Preface

This book takes new directions in several situations where the existing software has reached a dead end and, for this reason, it should be of interest to programmers at all levels of experience.

It does not matter in what language you program, but it will help if you have a basic knowledge of the C++ syntax because most examples are written in C++.

The book covers a wide range of subjects in great depth. It is quite concentrated and could scare a beginner if he/she attempted to read it front to cover. Using this guide, you may start with a few selected chapters, and even if you are an expert who has designed his/her own persistent system, you may return to it many times after the first reading.

The book tackles several major issues:

Issue 1: Persistency

We all have been preconditioned to the notion that, when we want to store data, especially a lot of data, it is best to use a database. In many cases this is not true, and using persistent objects greatly simplifies the code and can improve performance by an order of magnitude.

Note that *serialization* (or *archiving*) is only one of several ways to implement persistent objects, and the book will show you that its existing implementations in various languages have flaws and an unnecessarily complicated interface.

If you are a beginner you should be aware of the opportunities persistent objects provide; you will need them sooner than you think. If you are currently using serialization, this book will be an eye-opener for you. And if you have designed your own persistent data, you may find little tricks that will improve their performance.

We believe that an automatic persistence for Objective-C would greatly simplify the design of iPhone applications, and the book provides a special solution for this purpose.

Issue 2: Bi-directional and Intrusive Data Structures

Data structures and their libraries are closely related to persistence because both revolve around references (or pointers). Well-designed libraries simplify implementation and use of persistence, and, in order to make your data persistent, any library you use must be persistent.

Existing "standard" libraries are a major hindrance to progress in software engineering. They cannot store bi-directional associations such as graph or aggregate; yet about half of the relations in real-live applications are bi-directional. Also, these libraries ignore a multitude of pointer-based (intrusive) data structures developed over the past 2 decades.

The big advantage of intrusive data structures is that they can automatically catch many otherwise hard-to-find errors. They also are natural sets, and they can be sorted just as fast as array-based collections. Libraries of intrusive data structures can even store design patterns.

Issue 3: Pointers (or References) as Members of Application Classes

If you think about it, the only purpose of using references as class members[1] is to create data structures, and it is not good practice for your application to create its own data structures. All data structures should come from a well-tested library. Leaving raw references in your classes is an architectural error that increases code complexity, muddles the architecture, and creates the potential for nasty errors.

In the existing practice, raw pointers are mostly used as implicit one-to-one relations or as a complement to a collection when you need a bi-directional one-to-many relation (Aggregate) not available from the standard libraries.

Reference members may create a hard-to-manage network—a situation similar to the reasons for introducing structured programming[2], where networks of *goto* statements were removed from code as a potential hazard. For the same reason, we recommend that explicit references in the application classes are banned. All references should be managed transparently by class libraries, not by the application.

If we accept this design concept, the implementation of persistence also becomes much cleaner and simpler. Pointer management is now removed from the application and is handled by a library.

[1] Variables in Objective-C.

[2] Dijkstra (1976).

Issue 4: Visibility of the Relations

The essential part of the program architecture is the overall data organization or *framework*, which consists of classes and their relations. This is the information provided by the UML class diagram. The problem with the existing programming style and integrated environments is that they focus on classes, while relations are buried and spread through the classes, and are generally hard to find. Understanding a code written by someone else with 20+ classes and 30 relations is a nightmare.

When using our new libraries, you declare all the relations in a short block of statements, one line per relation. The concept is reversed. Instead of classes implementing the relations, relations connect classes, and they have the same visibility as the classes.

This, again, simplifies the implementation and use of persistence. Persistent objects replace a database, and this block of statements that declare relations becomes a database schema. We handle our data structures as if they were a database.

Issue 5: UML Class Diagram Driving the Code

Line by line, the block of statements that declare relations (the database schema) corresponds to the UML class diagram. That could greatly simplify code generators such as Rational Rose which generate a code skeleton implementing the framework. With our new style of libraries, all these generators would have to do would be to generate the appropriate block of statements that declare the relations.

However, we believe[3] that the process should be reversed. UML class diagram is a useful visual aid, but it is more practical to drive the architecture by the block of statements from within the code. This completely removes the problems with the synchronization between the diagram and the code, and a high quality UML class diagram can be automatically produced with each compilation.

Richmond, ON, Canada — Jiri Soukup
Brloh, Czech Republic — Petr Macháček

[3] This is discussed at length in Soukup (2007).

Suggested Reading Paths (Based on Reader's Experience)

Level 1: Beginner Able to Use Basic Collections

Start with Chap. 1, but skip Sect. 1.5. Its purpose is to show how difficult it is to use the built-in serializations, and it would be over your head.

Chapters 2 and 3 are the essential parts of this book, so work through them bit-by-bit. It will be a steep learning curve, but it will be worth it. Conceptually, it may move you ahead of many already "experienced" programmers.

Download the examples from the website and play with them as you read. The critical concepts to understand are the pointer masks, how we collect all active objects, the bitmap for memory blasting, persistent pointer class, the QSP algorithm (Sect. 2.5) which is completely new, and the block of statements that declare the data structures like a database schema.

Browse through Chaps. 4 and 5, not dwelling long on parts you don't understand. You will return to these chapters later. Skip Chap. 6 unless you are programming in Objective-C, and even then this chapter is mostly about implementing persistence for Objective-C, not about using it. For an example of how to use the new Objective-C persistence, experiment with the source of the benchmark (under chap6 or iPhone). By the time this book is published there should be more information and documentation on the website, plus a book blog.

