problem später noch klären…

## States

Wie können Schaltungen den State ihrer Inputs und ihrer Outputs repräsentieren? Nach der Konstruktion einer Schaltung sollten diese States bereits korrekt gesetzt und abfragbar sein.

Wir erweitern den Test:

// CircuitTest.cpp

// ...

TEST\_F(**CircuitTest**, **statesAfterConstruction**) {

EXPECT\_FALSE(c1->inputState(0));

EXPECT\_FALSE(c1->inputState(1));

EXPECT\_FALSE(c1->outputState(0));

EXPECT\_FALSE(c2->inputState(0));

EXPECT\_FALSE(c2->inputState(1));

EXPECT\_FALSE(c2->outputState(0));

EXPECT\_FALSE(c3->inputState(0));

EXPECT\_TRUE(c3->outputState(0));

}

Und erweitern die Implementierung:

// Circuit.h

class **Circuit** {

protected:

const char\* **name\_**;

const unsigned int **inputCount\_**;

const unsigned int **outputCount\_**;

bool\* const **inputs\_**;

bool\* const **outputs\_**;

public:

**Circuit**(const char\* name,

unsigned int inputCount, unsigned int outputCount) :

name\_(name),

inputCount\_(inputCount),

outputCount\_(outputCount),

inputs\_(new bool[inputCount]),

outputs\_(new bool[outputCount]) {

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

inputs\_[i] = false;

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

outputs\_[i] = false;

}

virtual **~Circuit**() {

delete[] inputs\_;

delete[] outputs\_;

}

// ...

bool **inputState**(unsigned int index) {

return inputs\_[index];

}

bool **outputState**(unsigned int index) {

return outputs\_[index];

}

};

Und die Klasse NotCircuit muss angepasst werden(!):

// Circuit.h

// ...

class **AndCircuit**: public Circuit { ... }

class **OrCircuit**: public Circuit { ... }

class **NotCircuit**: public Circuit {

public:

**NotCircuit**() :

Circuit("not", 1, 1) {

outputs\_[0] = true;

}

};

## Toggle

Eingänge müssen getoggelt werden können. Ein Eingang, der getoggelt wird, invertiert seinen Status. Dann müssen auch die Output-States wieder neu berechnet und gesetzt werden.

Die Erweiterung der Test-Klasse:

// CircuitTest.cpp

// ...

TEST\_F(**CircuitTest**, **toggle**) {

c1->toggle(0);

EXPECT\_TRUE(c1->inputState(0));

EXPECT\_FALSE(c1->inputState(1));

EXPECT\_FALSE(c1->outputState(0));

c1->toggle(1);

EXPECT\_TRUE(c1->inputState(0));

EXPECT\_TRUE(c1->inputState(1));

EXPECT\_TRUE(c1->outputState(0));

c1->toggle(0);

EXPECT\_FALSE(c1->inputState(0));

EXPECT\_TRUE(c1->inputState(1));

EXPECT\_FALSE(c1->outputState(0));

c2->toggle(0);

EXPECT\_TRUE(c2->inputState(0));

EXPECT\_FALSE(c2->inputState(1));

EXPECT\_TRUE(c2->outputState(0));

c2->toggle(1);

EXPECT\_TRUE(c2->inputState(0));

EXPECT\_TRUE(c2->inputState(1));

EXPECT\_TRUE(c2->outputState(0));

c2->toggle(0);

EXPECT\_FALSE(c2->inputState(0));

EXPECT\_TRUE(c2->inputState(1));

EXPECT\_TRUE(c2->outputState(0));

c3->toggle(0);

EXPECT\_TRUE(c3->inputState(0));

EXPECT\_FALSE(c3->outputState(0));

c3->toggle(0);

EXPECT\_FALSE(c3->inputState(0));

EXPECT\_TRUE(c3->outputState(0));

}

Die Erweiterung der Circuit-Klasse:

// Circuit.h

class **Circuit** {

// ...

public:

// ...

virtual void **toggle**(unsigned int inputIndex) = 0;

};

Die Erweiterungen der abgeleiteten Klassen:

// Circuits.h

// ...

class **AndCircuit**: public Circuit {

public:

// ...

virtual void **toggle**(unsigned int inputIndex) {

inputs\_[inputIndex] = ! inputs\_[inputIndex];

outputs\_[0] = inputs\_[0] && inputs\_[1];

}

};

class **OrCircuit**: public Circuit {

public:

// ...

virtual void **toggle**(unsigned int inputIndex) {

inputs\_[inputIndex] = ! inputs\_[inputIndex];

outputs\_[0] = inputs\_[0] || inputs\_[1];

}

};

class **NotCircuit**: public Circuit {

public:

// ...

virtual void **toggle**(unsigned int inputIndex) {

inputs\_[inputIndex] = ! inputs\_[inputIndex];

outputs\_[0] = ! inputs\_[0];

}

};

## Refactoring

Wir erkennen, dass in allen toggle-Methoden der abgeleiteten Klasse stets zunächst der index-te Eingang invertiert wird. Dies ist redundant – und sollte daher einmal in der Basisklasse erfolgen.

Die Testklasse bleibt unverändert.

Die Basisklasse implementiert nun toggle – unter Zuhilfenahme einer abstrakten evaluate-Methode, die dann in den abgeleiteten Klassen implementiert wird.

