Einleitung

Trotz der derzeitigen explosionsartigen Vermehrung von Programmiersprachen ist C++ nach wie vor im Einsatz. Laut dem Tiobe-Index vom Juli 2013 handelt es sich dabei um die am vierthäufigsten verwendete Programmiersprache. (Den aktuellen Index finden Sie auf *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*.) Der ISO-Standard 2011 (ISO/IEC 14822:2011, besser bekannt als C++11) ergänzt C++ um weitere Merkmale, die die Akzeptanz dieser Sprache steigern – oder zumindest die Einwände gegen ihre Verwendung abschwächen.

C++ ist nach wie vor eine der besten Möglichkeiten, um hochleistungsfähige Lösungen zu erstellen. Wenn Sie die Produkte Ihres Unternehmens auf Ihrer eigenen Hardware einsetzen, dann verfügen Sie wahrscheinlich selbst über ein ziemlich großes System, das in C++ geschrieben wurde. Ist Ihr Unternehmen schon seit den 1990er-Jahren oder noch länger im Geschäft, dann haben Sie wahrscheinlich ein langjähriges C++-System, und die Chancen stehen nicht schlecht, dass es Ihnen auch in den nächsten Jahren noch erhalten bleibt.

Vorausgesetzt Sie arbeiten mit C++, kommen Ihnen jetzt vielleicht einige Einwände in den Sinn:

- Wir schreiben das Jahr 2014! Warum soll ich mich wieder mit dieser schwierigen (wenn auch unterhaltsamen) Sprache beschäftigen, die ich schon vor Jahren aufgegeben habe? Wie kann ich damit leben, ohne mir dabei selbst zu schaden?
- Ich bin ein erfahrener C++-Experte. Ich kenne diese leistungsfähige Sprache in- und auswendig und wende sie schon seit Jahren erfolgreich an. Warum sollte ich meine Arbeitsweise ändern?
- Wer bezahlt mich dafür, dass ich mich hiermit beschäftige?

Ich selbst habe Anfang der 1990er-Jahre zum ersten Mal mit C++ gearbeitet, bevor Dinge wie Templates (und Template-Metaprogrammierung!), RTTI (Run-Time Type Information), STL (Standard Template Library) und Boost aufkamen. Seitdem musste ich einige wenige Male zu dieser leistungsstarken Sprache zurückkehren, was mich immer ein kleines bisschen erschreckt hat. Wie bei jeder anderen Sprache ist es natürlich auch bei C++ möglich, sich ins eigene Fleisch zu

schneiden, aber dabei merken Sie es manchmal gar nicht, bis es zu spät ist. Außerdem wird der Schaden höchstwahrscheinlich größer sein als bei der Verwendung von anderen Sprachen.

Wenn Sie schon seit Jahren mit C++ arbeiten, haben Sie sich wahrscheinlich viele Idiome und Techniken angewöhnt, um für eine hohe Qualität des Codes zu sorgen. Eingefleischte C++-Veteranen gehören zu den sorgfältigsten Entwicklern, da große Vorsicht und Achtsamkeit erforderlich sind, um so lange mit C++ zu überleben.

Angesichts dessen könnte man denken, dass die Qualität des Codes von C++-Systemen sehr hoch sein sollte. Trotzdem weisen viele C++-Systeme dieselben Probleme auf, die wir unabhängig von der Sprache immer wieder vorfinden:

- Riesige Quellcodedateien mit Tausenden von Zeilen
- Memberfunktionen mit Hunderten oder Tausenden von Zeilen unverständlichen Codes
- Datenträger voll totem Quellcode
- Build-Zeiten von mehreren Stunden
- Hohe Anzahl an Fehlern
- Zu verzwickte Logik, um schnelle Korrekturen gefahrlos anzubringen
- Redundanter Code in Dateien, Klassen und Modulen
- Code durchsetzt mit längst überholten Programmiertechniken

Sind diese Mängel unvermeidlich? Nein! Mit testgetriebener Entwicklung können Sie die Entropie Ihres Systems bekämpfen! Sie können damit vielleicht sogar Ihre Begeisterung für die Entwicklungsarbeit wieder neu anfachen.

