/THEORY/IN/PRACTICE

Masterminds of Programming

编程大师智慧（影印版）

与主流编程语言创造者的对话

Conversations with the Creators of Major Programming Languages

Federico Biancuzzi
Shane Warden 编

O'REILLY®
東南大學出版社

编程大师智慧(影印版)

Masterminds of Programming

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

编程大师智慧：英文/（美）比安库利（Biancuzzi, F.），（美）沃登（Warden, S.）著. —影印本. —南京：东南大学出版社，2010.6

书名原文：Masterminds of Programming

ISBN 978-7-5641-2262-1

Ⅰ. ①编… Ⅱ. ①比… ②沃…Ⅲ. ①程序设计－英文 Ⅳ. ① TP311.1

中国版本图书馆CIP数据核字（2010）第089084号

江苏省版权局著作权合同登记

图字：10-2010-163号

编程大师智慧（影印版）

出版发行：东南大学出版社

地　　址：南京四牌楼2号　　邮编：210096

出 版 人：江　汉

网　　址：http://press.seu.edu.cn

电子邮件：press@seu.edu.cn

印　　刷：扬中市印刷有限公司

开　　本：787毫米×980毫米　16开本

印　　张：31印张

字　　数：607千字

版　　次：2010年6月第1版

印　　次：2010年6月第1次印刷

书　　号：ISBN 978-7-5641-2262-1

印　　数：1~2000册

定　　价：68.00元（册）

本社图书若有印装质量问题，请直接与读者服务部联系。电话（传真）：025-83792328

Foreword

PROGRAMMING LANGUAGE DESIGN IS A FASCINATING TOPIC. There are so many programmers who think they can design a programming language better than one they are currently using; and there are so many researchers who believe they can design a programming language better than any that are in current use. Their beliefs are often justified, but few of their designs ever leave the designer's bottom drawer. You will not find them represented in this book.

Programming language design is a serious business. Small errors in a language design can be conducive to large errors in an actual program written in the language, and even small errors in programs can have large and extremely costly consequences. The vulnerabilities of widely used software have repeatedly allowed attack by malware to cause billions of dollars of damage to the world economy. The safety and security of programming languages is a recurrent theme of this book.

Programming language design is an unpredictable adventure. Languages designed for universal application, even when supported and sponsored by vast organisations, end up sometimes in just a niche market. In contrast, languages designed for limited or local use can win a broad clientele, sometimes in environments and for applications that their designers never dreamed of. This book concentrates on languages of the latter kind.

These successful languages share a significant characteristic: each of them is the brainchild of a single person or a small team of like-minded enthusiasts. Their designers are masterminds of programming; they have the experience, the vision, the energy, the persistence, and the sheer genius to drive the language through its initial implementation, through its evolution in the light of experience, and through its standardisation by usage (de facto) and by committee (de jure).

In this book the reader will meet this collection of masterminds in person. Each of them has granted an extended interview, telling the story of his language and the factors that lie behind its success. The combined role of good decisions and good luck is frankly acknowledged. And finally, the publication of the actual words spoken in the interview gives an insight into the personality and motivations of the designer, which is as fascinating as the language design itself.

—*Sir Tony Hoare*

Sir Tony Hoare, winner of an ACM Turing Award and a Kyoto Award, has been a leader in research into computing algorithms and programming languages for 50 years. His first academic paper, written in 1969, explored the idea of proving the correctness of programs, and suggested that a goal of programming language design was to make it easier to write correct programs. He is delighted to see the idea spread gradually among programming language designers.

Preface

WRITING SOFTWARE IS HARD—AT LEAST, WRITING SOFTWARE THAT STANDS UP UNDER TESTS, TIME, and different environments is hard. Not only has the software engineering field struggled to make writing software easier over the past five decades, but languages have been designed to make it easier. But what makes it hard in the first place?

Most of the books and the papers that claim to address this problem talk about architecture, requirements, and similar topics that focus on the *software*. What if the hard part was in the *writing*? To put it another way, what if we saw our jobs as programmers more in terms of communication—*language*—and less in terms of engineering?

Children learn to talk in their first years of life, and we start teaching them how to read and write when they are five or six years old. I don't know any great writer who learned to read and write as an adult. Do you know any great programmer who learned to program late in life?

And if children can learn foreign languages much more easily than adults, what does this tell us about learning to program—an activity involving a new language?

Imagine that you are studying a foreign language and you don't know the name of an object. You can describe it with the words that you know, hoping someone will understand what you mean. Isn't this what we do every day with software? We describe the object we have in our mind with a programming language, hoping the description will be clear enough to the compiler or interpreter. If something doesn't work, we bring up the picture again in our mind and try to understand what we missed or misdescribed.

With these questions in mind, I chose to launch a series of investigations into why a programming language is created, how it's technically developed, how it's taught and learned, and how it evolves over time.

Shane and I had the great privilege to let 27 great designers guide us through our journey, so that we have been able to collect their wisdom and experience for you.