// Circuit.h

class **Circuit** {

// ...

public:

// ...

void **toggle**(unsigned int inputIndex) {

inputs\_[inputIndex] = ! inputs\_[inputIndex];

evaluate();

}

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

};

Die abgeleiteten Klassen:

// Circuits.h

class **AndCircuit**: public Circuit {

public:

**AndCircuit**() :

Circuit("and", 2, 1) {

evaluate();

}

virtual void **evaluate**() {

outputs\_[0] = inputs\_[0] && inputs\_[1];

}

};

class **OrCircuit**: public Circuit {

public:

**OrCircuit**() :

Circuit("or", 2, 1) {

evaluate();

}

virtual void **evaluate**() {

outputs\_[0] = inputs\_[0] || inputs\_[1];

}

};

class **NotCircuit**: public Circuit {

public:

**NotCircuit**() :

Circuit("not", 1, 1) {

evaluate();

}

virtual void **evaluate**() {

outputs\_[0] = ! inputs\_[0];

}

};

Man beachte, dass evaluate nun auch in den Konstruktoren aufgerufen wird. Die Sonderbehandlung der Klasse NotCircuit entfällt nun also…

## Inputs und Outputs

// CircuitTest.cpp

// ...

TEST\_F(**CircuitTest**, **statesAfterConstruction**) {

EXPECT\_FALSE(c1->input(0).state());

EXPECT\_FALSE(c1->input(1).state());

EXPECT\_FALSE(c1->output(0).state());

EXPECT\_FALSE(c2->input(0).state());

EXPECT\_FALSE(c2->input(1).state());

EXPECT\_FALSE(c2->output(0).state());

EXPECT\_FALSE(c3->input(0).state());

EXPECT\_TRUE(c3->output(0).state());

}

TEST\_F(**CircuitTest**, **toggle**) {

c1->toggle(0);

EXPECT\_TRUE(c1->input(0).state());

EXPECT\_FALSE(c1->input(1).state());

EXPECT\_FALSE(c1->output(0).state());

c1->toggle(1);

EXPECT\_TRUE(c1->input(0).state());

EXPECT\_TRUE(c1->input(1).state());

EXPECT\_TRUE(c1->output(0).state());

c1->toggle(0);

EXPECT\_FALSE(c1->input(0).state());

EXPECT\_TRUE(c1->input(1).state());

EXPECT\_FALSE(c1->output(0).state());

c2->toggle(0);

EXPECT\_TRUE(c2->input(0).state());

EXPECT\_FALSE(c2->input(1).state());

EXPECT\_TRUE(c2->output(0).state());

c2->toggle(1);

EXPECT\_TRUE(c2->input(0).state());

EXPECT\_TRUE(c2->input(1).state());

EXPECT\_TRUE(c2->output(0).state());

c2->toggle(0);

EXPECT\_FALSE(c2->input(0).state());

EXPECT\_TRUE(c2->input(1).state());

EXPECT\_TRUE(c2->output(0).state());

c3->toggle(0);

EXPECT\_TRUE(c3->input(0).state());

EXPECT\_FALSE(c3->output(0).state());

c3->toggle(0);

EXPECT\_FALSE(c3->input(0).state());

EXPECT\_TRUE(c3->output(0).state());

}

// Circuit.h

struct **IInput** {

virtual **~IInput**() { };

virtual bool **state**() = 0;

};

struct **IOutput** {

virtual **~IOutput**() { };

virtual bool **state**() = 0;

};

class **Circuit** {

private:

struct **Input** : public IInput {

bool **state\_**;

**Input**() : state\_(false) {

}

void **setState**(bool newState) {

state\_ = newState;

}

bool **state**() {

return state\_;

}

};

struct **Output** : public IOutput {

bool **state\_**;

**Output**() : state\_(false) {

}

void **setState**(bool newState) {

state\_ = newState;

}

bool **state**() {

return state\_;

}

};

protected:

// ...

Input\* const **inputs\_**;

Output\* const **outputs\_**;

public:

**Circuit**(...) :

name\_(name),

inputCount\_(inputCount),

outputCount\_(outputCount),

inputs\_(new Input[inputCount]),

outputs\_(new Output[outputCount]) {

}

virtual **~Circuit**() {

delete[] inputs\_;

delete[] outputs\_;

}

// ...

IInput& **input**(unsigned int index) {

return inputs\_[index];

}

IOutput& **output**(unsigned int index) {

return outputs\_[index];

}

void **toggle**(unsigned int inputIndex) {

inputs\_[inputIndex].setState(! inputs\_[inputIndex].state());

evaluate();

}

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

};

// Circuits.h

class **AndCircuit**: public Circuit {

public:

// ...

virtual void **evaluate**() {

outputs\_[0].setState(inputs\_[0].state() && inputs\_[1].state());

}

};

class **OrCircuit**: public Circuit {

public:

// ...

virtual void **evaluate**() {

outputs\_[0].setState(inputs\_[0].state() || inputs\_[1].state());

}

};

class **NotCircuit**: public Circuit {

public:

// ...

virtual void **evaluate**() {

outputs\_[0].setState(! inputs\_[0].state());

}

};

## Refactoring

Wir reformulieren den toggle-Test:

// CircuitTest.h

// ...

