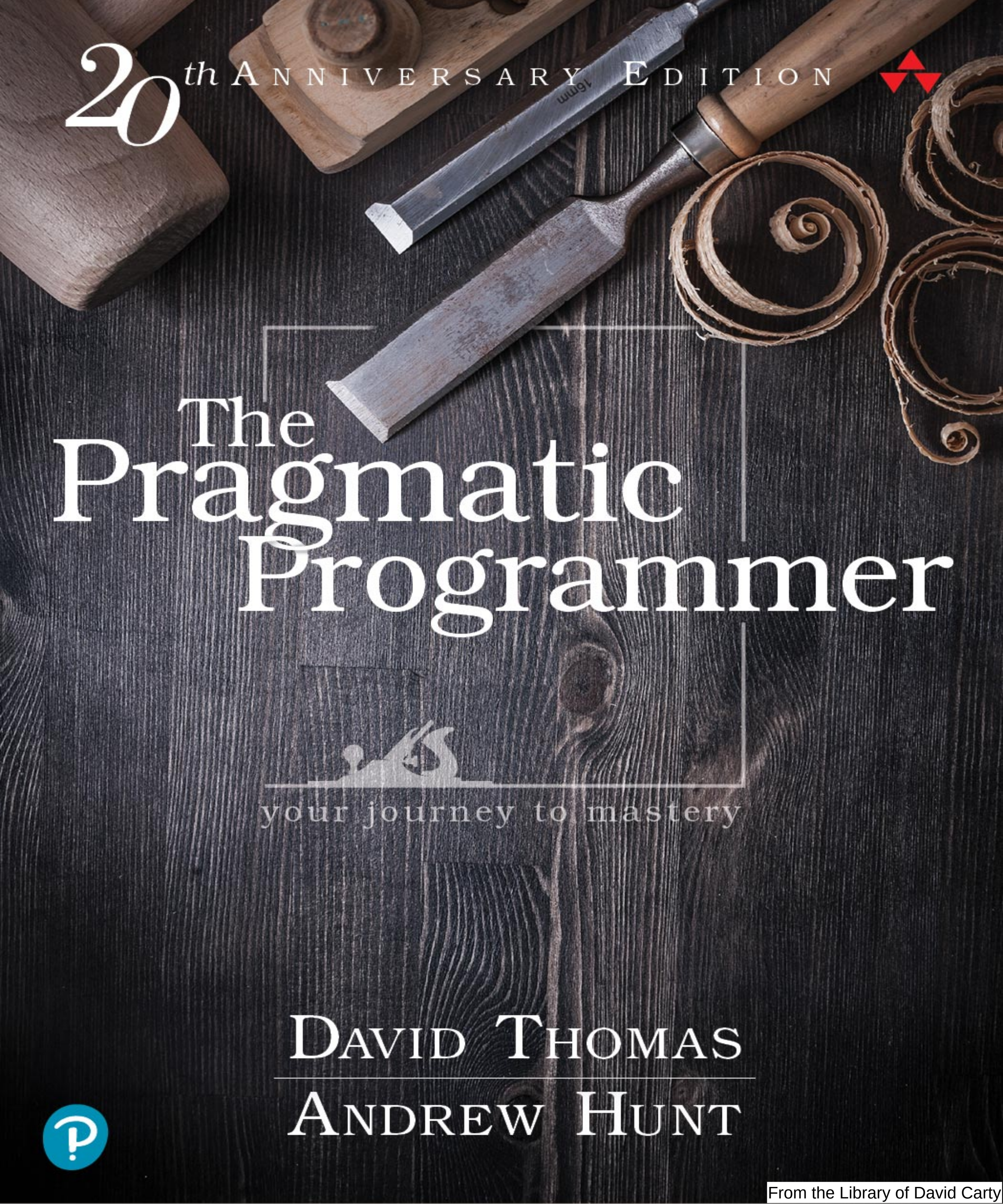

20th Anniversary Edition

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: 2019944178

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

Version: P2.0, October 28, 2019

Contents

Foreword

I remember when Dave and Andy first tweeted about the new edition of this book. It was big news. I watched as the coding community responded with excitement. My feed buzzed with anticipation. After twenty years, *The Pragmatic Programmer* is just as relevant today as it was back then.

It says a lot that a book with such history had such a reaction. I had the privilege of reading an unreleased copy to write this foreword, and I understood why it created such a stir. While it's a technical book, calling it that does it a disservice. Technical books often intimidate. They're stuffed with big words, obscure terms, convoluted examples that, unintentionally, make you feel stupid. The more experienced the author, the easier it is to forget what it's like to learn new concepts, to be a beginner.

Despite their decades of programming experience, Dave and Andy have conquered the difficult challenge of writing with the same excitement of people who've just learned these lessons. They don't talk down to you. They don't assume you are an expert. They don't even assume you've read the first edition. They take you as you are—programmers who just want to be better. They spend the pages of this book helping you get there, one actionable step at a time.

To be fair, they'd already done this before. The original release was full of tangible examples, new ideas, and practical tips to build your coding muscles and develop your coding brain that still apply today. But this updated edition makes two improvements on the book.

The first is the obvious one: it removes some of the older references, the out-of-date examples, and replaces them with fresh, modern content. You won't find examples of loop invariants or build machines. Dave and Andy have taken their powerful content and made sure the lessons still come through, free of the distractions of old examples. It dusts off old ideas like DRY (don't repeat yourself) and gives them a fresh coat of paint, really making them shine.

But the second is what makes this release truly exciting. After writing the first edition, they had the chance to reflect on what they were trying to say, what they wanted their readers to take away, and how it was being received. They got feedback on those lessons. They saw what stuck, what needed refining, what was misunderstood. In the twenty years that this book has made its way through the hands and hearts of programmers all over the world, Dave and Andy have studied this response and formulated new ideas, new concepts.

