How To Write Unmaintainable Code

Last updated Thursday, 18-Nov-1999 20:27:28 PDT by Roedy Green ©1997-1999 Canadian Mind Products.

This essay is also available in Spanish.

In the interests of creating employment opportunities in the Java programming field, I am passing on these tips from the masters on how to write code that is so difficult to maintain, that the people who come after you will take years to make even the simplest changes. Further, if you follow all these rules religiously, you will even guarantee **yourself** a lifetime of employment, since no one but you has a hope in hell of maintaining the code.

General Principles

To foil the maintenance programmer, you have to understand how he thinks. He has your giant program. He has no time to read it all, much less understand it. He wants to rapidly find the place to make his change, make it and get out and have no unexpected side effects from the change.

He views your code through a tube taken from the centre of a roll of toilet paper. He can only see a tiny piece of your program at a time. You want to make sure he can never get the big picture from doing that. You want to make it as hard as possible for him to find the code he is looking for. But even more important, you want to make it as awkward as possible for him to safely **ignore** anything.

Specific Techniques

1. Lie in the comments. You don't have to actively lie, just fail to keep comments as up to date with the code.
2. Pepper the code with comments like `/* add 1 to i */` however, never document wooly stuff like the overall purpose of the package or method.
3. Make sure that every method does a little bit more (or less) than its name suggests. As a simple example, a method named `isValid(x)` should as a side effect convert x to binary and store the result in a database.
4. Use acronyms to keep the code terse. Real men never define acronyms; they understand them genetically.
5. In the interests of efficiency, avoid encapsulation. Callers of a method need all the external clues they can get to remind them how the method works inside.
6. If, for example, you were writing an airline reservation system, make sure there are at least 25 places in the code that need to be modified if you were to add another airline. Never document where they are. People who come after you have no business modifying your code without thoroughly understanding every line of it.
7. In the name of efficiency, use cut/paste/clone/modify. This works much faster than using many small reusable modules.
8. **Never never** put a comment on a variable. Facts about how the variable is used, its bounds, its legal values, its implied/displayed number of decimal points, its units of measure, its display format, its data entry rules (e.g. total fill, must enter), when its value can be trusted etc. should be gleaned from the procedural code. If your boss forces you to write comments, lard method bodies with them, but

never comment a variable, not even a temporary!

9. Try to pack as much as possible into a single line. This saves the overhead of temporary variables, and makes source files shorter by eliminating new line characters and white space. Tip: remove all white space around operators. Good programmers can often hit the 255 character line length limit imposed by some editors. The bonus of long lines is that programmers who cannot read 6 point type must scroll to view them.

10. Cd wrttn wtht vwls s mch trsr. When using abbreviations inside variable or method names, break the boredom with several variants for the same word, and even spell it out longhand once in while. This helps defeat those lazy bums who use text search to understand only some aspect of your program. Consider variant spellings as a variant on the ploy, e.g. mixing International *colour*, with American *color* and dude-speak *kulerz*. If you spell out names in full, there is only one possible way to spell each name. These are too easy for the maintenance programmer to remember. Because there are so many different ways to abbreviate a word, with abbreviations, you can have several different variables that all have the same apparent purpose. As an added bonus, the maintenance programmer might not even notice they are separate variables.

11. Never use an automated source code tidier to keep your code aligned. Lobby to have them banned them from your company on the grounds they create false deltas in PVCS (version control tracking) or that every programmer should have his own indenting style held forever sacrosanct for any module he wrote. Banning them is quite easy, even though they save the millions of keystrokes doing manual alignment and days wasted misinterpreting poorly aligned code. Just insist that everyone use the **same** tidied format, not just for storing in the common repository, but while they are editing. This starts an RWAR and the boss, to keep the peace, will ban automated tidying. Without automated tidying, you are now free to *accidentally* misalign the code to give the optical illusion that bodies of loops and ifs are longer or shorter than they really are, or that else clauses match a different if than they really do. e.g.

```
if (a)
  if (b) x = y;
else x = z;
```

12. Never put in any { } surrounding your if/else blocks unless they are syntactically obligatory. If you have a deeply nested mixture of if/else statements and blocks, especially with misleading indentation, you can trip up even an expert maintenance programmer.