TEST\_F(**CircuitTest**, **toggle**) {

c1->input(0).toggle();

EXPECT\_TRUE(c1->input(0).state());

EXPECT\_FALSE(c1->input(1).state());

EXPECT\_FALSE(c1->output(0).state());

c1->input(1).toggle();

EXPECT\_TRUE(c1->input(0).state());

EXPECT\_TRUE(c1->input(1).state());

EXPECT\_TRUE(c1->output(0).state());

c1->input(0).toggle();

EXPECT\_FALSE(c1->input(0).state());

EXPECT\_TRUE(c1->input(1).state());

EXPECT\_FALSE(c1->output(0).state());

c2->input(0).toggle();

EXPECT\_TRUE(c2->input(0).state());

EXPECT\_FALSE(c2->input(1).state());

EXPECT\_TRUE(c2->output(0).state());

c2->input(1).toggle();

EXPECT\_TRUE(c2->input(0).state());

EXPECT\_TRUE(c2->input(1).state());

EXPECT\_TRUE(c2->output(0).state());

c2->input(0).toggle();

EXPECT\_FALSE(c2->input(0).state());

EXPECT\_TRUE(c2->input(1).state());

EXPECT\_TRUE(c2->output(0).state());

c3->input(0).toggle();

EXPECT\_TRUE(c3->input(0).state());

EXPECT\_FALSE(c3->output(0).state());

c3->input(0).toggle();

EXPECT\_FALSE(c3->input(0).state());

EXPECT\_TRUE(c3->output(0).state());

}

// Circuit.h

struct **IInput** {

virtual **~IInput**() {

}

virtual bool **state**() = 0;

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

};

struct **IOutput** {

virtual **~IOutput**() {

}

virtual bool **state**() = 0;

};

class **Circuit** {

private:

struct **Input**: public IInput {

bool **state\_**;

Circuit\* **owner\_**;

**Input**() : state\_(false), owner\_(0) {

}

void **setState**(bool newState) {

state\_ = newState;

}

bool **state**() {

return state\_;

}

void **toggle**() {

state\_ = !state\_;

owner\_->evaluate();

}

};

struct **Output**: public IOutput {

bool **state\_**;

Circuit\* **owner\_**;

**Output**() : state\_(false), owner\_(0) {

}

void **setState**(bool newState) {

state\_ = newState;

}

bool **state**() {

return state\_;

}

};

protected:

// ...

public:

**Circuit**(...) : ... {

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

inputs\_[i].owner\_ = this;

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

outputs\_[i].owner\_ = this;

}

// ...

};

## Connect

// CircuitTest.cpp

// ...

TEST\_F(**CircuitTest**, **connect**) {

c1->output(0).connectTo(c3->input(0));

EXPECT\_TRUE(c3->output(0).state());

c1->input(0).toggle();

EXPECT\_TRUE(c3->output(0).state());

c1->input(1).toggle();

EXPECT\_TRUE(c3->input(0).state());

EXPECT\_FALSE(c3->output(0).state());

}

// Circuit.h

struct **IInput** {

virtual **~IInput**() {

}

virtual bool **state**() = 0;

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

};

struct IOutput {

virtual **~IOutput**() {

}

virtual bool **state**() = 0;

virtual void **connectTo**(IInput& input) = 0;

};

class **Circuit** {

private:

struct Output;

struct **Input**: public IInput {

bool **state\_**;

Circuit\* **owner\_**;

Output\* **source\_**;

**Input**() :

state\_(false), owner\_(0), source\_(0) {

}

void **setState**(bool newState) {

if (state\_ != newState) {

state\_ = newState;

owner\_->evaluate();

}

}

bool **state**() {

return state\_;

}

void **toggle**() {

state\_ = !state\_;

owner\_->evaluate();

}

};

struct **Output**: public IOutput {

bool **state\_**;

Circuit\* **owner\_**;

vector<Input\*> **targets\_**;

**Output**() :

state\_(false), owner\_(0) {

}

void **setState**(bool newState) {

state\_ = newState;

for(vector<Input\*>::iterator iter = targets\_.begin();

iter != targets\_.end(); ++iter)

(\*iter)->setState(newState);

}

bool **state**() {

return state\_;

}

void **connectTo**(IInput& input) {

Input\* in = (Input\*)&input;

in->source\_ = this;

targets\_.push\_back(in);

in->setState(state\_);

}

};

protected:

// ...

public:

// ...

};

## Preconditions

// CircuitTest.cpp

// ...

TEST\_F(**CircuitTest**, **preconditions**) {

ASSERT\_TRUE(c3->input(0).isFree());

c1->output(0).connectTo(c3->input(0));

ASSERT\_FALSE(c3->input(0).isFree());

EXPECT\_ANY\_THROW(c2->output(0).connectTo(c3->input(0)));

EXPECT\_ANY\_THROW(c3->input(0).toggle());

}

// Circuit.h

// ...

struct **IInput** {

// ...

virtual bool **isFree**() = 0;

};

struct **IOutput** {

// ...

};

class **Circuit** {

private:

struct **Input**: public IInput {

// ...

void **toggle**() {

if (! isFree())

throw runtime\_error("input not free");

state\_ = !state\_;

owner\_->evaluate();

}

bool **isFree**() {

return source\_ == 0;

}

};

struct **Output**: public IOutput {

// ...

void **connectTo**(IInput& input) {

Input\* in = (Input\*)&input;

if (! in->isFree())

throw runtime\_error("input not free");

in->source\_ = this;

targets\_.push\_back(in);

in->setState(state\_);

}

};

// ...

};

## Logic

// CircuitTest.cpp

class **CircuitTest** : public testing::Test {

protected:

Circuit\* c1;

Circuit\* c2;

Circuit\* c3;

public:

virtual void **SetUp**() override {

c1 = new Circuit(AndLogic::instance);

c2 = new Circuit(OrLogic::instance);

c3 = new Circuit(NotLogic::instance);

}

virtual void **TearDown**() override {

delete c1;

delete c2;

delete c3;

}

};

// ...

