

Thinking about Programming

What is Software Engineering For?

Whenever we get confused, we must be able to see where we are going in order to know what action to take. We must know what we are trying to achieve.

We are software engineers. Why? What is software engineering for? What do software engineers do? We get some curious answers to this question. One chap said, 'They follow the procedures of the Software Engineering Standards!' Another said, 'They transliterate a requirement!'

Oh dear. We suggest that software engineers ensure the programs their customers need are running on their computers. That means our programs must do the right things. They must be robust. Sometimes we must know for certain that they are robust, and sometimes we will need to be able to prove it. We'd always like to be able to do all those things! The necessary programs must be running tomorrow as well, which usually means that our programs today must be maintainable. We must do our work cost-effectively, or we won't get the chance to write the programs in the first place. Our delivery must be timely.

We use all our inventiveness and the experience contained within our discipline to attain these goals. All our methodologies, standards, tools, languages are intended to assist us in attaining these goals.

We do nothing for the sake of it.

Software Engineering is Distributed Programming

The traditional view of the workplace is that the team is doing a job, and the individual is a part of this effort. But as mappers we can try looking at things in all sorts of odd ways, to see if they are informative. We can draw a system boundary around the programming team and notice that it does nothing that an individual programmer couldn't do. Activities such as requirement elicitation, design, implementation, test, management, review, build, archive and configuration management must all be performed by a single programmer doing even a small job. So we can see software engineering activities as the distribution of what a single individual could be doing quite effectively and responsibly in potter mode in his or her study!

We distribute programming for the same reasons that we distribute any kind of processing: availability, parallelism and specialisation.

This way of looking at things brings insights. We must select the divisions between tasks intelligently. Sometimes we can get benefits from putting two tasks with one person, where we need not be concerned if they remerge. For example, many organisations have a general practice of separating the identification of software requirements and architecture, but when they are following Booch style object modelling methodology, they take his advice and remerge these tasks. When we separate the skills of design and test, we can actually get added benefits from the situation, by controlling communication between the disciplines so that the test engineer's thinking is not compromised by the designer's. There was a project manager who was very much a packer. He didn't have a clear understanding of what he was doing and why, and had been led by the absence of any positive model of his job into thinking that a key objective was preventing this communication. The testers didn't know how to set up the conditions for the components they were to test, and the designers weren't allowed to tell them. Acrimonious arguments continued for days. These things really happen when we lose sight of the big picture.

We must make sure that the communication between distributed tasks is efficient, and that means that we must agree both a protocol and bear each others' needs in mind. Anything you'd need in your mind when you have completed one task and are about to embark on another, your colleague needs in his or hers. Your output will be no help to anyone if it doesn't tell

your colleague what they will need to do the next bit. We need to use our own ability to perform each others' jobs, no matter how naively, to monitor our own performance.

The final insight we need to raise at this point is that the black box of an individual programmer still exists in the team. The flow of information is not a linear series of transforms like a car factory, it is a fan-in of issues to a designer and a fan-out of solutions. The insight of the designer has not yet been distributed. Such an achievement would be a major result in AI.

What is Programming?

To understand software engineering we must understand a programmer. Let us allow a programmer to specify the requirement (to be identical with the user), and examine a scenario which ends in the construction of the simplest possible program: a single bit program.

Ada is sitting in a room.

In the evening the room becomes dark.

Ada turns on the light.

That is the fundamental act of programming. There is a problem domain (the room), which is dynamic (gets dark). There is order to the dynamic problem domain (it will be dark until morning), permitting analysis. There is a system that can operate within the problem domain (the light), and it has semantics (the switch state).

There is a desire (that the room shall remain bright), and there is an insight (that the operation of the switch will fulfill the desire).

Dynamic problem domains, systems and semantics are covered in detail elsewhere. On this course we are concentrating on understanding more about the desire and the insight.

It is worth pointing out here what we mean by a 'programmer'. A drone typing in the same RPG 3 invoicing system yet again might not be doing any real programming at all, but a project manager using *Excel* to gain an intuitive understanding of when the budget will get squeezed and what the key drivers are, most certainly is.

Programming is a Mapper's Game

We have a reasonable description of what programmers actually do, that makes sense. The two key words, 'desire' and 'insight', are things that it is difficult to discuss sensibly in packer business language, which concentrates on manifest 'objective' phenomena. While this is a very good idea when possible, it can hamper progress when applied as an absolute rule, which is how packers often apply rules.

It is worth making a philosophical point here. In order for any communication to take place, I must refer to something that is already there in your head. One way a thing can get into your head is as an image of something in the external world, and another is by being part of your own experience. If a part of your experience is unique to you (perhaps an association between pipe smoke and the taste of Christmas pudding, because of visits to your grandparents), we cannot speak of it without first defining terms. Even then, I cannot have the experience of the association, only an imagining of an association. But if the part of your experience is shared by all humans (perhaps our reaction to the sight of an albatross chick), we can speak of it 'objectively', as if the reaction to the chick was out there with the chick itself to be weighed and measured.

It has been argued that it is necessary to restrict the language of the workplace to the 'objective' because that is a limitation of the legal framework of the workplace. This is just silly. How do journalists, architects (of the civil variety) or even judges do it? This is the area where managers have to use their own insight to control risk exposure.

