Reading and Modifying Code

John Aycock

ISBN 978-0-9809555-0-7

For Cliff

Contents

Preface

If you already know how to read and modify code, this book is not for you. Go buy a good novel instead.

This book is intended for people who already know how to program, primarily at the university level. Code reading and modification is not a skill which is always taught, even in higher-level computer science courses. There are few good resources on this topic. In any case, pointing students to some mighty tome is often counterproductive. This book is meant to fill the gap, by providing a language-independent, low-cost, easy-to-carry guide, which can be used as a supplementary course text for programming courses.

Thanks to Darcy Grant, Nigel Horspool, Shannon Jaeger, Cliff Marcellus, Joe Newcomer, Craig Schock, and Jim Uhl for reading and commenting on various drafts. Rob Walker was the friendly neighborhood authority on aspect-oriented programming, and Margaret Nielsen pointed me to some interesting references.

I hope you find the advice in here useful.

1 ⬩ Introduction

To become a good writer, you practice writing. A lot. You also read the works of great writers. And study them – how is the plot developed? what words are selected and why? You also read a lot of work that isn't so great, and figure out why, so you don't make the same mistakes. You edit works in progress to improve their presentation.

Becoming a good programmer requires the same process. You must practice programming. You need to read and study the code of great programmers, as well as not-so-great programmers. You must determine how to modify and improve code.

Code is read many more times than it is written, so it makes sense to look at ways to create readable code. Maintenance programming is also a mainstay of programming, for better or worse.

This book is a guide to reading code, modifying code, testing and debugging modified code, and writing readable code. It does not include much code, on purpose. The ideas and advice in here are largely independent of constantly-changing programming languages and tools. For this reason,

generic terms are used where possible:

- "Subroutine" is used to mean a function, procedure, or method.
- "Module" refers to some discrete program unit, like a module, class, interface, or a file.
- "Name" means any identifier in a program. This may include names of variables, subroutines, modules, or constants.
- "Input" is used in a general sense to include all sources of input, like files, keyboards and network connections, as well as event sequences in a windowing environment.
- "Editor" includes both text editors and editing facilities in integrated development environments.

Only technical issues facing individual programmers are considered. Situations like programming in groups also involve communication and social issues which are outside the scope of this book.

You won't understand everything in this book the first time through. This is intentional. As you grow as a programmer, this book will grow with you, and increasingly more of the advice will make sense. Just like code, it is meant to be read and re-read.

2 ⬧ Reading Code

Code is a specialized form of communication from human to computer, but also from human to human. Just like other types of specialized communication – legal documents, recipes, patent applications – code takes practice and experience to properly interpret.

2.1 Have a Purpose

When you read a book or magazine, you have a specific goal in mind. This may include entertainment, education, reference, or simply killing time. Your goal determines what details you focus on and retain while reading.

You should have a goal in mind when reading code too, for the same reason. You may, for instance, be interested in the flow of control in the program, or you may be acutely interested in the details of one particular subroutine.

Some common reasons for reading code are:

Testing. When testing, you're interested in locating potential problem areas you need to test.

This is discussed further in Chapter 4.

Debugging. Reading code to track down a bug. As a programmer, you have a "mental model" of the code in your head, modeling what you think the code should be doing. A bug may indicate that your model is incorrect, and you need to discover where the code diverges from your model so that you can correct the code. Another possibility is that both the code and your mental model are correct by themselves, but there are complicating external factors to consider, like concurrency. When debugging, you need to read the code exactly as the computer would read it, which requires meticulous attention to detail. Debugging is the subject of Chapter 5.

Code review. Code review might imply some amount of software engineering, such as reading code to verify that a formal software specification is met. Less rigorously, a code review may just involve your code being read by another programmer as a secondary check against bugs.

Security auditing. Security auditing is a very specialized form of code reading. Roughly speaking, a code review verifies that code is doing what it's supposed to. A security audit goes beyond that to verify that code isn't doing anything it's not supposed to, *and* that code can't be coerced by an attacker into doing anything it's not supposed to. This requires specialized skills and is beyond the scope of this book.

Reverse engineering. Again requiring specialized skills, reverse engineering takes an existing piece of executable code and works backwards to reconstruct how it works. Reverse engineering typically relies upon tools, like disassemblers and decompilers. It is somewhat of a legal quagmire, because some software licenses strictly prohibit reverse engineering, yet there are often compelling reasons to do so.[1]

Design comprehension. Understanding code design means reading the code with a high-level perspective; you want to discover how all the different pieces of the code fit together and call one another. Design comprehension is often a prelude to other types of code reading. It can also be used for "design recovery," when dealing with old, legacy code whose original design has been lost or altered beyond recognition.

Documentation. Code may need to be read while writing documentation in order to verify details of its operation. Internal documentation, like comments, tends to be closely linked to the code; external documentation, on the other hand, may require reading the code for behavioral rather than implementation details. For example, an external document describing an API would probably omit implementation-specific information.

Maintenance. Reading for code maintenance purposes is done with a specific question in mind: where do I need to change the code so that it does *X*? Maintenance may involve debugging too: where do I need to change the code so that

it stops doing *X*? You need to read the code to find the target location, as well as to understand the target location's context and connection with the rest of the code.

Some types of reading are naturally more nebulous than others. The difference depends on whether you're looking for the known (e.g., a reproducible bug) or the unknown (e.g., any potential bug).

2.2 Understanding the Design

Even if you're not reading for design comprehension purposes, a basic understanding of the code's design will be of tremendous use.[2] Generally, you will be trying to identify three things:

Modules. You need to find the largest basic "chunks" or building blocks in the code. This is an initial level of abstraction when reading code.

Dependencies. Once you've found the modules, you must determine how they fit together. In other words, how do modules use and interact with one another? There are actually two types of dependency: inter-module dependencies are between modules; intra-module dependencies are within a single module.

Key data structures. Discovering the type and role of important data structures can allow the code manipulating them to be abstracted away.[3] For example, finding a table that encodes all the commands a program understands probably

means that you don't need to thoroughly read the code that interprets that table.

Modules and their dependencies may be looked for in a "directional" fashion: top-down, following the way the code would be executed; bottom-up, reading the code linearly and trying to piece it together; middle-out, using a combination of top-down and bottom-up reading.

In object-oriented code, you may also be looking for:

Design patterns. A design pattern is just that: a code design which can be applied in a specific situation that matches the pattern.[4] Recognizing such patterns in the code can quickly give you a high-level view of the code's design. In theory, design patterns aren't limited to object-oriented code, but they have found their widest usage there to date.

Class relationships. How are classes in the code related to one another? For example, they may be arranged in a hierarchy, and extend and be extended by other classes in various ways. Understanding class relationships is critical to understanding an object-oriented design.

Less frequently, you may read code whose actual design cannot be expressed well using the implementation language.[5] The code author may have made Herculean efforts to implement the design, and a deep understanding of the code can require abstracting away the excess implementation details.

You may find it helpful to construct hypotheses about the code design as you read through

the code. Understanding the design then becomes a matter of looking for evidence that supports or refutes your hypotheses.[6] For example, for a command-driven program, you might hypothesize that each command is handled in a separate piece of code; further, you might also hypothesize that there is a dispatch mechanism to direct each command to the appropriate handler code. To test these hypotheses, you might look for the presence of many small command-handling subroutines, and find out where they are invoked from.

2.3 Tools

The difficulty of reading code increases with the size of the program. A hundred lines of code usually presents no special challenge, but large bodies of code, millions of lines long, are not unusual. This can be overwhelming at first,[7] but there is one key observation:

> You don't need to understand all the details of code that is designed and written in a rational, logical way.