In *Masterminds of Programming*, you will discover some of the thinking and steps needed to build a successful language, what makes it popular, and how to approach the current problems that its programmers are facing. So if you want to learn more about successful programming language design, this book surely can help you.

If you are looking for inspiring thoughts regarding software and programming languages, you will need a highlighter, or maybe two, because I promise that you will find plenty of them throughout these pages.

—Federico Biancuzzi

Organization of the Material

The chapters in this book are ordered to provide a varied and provocative perspective as you travel through it. Savor the interviews and return often.

Chapter 1, C++, interviews Bjarne Stroustrup.

Chapter 2, *Python*, interviews Guido van Rossum.

Chapter 3, *APL*, interviews Adin D. Falkoff.

Chapter 4, *Forth*, interviews Charles H. Moore.

Chapter 5, *BASIC*, interviews Thomas E. Kurtz.

Chapter 6, *AWK*, interviews Alfred Aho, Peter Weinberger, and Brian Kernighan.

Chapter 7, *Lua*, interviews Luiz Henrique de Figueiredo and Roberto Ierusalimschy.

Chapter 8, *Haskell*, interviews Simon Peyton Jones, Paul Hudak, Philip Wadler, and John Hughes.

Chapter 9, *ML*, interviews Robin Milner.

Chapter 10, *SQL*, interviews Don Chamberlin.

Chapter 11, *Objective-C*, interviews Tom Love and Brad Cox.

Chapter 12, *Java*, interviews James Gosling.

Chapter 13, *C#*, interviews Anders Hejlsberg.

Chapter 14, *UML*, interviews Ivar Jacobson, James Rumbaugh, and Grady Booch.

Chapter 15, *Perl*, interviews Larry Wall.

Chapter 16, *PostScript*, interviews Charles Geschke and John Warnock.

Chapter 17, *Eiffel*, interviews Bertrand Meyer.

Contributors lists the biographies of all the contributors.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, filenames, and utilities.

`Constant width`
: Indicates the contents of computer files and generally anything found in programs.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596515171

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

CONTENTS

CHAPTER ONE

C++

C++ occupies an interesting space among languages: it is built on the foundation of C, incorporating object-orientation ideas from Simula; standardized by ISO; and designed with the mantras "you don't pay for what you don't use" and "support user-defined and built-in types equally well." Although popularized in the 80s and 90s for OO and GUI programming, one of its greatest contributions to software is its pervasive generic programming techniques, exemplified in its Standard Template Library. Newer languages such as Java and C# have attempted to replace C++, but an upcoming revision of the C++ standard adds new and long-awaited features. Bjarne Stroustrup is the creator of the language and still one of its strongest advocates.

Design Decisions

Why did you choose to extend an existing language instead of creating a new one?

Bjarne Stroustrup: When I started—in 1979—my purpose was to help programmers build systems. It still is. To provide genuine help in solving a problem, rather than being just an academic exercise, a language must be complete for the application domain. That is, a non-research language exists to solve a problem. The problems I was addressing related to operating system design, networking, and simulation. I—and my colleagues—needed a language that could express program organization as could be done in Simula (that's what people tend to call object-oriented programming), but also write efficient low-level code, as could be done in C. No language that could do both existed in 1979, or I would have used it. I didn't particularly want to design a new programming language; I just wanted to help solve a few problems.

Given that, building on an existing language makes a lot of sense. From the base language, you get a basic syntactic and semantic structure, you get useful libraries, and you become part of a culture. Had I not built on C, I would have based C++ on some other language. Why C? I had Dennis Ritchie, Brian Kernighan, and other Unix greats just down (or across) the hall from me in Bell Labs' Computer Science Research Center, so the question may seem redundant. But it was a question I took seriously.

In particular, C's type system was informal and weakly enforced (as Dennis Ritchie said, "C is a strongly typed, weakly checked language"). The "weakly checked" part worried me and causes problems for C++ programmers to this day. Also, C wasn't the widely used language it is today. Basing C++ on C was an expression of faith in the model of computation that underlies C (the "strongly typed" part) and an expression of trust in my colleagues. The choice was made based on knowledge of most higher-level programming languages used for systems programming at the time (both as a user and as an implementer). It is worth remembering that this was a time when most work "close to the hardware" and requiring serious performance was still done in assembler. Unix was a major breakthrough in many ways, including its use of C for even the most demanding systems programming tasks.

So, I chose C's basic model of the machine over better-checked type systems. What I really wanted as the framework for programs was Simula's classes, so I mapped those into the C model of memory and computation. The result was something that was extremely expressive and flexible, yet ran at a speed that challenged assembler without a massive runtime support system.

Why did you choose to support multiple paradigms?

Bjarne: Because a combination of programming styles often leads to the best code, where "best" means code that most directly expresses the design, runs faster, is most maintainable, etc. When people challenge that statement, they usually do so by either defining their favorite programming style to include every useful construct (e.g., "generic programming is simply a form of OO") or excluding application areas (e.g., "everybody has a 1GHz, 1GB machine").