13. Rigidly follow the guidelines about no goto, no early returns, and no labelled breaks especially when you can increase the if/else nesting depth by at least 5 levels.

14. Use very long variable names that differ from each other by only one character, or only in upper/lower case. An ideal variable name pair is *swimmer* and *swimner*. Exploit the failure of most fonts to clearly discriminate between `il1I1|` or `oO08` with identifier pairs like `parselnt` and `parseInt` or `D0Calc` and `DOCalc`. `l` is an exceptionally fine choice for a variable name since it will, to the casual glance, masquerade as the constant 1. Create varible names that differ from each other only in case e.g. HashTable and Hashtable.

15. Wherever scope rules permit, reuse existing unrelated variable names. Similarly, use the same temporary variable for two unrelated purposes (purporting to save stack slots). For a fiendish variant, morph the variable, for example, assign a value to a variable at the top of a very long method, and then somewhere in the middle, change the meaning of the variable in a subtle way, such as converting it from a 0-based coordinate to a 1-based coordinate. Be certain not to document this change in meaning.

16. Use lower case l to indicate long constants. e.g. 10l is more likely to be mistaken for 101 that 10L is.

17. Ignore the conventions in Java for where to use upper case in variable and class names i.e. Classes start with upper case, variables with lower case, constants are all upper case, with internal words capitalised. After all, Sun does (e.g. instanceof vs isInstanceOf, Hashtable). Not to worry, the

compiler won't even issue a warning to give you away. If your boss forces you to use the conventions, when there is any doubt about whether an internal word should be capitalised, avoid capitalising or make a random choice, e.g. use both inputFileName and outputfilename. You can of course drive your team members insane by inventing your **own** insanely complex naming conventions then berate others for not following them. The ultimate technique is to create as many variable names as possible that differ subtlely from each other only in case.

18. Never use `i` for the innermost loop variable. Use anything but. Use `i` liberally for any other purpose especially for non-int variables. Similarly use `n` as a loop index.

19. Never use local variables. Whenever you feel the temptation to use one, make it into an instance or static variable instead to unselfishly share it with all the other methods of the class. This will save you work later when other methods need similar declarations. C++ programmers can go a step further by making all variables global.

20. Never document gotchas in the code. If you suspect there may be a bug in a class, keep it to yourself. If you have ideas about how the code should be reorganised or rewritten, for heaven's sake, do not write them down. Remember the words of Thumper "*If you can't say anything nice, don't say anything at all*". What if the programmer who wrote that code saw your comments? What if the owner of the company saw them? What if a customer did? You could get yourself fired.

21. To break the boredom, use a thesaurus to look up as much alternate vocabulary as possible to refer to the same action, e.g. *display*, *show*, *present*. Vaguely hint there is some subtle difference, where none exists. However, if there are two similar functions that have a crucial difference, always use the same word in describing both functions (e.g. *print* to mean write to a file, and to a print on a laser, and to display on the screen). Under no circumstances, succumb to demands to write a glossary with the special purpose project vocabulary unambiguously defined. Doing so would be unprofessional breach of the structured design principle of *information hiding*.

22. In naming functions, make heavy use of abstract words like *it*, *everything*, *data*, *handle*, *stuff*, *do*, *routine*, *perform* and the digits e.g. `routineX48`, `PerformDataFunction`, `DoIt`, `HandleStuff` and `do_args_method`.

23. In Java, all primitives passed as parameters are effectively read-only because they are passed by value. The callee can modify the parameters, but that has no effect on the caller's variables. In contrast all objects passed are read-write. The reference is passed by value, which means the object itself is effectively passed by reference. The callee can do whatever it wants to the fields in your object. Never document whether a method actually modifies the fields in each of the passed parameters. Name your methods to suggest they only look at the fields when they actually change them.

