Practical C++ Design

From Programming to Architecture

Adam B. Singer

Apress®

Practical C++ Design: From Programming to Architecture

Adam B. Singer
The Woodlands, Texas, USA

ISBN-13 (pbk): 978-1-4842-3056-5 ISBN-13 (electronic): 978-1-4842-3057-2

DOI 10.1007/978-1-4842-3057-2

Library of Congress Control Number: 2017954981

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Michael Thomas
Coordinating Editor: Mark Powers
Copy Editor: Mary Behr

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484230565. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

For Terri, Caroline, and Rebecca

Table of Contents

About the Author

Adam B. Singer graduated first in his class at the Georgia Institute of Technology in 1999 with a bachelor's degree in chemical engineering. He subsequently attended the Massachusetts Institute of Technology on a National Defense, Science, and Engineering Graduate Fellowship. He graduated from MIT with a Ph.D. in chemical engineering in 2004 after defending his thesis titled "Global Dynamic Optimization." Since graduation, Adam has been a member of the research and engineering staff at the ExxonMobil Upstream Research Company,[1] where he has worked in software development, design, and project management in areas such as optimization, reservoir simulation, decision support under uncertainty, basin modeling, well log modeling, and process stratigraphy. He has also served on and chaired committees designing in-house training in the areas of technical software development and computational and applied mathematics. He currently holds a joint supervisory position in both the Computational Sciences and Stratigraphic and Reservoir Systems Functions. Adam additionally held the title of adjunct assistant professor in the Department of Computational and Applied Mathematics at Rice University from 2007-2012. In both 2006 and 2007, he taught a graduate level course, CAAM 520, on computational science. The course focused on the design and implementation of high performance parallel programs.

[1] The advice, information, and conclusions discussed in this book are those of the author and have not been endorsed by, or reflect the opinions or practices of, ExxonMobil Corporation or its affiliates.

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is in the medical sector, using mobile devices to accelerate information transfer between patients and health care providers.

Preface

Throughout my career, I have mentored both students and fellow employees in programming, and many of them have suggested that I write my thoughts down in book form. However, I have typically responded with the rebuttal that I felt I had nothing novel to present. Being a largely self-taught programmer, I have always been able to rattle off a long list of books from which I have derived most of my knowledge. Therefore, what could I write about that has not already been said?

I came to realize, however, that the majority of books that I encounter tend to focus only on pieces of design or implementation rather than taking a holistic approach. For example, if one wants to learn the C++ language, Stroustrup [24] or Lippman and Lajoie [15] are excellent references. For learning C++ best practices, one need only read the books by Sutter [25, 26, 27], Sutter and Alexandrescu [28], or Meyers [18, 17, 19]. Of course, learning to program extends well beyond C++. For data structures and algorithms, there are always the classics by Knuth [11, 12, 13] or the more accessible and concise book by Cormen *et al* [6]. To learn object-oriented analysis and design, the book by Booch *et al* [4] is an excellent reference. Of course, design patterns can be learned from Gamma *et al* [7], and general programming practices can be learned from many books such as those by McConnell [16], Spinellis [23], or Kernighan and Pike [10].

Certainly, the deeper the specialty one seeks, the more esoteric the book one can find (and should eventually read). This book is not such a book. Rather, I have striven to write a book that operates from the premise that the reader already possesses a working knowledge of the information encased in works such as the aforementioned titles. In this book, I instead attempt to ground the reader's theoretical knowledge of design through practice using a single case study.

Target Audience

As mentioned above, the goal of this book is not to present any specific topic but rather to explore the interrelationship of often compartmentalized subjects. The successful combination of these components to form a cohesive, maintainable, elegant piece of software is, in essence, design. As such, this book is intended to target practicing

professionals, in particular those who have several years of development experience but who do not yet possess sufficient experience to architect independently a large software project.

