

Seven Habits of Highly Effective Programmers

by Philip Chu

Table of contents

1 Publication Information.....	2
2 Understand Your Requirements.....	3
3 Keep It Real.....	4
4 Understand Your Code.....	5
5 Optimal Programming.....	8
6 Manage Thyself.....	9
7 Continuous Education.....	11
8 R-E-S-P-E-C-T.....	13

1. Publication Information

Copyright ©2004-2009 by Philip Chu All rights reserved.



As a software engineer, you might want any number of things out of your job - a steady paycheck, the opportunity to work on interesting projects, a springboard to the next better job, or maybe you just like hanging out with other programmers. But by "effective", I mean the ability to complete projects in a timely manner with the expected quality. After working on dozens of software releases, I believe the following practices will bring you there, and while they may involve sticking your neck out, I'd like to think they will also advance your professional reputation, career longevity, and personal satisfaction.

2. Understand Your Requirements

The first step in becoming an effective programmer is to ensure that you are spending your time wisely. And there is no greater waste of time than in working on something that is not useful or never shipped.

2.1. Build Early

Get a demonstrable system working as early as possible. This means establishing the interface first, whether it's an API or user interface, and stubbing the encapsulated functionality as necessary. This allows your "customers" to check it out, by exercising the user interface or writing code to the API, and any inconsistencies or omissions in the initial spec can be detected early. Chances are, you will notice problems or potential improvements even before releasing this first deliverable.

There is a classical school of thought that believes if you design everything up front, then all you have to do is write the code and you're done. That works great if you've done the exact same project before. Otherwise, it's more likely you'll run into a point where you're just guessing or operating on questionable assumptions.

- Upon joining an early-stage wireless internet startup, I found myself in two months of design meetings for a wireless portal and gateway due to launch in six months. Eventually we got tired of meeting and finally started coding. Within two weeks, my part of the project had no resemblance to the original design, and the first wireless connection test two months later revealed a fundamental misunderstanding of the wireless protocol.

This is not to say that design is unnecessary. But after a certain point, design is just speculation. Design should be validated with implementation, and better to do that early and continuously than late and, well, too late.

Even if the original design is sufficient, once you have something you can tweak, you can improve upon it. Hardware products (who designed this VCR?), buildings, and large-scale software projects suffer from interfaces that were frozen in "preproduction", but with software, you have an opportunity early in the project to refine your understanding of the requirements and produce a suitable interface. But it must be done early.

Getting something ready early is also good for your occupational well-being. Your boss will appreciate seeing evidence that something is actually getting done and having something available to demo. On the other hand, a drawn out period with nothing to show is a recipe for anxious management.

2.2. Deliver Often

Once you have something working, don't just leave it as a "proof of concept". Let people play with it, see their reactions, and let this guide and prioritize your development. There is no substitute for watching how people use your software. Customer questionnaires and focus studies might provide some useful input but run the risk of transferring feature and design decisions from the developer to the customer.

In particular get the software into the hands of the QA staff as soon as possible and feed them regular builds, preferably at scheduled intervals. Having them test automated daily builds is ideal, but even a weekly build is pretty good. This will help them feel involved in the full life-cycle of the project and they should be best-trained at identifying and reporting problems. The highest priority should be given to issues that prevent them from using the product, e.g. crashes or dead-end paths - you want them to cover as much as possible as soon as possible and get a feel for the whole product so design issues can be identified early.

- At a small 3D graphics software vendor, I was put in charge of porting the flagship product from SGI workstations to Windows NT. After six months, the port was so incomplete and crash-prone that I was reluctant to give the first "alpha" build our test group. Fortunately, the QA manager insisted, and the resulting bombardment of bug reports forced me to immediately focus on the problems that prevented the testers from exercising the application in any meaningful way. Left to my own devices, I would have worked on what seemed to be the harder and more important core 3D issues, and probably delayed too long on seemingly mundane issues like the user interface, load-save functionality, and compability with all the varieties of consumer hardware we were planning to support.

Programmers often don't want to release code to testers early - they don't want to hear about a bunch of bugs they already know about, and quite possibly the testers don't want to test something that barely works. But it's the testers' job to find these problems and programmers need to realize bug reports are a good thing, if they arrive early enough.

3. Keep It Real

Keep your software running in as close to a shipping state as possible. You never know when you'll have to demo the system, send out an evaluation copy, or even deliver ("OK, time to wrap things up!")