TEST\_F(**CircuitTest**, **logic**) {

ASSERT\_STREQ("and", c1->name());

c1->input(0).toggle();

ASSERT\_TRUE(c1->input(0).state());

ASSERT\_FALSE(c1->input(1).state());

ASSERT\_FALSE(c1->output(0).state());

c1->setLogic(OrLogic::instance);

EXPECT\_STREQ("or", c1->name());

EXPECT\_TRUE(c1->input(0).state());

EXPECT\_FALSE(c1->input(1).state());

EXPECT\_TRUE(c1->output(0).state());

c1->input(0).toggle();

EXPECT\_FALSE(c1->output(0).state());

}

// Logic.h

class **Logic** {

public:

const char\* const **name**;

const unsigned int **inputCount**;

const unsigned int **outputCount**;

**protected**:

**Logic**(const char\* name, unsigned int inputCount, unsigned int outputCount) :

name(name), inputCount(inputCount), outputCount(outputCount) {

}

public:

virtual **~Logic()** {

}

virtual void **calculate**(bool\* input, bool\* ouput) = 0;

};

// Logics.h

// ...

class **AndLogic** : public Logic {

protected:

**AndLogic**() : Logic("and", 2, 1) { }

void **calculate**(bool\* input, bool\* ouput) {

ouput[0] = input[0] && input[1];

}

public:

static AndLogic **instance**;

};

class **OrLogic** : public Logic {

protected:

**OrLogic**() : Logic("or", 2, 1) { }

void **calculate**(bool\* input, bool\* ouput) {

ouput[0] = input[0] || input[1];

}

public:

static OrLogic **instance**;

};

class **NotLogic** : public Logic {

protected:

**NotLogic**() : Logic("not", 1, 1) { }

void **calculate**(bool\* input, bool\* ouput) {

ouput[0] = ! input[0];

}

public:

static NotLogic **instance**;

};

// Logics.cpp

// ...

AndLogic **AndLogic::instance**;

OrLogic **OrLogic::instance**;

NotLogic **NotLogic::instance**;

// Circuit.h

// ...

struct **IInput** { ... }

struct **IOutput** { ... }

class **Circuit** {

private:

struct **Input**: public IInput { ... }

struct **Output**: public IOutput { ... }

protected:

Logic\* **logic\_**;

Input\* const **inputs\_**;

Output\* const **outputs\_**;

public:

**Circuit**(Logic& logic) :

logic\_(&logic),

inputs\_(new Input[logic.inputCount]),

outputs\_(new Output[logic.outputCount]) {

for (unsigned int i = 0; i < logic.inputCount; i++)

inputs\_[i].owner\_ = this;

for (unsigned int i = 0; i < logic.outputCount; i++)

outputs\_[i].owner\_ = this;

evaluate();

}

virtual **~Circuit**() {

delete[] inputs\_;

delete[] outputs\_;

}

const char\* **name**() {

return logic\_->name;

}

unsigned int **inputCount**() {

return logic\_->inputCount;

}

unsigned int **outputCount**() {

return logic\_->outputCount;

}

IInput& **input**(unsigned int index) {

return inputs\_[index];

}

IOutput& **output**(unsigned int index) {

return outputs\_[index];

}

void **evaluate**() {

bool input[logic\_->inputCount];

bool output[logic\_->outputCount];

for (unsigned int i = 0; i < logic\_->inputCount; i++)

input[i] = inputs\_[i].state();

logic\_->calculate(input, output);

for (unsigned int i = 0; i < logic\_->outputCount; i++)

outputs\_[i].setState(output[i]);

}

void **setLogic**(Logic& logic) {

logic\_ = &logic;

evaluate();

}

};

# Refactoring – Gruppenwechsel

## Start

Das im folgenden zu refaktorierende Programm benutzt den CSVReader:

// CSVReader.h

class **CSVReader** {

public:

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

// ...

string\* **readLine**();

};

Zur Erinnerung:

An den Konstruktor wird ein istream übergeben, aus welchem der CSVReader die Daten einlesen wird. seperator ist dasjenige Zeichen, welches die in einer Zeile enthaltenen Wörter trennt. tokenCount ist die Anzahl der Wörter, die in jeder Zeile erwartet werden.

Die readLine-Methode liefert einen dynamisch allokierten Array von strings zurück – von den in der letzten gelesenen Zeile enthaltenen Wörtern. Dieser Array muss vom Aufrufer der readLine-Methode wieder freigegeben werden.

Hier das vorgefundene Programm (es sieht ganz entsetzlich aus):

// 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;

}

}

Hier zunächst eine Sammlung von Kritikpunkten:

* Aufträge, Kunden und Produkte werden als Arrays von strings repräsentiert. Sie sollten besser als Objekte von Klassen (Order, Customer, Product) repräsentiert sein. Statt mittels eines Index auf die jeweiligen Daten zuzugreifen, könnten dann die Namen von Instanzvariablen genutzt werden. Dies würde die Lesbarkeit deutlich erhöhen.
* Die demo-Methode macht fast alles selbst. Sie implementiert nicht nur um den eigentlichen Gruppenwechsel-Algorithmus, sondern bestimmt z.B. auch die konkrete Form der Druckausgaben. Auch kümmert sie sich um die Einzelheiten des Einlesens der Aufträge und der Zugriffs auf die Kunden- und Produkt-Daten.
* Der Zugriff auf die Kunden- resp. Produkt-Daten ist nicht optimal. Statt die Daten in einem vector vorzuhalten und dann sequentiell suchen zu müssen, sollten sie vielleicht in einer map untergebracht werden – um somit dann performanter suchen zu können.
* Das Programm enthält Redundanzen: die Kunden-Daten werden auf ganz ähnliche Weise eingelesen wie die Produkt-Daten. Vielleicht kann man ein dieses einlesen verallgemeinern. Dasselbe gilt natürlich auf für den Zugriff auf diese Daten.

