Chapter 2
The Happy Hacker, Love at First Byte

King Kong was only 18 inches tall – in movie making everything is possible.
(Peter O’Toole in The Director)

The Hollywood school of unreality can create whatever is required. It devises stunts such that whatever is required seems to be made manifest. It’s the great illusion fantasy, and we love it. Similarly, computer technology (which, incidentally, has majestically pushed back the limits of fantasy made real in the film industry) is virtually unconstrained. It is almost free of the millstones of reality, and this is no small part of the allure. In programming everything is possible.¹ This is as true for the programmer as it is for the Hollywood director, which is to say that it’s almost true. Where it is not true turns out to be crucial, as we shall see when it comes to issuing guarantees that the programmer’s creations are correct.

In this chapter I shall explain how and why it is that certain specimens of humanity, who otherwise seem fairly normal and come from good homes, get hooked on computer programming. The problem certainly seems to occur more (much more) in the male than the female of the species, and it also strikes the young more frequently than the, let us say, mature individuals. Males in the years surrounding puberty seem to be definitely the most susceptible which is not at all surprising where issues of infatuation are concerned.

As I said, even the most level-headed can be struck down by the happy-hacker syndrome (henceforth HHS).² Indeed, I myself am proof of this. Yes, I have to admit it, I was a victim. It struck me in the early twenties, but only missed me earlier because in those dim and distant days home computers in the UK were no more than a gleam in Sir Clive Sinclair’s spectacles. It was at University that I first gained access to computer technology. Even then my exposure was very limited, and quite peripheral to my main reason for being there which was to obtain a degree in chemistry.

My chemistry degree course was three long years, and by the end of the first I had grave doubts about chemistry being the life for me. But after another year my doubts had vanished, in their stead was a firm conviction that I was not cut out to be chemist. It’s hard to put a finger on what exactly it was that I did not like about chemistry. Certainly, the endless washing up held no attraction for me (an inherently distasteful
task at the best of times). This coupled with the olfactory issue, at times a supreme miasmic event which had to be experienced to be believed, was a definite negative. Day in, day out sloshing about with dangerous substances was probably the clincher. It was not the known explosives and poisons that were worrying so much as the ones whose life-threatening capabilities were still being uncovered.

Benzene is a case in point: for a year or more we used it as a substitute for Fairy Liquid – it easily dissolves away incrustations that treat soap and water with utter disdain, it evaporates quickly (which eliminates the drying-up chore), and it is relatively cheap. Then one day, while up to my elbows in the stuff and at the centre of a pall of benzene vapour, a freshly printed directive from the Prof arrived. The gist of it was that benzene was poisonous, carcinogenic, etc. and all benzene-related activities were thus to be abandoned forthwith.

You can perhaps imagine that to a young, red-blooded person, in their prime, and looking confidently forward to a lot of prime-time activity in the next few years, such happenings were a very real disincentive. However, with only one more year to go, I decided to stick it out to the end, but to avoid chemicals as much as possible.

It turned out that in the third and final year the student was allowed to specialize, to choose a particular branch of chemistry as the area for a substantial project. For me there was no choice, theoretical chemistry just had to be my area of specialization. Theoretical chemistry is effectively a branch of mathematics whereas the other areas of specialization all involved that which I had forsworn – i.e., use of chemicals. In theoretical chemistry all that is required is a pencil, paper and an armchair, although a liking for calculation (not to mention an ability to do it) is also a help.

The theoretical chemist finds fulfilment calculating energies of molecules and of chemical reactions. This sort of mathematics typically requires the use of quantum mechanics (which we will not be delving into, rest assured). Quantum-mechanical calculations involve a lot of the usual messing about with numbers – i.e., additions, multiplications, etc. They also involve integration which is a particularly tricky species of mathematical manipulation invented by Sir Isaac Newton during his Easter holidays.

One of the awkward features of integration is that we don’t know how to do it exactly on many elaborate mathematical structures, but there are exhausting ways to get it approximately correct, as approximately correct as you wish (time and energy permitting). The upshot of all this is that, although a pencil and paper is all that is required in principle, human life is too short to take the in-principle route, and the few months available to me was hardly time to get started. So in practice, computers must be used which, as we all know, are just fast and accurate adding machines. The sort of number crunching exercise that my quantum-mechanical calculations involved was just the right stuff for computerization.