Wenn Sie einfach nur an der Bezahlung interessiert sind, dann gibt es für Sie genügend Arbeitgeber, die Sie als C++-Entwickler beschäftigen würden. C++ ist jedoch eine sehr technische und nuancenreiche Sprache. Ihr unachtsamer Gebrauch kann zu Mängeln, sporadischen Ausfällen und mehrtägigen Debugsitzungen führen – lauter Dinge, die Ihr Arbeitsverhältnis und Ihre Bezahlung gefährden. Auch hier kann sich testgetriebene Entwicklung als hilfreich erweisen.

Der Aufwand, der dazu erforderlich ist, den Funktionsumfang eines solchen umfangreichen, langjährigen C++-Systems zu erweitern, ist fast immer deprimierend und schwer einzuschätzen. Um einige wenige Codezeilen zu ändern, benötigen Sie oft Stunden oder gar Tage, um die entsprechende Passage zu verstehen. Die Produktivität wird noch weiter dadurch herabgesetzt, dass die Entwickler stundenlang warten müssen, bis endlich klar ist, ob die Änderungen kompiliert werden können, und es dauert noch länger, um zu sehen, ob sich diese Änderungen mit dem Rest des Systems vertragen.

Das muss aber nicht so sein. *Testgetriebene Entwicklung* (Test-Driven Development, TDD) ist eine in den späten 1990er-Jahren erfundene Technik für das Softwaredesign, mit der Sie Ihr C++-System niederzwingen und unter Kontrolle halten können, während Sie neue Funktionalität hinzufügen. Die Idee, erst die

Tests und dann den Code zu schreiben, besteht schon erheblich länger. Der formale TDD-Zyklus wurde jedoch von Ward Cunningham und Kent Beck erfunden – siehe »Test Driven Development: By Example« [Bec02].

Der Hauptzweck dieses Buches besteht darin, Ihnen eine disziplinierte Vorgehensweise für die Anwendung von TDD beizubringen. Sie werden dabei Folgendes lernen:

- Die grundlegenden Mechanismen von TDD
- Die möglichen Vorteile von TDD
- Designmängel mithilfe von TDD gleich im Ansatz bekämpfen
- Probleme und Kosten bei der Verwendung von TDD
- Verkürzen oder gar Vermeiden von Debugsitzungen dank TDD
- Pflegen von TDD

Eignet sich das auch für mich und mein System?

»Was soll dieser ganze Zirkus um Unit Tests? Das hilft mir doch alles nicht viel weiter!«

Vielleicht haben Sie sich schon einmal an Unit Tests versucht. Vielleicht haben Sie gerade jetzt damit zu kämpfen, Unit Tests für ein Altsystem zu schreiben. Angesichts dieser Erfahrungen mag es für Sie so aussehen, als eigne sich TDD für die seltenen Fälle, in denen jemand das Glück hat, an einem neuen System zu arbeiten. Löst es aber die Probleme, die sich bei Ihrer täglichen Arbeit an einem festgefahrenen, anspruchsvollen C++-System stellen?

TDD ist ein sehr nützliches Werkzeug, aber natürlich kein Allheilmittel für den Umgang mit Altsystemen. Zwar können Sie viele der neuen Features, die Sie dem System hinzufügen, mithilfe von TDD entwickeln, aber Sie müssen auch damit beginnen, den überflüssigen Code zu entfernen, der sich im Laufe der Jahre angesammelt hat. Zusätzliche Strategien und Taktiken für eine fortgesetzte Bereinigung sind erforderlich. Dazu müssen Sie Techniken zum Auflösen von Abhängigkeiten und für die gefahrlose Änderung von Code lernen, die Michael Feathers in seinem unverzichtbaren Buch »Effektives Arbeiten mit Legacy Code« [Fea04] vorstellt. Sie müssen wissen, wie Sie ein groß angelegtes Refactoring angehen, ohne entsprechende Probleme hervorzurufen. Dazu sollten Sie die Mikado-Methode kennenlernen (siehe »Behead Your Legacy Beast: Refactor and Restructure Relentlessly with the Mikado Method« [BE12]). In diesem Buch lernen Sie solche Hilfstechniken und mehr.

