Extracted from:

Resourceful Code Reuse

Write → Compile → Link → Run

This PDF file contains pages extracted from *Resourceful Code Reuse*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2021 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

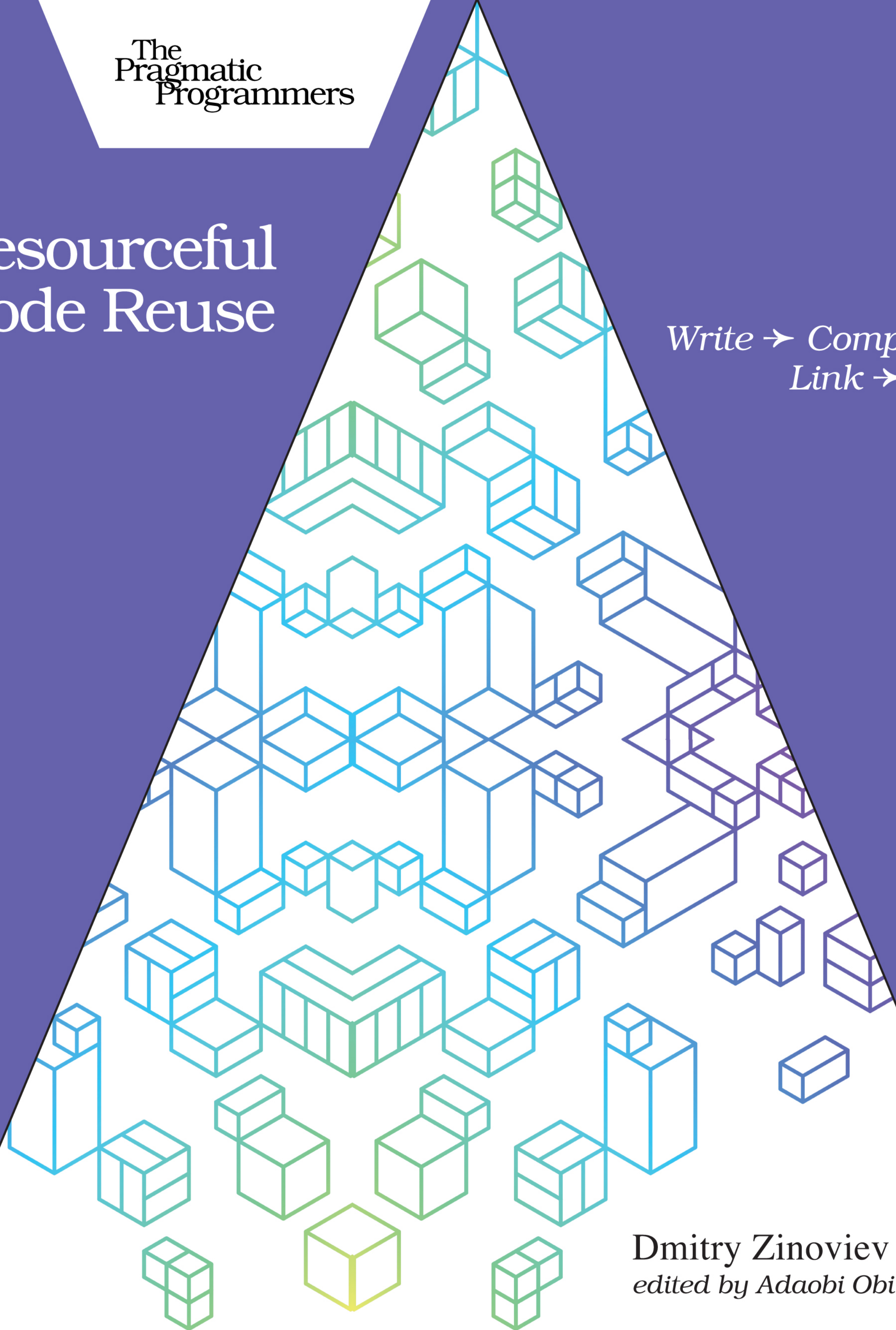
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Resourceful Code Reuse

Write → Compile → Link → Run



Dmitry Zinoviev

edited by Adaobi Obi Tulton

Resourceful Code Reuse

Write → Compile → Link → Run

Dmitry Zinoviev

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Corina Lebegioara
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2021 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-820-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2021

Introduction: Why Reuse Code?