We suggest that the real issue here is that we are not very good at software yet. We probably never will be - our aspirations will always be able to rise. We are culturally constrained, and further influenced by the mature objective metrics that our colleagues in the physical, rather than information disciplines, routinely use.

To get anywhere with programming we must be free to discuss and improve subjective phenomena, and leave the objective metrics to resultants such as bug reports.

First, desire. In the example above, Ada likely did not begin with a clear desire for greater light. Her environment became non-optimal, perhaps uncomfortable, and she had to seek for a clear description of exactly what she wanted. This clarifying of one's desire is usually a nested experience where incremental refinement is possible, and proceeds in tandem with design. We will have more to say about the User Requirements Document later-- for now let us remember that the clarification of desire always has the potential to turn into a journey of exploration together with the customer.

Next, insight. This is the moment of recognition when we see that the interaction of the problem and the desire can be fulfilled by a given use of the semantics. It's kind of like very abstract vector addition with a discontinuous solution space. Or to put it another way, it's like doing a jigsaw puzzle where you can change the shape of the pieces as well as their positions. It is supremely intellectually challenging.

There is a pattern here that relates computer programming to every other creative art. We have three phenomena, Problem, Semantics and Desire (heavy capitals to indicate Platonic essences and like that). Problem and Semantics are of no great interest to the AI or Consciousness Studies people, but that Desire has something odd about it. These three phenomena are addressed or coupled to by three activities of the programmer. Looking consists of internalising the features of the Problem. Seeing comprehends the meaning of the Desire. Telling exerts the Semantics. Looking and Telling are domain specific. The poet might observe commuters, while the ecologist samples populations. The poet writes structured words, while the ecologist introduces carefully selected species. All of us do the same kind of Seeing. Talk to any artist about the good bits of your job.

We need all those wonderful mapper faculties to handle this stuff.

Programming is a mapper's game.

General Tips on Mapping

Packers have a whole proceduralised culture that provides behavioural tramlines for just about everything. It's so complete you don't even notice it until you solve a problem perfectly effectively one day, by a method that's not on the list. It might be something as trivial as getting out of the car and buying the Pay and Display ticket before driving along the car park and pulling into a space. Apparently one is 'supposed' to park the car, walk to the machine, and walk back again.

Mappers hardly ever get the upper hand on these cultural issues, but when it does happen it can be hilarious. A packer gave a dinner party and it so happened that over half of the guests were mapper types, IT workers and others. The host pulled a pile of warm plates from the oven, and started handing them to the guy on his left. 'Just pass them around!', he cried cheerfully. Everything went well until he passed out the last plate. Then his expression changed from confusion, to amusement and a distinct moment of fear before he realised he needed to shout 'Stop!'

Or maybe it was just a plea from the heart.

Mappers don't have a general cultural context to learn from, so we are almost entirely self taught. Here we have collected some observations we have collected talking to mappers. We can learn a great deal about mapping by talking to others.

Problem Quake

After you've been telling yourself about what your customer needs to accomplish for a while, chasing around the elements of the problem, how it is related, the physical capabilities of the systems available, the problem will suddenly collapse into something much simpler. For some reason, we rarely get it quite right in that sudden moment of understanding. Be ready to shift your new understanding around, and make the most of your aftershocks. This is a good time to express your new understanding to colleagues, and allow them to look afresh at things you may have stopped seeing because of familiarity.

Incremental vs Catastrophic Change

Sudden realisations come when they are ready, and we can optimise conditions to produce them. They have their problems. They are exhilarating, convincing, and sometimes wrong. When you get them, check them through with respect to everything you know, and try your best to break them. A big quake is always important, even if it doesn't bring an instant solution. On the other hand, we can often get a great deal of reduction out of chunking the problem and just moving lumps around. Don't feel embarrassed about thinking 'crudely' - start doing it now and you might get to see something a week next Tuesday. By which time people whose thinking consists of looking very serious will know nothing.

Boundaries

Focus on your boundaries. There are three classes of components to your problem. These are things you care about, things that affect things you care about, and things you don't care about. One of the reasons that mappers have an easier life than packers is that they take the initiative and try to identify all the external effects that could give them problems, and they don't just concentrate on stuff listed on bits of paper they've been handed. If you can find your boundaries, your problem is well defined and you can start to solve it. If you can't you might need to talk to your customer again, or draw your own boundary, which involves making assumptions that should be explicitly justifiable.

Explore Permutations

When you have a green duck, a pink lion and a green lion, ask yourself where the pink duck has got to. Understanding trivial and impossible permutations can lead to greater overall understanding, and some permutations are just plain useful in their own right.

Work Backwards

We all know how to solve mazes in childrens' puzzle books, don't we!

Plate Spinning

You know when your unconscious mapping faculty is going because of a fidgety, uncomfortable, even grouchy feeling. When that feeling eases off, it's your call. If you've got a date, leave it be! But if you want results, just take a quick tour around your problem from a couple of different perspectives or directions, and the fidgetiness will come back. It's like the way platespinners nip back to each plate and spin it up again before it falls off its stick.

