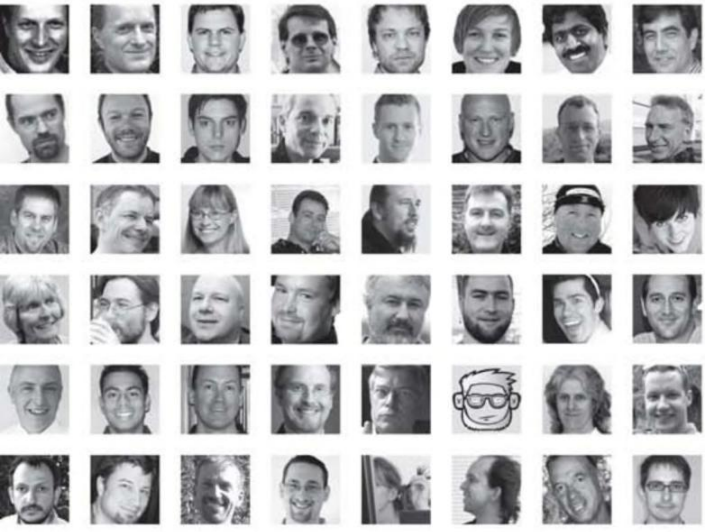
97



Collective Wisdom from the Experts

97 Things Every Programmer Should Know

O'REILLY®

Edited by Kevlin Henney

97 Things

Every Programmer Should Know

Collective Wisdom from the Experts

Edited by Kevlin Henney

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

97 Things Every Programmer Should Know
Edited by Kevlin Henney

Printed in the United States of America.

Published by O'Reilly Media, Inc. 1005 Gravenstein Highway North, Sebastopol CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Series Editor: Richard Monson-Haefel
Production Editor: Rachel Monaghan
Proofreader: Rachel Monaghan
Compositor: Ron Bilodeau
Indexer: Julie Hawks
Interior Designer: Ron Bilodeau
Cover Designers: Mark Paglietti and Susan Thompson

Print History:

February 2010: First Edition.

This book uses Repkover,™ a durable and flexible lay-flat binding.

ISBN: 978-0-596-80948-5

[SB]

Contents

Contributions by Category

Bugs and Fixes

Build and Deployment

Coding Guidelines and Code Layout

Design Principles and Coding Techniques

Domain Thinking

Errors, Error Handling, and Exceptions

Learning, Skills, and Expertise

Nocturnal or Magical

Performance, Optimization, and Representation

Professionalism, Mindset, and Attitude

Programming Languages and Paradigms

Refactoring and Code Care

Reuse Versus Repetition

Schedules, Deadlines, and Estimates

Simplicity

Teamwork and Collaboration

Tests, Testing, and Testers

Tools, Automation, and Development Environments

Users and Customers

Preface

The newest computer can merely compound, at speed, the oldest problem in the relations between human beings, and in the end the communicator will be confronted with the old problem, of what to say and how to say it.
—Edward R. Murrow

PROGRAMMERS HAVE A LOT ON THEIR MINDS. Programming languages, programming techniques, development environments, coding style, tools, development process, deadlines, meetings, software architecture, design patterns, team dynamics, code, requirements, bugs, code quality. And more. A lot.

There is an art, craft, and science to programming that extends far beyond the program. The act of programming marries the discrete world of computers with the fluid world of human affairs. Programmers mediate between the negotiated and uncertain truths of business and the crisp, uncompromising domain of bits and bytes and higher constructed types.

With so much to know, so much to do, and so many ways of doing so, no single person or single source can lay claim to "the one true way." Instead, *97 Things Every Programmer Should Know* draws on the wisdom of crowds and the voices of experience to offer not so much a coordinated big picture as a crowdsourced mosaic of what every programmer should know. This ranges from code-focused advice to culture, from algorithm usage to agile thinking, from implementation know-how to professionalism, from style to substance.

The contributions do not dovetail like modular parts, and there is no intent that they should—if anything, the opposite is true. The value of each contribution comes from its distinctiveness. The value of the collection lies in how the contributions complement, confirm, and even contradict one another. There is no overarching narrative: it is for you to respond to, reflect on, and connect together what you read, weighing it against your own context, knowledge, and experience.

Permissions

The licensing of each contribution follows a nonrestrictive, open source model. Every contribution is freely available online and licensed under a Creative Commons Attribution 3.0 License, which means that you can use the individual contributions in your own work, as long as you give credit to the original author:

http://creativecommons.org/licenses/by/3.0/us/

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

On the web page for this book, we list errata and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596809485/

The companion website for this book, where you can find all the contributions, contributor biographies, and more, is at:

http://programmer.97things.oreilly.com

You can also follow news and updates about this book and the website on Twitter:

http://twitter.com/97TEPSK

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com/

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new

titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

Many people have contributed their time and their insight, both directly and indirectly, to the *97 Things Every Programmer Should Know* project. They all deserve credit.

Richard Monson-Haefel is the 97 Things series editor and also the editor of the first book in the series, *97 Things Every Software Architect Should Know*, to which I contributed. I would like to thank Richard for trailblazing the series concept and its open contribution approach, and for enthusiastically supporting my proposal for this book.