Given this, the problem of reading code becomes a matter of discovering what you do need to pay attention to. Tools play an important role in this discovery process.

Improving Readability

Some code is formatted poorly – by any standard – and is hard to read in its original form. You are

free to improve the readability of the code when reading your own private copy. Ideally, you will want to do this quickly, with little or no effort on your part. Tools called “code formatters” or “pretty printers” take code as input and reformat it without changing its operation, by adding and deleting whitespace. Good formatters are highly configurable, and will permit you to tailor the code style to one which you will find easy to read.

Editors are also tools that can assist with the readability of code. “Syntax-highlighting” editors will automatically highlight parts of the source code, like comments and reserved words, using color, brightness, and font changes. For code with complicated nestings of braces, brackets, and parentheses, editors can show how pairs of these symbols match up.

Searching

Unlike normal prose, code is a very nonlinear form of communication to read; it jumps around from place to place. Fortunately, it usually does so in a fairly controlled, logical fashion because of how people tend to write code.

For this reason, search tools are the bread and butter of the code reader. Most editors have some search capability, but often you will want to search for a word in the entire code body, not just the files that happen to be currently open in the editor. Some useful tools are:

Multi-file search tools. Tools to search through a set of files for a specified term usually come standard with an operating system,[8] because

they are generally useful even to users who don't read code.

The primary distinguishing characteristic of these search tools is how sophisticated a term they will look for. Some will be limited to fixed terms; others will support simple wildcards; still others will look for patterns specified using regular expressions. For comparison:

`foo`	Fixed term, finds `foo` only
`f?o`	Simple wildcard, finds three-letter sequences starting with `f` and ending with `o`
`^(foo|bar)`	Regular expression, finds `foo` or `bar` when they appear at the start of a line

A multi-file search tool that is able to search files buried in subdirectories (a "recursive" directory traversal) is handy for code spread across multiple directories.

Tags. A common task when reading code is to go from the use of a name (like a subroutine) to the name's definition. Support for this task is given by "tags" utilities – a tool is run over a body of code which gathers up all definitions into a database. Tag-savvy editors are able to search this database, given a name used in the code, and instantly jump to the appropriate definition.

Design visualization tools. At the heavyweight end of the tool scale are design visualization tools.[9] These tools analyze the code automatically, and may be used to display dependencies within the code, name definitions and cor-

responding uses, and other potentially useful information.

Finding the correct term to search for in code is often a mixture of educated guesses, intuition, and luck. Say, for example, you want to find the joystick initialization code in an application. You might try the following sequence:

1. Search all files for `init`, being the common part of "init," "initialize," and "initialization." A case-insensitive search will also find instances with different capitalizations, like "doInit."

2. Filter out extraneous results, if necessary. A good way to do this is by searching the search results themselves, but negating the result – most search tools permit this. In other words, search the results for everything *except* some term.

3. Expand the search to include logical synonyms. In this case, you might also try "start" and "main."

4. Start looking through the code for clues. Initialization code is usually called early on, so you can start reading the code from the place where it would normally start executing. The idea is to look for likely search terms that you may have omitted – a call to a "setup" subroutine, for instance, might be the vital clue.

An alternative sequence:

1. Try to first narrow down the search to the joystick-related code, by searching for "joystick" in the code body or by simply looking for files with "joystick" or some related term in the filename.

2. Look at the volume of code you've discovered. For relatively small amounts of code, it can be faster to page through the code manually, skimming it for subroutines of interest. Otherwise, this (smaller) set of files can be searched using the usual tools.

2.4 Vital Information

Obviously, when reading code, the code itself is an excellent source of information. There is other information to draw upon, however – some is ignored by the computer, some is written in a shorthand way, and some isn't there at all.

Comments

Comments (and more generally, external documentation) appeal to humans reading code because the computer does not look at them.[10] Comments are an aside directed solely to humans.

Unfortunately, this is also the downfall of comments. There is nothing to ensure that the comments are correct and that they are in synch with the code.[11] Where comments are present, there

are four cases with respect to the correctness of code:

	code incorrect	code correct
comments incorrect	✗	✗
comments correct	✗	✓

You are, needless to say, only interested in the case where both comments and code are correct. The tricky part is deciding when that is. You should use comments as a guide to your reading, giving them the benefit of the doubt for efficiency's sake, but always remember that the comments may be misleading.

Idioms

Programming languages have idioms just as human languages do. Recognizing an idiom when reading code can give immediate understanding about a piece of code and what it's doing. Idioms are learned through the process of reading and writing code, and so require a certain amount of expertise in a given language. Fortunately, unlike human languages, the rigid nature of programming language semantics permits the meaning of a code idiom to be deciphered, even if the idiom itself is not recognized[12] – idioms are thus a code reading shortcut for experts.[13]

For example, some languages idiomatically iterate over arrays of size *N* from element *0* to element *N-1*. Recognizing this idiom immediately

conveys the higher-level understanding "the code is iterating through the entire array." Conversely, a red flag is raised when the array is iterated through using non-idiomatic bounds, say from *1* to *N-1*, indicating that something special is happening.

The Invisible

Some languages have magical side effects that happen when executing. These side effects are implicit, and not apparent from reading the code, so the only way to know about them is by being familiar with the language.[14] For example, variables starting with `I` through `N` in Fortran are normally integers;[15] in Perl, the statement `s/foo/bar/` uses and sets the variable `$_`;[16] C++ is notorious for quietly inserting default code into a program which may or may not behave like the code author intended. It's sometimes helpful to write short test programs to see the effect of magical statements.

Also, certain programming language features result in programs which are hard to follow not because of implicit side effects, but because of subtleties that make it hard to determine what name is being referenced. For example:

Dynamic scoping. Most languages have "static scoping," which means that it's always possible, given a name in the code, to decide what that name refers to just by looking at the code. With dynamic scoping, what a name refers to may change depending on how the program executes. In other words, determining what a name refers to in a dynamically-scoped language is undecidable.

Dynamic typing. In dynamically-typed programming languages, the type of a name depends on the type of what was last assigned to it as the program executes. As with dynamic scoping, it's not always possible to determine the exact type of a name.

Overloading. Some languages support overloading of subroutines or operators. This means that the exact code used in any given context may be dependent on the types of variables involved, and the number of arguments. For example, if the + operator is overloaded, the expression `a + b` may add `a` and `b` together, or it may post your credit card information to the Internet. When reading code in the presence of overloading, you must work out exactly what code will be executed.

Inheritance. Object-oriented programming languages allow classes to inherit variables, constants, and subroutines from one another. Like overloading, reading code with inheritance means that it can be difficult to determine what code will be executed.

For both inheritance and overloading, code is often spread across multiple files, compounding the problem. Good tools, like class browsers, can greatly assist with determining the structure of such code.

Aspects. Aspect-oriented programming allows an existing body of code to be extended without directly modifying the original code. A programmer defines "aspects," which are snippets of code that are automatically executed when

the original code does certain specified things, like return from a call to subroutine `foo`, or when subroutines `foo` and `bar` are called in succession. To properly read aspect-oriented code, you need to be aware of both the original code as well as all the aspects.[17]

This should not be construed as a general condemnation of these features, as each has advantages for solving certain types of programming problem. The tradeoff, however, is readability.

2.5 Complications

Code reading can be complicated because of peculiarities of the the code design and implementation, and also because of what happens when the code executes.

Spaghetti

A base assumption to make when reading code is that the code has been designed and written in a rational, logical way. Code can be extremely hard to read if this assumption turns out to be false.

There are, unfortunately, some special cases where this occurs:

Machine-generated code. Some code is automatically generated by tools rather than being written by humans.[18] Usually such tools operate from a high-level specification that *was* written by humans; here, it is preferable to read the specification instead of the generated code.