Natürlich können wir nach den ersten Refaktorierungen nicht sofort vernünftig testen. Wir können aber die bisherige result.txt-Datei umbenennen zu expected-result.txt – und nach jeder Refaktorierung das Programm erneut starten. Und dann jeweils result.txt manuell mit expected-result.txt vergleichen…

Wir werden aber sehen, dass wir sukzessive kleine Klassen entwickeln können, die ihrerseits dann testbar sind. Dann wird auch irgendwann das gesamte System testbar sein…

## Orders, Customer, Products

Wie beginnen damit, einige einfache Value-Klassen einzuführen. Kunden sollen in Form von Customer-Objekte, Artikel in Form von Product-Objekten und Aufträge in Form von Order-Objekten repräsentiert werden:

// 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;

}

};

// Customer.h

// ...

class **Customer** {

public:

const int **nr**;

const string **name**;

**Customer**(int nr, const string& name) :

nr(nr), name(name) {

}

bool **operator ==**(const Customer& other) const {

return nr == other.nr && name == other.name;

}

};

// Product.h

// ...

class **Product** {

public:

const int **nr**;

const string **name**;

const int **price**;

**Product**(int nr, const string& name, int price) :

nr(nr), name(name), price(price) {

}

bool **operator ==**(const Product& other) const {

return nr == other.nr && name == other.name && price == other.price;

}

};

Die Objekte all dieser Klassen sind immutable. Alle Klassen überladen zudem den operator ==.

Für diese Klassen werden wir (bewusst) auf Tests verzichten – die Klassen sind zu trivial.

Wir beginnen mit der Methode getCustomer. Diese lieferte in der vorgefundenen Lösung einen Array von strings zurück. Wir bauen diese Methode derart um, dass sie nun eine Referenz auf ein Customer-Objekte liefert. Das heißt dann natürlich auch, dass der in der demo-Methode verwendete vector<string\*> ersetzt werden wird durch einen vector<Customer>.

Nach diesen Ersetzungen wird der Compiler einige Fehlermeldungen anzeigen. Wie sorgen dann dafür, dass der Compiler wieder zufrieden ist.

Nachdem wir mit Customer-Objekten Erfolg gehabt haben, wenden wir dieselbe Refaktorierung auf Products an.

Und sorgen schließlich dann dafür, dass auch die Eingaben durch Order-Objekte repräsentiert werden – indem eine neue globale Funktion namens readOrder eingeführt wird.

Hier das Resultat der Refaktorierungen:

// main.cpp

// ...

const Customer& **getCustomer**(

int number, const vector<Customer>& customers) {

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

if (customers[i].nr == number)

return customers[i];

}

throw runtime\_error("customer not found");

}

const Product& **getProduct**(

int number, vector<Product>& products) {

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

if (products[i].nr == number)

return products[i];

}

throw runtime\_error("product not found");

}

Order\* **readOrder**(CSVReader& reader) {

string\* tokens = reader.readLine();

if (tokens == 0)

return 0;

Order\* order = new Order(

atoi(tokens[0].c\_str()),

atoi(tokens[1].c\_str()),

atoi(tokens[2].c\_str()));

delete[] tokens;

return order;

}

void **demo**() {

vector<Customer> customers = vector<Customer>();

ifstream customersStream("customers.txt");

CSVReader customersReader(customersStream, ',', 2);

string\* customerTokens = customersReader.readLine();

while (customerTokens != 0) {

Customer customer(

atoi(customerTokens[0].c\_str()),

customerTokens[1]);

customers.push\_back(customer);

customerTokens = customersReader.readLine();

}

customersStream.close();

vector<Product> products = vector<Product>();

ifstream productsStream("products.txt");

CSVReader productsReader(productsStream, ',', 3);

string\* productTokens = productsReader.readLine();

while (productTokens != 0) {

Product product(

atoi(productTokens[0].c\_str()),

productTokens[1],

atoi(productTokens[2].c\_str()));

products.push\_back(product);

productTokens = productsReader.readLine();

}

productsStream.close();

ofstream out("result.txt");

out << "Orders" << endl << endl;

ifstream ordersStream("orders.txt");

CSVReader ordersReader(ordersStream, ';', 3);

Order\* order = readOrder(ordersReader);

int sum = 0;

while (order != 0) {

int customerNr = order->customerNr;

const Customer& customer =

getCustomer(order->customerNr, customers);

out << customerNr << " " << customer.name << endl;

int groupSum = 0;

while (order != 0 && order->customerNr == customerNr) {

const Product& product =

getProduct(order->productNr, products);

int positionPrice = product.price \* order->amount;

out << "\t" <<

product.nr << " " <<

product.name << " " <<

order->amount << " " <<

product.price << " " <<

positionPrice << endl;

groupSum += positionPrice;

delete order;

order = readOrder(ordersReader);

}

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;

}

Die Lesbarkeit der demo-Method ist deutlich verbessert worden.

## Printer

Die konkrete Form der Ausgaben soll nicht mehr im eigentlichen Gruppenwechsel-Algorithmus hinterlegt werden, sondern in einer neuen Printer-Klasse. Die Verantwortlichkeit für's Drucken liegt dann ausschließlich bei dieser Klasse.