I would like to thank all those who devoted the time and effort to contribute items to this project: both the contributors whose items are published in this book and the others whose items were not selected, but whose items are also published on the website. The high quantity and quality of contributions made the final selection process very difficult—the hardcoded number in the book's title unfortunately meant there was no slack to accommodate just a few more. I am also grateful for the additional feedback, comments, and suggestions provided by Giovanni Asproni, Paul Colin Gloster, and Michael Hunger.

Thanks to O'Reilly for the support they have provided this project, from hosting the wiki that made it possible to seeing it all the way through to publication in book form. People at O'Reilly I would like to thank specifically are Mike Loukides, Laurel Ackerman, Edie Freedman, Ed Stephenson, and Rachel Monaghan.

It is not simply the case that the book's content was developed on the Web: the project was also publicized and popularized on the Web. I would like to thank all those who have tweeted, retweeted, blogged, and otherwise spread the word.

I would also like to thank my wife, Carolyn, for bringing order to my chaos, and to my two sons, Stefan and Yannick, for reclaiming some of the chaos.

I hope this book will provide you with information, insight, and inspiration.

Enjoy!

—Kevlin Henney

Act with Prudence

Seb Rose

> *Whatever you undertake, act with prudence and consider the consequences.*
>
> —Anon

NO MATTER HOW COMFORTABLE A SCHEDULE LOOKS at the beginning of an iteration, you can't avoid being under pressure some of the time. If you find yourself having to choose between "doing it right" and "doing it quick," it is often appealing to "do it quick" with the understanding that you'll come back and fix it later. When you make this promise to yourself, your team, and your customer, you mean it. But all too often, the next iteration brings new problems and you become focused on them. This sort of deferred work is known as *technical debt*, and it is not your friend. Specifically, Martin Fowler calls this *deliberate technical debt* in his taxonomy of technical debt,* and it should not be confused with *inadvertent technical debt*.

Technical debt is like a loan: you benefit from it in the short term, but you have to pay interest on it until it is fully paid off. Shortcuts in the code make it harder to add features or refactor your code. They are breeding grounds for defects and brittle test cases. The longer you leave it, the worse it gets. By the time you get around to undertaking the original fix, there may be a whole stack of not-quite-right design choices layered on top of the original problem, making the code much harder to refactor and correct. In fact, it is often only when things have got so bad that you *must* fix the original problem, that you actually do go back to fix it. And by then, it is often so hard to fix that you really can't afford the time or the risk.

* *http://martinfowler.com/bliki/TechnicalDebtQuadrant.html*

There are times when you must incur technical debt to meet a deadline or implement a thin slice of a feature. Try not to be in this position, but if the situation absolutely demands it, then go ahead. But (and this is a big *but*) you must track technical debt and pay it back quickly, or things go rapidly downhill. As soon as you make the decision to compromise, write a task card or log it in your issue-tracking system to ensure that it does not get forgotten.

If you schedule repayment of the debt in the next iteration, the cost will be minimal. Leaving the debt unpaid will accrue interest, and that interest should be tracked to make the cost visible. This will emphasize the effect on business value of the project's technical debt and enables appropriate prioritization of the repayment. The choice of how to calculate and track the interest will depend on the particular project, but track it you must.

Pay off technical debt as soon as possible. It would be imprudent to do otherwise.

Apply Functional Programming Principles

Edward Garson

FUNCTIONAL PROGRAMMING has recently enjoyed renewed interest from the mainstream programming community. Part of the reason is because *emergent properties* of the functional paradigm are well positioned to address the challenges posed by our industry's shift toward multicore. However, while that is certainly an important application, it is not the reason this piece admonishes you to *know thy functional programming*.

Mastery of the functional programming paradigm can greatly improve the quality of the code you write in other contexts. If you deeply understand and apply the functional paradigm, your designs will exhibit a much higher degree of *referential transparency*.

Referential transparency is a very desirable property: it implies that functions consistently yield the same results given the same input, irrespective of where and when they are invoked. That is, function evaluation depends less—ideally, not at all—on the side effects of mutable state.

A leading cause of defects in imperative code is attributable to mutable variables. Everyone reading this will have investigated why some value is not as expected in a particular situation. Visibility semantics can help to mitigate these insidious defects, or at least to drastically narrow down their location, but their true culprit may in fact be the providence of designs that employ inordinate mutability.

And we certainly don't get much help from the industry in this regard. Introductions to object orientation tacitly promote such design, because they often show examples composed of graphs of relatively long-lived objects that happily call mutator methods on one another, which can be dangerous.

However, with astute test-driven design, particularly when being sure to "Mock Roles, not Objects,"* unnecessary mutability can be designed away.

The net result is a design that typically has better responsibility allocation with more numerous, smaller functions that act on arguments passed into them, rather than referencing mutable member variables. There will be fewer defects, and furthermore they will often be simpler to debug, because it is easier to locate where a rogue value is introduced in these designs than to otherwise deduce the particular context that results in an erroneous assignment. This adds up to a *much higher degree* of referential transparency, and positively nothing will get these ideas as deeply into your bones as learning a functional programming language, where this model of computation is the norm.

Of course, this approach is not optimal in all situations. For example, in object-oriented systems, this style often yields better results with domain model development (i.e., where collaborations serve to break down the complexity of business rules) than with user-interface development.