Ease Off

After a great deal of physical work, you can attempt to lift something, but no movement occurs. The sensation of feebleness where you expected to be able to exert force is surprising and unmistakable. The mental equivalent feels very similar. There is absolutely no point pushing harder, but switching to rest mode instead of carrying on bashing away with your puny little neurons is not easy. This stuff runs on autopilot. You must obtain physical sensory stimulation. A shower, a noisy bar, a band. Get out of your surroundings. You can recover mental energy in a few hours if you stop when you know you can get no further.

Break Loops

The fidgety feeling that comes from effective background thinking is different to a stale sensation, sometimes even described as nauseous. Your brain has exhausted all the options it can find, and you need new empirical input. Get more data. Talk to someone. You obviously don't have some key datum, or your whole model is skew. So maybe you need to do a dragnet search of your problem. If it's a buggy program, put a diagnostic after every single line and put the output in a file. Then read it in detail over a cup of coffee. Sure it will take ages - do you have a better idea? If it's a hideous collection

of asynchronous events to be handled, write them out in a list by hand. This forces your attention onto one event after another, and you'll probably have new lines of inquiry before you are half way through.

Fault to Swapping

There are kinds of stupidity that only mappers have access to. Mappers can be paralysed by trying to optimise a sequence that is too big to fit in their heads. Perhaps they want to move the wedding cake before they put the spare wheel in the car so their hands are clear, but the spare wheel is here and the wedding cake is at Fred's, and so on. When this happens to a modern paged OS, it get itself out of thrashing pages by reverting to a swapping strategy. It just swaps out whole processes until the logjam clears, and then goes back to paging. Don't get paralysed - just do any job and then look again.

Duvet Stuffing

Turn the cover inside out, put your arms into it, and grab the far corners from the inside. Then grab the corners of the duvet with the corners of the cover, and shake the cover over the duvet. A bit of practice and you can do a king size one in less than 30 seconds.

Mapping and the Process

The purpose of software engineering is ensuring that the programs our customers need are running on their computers. Software engineering is distributed programming. From this perspective, we can define the process as a protocol for communicating with our colleagues through time and space. It provides a framework that tells our successors where to find design information they will need to do their jobs. By changing the process we communicate our experience to the future. It tells our colleagues in other parts of the team when we will meet, and provides a structure for our discussions. It provides common points in our projects where we can compare like with like, and so discuss aspects of our approach that we have varied.

The process is not a prescriptive meta-program for making other programs. While our activities must map to the process, it is not in itself sufficient for making programs. We think within the structure of the process, but there must always be a stage of interpreting the process definition in the light of any given problem. Remember that one always interprets the definition - abdicating this activity simply selects an arbitrary interpretation. One then usually ends up trying to manage the issues that would arise when building say, a futures trading system, when the problems that are emerging are those of the actual project say, a graphics rendering system. So you end up arguing about how you'll do requirements tracing for transaction journaling instead of worrying about the extra bits you need for the specular reflections!

Angels, Dragons and the Philosophers' Stone

Our ancestors were as smart as we are, and when it got dark at four o'clock in the afternoon, the other thing to do was play with the insides of their own heads. Understanding some puzzles from antiquity as the thinking of past mappers is useful not only because it is interesting, but because it shows us what the unaided human intellect is capable of. This is something we need to appreciate if we are to regain control of our work from the processes we have handed our lives and careers to.

Infinity was a hot topic, and our ancestors had broken this notion down into three different kinds. Conceptual infinity is easy - you just say 'forever' and you've got it, for what it's worth. Next there is potential infinity. You can give someone an instruction like, 'Keep counting forever'. In theory you could end up with an infinite collection of numbers that way, but could it ever really happen? Could you ever actually get an infinite number of things right in front of you, to do amazing conjuring tricks with? They realised that if an infinite collection of anything from cabbages to kings really existed, it would take up infinite space, so if there was an infinite collection of anything with any size to it, anywhere in the universe, we wouldn't be here. There would be nothing but cabbages - everywhere. We are here, so there is no infinite collection of anything with any size to it, anywhere. But there is still the possibility of an infinite collection of something infinitely small. If something can be infinitely small, then God (who is handy to have around for thought experiments because he can accomplish anything that can be done in this universe) should be able to get an infinite number of angels to dance on the

head of a pin.

Our ancestors felt that this idea was ridiculous, and that therefore there is no actual infinity in this universe. Today, we have two great theories of physics. One works at large scales and uses smooth curves to describe the universe. The other works at small scales and uses steps. We haven't got the two theories to mesh yet, so we don't know if the deeper theory behind them both uses steps to build curves, like a newspaper picture, or if it uses curves to build steps, like a stair carpet. It might be something we've not imagined yet of course, but if it's one or the other, our ancestors would guess the steps, because of the angels on the head of a pin.

What about the dragons? They roar and belch flame. Their noise travels faster than the wind. They collect precious jewels below the ground. They live in South America, China, Wales. They eat people. They are worms, and an ancient symbol for the world is the great world worm. They are a conceptual bucket in which our ancestors gathered together what we now call tectonic phenomena. They had no idea that the world is covered by solid plates wandering around on a liquid core, but they had eventually gathered all the effects together through mapping applied to direct observation. The dragon took the place of the real thing in their mental maps until by wandering around they discovered the real phenomena that produced the effects they tagged 'dragon'.

