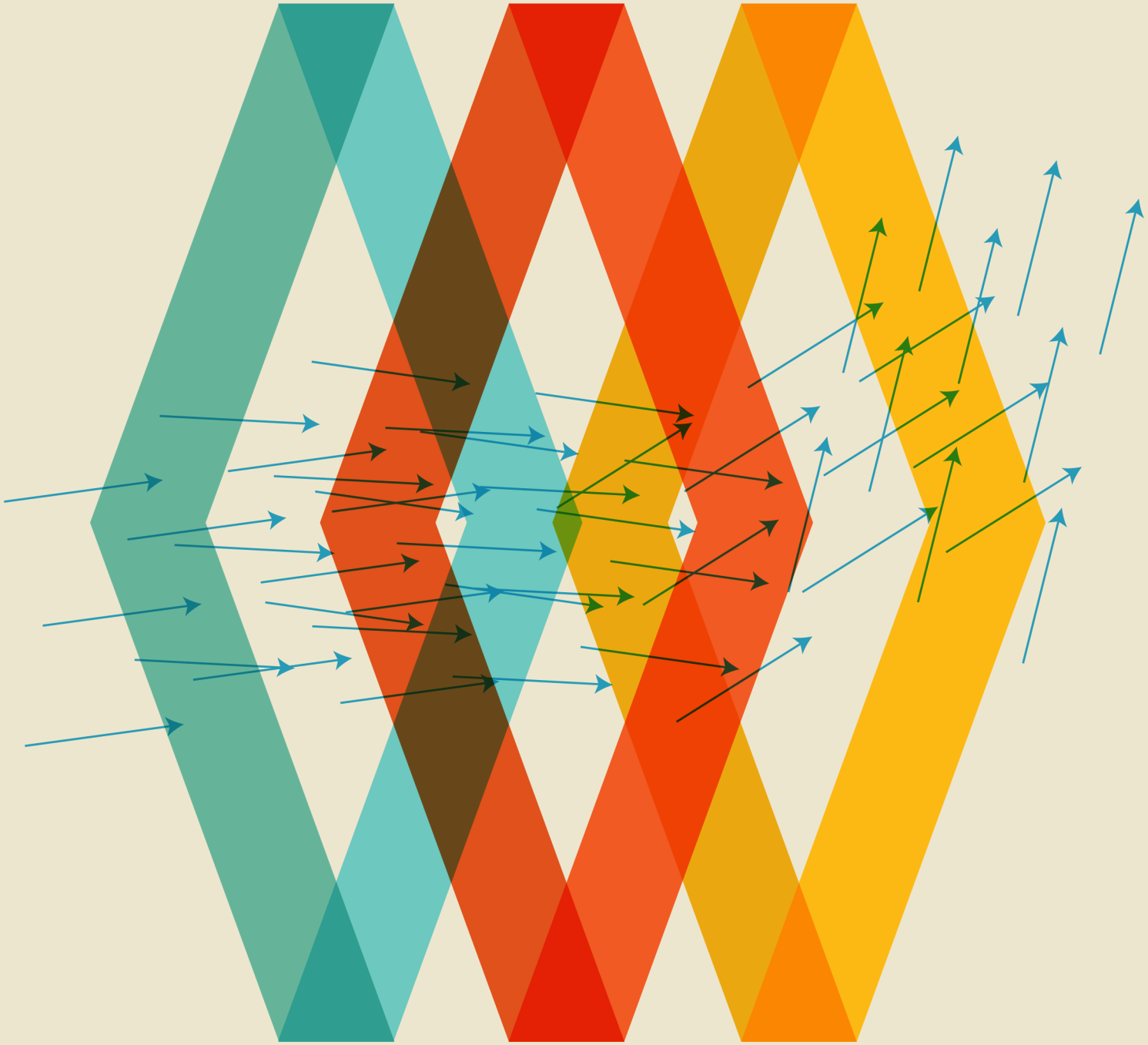
APPropriate Behaviour



Graham Lee

APPropriate Behaviour

Graham Lee

This book is for sale at http://leanpub.com/appropriatebehaviour

This version was published on 2019-10-11

Tweet This Book!