Master the functional programming paradigm so you are able to judiciously apply the lessons learned to other domains. Your object systems (for one) will resonate with referential transparency goodness and be much closer to their functional counterparts than many would have you believe. In fact, some would even assert that, at their apex, functional programming and object orientation are *merely a reflection of each other*, a form of computational yin and yang.

* *http://www.jmock.org/oopsla2004.pdf*

Ask, "What Would the User Do?" (You Are Not the User)

Giles Colborne

WE ALL TEND TO ASSUME THAT OTHER PEOPLE THINK LIKE US. But they don't. Psychologists call this the *false consensus bias*. When people think or act differently from us, we're quite likely to label them (subconsciously) as defective in some way.

This bias explains why programmers have such a hard time putting themselves in the users' position. Users don't think like programmers. For a start, they spend much less time using computers. They neither know nor care how a computer works. This means they can't draw on any of the battery of problem-solving techniques so familiar to programmers. They don't recognize the patterns and cues programmers use to work with, through, and around an interface.

The best way to find out how a user thinks is to watch one. Ask a user to complete a task using a similar piece of software to what you're developing. Make sure the task is a real one: "Add up a column of numbers" is OK; "Calculate your expenses for the last month" is better. Avoid tasks that are too specific, such as "Can you select these spreadsheet cells and enter a *SUM* formula below?"—there's a big clue in that question. Get the user to talk through his or her progress. Don't interrupt. Don't try to help. Keep asking yourself, "Why is he doing that?" and "Why is she not doing that?"

The first thing you'll notice is that users do a core of things similarly. They try to complete tasks in the same order—and they make the same mistakes in the same places. You should design around that core behavior. This is different from design meetings, where people tend to listen when someone says, "What if the user wants to…?" This leads to elaborate features and confusion over what users want. Watching users eliminates this confusion.

You'll see users getting stuck. When you get stuck, you look around. When users get stuck, they narrow their focus. It becomes harder for them to see solutions elsewhere on the screen. It's one reason why help text is a poor solution to poor user interface design. If you must have instructions or help text, make sure to locate it right next to your problem areas. A user's narrow focus of attention is why tool tips are more useful than help menus.

Users tend to muddle through. They'll find a way that works and stick with it, no matter how convoluted. It's better to provide one really obvious way of doing things than two or three shortcuts.

You'll also find that there's a gap between what users say they want and what they actually do. That's worrying, as the normal way of gathering user requirements is to ask them. It's why the best way to capture requirements is to watch users. Spending an hour watching users is more informative than spending a day guessing what they want.

Automate Your Coding Standard

Filip van Laenen

YOU'VE PROBABLY BEEN THERE, TOO. At the beginning of a project, everybody has lots of good intentions—call them "new project's resolutions." Quite often, many of these resolutions are written down in documents. The ones about code end up in the project's coding standard. During the kick-off meeting, the lead developer goes through the document and, in the best case, everybody agrees that they will try to follow them. Once the project gets underway, though, these good intentions are abandoned, one at a time. When the project is finally delivered, the code looks like a mess, and nobody seems to know how it came to be that way.

When did things go wrong? Probably already at the kick-off meeting. Some of the project members didn't pay attention. Others didn't understand the point. Worse, some disagreed and were already planning their coding standard rebellion. Finally, some got the point and agreed, but when the pressure in the project got too high, they had to let something go. Well-formatted code doesn't earn you points with a customer that wants more functionality. Furthermore, following a coding standard can be quite a boring task if it isn't automated. Just try to indent a messy class by hand to find out for yourself.

But if it's such a problem, why is it that we want a coding standard in the first place? One reason to format the code in a uniform way is so that nobody can "own" a piece of code just by formatting it in his or her private way. We may want to prevent developers from using certain antipatterns in order to avoid some common bugs. In all, a coding standard should make it easier to work in the project, and maintain development speed from the beginning to the end. It follows, then, that everybody should agree on the coding standard, too—it does not help if one developer uses three spaces to indent code, and another uses four.

There exists a wealth of tools that can be used to produce code quality reports and to document and maintain the coding standard, but that isn't the whole solution. It should be automated and enforced where possible. Here are a few examples:

- Make sure code formatting is part of the build process, so that everybody runs it automatically every time they compile the code.
- Use static code analysis tools to scan the code for unwanted antipatterns. If any are found, break the build.
- Learn to configure those tools so that you can scan for your own, project-specific antipatterns.
- Do not only measure test coverage, but automatically check the results, too. Again, break the build if test coverage is too low.

Try to do this for everything that you consider important. You won't be able to automate everything you really care about. As for the things that you can't automatically flag or fix, consider them a set of guidelines supplementary to the coding standard that is automated, but accept that you and your colleagues may not follow them as diligently.

Finally, the coding standard should be dynamic rather than static. As the project evolves, the needs of the project change, and what may have seemed smart in the beginning isn't necessarily smart a few months later.

Beauty Is in Simplicity

Jørn Ølmheim

THERE IS ONE QUOTE, from Plato, that I think is particularly good for all software developers to know and keep close to their hearts:

> *Beauty of style and harmony and grace and good rhythm depends on simplicity.*

In one sentence, this sums up the values that we as software developers should aspire to.

There are a number of things we strive for in our code:

- Readability
- Maintainability
- Speed of development
- The elusive quality of beauty

Plato is telling us that the enabling factor for all of these qualities is simplicity.