3.1. Use Real Data

If you just test with sample data, that big iceberg of real data out there is going to sink your

program.

- One of the leading semiconductor fabs evaluated a supply chain management product I was working on. After crunching out a milestone delivery to them, we got word back that the first batch of data they fed it from their own operations was still processing - for two days. I sympathized with the lead programmer, who had to dig down and emergency-optimized everything he could for two weeks with both management and client breathing down his neck. I'm just glad it wasn't me on the line.

3.2. Use Real Builds

Remember the development build on your machine is not the real build.

- On a recent game development project where I worked on the user interface, I got intermittent reports from QA that some colors were not correct. Eventually, I realized the problem only showed up in release builds and another programmer used the special console debugging hardware to track down the bug. Which turned out to be a silly mistake I'd made two months previous, failing to specify an initial color value in a few cases. The debug build always selected a specific default value, while the release build optimized that away and the result was less determinate. If I'd made a point of running the release build frequently, I would have spotted my mistake immediately, instead of losing it in the sands of time.

3.3. Merge Often

Don't procrastinate on merging your code with the main code base - the longer you wait, the harder it gets.

- I worked with a programmer who "couldn't be bothered with" all the new code and data changes that showed up in the repository every day. And certainly, daily merges did take up some time for all the other programmers, and this programmer was able to run some impressive standalone demos with a snapshot of the code and data. But every time we had a milestone delivery, it took days to get the isolated code reattached to the current codebase again, sometimes compromising the milestone delivery and risking the funding for the entire project.

Keeping your code out of the official build means that programmers cannot evaluate your code and testers cannot spot bugs early. Maybe you don't want people picking on your code or bugs, but it's better to identify those issues early than later - suck it up.

4. Understand Your Code

Life is full of wonderful mysteries, but your code is not the place for them. You don't have to know how your car works - if the engine starts making strange noises, you drop it off the mechanic. When it comes to your code, if you don't understand how it works, or doesn't work, no one will.

4.1. Code with Style

My childhood piano teacher once commented to me, "Your sister has a good sense of timing, and your brother has a good feel of the keyboard." Then he paused. "You, uh, you work hard."

Programming is one of those things that a lot of people are more or less competent at, but some in particular have a flair for it. I'm a lousy piano player despite years of lessons, and I'm a mediocre basketball player although I enjoy playing it immensely. But I do like to think I have a flair for programming and writing. And not surprisingly, I think good programming is like good writing. Both prose and code are textual, have grammar, syntax, spelling and semantics and spelling. For most coders and writers, this is enough, but the best writers and coders have an esthetic and their work features structure and style that can often be identified with the author.

- Many Windows programmers wonder why grumpy old Unix/Mac/Amiga/Lisp programmers rail against Win32/MFC/.NET, but if all the API's you've seen are from Microsoft, you probably don't know there's anything better.

Perhaps not everyone is capable of writing stylish code - I've heard it said that good object-oriented programmers, in particular, are born and not made. But like fine music, wine, and literature, you can learn to appreciate good code.

4.2. Cut-and-Paste

The opposite of stylish programming is cut-and-paste. Grab some code from somewhere that is supposed to do something like what you want, tweak it until it sort of works, stir, repeat, and voila, you have the software equivalent of mystery meat.

- A few months after leaving one company, a former coworker emailed me a single function consisting of ten pages of cut-and-paste code and asked why it wasn't working. I could have very well asked why it should work at all. If you can't explain how your own code is supposed to work, how can you expect anyone to help you with it? (He has since moved on to a management position at Microsoft)

I've even had trouble diagnosing my own code that was cut-and-pasted from sample code. It's a reasonable way to start new code, but you can't just leave it alone when it seems to work -

you have to go back and make sure you understand it line by line and clean it up for your own purposes.

4.3. Keep it Clean

The key to keeping your house/condo/apartment clean is to spend a little time cleaning it every day, or at least every week. If you wait until your abode is an unsightly mess, it's just too much damn trouble to clean it all up and you end up just doing a halfhearted job. Or you hire a cleaning service.

Assuming you don't have the luxury of hiring someone to come in and clean up your code every week, you should periodically inspect your code, sweep up accumulated hard-coded numbers, outdated comments, misleading function names, or you'll inevitably end up with uninhabitable code that's embarrassing to show anyone else. And if you're not embarrassed, well, you should be.

- One programmer I supervised kept reporting to me that her code was "done". This is what management normally likes to hear, but it drives me crazy. Code is never done - you have to debug it, maintain it, evolve it until it's put out to pasture.