Because my intent is to emphasize the utilization of one's existing knowledge effectively to design software, I make little effort to explain individual topics in great depth. I believe too many books classified as intermediate to advanced fail because the author devotes too much content describing prerequisites. The result is a massive tome filled with unnecessary detail for the advanced reader, while the beginner is left with a long and complicated exposition that is still inaccessible because the beginner does not possess sufficient knowledge or experience to grasp the subject regardless of the amount of detail devoted to the description. I have, therefore, aimed for conciseness over completeness. Often, I simply refer the reader to relevant material rather than myself describing a background topic in great detail. While this strategy may indeed make this book difficult for beginners, I hope experienced professionals appreciate both the brevity of the book and its tone, which assumes the reader is competent at his or her craft.

Structure of the Book

Learning most tasks in programming requires hands-on experience and repetition; design is no exception. My opinion is that design is best learned through a combination of self-exploration and mentoring from an expert. For this reason, I have chosen to organize this book as a study in design through the detailed working of a case study. Instead of discussing design elements in the abstract, we will instead examine the concrete application of design principles as they relate to the design and construction of a simplistic (but not too simplistic) software project. Importantly, we will not only examine successful results that led to the final implementation, but we will also spend time considering alternatives, some of which are viable and others which are not. Where multiple solutions may suffice, choosing one over the other is often either a matter of context or just taste. Regardless, experience is usually the arbiter of such decisions, and hopefully this book will enable the reader to learn from the author's experiences without having to repeat his mistakes. That is, I hope I have created a book that can serve as a self-contained master class in design.

Language Selection

Design as an abstract concept can be taught in the absence of a particular programming language. However, once committed to a concrete example, a specific language must be chosen for the implementation. I decided to write the case study exclusively using C++. While every line of the program does not appear in the text, all of the source code is available for the reader to examine. Despite this book's primary focus on design, reading the source code is a good way to learn how implementation details in a specific language enable or, at least, facilitate a chosen design. The source code also serves as a high quality (I hope) exemplar of a modern C++ implementation of a complete user application. I highly recommend reading and modifying the source code in conjunction with reading the text.

The decision to use C++ does not imply that C++ is the best choice for all programs, and, in fact, it may not even be the best choice for the program examined in this book. However, to ground the abstraction, a concrete language had to be selected. I chose C++ because it is standardized, widely deployed, and available at zero cost on many platforms. Selfishly, I also chose C++ because it is my most proficient language. While I could, perhaps, have chosen another language meeting the aforementioned objective criteria (e.g., Java), the resulting code would probably have been functional but non-idiomatic due to my relative lack of expertise.

During the writing of this book, C++0x was ratified as C++11 and then updated as C++14. Soon we will have C++17. This progression of standards occurring while I was writing this book should give the reader an idea of how long it takes to write a book in one's "free time" at night! The ratification of these standards and the relative paucity of design literature focused on using these new language and library features give me the unique opportunity to highlight how these new language elements can be used effectively in the design of a large scale program. That said, I have not gone out of my way to incorporate modern C++ features where they are inappropriate just to demonstrate usage. In some cases, new language features provide syntactic convenience (e.g., the `auto` keyword), and, in these instances, I use these features without further mention as they are now established elements of modern C++. In other instances, new language features imply new design semantics afforded by the language (e.g., smart pointers). Where new, modern C++ language or library features enable new design paradigms expressible in the language, I have chosen to highlight these points in a sidebar.

The above said, the reader should not expect this to be a book that simply focuses on various modern C++ features and how to use them effectively. Of particular note will be the relative sparseness of templated classes and the complete lack of some

advanced features such as template metaprogramming. This intentional omission is not meant to be a commentary on the importance of these language features or a critique of their usefulness as design constructs. In fact, they are essential for specific domains, particularly where efficiency and genericity dominate (e.g., the standard library, boost). Rather, the lack of usage of any particular feature reflects my pragmatic approach toward design: the appropriate language subset for a particular project should reflect only those features required to express clearly and efficiently the developer's intent. If your favorite language feature is missing, it may be because I felt that its use in this context was less clear than my chosen alternative. After all, we write programs in higher level languages to be read and understood by other programmers. The task of designing, implementing, maintaining, and extending code in a production environment is quite difficult in its own right. Displays of unnecessary cleverness are generally neither warranted nor appreciated.