What is beautiful code? This is potentially a very subjective question. Perception of beauty depends heavily on individual background, just as much of our perception of anything depends on our background. People educated in the arts have a different perception of (or at least approach to) beauty than people educated in the sciences. Arts majors tend to approach beauty in software by comparing software to works of art, while science majors tend to talk about symmetry and the golden ratio, trying to reduce things to formulae. In my experience, simplicity is the foundation of most of the arguments from both sides.

Think about source code that you have studied. If you haven't spent time studying other people's code, stop reading this right now and find some open source code to study. Seriously! I mean it! Go search the Web for some code in your language of choice, written by some well-known, acknowledged expert.

You're back? Good. Where were we? Ah, yes…I have found that code that resonates with me, and that I consider beautiful, has a number of properties in common. Chief among these is simplicity. I find that no matter how complex the total application or system is, the individual parts have to be kept simple: simple objects with a single responsibility containing similarly simple, focused methods with descriptive names. Some people think the idea of having short methods of 5–10 lines of code is extreme, and some languages make it very hard to do, but I think that such brevity is a desirable goal nonetheless.

The bottom line is that beautiful code is simple code. Each individual part is kept simple with simple responsibilities and simple relationships with the other parts of the system. This is the way we can keep our systems maintainable over time, with clean, simple, testable code, ensuring a high speed of development throughout the lifetime of the system.

Beauty is born of and found in simplicity.

Before You Refactor

Rajith Attapattu

AT SOME POINT, every programmer will need to refactor existing code. But before you do so, please think about the following, as this could save you and others a great deal of time (and pain):

- *The best approach for restructuring starts by taking stock of the existing codebase and the tests written against that code.* This will help you understand the strengths and weaknesses of the code as it currently stands, so you can ensure that you retain the strong points while avoiding the mistakes. We all think we can do better than the existing system…until we end up with something no better—or even worse—than the previous incarnation because we failed to learn from the existing system's mistakes.
- *Avoid the temptation to rewrite everything.* It is best to reuse as much code as possible. No matter how ugly the code is, it has already been tested, reviewed, etc. Throwing away the old code—especially if it was in production—means that you are throwing away months (or years) of tested, battle-hardened code that may have had certain workarounds and bug fixes you aren't aware of. If you don't take this into account, the new code you write may end up showing the same mysterious bugs that were fixed in the old code. This will waste a lot of time, effort, and knowledge gained over the years.
- *Many incremental changes are better than one massive change.* Incremental changes allows you to gauge the impact on the system more easily through feedback, such as from tests. It is no fun to see a hundred test failures after you make a change. This can lead to frustration and pressure that can in turn result in bad decisions. A couple of test failures at a time is easier to deal with, leading to a more manageable approach.

- *After each development iteration, it is important to ensure that the existing tests pass*. Add new tests if the existing tests are not sufficient to cover the changes you made. Do not throw away the tests from the old code without due consideration. On the surface, some of these tests may not appear to be applicable to your new design, but it would be well worth the effort to dig deep down into the reasons why this particular test was added.
- *Personal preferences and ego shouldn't get in the way*. If something isn't broken, why fix it? That the style or the structure of the code does not meet your personal preference is not a valid reason for restructuring. Thinking you could do a better job than the previous programmer is not a valid reason, either.
- *New technology is an insufficient reason to refactor*. One of the worst reasons to refactor is because the current code is way behind all the cool technology we have today, and we believe that a new language or framework can do things a lot more elegantly. Unless a cost-benefit analysis shows that a new language or framework will result in significant improvements in functionality, maintainability, or productivity, it is best to leave it as it is.
- *Remember that humans make mistakes*. Restructuring will not always guarantee that the new code will be better—or even as good as—the previous attempt. I have seen and been a part of several failed restructuring attempts. It wasn't pretty, but it was human.

Beware the Share

Udi Dahan

IT WAS MY FIRST PROJECT AT THE COMPANY. I'd just finished my degree and was anxious to prove myself, staying late every day going through the existing code. As I worked through my first feature, I took extra care to put in place everything I had learned—commenting, logging, pulling out shared code into libraries where possible, the works. The code review that I had felt so ready for came as a rude awakening—reuse was frowned upon!

How could this be? Throughout college, reuse was held up as the epitome of quality software engineering. All the articles I had read, the textbooks, the seasoned software professionals who taught me—was it all wrong?

It turns out that I was missing something critical.

Context.

The fact that two wildly different parts of the system performed some logic in the same way meant less than I thought. Up until I had pulled out those libraries of shared code, these parts were not dependent on each other. Each could evolve independently. Each could change its logic to suit the needs of the system's changing business environment. Those four lines of similar code were accidental—a temporal anomaly, a coincidence. That is, until I came along.

The libraries of shared code I created tied the shoelaces of each foot to the other. Steps by one business domain could not be made without first synchronizing with the other. Maintenance costs in those independent functions used to be negligible, but the common library required an order of magnitude more testing.

While I'd decreased the absolute number of lines of code in the system, I had increased the number of dependencies. The context of these dependencies is critical—had they been localized, the sharing may have been justified and had some positive value. When these dependencies aren't held in check, their tendrils entangle the larger concerns of the system, even though the code itself looks just fine.