They've learned the importance of agency and recognized that developers have arguably more agency than most other professionals. They start this book with the simple but profound message: "it's your life." It reminds us of our own power in our code base, in our jobs, in our careers. It sets the tone for everything else in the book—that it's more than just another technical book filled with code examples.

What makes it truly stand out among the shelves of technical books is that it understands what it means to be a programmer. Programming is about trying to make the future less painful. It's about making things easier for our teammates. It's about getting things wrong and being able to bounce back. It's about forming good habits. It's about understanding your toolset. Coding is just part of the world of being a programmer, and this book explores that world.

I spend a lot of time thinking about the coding journey. I didn't grow up coding; I didn't study it in college. I didn't spend my teenage years tinkering with tech. I entered the coding world in my mid-twenties and had to learn what it meant to be a programmer. This community is very different from others I'd been a part of. There is a unique dedication to learning and practicality that is both refreshing and intimidating.

For me, it really does feel like entering a new world. A new town, at least. I had to get to know the neighbors, pick my grocery store, find the best coffee shops. It took a while to get the lay of the land, to find the most efficient routes, to avoid the streets with the heaviest traffic, to know when traffic was likely to hit. The weather is different, I needed a new wardrobe.

The first few weeks, even months, in a new town can be scary. Wouldn't it be wonderful to have a friendly, knowledgeable neighbor who'd been living there a while? Who can give you a tour, show you those coffee shops? Someone who'd been there long enough to know the culture, understand the pulse of the town, so you not only feel at home, but become a contributing member as well? Dave and Andy are those neighbors.

As a relative newcomer, it's easy to be overwhelmed not by the act of programming but the process of becoming a programmer. There is an entire mindset shift that needs to happen—a change in habits, behaviors, and expectations. The process of becoming a better programmer doesn't just happen because you know how to code; it must be met with intention and deliberate practice. This book is a guide to becoming a better programmer efficiently.

But make no mistake—it doesn't tell you how programming should be. It's not philosophical or judgmental in that way. It tells you, plain and simple, what a Pragmatic Programmer is—how they operate, and how they approach code. They leave it up to you to decide if you want to be one. If you feel it's not for you, they won't hold it against you. But if you decide it is, they're your friendly neighbors, there to show you the way.

▶ *Saron Yitbarek*
Founder & CEO of CodeNewbie
Host of Command Line Heroes

CHAPTER 4

Pragmatic Paranoia

Tip 36 You Can't Write Perfect Software

Did that hurt? It shouldn't. Accept it as an axiom of life. Embrace it. Celebrate it. Because perfect software doesn't exist. No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first. And unless you accept this as a fact, you'll end up wasting time and energy chasing an impossible dream.

So, given this depressing reality, how does a Pragmatic Programmer turn it into an advantage? That's the topic of this chapter.

Everyone knows that they personally are the only good driver on Earth. The rest of the world is out there to get them, blowing through stop signs, weaving between lanes, not indicating turns, texting on the phone, and just generally not living up to our standards. So we drive defensively. We look out for trouble before it happens, anticipate the unexpected, and never put ourselves into a position from which we can't extricate ourselves.

The analogy with coding is pretty obvious. We are constantly interfacing with other people's code—code that might not live up to our high standards—and dealing with inputs that may or may not be valid. So we are taught to code defensively. If there's any doubt, we validate all information we're given. We use assertions to detect bad data, and distrust data from potential attackers or trolls. We check for consistency, put constraints on database columns, and generally feel pretty good about ourselves.

But Pragmatic Programmers take this a step further. *They don't trust themselves, either.* Knowing that no one writes perfect code, including themselves, Pragmatic Programmers build in defenses against their own mistakes. We describe the first defensive measure in *Design by Contract*: clients and suppliers must agree on rights and responsibilities.

In *Dead Programs Tell No Lies*, we want to ensure that we do no damage while we're working the bugs out. So we try to check things often and terminate the program if things go awry.

Assertive Programming describes an easy method of checking along the way—write code that actively verifies your assumptions.

As your programs get more dynamic, you'll find yourself juggling system resources—memory, files, devices, and the like. In *How to Balance Resources*, we'll suggest ways of ensuring that you don't drop any of the balls.

And most importantly, we stick to small steps always, as described in *Don't Outrun Your Headlights*, so we don't fall off the edge of the cliff.

In a world of imperfect systems, ridiculous time scales, laughable tools, and impossible requirements, let's play it safe. As Woody Allen said, "When everybody actually *is* out to get you, paranoia is just good thinking."

23 Design by Contract

Nothing astonishes men so much as common sense and plain dealing.

➤ *Ralph Waldo Emerson, Essays*

Dealing with computer systems is hard. Dealing with people is even harder. But as a species, we've had longer to figure out issues of human interactions. Some of the solutions we've come up with during the last few millennia can be applied to writing software as well. One of the best solutions for ensuring plain dealing is the *contract*.

A contract defines your rights and responsibilities, as well as those of the other party. In addition, there is an agreement concerning repercussions if either party fails to abide by the contract.

Maybe you have an employment contract that specifies the hours you'll work and the rules of conduct you must follow. In return, the company pays you a salary and other perks. Each party meets its obligations and everyone benefits.

It's an idea used the world over—both formally and informally—to help humans interact. Can we use the same concept to help software modules interact? The answer is "yes."

DBC

Bertrand Meyer (*Object-Oriented Software Construction [Mey97]*) developed the concept of *Design by Contract* for the language Eiffel.[1] It is a simple yet powerful technique that focuses on documenting (and agreeing to) the rights and responsibilities of software modules to ensure program correctness. What is a correct program? One that does no more and no less than it claims to do. Documenting and verifying that claim is the heart of *Design by Contract* (DBC, for short).

Every function and method in a software system *does something*. Before it starts that *something*, the function may have some expectation of the state of the world, and it may be able to make a statement about the state of the world when it concludes. Meyer describes these expectations and claims as follows:

Preconditions
: What must be true in order for the routine to be called; the routine's requirements. A routine should never get called when its preconditions would be violated. It is the caller's responsibility to pass good data (see the box on page 110).