Obfuscation. A code author may wish to release code (or an executable) in a form that is resistant to reverse engineering. This is particularly the case for scripting languages where the source code is executed directly, but there are other languages (like Java) which are especially susceptible to decompilation. Code may be transformed, or obfuscated, in such a way that it makes reverse engineering difficult – for example, changing all variables to look like `X00123`.[19] Obfuscation is usually done automatically with tools.

Spaghetti code. Human-written code that jumps around from place to place in a seemingly arbitrary manner is referred to as "spaghetti code." While this might be done intentionally to try and obfuscate the code, it is also considered the hallmark of a bad programmer.

In the latter two cases, stubborn persistence is needed to read the code. Taking notes may even be necessary to keep track of what the code is doing. It is worth taking extra time in advance, if necessary, to home in on the spot to read.

Concurrency

Concurrent programs can be challenging to read and write because of the interaction between different threads of execution. A good strategy when reading concurrent code is to identify resources shared between the threads, such as files, variables, and data structures. Code that manipulates these shared resources will require special atten-

tion to fully understand what the code is doing in relation to other threads. Apart from those trouble spots, it's safe to begin with the assumption that the code you're reading operates independently of all other code, the assumption you would usually make when reading code.

Interrupts

Code using interrupts – especially asynchronous interrupts, which can happen at unpredictable times – can also be difficult to read. Code in an interrupt handler can cause the program state to suddenly change in ways which are not obvious from reading the rest of the code. It's a good idea to identify interrupt handlers when reading code, to determine what they do and when they are triggered.

2.6 Practice

Good code reading skills are developed only through practice. A good way to start is by reading code for design comprehension. Fortunately, there is lots of source code readily available via the Internet; you can pick some application of interest to you and begin reading.

Different types of application and different programming paradigms will read differently. Graphical user interface code will be different from operating system code; functional programs will be different from imperative ones. A good code reader will be experienced in them all to some degree.

3 ⬩ Modifying Code

Good code modification is a disciplined, scientific process, which can be approached in a step-by-step manner. The basic assumption is that the code has been designed and written in a logical, rational way, in which case it isn't necessary to fully understand the whole body of code in order to make small, localized changes.

3.1 Good Practice

Take Notes

Good code modification is like conducting a scientific experiment. Like scientists, it is advisable to keep notes while making code modifications to keep track of what you've done. Not all the things you did and attempted will be reflected in the code or its backups. For instance, the way you build and install the code will not be there, nor will any modification dead-ends that you backed out of. Careful notetaking also allows you to record the rationale for making certain coding choices;

this may be obvious at the time you're immersed in the code, but obscure later.

Time and interruptions cause details to vanish. A good rule of thumb to start with is to write down anything for which you think "oh, I'll remember that" or "I can figure that out again."

Coding Style

When modifying code, you have informally joined a pre-existing team. Part of being on a team is conforming to certain team standards in preference to individual ones,[20] which in the case of code modification means that you must abide by the project's coding style *even if you don't like it*. A project involving ten different programmers and ten different coding styles is a maintenance nightmare.[21] Also, the chance of your changes being adopted by the original code author diminish considerably if you don't adopt their coding style.

Tools immediately enter into coding style debates. A common argument is that a particular editor doesn't support the code's style by default; the counterargument is that a professional should learn how to operate and configure their tools. Pretty printers can reformat code, and in theory code can be written in any style, then automatically reformatted to the project's coding style. Unfortunately, pretty printers are not always able to perfectly reformat and may make a mess of code in certain circumstances – it is safest not to rely on them.

A related issue is coding consistency. Your modified code should be consistent with the orig-

inal code in terms of the libraries and subroutines it calls to perform specific tasks, and the idioms it uses, for the same reasons that you follow the project's coding style.

Production vs. Test Systems

A production system is a system which is installed, running, and relied upon by people. Never directly modify a production system. Instead, you should set up a private test system which you can modify with impunity without affecting anyone else. The test system should mimic the production system as closely as possible. Eventually, once your changes have been made and tested, they can become part of the production system. The production system, unscathed by any code modifications, can also be thought of as a last-resort code backup for your test system.

3.2 Before the Change

Like baking, sex, and brain surgery, code modification requires some preparation to ensure a successful outcome.

Back up the Code

It's impossible to overstate the importance of backing up the code. Before you make any modifications, save a copy of the original as insurance. It's possible that your code modification may not work as planned, and by the time you realize this,

you may have made a lot of changes to the code or, worse, you may not remember what things you've changed.

The same principle applies when you're making a series of modifications to code, too. Once one part is functioning and stable, saving a snapshot leaves you with a fallback position in case of trouble later on.

As a bonus, by comparing the current code and a saved copy, you can easily determine what has been changed. Tools are available that compare files and directories and output a summary of the changes.[22] This is useful both for remembering where you left off, but also for constructing source code patches.

There are several ways of performing backups. The crudest is to simply make a copy of the code, file by file or *en masse* using an archiving program. A more sophisticated way is to use a revision control system to track changes; this approach has a learning curve, but allows fine-grained version tracking (even in the final executable code) and, depending on the package, can scale to permit multiple programmers to work on the same body of code concurrently.

Build the Code

The next step is to figure out how to build the code, converting it into some executable form. This tends to be a very language- and operating system-specific task, and in extreme cases may require a great deal of arcane system administration knowledge. Ultimately the code must be run,

however, so this step is a necessary evil.

Some common problems at this stage include:

Different tools. Commonly-used tools for building programs include compilers, assemblers, and linkers. It's easy to recognize a tool which is completely absent, but more subtle problems can arise if the tools you have installed are not the same as those used by the code's author. For example, a different version of a compiler may accept a slightly different language or contain different bugs. In extreme cases, building the code may require installing a different version of tools first.

Different environment. Your environment may be different from that of the code's author in other ways besides the tools you have. Often pathnames need to be changed or environment variables need to be set. These are usually quite easy to fix.

A much more difficult problem is where your version of the operating system is different from one the code supports. Obviously, due to the effort involved, it's preferable to avoid changing your operating system, but in some situations it may be the only choice. With luck, the difference can be smoothed over with some minor code changes – essentially, this amounts to porting the code. Often a good compiler is your guide, its error messages pointing you to the differences you need to patch.

Code dependencies. One piece of code may depend upon some other code being built first. Typically, the build instructions for code will

take this into account, but in case of build problems it is worthwhile to keep an eye out for this.

Missing pieces. As well as dependencies within the code, there may be dependencies on external things. Some code relies on third-party libraries and packages which must be installed to complete the build.

Test the built code to make sure that it works. Ideally, the code will come equipped with a test suite which can be run to verify its correct operation. (Practically, such test suites are more the exception than the rule.)

3.3 Making the Change

What constitutes a "change?" When modifying code, you are making a *logical* change, such as adding support for a new feature. Making this logical change may require multiple lines of code in multiple files to be added, changed, or deleted.

The process for modifying code emphasizes being careful and methodical. One change at a time is made, using a scientific approach of forming and testing a hypothesis.

One Change at a Time

Complex pieces of code can interact in complex ways. When you make a change to code, you need to ensure that it has the desired effect, and that any change in the code's behavior is due to the change

you just made in the code. If you make multiple changes to the code, there is always the danger that the changes will interact in some unpredictable and hard-to-debug fashion. Part of taking a scientific approach to code modification is that you must understand exactly the effect of each change.

Check Context

You should always be aware of the context in which your modifications will take place. For example, if you change the output format of a program, and other programs rely on that format, then you can break a lot of code in one fell swoop.[23] No code exists in a vacuum.

