A taxonomy of programmers.

David Boundy

boundy@apollo.hp.com

The following are a collection of notes I wrote for a C Programming course a couple years ago. I devoted one lecture to Kernighan & Plauger's "The Elements of Programming Style" (Bell Telephone Laboratories, 1974), and provided this as a take-home supplement.

These notes are somewhat dated in their ignorance of object-oriented methods.

The Elements of Programming Style is now fifteen years old. In many respects, it has accomplished its purpose: the truly terrible code common among FORTRAN, PL/I and BASIC programs that tEoPS takes as its bad examples have been largely rooted out. The main factor has been the shift to block-structured programming languages and the development of block-structured discipline (to which tEoPS was a large contributor): using Pascal or C, it takes a determined weasel to write really bad programs (indeed only about half the points from Kernighan and Plauger are relevant to you as C programmers). Most of the the programmers who must program in the older languages have usually been exposed to the modern languages and programming concepts, and bring these sensibilities to their work.

The issues we face as we try to become better programmers in 1991 are quite different than those that were stressed in the 1970's. Writing code that can be read is still just as important, but the emphasis has shifted from creating decent loop structures and other practices now taken for granted, to other areas.

The skill that most employers now search for in their hiring, which most schools seem not to teach well, and that I've tried to stress so you can start to teach yourself, is good **design** and good implementation strategies.

The issue of "good programming" is not purely academic nor aesthetic: along with punch cards, another facet of computer science today's students miss is *how much easier* programming is now than it was in the FORTRAN/BASIC days. I've experienced programs that came together in half the expected time using "good" programming languages and techniques. The number of bugs reported against well-designed software and spent fixing each individual bug is much lower.

Here's one more review of some of the stuff I've talked about, plus a look at what you'll be learning in years ahead.

The minimally competent programmer who passes his freshman course.

The minimally competent programmer's code expresses a complete thought in each statement, and each statement has most of the information needed to make sense of itself without requiring too much surrounding context.

The minimally competent programmer at least puts a header comment on **every** function telling **what** the function does, and what each argument to the function is for. If the function is at all complex, he also includes a note about **how** it does it and **why** it does it that way. There are exceptions for cases where this comment would be totally redundant because the function is utterly trivial and its purpose is completely specified and understandable from its name and argument list.

Half the documentation of a program is in its names. The minimally competent programmer at least follows **Boundy's Laws of Naming**:

0. **Choose names that are meaningful and specific!** Spell the name so that it has mnemonic value, and choose the name so that it means what it says! Think twice about any name that starts with "do" or that contains a pronoun -- can it be changed to a *module_$verb_object* or *module_$verb_adverb* form? Be sure that a name captures what the object means, and provides the necessary distinction from nearly-similar concepts (in the program or not) that the object does *not* mean.
1. **Don't give two names to one thing.** Assembler macros that expand to a single line should be questioned. C macros that expand to a single macro should be banned. Don't write one-line functions whose only action is to call another function.
2. **Don't give two things to one name.** If you have two separate concepts, use two different names. Many compilers are smart enough to assign two variables whose lifetimes do not overlap to a single location. Don't overload a name that already has an industry-standard meaning.
3. **The length of an identifier** should, in some sense, be proportional to the size of the scope of its object. A loop counter that is used only within a 2-statement loop may have a name like "`i`". Names of global data types and their members, global variables, and library functions and their arguments, etc. must be much more descriptive: "`module_$specific_action()`".

Three tests for names that are just "wrong":

- The type of the name does not match the type of the object. For instance,

```
CONST
    big_buffer  = 512;
VAR
    buffer      : ARRAY [ 0 .. big_buffer ] OF char;
```

  The type implied by the name `BIG_BUFF` is `char[]`; the actual type is `integer`.
- There are two objects whose names can be switched without making as substantial disimprovement. Both names need to be made more specific.
- It's possible to tell the order in which two objects were implemented, because one has a suffix: for instance if you see two functions, one named `add`, and one named `add_dynamic`. The first one should be renamed `add_static`.

There's nthng immrl abt vwls. Give names that are easily readable. My group will spend 10 times as much time reading any particular statement as was spent typing it in. It's awfully hard for anyone other than the programmer to reconstruct the meaning of a name like "dfhppt."

Another note on naming: Don't name versions "new," unless you're willing to name them "newer,", "newer_still," etc. If a format changes (and you have to preserve both the old and new functionality), name both with version numbers, dates, or similar.