Please help Graham Lee by spreading the word about this book on Twitter!

The suggested hashtag for this book is #appbehaviour.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#appbehaviour

Contents

Learning

When you started doing this stuff, whether "this stuff" is writing iPhone apps, UNIX minicomputer software or whatever future programming you meals-in-pill-form types get up to, you didn't know how to do it. You had to learn. Maybe you took a training course, or a computer science degree. Perhaps you read a book or couple. However you did it, you started with no information and ended with…some.

It doesn't stop there. As Lewis Carroll said:

> It takes all the running you can do, to keep in the same place.

He was talking about the Red Queen's race, but I'm talking about learning and personal development. If you stopped when you had read that first book, you might have been *OK* as beginner programmers go, but if the woman next to you in the library read another book then she would have been a step ahead.

We live in what is often called a knowledge economy. Francis Bacon said "knowledge is power". If you're not learning, and improving yourself based on the things you learn, then you're falling behind the people who are. Your education is like the race of the Red Queen, constantly running to keep in the same place.

Do As Much As You Can

There's no such thing as too much learning (though the real problem of "not enough working" can sometimes be found in proximity to a *lot* of learning). Not all education comes through formal settings like training or university courses[1]. Reading a book, magazine article or blog post in your lunch break can be very helpful, as can going to birds-of-a-feather developer meetings.

Bigger items like training courses and conferences obviously involve a larger time commitment. There is, obviously, such a thing as "not enough working", and that's a balance you'll need to address. If you're self-employed then you need to balance the opportunity cost (how much work are you turning down by attending the course?) and financial cost against the benefits (how much better will you be after taking the course? how much extra work will you be able to get? what good contacts will you meet at the conference?).

Of course, if you're employed this decision may be taken for you by your manager. You can help the decision along if you know how the training course fits with the company's direction…but I'll leave that to the chapters on teamwork and business.

[1] Indeed much material from University-level computer science programs is now available for free through schemes like iTunes U and Coursera. That can make for some interesting lunchtime reading, but I find I learn better when I've got the structure of a taught course and the pressure of a submission deadline. That said, you're not me and you might benefit from a more relaxed learning environment.

Don’t Stick to Your Own Discipline

Every field has its champions and superheroes: the people with tens of thousands of followers, whose blog posts are always read and quoted and who speak at all the conferences. People look to these champions to analyse and direct the way their community works. Often the leaders in one field will be contrasted with “them”, the leaders in a different field: that iPhone programmer is one of “us”, and the Android programmer giving a talk in the other room is talking to “them”.

This definition of “us” and “them” is meaningless. It needs to be, in order to remain fluid enough that a new “them” can always be found. Looking through my little corner of history, I can see a few distinctions that have come and gone over time. Cocoa vs Carbon. CodeWarrior vs Project Builder. Mach-O vs CFM. iPhone vs Android. Windows vs Mac. UNIX vs VMS. BSD vs System V. SuSE vs Red Hat. RPM vs dpkg. KDE vs GNOME. Java vs Objective-C. Browser vs native. BitKeeper vs Monotone. Dots vs brackets.

Sometimes it takes an idea from a different field to give you a fresh perspective on your own work. As an example, I’ve found lots of new ideas on writing Object-Oriented code by listening to people in the Functional programming community. You might find that the converse it true, or that you can find new ways to write Java code by listening to some C# programmers.

You could even find that leaving the programmers behind altogether for a bit and doing some learning in another field inspires you - or at least lets you relax and come back to the coding afresh later. The point is that if you focus on your narrow discipline to the exclusion of all others, you’ll end up excluding a lot of clever people and ideas from your experience.

Put it into Practice

At various points in history I’ve learnt a collection of languages, of the inter-human and computer programming varieties. The only ones I can remember anything about are the ones I use all the time.

I expect the same’s true for you. The reason I expect this is not that I believe everyone’s like me, but that there’s basis for it in theory. The Kolb learning cycle[2] says that there are four processes that form the practice of learning:

- Concrete Experience: actually doing a thing.
- Reflective Observation: analysing how you (or someone else) did a thing.
- Abstract Conceptualisation: building a model of how things should be done.
- Active Experimentation: just playing with the plasticine and seeing what comes out.