X Marks the Spot

Code modification should be a precise operation. Using your code reading skills, carefully pinpoint the areas you must change to get the desired effect. Take your time. A carpenter is supposed to measure twice and cut once; your advantage is that, unlike the carpenter, your changes can almost always be undone. The tradeoff is the amount of time you spend thinking ahead of time versus the amount of time you spend debugging afterwards.[24] For hard-to-understand code, it may be helpful to use a debugger ahead of time to step through the code and unravel its meaning.

When modifying code, you want to be a surgeon with a scalpel, not a monkey with three sizes of hammer.

Form Hypothesis

What do you expect to happen? Before changing any code, mentally form a hypothesis stating what you think will happen when you make your change to the code. Phrase it in terms of some observable, verifiable effect. For example:

> When I add this "print" statement, I will see the size of the list printed to the screen just before the error message box pops up.

Forming a hypothesis gives you a way to test both your understanding of how the code operates, and the efficacy of your code modification. It's important to do this before you make the change, since it's too tempting to fudge it after the fact ("yeah, that's what I thought would happen").

Make and Mark

Now, make the modification to the code. It's good practice to mark the change with a comment which briefly describes who made the change, when it was made, and why it was made. If you use your initials to record "who," then it gives you a mechanism to easily search for changes you made to the code. Also, you can think of marking your modifications as a professional courtesy to the original code author, so that they aren't held responsible for your modifications, and vice versa. (Some code licenses may legally require changes to be marked, too – always read the fine print.)

Test Hypothesis

Once the change is made, the code can be rebuilt. Then, run the code to test your hypothesis. Did you predict the outcome correctly? If you did, you should proceed to test the changed code extensively to ensure that you haven't introduced any bugs. If the code has a test suite, then it's good to add new test cases to it that exercise your code modification.

The other case is where your hypothesis failed. As part of the scientific process, you need to find out why this happened. Remember that you've modified a large piece of code which you may not fully understand – always start by assuming that the error is yours:

1. Examine your modified code for bugs. Does it behave the way your hypothesis said it should?

2. Re-read the code. Verify that you have correctly understood how the code you're modifying interacts with other code. Is it possible that you have chosen the wrong spot to modify?

Once these errors have been ruled out, you can start expanding the search:

3. Look for bugs in the original code. Your modification may be taxing the code in some new way that reveals a previously-hidden bug.

Finally:

4. Re-examine your hypothesis. If everything else checks out, then you may simply have incorrectly predicted the outcome of your modification. It's best to leave this possibility until last, because it's very easy to be lazy and change your hypothesis out of hand, potentially missing some problems.

At the very least, an inspection of this sort will increase your confidence that the change has been made properly.

3.4 After the Change

You're not done yet. Modifying the code may have opened up opportunities to restructure the code and, of course, thorough testing is required.

Restructuring

The final code should appear to be cohesive and well-structured, not a patchwork quilt of various code modifications. Once your modification is successfully made, you should examine the surrounding code to see if there is a better way to express it along with your changes. For example, if the original code looked for a special case, and your modification adds a check for a different special case, there may be a way to generalize both tests and end up with better code. Another example is where a modification duplicates code to the point where a subroutine is called for, a subroutine

which can be called from both the original and the modified code. Code modifications which involve copying code and altering it slightly are prone to needing this type of restructuring.

When looking for opportunities to restructure, pretend that you're writing the code from scratch – is the code's current form the best way to express it?[25]

Regression Test

Testing your modified code looks for bugs in the code you've added. You also need to make sure you haven't introduced any new bugs in the whole code, or re-introduced old bugs that had been fixed. If the code has a test suite, especially one containing examples of old bugs, then you can perform a regression test to verify that you haven't inadvertently broken something. Regression tests should ideally be automated and easy to run.

Some tests that previously succeeded may now erroneously fail as a result of the modifications you made. When a test fails, you need to carefully examine it to determine if it should indeed be failing – a bug – or if the test suite is now in error in light of your modification. In the latter case, you need to update the test suite appropriately.

3.5 Practice

Code modification becomes easier with practice. It is possible, but not very interesting, to contrive exercises that develop this skill: give a menu item

a blue background rather than grey, print “Hello, world!” at a specific point. A good way to practice code modification is to find an application you use (for which you can get the source code) and modify it in one of two ways. First, you can fix some irritating behavior that the program has; this might be something as simple as a bad user interface. Second, you can add some functionality that you want. You may also want to consider sending any generally useful changes back to the original code author for incorporation into the project.

4 ⬩ Testing Modified Code

The techniques for testing modified code are essentially the same as those for testing an entire program. The advantage when testing modified code, however, is that there is a clear focus on the modified code. You want to make sure that your modified code works, and that you haven't accidentally broken anything. Obviously, existing test suites will help ascertain the latter, as mentioned before. The question is: how do you test the modified code?

4.1 Mindset

A good tester is malicious. Users will not necessarily be gentle with a program, and you should stress test code beyond anything a normal user would do. Think evil thoughts, and ask yourself:

> What is the worst possible thing I can do to this code to make it crash?

4.2 Ways to Test

Black Box Testing

"Black box" testing is where the program is treated as a box whose code cannot be examined. Only the program's input can be manipulated, and its output can be checked to see if the program appears to be operating properly. This is of limited value when trying to test a very targeted part of code.

White Box Testing

Another approach to testing is called "white box" testing. Unlike black box testing, you can examine the code to find potential trouble spots to test.

Ideally, you want to achieve 100% code coverage – every single line in the code should be executed by at least one test. This is complicated by the fact that certain code is only run under extreme conditions, like error- and exception-handling code. Some failures can be induced: necessary files and database entries can be deleted; memory allotments can be set artificially low. To reach the ideal code coverage goal will take some creativity and persistence, though.

Code located in hard-to-reach areas may be easier to test in isolation. A separate test harness can be quickly constructed to exercise the modified code thoroughly, before incorporating it into the original code.

Boundary Conditions

A good place to test for problems is boundary conditions. Boundary conditions are places in the code where some kind of conditional test is made: execute this code or that code? run the loop again or not? is the buffer full? There are three possibilities to test:

Within the boundary. This is the "normal" case, where the code is running within acceptable limits.

At the boundary. Testing should be done both at, and close to, the boundary condition. Code can contain "off-by-one" errors which only manifest themselves close to the boundary.

Exceeding the boundary. Finally, look for ways to go beyond the boundary to test. This may not always be possible.

For large boundaries, like big buffer sizes, it may be easiest to temporarily lower the bound for testing. For example, a buffer size of 10 could be used instead of 10,000.

Some boundary conditions are not explicit in the code, but implicit in the semantics of the language, such as fixed-size integers quietly wrapping from their maximum positive value to their minimum negative value upon an increment. Implicit boundary conditions should be tested as well.

Ask for Help

Have other people test your code. Other programmers as well as ordinary users are all valuable in terms of testing, because they bring a fresh perspective which may be wildly different from your own.[26] It is also possible, as a programmer, to become unable or unwilling to see obvious flaws in code, especially where fixing the flaws is hard to do.[27]

4.3 Test-Friendly Coding

Error Conditions

Many system calls and library subroutines return an error status. You cannot properly test your code unless it checks for errors,[28] because otherwise parts of your code may be failing silently. All error return values should be checked and handled appropriately.

When an error is detected, a detailed, unique diagnostic should be produced. Certain programs, especially concurrent programs and programs which interact with others in complicated ways, may only produce an error under unusual conditions which are hard to duplicate. The more information available in these situations, the better.

Determinism

Code being tested with the same inputs, in the same environment, should do the same thing each

time it's run. Unfortunately this is not always possible: concurrency, for instance, may be a necessary part of a program's design. If there are sources of nondeterminism that can be disabled temporarily, writing code to allow for it will simplify both testing and debugging. For example, imagine a program using a pseudo-random number generator, whose initial seed is the current time. The program can have code allowing the generator's initial seed to be specified, yielding the same pseudo-random sequence each time.