These mistakes are insidious in that, at their core, they sound like a good idea. When applied in the right context, these techniques are valuable. In the wrong context, they increase cost rather than value. When coming into an existing codebase with no knowledge of where the various parts will be used, I'm much more careful these days about what is shared.

Beware the share. Check your context. Only then, proceed.

The Boy Scout Rule

Robert C. Martin (Uncle Bob)

THE BOY SCOUTS HAVE A RULE: "Always leave the campground cleaner than you found it." If you find a mess on the ground, you clean it up regardless of who might have made it. You intentionally improve the environment for the next group of campers. (Actually, the original form of that rule, written by Robert Stephenson Smyth Baden-Powell, the father of scouting, was "Try and leave this world a little better than you found it.")

What if we followed a similar rule in our code: "Always check a module in cleaner than when you checked it out"? Regardless of who the original author was, what if we always made some effort, no matter how small, to improve the module? What would be the result?

I think if we all followed that simple rule, we would see the end of the relentless deterioration of our software systems. Instead, our systems would gradually get better and better as they evolved. We would also see *teams* caring for the system as a whole, rather than just individuals caring for their own small part.

I don't think this rule is too much to ask. You don't have to make every module perfect before you check it in. You simply have to make it *a little bit better* than when you checked it out. Of course, this means that any code you *add* to a module must be clean. It also means that you clean up at least one other thing before you check the module back in. You might simply improve the name of one variable, or split one long function into two smaller functions. You might break a circular dependency, or add an interface to decouple policy from detail.

Frankly, this just sounds like common decency to me—like washing your hands after you use the restroom, or putting your trash in the bin instead of dropping it on the floor. Indeed, the act of leaving a mess in the code should be as socially unacceptable as littering. It should be something that *just isn't done*.

But it's more than that. Caring for our own code is one thing. Caring for the *team*'s code is quite another. Teams help one another and clean up after one another. They follow the Boy Scout rule because it's good for everyone, not just good for themselves.

Check Your Code First Before Looking to Blame Others

Allan Kelly

DEVELOPERS—ALL OF US!—often have trouble believing our own code is broken. It is just so improbable that, for once, it must be the compiler that's broken.

Yet, in truth, it is very (very) unusual that code is broken by a bug in the compiler, interpreter, OS, app server, database, memory manager, or any other piece of system software. Yes, these bugs exist, but they are far less common than we might like to believe.

I once had a genuine problem with a compiler bug optimizing away a loop variable, but I have imagined my compiler or OS had a bug many more times. I have wasted a lot of my time, support time, and management time in the process, only to feel a little foolish each time it turned out to be my mistake after all.

Assuming that the tools are widely used, mature, and employed in various technology stacks, there is little reason to doubt the quality. Of course, if the tool is an early release, or used by only a few people worldwide, or a piece of seldom downloaded, version 0.1, open source software, there may be good reason to suspect the software. (Equally, an alpha version of commercial software might be suspect.)

Given how rare compiler bugs are, you are far better putting your time and energy into finding the error in your code than into proving that the compiler is wrong. All the usual debugging advice applies, so isolate the problem, stub out calls, and surround it with tests; check calling conventions, shared libraries, and version numbers; explain it to someone else; look out for stack corruption and variable type mismatches; and try the code on different machines and different build configurations, such as debug and release.

Question your own assumptions and the assumptions of others. Tools from different vendors might have different assumptions built into them—so too might different tools from the same vendor.

When someone else is reporting a problem you cannot duplicate, go and see what they are doing. They may be doing something you never thought of or are doing something in a different order.

My personal rule is that if I have a bug I can't pin down, and I'm starting to think it's the compiler, then it's time to look for stack corruption. This is especially true if adding trace code makes the problem move around.

Multithreaded problems are another source of bugs that turn hair gray and induce screaming at the machine. All the recommendations to favor simple code are multiplied when a system is multithreaded. Debugging and unit tests cannot be relied on to find such bugs with any consistency, so simplicity of design is paramount.

So, before you rush to blame the compiler, remember Sherlock Holmes's advice, "Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth," and opt for it over Dirk Gently's, "Once you eliminate the improbable, whatever remains, no matter how impossible, must be the truth."

Choose Your Tools with Care

Giovanni Asproni

MODERN APPLICATIONS ARE VERY RARELY BUILT FROM SCRATCH. They are assembled using existing tools—components, libraries, and frameworks—for a number of good reasons:

- Applications grow in size, complexity, and sophistication, while the time available to develop them grows shorter. It makes better use of developers' time and intelligence if they can concentrate on writing more business-domain code and less infrastructure code.
- Widely used components and frameworks are likely to have fewer bugs than the ones developed in-house.
- There is a lot of high-quality software available on the Web for free, which means lower development costs and greater likelihood of finding developers with the necessary interest and expertise.
- Software production and maintenance is human-intensive work, so buying may be cheaper than building.

However, choosing the right mix of tools for your application can be a tricky business requiring some thought. In fact, when making a choice, you should keep in mind a few things:

- Different tools may rely on different assumptions about their context—e.g., surrounding infrastructure, control model, data model, communication protocols, etc.—which can lead to an *architectural mismatch* between the application and the tools. Such a mismatch leads to hacks and workarounds that will make the code more complex than necessary.
- Different tools have different lifecycles, and upgrading one of them may become an extremely difficult and time-consuming task since the new functionality, design changes, or even bug fixes may cause incompatibilities with

the other tools. The greater the number of tools, the worse the problem can become.