Es nützt gewöhnlich nicht viel, einfach Unit Tests für bereits bestehenden Code hinzuzufügen (was ich als TAD für *Test-After Development* bezeichne), da Sie dabei immer damit zu kämpfen haben, »dass das System so ist, wie es ist«. Sie können Tausende von Mannstunden in das Schreiben von Tests investieren, ohne die Qualität des Systems dadurch messbar zu steigern.

Wenn Sie TDD einsetzen, um Ihr System zu gestalten, dann wird Ihr Design definitionsgemäß testfähig. Sie werden dabei auch andere Designs entwickeln als ohne TDD, und zwar Designs, die in vieler Hinsicht besser sind. Je besser Sie verstehen, was gutes Design bedeutet, umso stärker wird Ihnen TDD dabei helfen, es zu erreichen.

Um Ihnen den Wechsel zu einer neuen Ansicht über Design zu erleichtern, betone ich in diesem Buch die Prinzipien für gutes Codemanagement, z.B. die SOLID-Prinzipien für objektorientiertes Design, die in »Agile Software Development, Principles, Patterns, and Practices« [Mar02] beschrieben werden. Ich zeige Ihnen, wie ein Gefühl für gutes Design die weiter gehende Entwicklung und Produktivität fördert und wie TDD aus Ihnen einen aufmerksamen Entwickler macht, der konsistentere und zuverlässigere Ergebnisse erzielt.

Zielgruppe

Dieses Buch ist für C++-Entwickler unabhängig von ihrem Kenntnisstand gedacht – für Anfänger, die gerade einmal ein grundlegendes Verständnis gewonnen haben, bis hin zu alten Hasen, die auch in die esoterischen Aspekte der Sprache eingeweiht sind. Wenn Sie längere Zeit nicht mehr mit C++ gearbeitet haben, werden Sie feststellen, dass Sie sich durch die schnellen Feedbackzyklen von TDD schnell wieder daran gewöhnen werden.

Der Hauptzweck dieses Buches besteht zwar darin, Ihnen TDD beizubringen, doch werden Sie auch unabhängig von Ihren TDD-Kenntnissen viel Wertvolles darin finden. Wenn Ihnen das Prinzip, Unit Tests für Ihren Code zu schreiben, noch völlig unbekannt ist, lernen Sie hier Schritt für Schritt die Grundlagen von TDD kennen. Wenn für Sie TDD noch relativ neu ist, werden Sie überall in diesem Buch viel nützliches Fachwissen entdecken, das Ihnen auf einfache Weise durch praktische Beispiele nahegebracht wird. Aber auch erfahrene TDD-Entwickler werden hier einen reichen Schatz an Kenntnissen finden, ein fundierteres theoretisches Fundament für die praktische Anwendung erhalten sowie neue Aspekte kennenlernen, die eine genauere Untersuchung wert sind.

Sollten Sie noch skeptisch sein, können Sie sich TDD hier aus verschiedenen Blickwinkeln ansehen. Ich werde Ihnen immer wieder zeigen, warum TDD meiner Meinung nach gut funktioniert, und ich werde Ihnen auch meine Erfahrungen mit den Projekten weitergeben, in denen es nicht gut funktioniert hat, und die Gründe dafür nennen. Dieses Buch ist keine Werbebroschüre, sondern eine mit offenen Augen durchgeführte Untersuchung einer umwälzenden Technik.

Unabhängig von Ihrem Hintergrund werden Sie auch Hinweise dazu finden, wie Sie TDD in Ihrem Team pflegen und erhalten können. Es ist leicht, mit TDD anzufangen, aber Ihr Team wird unterwegs auf anspruchsvolle Herausforderungen stoßen. Wie können Sie es vermeiden, dass Ihr Wechsel zu TDD durch diese Probleme aus der Bahn geworfen wird? Wie verhindern Sie solche Katastrophen?