The Impossible

Some conditions simply "can't happen" in code. While the impossible is rather trying to test, it is wise to at least guard such cases using assertions. An assertion is a check placed in code that causes it to fail in a controlled fashion should the condition ever arise when the program is executing. Program conditions thought impossible when the code is written are known to arise occasionally as a result of code modifications.[29]

4.4 Tools

Various tools exist to help with code testing.

Code coverage. Some code profiling tools can dynamically determine code coverage when a program executes.

Memory. Languages prone to memory problems can benefit from testing with memory analysis tools. Such tools may watch for allocated

memory areas being exceeded, look for memory leaks, or spot memory which goes unused for suspicious amounts of time.

Noise generators. Often the best test input is not a human-devised one. Noise generators produce long, random program inputs which can be fed to programs to watch their behavior under unusual circumstances.[30] More sophisticated methods being researched also include learning algorithms to automatically develop and learn input sequences that cause program malfunctions.[31]

Debuggers. The primary purpose of debuggers is debugging, obviously. However, their ability to stop an executing program at a specific spot and modify its state can be used to force code into places which are otherwise hard or impossible to reach.

5 ⬩ Debugging Modified Code

Debugging modified code is like testing modified code: the techniques for modified code are much the same as you would use for a whole program.

The base assumption when debugging modified code is that new changes are responsible for new behavior. Your code modifications are likely suspects for any new deviant behavior, using the behavior of the unmodified code as a basis for comparison. If you've made only one change at a time, this further narrows down the culprit. A bug may be deceptive, though – it may not manifest itself directly in the modified code, but may cause other code to break.

5.1 Vital Information

To debug effectively, you need information about the state of the code and the internal state of the executing program.

Know What has Changed

You should ensure that you know exactly what code has been changed, since any of the changes may be contributing to the problem. In some cases, the changed code will be obvious, but in others it may be scattered throughout the body of code. The differences between your modified code and the original code can be found automatically using tools, by comparing the current code against a backup copy.

Internal State Information

It is essential when debugging to have information about the internal state of an executing program. There are several ways to gather this information:

Output. Any visible form of output can be used to relay state information from a program. This includes print statements and log messages, as well as low-bandwidth outputs like LEDs and foreground/background colors – all these can be used to convey information.

The idea is to add debugging code into the program in places where you want to query its state. Debugging code is often "quick and dirty" code added in haste, but care should be taken:

- The program's normal operation must not be changed by adding the debugging code.
- Double-check that the state information being output is in fact the information you *think* is being output.

- Make sure that potential error conditions in the debugging code are handled.

Carelessly-written debugging code can waste lots of time with wild goose chases.

It's good practice to flag debugging code (using specially-marked comments, or by outdenting it) or conditionally compile it in, so that it can be found and removed easily once the bug is fixed.

Debuggers. A good debugger is an invaluable tool. Among other things, it allows program execution to be stopped at specified breakpoints, internal state to be easily queried and modified, and execution to be stepped through with fine granularity. The time invested learning how to use a debugger will be repaid many times over. The only caveat is that a debugger focuses attention on a very small area of code, and it's easy to not see the forest for the trees.

Core dumps. Some systems take a snapshot of a program's memory when it fails in some unrecoverable way; for historical reasons, these are often called core dumps. A good debugger can take a program's core dump and effectively reconstruct the program's state at the point at which it malfunctioned. Using the debugger, you can gather a lot of useful information which often leads right to the bug: where exactly did the program fail? what values did its variables have? what sequence of subroutine calls led to the failure?

Tracing tools. Sometimes tools are available that are able to track a program's interaction with

another part of the system. For example, a tool may print out all the system calls or API calls a program makes as it executes. This doesn't give a fine-grained look inside the program, but may give enough insight to help pinpoint a problem.

5.2 The Debugging Process

Collecting debugging information is only part of debugging. The debugging process involves using debugging information, along with a variety of other techniques, to track down bugs.

Take Notes

Debugging is a methodical, scientific process. As with code modification, it's a good idea to record your work. This helps avoid duplicating work by keeping track of what you've done throughout a complicated debugging session; it also leaves a record which can be referred to later if a similar bug arises ("how did I fix that before?").[32]

Reproduce the Problem

If you can't observe a problem, you can't fix it. The first step when debugging is to reproduce the problem. This may also be the hardest step; some bugs only crop up under unusual circumstances, like high loads or complex interactions with other programs. If you're not able to reproduce the problem, then you're reduced to blindly reading

the code for bugs.

Ideally, you want to not just reproduce the problem, but reproduce it in the simplest, shortest way. Any inputs should be pared down to the bare minimum necessary; this reduces the amount of code to wade through before reaching the suspect parts.

Sometimes, spurious bugs may be reproduced by stress testing, repeatedly testing the suspect area of code until a failure occurs.[33]

The Obvious

Always start debugging by looking for obvious problems. Although it may seem silly, it's possible to waste a great deal of time looking for a complicated answer to a problem when a simple one suffices.[34]

One obvious thing to verify is whether or not you're actually seeing a bug. Sometimes, the code is correct, and the error legitimately lies in the input or an incorrect interpretation of the output. Double-check inputs and outputs, keeping in mind that some things (e.g., whitespace, control characters, nul characters) may not be visible to the naked eye. Tools that overtly "dump," or print, input and output may be helpful; such tools can be quickly constructed if they are not readily available.

Another thing to check is the resources that the program needs. Is any required hardware attached and operational? Is there enough disk space, and are file permissions set correctly? Is the program executing in the correct environment and location?

Hypothesize and Test

Internal state information is used to probe the state of a malfunctioning program. A scientific approach can be taken, just like the one used when modifying code.[35] Make a specific hypothesis about the program's state that can be verified by gathering internal state information. For example, "at line 452, the pointer variable `p` should point to an element of the array `A`." Then, gather information to test your hypothesis. If the hypothesis is wrong, then you are on the trail of the bug, or your understanding of the code is incorrect (but arguably, you're still on the trail of the bug).

Instead of probing a specific point, another approach is to hypothesize how the program's state should be changing as it executes.[36] Here, you would form preconditions and postconditions about parts of the code:

> Before subroutine `foo` is called, `p` must not be `NULL`; after `foo` returns, `p` will be `NULL` and `count` will have incremented by one.

Both these conditions could then be verified with internal state information.[37]

Divide and Conquer

The way that you look up a word in a dictionary or a name in a phone book – a binary search – is a very effective way to track down bugs. The idea is to disable approximately half of the suspect code, usually by commenting it out.[38] Then you

begin an iterative search process: if the bug is still present, disable another half of the code, and keep doing so until the bug vanishes. The last piece of the code to be disabled is likely responsible for the problem, at least in part.

A strict divide-and-conquer approach can reduce code's functionality to the point where it can no longer be executed. This problem can sometimes be ameliorated by replacing the code to be disabled with trivial stubs that fake values, for debugging purposes.

Undo

The logical limit of divide-and-conquer is to disable the modified code completely. Remember that the base assumption was that the original code was working, and that your modifications somehow introduced a bug. If the bug doesn't appear to be the result of the modified code, then this assumption should be challenged. It could be the case that the original code was flawed to begin with, but the flaw hadn't been exposed through testing yet.

Ask for Help

Programmers tend to see what they think the code is doing. This is a natural side effect of abstraction. Unfortunately, debugging requires that you see what the computer is actually doing.

How can you see this? The cause of stubborn bugs may be immediately apparent to another person, or may become apparent in the process of ex-

plaining the problem.[39] Another approach is to simply take a break from the computer, or get a printout of the troublesome code and analyze it instead.