Despite its name, modern computer science isn't just a science—it's also a 200-year-old arcane art, if we accept Charles Babbage as the first theoretical computer scientist. It comes loaded with legends, anecdotes, apocrypha, unwritten rules, and other words of wisdom. One example is the *proudly found elsewhere (PFE)* or *invented elsewhere (IE)* predisposition: if the wheel has been already invented, you shouldn't design it again without a strong reason but reuse the existing design. PFE/IE is a special case of the more general philosophical Occam's razor principle: entities shouldn't be multiplied without necessity.

As far as this book is concerned, both Occam's razor and PFE/IE apply to a simple concept: you should write a large code fragment only if you or someone else hasn't written that code fragment before. If the fragment has been written, use it again—*reuse* it.

Several reasons exist for code reuse:

1. Code reuse puts into practice the advice of none other than William of Occam (or Ockham, depending on whom you ask). If you and I were philosophers, that alone would suffice.

2. Code reuse improves your productivity. Code fragments that you develop to deal with frequently occurring tasks (for example, reading from and writing to JSON files or supporting heterogeneous arrays and maps) become parts of your programming portfolio. You can readily incorporate them into your new projects, most likely bypassing the unit testing stage. Third-party libraries (for example, `libjson`) are usually even more powerful and more reliable. Ideally, you'd like to build your new program out of existing pieces with as little new code of your own as possible.

3. Code reuse improves software quality. A reuseable code unit that has been a part of another project must have been thoroughly debugged and tested in the past. It never hurts to test it again, but at least you can expect to have a good baseline.

• Click HERE to purchase this book now. discuss

4. Code reuse improves software configurability. Runtime code reuse techniques from Chapter 3, Reuse Code at Runtime (C and Python), on page ?, allow your program to postpone deciding what code to use until the program runs. You can develop a skeletal program (a *framework*) with slots for future code fragments to be filled as needed. In a sense, you get a program that may configure itself or be configured by the user on the fly.

You can reuse code in many ways and at many stages. This book will explain how to reuse your *source code* (the program's text written in a human-readable programming language, such as C or Python) in Chapter 1, Reuse Code at Compile Time (C and Python), on page ?. You'll learn how to speed up program build time and avoid disclosing the source code (if desirable) by creating object files and libraries in Chapter 2, Reuse Code at Link Time (C Only), on page ?. Finally, in Chapter 3, Reuse Code at Runtime (C and Python), on page ?, you'll learn how to physically separate the proper program code and the reusable fragments and either bind them when the program runs (Harnessing Dynamic Loading, on page ?) or not bind them at all and instead use a network communication protocol to request services and obtain results (Getting a Taste of Remote Procedure Calls, on page ?). Incidentally, the latter form of reuse makes your code available to both your own programs and any other programs you authorize.

C vs. Python

You'll see that C and Python software each require different code reuse techniques.

C programs are usually compiled—they're converted into files that contain native CPU instructions with little or no "memory" of the original development language. These converted files are relatively language-agnostic and can be combined with other files developed in other compiled programming languages (such as C++ or Rust). It makes sense then to organize them into libraries for further reuse.

Python programs are usually interpreted by an interpreter or byte-compiled —they're converted into files that contain instructions for a virtual machine. These converted files have more limited reusability, get along only with other Python files, and often have to be byte-compiled again before reuse. This limitation makes building complex shared Python libraries impractical.

• Click HERE to purchase this book now. discuss

Running Example

Throughout the book, I'll show you how to reuse code written mostly in the C language and occasionally in Python. For that, we're going to need a running example. We'll start with a program in C that isn't necessarily unstructured, cluttered, or hard to maintain, but it's also not designed to be reused in the future. We'll slice, dice, and even tear this program apart and put it on different computers to make the code more reusable.

Interestingly, you don't even need to know anything about the problem that the program solves to understand the reuse mechanisms. A differential equation solver is as good as a web browser, which, in turn, is as good as a program that controls a unicorn feeder. So, let's pick a common problem that has been automated time and again: a program that reads the content of a JSON file into a C or Python data structure, modifies it somehow, and writes the modified data structure to another JSON file.