4.4. Questions? Comments?

Some like to think of programming as a craft. Others, engineering. More often than not, it's archaeology. You dig through sediments of code and wonder what purpose all these strange artifacts served. Do future generations a favor and leave some clues.

- I asked the aforementioned engineer whose code was "done" to add comments. The result, a function named GetData was prefaced by the comment "Gets data." That's not just useless - it's insulting. What data? (factory automation schedules) In what format? (a proprietary XML format) From where? (an in-house server, using TCP/IP) Not to mention little details like what happens when the server is unavailable or the transmission is broken.

Document your code as if someone else might have to take it over at any moment and know what to do with it. That person might actually be you - how often have you had to revisit your own code and thought to yourself, what was I trying to do here?

- On a contract with a previous employer, I was asked to look over a piece of code that no one else had time to attend to. At first, I thought it was a mess and didn't know what was going on in there. Then I gradually figured out what the code was doing, and I grudgingly admitted the code wasn't too bad. And then I eventually realized that I had written the code two years ago. Note to self: need more comments.

With that in mind, annotate your code as you write it, instead of waiting for a convenient cleanup phase in "post" - annotating as you code can even clarify your thoughts while you're programming. You can be your own pair-programming buddy.

As a bonus, these days you can generate nice HTML or otherwise-formatted documentation from source code comments, using javadoc, doxygen, whatever. Ideally, the doc-generation is part of your nightly build and available via your intranet.

4.5. Full Warning

Ignore compiler and runtime warnings at your own peril. They are called "warnings" for a reason.

- I shipped one Unix-based application that had a problem linking some functions successfully - we worked around it by relinking those functions at runtime. When we performed a clean rebuild six months later for the next release, it was revealed that we had turned off linker warnings which would have alerted us of a known linker bug. In our defense, we had swept the linker problems under the carpet at the Unix vendor's suggestion, (thanks, SGI!) but it turned out we could get the link to work just by reordering our libraries.

Crank up the warning levels on your compilers, sprinkle your code with assertions, and log the build and test-time warnings. Better yet, include a count of those warnings in your build metrics so you know if you're dealing with them or letting them accumulate.

5. Optimal Programming

5.1. Code with Purpose

On the other extreme from cut-and-pasters are those who change code just to make it look prettier (at least to them). While it's laudable to have a programming esthetic, it's a waste of time (and a useless risk) to change code just so it looks better to you. It's aggressively annoying to go through and change code that other people have written just so it looks better to you.

- A fastidious coworker of mine went through our codebase and removed all the expletives. Probably no one would have complained if he had just cleaned up the code written by the entry level employees, but the expletives belonged to the technical lead of our group who was also one of the few distinguished Fellows at the company.

5.2. Do No Harm

"Refactoring" is all the rage, now, but programmers often take it to mean any code cleanup or redesign. The trick is in reorganizing code for the better without breaking anything. If you break existing functionality in the name of progress, you're sending one of two messages: 1) your time is more important than everyone else's, or 2) you're incapable of touching code without breaking it.

- I had one particularly contentious coworker who decided to reimplement the parser in our system but left the code in an unbuildable state by everyone else. I asked him to revert his changes and then found the code was buildable but not runnable - asked about it, he replied that he removed the parser entirely "per your request". Not a team player.

Keeping the code functioning takes some patience and extra work - you have to be diligent about regression-testing your work and chances are you'll need to keep old code and interfaces around for a while as you migrate functionality to your new code. But for everyone to work with the same codebase, that's what you have to do.

5.3. Find the Bottleneck

People always talk about "optimization", but that isn't really a correct word. We're rarely shooting for the optimum - instead, we make improvements and tradeoffs to achieve good-enough performance.

- I was asked in a phone interview with Google how I would search for a number in an array of ordered numbers. Obviously, the questioner was asking for a CS 101 answer - binary search. But in real life, I would probably do the "wrong" thing - search the array from beginning to end. There's no point in taking twice the time to write twice as much code that has to be maintained and debugged if the application performance is good enough, and in particular if that piece of code is not the bottleneck in the application. (And I seriously doubt you'd have that data laid out linearly in a fixed array like that if it was the bottleneck)

If you do need to optimize for speed or space in you application, attacking anything other than the bottleneck is a waste of time.

6. Manage Thyself

You probably have a lot of complaints about your boss being a lousy manager, and you're probably right. So you have to be your own manager. Even if you have a decent boss, he's not going to stand behind you telling you what to type and how fast (although I'm sure many

would like to).