5.3 The Impossible

> "Eliminate all other factors, and the one which remains must be the truth."
>
> – Sherlock Holmes[40]

Very rarely, bugs will have exotic causes. There are some things which you normally assume to be correct when debugging: the operating system, system libraries, output from the compiler, the hardware. It is possible, albeit very unlikely, for these assumptions to be wrong. You should consider this possibility only as a last resort, after all normal causes have been ruled out; even then, such a claim ("my code doesn't work because the compiler is broken") should be backed up with convincing evidence. The debugging task then becomes a search for a way to work around or fix this new problem.

There are also bugs – called "Heisenbugs" – that disappear when you look for them.[41] The mere act of adding output statements or running the code in a debugger changes the program just enough to make the problem go away. This does not, however, mean that the bug has been fixed. Until the cause of a bug has been determined, debugging should continue.

6 ⬩ Writing Readable Code

The coding style of an existing body of code should be adhered to when making changes. But suppose you're writing brand-new code. How can you write it so that it's readable?

6.1 Remember Your Audience

A standard piece of advice for any communication – verbal, written, or otherwise – is to remember your audience. The same is doubly true for computer code. With code, you not only have to express yourself precisely to the computer, but you also must leave something understandable for humans.

There are almost always multiple ways to write a piece of code. Making your code readable for your human audience should help guide your coding choices. How much should you document? What should you document? How densely should the code be written? What obscure language idioms can you use?

Sometimes it's useful to use yourself as a reference point. Ask yourself, will I understand this code in a year's time? You are your own audience, too.

6.2 Design

Code design is something which is best taught by experience.[42] Reading and modifying someone else's code is instructive, although the exact lesson depends on whether the code has a good or a bad design. Similarly, implementing and using your own code design is valuable, especially when you make mistakes – there is nothing to cement a design lesson like working with a flawed design of your own making.

There are some standard approaches to good design which are worth considering:

Isolating dependent code. Ideally, any code that is dependent upon something else should be separated out. Code can be dependent on many things: target architecture, operating system, windowing system, specific libraries. Identifying and isolating this dependent code helps abstract your design away from minute details, and makes your code more portable.

Directional design. There are three "directional" design methods. A top-down approach starts from a very high level and progressively breaks the programming task down into smaller and smaller pieces. Bottom-up design starts with the low-level building blocks of a program

which actually do the work, piecing them together until the program is complete. Finally, a middle-out design strikes a balance between the two approaches, building and breaking down.

The design method may vary with the programming task. Creating a good set of building blocks for a bottom-up design comes through experience. Top-down designs are useful for prototyping, where you may not yet know how to construct the building blocks; it is also useful for undesirable programming tasks, because it allows the "real" work to be deferred as long as possible.

Coupling and cohesion. The parts of a good design – call them modules – should exhibit a high degree of cohesion and a low degree of coupling. High cohesion means that a module does one specific task, like implementing a data structure, and everything in that module is used toward that end. Low coupling means that a module is not intimately connected with the inner workings of another module.

Design patterns. Object-oriented designs have a wealth of "design patterns" to draw upon. Effectively, this creates a shorthand vocabulary for describing certain designs. The drawback is that a person reading the code must understand this same vocabulary for the shorthand communication to be useful. At the very least, design pattern bestiaries can act as a helpful source of design inspiration.

Ultimately, good code design is a black art. As a heuristic, try and imagine if your design will

make the code easy to read and modify using the approach of the last few chapters – in other words, is your design rational and logical?

6.3 Code

Name Your Poison

The programming language you write your code in will undoubtedly bring coding style constraints with it. Some constraints are more subtle than others.

Write-only languages. Some languages are referred to as "write-only" languages, because code is fully understood only once, when it is written, and it is next to impossible to read afterwards. Perl is the current frontrunner in this category; past winners have included PostScript, Forth, and APL.

This is a somewhat unfair designation for a programming language, for three reasons.

1. Code written in such a language is quite meaningful to an expert who is regularly immersed in the intricacies of the language. Such experts are not the norm, however.
2. Some languages require different ways of thinking about programs. Going from one language paradigm to another, for example, is not necessarily an easy task.
3. It's possible to write bad code in any language. There are even contests to write

bad and/or obfuscated code; here is one prize-winner for a C obfuscation contest:[43]

```
#include <ctype.h>
#include <stdio.h>
#define _ define
#_ A putchar
#_ B return
#_ C index
char*r,c[300001],*d=">=<=!===||&&->++-->><<",*i,*l,*j,
*m,*k,*n,*h,*y;e,u=1,v,w,f=1,p,s,x;main(a,b)char**b;{p
=a>1?atoi(b[1]):79;r=c+read(0,j=l=i=c,300000);v=g(j,&m
);for(k=m;v!=2;j=k,m=n,v=w,k=m){w=g(k,&n);if(v==1&&m-j
==1&&*j==35)e&&A(10),e=f=0;if(!f&&v==3&&(char*)C(j,10)
<m)A(10),e=0,f=1;else if(v>2&&(u||w)&&(f||u)&&(l-i>1||
*i!=61||n-k>1|||C("-*&",*k)))continue;else if(v==3)if(
f&&e+1+n-k>p&&e)A(10),e=0;else A(32),e++;else{if(f&&e+
m-j>p&&e)A(10),e=0;e+=m-j;k=j;while(k<m)A(*k++);}i=j;l
=m;u=v;}e&&A(10);}g(j,m)char*j,**m;{if(j>=r)B*m=j,2;s=
isdigit(*j)||*j==46&&isdigit(j[1]);for(h=j;h<r;h++)if(
!isalnum(*h)&&*h!=95&&(!s||*h!=46)&&(!s||h[-1]!=10l&&h
[-1]!=69||!C("+-",*h)))break;if(h>j)B*m=h,0;x=1;for(h=
j;h<r&&C(" \t\n",*h);h++);if(h>j)h--,x=3;if(*j==34||*j
==39)for(h=j+1;h<r&&*h!=*j;h++)if(*h==92)h++;for(y=d;*
y&&strncmp(y,j,2);y+=2);if(*y)h=j+1;if(!strncmp("/*",j
,2)){h=j+2;while(*++h!=42||*++h!=47);x=4;}*m=h+1;B x;}
```

Hard style guidelines. Although more the exception than the rule, some programming languages enforce certain style guidelines. Python, for instance, groups statements together using indentation.

There may be other constraints which are not fixed, but may be difficult or time-consuming to work around. Some programming tools, like editors, may support a specific code layout by default which may not be ideal. However, especially when working with a group of people, there may be a tradeoff involved between the perfect layout and having everyone reconfigure their tools.

Soft style guidelines. Established languages are likely to have established coding style guidelines. (More likely, they will have several competing guidelines!) If you devise a "better" cod-

ing style, you run the risk of rendering your code unreadable by others, simply by virtue of being different. Another tradeoff to consider.

Idioms. Experienced code readers will be expecting language-specific idioms to be present and used appropriately in the code. Using code idioms can impart a lot of information very quickly.

Spacing and Indentation

Youwillprobablyfindthissentencehardtoread.

Spacing plays the same role in code as it does in prose. Or imagine your favorite music, played without any rests. In music, when you don't play is as important as when you do play, and the same concept is true for readable code.

There is no advantage to writing reams of code with insufficient space.[44] Your code doesn't run any faster, and you don't save any substantial amount of disk space. As a concrete example, for many languages you can indent code with tabs, where a tab is eight spaces, and use spaces liberally elsewhere. Visually, your code should look like it has "elbow room" – it shouldn't look cramped.