24. Never document the units of measure of any variable, input, output or parameter. e.g. feet, metres, cartons. This is not so important in bean counting, but it is very important in engineering work. As a corollary, never document the units of measure of any conversion constants, or how the values were derived. It is mild cheating, but very effective, to salt the code with some incorrect units of measure in the comments. If you are feeling particularly malicious, make up your **own** unit of measure; name it after yourself or some obscure person and never define it. If somebody challenges you, tell them you did so that you could use integer rather than floating point arithmetic.

25. In engineering work there are two ways to code. One is to convert all inputs to S.I. (metric) units of measure, then do your calculations then convert back to various civil units of measure for output. The other is to maintain the various mixed measure systems throughout. Always choose the second. It's the American way!

26. I am going to let you in on a little-known coding secret. Exceptions are a pain in the behind. Properly-written code never fails, so exceptions are actually unnecessary. Don't waste time on them. Subclassing exceptions is for incompetents who know their code will fail. You can greatly simplify your program by having only a single try/catch in the entire application (in main) that calls System.exit(). Just stick a perfectly standard set of throws on every method header whether they

could throw any exceptions or not.

27. C compilers transform `myArray[i]` into `*(myArray + i)`, which is equivalent to `*(i + myArray)` which is equivalent to `i[myArray]`. Experts know to put this to good use. Unfortunately, this technique can only be used in native classes.

28. If you have an array with 100 elements in it, hard code the literal 100 in as many places in the program as possible. Never use a static final named constant for the 100, or refer to it as `myArray.length`. To make changing this constant even more difficult, use the literal 50 instead of 100/2, or 99 instead of 100-1. You can futher disguise the 100 by checking for `a == 101` instead of `a > 100` or `a > 99` instead of `a >= 100`.

    Consider things like page sizes, where the lines consisting of x header, y body, and z footer lines, you can apply the obfuscations independently to each of these **and** to their partial or total sums.

    These time-honoured techniques are especially effective in a program with two unrelated arrays that just accidentally happen to both have 100 elements. There are even more fiendish variants. To lull the maintenance programmer into a false sense of security, dutifully create the named constant, but very occasionally "*accidentally*" use the literal 100 value instead of the named constant. Most fiendish of all, in place of the literal 100 or the correct named constant, sporadically use some other unrelated named constant that just accidentally happens to have the value 100, for now. It almost goes without saying that you should avoid any consistent naming scheme that would associate an array name with its size constant.

29. Eschew any form of table-driven logic. It starts out innocently enough, but soon leads to end users proofreading and then *shudder*, even modifying the tables for themselves.

30. Nest as deeply as you can. Good coders can get up to 10 levels of ( ) on a single line and 20 { } in a single method. C++ coders have the additional powerful option of preprocessor nesting totally independent of the nest structure of the underlying code. You earn extra Brownie points whenever the beginning and end of a block appear on separate pages in a printed listing. Wherever possible, convert nested ifs into nested [? :] ternaries.

31. Join a computer book of the month club. Select authors who appear to be too busy writing books to have had any time to actually write any code themselves. Browse the local bookstore for titles with lots of cloud diagrams in them and no coding examples. Skim these books to learn obscure pedantic words you can use to intimidate the whippersnappers that come after you. Your code should impress. If people can't understand your vocabulary, they must assume that you are very intelligent and that your algorithms are very deep. Avoid any sort of homely analogies in your algorithm explanations.

32. Make "improvements" to your code often, and force users to upgrade often - after all, no one wants to be running an outdated version. Just because they think they're happy with the program as it is, just think how much happier they will be after you've "fixed" it! Don't tell anyone what the differences between versions are unless you are forced to - after all, why tell someone about bugs in the old version they might never have noticed otherwise?

33. The About Box should contain only the name of the program, the names of the coders and a copyright notice written in legalese. Ideally it should link to several megs of code that produce an entertaining animated display. However, it should **never** contain a description of what the program is for, its minor version number, or the date of the most recent code revision, or the website where to get the updates, or the author's email address. This way all the users will soon all be running on different versions, and will attempt to install version N+2 before installing version N+1.