6.1. Are We There, Yet?

Programmers are notoriously inadequate at providing useful schedule estimates. I think this is a bad rap, since management, left to their own devices, often make even worse predictions, and unwelcome news from engineers is often ignored. (A common theme in any engineering disaster). But still, awareness of the schedule is critical to actually getting the project done on time.

- On one commercial software project, some of my coworkers were blissfully unaware of the product release date - one inquired whether it had been released already, another was surprised to find it was going out in a few days.

The worst, and most common, schedule estimate that a programmer can give is "it should just take a couple of days". Every time I hear that, even from my own mouth, I wince.

- The president of a graphics software company really wanted support for VRML (at the time it was the Next Big Thing) included in the product we were releasing in two months. He probably figured (correctly) that I would resist starting a new feature, so he went to another engineer and got the answer he wanted: "a couple of days". Two days later, I told the president we-just-wasted-two-days-of-his-time-and-mine-while-there-are-two-hundred-high-priority-bugs-to-fix, which he found to be a persuasive argument. (postscript: VRML didn't exactly take off like gangbusters)

And then there are programmers who are unable to come up with time estimates at all. But there's no need to get thrown off by the fuzzy nature of the request - it is an estimate after all, and in fact you should avoid using formulas. If you're an experienced engineer, you know how long comparable tasks have taken you before, and if you're not experienced, you can ask someone who is.

- A smart friend of mine who was often assigned to developing experimental prototypes asked me, "how can you schedule research". I think it was a rhetorical question, but even pure researchers have schedules. Someone is paying them and expects results, even if it's a number of demos or published papers in a certain timespan. And if you really don't have the foggiest idea how long something will take, then you're the wrong person for the task.

Sometimes programmers are reluctant to commit to a schedule because they're afraid of the accountability. It is true, in this imperfect world, managers will try to bargain you down on schedules, political factions may saddle you with tough or unrealistic schedules in the hopes

that you will fail, and it is a sadly common story that after you commit to a schedule, you don't get everything you need.

- I had one boss who after asking for an estimated completion time would say, "do you promise?" But ask for a commitment on the required hardware and other dependencies, and it was "I'll try."

All I can say is, stick to your guns and give a realistic prediction. Any concessions should be based on pragmatic tradeoffs between features and resources. Be clear about the assumptions, dependencies and resources on which the schedule is based, and get it written down somewhere so you can jog defective memories later.

6.2. Plan Your Progress

You wouldn't just hop into your car before deciding where you want to go, right? And probably you have a route in mind before you start driving, too. Similarly, before you sit down at our computer, you should know what you want to accomplish that day and have some idea how.

Distractions will come up day-to-day, so you won't always be able to accomplish what you want. And contrary to those who treat software engineering groups as vending machines (they would probably shake us vigorously if they could) some tasks take more than a day. So think about what you want to accomplish by Friday, and if you do, then you can enjoy the weekend that much more.

7. Continuous Education

A corporate soccer team member once asked me, as we were lacing up our cleats, "what's the secret to C programming?" If there was such a secret, I'd be hawking it on late night TV along with ab machines and how to get rich in real estate. Sorry, there's no shortcut - you have to learn and practice and make some mistakes. And you don't necessarily have to rely on corporate training or going back to school - there are plenty of (inter)national and local professional groups, books, and of course, the Internet.

7.1. It's Science

It's called "computer science" for a reason. It's easy (maybe too easy) for anyone to start programming, without a formal computer science education. In particular, those from other engineering and science disciplines can pick up programming quickly and make a good living. But to effectively tackle non-trivial tasks, you need to know the inherent capabilities and limitations of software and recognize prior work, so you don't waste time reinventing the

wheel, badly. You don't have to know everything under the sun, but you should have at least a cursory familiarity with many areas and be prepared to do some additional research as necessary.

For example, anyone who creates a new file format should know something about compilers. I don't mean all the code-generation optimizations like loop unrolling, but the basic issues and various phases of compilation and most of all the importance of specifying the tokenization and grammar. Nowadays, most people by default will use XML, and that's a good thing, but before then it was typical to cobble up some text format, point to some generated sample files as documentation, and then everyone else who wrote another parser would cobble something up that would read in some files but not all. In the problematic cases then you could point fingers either way - either the reader is bad or the writer is bad. Whichever product is more popular wins.