Postconditions
: What the routine is guaranteed to do; the state of the world when the routine is done. The fact that the routine has a postcondition implies that it *will* conclude: infinite loops aren't allowed.

Class invariants
: A class ensures that this condition is always true from the perspective of a caller. During internal processing of a routine, the invariant may not hold, but by the time the routine exits and control returns to the caller, the invariant must be true. (Note that a class cannot give unrestricted write-access to any data member that participates in the invariant.)

The contract between a routine and any potential caller can thus be read as

> If all the routine's preconditions are met by the caller, the routine shall guarantee that all postconditions and invariants will be true when it completes.

If either party fails to live up to the terms of the contract, then a remedy (which was previously agreed to) is invoked—maybe an exception is raised, or the program terminates. Whatever happens, make no mistake that failure to live up to the contract is a bug. It is not something that should ever happen,

1. Based in part on earlier work by Dijkstra, Floyd, Hoare, Wirth, and others.

which is why preconditions should not be used to perform things such as user-input validation.

Some languages have better support for these concepts than others. Clojure, for example, supports pre- and post-conditions as well as the more comprehensive instrumentation provided by *specs*. Here's an example of a banking function to make a deposit using simple pre- and post-conditions:

```
(defn accept-deposit [account-id amount]
   { :pre [  (> amount 0.00)
             (account-open? account-id) ]
     :post [ (contains? (account-transactions account-id) %) ] }
   "Accept a deposit and return the new transaction id"
   ;; Some other processing goes here...
   ;; Return the newly created transaction:
   (create-transaction account-id :deposit amount))
```

There are two preconditions for the accept-deposit function. The first is that the amount is greater than zero, and the second is that the account is open and valid, as determined by some function named account-open?. There is also a postcondition: the function guarantees that the new transaction (the return value of this function, represented here by '%') can be found among the transactions for this account.

If you call accept-deposit with a positive amount for the deposit and a valid account, it will proceed to create a transaction of the appropriate type and do whatever other processing it does. However, if there's a bug in the program and you somehow passed in a negative amount for the deposit, you'll get a runtime exception:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError: Assert failed: (> amount 0.0)
```

Similarly, this function requires that the specified account is open and valid. If it's not, you'll see that exception instead:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError: Assert failed: (account-open? account-id)
```

Other languages have features that, while not DBC-specific, can still be used to good effect. For example, Elixir uses *guard clauses* to dispatch function calls against several available bodies:

```
defmodule Deposits do
  def accept_deposit(account_id, amount) when (amount > 100000) do
    # Call the manager!
  end
  def accept_deposit(account_id, amount) when (amount > 10000) do
    # Extra Federal requirements for reporting
    # Some processing...
  end
  def accept_deposit(account_id, amount) when (amount > 0) do
    # Some processing...
  end
end
```

In this case, calling `accept_deposit` with a large enough amount may trigger additional steps and processing. Try to call it with an amount less than or equal to zero, however, and you'll get an exception informing you that you can't:

```
** (FunctionClauseError) no function clause matching in Deposits.accept_deposit/2
```

This is a better approach than simply checking your inputs; in this case, you simply *can not* call this function if your arguments are out of range.

Tip 37 Design with Contracts

In *Orthogonality*, we recommended writing "shy" code. Here, the emphasis is on "lazy" code: be strict in what you will accept before you begin, and promise as little as possible in return. Remember, if your contract indicates that you'll accept anything and promise the world in return, then you've got a lot of code to write!

In any programming language, whether it's functional, object-oriented, or procedural, DBC forces you to *think.*

Class Invariants and Functional Languages

It's a naming thing. Eiffel is an object-oriented language, so Meyer named this idea "class invariant." But, really, it's more general than that. What this idea really refers to is *state*. In an object-oriented language, the state is associated with instances of classes. But other languages have state, too.

In a functional language, you typically pass state to functions and receive updated state as a result. The concepts of invariants is just as useful in these circumstances.

DBC and Test-Driven Development

Is Design by Contract needed in a world where developers practice unit testing, test-driven development (TDD), property-based testing, or defensive programming?

The short answer is "yes."

DBC and testing are different approaches to the broader topic of program correctness. They both have value and both have uses in different situations. DBC offers several advantages over specific testing approaches:

- DBC doesn't require any setup or mocking
- DBC defines the parameters for success or failure in *all* cases, whereas testing can only target one specific case at a time
- TDD and other testing happens only at "test time" within the build cycle. But DBC and assertions are forever: during design, development, deployment, and maintenance
- TDD does not focus on checking internal invariants within the code under test, it's more black-box style to check the public interface
- DBC is more efficient (and DRY-er) than defensive programming, where *everyone* has to validate data in case no one else does.

TDD is a great technique, but as with many techniques, it might invite you to concentrate on the "happy path," and not the real world full of bad data, bad actors, bad versions, and bad specifications.

Implementing DBC

Simply enumerating what the input domain range is, what the boundary conditions are, and what the routine promises to deliver—or, more importantly, what it *doesn't* promise to deliver—before you write the code is a huge leap forward in writing better software. By not stating these things, you are back to *programming by coincidence* (see the discussion on page 197), which is where many projects start, finish, and fail.

In languages that do not support DBC in the code, this might be as far as you can go—and that's not too bad. DBC is, after all, a *design* technique. Even without automatic checking, you can put the contract in the code as comments or in the unit tests and still get a very real benefit.

Assertions

While documenting these assumptions is a great start, you can get much greater benefit by having the compiler check your contract for you. You can partially emulate this in some languages by using *assertions:* runtime checks of logical conditions (see Topic 25, *Assertive Programming*, on page 115). Why only partially? Can't you use assertions to do everything DBC can do?

Unfortunately, the answer is no. To begin with, in object-oriented languages there probably is no support for propagating assertions down an inheritance hierarchy. This means that if you override a base class method that has a contract, the assertions that implement that contract will not be called correctly (unless you duplicate them manually in the new code). You must remember to call the class invariant (and all base class invariants) manually before you exit every method. The basic problem is that the contract is not automatically enforced.