Aus systematischen Gründen (Austauschbarkeit, Ersetzen der richtigen Implementierung durch ein Mock-Objekt, Verwendung von Proxies etc.) beginnen wird mit einem Interface:

// IPrinter.h

// ...

class **IPrinter** {

public:

virtual **~IPrinter**() { }

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

virtual void **printGroupBegin**(const Customer& customer) = 0;

virtual void **printPosition**(

const Product& product, int amount, int positionValue) = 0;

virtual void **printGroupEnd**(int groupSum) = 0;

virtual void **printEnd**(int totalSum) = 0;

};

Hier eine Implementierung, deren konkrete Ausgabe-Anweisungen eins-zu-eins aus der bisherigen demo-Methode übernommen wurde (wir haben die bisherige Funktionalität also nicht geändert, sondern nur verschoben):

// Printer.h

// ...

class **Printer** : public IPrinter {

private:

ostream& **out\_**;

public:

**Printer**(ostream& out) : out\_(out) {

}

void **printBegin**() {

out\_ << "Orders" << endl << endl;

}

void **printGroupBegin**(const Customer& customer) {

out\_ << customer.nr << " " << customer.name << endl;

}

void **printPosition**(

const Product& product, int amount, int positionValue) {

out\_ << "\t" <<

product.nr << " " <<

product.name << " " <<

amount << " " <<

product.price << " " <<

positionValue << endl;

}

void **printGroupEnd**(int groupSum) {

out\_ << "\t---------" << endl;

out\_ << "\t" << groupSum << endl;

out\_ << endl;

}

void **printEnd**(int totalSum) {

out\_ << "Total:\t" << totalSum << endl;

}

};

Man beachte, dass der Konstruktor mit einem ostream zufrieden ist (man kann also statt eines ofstreams auch einen stringstream übergeben…)

Die drei Hilfsmethoden (getCustomer, getProduct und readOrder) bleiben unverändert. Nur die demo-Methode wird geändert. Sie definiert nun eine Printer-Variable und delegiert alle Ausgabe-Operationen nun an diesen Printer:

// main.cpp

// ...

const Customer& **getCustomer**(...) { ... }

const Product& **getProduct**(...) { ... }

Order\* **readOrder**(CSVReader& reader) { ... }

void **demo**() {

vector<Customer> customers = vector<Customer>();

ifstream customersStream("customers.txt");

CSVReader customersReader(customersStream, ',', 2);

string\* customerTokens = customersReader.readLine();

while (customerTokens != 0) {

Customer customer(

atoi(customerTokens[0].c\_str()),

customerTokens[1]);

customers.push\_back(customer);

customerTokens = customersReader.readLine();

}

customersStream.close();

vector<Product> products = vector<Product>();

ifstream productsStream("products.txt");

CSVReader productsReader(productsStream, ',', 3);

string\* productTokens = productsReader.readLine();

while (productTokens != 0) {

Product product(

atoi(productTokens[0].c\_str()),

productTokens[1],

atoi(productTokens[2].c\_str()));

products.push\_back(product);

productTokens = productsReader.readLine();

}

productsStream.close();

ofstream out("result.txt");

Printer printer(out);

printer.printBegin();

ifstream ordersStream("orders.txt");

CSVReader ordersReader(ordersStream, ';', 3);

Order\* order = readOrder(ordersReader);

int sum = 0;

while (order != 0) {

int customerNr = order->customerNr;

const Customer& customer =

getCustomer(order->customerNr, customers);

printer.printGroupBegin(customer);

int groupSum = 0;

while (order != 0 && order->customerNr == customerNr) {

const Product& product =

getProduct(order->productNr, products);

int positionPrice = product.price \* order->amount;

printer.printPosition(

product, order->amount, positionPrice);

groupSum += positionPrice;

delete order;

order = readOrder(ordersReader);

}

sum += groupSum;

printer.printGroupEnd(groupSum);

}

printer.printEnd(sum);

ordersStream.close();

out.close();

cout << "Done. See results.txt" << endl;

}

Im Augenblick ist von dem Interface IPrinter noch kein Gebrauch gemacht worden. Das Interface wird erst später relevant werden.

## OrderReader

Im nächsten Schritt verschieben wird die im vorletzten Schritt eingeführte globale Funktion readOrder in eine neue Klasse – in die Klasse OrderReader. Auch diese Funktionalität ist aus systematischen Gründen zunächst wieder durch ein Interface spezifiziert:

// IOrderReader.h

// ...

class **IOrderReader** {

public:

virtual **~IOrderReader**() { }

virtual Order\* **read**() = 0;

};

// OrderReader.cpp

// ...

class **OrderReader** : public IOrderReader {

private:

CSVReader **reader**;

public:

**OrderReader**(istream& in) : reader(in, ';', 3) {

}

Order\* **read**() {

string\* tokens = reader.readLine();

if (tokens == 0)

return 0;

Order\* order = new Order(

atoi(tokens[0].c\_str()),

atoi(tokens[1].c\_str()),

atoi(tokens[2].c\_str()));

delete[] tokens;

return order;

}

};

Der Konstruktor ist mit jedem istream zufrieden (ifstream, stringstream…).

Die read-Methode liefert jeweils die nächste Auftragszeile in Form eines dynamisch allokierten Order-Objekts zurück (welches der Aufrufer der read-Methode freigeben muss).