Having said this, the need for too many levels of indentation may indicate a design flaw. The code may need restructuring with subroutines, or perhaps there are an excessive number of special cases that can be generalized.

Line Length

Line length is obviously tied in with code spacing and indentation. It may seem like a holdover from the dark ages of computing, from punched cards and character-only video displays, and to a certain extent it is. However, there are some good reasons to strictly adhere to a certain fixed line length – typically, 80 columns is advisable.

First, a relatively short line length improves readability. Newspapers, for instance, still use narrow columns to allow a good reader to simply read down the column with no wasted eye movement.[45] The same principle applies to computer code. A long line, or worse, a long line wrapped around the screen or a printout, means extra work for a reader to put all the pieces together.

Second, when combined with good spacing and indentation, a fixed line length is a good heuristic measure of code complexity. If you can't express a line of code in 80 columns using tabs for indentation, then it's a strong indicator that you should examine what you're doing. A subroutine may be needed, or it may suggest that the code needs restructuring or a completely different approach. If the code is hard to write, it will likely be hard to read too.

Cut and Paste

"Cut and paste" coding is the derogatory term used to describe copying code from one place to another in a body of code, possibly making a small number of changes to the copied code. This sends

a strong signal that code restructuring opportunities are present. It also makes code less maintainable, because bugs are also copied – fixing a bug fully means tracking down all similar copies of the buggy code.

From the readability point of view, copying code burdens the code reader by forcing them to read the same code again and again. It also makes a reader laboriously figure out what differences, if any, exist between various copies of the code.

Laziness

Give a reader less code to read: sometimes the best code is no code at all. A keen sense of laziness, the desire to avoid writing lots of code, is key to identifying and exploiting self-similar structure. Factor out the dissimilar parts of otherwise similar code into a table, and have one piece of code do the job of many pieces by simply indexing into the table. In more complicated cases, the table may require a little code "engine" to interpret it properly, but even the combination of code plus table can be much shorter than the naïve, brute-force code.

Or, give a reader simpler code. Be lazy and solve a problem in multiple simple steps rather than one complex step. There are also times when it's easier to write code to do something poorly and then write code to fix up the result. For example, writing a compiler that produces good output in one step would be next to impossible. The code would be far too complicated. It's far easier for a compiler to generate bad but correct output,

then apply multiple simple transformations to fix up the bad output.

6.4 Documentation

Part of documentation is in the code itself. Using meaningful variable names, constant names, and subroutine names are all important cues to someone reading code.[46] The use of "magic values," numbers and other literal values that are used in code whose meaning is not immediately apparent, should be avoided.

Beyond the code, you can have external documentation, like user manuals or manual pages, or code comments. There is always the danger of the code, comments, and external documentation getting out of synch, and there are a variety of ways to manage this:

Ignore the problem. Maintain the code, comments, and external documentation separately.

Embedded documentation. Some systems permit external documentation to be embedded in the code, marked using specially-denoted comments. This documentation is then automatically extracted to create the external documentation. Currently, the Javadoc system for Java is the prime example of this technique. The theory is that, by merging code and documentation in this way, programmers will find it easier to write and update documentation.

Embedded code. Another approach is called "literate programming."[47] A literate program has

the code embedded in the documentation; here, the code is extracted automatically from the documentation.

What should be documented? Again, remember your audience. It is safe to assume a certain base level of programming knowledge. Thus, comments like:

```
x = x + 1;     /* add one to x */
```

supply as much useful information as:

```
x = x + 1;     /* x is the 24th letter
                  of the alphabet */
```

Comments of this sort should always be avoided. Instead, describe your code from a high-level point of view – the details are in the code if needed. Having said that, be sure to document any tricky or non-obvious details too. The interface to your code should be documented as well. When in doubt, err on the side of documentation quality rather than quantity.[48]

You should always give credit where it is due. If your code is based on (or blatantly stolen from) some other code, document the source. Failure to do this in the academic world would be plagiarism; in industry, it would be grounds for intellectual property lawsuits. Some code, while freely-available, has licensing restrictions which requires users to note its usage in any documentation – always check the fine print.

Problem Areas

It's important to document what your code does, but it's also important to document what it doesn't do. Depending on the sort of documentation you are producing, this information can go in either the user documentation or in code comments.

Bugs. It's unlikely that you'll know what all the bugs are in your code, but it is likely that you may know about several when writing the code. Even if you don't fix the bugs, you can at least leave warnings about them.

Limitations. Limitations are not bugs *per se*, and do not cause incorrect execution, but impose constraints of some form. A typical example of a limitation would be the use of a fixed-size input buffer as opposed to a dynamically-sized one.

Tunable parameters. Situations where arbitrary values are used in code should be noted. These values, while correct, may present later opportunities for tuning and optimization.

Better algorithms. Better choices for algorithms may come to mind when writing code, like the possibility of using a binary search instead of a linear search, but you may not have the chance to implement them. It's always a good idea to add a note about what algorithm should be used – at the very least, it tells people reading your code that you did know what you were doing!

When writing about such problem areas in comments, it's good practice to mark them so that

they may be easily searched for later. The strings "XXX" and "TODO" are often used for this purpose:

```
/* XXX - find an algorithm to see
         if this code terminates */
```

6.5 Practice

Coding and design skills improve with practice. It's wise to start small, with coding problems you can finish in one sitting. Programming language textbooks often have short exercises in them which are suitable, or use problems from programming competitions. For larger projects, choose something you're interested in, or a program you need that doesn't exist. If you don't want to start coding from scratch, there are a seemingly infinite number of open-source projects which are both available and in dire need of major coding contributions.

7 ⬩ Summary

The three most important things in real estate are location, location, and location. In code reading, writing, and modifying, the three most important things are practice, practice, and practice. The advice in this book doesn't magically help, unfortunately; it's only a starting point for developing your skills.

Happy reading!

Notes

[1] For example, anti-virus researchers may need to partially reconstruct legitimate code when determining how malicious code operates.

[2] Understanding the design when reading code has some overlap with the design of new code (Section 6.2). What separates the two is that, when reading code, you're looking at the end product, not the means used to get there.

[3] There is a famous quote by Brooks [1995, page 102]:

> Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

[4] The standard design pattern reference is Gamma et al. [1995], known as the "Gang of Four" or "GoF" book.

[5] The BSD filesystem code, for example, is an object-oriented design trapped in a body of C code. See Vahalia [1996, page 236] and McKusick et al. [1996, page 205].

[6] Brooks [1983] theorizes that programmers understand programs using a top-down approach, making and refining hypotheses. He suggests that evidence for hypotheses is gathered by looking for "beacons" in the code whose presence signals certain data structures or operations. Wiedenbeck [1986] gives some experimental evidence for the existence of beacons.

[7] It can be overwhelming at second, too.

[8] For example, Microsoft Windows includes `FIND`, and Unix systems have the `grep` family of tools. Some visual programming environments have multi-file search tools as well.

[9] For example, see Rigi and SHriMP: Wong [1998] and Wu

and Storey [2000].

[10] Specially-marked compiler directives and JCL notwithstanding.

[11] Archaeologists take note: incorrect comments may indicate the original intent of code which has since evolved.

[12] The same isn't true in human languages. No amount of training in English will help decipher "Bob's your uncle."

[13] Humans naturally group, or "chunk," related information together [Miller, 1956]. McKeithen et al. [1981] and Shneiderman [1976] have verified experimentally that programmers chunk program code, and that experts are better at doing this than novices. Idioms may play a role in the effectiveness of chunking.

[14] *The Story of Mel* is an epic programming tale which brilliantly takes advantage of implicit side effects. It can be found online [Raymond, 2003]. Also, not just languages have invisible side effects. Sometimes the library subroutines called from a language have them too.

[15] Ellis [1982, page 15].