Not everybody goes through all of the items in the cycle, but most people start out somewhere and progress through at least a couple of the points, probably in the order presented (acknowledging that as a cycle it should be, well, cyclic). Therefore almost everyone who learns something goes through

[2] http://www.businessballs.com/kolblearningstyles.htm

either experimentation or building experience: it's very hard to learn something without trying it out.

Perhaps more importantly, it's hard to adapt what you've learned to fit everything else you do if you don't try it out. An idea on its own isn't really doing anything useful; when it's put into practice it becomes combined with other ideas and techniques and adds something valuable.

Collaborate and Share what you Learn

There are numerous benefits to sharing the things that you learn. The first is that everybody you share with will have had different experiences, and can tell you how what you've learned applies (or doesn't) in their context. That insight can give you a more complete picture of what you learned, especially on where it might be limited. Conference talks and books are often delivered with a spin on being persuasive—not because the author is being disingenuous, but because the material will be more successful if you go away wanting to apply what you've learned.

Listening to other people who've found that what you want to do does (or doesn't) work in particular situations, then, can give you a more complete picture of a concept and its applications than just relying on the first source you discovered. In return, you'll probably tell the person you're talking to about *your* experiences and problems, so you both get to learn.

That's the real reason I'm keen on shared learning—everyone benefits. That includes the teacher, if you're collaborating in a formal learning environment such as a training course or a class. Even if you've got a lot less experience than the teacher, you'll have unique insight and ideas that are more useful out in the open than kept quiet.

Publications like *Communications of the ACM* frequently cover problems associated with teaching computing. Indeed, in the issue that was current at time of writing, two[3] articles[4] discuss a shortage of computer science teaching. I believe that to address such problems, we need to get feedback not from experts (who managed to make it through the initial learning phase, no matter how shoddy the resources available) but from neophytes. We need to get more feedback on what's currently making it hard for beginners to make progress, if we're to scale the industry and allow new colleagues to quickly get to the point of doing better than we do.

Of course, listening to newbies will work best if the newbies are talking to us; specifically, telling us what's going well and what's going wrong. A great way to encourage that would be to lead by example. Unfortunately it doesn't seem like this is popular. In the world of Objective-C programming, two great aggregators of blog content are the Cocoa Literature List[5] and iOS Dev Weekly[6]. Maybe I'm just getting jaded, but it seems like a lot of the content on both of these sites comprises tutorials and guides. These either rehash topics covered in the first-party documentation, or demonstrate some wrapper class the author has created without going into much depth on the tribulations in getting there.

[3] http://cacm.acm.org/magazines/2012/11/156579-learning-to-teach-computer-science/fulltext
[4] http://cacm.acm.org/blogs/blog-cacm/156531-why-isnt-there-more-computer-science-in-us-high-schools/fulltext
[5] http://cocoalit.com
[6] http://iosdevweekly.com/issues/

What we really need to understand, from neophytes and experienced developers alike, is actually closer to the content of Stack Overflow[7] than the content of the blogosphere. If lots of inexperienced programmers are having trouble working out how two objects communicate (and plenty do[8]) then maybe OOP isn't an appropriate paradigm for people new to programming; or perhaps the way that it's taught needs changing.

So this is a bit of a request for people who want to improve the field of programming to mine Stack Overflow and related sites to find out what the common problems are—trying to decide the experience level of any individual user can be difficult so organising problems into "newbie problem" versus "expert problem" will be difficult. It's also a request for people who are having trouble to post more Stack Overflow questions. The reasons?

- usually, in the process of crafting a good question, you end up working out what the answer is anyway. The effort isn't wasted on Stack Overflow; you can answer your own question when you post it then everyone can see the problem and how you solved it.
- the reputation system (to a first approximation) rewards good questions and answers, so the chance that you'll get a useful answer to the question is high.
- such questions and answers can then be mined as discussed above.