The minimally competent programmer at least uses a consistent and illuminating indentation style. This style is self-consistent: the rules for `BEGIN/END`'s or { }'s around function bodies is the same as for `if`'s, `switch/case`, data structure declarations, etc. The `END`'s or close-curlies unwind uniformly at the bottom of nested structures (so that a missing one stands out). If his work group already has such a style in place (or if there is a universally-accepted "standard", like "C" as presented in The White Book), he adopts it (or works to change the group's style if the existing one is inconsistent).

He understands that `if / else if / else` is a parallel control structure (like `switch / case`), not a nested one, and codes it appropriately.

He uses white space to denote "paragraphs" of his program. He uses naming and indentation conventions to add meaning "between the lines", and follows these conventions absolutely consistently.

Put the comment where I'll see it, not where I have to look for it. If the choice about whether a loop should go from 1 to N or N to 1 is not completely obvious (or arbitrary), put a comment at the top of the loop where I can't help tripping over it, not in a file header comment.. Header comments should describe the overview of a module — the big ideas I need to know so that the details make sense. Source comments should document the data- or control-structure by -structure microview.

The minimally competent programmer never, ever bakes constants into his code. He would rather die than write something like:

```
while( number_of_vars < 255 )
```

He declares a named constant, something like "`max_nr_trackable_vars`."

The minimally competent programmer worries about little things: "A detail here, a detail there — sooner or later, you've got real engineering!" He pays enough attention to what he's doing so he doesn't end up with:

```
while( TRUE )    {
    if( xp->kind == blk_end_node && xp->blk_end_node.bp->kind != inner_blk)
        break;
    ...
}
```

when this would suffice:

```
while( xp->kind != blk_end_node || xp->blk_end_node.bp->kind == inner_blk) {
    ...
}
```

As he programs, he has a firm idea of what each statement, each loop, and each function contributes to the mission of the entire program. He knows what procondition must be established on entry to each control structure and what postcondition is established by that structure. He knows that these conditions make very valuable comments.

The minimally competent programmer begins to direct his style toward writing code that is to be read by humans, and that can be read and understood by any other minimally competent programmer.

The minimally competent programmer can write a big, complicated program to find the answer, and add features to a program by adding more code.

The competent college graduate, journeyman programmer.

The competent programmer extends the concepts he uses to write statements clearly to functions. He starts worrying about writing in a style that will result in fewer bugs, both in initial implementation and during lifecycle maintenance.

Top-down design, bottom-up implementation: The first thing he does in any programming task is to analyze the entire problem into smaller problems that can be solved separately. He begins coding by writing functions that implement the primitives and building blocks he will need. The rest of the program then almost writes itself!

The journeyman programmer writes code that has very little redundancy — if four (or so) lines of code appear more than once, he realizes that his function boundaries are flawed. He may simply extract the redundant lines into a function, or he may undertake a larger reorganization. This technique reduces the number of modifications he has to make later, either to fix bugs or to add new functionality. Combining separate instances of nearly-similar code make the similarities obvious; the control-flow in the function and the differing arguments in the calls enhance the contrasts. It also increases the probability that when he later finds another nearly-similar case, it will already be covered by the code he already has. Fewer, more general routines get tested better than less-general, less-frequently called ones.

By the time a programmer takes his first "real" job, he should be writing functions that hide details of *operation* from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes. A function should package one "idea": its purpose should be describable in one sentence without ANDs or ORs — and, most importantly, without EXCEPTs! A function should isolate dependencies, so that a change to one part of the program is less likely to cause a strange bug somewhere else.

The competent programmer makes sure that his customers only see a bug once. He develops a test system in parallel with his product. For **every** feature he adds to his program, he writes a thorough test. **No** bug that is reported is closed until a test is added to his test suite.

The competent programmer writes code that checks up on itself. His programs contain assertion checks like:

```
if( something_has_started_to_go_wrong( here ) )    {
    printf( "problem xxxx at f() line %d\n", __LINE__ );
    exit( horrible_death );
}
```

Users of his program are occasionally disappointed when the program prints out an internal error check message and dies; they can usually figure out how to change their input so they won't hit the bug, and having done so, they have confidence that they're back on safe ice. They are very seldom enraged when the program gives wrong answers. (It's also much easier to debug a program that diagnoses itself.)

The journeyman programmer writes code that he can forget. He always looks at his programs through the eyes of a person new to his project: could he pick it up and begin to work on it quickly? (A bit of career advice: if no one else can work on your program, you'll find yourself stuck to your tar-baby long past the time you wish you were doing something else!)

A competent programmer can pick up a listing of a great programmer's code, and, after reading the introductory comments and the comments in the data type descriptions, flip to a random page in the program and quickly determine what that function is doing, the context in which it operates, and generally how it accomplishes it. A competent programmer can safely be expected to make minor changes to a great programmer's code, without having previously seen that code, in a minimal amount of time.