[16] Wall et al. [1996, page 72].

[17] A number of introductory articles on aspect-oriented programming can be found in the October 2001 issue of *Communications of the ACM*. A more critical look is provided by Constantinides et al. [2004].

[18] For example, the Lex and Yacc compiler tools: Levine et al. [1992].

[19] For a more complete list, see Collberg et al. [1997].

[20] As immortalized by the slogan "There is no 'I' in 'team'." There is no 'I' in 'moose' either, but this is probably coincidence.

[21] Mohan and Gold [2004] have done a study of how code style changes over time with maintenance programming, i.e., code modification.

[22] Like `diff` on Unix systems.

[23] A graduate student at the author's university did this in the early 1980s; he changed the output of the "ls" program which caused a backup script to quietly fail; the problem was not discovered for several months afterwards, when a file needed to be restored.

[24] Littman et al. [1986] studies two strategies used by programmers for a code maintenance task: systematic, where the programmer would study the code extensively before making changes; as-needed, where the programmer would take a lazy

approach to studying the code. In their study, only systematic programmers were successful. They point out, however, that the key is constructing a strong mental model of what's happening in the code.

[25] The "best" way to design and implement code often depends on the context. For instance, engineering tradeoffs are commonly made between simplicity and efficiency, or between time and space.

[26] This idea is nicely captured by the aphorism "Each new user of a new system uncovers a new class of bugs." (Kernighan, as quoted in Bentley [1988, page 60].)

[27] Brooks [1995, page 55] and Myers [1976, page 191] make the argument that testing is an inherently destructive process, and the creator of some code isn't really going to want to destroy it, especially when finding flaws in the code may reflect on the skill and ego of the programmer.

[28] Or handles an error in some other way, like catching an exception.

[29] Some programmers use assertions only for testing and debugging, then disable them in the production version of a program. Whether or not this is wise can be debated at length.

[30] For example, a noise generator has been used to give Unix utilities a workout [Miller et al., 1990]. This technique is also referred to as "Monte Carlo" debugging [Bell, 1983], "fuzzing" [Sutton et al., 2007] or, touchingly, Gremlins [Maas, 2003].

[31] Chan et al. [2004] describes such a system being used to test a commercial computer game, for instance.

[32] This record can also be analyzed to gain insight into program design, bugs, and debugging. Knuth [1989], for instance, dissects the log book he kept for ten years' worth of TeX development.

[33] One spurious bug in a program was found by running the program repeatedly with a script. The bug, on average, showed up once every 100 times the program was executed.

[34] The author once found a computer whose monitor wasn't displaying anything. He spent a great deal of time searching for the problem – logging in to the computer remotely to make sure it was working, checking the cables, fiddling with the contrast and brightness knobs, to no avail. The problem was that the monitor had been turned off.

[35] Gould [1975] theorizes that people debug programs by iteratively generating and testing hypotheses until a clue to the

bug is discovered; this approach is used by both novices and experts [Gugerty and Olson, 1986].

[36] Yet another approach would hypothesize how the program's state *shouldn't* be changing, or program invariants.

[37] Pre/postconditions and invariants can be part of code testing and code design, too. The latter is referred to as "design by contract." See Meyer [1997, Chapter 11].

[38] Depending on the language, other methods may be available. A `return` statement may be inserted prematurely to avoid executing certain code, or a preprocessor may be used (in C, with `#if 0`...`#endif`) to quickly block out chunks of code. A common mistake, especially for languages that have matching comment delimiters, is to forget to end a comment and disable much more code than you intended. Syntax-highlighting (a.k.a. colorizing) editors help catch this mistake.

[39] Some labs and help desks have resident stuffed animals, whom you have to explain your problem to first. Apparently a number of problems are solved this way.

[40] Okay, Sir Arthur Conan Doyle *really* wrote it, but it's still a good quote. From *The Sign of the Four*.

[41] Bourne [2004].

[42] Jeffries et al. [1981] examines the role experience plays in design, and how experts and novices differ in their approach to code design.

[43] In case it's not immediately obvious, the code breaks long input lines. This was Paul Heckbert's 1987 winner in the International Obfuscated C Code Contest. Don Dodson's English to Pig Latin translator was an artistic 1995 contest winner:

[44] Unfortunately, as summarized by Oman and Cook [1990], formal studies of indentation effects have produced mixed results.

[45] The optimum line length for readability is one and a half times the length of the lowercase alphabet (Arnold, 1981, pages 33–34; Turnbull and Baird, 1975, page 67). Assuming a monospace font, which is commonly used for code, the optimum line length for code would be 39 characters (not including leading whitespace).

[46] "Meaningful" names do not imply excessively long names, however.

[47] An idea first proposed by Knuth [1984].

[48] Studies indicate that adding comments of any sort, even good comments, decrease the readability of code [Brooks, 1995, page 224].

Bibliography

E. C. Arnold. *Designing the Total Newspaper*. Harper & Row, 1981.

R. C. Bell. Monte Carlo debugging: a brief tutorial. *Communications of the ACM*, 26(2):126–127, February 1983.

J. Bentley. *More Programming Pearls*. Addison-Wesley, 1988.

S. Bourne. A conversation with Bruce Lindsay. *ACM Queue*, 2(8):22–33, 2004.

F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995.

R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan. Evolutionary behavior testing of commercial computer games. In *Proceedings*

of the 2004 Congress on Evolutionary Computation, pages 125–132, 2004.

C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, Department of Computer Science, 1997.

C. Constantinides, T. Skotiniotis, and M. Stoerzer. AOP considered harmful. In *European Interactive Workshop on Aspects in Software*, 2004. Position paper for panel session.

T. M. R. Ellis. *A Structured Approach to FORTRAN 77 Programming*. Addison-Wesley, 1982.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

J. D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7: 151–182, 1975.

L. Gugerty and G. M. Olson. Comprehension differences in debugging by skilled and novice programmers. In Soloway and Iyengar [1986], pages 13–27.

R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood. The processes involved in designing software. In J. R. Anderson, editor, *Cognitive Skills and Their Acquisition*, pages 255–283. Lawrence Erlbaum Associates, 1981.

D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

D. E. Knuth. The errors of TeX. *Software – Practice and Experience*, 19(7):607–685, July 1989.

J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly, second edition, 1992.

D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In Soloway and Iyengar [1986], pages 80–98.

B. Maas. *Using Palm OS Emulator*. PalmSource, 2003.

K. B. McKeithen, J. D. Reitman, H. H. Rueter, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for

processing information. *The Psychological Review*, 63(2):81–97, March 1956.

A. Mohan and N. Gold. Programming style changes in evolving source code. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 236–240, 2004.

G. J. Myers. *Software Reliability: Principles and Practices*. Wiley, 1976.

P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.

E. Raymond, editor. *Jargon File (version 4.4.7)*. 2003. http://www.catb.org/ esr/jargon.

B. Shneiderman. Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences*, 5(2):123–143, 1976.

E. Soloway and S. Iyengar, editors. *Empirical Studies of Programmers*, 1986. Ablex Publishing Corporation.

M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

A. T. Turnbull and R. N. Baird. *The Graphics of Communication*. Holt, Rinehart, and Winston, third edition, 1975.

U. Vahalia. *UNIX Internals: The New Frontier*. Prentice Hall, 1996.

L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly, second edition, 1996.

S. Wiedenbeck. Processes in computer program comprehension. In Soloway and Iyengar [1986], pages 48–53.

K. Wong. *Rigi User's Manual (version 5.4.4)*. University of Victoria, 1998.

J. Wu and M.-A. D. Storey. A multi-perspective software visualization environment. In *CASCON 2000 Proceedings*, pages 41–50, 2000.

Index