34. The more changes you can make between versions the better, you don't want users to become bored with the same old API or user interface year after year. Finally, if you can make this change without the users noticing, this is better still - it will keep them on their toes, and keep them from becoming

complacent.

35. If you have to write classes for some other programmer to use, put environment-checking code (getenv() in C++ / System.getProperty() in Java) in your classes' nameless static initializers, and pass all your arguments to the classes this way, rather than in the constructor methods. The advantage is that the initializer methods get called as soon as the class program binaries get *loaded*, even before any of the classes get instantiated, so they will usually get executed before the program main(). In other words, there will be no way for the rest of the program to modify these parameters before they get read into your classes - the users better have set up all their environment variables just the way you had them!

36. Choose your variable names to have absolutely no relation to the labels used when such variables are displayed on the screen. E.g. on the screen label the field "*Postal Code*" but in the code call the associated variable "*zip*".

37. Java lets you create methods that have the same name as the class, but that are not constructors. Exploit this to sow confusion.

38. Never use layouts. That way when the maintenance programmer adds one more field he will have to manually adjust the absolute co-ordinates of every other thing displayed on the screen. If your boss forces you to use a layout, use a single giant GridBagLayout, and hard code in absolute grid co-ordinates.

39. In Java, disdain the interface. If your supervisors complain, tell them that Java interfaces force you to "cut-and-paste" code between different classes that implement the same interface the same way, and they *know* how hard that would be to maintain. Instead, do as the Java AWT designers did - put lots of functionality in your classes that can only be used by classes that inherit from them, and use lots of "instanceof" checks in your methods. This way, if someone wants to reuse your code, they have to extend your classes. If they want to reuse your code from two different classes - tough luck, they can't extend both of them at once!

40. Make all of your leaf classes final. After all, *you're* done with the project - certainly no one else could possibly improve on your work by extending your classes. And it might even be a security flaw - after all, isn't java.lang.String final for just this reason? If other coders in your project complain, tell them about the execution speed improvement you're getting.

41. Make as many of your variables as possible static. If *you* don't need more than one instance of the class in this program, no one else ever will either. Again, if other coders in the project complain, tell them about the execution speed improvement you're getting.

42. Keep all of your unused and outdated methods and variables around in your code. After all - if you needed to use it once in 1976, who knows if you will want to use it again sometime? Sure the program's changed since then, but it might just as easily change back, you "don't want to have to reinvent the wheel" (supervisors love talk like that). If you have left the comments on those methods and variables untouched, and sufficiently cryptic, anyone maintaining the code will be too scared to touch them.

43. On a method called *makeSnafucated* insert only the comment `/* make snafucated */`. Never define what *snafucated* means **anywhere**. Only a fool does not already know, with complete certainty, what *snafucated* means.

44. Reverse the parameters on a method called drawRectangle(height, width) to drawRectangle(width, height) without making any change whatsoever to the name of the method. Then a few releases later, reverse it back again. The maintenance programmers can't tell by quickly looking at any call if it has been adjusted yet. Generalisations are left as an exercise for the reader.

45. Instead of using a parameters to a single method, create as many separate methods as you can. For example instead of `setAlignment(int alignment)` where alignment is an enumerated constant, for left, right, center, create three methods: `setLeftAlignment`, `setRightAlignment`, and `setCenterAlignment`. Of course, for the full effect, you must clone the common logic to make it hard to keep in sync.

46. The *Kama Sutra* technique has the added advantage of driving any users or documenters of the package to distraction as well as the maintenance programmers. Create a dozen overloaded variants of the same method that differ in only the most minute detail. I think it was Oscar Wilde who observed that positions 47 and 115 of the Kama Sutra were the same except in 115 the woman had her fingers crossed. Users of the package then have to carefully peruse the long list of methods to figure out just which variant to use. The technique also balloons the documentation and thus ensures it will more likely be out of date. If the boss asks why you are doing this, explain it is solely for the convenience of the users. Again for the full effect, clone any common logic.