In other environments, the exceptions generated from DBC-style assertions might be turned off globally or ignored in the code.

Also, there is no built-in concept of "old" values; that is, values as they existed at the entry to a method. If you're using assertions to enforce contracts, you must add code to the precondition to save any information you'll want to use in the postcondition, if the language will even allow that. In the Eiffel language, where DBC was born, you can just use `old` *expression*.

Finally, conventional runtime systems and libraries are not designed to support contracts, so these calls are not checked. This is a big loss, because it is often at the boundary between your code and the libraries it uses that the most problems are detected (see Topic 24, *Dead Programs Tell No Lies*, on page 112 for a more detailed discussion).

DBC and Crashing Early

DBC fits in nicely with our concept of crashing early (see Topic 24, *Dead Programs Tell No Lies*, on page 112). By using an assert or DBC mechanism to validate the preconditions, postconditions, and invariants, you can crash early and report more accurate information about the problem.

For example, suppose you have a method that calculates square roots. It needs a DBC precondition that restricts the domain to positive numbers. In languages that support DBC, if you pass `sqrt` a negative parameter, you'll get an informative error such as `sqrt_arg_must_be_positive`, along with a stack trace.

This is better than the alternative in other languages such as Java, C, and C++ where passing a negative number to `sqrt` returns the special value `NaN` (Not a Number). It may be some time later in the program that you attempt to do some math on `NaN`, with surprising results.

Who's Responsible?

Who is responsible for checking the precondition, the caller or the routine being called? When implemented as part of the language, the answer is neither: the precondition is tested behind the scenes after the caller invokes the routine but before the routine itself is entered. Thus if there is any explicit checking of parameters to be done, it must be performed by the *caller*, because the routine itself will never see parameters that violate its precondition. (For languages without built-in support, you would need to bracket the *called* routine with a preamble and/or postamble that checks these assertions.)

Consider a program that reads a number from the console, calculates its square root (by calling sqrt), and prints the result. The sqrt function has a precondition—its argument must not be negative. If the user enters a negative number at the console, it is up to the calling code to ensure that it never gets passed to sqrt. This calling code has many options: it could terminate, it could issue a warning and read another number, or it could make the number positive and append an *i* to the result returned by sqrt. Whatever its choice, this is definitely not sqrt's problem.

By expressing the domain of the square root function in the precondition of the sqrt routine, you shift the burden of correctness to the caller—where it belongs. You can then design the sqrt routine secure in the knowledge that its input will be in range.

It's much easier to find and diagnose the problem by crashing early, at the site of the problem.

Semantic Invariants

You can use *semantic invariants* to express inviolate requirements, a kind of "philosophical contract."

We once wrote a debit card transaction switch. A major requirement was that the user of a debit card should never have the same transaction applied to their account twice. In other words, no matter what sort of failure mode might happen, the error should be on the side of *not* processing a transaction rather than processing a duplicate transaction.

This simple law, driven directly from the requirements, proved to be very helpful in sorting out complex error recovery scenarios, and guided the detailed design and implementation in many areas.

Be sure not to confuse requirements that are fixed, inviolate laws with those that are merely policies that might change with a new management regime. That's why we use the term *semantic* invariants—it must be central to the very *meaning* of a thing, and not subject to the whims of policy (which is what more dynamic business rules are for).

When you find a requirement that qualifies, make sure it becomes a well-known part of whatever documentation you are producing—whether it is a bulleted list in the requirements document that gets signed in triplicate or just a big note on the common whiteboard that everyone sees. Try to state it clearly and unambiguously. For example, in the debit card example, we might write

> Err in favor of the consumer.

This is a clear, concise, unambiguous statement that's applicable in many different areas of the system. It is our contract with all users of the system, our guarantee of behavior.

Dynamic Contracts and Agents

Until now, we have talked about contracts as fixed, immutable specifications. But in the landscape of autonomous agents, this doesn't need to be the case. By the definition of "autonomous," agents are free to *reject* requests that they do not want to honor. They are free to renegotiate the contract—"I can't provide that, but if you give me this, then I might provide something else."

Certainly any system that relies on agent technology has a *critical* dependence on contractual arrangements—even if they are dynamically generated.

Imagine: with enough components and agents that can negotiate their own contracts among themselves to achieve a goal, we might just solve the software productivity crisis by letting software solve it for us.

But if we can't use contracts by hand, we won't be able to use them automatically. So next time you design a piece of software, design its contract as well.

Related Sections Include

- Topic 24, *Dead Programs Tell No Lies*, on page 112
- Topic 25, *Assertive Programming*, on page 115
- Topic 38, *Programming by Coincidence*, on page 197
- Topic 42, *Property-Based Testing*, on page 224
- Topic 43, *Stay Safe Out There*, on page 231
- Topic 45, *The Requirements Pit*, on page 244

Challenges

- Points to ponder: If DBC is so powerful, why isn't it used more widely? Is it hard to come up with the contract? Does it make you think about issues you'd rather ignore for now? Does it force you to *THINK!*? Clearly, this is a dangerous tool!

Exercises

Exercise 14 (possible answer on page 298)

Design an interface to a kitchen blender. It will eventually be a web-based, IoT-enabled blender, but for now we just need the interface to control it. It has ten speed settings (0 means off). You can't operate it empty, and you can change the speed only one unit at a time (that is, from 0 to 1, and from 1 to 2, not from 0 to 2).

Here are the methods. Add appropriate pre- and postconditions and an invariant.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

Exercise 15 (possible answer on page 299)

How many numbers are in the series 0, 5, 10, 15, …, 100?