You may find Chap. 7 useful for your decision as to what existing software would be most appropriate for your future projects.

Skip Chap. 8, which is beyond your interest and knowledge.

Level 2: Practitioner Already Using Serialization

Start with Chap. 1. Don't attempt to understand all the details in Sect. 1.5. Just note all the work you have to do when using the built-in serialization, and how convoluted some internal formats are.

Chapters 2 and 3 are the essential parts of this book, and you should read them carefully. Many parts are completely new and have not been published before. Download the examples from the website and play with them as you read. The

critical concepts to understand are the pointer mask, how we collect all active objects, the bitmap for memory blasting, persistent pointer class, the QSP algorithm (Sect. 2.5), and the block of statements that declare the data structures (the "database schema").

Browse through Chaps. 4 and 5; they provide additional considerations you may not be familiar with. Skip Chap. 6 unless you are programming in Objective-C, and even then this chapter is mostly about implementing persistence for Objective-C, not about using it. For an example of how to use the new Objective-C persistence, experiment with the source of the benchmark (under chap6 or iPhone). By the time this book is published there should be more information and documentation on the website, plus a book blog.

You may find Chap. 7 extremely interesting and useful.

Skip Chap. 8 which is beyond your interest and knowledge.

Level 3: Software Architect

This book is mostly about the implementation of persistence and about the new style of implementing (and using!) class libraries. You may not be interested in technical details, but the book describes many new concepts that relates to your work and thinking.

Start with Chap. 1, and without going into the details in Sect. 1.5, note all the work required for the built-in serialization, and how convoluted some internal formats are.

Chapters 2 and 3 are the essential parts of this book, and you may find them interesting because of the algorithms they describe. Many parts are completely new and have not been published before. The critical concepts are the pointer mask, how to collect all active objects, the bitmap for memory blasting, persistent pointer class, the QSP algorithm (Sect. 2.5), and the block of statements that declare the data structures (the *database schema*).

The new handling of data structures (Chap. 3) leads to a major improvement in the cooperation between architect and coders. By controlling the database schema, the architect controls the architecture. The coders cannot deviate from it, but the database schema becomes the key communication tool between the two parties.

Chapters 4 and 5 provide additional details you should consider. Unless you are programming in Objective-C, skip Chap. 6, but even when you use Objective-C, this chapter is more about implementing persistence than about using it. However, Sect. 6.4 is about the importance of NS classes and difficulties with their conversion and it may be of interest to you.

You may find the performance benchmarks of different persistent systems in Chap. 7 most interesting and useful.

Browse quickly through Chap. 8 just to acquaint yourself with this proposal.

Please join the book blog, especially if you have any comments to make about Chap. 9.

Level 4: Expert or Someone Building Class Libraries or Persistent Systems

Read the book from front to back, paying special attention to all *Useful Tricks* and Chaps. 2 and 3. Chapters 4 and 5 provide additional details. Chapter 6 is important if you are using Objective-C, otherwise you can skip it. Chapter 7 has a wealth of information about the performance of different styles of implementing persistence. Chapter 8 is important for designers of class libraries and compilers.

Please join the book blog, especially if you have any comments to make about Chap. 9.

Book Website

The book website is at **http://www.codefarms.com/book**, with a blog and other information. It also allows you to download a zip file with source of all the Listings that are longer than just a few lines—usually expanded so that you can compile and run them. We highly recommend that you play with these programs and modify and evolve them. Besides these programs, which are usually in a single file such as List2_3.cpp, multi-file examples are stored as subdirectories that may even include tt.bat which compiles them, and rr.bat which runs them. All this is organized by chapter, as you would expect.

When you extract this zip file, you get directory **bk** with all the examples organized by chapter. When we refer to directory bk anywhere in the text, for example recommending you to look at bk\chap1\javaSerialization\readme.txt, we refer to this directory.

The website also provides access to Code Farms libraries: DOL, PTL, PPF, InCode, ObjcLib, Layout. They are open source subject to a simple, no-nonsense license which allows commercial use.

Acknowledgements

This book is the result of a truly international cooperation. Authors Jiri Soukup and Petr Macháček are from Canada and the Czech Republic, respectively. The publisher is a global corporation, but the editor Ralf Gerstner works from Germany. James Noble, David Pearce, and Steve Nelson—all from New Zealand—helped us with J-Aspects, while Olaf Spinczyk from Norway wrote the C++ Aspect example. The help with Objective-C came from Sean Yixiang Lu[4] in Singapore, and Raj Lokanath from California became a co-author of two chapters. Cay Horstmann from California helped us with advanced features of both Java and C++, including the format of Java serialization. We owe a lot to our pre-release expert reviewers Mark Bales (California) and Naresh Bhatia (Massachusetts), and to our contact in ObjectStore (Don White, Massachusetts). The author of the BOOST persistence, Richard Ramey from California, helped us with the BOOST implementation of the benchmark.

Most of all, we want to thank our wives Hana and Vera, both engineers and programmers like us, for their support and patience. This project took much more time than we anticipated.

[4] Who in the meantime relocated to Seattle.

http://www.springer.com/978-3-642-39322-8

Serialization and Persistent Objects
Turning Data Structures into Efficient Databases
Soukup, J.; Macháček, P.
2014, XVIII, 263 p. 133 illus., Hardcover
ISBN: 978-3-642-39322-8