The competent programmer realizes that he must sacrifice a few percent of his productivity today for the long-term good of his organization. It's penny-wise for him to do a quick-and-dirty job on something that his work-group will have to maintain at pound-foolish expense for many years.

The competent programmer realizes that terse code and terse programming languages require verbose comments.

The competent programmer can write a small, simple program to find the answer, and can add features to a program by adding new members to data structures.

The good programmer, or as we call him in 1989, "the software engineer".

The good programmer uses techniques that will result in programs that work. But more importantly, his programs will continue to work even when other parts of the program change. His programs can be extended — he knows that he will be asked to add new features, and to cover new cases.

This programmer writes a lot more than programs. He writes a document that tells what his program will do — what problem he's going to solve, so that the customer and the programmer know what they expect of each other. He then writes a document that tells how his program will behave — a preliminary user manual. Then the programmer writes a document that describes how the program will work: data structures, algorithms, and structure. Only after he's gotten the customer's blessing for the first two and he's sure he understands the third, does he begin coding.

The good programmer thinks in terms of modules and managers. He groups related functions together and publishes an interface in a header file. He hides large parts of the implementation (data structures, helper functions, etc.) inside the module, so he can change the implementation without changing the interfaces.

About this time, the good programmer begins to glimpse the full power of good data structure design. As he does his top-down designs, the data structures tend to be one step more specific than the "algorithm" part of the design.

The good programmer uses the strongest possible invariants (consistency properties between data that must be preserved by each statement or loop iteration), even at some cost in execution time or memory consumption. He makes rules for himself and his program and absolutely rigorously follows them. These rules may be extremely clever, but should be easily-stated concepts. "Data structures are always linked this way." "Member `a` of `struct s` always describes the left child, never the right (even if some members of the `union` inside of `struct s` only have a right child!)." Imposing good laws on oneself is very liberating: when working on one portion of a program, he knows that there are certain behaviors he can expect from other parts — because *all* parts obey certain rules. These rules are described in comments — and "enforced by appropriate legislation": the program checks as many as it can.

The good programmer writes software that can be reused, even if he doesn't see a reuse immediately on the horizon. His experience with things seen and faith in things not seen assures him that someone from the next hallway, reasonably soon, will be asking, "Do you have a module that ..." He also knows that writing reusably will force him define clean interfaces.

The good programmer took more than computer science courses. He has a strong concentration in basic mathematics: calculus and differential equations (a programmer must understand the real-world dynamic systems he's modeling), discrete mathematics and graph theory (sets, functions, relations, graphs, etc. form the basis for most good programs), logic, linear algebra, probablilty and applied statistics, and optimization theory. He should have some chemistry and physics (experience in applied mathematics). Every "engineer" should have some understanding of electricity and magnetism and mechanics. His computing science coursework should go beyond "C Programming:" problem solving and program design, systems and control theory, information theory, digital systems and logic design, computer system documentation, design and analysis of data structures and algorithms, computer architecture, and numerical analysis. A background in the engineering disciplines forms a much better base for a growth career of life-long learning than any amount of noodling with today's whizzy but soon-to-be-obsolete databases, compilers, artificial intelligence tools, or operating systems.[1]

The good programmer understands the analysis of algorithms. If his algorithm is slower than $n \log n$, he looks hard for a faster one.

The good programmer understands something about the underlying mechanisms common to all computers (the kind of information taught in assembly language and computer architecture classes). He uses this information to chose programming techniques that improve efficiency, while never sacrificing readability, maintainability, or reusability. For instance, he arranges things so most comparisons are to zero (for instance, loops count down to zero) — all machines can do comparisons to zero for free, and must execute at least one more instruction to compare to anything else. The most-often referenced member of a `struct` is at offset zero into that `struct` — many machines can reference a datum at zero off a pointer faster than they can at some displacement off a pointer. Multi-dimensional arrays are laid out so that addressing is simplified. Some machines insist on naturally-aligned data; all prefer it. Most machines' virtual memory and vector hardware schemes are quite similar; if one pays attention to working with these features instead of ignorantly working against them, one can dramatically improve performance.

---

[1] David Parnas, "Education for Computing Professionals", IEEE Computer, January 1990

The good programmer also paid attention in his architecture class when *differences* between architectures were discussed. His programs do not break when ported to machines where an `int` is 16 bits. They do not break when ported from big-endian to little-endian machines, nor from stack-grows-down operating systems to stack-grows-up.