24 Dead Programs Tell No Lies

Have you noticed that sometimes other people can detect that things aren't well with you before you're aware of the problem yourself? It's the same with other people's code. If something is starting to go awry with one of our programs, sometimes it is a library or framework routine that catches it first. Maybe we've passed in a nil value, or an empty list. Maybe there's a missing key in that hash, or the value we thought contained a hash really contains a list instead. Maybe there was a network error or filesystem error that we didn't catch, and we've got empty or corrupted data. A logic error a couple of million instructions ago means that the selector for a case statement is no longer the expected 1, 2, or 3. We'll hit the default case unexpectedly. That's also one reason why each and every case/switch statement needs to have a default clause: we want to know when the "impossible" has happened.

It's easy to fall into the "it can't happen" mentality. Most of us have written code that didn't check that a file closed successfully, or that a trace statement got written as we expected. And all things being equal, it's likely that we didn't need to—the code in question wouldn't fail under any normal conditions. But we're coding defensively. We're making sure that the data is what we think it is, that the code in production is the code we think it is. We're checking that the correct versions of dependencies were actually loaded.

All errors give you information. You could convince yourself that the error can't happen, and choose to ignore it. Instead, Pragmatic Programmers tell themselves that if there is an error, something very, very bad has happened. Don't forget to Read the Damn Error Message (see *Coder in a Strange Land,* on page 91).

Catch and Release Is for Fish

Some developers feel that is it good style to catch or rescue all exceptions, re-raising them after writing some kind of message. Their code is full of things like this (where a bare raise statement reraises the current exception):

```
try do
   add_score_to_board(score);
rescue InvalidScore
   Logger.error("Can't add invalid score. Exiting");
   raise
rescue BoardServerDown
   Logger.error("Can't add score: board is down. Exiting");
   raise
rescue StaleTransaction
   Logger.error("Can't add score: stale transaction. Exiting");
   raise
end
```

Here's how Pragmatic Programmers would write this:

```
add_score_to_board(score);
```

We prefer it for two reasons. First, the application code isn't eclipsed by the error handling. Second, and perhaps more important, the code is less coupled. In the verbose example, we have to list every exception the add_score_to_board method could raise. If the writer of that method adds another exception, our code is subtly out of date. In the more pragmatic second version, the new exception is automatically propagated.

Tip 38 Crash Early

Crash, Don't Trash

One of the benefits of detecting problems as soon as you can is that you can crash earlier, and crashing is often the best thing you can do. The alternative may be to continue, writing corrupted data to some vital database or commanding the washing machine into its twentieth consecutive spin cycle.

The Erlang and Elixir languages embrace this philosophy. Joe Armstrong, inventor of Erlang and author of *Programming Erlang: Software for a Concurrent World [Arm07]*, is often quoted as saying, “Defensive programming is a waste of time. Let it crash!” In these environments, programs are designed to fail, but that failure is managed with *supervisors*. A supervisor is responsible for running code and knows what to do in case the code fails, which could include cleaning up after it, restarting it, and so on. What happens when the supervisor itself fails? Its own supervisor manages that event, leading to a design composed of *supervisor trees*. The technique is very effective and helps to account for the use of these languages in high-availability, fault-tolerant systems.

In other environments, it may be inappropriate simply to exit a running program. You may have claimed resources that might not get released, or you may need to write log messages, tidy up open transactions, or interact with other processes.

However, the basic principle stays the same—when your code discovers that something that was supposed to be impossible just happened, your program is no longer viable. Anything it does from this point forward becomes suspect, so terminate it as soon as possible.

A dead program normally does a lot less damage than a crippled one.

Related Sections Include

- Topic 20, *Debugging*, on page 88
- Topic 23, *Design by Contract*, on page 104
- Topic 25, *Assertive Programming*, on page 115
- Topic 26, *How to Balance Resources*, on page 118
- Topic 43, *Stay Safe Out There*, on page 231

25 Assertive Programming

There is a luxury in self-reproach. When we blame ourselves we feel no one else has a right to blame us.

➤ *Oscar Wilde, The Picture of Dorian Gray*

It seems that there's a mantra that every programmer must memorize early in his or her career. It is a fundamental tenet of computing, a core belief that we learn to apply to requirements, designs, code, comments, just about everything we do. It goes

> This can never happen…

"This application will never be used abroad, so why internationalize it?" "`count` can't be negative." "Logging can't fail."

Let's not practice this kind of self-deception, particularly when coding.

Tip 39 Use Assertions to Prevent the Impossible

Whenever you find yourself thinking "but of course that could never happen," add code to check it. The easiest way to do this is with assertions. In many language implementations, you'll find some form of `assert` that checks a Boolean condition.[2] These checks can be invaluable. If a parameter or a result should never be `null`, then check for it explicitly:

```
assert (result != null);
```

In the Java implementation, you can (and should) add a descriptive string:

```
assert result != null && result.size() > 0 : "Empty result from XYZ";
```

Assertions are also useful checks on an algorithm's operation. Maybe you've written a clever sort algorithm, named `my_sort`. Check that it works:

```
books = my_sort(find("scifi"))
assert(is_sorted?(books))
```

Don't use assertions in place of real error handling. Assertions check for things that should never happen: you don't want to be writing code such as the following:

2. In C and C++ these are usually implemented as macros. In Java, assertions are disabled by default. Invoke the Java VM with the `-enableassertions` flag to enable them, and leave them enabled.

```
puts("Enter 'Y' or 'N': ")
ans = gets[0] # Grab first character of response
assert((ch == 'Y') || (ch == 'N'))    # Very bad idea!
```

And just because most assert implementations will terminate the process when an assertion fails, there's no reason why versions you write should. If you need to free resources, catch the assertion's exception or trap the exit, and run your own error handler. Just make sure the code you execute in those dying milliseconds doesn't rely on the information that triggered the assertion failure in the first place.