Consequently, I had to forego my Easter hols (much like Sir Isaac, although I don’t know who was twisting his arm and forcing him to invent the differential calculus) and attend a programming course. Needless to say, I was not delighted by this intrusion of the educational world on my Easter break. But given that it would, in effect, permit me to avoid chemicals for the remainder of my degree course, I was prepared to make the sacrifice.
The computer programming language on offer was called FORTRAN, FORTRAN II. Much to my surprise I liked FORTRAN. In fact, within a week I loved it. Now, as a professional computer scientist, this is an embarrassing admission.

"Why such a speedy emotional entrapment?" you might well ask, puzzled by this almost instantaneous infatuation that I have confessed to. I now ask myself the same question, I didn’t at the time, of course. Whilst being tumbled along in the midst of a growing romantic attachment one never does, does one? But now, with some few decades and many thousands of lines of FORTRAN between me and that first close encounter of the computational kind, I think I can explain everything.

Fundamentally, it is a question of power, power and the satisfaction of the ‘look ma, no hands’ phenomenon. A final significant factor is the personal, and at the same time non-personal, nature of human-computer interaction. It’s just you and the machine trying to work things out, the rest of humanity is effectively excluded, prevented from intruding.

This last point raises a complex issue, one more in the realm of the sociologist than the computer scientist. So I’ll say no more about it for the moment, except to note that it may be at the root of why males are readily afflicted with HHS while females appear almost immune. When you have gained a better appreciation of the technical details that underlie this liaison dangereux, we can usefully begin to explore the more nebulous psychological aspects of the problem. There is, I will argue, a significant macho element to software engineering disasters.

Having constructed a good program, we are left with the feelings of power and satisfaction that a successful computer program engenders in the programmer. The notion of power comes from having an exceedingly complex and intricate machine do your bidding. For the reader who views computers as normal home appliances like electric kettles, I might point out that back in the dark ages when I got hooked my university possessed one computer* for the use of everyone. So getting control of this beast – a large room full of electronics – may well have been more satisfying to the closet megalomaniac than successfully instructing a modern laptop, but the same principle still holds: the successful programmer has gained control of an intricate and powerful machine. (But note: today most mobile phones4 are more powerful computers than my room full of whirring machinery – card reader, card punch, tape readers both paper and magnetic, crude printer, massive disc storage device, air conditioning and, oh yes, the central processing unit.)

One can quite easily make a computer number crunch or data shuffle for long periods churning out vast quantities of results with very little programming effort. In fact, it is all too easy to make a computer repeat the same series of computations endlessly (endlessly until someone pulls the plug that is). This is actually one of the problems with this technology, and we will revisit it in due course.

---

*Definitional use of special terms, which is also usually the first mention, are italicized and underlined; they are also Glossary entries. Subsequent use is also italicized if there is a need to emphasize the special meaning used in this book for a word or term that has a different everyday meaning. Because I use bold for programming language keywords, I am left with underline for ordinary emphasis, and occasional foreign words are also italicized.
The great satisfaction that programming can provide stems from the stringent demands of most programming languages in conjunction with the absence of limitations on what might be achieved – in programming it seems as though everything is possible if you, the programmer, are smart enough. What testosterone filled young man is going to throw in the keyboard and admit that they’re not smart enough – especially when he can quickly and quietly try whatever comes into his head?

The obsessive concern for detailed correctness exhibited by most computers means that significant programs never work first time. After hours, maybe days of pondering the programmer submits his best effort to the computer which returns the program virtually instantaneously together with a whole catalogue of errors it has found; the programmer (if he wants to remain a programmer) has no option but to think again. This to-and-fro process can continue for very long periods until eventually the programmer gives up, or the computer accepts the program as grammatically correct and performs the computations that it specifies.

Suddenly, after repeated rejections, the behaviour that your program has specified emerges from the computer (although often not exactly the behaviour that was expected). However, the computer is finally doing your bidding, and the resultant feeling of satisfaction and power is not surprising. Actually, the computer is, of course, doing your program’s bidding, but that’s a start. The challenge then is to make yours and your program’s biddings identical.

By sheer persistence and intelligent reasoning (the relative mix of these two quantities can vary considerably) the programmer succeeds in compelling the hitherto superior machine to obey his instructions. You, the programmer, start out as the servant spending long periods laboriously sorting out all the errors so quickly and surely spotted by the computer, but once the last error has been corrected, the tables are turned – the computer has no choice but to compute for as long and as tediously as your program specifies. An acceptable program (acceptable to the computer that is) is a powerful object, a talisman of the technological age, and the programmer is (of course) the witchdoctor.