Die neue demo-Methode:

void **demo**() {

vector<Customer> customers = vector<Customer>();

ifstream customersStream("customers.txt");

CSVReader customersReader(customersStream, ',', 2);

string\* customerTokens = customersReader.readLine();

while (customerTokens != 0) {

Customer customer(

atoi(customerTokens[0].c\_str()),

customerTokens[1]);

customers.push\_back(customer);

customerTokens = customersReader.readLine();

}

customersStream.close();

vector<Product> products = vector<Product>();

ifstream productsStream("products.txt");

CSVReader productsReader(productsStream, ',', 3);

string\* productTokens = productsReader.readLine();

while (productTokens != 0) {

Product product(

atoi(productTokens[0].c\_str()),

productTokens[1],

atoi(productTokens[2].c\_str()));

products.push\_back(product);

productTokens = productsReader.readLine();

}

productsStream.close();

ofstream out("result.txt");

Printer printer(out);

printer.printBegin();

ifstream ordersStream("orders.txt");

OrderReader orderReader(ordersStream);

Order\* order = orderReader.read();

int sum = 0;

while (order != 0) {

int customerNr = order->customerNr;

const Customer& customer =

getCustomer(order->customerNr, customers);

printer.printGroupBegin(customer);

int groupSum = 0;

while (order != 0 && order->customerNr == customerNr) {

const Product& product =

getProduct(order->productNr, products);

int positionPrice = product.price \* order->amount;

printer.printPosition(

product, order->amount, positionPrice);

groupSum += positionPrice;

delete order;

order = orderReader.read();

}

sum += groupSum;

printer.printGroupEnd(groupSum);

}

printer.printEnd(sum);

ordersStream.close();

out.close();

cout << "Done. See results.txt" << endl;

}

## Repository

Um auf die Customer- resp. Product-Daten zuzugreifen, soll sich die demo-Methode im folgenden Refaktorierungs-Schritt eines Repositories bedienen, welches (wiederum aus denselben systematischen Gründen wie beim Printer) mittels eines Interfaces spezifiziert ist:

// IRepository

// ...

class **IRepository** {

public:

virtual **~IRepository**() {

}

virtual const Customer& **getCustomer**(int nr) = 0;

virtual const Product& **getProduct**(int nr) = 0;

};

Hier eine Implementierung:

// IRepository

// ...

class **Repository** : public IRepository {

private:

vector<Customer> **customers\_**;

vector<Product> **products\_**;

public:

**Repository**(istream& customerStream, istream& productStream) {

CSVReader customerReader(customerStream, ',', 2);

CSVReader productReader(productStream, ',', 3);

customers\_ = vector<Customer>();

string\* customerTokens = customerReader.readLine();

while (customerTokens != 0) {

Customer customer(

atoi(customerTokens[0].c\_str()),

customerTokens[1]);

customers\_.push\_back(customer);

customerTokens = customerReader.readLine();

}

products\_ = vector<Product>();

string\* productTokens = productReader.readLine();

while (productTokens != 0) {

Product product(

atoi(productTokens[0].c\_str()),

productTokens[1],

atoi(productTokens[2].c\_str()));

products\_.push\_back(product);

productTokens = productReader.readLine();

}

}

const Customer& **getCustomer**(int nr) {

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

if (customers\_[i].nr == nr)

return customers\_[i];

}

throw runtime\_error("customer not found");

}

const Product& **getProduct**(int nr) {

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

if (products\_[i].nr == nr)

return products\_[i];

}

throw runtime\_error("product not found");

}

};

Auch bei dieser Implementierung wurden einfache bestehende Zeilen aus dem alten Programm in diese neue Klasse kopiert. Die Funktionalität ist also auf jeden Fall erhalten geblieben.

Natürlich könnte man später – in einem weiteren Schritt – die Performance der hier implementierten Suche verbessern: indem die vector-Objekte z.B. durch map-Objekte ersetzt werden. Wir sind aber zunächst einmal mit der einfachen Verschiebung zufrieden.

Das Hauptprogramm schrumpft nun mächtig zusammen:

// main.cpp

// ...

void **demo**() {

ifstream customersStream("customers.txt");

ifstream productsStream("products.txt");

Repository repository(customersStream, productsStream);

customersStream.close();

productsStream.close();

ofstream out("result.txt");

Printer printer(out);

printer.printBegin();

ifstream ordersStream("orders.txt");

OrderReader orderReader(ordersStream);

Order\* order = orderReader.read();

int sum = 0;

while (order != 0) {

int customerNr = order->customerNr;

const Customer& customer =

repository.getCustomer(order->customerNr);

printer.printGroupBegin(customer);

int groupSum = 0;

while (order != 0 && order->customerNr == customerNr) {

const Product& product =

repository.getProduct(order->productNr);

int positionPrice = product.price \* order->amount;

printer.printPosition(product, order->amount, positionPrice);

groupSum += positionPrice;

delete order;

order = orderReader.read();

}

sum += groupSum;

printer.printGroupEnd(groupSum);

}

printer.printEnd(sum);

ordersStream.close();

out.close();

cout << "Done. See results.txt" << endl;

}

## OrderGroupChanger

In nächsten Schritt verschieben wird den eigentlichen Gruppenwechsel-Algorithmus in eine neue Klasse – in die run-Methode der Klasse OrderGroupChanger. Dieser run-Methode werden drei Parameter übergeben: eine IOrderReader-Referenz, eine IRepository-Referenz und eine IPrinter-Referenz. Die run-Methode ist also nicht abhängig von konkreten Implementierungen dieser Interfaces. Die run-Methode ist einfach static:

// OrderGroupChanger.h

// ...

class **OrderGroupChanger** {

public:

static void **run**(

IOrderReader& reader,

IRepository& repository,

IPrinter& printer) {

printer.printBegin();

Order\* order = reader.read();

int sum = 0;

while (order != 0) {

int customerNr = order->customerNr;

const Customer& customer =

repository.getCustomer(order->customerNr);

printer.printGroupBegin(customer);

int groupSum = 0;

while (order != 0 && order->customerNr == customerNr) {

const Product& product =

repository.getProduct(order->productNr);

int positionPrice = product.price \* order->amount;

printer.printPosition(

product, order->amount, positionPrice);

groupSum += positionPrice;

delete order;

order = reader.read();

}

sum += groupSum;

printer.printGroupEnd(groupSum);

}

printer.printEnd(sum);

}

};

Die demo-Methode beschränkt sich nun darauf, die für den Gruppenwechsel erforderlichen Objekte zu definieren und dann schließlich die oben definiert run-Methode aufzurufen:

// main.cpp

// ...

void **demo**() {

cout << "Refactoring-OrderGroupChanger" << endl;

try {

ifstream ordersStream("orders.txt");

OrderReader orderReader(ordersStream);

ifstream customersStream("customers.txt");

ifstream productsStream("products.txt");

Repository repository(customersStream, productsStream);

customersStream.close();

productsStream.close();

ofstream out("result.txt");

Printer printer(out);

OrderGroupChanger::run(orderReader, repository, printer);

ordersStream.close();

out.close();

cout << "Done. See results.txt" << endl;

}

catch(const runtime\_error& e) {

cout << "Exception: " << e.what() << endl;

}

}

Das Resultat – dargestellt in Form eines Objekt-Diagramms:

: OrderReader

read ( )

*IOrderReader*

: Repository

getCutomer (…)

getCutomer (…)

*IRepository*

: Printer

*IPrinter*

printBegin ( )

printGroupBegin (…)

printPosition (…)

printGroupEnd (…)

printEnd (…)

OrderGroupChanger::run (…)

## Repository Test

Natürlich hätten wir zwischenzeitlich auch die einzelnen Komponenten testen können. Hier z.B. eine kleine Klasse zum Testen des Respositories:

// RepositoryTest.h

// ...

TEST(**RepositoryTest**, **customers**) {

stringstream customers(

"1000,Nowak\n2000,Rueschenpoehler\n3000,Montjoe\n");

stringstream products;

Repository r(customers, products);

EXPECT\_EQ(Customer(1000, "Nowak"), r.getCustomer(1000));

EXPECT\_EQ(Customer(2000, "Rueschenpoehler"), r.getCustomer(2000));

EXPECT\_EQ(Customer(3000, "Montjoe"), r.getCustomer(3000));

EXPECT\_THROW(r.getCustomer(9999), runtime\_error);

}

TEST(RepositoryTest, products) {

// ....

}

Auf ähnliche Weise könnten wir auch den OrderReader testen.

Und nun könnten wird schließlich auch die run-Methode des OrderGroupChangers testen – indem wir die drei Hilfsobjekte, welche die run-Methode benötigt, mocken…

## Proxy

Da unser System Interface-basiert aufgebaut ist, könnten wird nun auch Proxies verwenden:

: OrderReader

read ( )

*IOrderReader*

: Printer

*IPrinter*

printBegin ( )

printGroupBegin (…)

printPosition (…)

printGroupEnd (…)

printEnd (…)

OrderGroupChanger::run (…)

: Repository

getCutomer (…)

getCutomer (…)

*IRepository*

: RepositoryTraceProxy

getCutomer (…)

getCutomer (…)

*IRepository*

// RepositoryTraceProxy.h

// ...

class **RepositoryTraceProxy** : public IRepository {

private:

const IRepository\* **target\_**;

public:

**RepositoryTraceProxy**(const IRepository\* target) :

target\_(target) {

}

const Customer& **getCustomer**(int nr) const {

cout << ">> IRepoitory.getCustomer(" << nr << ")" << endl;

const Customer& customer = target\_->getCustomer(nr);

cout << "<< IRepoitory.getCustomer ==> Customer ["

<< customer.nr << ", "

<< customer.name << "]" << endl;

return customer;

}

const Product& **getProduct**(int nr) const {

cout << ">> IRepoitory.getProduct(" << nr << ")" << endl;

const Product& product = target\_->getProduct(nr);

cout << "<< IRepoitory.getProduct ==> Product ["

<< product.nr << ", "

<< product.name << ", "

<< product.price << "]" << endl;

return product;

}

};

// main.cpp

// ...

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

cout << "Refactoring-Proxy" << endl;

try {

ifstream ordersStream("orders.txt");

OrderReader orderReader(ordersStream);

ifstream customersStream("customers.txt");

ifstream productsStream("products.txt");

Repository repository(customersStream, productsStream);

customersStream.close();

productsStream.close();

ofstream out("result.txt");

Printer printer(out);

RepositoryTraceProxy p1(&repository);

OrderGroupChanger::run(orderReader, p1, printer);

// RepositoryTraceProxy p1(&repository);

// RepositoryTraceProxy p2(&p1);

// OrderGroupChanger::run(orderReader, p2, printer);

ordersStream.close();

out.close();

cout << "Done. See results.txt" << endl;

}

catch(const runtime\_error& e) {

cout << "Exception: " << e.what() << endl;

}

}