Assertions and Side Effects

It's embarrassing when the code we add to detect errors actually ends up creating new errors. This can happen with assertions if evaluating the condition has side effects. For example, it would be a bad idea to code something such as

```
while (iter.hasMoreElements()) {
  assert(iter.nextElement() != null);
  Object obj = iter.nextElement();
  // ....
}
```

The .nextElement() call in the assertion has the side effect of moving the iterator past the element being fetched, and so the loop will process only half the elements in the collection. It would be better to write

```
while (iter.hasMoreElements()) {
  Object obj = iter.nextElement();
  assert(obj != null);
  // ....
}
```

This problem is a kind of Heisenbug[3]—debugging that changes the behavior of the system being debugged.

(We also believe that nowadays, when most languages have decent support for iterating functions over collections, this kind of explicit loop is unnecessary and bad form.)

Leave Assertions Turned On

There is a common misunderstanding about assertions. It goes something like this:

3. http://www.eps.mcgill.ca/jargon/jargon.html#heisenbug

> Assertions add some overhead to code. Because they check for things that should never happen, they'll get triggered only by a bug in the code. Once the code has been tested and shipped, they are no longer needed, and should be turned off to make the code run faster. Assertions are a debugging facility.

There are two patently wrong assumptions here. First, they assume that testing finds all the bugs. In reality, for any complex program you are unlikely to test even a minuscule percentage of the permutations your code will be put through. Second, the optimists are forgetting that your program runs in a dangerous world. During testing, rats probably won't gnaw through a communications cable, someone playing a game won't exhaust memory, and log files won't fill the storage partition. These things might happen when your program runs in a production environment. Your first line of defense is checking for any possible error, and your second is using assertions to try to detect those you've missed.

Turning off assertions when you deliver a program to production is like crossing a high wire without a net because you once made it across in practice. There's dramatic value, but it's hard to get life insurance.

Even if you *do* have performance issues, turn off only those assertions that really hit you. The sort example above may be a critical part of your application, and may need to be fast. Adding the check means another pass through the data, which might be unacceptable. Make that particular check optional, but leave the rest in.

Use Assertions in Production, Win Big Money

A former neighbor of Andy's headed up a small startup company that made network devices. One of their secrets to success was the decision to leave assertions in place in production releases. These assertions were well crafted to report all the pertinent data leading to the failure, and presented via a nice-looking UI to the end user. This level of feedback, from real users under actual conditions, allowed the developers to plug the holes and fix these obscure, hard-to-reproduce bugs, resulting in remarkably stable, bullet-proof software.

This small, unknown company had such a solid product, it was soon acquired for hundreds of millions of dollars.

Just sayin'.

Exercise 16 (possible answer on page 299)

A quick reality check. Which of these "impossible" things can happen?

- A month with fewer than 28 days
- Error code from a system call: can't access the current directory
- In C++: `a = 2; b = 3;` but `(a + b)` does not equal `5`
- A triangle with an interior angle sum ≠ 180°
- A minute that doesn't have 60 seconds
- `(a + 1) <= a`

Related Sections Include

- Topic 23, *Design by Contract*, on page 104
- Topic 24, *Dead Programs Tell No Lies*, on page 112
- Topic 42, *Property-Based Testing*, on page 224
- Topic 43, *Stay Safe Out There*, on page 231

26 How to Balance Resources

To light a candle is to cast a shadow…

➤ *Ursula K. Le Guin, A Wizard of Earthsea*

We all manage resources whenever we code: memory, transactions, threads, network connections, files, timers—all kinds of things with limited availability. Most of the time, resource usage follows a predictable pattern: you allocate the resource, use it, and then deallocate it.

However, many developers have no consistent plan for dealing with resource allocation and deallocation. So let us suggest a simple tip:

Tip 40 Finish What You Start

This tip is easy to apply in most circumstances. It simply means that the function or object that allocates a resource should be responsible for deallocating it. Let's see how it applies by looking at an example of some bad code—part of a Ruby program that opens a file, reads customer information from it, updates a field, and writes the result back. We've eliminated error handling to make the example clearer:

```
def read_customer
  @customer_file = File.open(@name + ".rec", "r+")
  @balance       = BigDecimal(@customer_file.gets)
end

def write_customer
  @customer_file.rewind
  @customer_file.puts @balance.to_s
  @customer_file.close
end

def update_customer(transaction_amount)
  read_customer
  @balance = @balance.add(transaction_amount,2)
  write_customer
end
```

At first sight, the routine update_customer looks reasonable. It seems to implement the logic we require—reading a record, updating the balance, and writing the record back out. However, this tidiness hides a major problem. The routines read_customer and write_customer are tightly coupled[4]—they share the instance variable customer_file. read_customer opens the file and stores the file reference in customer_file, and then write_customer uses that stored reference to close the file when it finishes. This shared variable doesn't even appear in the update_customer routine.

Why is this bad? Let's consider the unfortunate maintenance programmer who is told that the specification has changed—the balance should be updated only if the new value is not negative. They go into the source and change update_customer:

```
def update_customer(transaction_amount)
  read_customer
  if (transaction_amount >= 0.00)
    @balance = @balance.add(transaction_amount,2)
    write_customer
  end
end
```

All seems fine during testing. However, when the code goes into production, it collapses after several hours, complaining of *too many open files*. It turns out that write_customer is not getting called in some circumstances. When that happens, the file is not getting closed.

A very *bad* solution to this problem would be to deal with the special case in update_customer:.

4. For a discussion of the dangers of coupled code, see Topic 28, *Decoupling*, on page 130.

```
def update_customer(transaction_amount)
  read_customer
  if (transaction_amount >= 0.00)
    @balance += BigDecimal(transaction_amount, 2)
    write_customer
  else
    @customer_file.close # Bad idea!
  end
end
```

This will fix the problem—the file will now get closed regardless of the new balance—but the fix now means that *three* routines are coupled through the shared variable customer_file, and keeping track of when the file is open or not is going to start to get messy. We're falling into a trap, and things are going to start going downhill rapidly if we continue on this course. This is not balanced!