And alchemy? The futile search for a procedure for turning base metals into gold and getting rich quick? An alchemical or Hermetic journey consists of a series of operations (which may or may not have physical manifestation such as a diagram or experiment, or may be just a thought experiment), performed by the operator. The journey ends at the same place that it begins, and during the journey the operator's perception of the world is changed. The operator's consciousness has been deepened and enhanced, and it is he, not the stuff on his desk, that is transformed. The return to the beginning is necessary because it is only then that he sees that that which was obscure is now clear. Alchemy is mapping.

In the great cathedrals of Europe there are many arches holding up the roofs. In these days we'd probably get a symmetric multiprocessor to grind out a finite element analysis, but the builders didn't have the hardware or the algorithms. They didn't have the nice equations we have in mechanics, or even Newton's own Latin prose. Most of them were illiterate. But if you compute the optimal arch strength/mass curve for the spans, you usually find they were bang on. They did this with the only tools to hand - their own experience, and the ability we have to get a feel for anything with the neural net between our ears.

Make sure you have a realistic evaluation of your own capabilities. The usually necessary correction is up! Getting good at anything takes practice, but given that you'll be doing the work anyway, it's nice to know how good you can get.

Creative hacking and responsible engineering are orthogonal, not contradictory. We can have the pleasure of stretching our faculties to their limits, and still fulfill our obligations to our colleagues.

Literary Criticism and Design Patterns

There is an important difference between intentionality and action. A scriptwriter might intend to tell us that the bad guy is horrible, and will do it by writing scenes involving nasty deeds. Our intention might be to signal that a memory page in cache is no longer valid, our action is to set the dirty flag.

To starkly expose this point, consider an assembly language. An opcode might perform the most peculiar settings of the processor's outputs given the inputs, but we think of the opcode by its mnemonic, say DAA (Decimal Adjust Accumulator). Even though there is an identity between opcode and mnemonic, the high level intentionality of the mnemonic can mask the action of the opcode on the accumulator, which just flips bits according to an algorithm. If we see the processing opportunities in the opcode, are we 'abusing' it? The answer depends on the circumstance.

Whenever we have a distinction between intentionality and action, we have the opportunity to look at the effectiveness of the action, and ask what we can learn about the intent, or the domain of the intent, from the structure of the selected action. Might another action have been better? Do problems in the action reveal issues in the intentionality? When we do this with books it is called literary criticism, and taken seriously. If we are to learn how to write better programs, we need to learn as much as possible about our kind of lit crit, because that's the only way we'll be able to have a sensible discussion of the interplay of structure and detail that characterises style. The really nice thing is, unlike prose lit crit, program lit crit is informed by experimental evidence such as failure reports. This ups the gusto and cuts the waffle, leaving the learning

enhanced.

We can get a rigorous and elegant coding discipline out of the difference between intentionality and action. Consider the following fragment:

```
// Search the list of available dealers and find those that
// handle the triggering stock. Send them notification of
// the event.
```

```
for(DealerIterator DI(DealersOnline); DI.more(); DI++)
    if(DI.CurrentDealer()->InPortfolio(TheEvent.GetStock()))
        DI.CurrentDealer()->HandleEvent(TheEvent);
```

The definition of the objects has allowed the intentionality of the use case to be expressed succinctly. However, there is really no smaller granularity where we can cluster intentionality into comment and action into code without the comments getting silly.

If we interleave comment and code at this natural level of granularity, we can ensure that all lines in the program are interpreted in comment. We are motivated to design objects (or functions) that we can use economically in this way. We find it easier to correct some inelegance than to explain it away.

By being conscious of the difference between intentionality and action, we can make both simultaneously economical, and fulfill the goals of a detailed design document's pseudo code and an implementation's comments, while helping the implementation's verifiability. By putting everything in one place, we assist the coherence of the layers.

This concept is taken further in Donald Knuth's idea of 'Literate Programming', which to be done well, really needs tool support from systems like his Web environment (predating the World Wide Web). But you don't need to buy all the gear to enjoy the sport - literate programming is more an attitude than a tool.

It is at this level of programming lit crit that we can seriously benefit from studying design patterns. These are chunks of architectural technique more complex than the usual flow control, stream management with error handling and other typical kinds of idiom. They are extremely powerful, and very portable. See the wonderful book by Gamma, Helm, Johnson and Vlissides, where they describe a pattern as something that:

'... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice.'

The theme that underlies all the issues discussed in this section is Aesthetical Quality. We all know a mess when we see one, but too often we are in danger of being paralysed, unable to act on the evidence of our own senses because there is no procedural translation of 'It works but it's ugly.' When an experienced professional feels aesthetical disquiet and cares enough to say so, we should always take notice. Our standards of beauty change from generation to generation, and for some reason always follow function. That is why making code beautiful exploits a huge knowledge base that we may not have consciously integrated, and leads to cost effective solutions. The stuff is less likely to incur vast maintenance costs downstream if it's beautiful. That's what beauty is. Aesthetical quality is probably the only criterion against which one can honestly argue that the wrong language has been used. An attempt to do an impressionist dawn in acrylic would be horrible even if the airbrush work were perfect.