There are downsides, of course:

- Duplicate questions cannot easily be measured, because they're usually closed and often deleted. Or people will find existing questions that cover the same ground (as they're supposed to, within the "rules") and not ask their duplicate. The voting system and view count have to be used as proxies to the "popularity" of a question; an inexact system.
- The voting system tends to reward received dogma over novel ideas or technical accuracy; upvoted answers are "popular", which is not the same as being "correct".

A better system for teaching programming would base its content on the total collection of all feedback received by instructors at programming classes ever. But we're unlikely to get that. In the meantime, Stack Overflow's pretty good. What I'm saying is that you shouldn't just share what you learn, you should share what you're stuck on too.

Opportunities to Learn

So your training budget's used up, the conference you like was last month and won't be around for another year; is that it? When else are you going to get a chance to get yourself into the learning frame of mind?

All the time. Here are a couple of examples of how I squeeze a little extra study into life:

[7] http://www.stackoverflow.com
[8] http://stackoverflow.com/questions/6494055/set-object-in-another-class

- I drive about an hour each way on my commute. That's two podcast episodes per day, ten per week.
- Once a week, my developer team has "code club", an hour long meeting in which one member makes a presentation or leads a discussion. Everybody else is invited to ask questions or share their experiences.
- There's a little time at lunch to read some articles.
- I go to one or two local developer groups a month.

You don't necessarily need to deep-dive on some information in order to make use of it. Just knowing that it's out there and that you can find it again is enough to give it a space in your mental pigeonhole. When you've got a related problem in the future, you'll likely remember that you read about it in *this* article, or made *that* note in Evernote. Then you can go back and find the actual data you need.

Of course, conferences and training courses *are* great places to learn a lot. One reason is that you can (to some extent, anyway) put aside everything else and concentrate on what's being delivered.

Ranty aside

One of the saddest things to see at a conference is someone who's doing some work on their laptop instead of focussing on the session. They're missing out—not just on the content, but on the shared experience to talk about with other delegates in the next break. It's not a good environment to work in because of the noise and the projected images, and they don't get anything out of the sessions either.

Rediscovering lost knowledge

You might think that with software being such a fast-moving field, everything we're doing now is based on everything we were doing last year, with induction proving that there's a continuous unbroken history linking current practice to the "ENIAC girls" and Colossus wrens of the 1940s. In fact the truth is pretty much the exact opposite of that; practices seen as out of date are just as likely to be rejected and forgotten as to be synthesised into modern practice.

As an example, I present my own experience with programming. I was born into the microcomputer revolution, and the first generation of home computers. Programming taught on these machines was based on either the BASIC language of the 1960s or using assemblers. The advances coming from structured programming, object-oriented programming, procedural programming or functional programming were all either ignored or thought of as advanced topics inappropriate to micro programming. It wasn't until much later that I was introduced to "new" concepts such as 1973's C, and had to come to grips with any form of code organisation or modularity.

Armed with a history book, or a collection of contemporary literature, on computer programming, it's easy to see that I'm not alone in ignoring or losing earlier results in the discipline. After all, what is agile programming's "self-organising team" but a reinvention of Weinberg's adaptive

programming[9]? Is there are clear lineage, or has the concept been reinvented? Is the "new" field of UX really so different from the "Human-relations aspects" of Boehm's software engineering economics[10]? As described in the documentation chapter, many developers no longer use UML; how long until UML is invented to replace it?

The teaching of software creation

My mitigation for the rediscovery problem outlined above could be that you undertake the heroic effort of discovering what's out there from nearly 70 years of literature, identify the relevant parts and synthesise a view on software making from that. That would be crazy. But in the short term, that's probably the only route available.

Like many people, I learned programming by experimentation, and by studying books and magazines of varying quality. This means that like many programmers my formative experiences were not guided (or tainted, depending on your position) by a consistent theory of the pedagogy of programming. Indeed, I don't think that one exists. Programming is taught differently by professional trainers and by university departments; indeed it's taught differently by different departments in the same university (as I discovered when I was teaching it in one of them). There's no consistent body of knowledge that's applied or even referred to, and different courses will teach very different things. I'm not talking about differences at the idiomatic level, which are true across all of teaching; you could learn the same programming language from two different teachers and discover two disjoint sets of concepts.