A good programmer must understand numbers. For any problem someone states, he can come up with an answer that's correct within a factor of two. "What's the weight of the air in a Boeing 747?" "P.T. Barnum said 'There's a sucker born every minute.' What portion of the population, as of 1890, when he said it, were suckers?" When your morning newscaster says, "Massachusetts Electric asked the Public Utilities Commission for a $20 million rate increase today," you should know that that's a meaningless statistic (for everyone except Mass Electric!), because unless you already know the total income of Mass Electric, you have no way of relating $20M to your electric bill. Even if the newscaster says "This is expected to add $2 to the average monthly utility bill," you should think realize that nobody is an average customer — those who have electric heat will be affected much more than average, and those without will be affected much less. You should call him and ask him to announce that "this will add 8% to electricity bills." A good programmer has to know what numbers it's useful for his program to produce. Almost all programs model some feature of the real world; before he can teach a computer about that real world, he must understand it better than the computer.

A good programmer's standing as a good programmer is not damaged by rewriting bad code to improve functionality and readability. The side benefit is that rewriting is one of the best ways to learn the most hideous, otherwise inscrutable malformations: by the time he's done, he understand intimately (and so will the next guy who reads it).

The good progammer knows that a convenient bad design decision today will force a string of painful decisions between bad alternatives tomorrow.

The good programmer searches for a tool-based approach to his problem.

The good programmer tries to become a great programmer by seeking out a great programmer to work with and study under, and great programs to read.

The good programmer knows of a small, simple program that already finds the answer, and can add features to a program with no appreciable change in code or data.

The great programmer.

The great programmer focuses on designs that will give his large programs an "organic unity." He develops concepts that will be used throughout the program that will tie it together, and allow someone who understands one part of it to work comfortably on another part with which he is less familiar.

A great programmer writes many languages well (English not least among them!), knows more than one operating system, and understands the benefits and limitations of each. He carries the benefits over into his other environments, and borrows approaches to overcoming the limitations.

The great programmer thinks in terms of layers; each layer manages a data type. The calls in the program are either across within a single layer, or down to a lower layer. The interfaces between layers are very general, even at some cost in efficiency.

The great programmer takes pains to learn something about the layer beneath him — for instance, an applications programmer learns about database systems so that he will lay out his data to work with the database manager, not against it. A great systems programmer can hold an intelligent conversation with a processor designer.

A great programmer knows that the finished product will have the best quality and lowest cost if it's done right from the beginning, even if it may be marginally slower getting out the door. He knows that trying to maintain a system architected less than right is a black hole.

The great programmer solves the whole problem. He knows that relying on nuances of the problem specification to decide whether to include or exclude functionality will make enemies. He also knows that problem specifications change with the weather, and so produces a general design.

The first architectural point to which a great programmer turns his attention is his data structures. Good data structure design is often literally the difference between a program that accomplishes its task, makes it look easy, and keeps its programmers excited vs. a program that accomplishes only part of its specified function, is finished late and over budget, is a continual source of bugs, and turns well-adjusted, productive engineers into unhappy, unbearable jerks.

The great programmer understands that good data structure design can make the difference between an algorithm of n^3 time complexity and a linear-time one.

The great programmer designs the fewest possible of the most generic possible mechanisms that will provide a means of solving any problem he's likely to encounter during any phase of development. This mechanism will, no doubt, be slightly more complicated than the *ad hoc* method that would have been used to solve any one individual subproblem. But the whole system is much simpler (and better tested and more reliable) because the number of mechanisms is smaller, and the various parts can use common utility routines.

The great programmer comments each function or code section with a statement of the "weakest precondition," the minimum set of things that a must be true before the function is executed. This accomplishes several things:

- If he finds some kind of ridiculous requirement, having to write this statement embarrasses him into appropriately "bullet-proofing" his code.
- It forces him to really think about what the function is doing; he will likely write a better implementation the first time through.
- When the function is reused, a comment of this form makes it absolutely clear what the caller has to do in order to use the function successfully.

The great programmer takes the time to become comfortable with elementary theory of computation. Finite state machines and pushdown automata are valuable tools for a wide variety of programming tasks.

The great programmer uses table-driven approaches. His program looks at the input, classifies it, then looks in a table for a list of actions to take, rather than using a complex series of nested `ifs` to decide what to do. This technique resolves complex interactions into small, independent, easy-to-debug pieces.

The great programmer is familiar with the techniques for mathematically rigorously proving programs correct. It's not that he takes delirious joy in an activity where grad students find only tedium, but he understands that if he writes programs that *could* be proven correct, he will have written programs whose control flow is clean and that more likely *are* correct.