We should be willing to look at the source code we produce not as the end product of a more interesting process, but as an artifact in its own right. It should look good stuck up on the wall. The up-front costs of actually looking at our code, and exploiting the mapping of the geometrical patterns of black and white, the patterns in the syntax, and the patterns in the problem domain aren't that great, given that they sometimes literally let you spot bugs from six feet away.

With all this literary criticism, what of religious wars? Some of it is done for entertainment of course, and we don't want to impede the pleasure of ridiculing the peculiarities of our friends' favourite tools and techniques! But sometimes intelligent programmers get caught in distressing and counter-productive squabbles that just go around in circles. We forget that we can use structured arguments rigorously between ourselves. When an unwanted religious war breaks out, ask the following questions:

1. What is the global position that includes both special cases?
2. Is there a variation in intentionality between the positions?
3. What is the overall objective?

For example, you value the facilities of a powerful integrated environment. You use *Emacs* at work and have extended it to control your coffee machine. I use many machines, and bizarre though it may be, I know *vi* will always be there. We install *Emacs* on new sites, and teach *vi* to novices. Your LISP technique of course, sucks.

This evaluation of options against objectives often produces a genuine convergence of opinion amongst experienced people. Agreeing on the best idioms to get a job done in a well-understood environment does not mean that everyone is coerced to conform - they just agree. Contrary to popular opinion there often is a right answer. Talking to an experienced person about a new environment's idioms can teach you an awful lot very quickly, while using the idioms from your old environment in a new one can lead to fighting all the way.

Cognitive Atoms

In any task that requires understanding, we will always find at least one 'cognitive atom'. A cognitive atom is a part of the problem that can only be adequately treated by loading up its elements, features, signs, whatever into the mind of a single mapper and getting the best result possible. The word 'adequate' is important here - there are a whole bunch of problems that given unlimited resources could be tackled slapdash, but need thinking about in the real world. For example, any bunch of idiots could pull off the set changes needed on the stage of a big musical show, given several weeks to do it. Doing the same thing in the time it takes the leading lady to sing a melancholy song under a single spotlight takes a logistical genius.

Experienced project planners discover that recognising and managing the cognitive atoms within a project is a crucial early step in gaining control. First we must recognise the cognitive atoms. There is a relationship between the system architecture and the cognitive atoms it contains - the architect will have to use intuition and experience to identify solvable, but as yet unsolved problems. The problems the architect believes can be solved during development will influence the design, because nobody wants to design an architecture that is not implementable!

The architect can therefore move the boundaries of the cognitive atoms around somewhat. For example, in a data mining system, practical combinatorial problems might be concentrated in the database design, or in the higher level application logic. The correct identification of the cognitive atoms will control both the architecture and the work packages devolved to the team members. Each atom must be given to one person or subgroup to kick around, but they may find themselves working on more than one part of the system to solve their problem. The parts must therefore be well layered so that modules do not turn into time wasted battles. Identifying atoms usually requires balancing time, space, comms, risk, team skills, portability, development time, and all of this must be done with proposed atoms whose solvability is not certain. The architect must therefore be able to see the heart of the problem, and express, at least in his or her own head, the nature of the tradeoff options. It is quite possible to be able to recognise a set of clearly seen tradeoffs that it is very hard to express to another without the same mapper ability to see the structure. Serialising a mental model is always difficult, because we do not think in technical papers that we download like ftp transfers.

When identifying cognitive atoms it is important to avoid being confused by a specific fallacy that one sees over and over again. It is often possible to keep breaking atoms into smaller ones without much thought, and thus reach code level without much effort. When such reductions reach implementation however, all turns to chaos. The real problems haven't gone away, they've just been squeezed down into ugly subsystem APIs, performance problems, fragility and the like. The boundaries of the cognitive atoms have been squeezed smaller and smaller, until... Pop! They re-appear around the whole system itself! The doctrine of simplistic stepwise refinement without regular reality checks and rigorous attempts to falsify the design has been responsible for a great many tragedies, involving wasting most of the time budget on attempting to do the actual design work by open-house informal acrimony, followed by desperate kludging attempts.

The reduction of cognitive atom boundaries can be cyclic, and the skilled architect will pick the right place for them, whether that is high level, low level or in between. Some initial studies can be a huge, single cognitive atom, that one just has to hand to a trusted worker and say 'Try to sort this mess out please!'

By definition we don't know how to best approach a cognitive atom. If we did, it wouldn't be an atom. So it follows that it cannot be planned on a project planning Gantt chart in terms of subgoals. It must be entered as a single task, and the duration must be guessed at. Experienced mappers get pretty good at guessing, but they cannot explain why the problem smells like a two day one, a week one, a six month one. Therefore there is little point arguing with someone who has given their best guess. The fear of the subsequent argument is an important factor that often prevents mappers engaging their intuitive skills, and giving the numbers that are needed for project planning.