This is consistent with the idea of programming being merely a tool to solve problems; different courses will be written with solving different problems in mind. But it means there isn't a shared collection of experiences and knowledge among neophyte programmers; we're doomed to spend the first few years of our careers repeating everyone else's mistakes.

Unfortunately I don't have a quick solution to this: all I can do is make you aware that there's likely to be *loads* of experience in the industry that you haven't even been able to make secondary use of. The effort to which you go to discover, understand and share this experience is up to you, but hopefully this chapter has convinced you that the more you share knowledge with the community, the better your work and that of the community as a whole will be.

The particular material I learned from was long on descriptions of how operators work and how to use the keywords of the language, but short on organisation, on planning, on readability[11]; i.e. on everything that's beyond writing code and into writing *usable* code. Yes I learned how to use GOSUB, but not *when* to use GOSUB.

There's a lot of good material out there on these other aspects of coding. When it comes to organisation, for example, even back when I was teaching myself programming there were books out there that explained this stuff and made a good job of it: The Structure and Interpretation of

[9] http://dl.acm.org/citation.cfm?id=61465
[10] http://userfs.cec.wustl.edu/~cse528/Boehm-SE-Economics.pdf
[11] There's an essay on what it means for code to be readable in the chapter on critical analysis.

Computer Programs[12]; Object-Oriented Programming: an evolutionary approach[13]; Object-Oriented Software Construction[14]. The problem then was not that the information did not exist, but that I did not know I needed to learn it. It was, if you like, an Unknown Unknown.

You could argue that organisation of code is an intermediate or advanced topic, beyond the scope of an introductory book or training course. Or you could argue that while it *is* something a beginner should know, putting it in the same book as the "this is how you use the + operator" material would make things look overwhelmingly complex, and could put people off.

Firstly, let me put forward the position that neither of these is true. I argue from analogy with Roger Penrose's book The Road to Reality[15], which starts from fundamental maths (Pythagoras's theory, geometry and so on) and ends up at quantum gravity and cosmology. Each chapter is challenging, more so than the previous one, but can be understood given an understanding of what came before. People (myself included) have been known to spend years working through the book, working through the exercises at the end of each chapter before starting the next. And yet it's a single book, barely more than 1100 pages long.

Could the same be done for computing? Could a "The Road to Virtual Reality" take people from an introduction to programming to a comprehensive overview of software making? I'll say this; the field is *much* smaller than theoretical physics.

Now here's a different argument. I'll accept the idea that the field is either too big or too complex to all go into a single place, even for a strongly motivated learner. What's needed in this case is a curriculum: a guide to how the different parts of software making are related, which build on the others and a proposed order in which to learn them.

Such curricula exist, of course. In the UK, A-level computing[16] doesn't just teach programming, but how to identify a problem that can be solved by a computer, design and build that solution and document it. Now where do you go from there? Being able to estimate the cost and risk associated with building the solution would be helpful; working on solutions built by more than one person; maintaining existing software; testing the proposed solution…these are all things that build on the presented topics. They're covered by Postgraduate courses in software engineering[17]; there's some kind of gap in between learning how to program and improving as a professional programmer where you're on your own.

And these curricula are only designed for taught courses. Need the self-taught programmer be left out?[18] There are numerous series of books on programming; the Kent Beck signature series[19] on

[12] http://mitpress.mit.edu/sicp/full-text/book/book.html
[13] http://books.google.co.uk/books/about/Object_oriented_programming.html?id=U8AgAQAAIAAJ&redir_esc=y
[14] http://docs.eiffel.com/book/method/object-oriented-software-construction-2nd-edition
[15] http://books.google.co.uk/books/about/The_Road_to_Reality.html?id=ykV8cZxZ80MC
[16] http://www.cie.org.uk/qualifications/academic/uppersec/alevel/subject?assdef_id=738
[17] http://www.cs.ox.ac.uk/softeng/courses/subjects.html
[18] Some in the field would say yes, that programming should be a professional discipline open only to professionals—or at least that there should be a designated *title* available only to those in the know, in the way that anybody can be a nutritionist but only the qualified may call themselves dieticians. Some of these people call themselves "software engineers" and think that software should be an exclusive profession like an engineering discipline, others call themselves "software craftsmen" and use the mediaeval trade guilds as their models for exclusivity. I will leave my appraisal of these positions for later, but for now it's worth reflecting on the implicit baggage that comes with *any* description of our work.
[19] http://www.informit.com/imprint/series_detail.aspx?ser=2175138

