

Customs and Practices

The Codeface Leads

TQM is all about awareness. Awareness of what we are doing when we perform repetitive procedures or do similar kinds of jobs allow us to capture those rare moments of insight, where we see a way to do things more effectively, and communicate them to our colleagues by modifying our process. This means that the process definition might contain boring stuff necessary for establishing communication between groups, but the stylistic or approach issues discussed should all be little treasures, worthy of transmission in such a high visibility document. The idea of this aspect of the process is not to specify every last little action, but to retain knowledge. This gives us a test, albeit still a subjective, matter of opinion type test, for whether a clause is worthy of the document. For example, microspecification of header ordering is not appropriate for inclusion in the coding standard because apart from anything else, it will almost certainly be violated on every real platform if the code is to compile. However, the technique of using conditional compilation macros around the whole module contents to prevent multiple inclusion is a little treasure that belongs somewhere that everyone can see it and follow the protocol.

In car factories, the shop floor leads continuous improvement, because the people that do inefficient jobs recognise that they are inefficient and correct them. The genuine parallel with software engineering should be that the codeface leads improvements to the process. One of the most costly consequences of the mapper/packer communication barrier is that packers, in panic because of the "software crisis" are obliged to assert that 'the process' is an inexplicable, mysterious source of all good bits, and as such, is correct. In some organisation, the process becomes the mechanism of a coercive attempt to enforce a rigid packer roboticism on the whole workforce, as this is seen to be the correct mindset to magic success out of nothing.

To get an idea of the scale of this problem, consider the evolution of programming languages and models, development environments, CASE tools and so on over the last thirty years. There is really no comparison between the two ends of the interval, in anything except coding standards. Some aspects of the discussion begun by Dijkstra on structured programming were captured as ritualistic dogma, and then the dogma has been copied from standard to standard ever since. Indeed, a major feature of most coding standards eagerly proclaimed by their promulgators is that they have copied the thing from someone else. This is how one sells rubbish to the frightened and ignorant. Saying that one has nicked something off someone else is a good way of adding false provenance to something with no inherent value, as well as learning from something with inherent value. Coding standards are handed down from management to programmers, rather than discovered at the codeface and passed upwards. The kind of lively and informed (if primitive) discussion that led to the original coding standards, that were wonderful at their time, has not been repeated since. As soon as the first standards were defined, and improvements were noted, the packer business world seized them, set them in stone, and announced that they constituted 'proper procedure'. Stylistic debate has been sidelined to the religious wars, where it does not face the responsibility of running the industry and so gets silly. Meanwhile the existence of these 'proper procedures' implicitly denies the existence of new stylistic issues coming along with new languages and models, that need informed debates as intense as those surrounding structured programming if we are to learn how to use these languages well.

A programmer using something like the ParcWorks Smalltalk development environment gets as much benefit out of a sanctimonious mouthing about not using gotos and putting data no- one ever looks at in standard comment lines at the top, as a modern air traffic controller gets out of the bits in Deuteronomy about the penalties for having sex with camels.

Who Stole My Vole?

This section is about complexity. We'll start with a thought experiment, involving an imaginary Martian ecology.

On Mars (as everyone knows) there are rocks. There are also two kinds of lifeforms. There are Martians, who eat voles, and luckily for the Martians, there are voles. Voles hide behind rocks and eat them.

Not much happens on Mars. Martians mainly spend their time sitting in the desert and watching for voles darting between rocks. Because there are rocks as far as the eye can see on Mars, in all directions, a Martian needs to be able to see well in all directions at once. That is why Martians evolved their characteristic four large eyes, each mounted on stalks and pointing in different directions.

Not much happens on Mars, so Martian evolution has progressed entirely in the direction of vole spotting. Each huge eye has an enormous visual cortex behind it, that can spot a vole miles away in all manner of light conditions. Most of a Martian's brain consists of visual cortex, and these four sub-brains are richly cross-connected to allow for weird light conditions to be compensated for. Martians do a great deal of processing up front, in the semi-autonomous sub-brains, so they don't really have 'attention' like humans - they focus on the play of input between their 'attentions' instead.

When they spot a vole, the Martians have to sneak up on it. This means keeping the vole's rock between themselves and the Vole. It requires Intelligence. Soon after the Martians evolved Intelligence, they invented Great Literature. This is scratched on big rocks using little rocks. It obeys the rules of Martian grammar, and uses the North voice for emotion, the South voice for action, the East voice for speech, and the West voice for circumstance. Not much happens on Mars, so the closest Martian equivalent to our own *Crime and Punishment* is called *Who Stole My Vole?*:

Emotion	Action	Speech	Circumstance
Grumpy	Sneak	Horrid Martian	Cold, Gloomy
Determined	Bash	Die! Die! Die!	Old Martian's Cave
Ashamed	Steal Vole		Dead Martian

Breathtaking, isn't it? That sudden void in the East voice as the South voice swoops - a view into the very mouth of an undeniable damnation, real albeit of and by the self! I'm told it helps to have the right brain structure...