In Kapitel 11 stelle ich einige Ideen vor, die nach meiner Erfahrung gut funktioniert haben.

Was brauchen Sie noch?

Um den Code der Beispiele in diesem Buch zu schreiben, brauchen Sie natürlich einen Compiler und ein Unit-Test-Framework. Für einige Beispiele sind auch Bibliotheken von Drittanbietern erforderlich. Nachfolgend sehen wir uns diese drei Elemente an. Weitere Einzelheiten darüber erfahren Sie in Kapitel 1.

Das Unit-Test-Framework

Unter den Dutzenden von verfügbaren Unit-Test-Frameworks für C++ habe ich mich bei den meisten Beispielen in diesem Buch für Google Mock entschieden (das auf Google Test aufbaut). Bei einer Websuche nach einem solchen Framework erhält Google Mock zurzeit die meisten Treffer. Hauptsächlich aber habe ich es gewählt, da es die Hamcrest-Schreibweise unterstützt (eine Form von Zusicherungen auf der Grundlage von Matchern, mit der Sie sehr aussagekräftige Tests schreiben können). Anhand der Informationen in Kapitel 1 können Sie sich schnell in Google Mock einarbeiten.

Dieses Buch ist jedoch weder eine umfangreiche Abhandlung über Google Mock noch eine Werbebroschüre dafür, sondern es soll Ihnen TDD beibringen. Sie lernen aber genug über Google Mock, um TDD darin wirksam einsetzen zu können.

Für einige der Beispiele werden Sie auch ein anderes Unit-Test-Framework verwenden, nämlich CppUTest. Sie werden feststellen, dass es ziemlich einfach ist, sich mit einem weiteren solchen Framework vertraut zu haben. Das sollte bei Ihnen auch jegliche Bedenken ausräumen, wenn Sie etwas anderes verwenden als Google Mock oder CppUTest.

Wenn Sie schon ein anderes Framework im Einsatz haben, etwa CppUnit oder Boost.Test, dann machen Sie sich keine Sorgen. In der prinzipiellen Verwendung und oft auch in der Implementierung weisen sie viele Ähnlichkeiten mit Google Mock auf. Sie können den Ausführungen problemlos folgen und die TDD-Beispiele in praktisch jedem anderen der verfügbaren C++-Unit-Test-Frameworks ausprobieren. Worauf Sie bei der Auswahl eines solchen Frameworks unbedingt achten sollten, erfahren Sie in Anhang A.

Für Mocks und Stubs (siehe Kap. 5) wird in den meisten Beispielen in diesem Buch Google Mock verwendet. Natürlich arbeiten Google Mock und Google Test zusammen, aber Sie können Google Mock auch mit einem anderen Unit-Test-Framework erfolgreich einsetzen.

Der Compiler

Sie benötigen einen Compiler mit Unterstützung für C++11. Der Beispielcode zu diesem Buch wurde ursprünglich in gcc geschrieben und funktioniert ohne Anpassungen unter Linux und Mac OS. Wie Sie die Beispiele auch unter Windows zum Laufen bekommen, erfahren Sie in Kapitel 1. In allen Beispielen wird STL verwendet, ein wesentlicher Aspekt der modernen C++-Entwicklung für viele Plattformen.

Bibliotheken von Drittanbietern

In einigen der Beispiele werden frei erhältliche Bibliotheken von Drittanbietern verwendet. Eine Liste der Bibliotheken zum Download finden Sie in Kapitel 1.

Wie Sie dieses Buch lesen sollten

Die Kapitel in diesem Buch habe ich als möglichst eigenständige Abschnitte gestaltet. Somit können einzelne Kapitel herausgegriffen und durchgearbeitet werden, ohne alle anderen Kapitel lesen zu müssen. Wo es angebracht ist, habe ich Querverweise verwendet, sodass Sie korrespondierende Stellen auf einfache Weise finden können.