GUI Framework and Platforms

While graphical user interfaces (GUI) are not the primary focus of this book, our program will eventually acquire one; hence, selecting a particular GUI framework was necessary. I selected Qt for this purpose and have successfully tested the code with versions 4 and 5. As with C++, Qt is widely deployed, cross platform, and free. As with C++, again, selfishly, I chose Qt because I am experienced using desktop Qt and personally enjoy using it. A big advantage of using Qt is that it makes nearly all of the source code for the project platform-independent. Where platform independence was not achievable, I encapsulated the platform-dependent code behind an abstract interface. We'll explore this design pattern, in depth, in Chapter 7.

In general, I always strive to write standards-conforming, platform-independent code. The source code in this book should compile with any C++14-compliant compiler and on any platform where Qt is supported. That said, every platform and every compiler somehow manage to have their own unique idiosyncrasies, and testing every possible combination of platform and compiler is impossible. I personally have tested the source code for this book on Windows 7, Windows 10, and Linux using gcc (the mingw variant on Windows). I do not own an Apple computer, so I have not tested the source code on Mac OS X. However, the code should work as is or require only very minor changes to get it to compile there if that is your platform of choice. Any known platform-specific code is compartmentalized and encapsulated; extension to other platforms should be straightforward.

The Case Study

To this point, I have not yet mentioned the subject of the case study central to this book. In my opinion, an ideal problem would be one that is not too daunting and yet not so trivial that it needs no design. Additionally, the domain of the problem should be readily understandable to any reader. For this reason, I have chosen to implement a stack-based, Reverse Polish Notation (RPN) calculator that I named pdCalc, short for Practical Design Calculator. Functionally, pdCalc was inspired by the HP48S calculator, an old favorite calculator of mine from high school and college (that still sits on my desk today). I cannot imagine any reader being unfamiliar with the basic operations of a calculator, and making the calculator use RPN adds a little twist that I hope will make the project interesting.

The Source Code

The source code (and supporting unit tests) for pdCalc is available, in its entirety, from Apress's GitHub repository. The easiest way to find the repository is via the **Download Source Code** button located at `www.apress.com/9781484230565`. Appendix A describes, in detail, how to download and build the source code. Although most readers will likely prefer to download the source using a git client, the entire source code is also available from GitHub as a single zip file.

The source code itself is licensed under the GNU Public License (GPL) version 3. I hope you find the source code useful for learning about design and implementation. Beyond its use as an educational aide for this book, you are, of course, free to do anything you like with the source code within the rights granted to you by the GPL.

This book uses two distinct fonts, the standard font you are reading now and a fixed width font used for source code. The fixed width font is demonstrated in the following list of standard template library containers: `vector`, `list`, and `map`. When clear from context, to save space, I often omit namespaces and template arguments. When the discussion requires a block of source code, it will be offset from the rest of the text as follows:

```
class ExampleClass
{
public:
  // your implementation here
};
```

In general, I tried to reproduce the code in the text identically to the code in the repository. However, in some cases, parts of the source were deliberately omitted in the text either for brevity or clarity. I have tried to note instances where the differences are significant. Where the two differ, assume the code in the repository is the more correct and complete version.

Contacting the Author

I suspect that as you read this book and explore the source code, you will invariably have questions. Please feel free to contact me with questions about the content of the book, questions about the source code, errors, or improvements to my design or implementation. I can be contacted at `PracticalDesignBook@gmail.com`. I will make my best effort to reply to all reasonable email related to pdCalc, but without knowing the volume of email I'll receive, I can make no ironclad guarantee that I will be able to respond to every request. If I find that there is a significant interest in a community discussion of pdCalc and its design, I'll investigate establishing a web page to accompany this book. Again, I can offer no promises.

Parting Advice

Finally, in addition to learning something, I hope that you, the reader, have fun with the subject. My personal suggestion is to try to think about the design decisions yourself before reading my solutions. If you are truly industrious, you may even want to implement your own calculator using a completely different, possibly better design. After all, as I said before, design is ultimately about both experience and taste, and your experience and taste may differ significantly from mine.