Even a great programmer understands that, after being away from a piece of code for a while, he will have little advantage over any other reader. He thoroughly documents even code that may never be seen by others. This is done clearly, concisely and without the assumption that his reader knows the architecture of the entire system.

Even a great programmer realizes that anyone can make simple mistakes or oversights, and tries to make sure that no code goes into production that has not been read by one or more peers.

The great programmer knows how to put together a prototype of a proposed product by wiring together pre-existing tools, but knows that for a production quality product he'll have to rewrite the prototype. He also knows that all too often it's the prototype that gets shipped, and plans accordingly.

The great programmer knows and appreciates the the rules about "no `gotos`", limiting the number of global variables, etc. (but also has the judgment to rarely decide to break them!).

The great programmer knows of another group whose program already finds the answer, and adds his customer to that group's mailing list. The great programmer adds features to a program by removing code.

The truly great, world-class programmer.

The really great programmer doesn't write programs. He writes tools that write his programs for him.

World-class programmers plagiarize world-class programmers.

Truly great programmers realize that programs, processes, languages and data can be molded and connected at will to best solve the problem at hand, and have the creativity to come up with a lot of will. New programming styles can be chosen or invented. Languages can be tailored to the problem. Data can be massaged. Processes and processors can be connected in new and unique ways. In the hands of a truly great programmer, the boundaries between application software, system software and hardware are flexible.

This section is very short because I'm still trying to discover the secrets of the world-class programmers I work with!

Good software engineering is not magic, it's taking the time up front to develop a robust, extensible, reusable architecture, paying the price to either stay within or carefully extend the architecture, and then paying attention to details during implementation. There are lots of choices we make every day about whether our software is going to be a joy or a misery to maintain. Here's to good choices!

Thanks to world-class examples John Yates and Bill Ackerman, and to Jim Perry, Joel Stave, Joe Bowbeer, Debbie Minard, Tom Van Court, and Roz Hoffman for their contributions.

Safety-Related Skills for the BCS Industry Structure Model, Michael E. Falla

The British Computer Society publish an 'Industry Structure Model' which provides a set of performance standards covering most areas of work in the field of Information Systems Engineering. The ISM is designed to provide guidelines to define training needs, career development structures and experience requirements for those who wish to become professionally qualified within the industry. The ISM has been well received in the UK and is now in use in the USA and elsewhere. It is also shortly to be translated for use as a tool for skills identification across the European Community.

At many points it refers to the need for special attention to be paid to safety matters, and the BCS felt that an appendix should be provided in the next release indicating the specific attributes, knowledge and competence regarded as important for those engaged in the development, maintenance or management of safety-related software. Some initial ideas were put together by Prof John McDermid of York University and the late Alan Taylor of the BCS, and I was contracted to draft a text, consult expert opinion and to produce an appendix in line with this advice.

We have been fortunate in receiving help and comments from 26 experts from industry, academia and government, three of whom have been particularly generous with their time. By circulating a draft which bore some relationship to the final text we were able to maximise the input from people who are inevitably extremely busy. It is testimony, however, to how important these experts feel the subject to be, that so many have been prepared to make time to be so helpful.

Much of the appendix proved uncontentious - as one would hope, but a few issues of interest came up. One concerned the educational requirements for those who work on safety related topics: is a university degree or equivalent a sensible requirement for everyone? Is Chartered Engineer status desirable for this work? Similarly, in the absence (to date) of accepted course curricula, what can be said about training requirements?

On the technical side there was some doubt about whether it was sensible to consider safety-related software as a separate issue, as opposed to complete systems. On balance it seemed best to concentrate on software, but set in a systems context.

The appendix tries to give an indication of the broad classes of technical issues which practitioners and managers should be familiar with, and quotes examples of techniques and tools to illustrate these. There turned out to be more consensus about these topics than I expected.

An important issue concerned attitudes of personal responsibility and how these can be defined, particularly in the presence of other pressures, such as those of time, budget or unsympathetic management.

It became clear that the role of the safety assessor needed special attention, and from that emerged the need for a distinct career 'substream' (in the terminology of the ISM) of 'Software Safety Specialist'. Since such people already exist, and their numbers are likely to grow with the increasing importance of software in this field, it was felt necessary to recognise this career path explicitly.

Further information on the ISM can be obtained form the PDS Administrator at the British Computer Society, PO Box 1454, Station Road, Swindon SN1 1TG.

Mike Falla is an Independent Consultant and can be reached at 35 Benbrook Way, Gawsworth, Macclesfield, Cheshire, SK11 9RT, United Kingdom, Tel +44 625 431301.