The upside of this is that once a cognitive atom fissions, the worker can usually lay out a very detailed set of task descriptions based on a solid understanding of what has to be done. Therefore many projects should plan to update their Gantt charts as the cognitive atoms fission. We suggest that the high proportion of projects that attempt to Gantt everything on day one indicates the pervasiveness of the production line model. The programmers working under such Gantt charts cannot be benefiting from intelligent management of cognitive atoms. Instead of opening their minds to the problems to be solved, they will be arguing about whether or not they are any good at their jobs and being 'put under pressure', as if it is possible to make someone think more clearly by belittling them. This is stressful and counter-productive.

The Quality Plateau

When one adopts the strategy of forming one's own mental map of a problem domain and attempting to simplify it, one faces the problem of when to stop working on the map. This applies at every level of design. The extraordinary thing is that there almost always is a deep solution, that is significantly simpler than anything else, and manifestly minimal. (There may be more than one way of expressing it, but then the relationship will be manifest.) Although homilies like 'You'll know it when you see it!' are undoubtably true, they don't tell one where to look.

As the only honest argument we can offer here is the promise that this really happens. And although it proves nothing, all we can do is show you a worked example reduced to a minimal state. But it does work - ask anyone who's tried.

The example we will present is from Jeffrey Richter's excellent *Advanced Windows*. This book is essential reading for anyone intending to program against Microsoft's Win32 Application Programming Interface (API) (because otherwise you won't have a mental map of the system semantics).

Richter sets out to provide as clear an exposition of how to use Win32 as he can, but even in his examples (and partially as a result of the conventions he is following), complexity that we can lose appears. On page 319, there is a function `SecondThread()`. We'll just look at the function, and leave the remainder of the program and some global definitions:

```
DWORD WINAPI SecondThread (LPVOID lpwThreadParm) {
    BOOL fDone = FALSE;
    DWORD dw;

    while (!fDone) {
        // Wait forever for the mutex to become signaled.
        dw = WaitForSingleObject(g_hMutex, INFINITE);

        if (dw == WAIT_OBJECT_0) {
            // Mutex became signalled.
            if (g_nIndex >= MAX_TIMES) {
                fDone = TRUE;
            } else {
                g_nIndex++;
                g_dwTimes[g_nIndex - 1] = GetTickCount();
            }

            // Release the mutex.
            ReleaseMutex(g_hMutex);
        } else {
            // The mutex was abandoned.
            break; // Exit the while loop.
        }
    }
}
```

```

    }
}
return(0);
}

```

First let's just simplify the brace style, loose the extra space between keyword and open bracket, and the redundant ReleaseMutex comment. We are aware that there is a religious war between the followers of K&R and Wirth on brace style, but getting the blocking symmetric really does make things easier to see. The extra line it takes will be won back later - bear with us!

```

DWORD WINAPI SecondThread(LPVOID lpwThreadParm)
{
    BOOL fDone = FALSE;
    DWORD dw;

    while(!fDone)
    {
        // Wait forever for the mutex to become signaled.
        dw = WaitForSingleObject(g_hMutex, INFINITE);

        if(dw == WAIT_OBJECT_0)
        {
            // Mutex became signalled.
            if(g_nIndex >= MAX_TIMES)
            {
                fDone = TRUE;
            }
            else
            {
                g_nIndex++;
                g_dwTimes[g_nIndex - 1] = GetTickCount();
            }

            ReleaseMutex(g_hMutex);
        }
        else
        {
            // The mutex was abandoned.
            break;// Exit the while loop.
        }
    }
    return(0);
}

```

It's easy to lose one local variable: dw is assigned then tested in the next statement. Inverting the sense of the test helps locality of reference (testing then changing g_nIndex). And while we are about it there is no point incrementing g_nIndex just to subtract 1 from its current value in the next operation! We are already using the C post-increment operator, which was provided for just this sort of job.

```

DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
{
    BOOL fDone = FALSE;

    while (!fDone)
    {
        // Wait forever for the mutex to become signaled.
        if (WaitForSingleObject(g_hMutex, INFINITE)==WAIT_OBJECT_0)
        {

```

```

        // Mutex became signalled.
        if (g_nIndex < MAX_TIMES)
        {
            g_dwTimes[g_nIndex++] = GetTickCount();
        }
        else
        {
            fDone = TRUE;
        }
        ReleaseMutex(g_hMutex);
    }
    else
    {
        // The mutex was abandoned.
        break;// Exit the while loop.
    }
}
return(0);
}

```

The break depends only on the result of WaitForSingleObject, so it is a simple matter to move the test up into the controlling expression, eliminating both the break and a level of indentation:

```

DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
{
    BOOL fDone = FALSE;

    while (!fDone && WaitForSingleObject(g_hMutex, INFINITE)==WAIT_OBJECT_0)
    {
        // Mutex became signalled.
        if (g_nIndex < MAX_TIMES)
        {
            g_dwTimes[g_nIndex++] = GetTickCount();
        }
        else
        {
            fDone = TRUE;
        }
        ReleaseMutex(g_hMutex);
    }
    return(0);
}

```

Now just squeeze... We know that lots of coding standards say that we must always put the curly brackets in because sometimes silly people made unreadable messes, but look what happens when we dump the rule, and concentrate on the intent of making the code readable.

```

DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
{
    BOOL fDone = FALSE;

    while (!fDone && WaitForSingleObject(g_hMutex, INFINITE)==WAIT_OBJECT_0)
    {
        if (g_nIndex < MAX_TIMES)
            g_dwTimes[g_nIndex++] = GetTickCount();
        else
            fDone = TRUE;
    }
}

```

```

        ReleaseMutex(g_hMutex);
    }
    return(0);
}

```

Now for some real heresy. Gosh, by the time we've finished this total irresponsibility the result will be totally illegible. (Or of course, common sense can do more good than rules.)