47. Declare every method and variable public. After all, somebody, sometime might want to use it. Once a method has been declared public, it can't very well be retracted, now can it? This makes it very difficult to later change the way anything works under the covers. It also has the delightful side effect of obscuring what a class is for. If the boss asks if you are out of your mind, tell him you are following the classic principles of transparent interfaces.

48. In C++, overload library functions by using #define. That way it looks like you are using a familiar library function where in actuality you are using something totally different.

49. In C++, overload +,-,*,/ to do things totally unrelated to addition, subtraction etc. After all, if the Stroustroup can use the shift operator to do I/O, why should you not be equally creative? If you overload +, make sure you do it in a way that `i = i + 5;` has a totally different meaning from `i += 5;`

50. When documenting, and you need an arbitrary name to represent a filename use *"file"*. Never use an obviously arbitrary name like *"Charlie.dat"* or *"Frodo.txt"*. In general, in your examples, use arbitrary names that sound as much like reserved keywords as possible. For example, good names for parameters or variables would be: *"bank"*, *"blank"*, *"class"*, *"const"*, *"constant"*, *"input"*, *"key"*, *"keyword"*, *"kind"*, *"output"*, *"parameter""parm"*, *"system"*, *"type"*, *"value"*, *"var"* and *"variable"*. If you use actual reserved words for your arbitrary names, which would be rejected by your command processor or compiler, so much the better. If you do this well, the users will be hopelessly confused between reserved keywords and arbitrary names in your example, but you can look innocent, claiming you did it to help them associate the appropriate purpose with each variable.

51. Always document your command syntax with your own, unique, undocumented brand of BNF notation. Never explain the syntax by providing a suite of annotated sample valid and invalid commands. That would demonstrate a complete lack of academic rigour. Railway diagrams are almost as gauche. Make sure there is no obvious way of telling a terminal symbol (something you would actually type) from an intermediate one -- something that represents a phrase in the syntax. Never use typeface, colour, caps, or any other visual clues to help the reader distinguish the two. Use the exact same punctuation glyphs in your BNF notation that you use in the command language itself, so the reader can never tell if a (...), [...], {...} or "..." is something you actually type as part of the command, or is intended to give clues about which syntax elements are obligatory, repeatable or optional in your BNF notation. After all, if they are too stupid to figure out your variant of BNF, they have no business using your program.

52. The macro preprocessor offers great opportunities for obfuscation. The key technique is to nest macro expansions several layers deep so that you have to discover all the various parts in many different *.hpp files. Placing executable code into macros then including those macros in every *.cpp file (even those that never use those macros) will maximize the amount of recompilation necessary if ever that code changes.

53. Java is schizophrenic about array declarations. You can do them the old C, way String x[], (which uses mixed pre-postfix notation) or the new way String[] x, which uses pure prefix notation. If you want to really confuse people, mix the notations: e.g.

    byte[] rowvector, colvector, matrix[];

which is equivalent to:

    byte[] rowvector;

```
        byte[] colvector;
        byte[][] matrix;
```

54. Java offers great opportunity for obfuscation whenever you have to convert. As a simple example, if you have to convert a double to a String, go circuitously, via Double with `new Double(d).toString` rather than the more direct `Double.toString(d)`. You can, of course, be far more circuitous than that! Avoid any conversion techniques recommended by the Conversion Amanuensis. You get bonus points for every extra temporary object you leave littering the heap after your conversion.

55. Use threads with abandon.

Philosophy

The people who design languages are the people who write the compilers and system classes. Quite naturally they design to make their work easy and mathematically elegant. However, there are 10,000 maintenance programmers to every compiler writer. The grunt maintenance programmers have absolutely no say in the design of languages. Yet the total amount of code they write dwarfs the code in the compilers.