Jedes Kapitel beginnt mit einem kurzen Überblick und endet mit einer Zusammenfassung sowie einer Vorschau auf das nächste Kapitel. Als Namen dieser Abschnitte habe ich die Bezeichnungen für die Initialisierungs- und Aufräumphasen verwendet, wie sie in vielen Unit-Test-Frameworks gebraucht werden: Setup und Teardown.

Der Quellcode

Das Buch enthält zahlreiche Codebeispiele. Meistens ist ein Dateiname dazu angegeben. Den kompletten Beispielcode zu diesem Buch finden Sie auf *http://pragprog.com/book/lotdd/modern-c-programming-with-test-drivendevelopment* sowie auf meiner GitHub-Seite *http://github.com/jlangr*.

Innerhalb der Codesammlung sind die Beispiele nach Kapiteln geordnet, und innerhalb der Verzeichnisse für die einzelnen Kapitel finden Sie wiederum Verzeichnisse, die mit einer Versionsnummer versehen sind. (Dadurch ist es möglich, die fortschreitende Veränderung des Codes in den einzelnen Kapiteln zu zeigen.) Beispielsweise enthält das Listing mit der Bezeichnung *c2/7/SoundexTest.cpp* den Quellcode der Datei `SoundexTest.cpp`, die sich in der siebten Überarbeitung im Codeverzeichniss für Kapitel 2 (`c2`) befindet.

Diskutieren Sie über das Buch!

Bitte beteiligen Sie sich (in englischer Sprache) an dem Diskussionsforum auf *https://groups.google.com/forum/?fromgroups#!forum/modern-cpp-with-tdd*. Dieses Forum dient dazu, das Buch sowie TDD im Allgemeinen zu besprechen. Ich werde dort auch wichtige Informationen über dieses Buch einstellen.

Für TDD-Neulinge: Der Inhalt dieses Buches

Dieses Buch richtet sich zwar an alle, insbesondere aber an Entwickler, für die TDD noch etwas Neues ist. Die Anordnung der Kapitel ist eigens auf diese Zielgruppe zugeschnitten. Ich empfehle Ihnen sehr, die Übung in Kapitel 2, »Testgetriebene Entwicklung: Ein erstes Beispiel«, durchzuarbeiten. Dadurch erhalten Sie ein gutes Gefühl für die Ideen, die hinter TDD stehen. Lesen Sie das Kapitel nicht nur, sondern geben Sie den Code ein und sorgen Sie dafür, dass die Tests auch bestanden werden, wenn sie es sollen.

Die nächsten beiden Kapitel, »Testgetriebene Entwicklung: Grundlagen« (Kap. 3) und »Tests konstruieren« (Kap. 4), stellen ebenfalls grundlegende Lektüre dar. Sie zeigen, was TDD ist (und was es nicht ist) und wie Sie Ihre Tests aufbauen. Machen Sie sich mit dem Inhalt in diesen Kapiteln gut vertraut, bevor Sie zu Mocks übergehen (in Kap. 5, »Testdoubles«), einer Technik, die für die meisten Produktionssysteme unverzichtbar ist.

Das Kapitel über Design und Refactoring (Kap. 6, »Inkrementelles Design«) sollten Sie auf keinen Fall überspringen, auch dann nicht, wenn Sie schon zu wissen glauben, was inkrementelles Design bedeutet. Ein wesentlicher Grund für die Anwendung von TDD besteht darin, das Design ständig weiterentwickeln und den Code durch Refactorings kontinuierlich verbessern zu können. Die meisten Systeme weisen ein schlechtes Design und komplizierten Code auf, was teilweise daran liegt, dass die Entwickler nicht zu einem ausreichenden Refactoring bereit sind oder nicht wissen, wie das geht. Sie lernen hier, was dafür ausreichend ist und wie Sie die Vorteile kleiner, einfacher Systeme ausnutzen können.

Zum Abschluss der TDD-Techniken sehen wir uns in Kapitel 7, »Qualitativ hochwertige Tests«, eine Reihe von Möglichkeiten an, mit denen sich Ihre Investitionen in TDD noch stärker auszahlen. Einige dieser Techniken können den Unterschied zwischen bloßem Überleben und erfolgreicher Anwendung von TDD ausmachen.