Heresies are, if we know what our variables are, we'll know their type. If we don't know what a variable is for, knowing its type won't help much. Anyway, the compilers type-check every which way these days. So drop the Hungarian, and gratuitous fake type extensions that are just `#defined` to nothing somewhere. Hiding dereferences in typedefs is another pointless exercise because although it accomplishes a kind of encapsulation of currency, it is never sufficiently continent that we never have to worry about it, and then a careful programmer has to keep the real types in mind. Maintaining a concept of long pointers in variable names in what is a flat 32 bit API is pretty silly too.

```

DWORD SecondThread (void *ThreadParm)
{
    BOOL done = FALSE;

    while (!done && WaitForSingleObject(Mutex, INFINITE) != WAIT_OBJECT_0)
    {
        if (Index < MAX_TIMES)
            Times[Index++] = GetTickCount();
        else
            done = TRUE;

        ReleaseMutex(Mutex);
    }
    return(0);
}

```

Now watch. We will hit the Quality Plateau...

```

DWORD SecondThread(void *ThreadParm)
{
    while(Index < MAX_TIMES &&
        WaitForSingleObject(Mutex, INFINITE) == WAIT_OBJECT_0)
    {
        if (Index < MAX_TIMES)
            Times[Index++] = GetTickCount();
        ReleaseMutex(Mutex);
    }
    return(0);
}

```

Eleven lines vs 26. One less level of indentation, but the structure completely transparent. Two local variables eliminated. No else clauses. Absolutely no nested elses. Less places for bugs to hide.

Finally, the text has made it clear that different threads execute functions in different contexts. It is not necessary to define one function called `FirstThread()`, with exactly the same cut and paste definition as `SecondThread()`, and call them,

```

hThreads[0] = CreateThread(..., FirstThread, ...);
hThreads[1] = CreateThread(..., SecondThread, ...);

```

When it could just say,

```

hThreads[0] = CreateThread(..., TheThread, ...);
hThreads[1] = CreateThread(..., TheThread, ...);

```

About a third of this example is actually clone code! If we did see a bug in one instance, we'd have to remember to correct

the other one too. Why bother when we can just junk it. It's this kind of thing that stresses deadlines.

Knowledge, Not KLOCS

Programmers are expensive. The results of their work must be captured and used to the benefit of their organisation. The trouble is, the traditional packer way to count results is to count what they can be seen producing. The results of a programming team studying a problem, coming to an understanding, and testing that understanding down to the ultimate rigour of executable code are not the KLOCS of code they typed in when they were learning. They are the final understanding that they came to when they had finished.

The reason why it is important to identify value that way around is that sometimes, the understanding shows a much easier way of doing things than the design the team started with. A classic mapper/packer battleground in programming consists of the mappers seeing that with what they know now, a reimplementaion could be done in a fraction of the time, and would not suffer from maintenance issues that they see looming in the existing code. The packers see the mappers insanely trying to destroy all their work (as if there weren't backups), and repeat the last few months, which have been terrible because they obviously didn't know what they were doing anyway (they kept changing things). The packers set out on one of their crusades to stop the mappers, and the organisation has to abandon its understanding, which cannot be used in the context of the existing source code.

The intelligent organisation wants the most understanding and the least source code it can achieve. The organisation stuck in inappropriate physical mass production models doesn't count understanding, and counts its worth by its wealth of source code, piled higher and deeper.

Good Composition and Exponential Benefits

A definition of a good composition that is often used on Arts foundation courses is that is such that 'if any element were to be missing or changed, the whole would be changed'. Perhaps it is a seascape, with a lighthouse making a strong vertical up one side, guiding the eye and placing itself in relation to the waves beneath. The situation of the lighthouse (and the waves) is one we recognise, and this is where the painting gets its power. If the noble lighthouse was a squat concrete pillbox, the picture would say something else. If the waves were an oilslick or a crowd of frisbee players, there would be still others messages in the painting.

The point is, there shouldn't be anything around that does not have a carefully arranged purpose with respect to the other elements of the composition. The artist needs to keep control of the message, and if the picture contains random bits, they will trigger unpredictable associations in the viewers' minds, and obscure the relationships between the important elements that the picture needs to work at all.

Logicians examining axiom sets face exactly the same issue. They have a much more precise term for what they mean, but this comes simply from the tighter formal structures that they make their observations and propositions within. They say that an axiom set should be 'necessary and sufficient'. A necessary and sufficient set allows one to see clearly the 'nature' of the 'universe' being considered. It allows one to be confident that the consequences one finds are truly consequences of the area of interest, and not some arbitrary assumption.