An example of the result of this sort of elitist thinking is the JDBC interface. It makes life easy for the JDBC implementor, but a nightmare for the maintenance programmer. It is far **clumsier** than the Fortran interface that came out with SQL three decades ago.

Maintenance programmers, if somebody ever consulted them, would demand ways to hide the housekeeping details so they could see the forest for the trees. They would demand all sorts of shortcuts so they would not have to type so much and so they could see more of the program at once on the screen. They would complain loudly about the myriad petty time-wasting tasks the compilers demand of them.

There are some efforts in this direction: NetRexx, Bali, and visual editors (e.g. IBM's Visual Age is a start) that can collapse detail irrelevant to the current purpose.

The Shoemaker Has No Shoes

Imagine having an accountant as a client who insisted on maintaining his general ledgers using a word processor. You would do you best to persuade him that his data should be structured. He needs validation with cross field checks. You would persuade him he could do so much more with that data when stored in a database, including controlled simultaneous update.

Imagine taking on a software developer as a client. He insists on maintaining all his data with a text editor. He is not yet even exploiting the word processor's colour, type size or fonts.

Think of what might happen if we started storing source code as structured data. We could view the **same** source code in many alternate ways, e.g. as Java, as NextRex, as a decision table, as a flow chart, as a loop structure skeleton (with the detail stripped off), as Java with various levels of detail or comments removed, as Java with highlights on the variables and method invocations of current interest, or as Java with generated comments about argument names and/or types. We could display complex arithmetic expressions in 2D, the way TeX and mathematicians do. You could see code with additional or fewer parentheses, (depending on how comfortable you feel with the precedence rules). Parenthesis nests could use varying size and colour to help matching by eye. With changes as transparent overlay sets that you can optionally remove or apply, you could watch in real time as other programmers on your team,

working in a different country, modified code in classes that you were working on too.

You could use the full colour abilities of the modern screen to give subliminal clues, e.g. by automatically assigning a portion of the spectrum to each package/class using a pastel shades as the backgrounds to any references to methods or variables of that class. You could bold face the definition of any identifier to make it stand out.

You could ask what methods/constructors will produce an object of type X? What methods will accept an object of type X as a parameter? What variables are accessible in this point in the code? By clicking on a method invocation or variable reference, you could see its definition, helping sort out which version of a given method will actually be invoked. You could ask to globally visit all references to a given method or variable, and tick them off once each was dealt with. You could do quite a bit of code writing by point and click.

Some of these ideas would not pan out. But the best way to find out which would be valuable in practice is to try them. Once we had the basic tool, we could experiment with hundreds of similar ideas to make like easier for the maintenance programmer.

I discuss this further under SCID and in the SCID student project.

Contributors

The following are some of the people who contributed to this list. My lawyers recommended I exclude those who taught by example.

- Hugh McDonald, hughmcd@ican
- Gareth Meyrick, gareth@pangloss.ucsf.edu
- Jarle Stabell, jarle.stabell@dokpro.uio.no
- Ko-Haw Nieh, niko@quality.com
- Jim Johnson, jimj@jumpmusic.com
- Jim Hyslop, Jim.Hyslop@mars.leitch.com
- George Ruban, gruban%adsl4@gte.com
- Mats Carlid, mats@adbk.se
- John P. McGrath, mcgrath@enter.net
- Brian Hurt, brianh@bit3.com
- Chris Schlenker, Christoph.Schlenker@gfk.de
- Nicholas Widdows, nicholas.widdows@traceplc.co.uk
- Greg Compestine, gregcompestine@caleb-bldr.com
- Carl L. Gay, sigue@thecia.net

This article appeared in Java Developers' Journal (volume 2 issue 6). I also spoke on this topic in 1997 November at the Colorado Summit Conference. It has been gradually growing ever since. I have had quite a few requests for permission to build links here. You are welcome to.

If you enjoyed this essay you might like this one on how to write like a newbie.

.

Canadian Mind   You can get an updated copy of this page from http://mindprod.com/unmain.html

The Mining Company's Focus on Java

Products

Best of the Net
Award