In short, programming is a game. It is a private game. You are free to blunder at the keyboard, no one will laugh at you, and when you succeed they will be mightily impressed. Programming is fun, very private fun. There’s no one to ridicule your more feeble efforts. All mistakes and blunders can be kept strictly between you and the machine which doesn’t, of course, exhibit any judgemental behaviour. So that’s how I, and many thousands of others, got hooked, although I wasn’t affected by the introspective allure of the exercise, of course. But then, in the memorable phraseology of Mandy Rice-Davies: I would say that wouldn’t I?

‘Game’ is actually too broad a word for the particular phenomenon that we are considering. It is misleading because it suggests the presence of attendant qualities like luck and chance which don’t really come into the programming game. One can, it is true, get something right in a program quite by chance, and good luck never comes amiss in life, but serendipity in any of its manifestations has no real part to play in the game of programming.

This state of affairs is unusual in the gaming world. Most games, it seems, include a good portion of luck, and people presumably want it so. It provides an
element of unpredictability to game outcomes. However good the player, there is no guarantee that they will win. Chess is perhaps the popular game that comes closest to being devoid of fortuitousness. It is also a complex game that computers have come close to mastering – anybody but a real chess expert will lose to the best chess-playing computers.\(^5\) I’m not at all sure that there is any causal relationship between the chanceless nature of chess and the high quality of chess-playing computers, but there just might be. The allure of chess and that of programming have a number of features in common.

Those of you who have either dabbled with a home computer or have children or partners that do so will have noted the hours that can be consumed locked with this machine in a struggle for supremacy. It might be Dungeons and Dragons, a war game, surfing the Internet or whatever, but computer games often rival the popularity of television as a face-to-screen activity in the modern world. Television is easier, you just sit back and take what comes more or less, the brain can be put on idle for the duration. But when facing a computer screen considerable high-geared brain-work is absolutely essential.

Computer games can constitute a considerable mental challenge, and they offer you the opportunity to influence events. Success demands concentration and commitment but many are prepared to give it, and for extraordinarily long periods of time. Programming can be viewed as the next step along, a further enrichment of person-machine interaction.

Computer games, however elaborate, contain inherent limitations.\(^6\) The avid computer gamer soon hits up against some of these limitations, and needs to move on. The onward move is often to a new game, but the significant step forward, the one that opens up a whole new and virtually limitless challenge, is the move to become a programmer. Computer games are programs. The scope and limitations built into any particular game are, in a general sense, imposed by speed and memory limitations of particular computers but how these general constraints become manifest in the game depends upon the programmer. To a significant degree, the limitations and opportunities within any computer game are determined by the time, skill and imagination of the person(s) who wrote the program. So, when the excitement of computer gaming begins to pale, the solution is to become a programmer and invent your own computer games. The future then is determined by you, and its limitations are yours too.

The lack of limitations is much the same as the lack of limitations in writing a murder mystery: there are certain fixed points, like someone has to be murdered (or appear to be), but there are infinitely many ways to actually write the story. The real limitations are determined only by the limits of the writer’s imagination.

A peculiarity of the programming game is that the rules are open and loose at a general level, but they are tight, fixed (and often quite pedantic) at the detailed level – i.e., the rules about what constructions the computer system will actually accept and act upon.

The general framework of the programming game is:

[a.] spend considerable time writing a program  
[b.] give it to the computer
[c.] the computer immediately rejects it indicating numerous errors
[d.] spend time sorting out the errors, and modify the program
[e.] go back to step [b.]

This series of five steps may not strike you as an obviously fun thing to do, but you should be able appreciate that it is a challenge. Fun, after all, exists only in the brain of the player. It is not an inherent property of certain activities. In this game the human intellect is being pitted against a sophisticated electronic machine, an incredibly fast and accurate machine. There is, however, no laughing at the inevitable human errors and blunders. There is no peer pressure. All your weaknesses and inadequacies are just between you and this machine which is lightning fast, unerringly correct and totally committed to the contest, but in an entirely unemotional way. The rest of humanity is excluded from the duel; the rest of the world might well not exist.

What may strike you about my list of five steps (if you’re really paying the proper attention) is that it describes an endless procedure. I have in fact omitted some crucial details. Step [c.] should really describe alternative outcomes: the rejection given above, and an acceptance of your program as a grammatically correct one. When your program gets the green light rather than the usual red one, as it were, you suddenly become the driver of the machine. The computer has no alternative; it must now do exactly what your program tells it to do (which is usually not quite the same as what you thought you had told it to do). At this point you’ve won (but just a battle, not the war). The human intellect comes out on top just as it should. Man triumphs over machine yet again. With this victory securely in your pocket, you are free to move on and fix up the program to get precisely the behaviour you intended, or maybe elaborate on the current program to try to make the machine do something just a bit more demanding, or something totally different. And the game goes on.