In neither of these disciplines would it be necessary to remind people of the importance of keeping things as small as possible, as an ongoing area of concern. Unfortunately, the practical usefulness of our art means that people are often keen to see new functionality, which we try to construct as quickly as possible. When established, functionality becomes part of the background, and all of us, from corporate to individual entities, start to become ensnarled in our own legacy systems.

Although this may seem like an eternal unavoidable of the Programmer's Condition, one does see people breaking out of this cyclic degeneration, and from this perspective of programming as a creative art, we can describe how they do it.

The fundamental difficulty in keeping control of legacy structures, be they artifacts of the customer's transport strategy that have made it into the specification for the fixed costs amortisation logic, or an ancient CODASYL indexing system that one is being asked to recreate in an object database, is time. This is sometimes expressed as 'cost', but the issue is rarely

cost. It is deadlines. Apart from circumstances where the misguided cry `Wolf!' there is no getting away from deadlines. They are a commercial reality over which we have no control. That's OK - we just think about them realistically and manage their problems rather than use them to justify poor products.

The first point of leverage against deadlines is recognising that work proceeds in a clean environment without odd flags on functions, inconsistent calling conventions, multiple naming conventions and the like, than with the junk in place. Days after cleanup count more than days before cleanup. So do the cleanup first, when everyone can see a long project ahead of them, and get the time back later. You will nearly always have to do a cleanup - the code that most organisations put in their repository is usually the first that passes all test cases. This does not matter. Do your own cleanup for this phase, regression test and don't even discuss your own deltas until you can see straight.

The warning that comes with this observation, is to be realistic about how long your cleanup will take. The nastier the tangle, the bigger the multiplier a cleanup will give, but the greater the risk that you won't have time to sort it out and do the work. A useful question often is, `How complex is the black box functionality of this thing?' If the answer is `Not very!', then you know that as you incrementally comb the complexity out, it will collapse to something simple, even if you can't see the route at all.

The second point of leverage comes from the exponential collapse of complexity in software. If you have a cleaner algorithm, the minimal implementation will be simpler. The less code you have, the easier it is to see the structure in the code, and the chance of off-concept bugs is reduced. At the same time, less code means fewer opportunities for syntax errors, mistyping of variables and so on. Fewer bugs means fewer deltas, fewer deltas mean fewer tests. It doesn't take long in any team of more than half a dozen people for most of their activity to descend into a mayhem of mutual over-patching, with repository access being the bandwidth bottleneck. Letting loose stuff through the process into later stages can plant a time-bomb that will blossom when it is too late to do anything about it. On the other hand, a frenzy of throwing away in the midst of such a situation can regain calm in a matter of days.

The third part of leverage is the `skunkworks', so called because the original Skunkworks was located by Lockheed Martin, at a remove from its corporate centre, `because it stunk.' This fearsome technique can be used by excessively keen teams in secret on winter evenings, or can be mandated by enlightened managements. As with everything on this course, we will offer an insight into why skunkworks work.

In industrial age activities like housebuilding, we have physical objects (bricks) which are awkward to manage. Instead of piling up bricks against reference piles to see how many we will need to build a house, we count them. The abstraction from physical to informational gives us enormous leverage in managing bricks. Eventually we have so many numbers telling us about supply, transport and demand that we have to organise our numbers into patterns to manage them. We use spreadsheets, and the abstraction from informational to conceptual again gives us enormous leverage.

In information activities such as programming, we don't start with the physical and get immediate leverage by moving to the informational. We start with informational requirements, listings etc., and we have to manage these with informational tools. We have to do this for good reasons, such as informational contracts with suppliers, and informational agreements on meetings with colleagues contained in our process. We also sometimes do this for bad reasons, such as a too literal translation of informational techniques for managing housebricks into the informational arena, such as counting productivity by KLOCS.

The trouble is, in normal working we have no leverage. The information content of a meeting's minutes can be bigger than the requirement they discuss! As an activity performed by humans, the meeting has negative leverage! We only win because we can sell our new bit either many times, or because in collaboration with other bits it gives greatly added value to the process.

This leaves the opportunity to use understanding to gain leverage over information. The skunkworks is sometimes seen as an abandonment of the process in the interests of creativity. Nothing could be further from the truth. One needs a high proportion of experienced people to pull the trick off, because they must lay down highly informed dynamic personal processes to get anything done at all. What one trades off is the understanding contained in an exhaustive process, for the understanding contained in experienced people. From this comes the precondition for the skunkworks. By abandoning the detailed process, one accepts that risk is inevitable, and loses the personal protection given by simple, well-defined objectives. Everybody must accept that a skunkworks may fail, that what it delivers might not be what was expected, and that there may be issues reinserting the results into traditional management streams. But when they work, they work

magnificently!

All successful startups are skunkworks. So are unsuccessful startups. A skunkworks effort can turn a major maintainability bloat risk into a small upfront time risk. In these situations, it can be an effective risk management tool.

This file last updated 3 October 1999

Copyright (c) Alan G Carter and Colston Sanger 1997

alan@melloworld.com

colston@shotters.dircon.co.uk