- Some tools require quite a bit of configuration, often by means of one or more XML files, which can grow out of control very quickly. The application may end up looking as if it was all written in XML plus a few odd lines of code in some programming language. The configurational complexity will make the application difficult to maintain and to extend.
- Vendor lock-in occurs when code that depends heavily on specific vendor products ends up being constrained by them on several counts: maintainability, performances, ability to evolve, price, etc.
- If you plan to use free software, you may discover that it's not so free after all. You may need to buy commercial support, which is not necessarily going to be cheap.
- Licensing terms matter, even for free software. For example, in some companies, it is not acceptable to use software licensed under the GNU license terms because of its viral nature—i.e., software developed with it must be distributed along with its source code.

My personal strategy to mitigate these problems is to start small by using only the tools that are absolutely necessary. Usually the initial focus is on removing the need to engage in low-level infrastructure programming (and problems), e.g., by using some middleware instead of using raw sockets for distributed applications. And then add more if needed. I also tend to isolate the external tools from my business domain objects by means of interfaces and layering, so that I can change the tool if I have to with a minimal amount of pain. A positive side effect of this approach is that I generally end up with a smaller application that uses fewer external tools than originally forecast.

Code in the Language of the Domain

Dan North

PICTURE TWO CODEBASES. In one, you come across:

```
if (portfolioIdsByTraderId.get(trader.getId())
    .containsKey(portfolio.getId())) {...}
```

You scratch your head, wondering what this code might be for. It seems to be getting an ID from a trader object; using that to get a map out of a, well, map-of-maps, apparently; and then seeing if another ID from a portfolio object exists in the inner map. You scratch your head some more. You look for the declaration of `portfolioIdsByTraderId` and discover this:

```
Map<int, Map<int, int>> portfolioIdsByTraderId;
```

Gradually, you realize it might have something to do with whether a trader has access to a particular portfolio. And of course you will find the same lookup fragment—or, more likely, a similar but subtly different code fragment—whenever something cares whether a trader has access to a particular portfolio.

In the other codebase, you come across this:

```
if (trader.canView(portfolio)) {...}
```

No head scratching. You don't need to know how a trader knows. Perhaps there is one of these maps-of-maps tucked away somewhere inside. But that's the trader's business, not yours.

Now which of those codebases would you rather be working in?

Once upon a time, we only had very basic data structures: bits and bytes and characters (really just bytes, but we would pretend they were letters and punctuation). Decimals were a bit tricky because our base-10 numbers don't work very well in binary, so we had several sizes of floating-point types. Then came arrays and strings (really just different arrays). Then we had stacks and queues and hashes and linked lists and skip lists and lots of other exciting data structures that *don't exist in the real world*. "Computer science" was about spending

lots of effort mapping the real world into our restrictive data structures. The real gurus could even remember how they had done it.

Then we got user-defined types! OK, this isn't news, but it does change the game somewhat. If your domain contains concepts like traders and portfolios, you can model them with types called, say, `Trader` and `Portfolio`. But, more importantly than this, you can model *relationships between them* using domain terms, too.

If you don't code using domain terms, you are creating a tacit (read: secret) understanding that *this* `int` over here means the way to identify a trader, whereas *that* `int` over there means the way to identify a portfolio. (Best not to get them mixed up!) And if you represent a business concept ("Some traders are not allowed to view some portfolios—it's illegal") with an algorithmic snippet—say, an existence relationship in a map of keys—you aren't doing the audit and compliance guys any favors.

The next programmer to come along might not be in on the secret, so why not make it explicit? Using a key as a lookup to another key that performs an existence check is not terribly obvious. How is someone supposed to intuit that's where the business rules preventing conflict of interest are implemented?

Making domain concepts explicit in your code means other programmers can gather the intent of the code much more easily than by trying to retrofit an algorithm into what they understand about a domain. It also means that when the domain model evolves—which it will, as your understanding of the domain grows—you are in a good position to evolve the code. Coupled with good encapsulation, the chances are good that the rule will exist in only one place, and that you can change it without any of the dependent code being any the wiser.

The programmer who comes along a few months later to work on the code will thank you. The programmer who comes along a few months later might be you.

Code Is Design

Ryan Brush

IMAGINE WAKING UP TOMORROW and learning that the construction industry has made the breakthrough of the century. Millions of cheap, incredibly fast robots can fabricate materials out of thin air, have a near-zero power cost, and can repair themselves. And it gets better: given an unambiguous blueprint for a construction project, the robots can build it without human intervention, all at negligible cost.

One can imagine the impact on the construction industry, but what would happen upstream? How would the behavior of architects and designers change if construction costs were negligible? Today, physical and computer models are built and rigorously tested before investing in construction. Would we bother if the construction was essentially free? If a design collapses, no big deal—just find out what went wrong and have our magical robots build another one. There are further implications. With models obsolete, unfinished designs evolve by repeatedly building and improving upon an approximation of the end goal. A casual observer may have trouble distinguishing an unfinished design from a finished product.