In fact, the game has only just begun. Once the computer has accepted your program and starts to do what your program dictates, that’s when the real fight begins. The next battle is to get the emergent behaviour to be exactly what you want it to be – no more and no less. It is mostly this activity that eats up years of system-development time on large IT projects.7

How come? The program must be correct before the computer will accept it? This is true, but there are two very different levels of correctness to master. The first, is grammatical correctness – have you got the detailed structures of the program instructions correct? If yes, then the next level of correctness (the really tricky one) is: behavioural correctness, does this correctly structured program instruct the computer to do what you want it to do? Is the program behaving correctly?8

Computers operate like your worst nightmare of an English teacher. Suppose you text9 or email a friend with: “Is their room in your car for me. I need to know immediately?” and your friend replies: “Sorry, your question was grammatically incorrect, please correct it and try again.” You might well be miffed, but that’s how computers behave: they will make no attempt to interpret the meaning of your program until the program is grammatically correct – every comma and semicolon is precisely where it should be (and similar-sounding words must be spelt correctly).
I realize that my explanation of the lure of computers may be unconvincing to the non-susceptible and to those who have never risked being smitten. Like train spotting, bird watching and other forms of minor lunacy that various of us indulge in, if that’s your thing then the irresistibility is obvious, but if you’re immune to a specific affliction it is next to impossible to appreciate the nature of that particular compulsion.

A compulsion to practise and experiment with playing various pieces of music on, say, the piano might be a somewhat better analogy with the activities of those afflicted with the HHS. This musical madness is not primarily acquisitive like train spotting and the similar collecting obsessions that so many of us are prone to exhibit. Exploring the possibilities of a musical score is a much richer activity, and one whose reward is a good deal less tangible rather like programming, although there clearly are a number of significant differences between what the musician and the programmer aim to produce.

I don’t expect the hardened anti-computerists amongst you to be convinced by the foregoing, but I hope that if you tackle the subsequent chapter on what it means to program a computer you will begin, at least, to appreciate why some individuals do fall in love at first byte.

So what progress towards the necessary understanding have we made with this confessional chapter?

- To some, programming is an alluring man-machine tussle that ultimately promises to put great power in the hands of the man.
- The first challenge (the easy one) is that a program must be grammatically correct before a computer will accept it.
- But the (much) bigger challenge is getting an accepted program to cause the computer to behave in precisely the way you want it to behave; it is behavioural correctness that is the really big problem.

By looking quite closely into the nature of programs we will begin to develop an appreciation of the enormity of this latter challenge in the next chapter.

**Endnotes**

1. There are many things that we know that computers cannot do, and will never be able to do – see *COMPUTERS LTD What they really can’t do by D. Harel* (Oxford University Press, 2000) for a very readable explanation of the many fundamental limitations of modern computer technology. But these are the technical limitations on the size of various computational problems.

   “The great thing about software [i.e., a computer program] is that it allows a computer to do almost anything. Actually, this isn’t quite true. In reality, software allows a computer to do anything – almost.” So said David Lubar on page 20 of in his 1995 book “It’s Not a Bug, It’s a Feature: computer wit and wisdom” published by Addison-Wesley.

   In chess playing, for example, no computer will ever be able to compute all of the alternative moves and countermoves through to all win, lose or draw endpoints – there are just too many of them. So no computer (nor human, of course) will ever play perfect chess which is precisely what makes it an interesting game. Nevertheless, computers now play very high quality chess, and it is unclear how good they will eventually become, but we know they’ll never
be perfect. In this book we are concerned with the lack of similar clear limitations on the quality of solution that computers can be programmed to produce rather than the well-defined limitations on the size of certain problems.

The surprisingness of this size constraint is exemplified in the story of the Chinese Emperor who wanted to reward a local sage. “Give me some rice,” said the sage.

“Some rice!” The Emperor spluttered, “Not gold, or silver, or precious stones?”

“No. Just give me the rice that results from one grain on the first square of your chessboard, two on the second, four on the third, and continue doubling up through all 64 squares.”

“Well, the rice is yours,” declared the Emperor who did not know that he had just promised more rice than existed in the world.