Java focuses solely on object-oriented programming. Does this make Java code more complex in some cases where C++ can instead take advantage of generic programming?

Bjarne: Well, the Java designers—and probably the Java marketers even more so—emphasized OO to the point where it became absurd. When Java first appeared, claiming purity and simplicity, I predicted that if it succeeded Java would grow significantly in size and complexity. It did.

For example, using casts to convert from `Object` when getting a value out of a container (e.g., `(Apple)c.get(i)`) is an absurd consequence of not being able to state what type the objects in the container is supposed have. It's verbose and inefficient. Now Java has generics, so it's just a bit slow. Other examples of increased language complexity (helping the programmer) are enumerations, reflection, and inner classes.

The simple fact is that complexity will emerge somewhere, if not in the language definition, then in thousands of applications and libraries. Similarly, Java's obsession with putting every algorithm (operation) into a class leads to absurdities like classes with no data consisting exclusively of static functions. There are reasons why math uses `f(x)` and `f(x,y)` rather than `x.f()`, `x.f(y)`, and `(x,y).f()`—the latter is an attempt to express the idea of a "truly object-oriented method" of two arguments and to avoid the inherent asymmetry of `x.f(y)`.

C++ addresses many of the logical as well as the notational problems with object orientation through a combination of data abstraction and generic programming techniques. A classical example is `vector<T>` where `T` can be any type that can be copied—including built-in types, pointers to OO hierarchies, and user-defined types, such as strings and complex numbers. This is all done without adding runtime overheads, placing restrictions on data layouts, or having special rules for standard library components. Another example that does not fit the classical single-dispatch hierarchy model of OO is an operation that requires access to two classes, such as `operator*(Matrix,Vector)`, which is not naturally a "method" of either class.

One fundamental difference between C++ and Java is the way pointers are implemented. In some ways, you could say that Java doesn't have real pointers. What differences are there between the two approaches?

Bjarne: Well, of course Java has pointers. In fact, just about everything in Java is implicitly a pointer. They just call them *references*. There are advantages to having pointers implicit as well as disadvantages. Separately, there are advantages to having true local objects (as in C++) as well as disadvantages.

C++'s choice to support stack-allocated local variables and true member variables of every type gives nice uniform semantics, supports the notion of value semantics well, gives compact layout and minimal access costs, and is the basis for C++'s support for general resource management. That's major, and Java's pervasive and implicit use of pointers (aka references) closes the door to all that.

Consider the layout tradeoff: in C++ a `vector<complex>(10)` is represented as a handle to an array of 10 complex numbers on the free store. In all, that's 25 words: 3 words for the vector, plus 20 words for the complex numbers, plus a 2-word header for the array on the free store (heap). The equivalent in Java (for a user-defined container of objects of user-defined types) would be 56 words: 1 for the reference to the container, plus 3 for the container, plus 10 for the references to the objects, plus 20 for the objects, plus 24 for the free store headers for the 12 independently allocated objects. Obviously, these numbers are approximate because the free store (heap) overhead is implementation defined in both languages. However, the conclusion is clear: by making references ubiquitous and implicit, Java may have simplified the programming model and the garbage collector implementation, but it has increased the memory overhead dramatically—and increased the memory access cost (requiring more indirect accesses) and allocation overheads proportionally.

What Java doesn't have—and good for Java for that—is C and C++'s ability to misuse pointers through pointer arithmetic. Well-written C++ doesn't suffer from that problem either: people use higher-level abstractions, such as iostreams, containers, and algorithms, rather than fiddling with pointers. Essentially all arrays and most pointers belong deep in implementations that most programmers don't have to see. Unfortunately, there is also lots of poorly written and unnecessarily low-level C++ around.

There is, however, an important place where pointers—and pointer manipulation—is a boon: the direct and efficient expression of data structures. Java's references are lacking here; for example, you can't express a `swap` operation in Java. Another example is simply the use of pointers for low-level direct access to (real) memory; for every system, some language has to do that, and often that language is C++.

The "dark side" of having pointers (and C-style arrays) is of course the potential for misuse: buffer overruns, pointers into deleted memory, uninitialized pointers, etc. However, in well-written C++ that is not a major problem. You simply don't get those problems with pointers and arrays used within abstractions (such as `vector`, `string`, `map`, etc.). Scoped resource management takes care of most needs; smart pointers and specialized handles can be used to deal with most of the rest. People whose experience is primarily C or old-style C++ find this hard to believe, but scope-based resource management is an immensely powerful tool and user-defined with suitable operations can address classical problems with less code than the old insecure hacks. For example, this is the simplest form of the classical buffer overrun and security problem:

```
char buf[MAX_BUF];
gets(buf); // Yuck!
```

Use a standard library string and the problem goes away:

```
string s;
cin >> s;    // read whitespace separated characters
```

These are obviously trivial examples, but suitable "strings" and "containers" can be crafted to meet essentially all needs, and the standard library provides a good set to start with.