Zwangsläufig werden Sie irgendwann mit einem bereits vorhandenen System zu tun haben, das nicht mithilfe von TDD entwickelt wurde. Einen Schnellkurs in Techniken zur Arbeit mit solchem Altcode bietet Kapitel 8, »Herausforderungen durch Legacy-Code«.

Darauf folgt Kapitel 9, »TDD für Threads«, in dem es um die Entwicklung von Multithread-Anwendungen mit TDD geht. Die testgetriebene Vorgehensweise bietet hier einige Überraschungen für Sie.

Kapitel 10, »Weitere Aspekte von TDD«, behandelt einige spezielle Gebiete und Probleme von TDD ausführlicher. Hier erläutern wir einige moderne Ideen zu TDD sowie alternative Vorgehensweisen, die ein wenig von dem abweichen, was Sie sonst in diesem Buch gelesen haben.

Am Ende wollen Sie wahrscheinlich erfahren, wie Sie TDD in Ihrem Team einführen und dafür sorgen können, dass sich Ihre Investitionen darin auch langfristig auszahlen. In Kapitel 11, »Wachstum und Pflege von TDD«, finden Sie schlussendlich einige Ideen, die Sie in Ihrem Unternehmen umsetzen können.

Für Leser mit TDD-Erfahrung

Es ist durchaus möglich, sich nach Belieben einzelne Kapitel herauszugreifen. Allerdings sind überall in diesem Buch viele unter großen Mühen erworbene Weisheiten verstreut.

Schreibweisen in diesem Buch

Alle Codeabschnitte sind vom Rest des Texts abgesetzt. Wenn im Text Codeelemente erwähnt werden, so werden dafür folgende Schreibweisen verwendet:

- `Klassennamen` und `Testnamen` erscheinen in nicht proportionaler Schrift und in UpperCamelCase-Schreibweise.
- Auch alle anderen Codeelemente werden in nicht proportionaler Schrift dargestellt:
  - `funktionsnamen()` – Selbst wenn eine Funktion eine oder mehrere Parameter deklariert, wird sie mit leerer Argumentliste geschrieben. Memberfunktionen bezeichne ich manchmal auch als *Methoden*.
  - `variablennamen`
  - `schlüsselwörter`
  - Alle anderen Codebeispiele

Zur Vereinfachung und um kein Papier zu verschwenden, wird in den Listings häufig Code ausgelassen, der mit dem besprochenen Thema nichts zu tun hat. Stattdessen sehen Sie einen Kommentar mit Auslassungspunkten. Beispielsweise wird im folgenden Code der Rumpf der `for`-Schleife auf diese Weise ersetzt:

```
for (int i = 0; i < count; i++) {
   // ...
}
```

Wer ist »wir«?

Ich habe dieses Buch als Dialog zwischen Ihnen und mir geschrieben. Im Allgemeinen spreche ich Sie als Leser an. Wenn ich von mir selbst spreche, dann geht es gewöhnlich um meine persönlichen Erfahrungen, Meinungen oder Vorlieben. Dadurch mache ich deutlich, dass meine Äußerung keine allgemein anerkannte Regel ist (aber trotzdem eine gute Idee sein kann!).

Wenn es ans Schreiben der Programme geht, möchte ich nicht, dass Sie sich allein fühlen, vor allem da Sie ja noch lernen. Bei all den Codebeispielen in diesem Buch arbeiten *wir* zusammen.

Wer bin ich?

Ich programmiere seit 1980, als ich noch an der High School war, und beruflich seit 1982 (ich habe schon für die University of Maryland gearbeitet, während ich noch meinen Bachelor in Informatik machte). Im Jahr 2000 wechselte ich vom Entwickler zum Berater, wobei ich das Vergnügen hatte, für Bob Martin und gelegentlich für andere großartige Personen bei Object Mentor zu arbeiten.

