Extracted from:

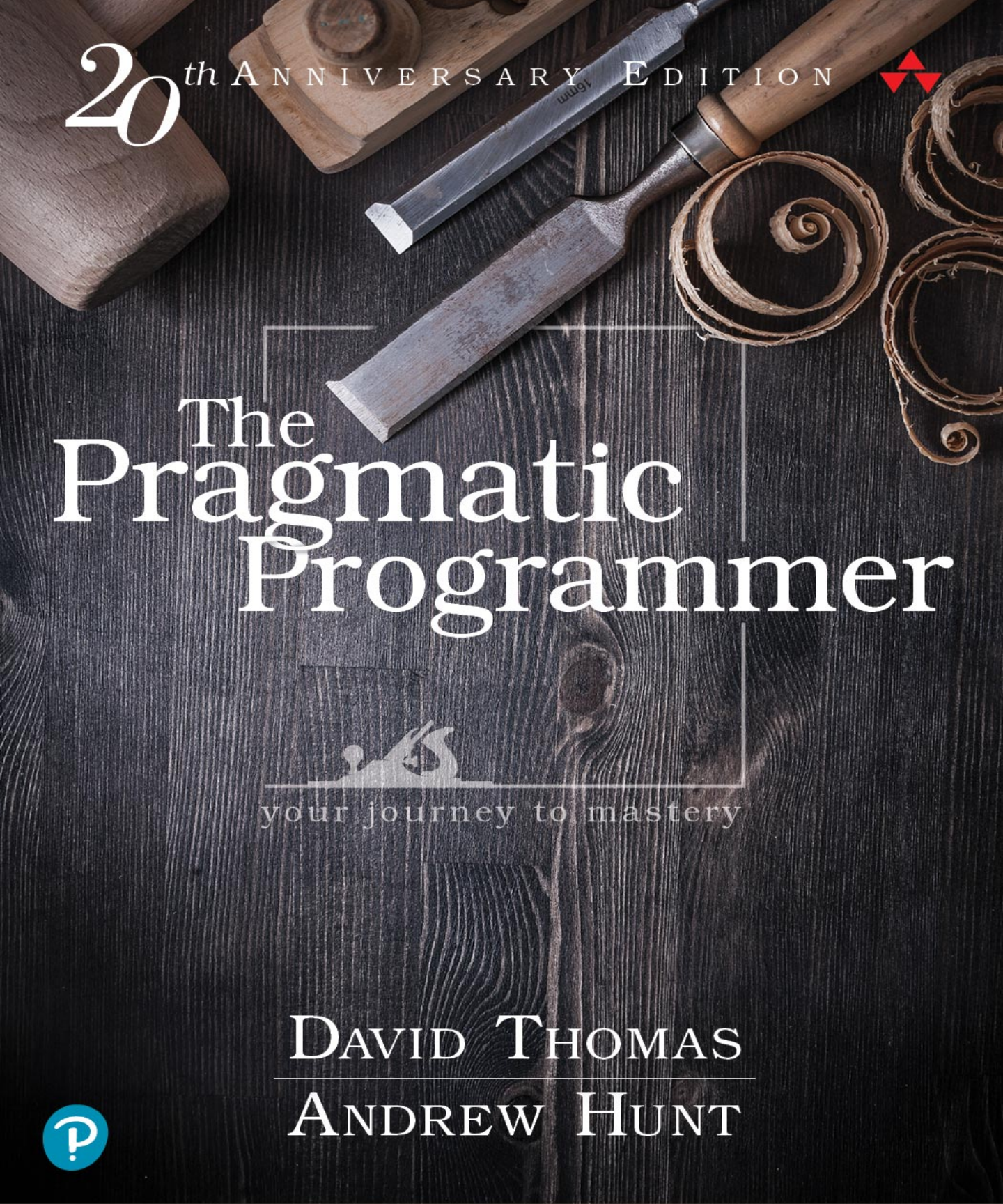
The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo


20th Anniversary Edition
The Pragmatic Programmer
your journey to mastery
David Thomas
Andrew Hunt

The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Dave Thomas
Andy Hunt

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: [to come from ITP]

ISBN-13: 978-0-13-595705-9
ISBN-10: 0-13-595705-2

1 19

The Evils of Duplication

Giving a computer two contradictory pieces of knowledge was Captain James T. Kirk's preferred way of disabling a marauding artificial intelligence. Unfortunately, the same principle can be effective in bringing down *your* code.