- One of my pet peeves with the 3D graphics industry is the plethora of ill-defined file formats. When I implemented an OBJ file parser for a 3D content creation product, each exporting product that I tested against generated markedly different files, using different whitespace and newline conventions, for example. In refreshing contrast, a coworker of mine fresh out of school designed a new game interchange format using a grammar and lexer specification. (These days, it's not much of an issue anymore - most new graphics file formats seem to be based on XML.)

And if anything differentiates programmers who can just put together simple scripts and user interfaces and those who can tackle real problems, it's an understanding of computational complexity, i.e. how algorithms scale with the size of the problem. Every programmer should know basic complexity terminology and have a general knowledge of the complexity of common problems.

- My first job was in computer-aided semiconductor design, which has a lot scalability issues, including some NP-complete (essentially intractable) problems. But some of the engineers would run around excitedly saying "it's the traveling salesman problem!" every time they saw a problem that couldn't be solved in linear time, and in other cases we boasted of "linear-time" algorithms which probably meant linear-time most-of-the-time. Or some of the time.

7.2. Free Beer, Free Speech, Free Software

OK, there's no free beer, but this is a good time to be a programmer (well, recession and outsourcing controversy notwithstanding) - just about everything you need is on the Internet tutorials, discussion lists, and free software. All you need is the hardware and a broadband connection.

8. R-E-S-P-E-C-T

One requirement for being an effective software engineer is to be taken seriously. You need to have the respect of your peers and managers, at least for your technical capabilities, to have control over your own work and influence over others.

8.1. There is Such a Thing as a Stupid Question

Really, there are lots of stupid questions. Asking intelligent questions that enhance others' respect for you is a professional skill. A good question that exposes unresolved issues tells people that you know your stuff and you're sharp enough to catch all the implications. Asking for clarification about a specification shows you know how to find and read the spec and your ability to detect ambiguities.

If you don't get any answers to your question, chances are there's something wrong with the question, so don't just repeat it. Ask the question differently, with more specifics, or more background. If you've been on the other end of a technical support line or even just spent time on discussion lists answering newbie questions, you'll appreciate the consideration.

- I pride myself on cultivating good relations with developer support staff by submitting elaborate bug reports and precise questions. But I do remember one lapse where I tossed out something along the lines of "What's the deal with that issue that came up several weeks ago?" You can imagine the prickly response - "What do you mean by what's the deal, and what issue are you talking about?"

It doesn't pay to be rude, especially if you're essentially asking for free tutoring or consulting on a discussion list. Even if you're asking under the auspices of a support contract, irritating your support contact isn't going to help you in the long term.

- I used to take pains to explain to belligerent newbies why their questions didn't make sense or what they were fundamentally doing wrong. Now, the bozo filter kicks in quickly - one "All I want to know is....", and they're ignored.

Let everyone know that you read the documentation and googled the subject. Besides avoiding the inevitable "RTFM" and "Google is your friend" responses, this shows you've done your homework and those who want to be of assistance don't have to search through the same resources. If you do expect them to search through those resources for you, then you're saying your time is more important than theirs, and you are just one more perpetrator of the "tragedy of the commons".

8.2. There is Such a Thing as a Stupid Answer

If you're going to act like you know what you're talking about, you really better know what you're talking about. Engineers sometimes communicate more to show off their own knowledge rather than to inform (although, if you can do both, kudos to you). This is often inflicted in employment interviews, under the guise of "finding out how you think" the candidate is asked inane puzzle questions. This can backfire, though, if the candidate has any self respect.

- One CTO interviewed me over the phone by asking me to sketch out the resulting stack frame from a C++ compilation and then report the result back to him verbally. We went through it step by step and every time I gave him a correct answer he asked me to give a wrong answer instead so we could go over why that choice wouldn't work. I couldn't tell if we were trying to demonstrate how smart I was or how smart he was.

There's also the blame game. As an engineer, you can't rely on your money and looks - all you've got is your credibility. So if you make a mistake - 'fess up.

- I had the privilege of working with a senior engineer who was never wrong. When his Java code was crashing on multiprocessor systems, it was Sun's bug. When I took over the code and pointed out the UI code was not supposed to run in multiple threads, he insisted there was only one thread. When I listed the seven threads (that I could find) in the code, he agreed I shouldn't have all those threads and I'd better fix it. He programmed in that fashion too - he didn't fix any bugs, he just covered them up with more code.

Finally, a bit of time-saving advice: Don't get dragged into stupid arguments. Stupidity is contagious.