2003 habe ich Langr Software Solutions für die Beratung und Schulung in der agilen Softwareentwicklung gegründet. Meistens wende ich zusammen mit anderen Entwicklern als Paar TDD an oder ich lehre diese Technik. Mir ist es sehr wichtig, zwischen meiner Beratungs- und Lehrtätigkeit und »echter« Entwicklungstätigkeit in einem richtigen Entwicklungsteam abzuwechseln, sodass ich immer auf dem neuesten Stand bleibe. Seit 2002 war ich als Vollzeitentwickler jeweils lange Zeit für vier verschiedene Unternehmen tätig.

Ich schreibe gern über Softwareentwicklung. Das gehört für mich dazu, um selbst tiefere Kenntnisse zu gewinnen, aber ich freue mich auch darüber, anderen zu helfen, guten Code zu schreiben. Dies ist mein viertes Buch. Zuvor habe ich »Essential Java Style: Patterns for Implementation« [Lan99] und »Agile Java: Crafting Code With Test-Driven Development« [Lan05] sowie als Koautor »Agile in a Flash« [OL11] zusammen mit Tim Ottinger geschrieben. Des Weiteren habe ich einige Kapitel zu »Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code« [Mar09] von »Uncle Bob« Martin beigetragen. Ich habe über hundert Artikel für andere Websites als meine eigene geschrieben, führe regelmäßig mein eigenes Blog (auf *http://langrsoft.com/jeff*) und habe über hundert Blogeinträge für das Projekt »Agile in a Flash« auf *http://agileinaflash.com* verfasst oder mitverfasst.

Neben C++ habe ich ausführlich in mehreren anderen Sprachen programmiert: Java, Smalltalk, C, C# und Pascal sowie einige andere, die hier unerwähnt bleiben sollen. Zurzeit lerne ich Erlang und kann auch genügend Python und Ruby, um zu überleben. Außerdem habe ich mindestens ein weiteres Dutzend Sprachen ausprobiert, um zu sehen, wie sie sind (oder um ein kurzfristiges Projekt zu unterstützen).

Der C++-Stil in diesem Buch

Ich habe zwar umfassende Erfahrungen mit C++-Systemen aller Größen, aber für einen Sprachexperten halte ich mich trotzdem nicht. Ich habe die wichtigen Bücher von Meyers und Sutter sowie einige andere gelesen, und ich weiß, wie ich C++ für meine Zwecke einsetzen kann und wie ich den resultierenden Code aussagekräftig und wartungsfreundlich gestalte. Darüber hinaus bin ich mir auch der meisten nur für Eingeweihte bekannten Aspekte der Sprache bewusst, vermeide aber absichtlich Lösungen, für die sie erforderlich sind. Meine Definition von »cleverer Programmierung« bedeutet »schwer zu warten«. Ich möchte Sie in eine bessere Richtung führen.

Mein C++-Stil ist sehr stark objektorientiert (was zweifellos auf die viele Programmierarbeit in Smalltalk, Java und C# zurückgeht). Ich bevorzuge Code mit dem Gültigkeitsbereich einer Klasse. Die meisten Beispiele in diesem Buch folgen diesem Stil. Beispielsweise erstelle ich den Soundex-Code im ersten Beispiel (siehe Kap. 2) als Klasse, obwohl das nicht sein müsste. Ich mag es einfach so. Wenn es Sie stört, können Sie es auf Ihre Weise machen.

TDD bietet unabhängig von Ihrem C++-Stil viel Nutzen. Lassen Sie sich also durch meinen Stil nicht daran hindern, das wahre Potenzial dieser Technik auszuschöpfen. Die stärkere Betonung auf Objektorientiertheit vereinfacht jedoch die Einführung von Testdoubles (siehe Kap. 5), wenn Sie problematische Abhängigkeiten auflösen müssen. Wenn Sie sich in TDD vertiefen, werden Sie feststellen, dass sich Ihr Stil mit der Zeit in diese Richtung bewegt. Und das ist keine schlechte Sache!