The *finish what you start* tip tells us that, ideally, the routine that allocates a resource should also free it. We can apply it here by refactoring the code slightly:

```
def read_customer(file)
  @balance=BigDecimal(file.gets)
end

def write_customer(file)
  file.rewind
  file.puts @balance.to_s
end

def update_customer(transaction_amount)
  file=File.open(@name + ".rec", "r+")          # >--
  read_customer(file)                           #    |
  @balance = @balance.add(transaction_amount,2) #    |
  file.close                                    # <--
end
```

Instead of holding on to the file reference, we've changed the code to pass it as a parameter.[5] Now all the responsibility for the file is in the update_customer routine. It opens the file and (finishing what it starts) closes it before returning. The routine balances the use of the file: the open and close are in the same place, and it is apparent that for every open there will be a corresponding close. The refactoring also removes an ugly shared variable.

There's another small but important improvement we can make. In many modern languages, you can scope the lifetime of a resource to an enclosed

5. See the tip on page 153.

block of some sort. In Ruby, there's a variation of the file open that passes in the open file reference to a block, shown here between the do and the end:

```
def update_customer(transaction_amount)
  File.open(@name + ".rec", "r+") do |file|       # >--
    read_customer(file)                           #    |
    @balance = @balance.add(transaction_amount,2) #    |
    write_customer(file)                          #    |
  end                                             # <--
end
```

In this case, at the end of the block the file variable goes out of scope and the external file is closed. Period. No need to remember to close the file and release the source, it is guaranteed to happen for you.

When in doubt, it always pays to reduce scope.

Tip 41 Act Locally

Nest Allocations

The basic pattern for resource allocation can be extended for routines that need more than one resource at a time. There are just two more suggestions:

- Deallocate resources in the opposite order to that in which you allocate them. That way you won't orphan resources if one resource contains references to another.
- When allocating the same set of resources in different places in your code, always allocate them in the same order. This will reduce the possibility of deadlock. (If process A claims resource1 and is about to claim resource2, while process B has claimed resource2 and is trying to get resource1, the two processes will wait forever.)

It doesn't matter what kind of resources we're using—transactions, network connections, memory, files, threads, windows—the basic pattern applies: whoever allocates a resource should be responsible for deallocating it. However, in some languages we can develop the concept further.

Balancing Over Time

In this topic we're mostly looking at ephemeral resources used by your running process. But you might want to consider what other messes you might be leaving behind.

For instance, how are your logging files handled? You are creating data and using up storage space. Is there something in place to rotate the logs and clean them up? How about for your unofficial debug files you're dropping? If you're adding logging records in a database, is there a similar process in place to expire them? For anything that you create that takes up a finite resource, consider how to balance it.

What else are you leaving behind?

Objects and Exceptions

The equilibrium between allocations and deallocations is reminiscent of an object-oriented class's constructor and destructor. The class represents a resource, the constructor gives you a particular object of that resource type, and the destructor removes it from your scope.

If you are programming in an object-oriented language, you may find it useful to encapsulate resources in classes. Each time you need a particular resource type, you instantiate an object of that class. When the object goes out of scope, or is reclaimed by the garbage collector, the object's destructor then deallocates the wrapped resource.

This approach has particular benefits when you're working with languages where exceptions can interfere with resource deallocation.

Balancing and Exceptions

Languages that support exceptions can make resource deallocation tricky. If an exception is thrown, how do you guarantee that everything allocated prior to the exception is tidied up? The answer depends to some extent on the language support. You generally have two choices:

1. Use variable scope (for example, stack variables in C++ or Rust)
2. Use a `finally` clause in a `try`…`catch` block

With usual scoping rules in languages such as C++ or Rust, the variable's memory will be reclaimed when the variable goes out of scope via a return, block exit, or exception. But you can also hook in to the variable's destructor to cleanup any external resources. In this example, the Rust variable named `accounts` will automatically close the associated file when it goes out of scope:

```
{
  let mut accounts = File::open("mydata.txt")?; // >--
  // use 'accounts'                             //    |
  ...                                           //    |
}                                               // <--
// 'accounts' is now out of scope, and the file is
// automatically closed
```

The other option, if the language supports it, is the finally clause. A finally clause will ensure that the specified code will run whether or not an exception was raised in the try…catch block:

```
try
  // some dodgy stuff
catch
  // exception was raised
finally
  // clean up in either case
```

However, there is a catch.

An Exception Antipattern

We commonly see folks writing something like this:

```
begin
   thing = allocate_resource()
   process(thing)
finally
   deallocate(thing)
end
```

Can you see what's wrong?

What happens if the resource allocation fails and raises an exception? The finally clause will catch it, and try to deallocate a *thing* that was never allocated.

The correct pattern for handling resource deallocation in an environment with exceptions is

```
thing = allocate_resource()
begin
   process(thing)
finally
   deallocate(thing)
end
```

When You Can’t Balance Resources

There are times when the basic resource allocation pattern just isn’t appropriate. Commonly this is found in programs that use dynamic data structures. One routine will allocate an area of memory and link it into some larger structure, where it may stay for some time.

The trick here is to establish a semantic invariant for memory allocation. You need to decide who is responsible for data in an aggregate data structure. What happens when you deallocate the top-level structure? You have three main options:

- The top-level structure is also responsible for freeing any substructures that it contains. These structures then recursively delete data they contain, and so on.
- The top-level structure is simply deallocated. Any structures that it pointed to (that are not referenced elsewhere) are orphaned.
- The top-level structure refuses to deallocate itself if it contains any substructures.

The choice here depends on the circumstances of each individual data structure. However, you need to make it explicit for each, and implement your decision consistently. Implementing any of these options in a procedural language such as C can be a problem: data structures themselves are not active. Our preference in these circumstances is to write a module for each major structure that provides standard allocation and deallocation facilities for that structure. (This module can also provide facilities such as debug printing, serialization, deserialization, and traversal hooks.)