Our ability to predict timelines will fade away. Construction costs are more easily calculated than design costs—we know the approximate cost of installing a girder, and how many girders we need. As predictable tasks shrink toward zero, the less predictable design time starts to dominate. Results are produced more quickly, but reliable timelines slip away.

Of course, the pressures of a competitive economy still apply. With construction costs eliminated, a company that can quickly complete a design gains an

edge in the market. Getting design done fast becomes the central push of engineering firms. Inevitably, someone not deeply familiar with the design will see an unvalidated version, see the market advantage of releasing early, and say, "This looks good enough."

Some life-or-death projects will be more diligent, but in many cases, consumers learn to suffer through the incomplete design. Companies can always send out our magic robots to "patch" the broken buildings and vehicles they sell. All of this points to a startlingly counterintuitive conclusion: our sole premise was a dramatic reduction in construction costs, with the result that *quality got worse*.

It shouldn't surprise us that the preceding story has played out in software. If we accept that code is design—a creative process rather than a mechanical one—the *software crisis* is explained. We now have a *design crisis*: the demand for quality, validated designs exceeds our capacity to create them. The pressure to use incomplete design is strong.

Fortunately, this model also offers clues to how we can get better. Physical simulations equate to automated testing; software design isn't complete until it is validated with a brutal battery of tests. To make such tests more effective, we are finding ways to rein in the huge state space of large systems. Improved languages and design practices give us hope. Finally, there is one inescapable fact: great designs are produced by great designers dedicating themselves to the mastery of their craft. Code is no different.

Code Layout Matters

Steve Freeman

AN INFEASIBLE NUMBER OF YEARS AGO, I worked on a Cobol system where staff members weren't allowed to change the indentation unless they already had a reason to change the code, because someone once broke something by letting a line slip into one of the special columns at the beginning of a line. This applied even if the layout was misleading, which it sometimes was, so we had to read the code very carefully because we couldn't trust it. The policy must have cost a fortune in programmer drag.

There's research suggesting that we all spend much more of our programming time navigating and reading code—finding *where* to make the change—than actually typing, so that's what we want to optimize for. Here are three such optimizations:

Easy to scan

People are really good at visual pattern matching (a leftover trait from the time when we had to spot lions on the savannah), so I can help myself by making everything that isn't directly relevant to the domain—all the "accidental complexity" that comes with most commercial languages—fade into the background by standardizing it. If code that behaves the same looks the same, then my perceptual system will help me pick out the differences. That's why I also observe conventions about how to lay out the parts of a class within a compilation unit: constants, fields, public methods, private methods.

Expressive layout

> We've all learned to take the time to find the right names so that our code expresses as clearly as possible what it does, rather than just listing the steps—right? The code's layout is part of this expressiveness, too. A first cut is to have the team agree on an automatic formatter for the basics, and then I might make adjustments by hand while I'm coding. Unless there's active dissension, a team will quickly converge on a common "hand-finished" style. A formatter cannot understand my intentions (I should know, I once wrote one), and it's more important to me that the line breaks and groupings reflect the intention of the code, not just the syntax of the language. (Kevin McGuire freed me from my bondage to automatic code formatters.)

Compact format

> The more I can get on a screen, the more I can see without breaking context by scrolling or switching files, which means I can keep less state in my head. Long procedure comments and lots of whitespace made sense for eight-character names and line printers, but now I live in an IDE that does syntax coloring and cross linking. Pixels are my limiting factor, so I want every one to contribute to my understanding of the code. I want the layout to help me understand the code, but no more than that.

A nonprogrammer friend once remarked that code looks like poetry. I get that feeling from really good code—that everything in the text has a purpose, and that it's there to help me understand the idea. Unfortunately, writing code doesn't have the same romantic image as writing poetry.

Code Reviews

Mattias Karlsson

YOU SHOULD DO CODE REVIEWS. Why? Because they *increase code quality* and *reduce defect rate*. But not necessarily for the reasons you might think.

Because they may previously have had some bad experiences with code reviews, many programmers tend to dislike them. I have seen organizations that require that all code pass a formal review before being deployed to production. Often, it is the architect or a lead developer doing this review, a practice that can be described as *architect reviews everything*. This is stated in the company's software development process manual, so the programmers must comply.

There may be some organizations that need such a rigid and formal process, but most do not. In most organizations, such an approach is counterproductive. Reviewees can feel like they are being judged by a parole board. Reviewers need both the time to read the code and the time to keep up to date with all the details of the system; they can rapidly become the bottleneck in this process, and the process soon degenerates.

Instead of simply correcting mistakes in code, the purpose of code reviews should be to *share knowledge* and establish common coding guidelines. Sharing your code with other programmers enables collective code ownership. Let a random team member *walk through the code* with the rest of the team. Instead of looking for errors, you should review the code by trying to learn and understand it.

Be gentle during code reviews. Ensure that comments are *constructive, not caustic*. Introduce different *roles* for the review meeting to avoid having organizational seniority among team members affect the code review. Examples of roles could include having one reviewer focus on documentation, another on exceptions, and a third to look at the functionality. This approach helps to spread the review burden across the team members.