As programmers, we collect, organize, maintain, and harness knowledge. We document knowledge in specifications, we make it come alive in running code, and we use it to provide the checks needed during testing.

Unfortunately, knowledge isn't stable. It changes—often rapidly. Your understanding of a requirement may change following a meeting with the client. The government changes a regulation and some business logic gets outdated. Tests may show that the chosen algorithm won't work. All this instability means that we spend a large part of our time in maintenance mode, reorganizing and reexpressing the knowledge in our systems.

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong. Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive and existing requirements evolve as we're heads-down on the project. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity, but a routine part of the entire development process.

When we perform maintenance, we have to find and change the representations of things—those capsules of knowledge embedded in the application. The problem is that it's easy to duplicate knowledge in the specifications, processes, and programs that we develop, and when we do so, we invite a maintenance nightmare—one that starts well before the application ships.

We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the DRY principle:

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Why do we call it DRY?

Tip 15 DRY—Don't Repeat Yourself

The alternative is to have the same thing expressed in two or more places. If you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction. It isn't a question of whether you'll remember: it's a question of when you'll forget.

You'll find the DRY principle popping up time and time again throughout this book, often in contexts that have nothing to do with coding. We feel that it is one of the most important tools in the Pragmatic Programmer's tool box.

In this section we'll outline the problems of duplication and suggest general strategies for dealing with it.

DRY is More Than Code

Let's get something out of the way up-front. In the first edition of this book we did a poor job of explaining just what we meant by *Don't Repeat Yourself*. Many people took it to refer to code only: they thought that DRY means "don't copy-and-paste lines of source."

That *is* part of DRY, but it's a tiny and fairly trivial part.

DRY is about the duplication of *knowledge*, of *intent*. It's about expressing the same thing in two different places, possibly in two totally different ways.

Here's the acid test: when some single facet of the code has to change, do you find yourself making that change in multiple places, and in multiple different formats? Do you have to change code and documentation, or a database schema and a structure that holds it, or…? If so, your code isn't DRY.

So let's look at some typical examples of duplication.

Duplication in Code

It may be trivial, but code duplication is oh, so common. Here's an example:

```
def print_balance(account)
  printf "Debits:  %10.2f\n", account.debits
  printf "Credits: %10.2f\n", account.credits
  if account.fees < 0
    printf "Fees:    %10.2f-\n", -account.fees
  else
    printf "Fees:    %10.2f\n", account.credits
  end
  printf "         ----------\n"
  if account.balance < 0
    printf "Balance: %10.2f-\n", -account.balance
```

```
  else
    printf "Balance: %10.2f\n", account.balance
  end
end
```

For now ignore the implication that we're committing the newbie mistake of storing currencies in floats. Instead see if you can spot duplications in this code. (We can see at least three things, but you might see more).

What did you find? Here's our list.

First, there's clearly a copy-and-paste duplication of handling the negative numbers. We can fix that by adding another function:

```
def format_amount(value)
  result = sprintf("%10.2f", value.abs)
  if value < 0
    result + "-"
  else
    result + " "
  end
end

def print_balance(account)
  printf "Debits:  %10.2f\n", account.debits
  printf "Credits: %10.2f\n", account.credits
  printf "Fees:    %s\n",     format_amount(account.fees)
  printf "         ———-\n"
  printf "Balance: %s\n",     format_amount(account.balance)
end
```

Another duplication is the repetition of the field width in all the printf calls. We *could* fix this by introducing a constant and passing it to each call, but why not just use the existing function?

```
def format_amount(value)
  result = sprintf("%10.2f", value.abs)
  if value < 0
    result + "-"
  else
    result + " "
  end
end

def print_balance(account)
  printf "Debits:  %s\n", format_amount(account.debits)
  printf "Credits: %s\n", format_amount(account.credits)
  printf "Fees:    %s\n", format_amount(account.fees)
  printf "         ———-\n"
  printf "Balance: %s\n", format_amount(account.balance)
```

```
end
```