Leider bin ich ein bisschen faul. Angesichts des geringen Umfangs der Beispiele habe ich die Verwendung von Namespaces auf das Minimum beschränkt. Für Produktionscode sollten Sie natürlich auf Namespaces zurückgreifen.

Außerdem gestalte ich meinen Code gern so stromlinienförmig wie möglich und vermeide daher das, was ich als optisch ablenkend empfinde. In den meisten Implementierungsdateien finden Sie daher `using namespace std;`, was viele für eine schlechte Vorgehensweise halten. (Dadurch, dass wir die Klassen und Funktionen klein und zielgerichtet halten, sind solche und ähnliche Richtlinien wie »alle Funktionen sollten nur einen Rückgabewert haben« weniger nützlich.) Aber keine Sorge: TDD hindert Sie nicht daran, bei Ihren eigenen Standards zu bleiben, und ich werde es auch nicht tun.

Noch ein letztes Wort zu C++: Es ist eine umfangreiche Sprache. Ich bin mir sicher, dass es bessere Möglichkeiten gibt, den Code für einige der Beispiele in diesem Buch zu schreiben, und ich wette, dass es Bibliothekskonstrukte gibt, die ich nicht genutzt habe. Das Schöne an TDD ist, dass Sie eine Implementierung auf Dutzende Weisen umarbeiten können, ohne Gefahr zu laufen, etwas kaputt zu machen. Senden Sie mir trotzdem Ihre Verbesserungsvorschläge – aber nur, wenn Sie bereit sind, sie testgesteuert zu entwickeln!

Danksagungen

Ich danke meinem Lektor Michael Swaine und den großartigen Leuten bei PragProg für die Anleitung und die erforderlichen Ressourcen, um dieses Buch fertigzustellen.

Danke, Onkel Bob, für das überschwängliche Vorwort!

Vielen Dank an Dale Stewart, meinen Fachgutachter, für die wertvolle Unterstützung, vor allem für die Rückmeldung und die Hilfe bei dem C++-Code im ganzen Buch.

Beim Schreiben habe ich immer um gnadenlos ehrliche Rückmeldung gebeten, und Bas Vodde hat mir genau das gegeben und mich mit umfangreicher Kritik zum gesamten Buch versorgt. Er war der unsichtbare Partner, den ich brauchte, um ehrlich zu mir selbst zu sein.

Mein besonderer Dank gilt Joe Miller, der die meisten Beispiele akribisch konvertiert hat, sodass sie unter Windows erstellt und ausgeführt werden können.

Vielen Dank auch an alle anderen Personen, die Ideen und unschätzbares Feedback zu diesem Buch beigesteuert haben: Steve Andrews, Kevin Brothaler, Marshall Clow, Chris Freeman, George Dinwiddie, James Grenning, Michael Hill, Jeff Hoffman, Ron Jeffries, Neil Johnson, Chisun Joung, Dale Keener, Bob Koss, Robert C. Martin, Paul Nelson, Ken Oden, Tim Ottinger, Dave Rooney, Tan Yeong Sheng, Peter Sommerlad und Zhanyong Wan. Ich bitte um Entschuldigung, falls ich irgendjemand nicht erwähnt haben sollte.

Ein Dankeschön auch an all diejenigen, die mir auf der Errata-Seite von PragProg Rückmeldung gegeben haben: Bradford Baker, Jim Barnett, Travis Beatty, Kevin Brown, Brett DiFrischia, Jared Grubb, David Pol, Bo Rydberg, Jon Seidel, Marton Suranyi, Curtis Zimmerman und viele andere.

Noch einmal Dank an Tim Ottinger, der einige der Worte in der Einleitung sowie einige Ideen zu diesem Buch beigetragen hat. Ich habe dich als Mitverschwörer vermisst!

Danke an all diejenigen, die geholfen haben, dieses Buch besser zu machen, als ich es jemals allein hätte schaffen können!

Widmung

Dieses Buch ist all denen gewidmet, die mich fortwährend dabei unterstützen, das zu tun, was ich gern tue, insbesondere meiner Frau Kathy.