2. I use the word “hacker” in its older, and (I think) more appropriate meaning, i.e., someone who is addicted to computer technology. In more recent times the word has been hijacked and forced to carry the meaning of someone who gains illegal access to computer systems which is, in my view, merely one of the diversions that the hacker indulges in to satisfy his craving. Steven Levy’s book Hackers: heroes of the computer revolution (Penguin, 1984), introduces many of these characters.

3. Mathematics is full of useful numbers that can only be calculated approximately. The square root of 4 is 2, because \(2 \times 2 = 4\); hence we write \(\sqrt{4} = 2\). So what’s the square root of 2, written \(\sqrt{2}\), the number that when multiplied by itself gives 2, i.e. \(\sqrt{2} \times \sqrt{2} = 2\)? Like its more famous relative \(\pi\) (the ratio of the circumference of a circle to its diameter), \(\sqrt{2}\) can only be calculated approximately. In the case of \(\sqrt{2}\) we have:

\[
\sqrt{2} = 1 \frac{1}{2} + \frac{3}{8} + \frac{15}{64} + \frac{35}{256} + \frac{315}{1024} + \frac{693}{4096} + \ldots
\]

the add-ons are getting smaller and smaller, but each one added on gives us a more accurate approximation to the square root of 2.

4. Following the ‘phone connection: something of a modern manifestation of the old game, but with the possibility of serious money attached, is the current enthusiasm for the entrepreneurial whiz-kid to write an ‘application’ for the iPhone – iPhone Apps and the like. It is true that with Gigabytes of memory to play with the pressure to compress cannot be as intense, but nevertheless App creation does appear to echo some aspects of the early days of programming.

5. As Gary Kasparov demonstrated in his famous 1997 encounter with IBM’s Deep Blue chess-playing computer, the best can sometimes be beaten. Contrary to many gee-whiz descriptions of the marvels of artificial intelligence, this one-off event showed nothing more than that Kasparov is human, and has his off days. It may not have escaped your notice that more than a decade on from that supposed breakthrough computers are still not contenders at World Chess Championships. Some claim this is because the human chess experts are excluding the chess programs to preserve human supremacy; others point out that Kasparov played a computer program and a team of human experts who ‘adjusted’ the Deep Blue system after every game, and no stand-alone program is good enough to compete. But did Kasparov stand alone? And so on … A substantial reflective article by Kasparov can be found on pages 2 and 3 of The New York Review of Books, 23 Jan. 2010.

6. Steven Johnson in his book Emergence (Penguin, 2001) holds up modern computer games as examples at the forefront of self-adaptive (i.e., ‘emergent’ in his use of the word) software, but admits that their adaptivity is far from open-ended learning. As is usual with these descriptions of hopeware, he consults the ever-optimistic (despite decades of failed predictions) crystal ball: “A few decades from now [2001], the forces unleashed by the bottom-up revolution may well dictate that we redefine intelligence itself, as computers begin to convincingity simulate the human capacity for open-ended learning,” p. 208. And it may well not. Indeed, it will certainly not, unless some huge leaps forward in the field of Machine Learning (ML) occur and surprise us all. Geoff Hinton, who knows as much about ML as anyone, once described ML as: a complex and difficult problem composed of very many, very complex and very difficult
subproblems (if I remember him correctly – I’m sure the gist is right). We only have a decade
to wait on this prediction.

7. The long-running saga of National Air Traffic Services’ £699m IT system for their new
Swanwick centre is well-documented. When already multi-millions over budget and years late,
the developers claimed to be working on 1,260 known bugs, i.e. known errors, in the system
which were (optimistically) reckoned to be “fixed” at a rate of 70 per week, although each
such week’s work introduced about 10 new bugs according to Professor Les Hatton in a letter
had the following front page story: More bugs hatch in new air traffic control IT systems
– a rise from 200 to 217 was reported.

8. My neat distinction between grammatically correct and behaviourally correct is really not so
clear cut. But for the purposes of my argument it is not misleadingly simplistic. Even with
formal languages, such as a programming language, syntax and semantics (grammar and
behaviour, respectively) are not well-defined, disjoint concepts, but they form the basis of a
useful distinction for many purposes.

9. Curiously, the accepted norms of mobile-phone texting explicitly eschew the syntactic precision
that a computer demands. But then a text message is for human comprehension; it is not instruc-
tions for a machine.
The Seductive Computer
Why IT Systems Always Fail
Partridge, D.
2011, XVI, 323 p. 56 illus., 6 illus. in color., Softcover