A Note on JSON

The name "JavaScript Object Notation" (JSON) is misleading in almost every aspect. For starters, JavaScript itself may have more in common with Java coffee than Java language, but that's not JSON's fault. Second, JSON was designed to work with JavaScript, but now it's used as a programming language–agnostic *data exchange language* and isn't attached to JavaScript anymore. Third, JSON doesn't describe objects as we know them in OOP. JSON supports the following data types (with the C and Python equivalents shown in parentheses):

- `null` (`NULL`, `None`)
- Booleans: `true` (`1`, `True`) and `false` (`0`, `False`)
- Numbers (same as in C and Python)
- Strings (`char*`, `str`). JSON strings must use single quotation marks. Example: `'JSON or Jason?'`
- Arrays (somewhat similar to arrays in C; lists in Python). JSON arrays are heterogeneous. They aren't the same as C arrays. Example: `[1, -3.14, "hello", null]`
- Objects (don't exist in C; dictionaries in Python). Example: `{"name": "Dmitry", "smart": true, "children": ["Roman", "Eugenia"]}`

You'd have to implement heterogeneous linear arrays and key-value associative arrays to support JSON arrays and objects in C. Not a big deal. let's assume that all the necessary arrays and their access functions have been already implemented elsewhere, and we can readily *reuse* them—which makes this book unexpectedly recursive.

• Click HERE to purchase this book now. discuss

The program will rely on two functions: read_json(char *fname) and write_json(char *fname, JSON_Object *object). Of course, it also has the main() function—but, as a rule, the main() function is so specific to a program that you cannot reuse it.

The read_json(char *fname) function attempts to open the file named fname, read valid JSON from the file, and construct and return a JSON object. Converting a serial, often human-readable representation of an object into an actual binary object is called *deserialization*. The write_json(char *fname, JSON_Object *object) function attempts to create a new file named fname, convert a JSON object into a string, and write the string into the file. Converting a complex binary object into a serial, often human-readable representation is called *serialization*.

The internal organization of a JSON object isn't our concern, and neither is the internal organization of the functions. We only need to know that the object definition and the functions exist and we plan to make them reusable. Here is how the object and the functions could be declared in C:

jsontool.c

```
typedef struct JSON_Object {
  // Lots of lines of code here
  // ...
} JSON_Object;

// Lots of helper functions, data types, and global variables
int json_errno = 0;
bool read_boolean(FILE* infile) { /* ... */; return false; }
char *read_string(FILE* infile) { /* ... */; return NULL; }
void *obscure_helper() { /* ... */ ; return NULL; }
// ...

// JSON parser and writer
JSON_Object *read_json(char *fname) {
  JSON_Object *object = NULL;
  // Lots of lines of code here
  // ...
  return object;
}

int write_json(char *fname, JSON_Object *object) {
  int status = 0;
  // Lots of lines of code here
  // ...
  return status;
}
```

Here is how to declare the same object and functions in Python:

jsontool.py

```
def read_json(fname):
    object = None
```

```
    # Lots of lines of code here - but fewer than in C!
    # ...
    return object

def write_json(fname, object):
    status = 0
    # Fewer lines of code here - we love Python for its brevity!
    # ...
    return status
```

Note that these functions aren't members of any Python class. Unlike C++ classes and C programs, Python and Java classes are monolithic. They cannot be split into separate compilation units. Designing for code reuse in an object-oriented language is a topic for another book.

There may be various helper functions and global variables in the example program, such as `read_string()` for reading quoted strings, `read_boolean()` for reading boolean values, `read_array()` for reading an array in square brackets, and even the fictitious and suspiciously named `obscure_helper()` whose job is to optimize the inner machinery of `read_json()`. The variable `json_errno` holds the code of the most recent error in any JSON-related function (similar to the variable `errno` from the standard C library file `errno.h`).

Finally, bear in mind that this book doesn't explain how to parse, process, or generate JSON. The book treats the functions previously mentioned as "black boxes" that you could and would develop independently. This approach allows you to concentrate on the reuse practices rather than on the domain-specific details.

The stage is set. We're ready to look at the first round of code reuse techniques, and we'll start with compile-time reuse.

• Click HERE to purchase this book now. discuss