Have a regular *code review day* each week. Spend a couple of hours in a review meeting. Rotate the reviewee every meeting in a simple round-robin pattern. Remember to switch roles among team members every review meeting, too. *Involve newbies* in code reviews. They may be inexperienced, but their fresh university knowledge can provide a different perspective. *Involve experts* for their experience and knowledge. They will identify error-prone code faster and with more accuracy. Code reviews will flow more easily if the team has coding conventions that are checked by tools. That way, code formatting will never be discussed during the code review meeting.

Making code reviews fun is perhaps the most important contributor to success. Reviews are about the people reviewing. If the review meeting is painful or dull, it will be hard to motivate anyone. Make it an *informal code review* whose principal purpose is to share knowledge among team members. Leave sarcastic comments outside, and bring a cake or brown-bag lunch instead.

Coding with Reason

Yechiel Kimchi

TRYING TO REASON about software correctness by hand results in a formal proof that is longer than the code, and more likely to contain errors. Automated tools are preferable but not always possible. What follows describes a middle path: reasoning semiformally about correctness.

The underlying approach is to divide all the code under consideration into short sections—from a single line, such as a function call, to blocks of less than 10 lines—and argue about their correctness. The arguments need only be strong enough to convince your devil's advocate peer programmer.

A section should be chosen so that at each endpoint, the *state of the program* (namely, the program counter and the values of all "living" objects) satisfies an easily described property, and so that the functionality of that section (state transformation) is easy to describe as a single task; these guidelines will make reasoning simpler. Such endpoint properties generalize concepts like *preconditions* and *postconditions* for functions, and *invariants* for loops and classes (with respect to their instances). Striving for sections to be as independent of one another as possible simplifies reasoning and is indispensable when these sections are to be modified.

Many of the coding practices that are well known (although perhaps less well followed) and considered "good" make reasoning easier. Hence, just by *intending* to reason about your code, you already start moving toward a better style and structure. Unsurprisingly, most of these practices can be checked by static code analyzers:

- Avoid using *goto* statements, as they make remote sections highly interdependent.
- Avoid using modifiable global variables, as they make all sections that use them dependent.

- Each variable should have the smallest possible scope. For example, a local object can be declared right before its first usage.
- Make objects *immutable* whenever relevant.
- Make the code readable by using spacing, both horizontal and vertical—e.g., aligning related structures and using an empty line to separate two sections.
- Make the code self-documenting by choosing descriptive (but relatively short) names for objects, types, functions, etc.
- If you need a nested section, make it a function.
- Make your functions short and focused on a single task. The old *24-line limit* still applies. Although screen size and resolution have changed, nothing has changed in human cognition since the 1960s.
- Functions should have few parameters (four is a good upper bound). This does not restrict the data communicated to functions: grouping related parameters into a single object localizes *object invariants*, which simplifies reasoning with respect to their coherence and consistency.
- More generally, each unit of code, from a block to a library, should have a *narrow interface*. Less communication reduces the reasoning required. This means that *getters* that return internal state are a liability—don't ask an object for information to work with. Instead, ask the object to do the work with the information it already has. In other words, *encapsulation is all—and only—about narrow interfaces*.
- In order to preserve class *invariants*, usage of *setters* should be discouraged. Setters tend to allow invariants that govern an object's state to be broken.

As well as reasoning about its correctness, arguing about your code helps you better understand it. Communicate the insights you gain for everyone's benefit.

A Comment on Comments

Cal Evans

IN MY FIRST PROGRAMMING CLASS IN COLLEGE, my teacher handed out two BASIC coding sheets. On the board, the assignment read, "Write a program to input and average 10 bowling scores." Then the teacher left the room. How hard could this be? I don't remember my final solution, but I'm sure it had a `FOR/NEXT` loop in it and couldn't have been more than 15 lines long in total. Coding sheets—for you kids reading this, yes, we used to write code out longhand before actually entering it into a computer—allowed for around 70 lines of code each. I was very confused as to why the teacher would have given us two sheets. Since my handwriting has always been atrocious, I used the second one to recopy my code very neatly, hoping to get a couple of extra points for style.

Much to my surprise, when I received the assignment back at the start of the next class, I received a barely passing grade. (It was to be an omen to me for the rest of my time in college.) Scrawled across the top of my neatly copied code was "No comments?"

It was not enough that the teacher and I both knew what the program was supposed to do. Part of the point of the assignment was to teach me that my code should explain itself to the next programmer coming behind me. It's a lesson I've not forgotten.

Comments are not evil. They are as necessary to programming as basic branching or looping constructs. Most modern languages have a tool akin to javadoc that will parse properly formatted comments to automatically build an API document. This is a very good start, but not nearly enough. Inside your code should be explanations about what the code is supposed to be doing. Coding by the old adage, "If it was hard to write, it should be hard to read," does a disservice to your client, your employer, your colleagues, and your future self.

On the other hand, you can go too far in your commenting. Make sure that your comments clarify your code but do not obscure it. Sprinkle your code with relevant comments explaining what the code is supposed to accomplish. Your header comments should give any programmer enough information to use your code without having to read it, while your inline comments should assist the next developer in fixing or extending it.

At one job, I disagreed with a design decision made by those above me. Feeling rather snarky, as young programmers often do, I pasted the text of the email instructing me to use their design into the header comment block of the file. It turned out that managers at this particular shop actually reviewed the code when it was committed. It was my first introduction to the term *career-limiting move*.