Checking the Balance

Because Pragmatic Programmers trust no one, including ourselves, we feel that it is always a good idea to build code that actually checks that resources are indeed freed appropriately. For most applications, this normally means producing wrappers for each type of resource, and using these wrappers to keep track of all allocations and deallocations. At certain points in your code, the program logic will dictate that the resources will be in a certain state: use the wrappers to check this. For example, a long-running program that services requests will probably have a single point at the top of its main processing loop where it waits for the next request to arrive. This is a good place to ensure that resource usage has not increased since the last execution of the loop.

At a lower, but no less useful level, you can invest in tools that (among other things) check your running programs for memory leaks.

Related Sections Include

- Topic 24, *Dead Programs Tell No Lies*, on page 112
- Topic 30, *Transforming Programming*, on page 147
- Topic 33, *Breaking Temporal Coupling*, on page 170

Challenges

- Although there are no guaranteed ways of ensuring that you always free resources, certain design techniques, when applied consistently, will help. In the text we discussed how establishing a semantic invariant for major data structures could direct memory deallocation decisions. Consider how Topic 23, *Design by Contract*, on page 104, could help refine this idea.

Exercise 17 (possible answer on page 299)

Some C and C++ developers make a point of setting a pointer to `NULL` after they deallocate the memory it references. Why is this a good idea?

Exercise 18 (possible answer on page 299)

Some Java developers make a point of setting an object variable to `NULL` after they have finished using the object. Why is this a good idea?

---

27 Don’t Outrun Your Headlights

It’s tough to make predictions, especially about the future.

➤ *Lawrence "Yogi" Berra, after a Danish Proverb*

It’s late at night, dark, pouring rain. The two-seater whips around the tight curves of the twisty little mountain roads, barely holding the corners. A hairpin comes up and the car misses it, crashing though the skimpy guardrail and soaring to a fiery crash in the valley below. State troopers arrive on the scene, and the senior officer sadly shakes their head. “Must have outrun their headlights.”

Had the speeding two-seater been going faster than the speed of light? No, that speed limit is firmly fixed. What the officer referred to was the driver’s ability to stop or steer in time in response to the headlight’s illumination.

Headlights have a certain limited range, known as the *throw distance*. Past that point, the light spread is too diffuse to be effective. In addition, headlights only project in a straight line, and won't illuminate anything off-axis, such as curves, hills, or dips in the road. According to the National Highway Traffic Safety Administration, the average distance illuminated by low-beam headlights is about 160 feet. Unfortunately, stopping distance at 40mph is 189 feet, and at 70mph a whopping 464 feet.[6] So indeed, it's actually pretty easy to outrun your headlights.

In software development, our "headlights" are similarly limited. We can't see too far ahead into the future, and the further off-axis you look, the darker it gets. So Pragmatic Programmers have a firm rule:

Tip 42 Take Small Steps—Always

Always take small, deliberate steps, checking for feedback and adjusting before proceeding. Consider that the rate of feedback is your speed limit. You never take on a step or a task that's "too big."

What do we mean exactly by feedback? Anything that independently confirms or disproves your action. For example:

- Results in a REPL provide feedback on your understanding of APIs and algorithms
- Unit tests provide feedback on your last code change
- User demo and conversation provide feedback on features and usability

What's a task that's too big? Any task that requires "fortune telling." Just as the car headlights have limited throw, we can only see into the future perhaps one or two steps, maybe a few hours or days at most. Beyond that, you can quickly get past *educated guess* and into *wild speculation.* You might find yourself slipping into fortune telling when you have to:

- Estimate completion dates months in the future
- Plan a design for future maintenance or extendability
- Guess user's future needs
- Guess future tech availability

But, we hear you cry, aren't we supposed to design for future maintenance? Yes, but only to a point: only as far ahead as you can see. The more you have to predict what the future will look like, the more risk you incur that you'll be wrong. Instead of wasting effort designing for an uncertain future, you can always fall back on designing your code to be replaceable. Make it easy to

6. Per the NHTSA, Stopping Distance = Reaction Distance + Braking Distance, assuming an average reaction time of 1.5s and deceleration of 17.02ft/s^2.

throw out your code and replace it with something better suited. Making code replaceable will also help with cohesion, decoupling, and DRY, leading to a better design overall.

Even though you may feel confident of the future, there's always the chance of a black swan around the corner.

Black Swans

In his book, *The Black Swan: The Impact of the Highly Improbable [Tal10]*, Nassim Nicholas Taleb posits that all significant events in history have come from high-profile, hard-to-predict, and rare events that are beyond the realm of normal expectations. These outliers, while statistically rare, have disproportionate effects. In addition, our own cognitive biases tend to blind us to changes creeping up on the edges of our work (see *Stone Soup and Boiled Frogs*).

Around the time of the first edition of *The Pragmatic Programmer*, debate raged in computer magazines and online forums over the burning question: "Who would win the desktop GUI wars, Motif or OpenLook?"[7] It was the wrong question. Odds are you've probably never heard of these technologies as neither "won" and the browser-centric web quickly dominated the landscape.

Tip 43 Avoid Fortune-Telling

Much of the time, tomorrow looks a lot like today. But don't count on it.

Related Sections Include

- Topic 12, *Tracer Bullets*, on page 50
- Topic 13, *Prototypes and Post-it Notes*, on page 56
- Topic 40, *Refactoring*, on page 209
- Topic 41, *Test to Code*, on page 214
- Topic 48, *The Essence of Agility*, on page 259
- Topic 50, *Coconuts Don't Cut It*, on page 270

7. Motif and OpenLook were GUI standards for X-Window based Unix workstations.

Dave Thomas and Andy Hunt are internationally recognized as leading voices in the software development community. They consult and speak around the world. Together, they founded the Pragmatic Bookshelf, publishing award-winning, leading-edge books for software developers. They were two of the authors of the Agile Manifesto.

Dave currently teaches college, turns wood, and plays with new technology and paradigms.

Andy writes science fiction, is an active musician, and loves to tinker with technology.

But, most of all, they're both driven to keep learning.

Dave

Andy

pragdave.me

toolshed.com