management methodologies and approaches to testing, for example, or the Spring Into[20] series of short introductions.

These published series are often clustered around either the beginner level, or are deep and focus on experienced developers looking for information on specific tasks. There's no clear route from one to the other, whether editorially curated by some publisher or as an external resource. Try a web search for "what programming books to read" and you'll get more than one result for every programmer who has opined on the topic—as Jeff Atwood has written on it more than once.

Building a curriculum is hard. Harder than building a list of books you've read and you'd like to pretend you'd read, then telling people they can't be a programmer until they read them. You need to decide what's really relevant, and what to leave aside. You need to work out whether different material fits with a consistent theory of learning; whether people who get value from one book would derive anything from another. You need to decide where people need to get more experience, need to try things out before proceeding: and how appropriate it is for their curriculum to tell them to do that. You need to accept that different people learn in different ways, and be ready for the fact that your curriculum won't work for everyone.

What all of this means is that there is still, despite 45 years of systematic computer science education, room for *multiple* curricula on the teaching of making software. That the possibility to help the next generation of programmers avoid the minefields that we (and the people before us, and the people before them) blundered into is open. That the "heroic effort" of rediscovery described at the beginning of this section need be done, but only a small number of times.

Reflective Learning

Many higher education institutions promote the concept of reflective learning[21]; analysing what you're learning introspectively and restrospectively, deciding what's gone well and what hasn't, and planning changes to favour the good parts over the bad. Bearing in mind what we've seen in this chapter, that there are manifold sources of information and that different people learn well from different media, reflective learning is a good way to sort through all of this information and decide what works for you.

This is far from being a novel idea. In his book The Psychology of Computer Programming[22], Gerald M. Weinberg describes how some programmers will learn well from lectures, some from books, some from audio recordings. Some will—as we saw when discussing the Kolb cycle—want to start out with experimentation, whereas others will want to start with the theory. As he tells us to try these things out and discover which we benefit from most, he's telling us to *reflect* on our learning experiences and use that reflection to improve those experiences.

Reflective learning is also a good way to derive lessons from your everyday experiences. I have a small notebook here in which, about four years ago, I wrote a paragraph every day based on the

[20] http://www.informit.com/imprint/series_detail.aspx?st=61172
[21] http://www.heacademy.ac.uk/hlst/resources/a-zdirectory/reflectivelearning
[22] http://www.geraldmweinberg.com/Site/Programming_Psychology.html

work I did that day. I thought about the problems I'd seen, and whether I could do anything to address them: I also thought about what had gone well and whether I could derive anything general from those successes. Here's an example entry.

> Delegated review of our code inspection process to [colleague]. Did I give him enough information, and explain why I gave him the task? Discovered a common problem in code I write, there have been multiple crashes due to inserting `nil` into a collection. In much ObjC, the `nil` object can be used as normal but not in collections, and I already knew this. Why do I miss this out when writing code? Concentrate on ensuring failure cond^ns are handled in future code, & get help to see them in code reviews. Chasing a problem with [product] which turned out to be something I'd already fixed on trunk & hadn't integrated into my work branch. What could I have done to identify that earlier? Frequent integrations of fixes from trunk onto my branch would have obviated the issue.

You don't necessarily have to write your reflections down, although I find that keeping a journal or a blog does make me structure my thoughts more than entirely internal reflection does. In a way, this very book is a reflective learning exercise for me. I'm thinking about what I've had to do in my programming life that isn't directly about writing code, and documenting that. Along the way I'm deciding that some things warrant further investigation, discovering more about them, and writing about those discoveries.