Anything more? Well, what if the client asks for an extra space between the labels and the numbers? We'd have to change 5 lines. Let's remove that duplication.

```
def format_amount(value)
  result = sprintf("%10.2f", value.abs)
  if value < 0
    result + "-"
  else
    result + " "
  end
end

def print_line(label, value)
  printf "%-9s%s\n", label, value
end

def report_line(label, amount)
  print_line(label + ":", format_amount(amount))
end

def print_balance(account)
  report_line("Debits",  account.debits)
  report_line("Credits", account.credits)
  report_line("Fees",    account.fees)
  print_line("",         "————-")
  report_line("Balance", account.balance)
end
```

If we have to change the formatting of amounts, we change format_amount. If we want to change the label format, we change report_line.

There's still an implicit DRY violation: the number of hyphens in the separator line is related to the width of the amount field. But it isn't an exact match: it's currently one character shorter, so any trailing minus signs extend beyond the column. This is the customer's intent, and it's a different intent to the actual formatting of amounts.

Not All Code Duplication is Knowledge Duplication

As part of your online wine ordering application you're capturing and validating your user's age, along with the quantity they're ordering. According to the site owner, they should both be numbers, and both greater than zero. So you code up the validations:

```
def validate_age(value):
    validate_type(value, :integer)
    validate_min_integer(value, 0)
```

```
def validate_quantity(value):
    validate_type(value, :integer)
    validate_min_integer(value, 0)
```

During code review, the resident know-all bounces this code, claiming it's a DRY violation: both function bodies are the same.

They are wrong. The code is the same, but the knowledge they represent is different. The two functions validate two separate things that just happen to have the same rules. That's a coincidence, not a duplication.

Duplication in Documentation

Eat your vegetables. Get 8 hours sleep. Comment your functions.

And so we often see something like this:

```
# Calculate the fees for this account.
#
# * Each returned check costs $20
# * If the account is in overdraft for more than 3 days,
#   charge $10 for each day
# * If the average account balance is greater that $2,000
#   reduce the fees by 50%

def fees(a)
  f = 0
  if a.returned_check_count > 0
    f += 20 * a.returned_check_count
  end
  if a.overdraft_days > 3
    f += 10*a.overdraft_days
  end
  if a.average_balance > 2_000
    f /= 2
  end
  f
end
```

The intent of this function is given twice: once in the comment and again in the code. The customer changes a fee, and we have to update both. Given time, we can pretty much guarantee the comment and the code will get out of step.

Ask yourself when the comment adds to the code. From our point of view, it simply compensates for some bad naming and layout. How about just this:

```
def calculate_account_fees(account)
  fees  = 20 * account.returned_check_count
```

```
  fees += 10 * account.overdraft_days  if account.overdraft_days > 3
  fees /= 2                            if account.average_balance > 2_000
  fees
end
```

The name says what it does, and if someone needs details, they're laid out in the source. That's DRY!

DRY Violations in Data

Our data structures represent knowledge, and they can fall afoul of the DRY principle. Let's look at a class representing a line:

```
class Line {
  Point  start;
  Point  end;
  double length;
};
```

At first sight, this class might appear reasonable. A line clearly has a start and end, and will always have a length (even if it's zero). But we have duplication. The length is defined by the start and end points: change one of the points and the length changes. It's better to make the length a calculated field:

```
class Line {
  Point  start;
  Point  end;
  double length() { return start.distanceTo(end); }
};
```

Later on in the development process, you may choose to violate the DRY principle for performance reasons. Frequently this occurs when you need to cache data to avoid repeating expensive operations. The trick is to localize the impact. The violation is not exposed to the outside world: only the methods within the class have to worry about keeping things straight.

class Line { private double length; private Point start; private Point end;

public Line(Point start, Point end) { this.start = start; this.end = end; calculateLength() }

// public void setStart(Point p) { this.start = p; calculate_length() } void setEnd(Point p) { this.end = p; calculate_length() }

Point getStart(void) { return start; } Point getEnd(void) { return end; }

double getLength() { return length; }

private void calculate_length() { this.length = start.distanceTo(end); } }; ~~~

This example also illustrates an important issue: whenever a module exposes a data structure, you're coupling all the code that uses that structure to the implementation of that module. Where possible, always use accessor functions to read and write the attributes of objects. It will make it easier to add functionality in the future.

This use of accessor functions ties in with Meyer's *Uniform Access principle*, described in *Object-Oriented Software Construction [Mey97]*, which states that

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation."

Representational Duplication

Your code interfaces to the outside world: other libraries via APIs, other services via remote calls, data in external sources, and so on. And pretty much each time you do, you introduce some kind of DRY violation: your code has to have knowledge that is also present in the external *thing*. It needs to know the API, or the schema, or the meaning of error codes, or whatever. The duplication here is that two things (your code and the external entity) have to have knowledge of the representation of their interface. Change it at one end, and the other end breaks.

This kind of duplication is inevitable, but can be mitigated. Here are some strategies.

Duplication Across Internal APIs

For internal APIs, look for tools that let you specify the API in some kind of neutral format. These tools will typically generate documentation, mock APIs, functional tests, and API clients, the latter in a number of different languages. Ideally the tool will store all your APIs in a central repository, allowing them to be shared across teams.

Duplication Across External APIs

Increasingly, you'll find that public APIs are documented formally using something like OpenAPI[2]. This allows you to import the API spec into your local API tools and integrate more reliably with the service.

2. https://github.com/OAI/OpenAPI-Specification

If you can't find such a specification, consider creating one and publishing it. Not only will others find it useful; you may even get help maintaining it.

Duplication With Data Sources

Many data sources allow you to introspect on their data schema. This can be used to remove much of the duplication between them and your code. Rather than manually creating the code to contain this stored data, you can generate the containers directly from the schema. Many persistence frameworks will do this heavy lifting for you.

There's another option, and one we often prefer. Rather than writing code that represents external data in a fixed structure (an instance of a struct or class, for example), just stick it into a key/value data structure (your language might call it a map, hash, dictionary, or even object).

On its own this is a risky thing to do: you lose a lot of the security of knowing just what data you're working with. So we recommend adding a second layer to this solution: a simply table driven validation suite that verifies that the map you've created contains at least the data you need, in the format you need it. Again, you might be able to use your API generation tool to perform this validation.

Interdeveloper Duplication

Perhaps the hardest type of duplication to detect and handle occurs between different developers on a project. Entire sets of functionality may be inadvertently duplicated, and that duplication could go undetected for years, leading to maintenance problems. We heard firsthand of a U.S. state whose governmental computer systems were surveyed for Y2K compliance. The audit turned up more than 10,000 programs that each contained a different version of Social Security number validation code.

At a high level, deal with the problem by building a strong, tight-knit team with good communications.

However, at the module level, the problem is more insidious. Commonly needed functionality or data that doesn't fall into an obvious area of responsibility can get implemented many times over.

We feel that the best way to deal with this is to encourage active and frequent communication between developers.

Maybe run a daily scrum standup meeting. Set up forums (such as Slack channels) to discuss common problems. This provides a nonintrusive way of

communicating—even across multiple sites—while retaining a permanent history of everything said.

Appoint a team member as the project librarian, whose job is to facilitate the exchange of knowledge. Have a central place in the source tree where utility routines and scripts can be deposited. And make a point of reading other people's source code and documentation, either informally or during code reviews. You're not snooping—you're learning from them. And remember, the access is reciprocal—don't get twisted about other people poring (pawing?) through *your* code, either.

Tip 16 Make It Easy to Reuse

What you're trying to do is foster an environment where it's easier to find and reuse existing stuff than to write it yourself. *If it isn't easy, people won't do it.* And if you fail to reuse, you risk duplicating knowledge.

Related Sections Include

- Topic 38, *Programming by Coincidence*, on page ?
- Topic 32, *Configuration*, on page ?
- Topic 28, *Decoupling*, on page ?
- Topic 8, *The Essence of Good Design*, on page ?
- Topic 40, *Refactoring*, on page ?