What is the point of this Weird Tale? Well, imagine what a Martian programmer might make of C.A.R. Hoare's Communicating Sequential Processes (CSP) theory. Its brain (easy to avoid gender pronoun here - Martians have seventeen sexes, and a one night stand can take ten years to arrange) is already hardwired to enable it to apprehend complex relationships between independent activities, so the processes Hoare renders linear by series of symbolical transforms, and thus intelligible to a human are already obvious to a Martian's inspection. On the other tentacle, by squeezing all the work into a huge tangle that only one sub-brain can see, the human readable version is made unintelligible to the Martian.

A joint human Martian spaceship management system design effort, with lots of communicating sequential processes controlling warp drives, blasters, ansibles and so on would face problems bleaker than the mapper/packer communication barrier, even though theorem provers using CSP could perform automated translations of many of the ideas.

The point is, complexity is in the eye of the beholder. We don't need an alien physiology to notice the difference between how individuals rate complexity - it's the whole point of mental maps. When we discover a structure that lets us understand what is happening, we can apprehend the antics of many more entities in a single glance. Think of any complex situation that you understand, perhaps the deck of a yacht or the stage of an amateur dramatics company. When you first saw it, it would have looked like a chaos of ropes, pulleys, vast sheets of canvas, boxes, walkways, and odd metal fitments of totally indiscernible purpose. When you had acquired a pattern by finding out what all that stuff was for, the deck, stage or whatever seemed emptier, tidier than when you first saw it. But it hasn't changed, you have.

No sane skipper or director would attempt to operate in such a way that the greenest novice could understand what is going at first glance. Just sailing around the harbour or raising the curtain take some training.

In the software industry the leverage issue, mapper/packer communication barrier and language specialisation have all acted to mask this point. The leverage issue says that the benefit we get from moving numbers around instead of bricks means that we can afford the investment to ensure that the simple numbers describing bricks are presented in a form accessible to all. The industrial and commercial context of many programming operations has the notion that the competent just have their information organised, that complexity is handled by not having any, and that in that way, the progress of the bricks is assured. This is all true, and is the correct attitude to information about bricks. But when we substitute information for bricks, we get a problem. We can't abstract from a big ugly brick to an informational '1'. We can

abstract of course, but every time we do, we lose important information which may bite us later. We can't just say that the competent just have their data organised, because the job is now organising the huge pile of data that has just been dumped on us. We no longer need the fork lift driver's skills, but we need new ones. And we can't handle the representation of complexity just by requiring that the representation be simple. The mapper/packer communication barrier makes the situation with this inappropriate analogy between information and bricks harder to discuss, because just about every step of the brick logic is there in the information argument, but instead of 1% control and 99% payload, the management problem is more like 90% control and 10% payload. This relationship is what makes the difference in all the brick management heuristics, and it's relationships that packers score badly on. They think they recognise the situation, trot out the knowledge packet about being neat and documenting everything, and off they go. The idea that they may be creating a stage rocket that will never get off the ground because the engines are too inefficient, and exponentially greater management of the management will is needed to compensate for lack of subtlety is hard for someone to grasp if they aren't trained to draw mental models and see patterns in them. Finally, the existence of specialist languages increases the appearance of the possible. If only everything could be as easy as SQL. One must remember:

1. It's taken 30 years
2. The kind of things it does are very restricted.
3. It's very processor intensive

SQL isn't easy. It's exactly what we've described - control through familiarity with idioms - everybody understands horrors like outer joins.

The Knight's Fork appears again here. Is it better to make a really sophisticated code, send it, and then a short message, or to send a simple code, and a longer message. What is the expected calibre and experience of the maintainer? How much can we extend their understanding in the documentation, so we can use more `complex' idioms? A long `switch()` statement is usually a pretty grim way to do control flow, unless you're writing a GUI event loop, where we all expect it and go looking for it.

There is no absolute measure of `complexity'. This must be born in mind when talking of complexity of algorithms and code style, and whatever it is that complexity analysis tools produce after the pictorial representations of the system, which can be very valuable. Complexity in these pictures (after the system has been reduced to necessary sufficiency) is not A Bad Thing - it is the essence of your problem laid bare. We should not be trying to drive out the inherent complexity of problems. It is a futile strategy that leads us away from the development of abstract understanding that will let us organise it.

Reviews and Previews

A fundamental element of the packer view of work is control by threat. Perform action so-and-so, or else. To make the threat effective, the rule must be policed, so we must look, after the fact, to ensure that the rule has been followed. Then the idle and untrustworthy workers will know they will get caught out and will perform the action as specified, because of their fear of retribution. In fact, an important aspect of denying the nature of mapper work is supporting the falsehood that mapper jobs can be microspecified, and the only reason why that must be done is so that rules can be specified. Only with rules written on pieces of paper can the programmers be caught breaking them, and it's the catching out that is central to the whole model!

Of course, there is an additional purpose, in that a review can also spot small oversights than must be corrected before the work is allowed to go to the customer. This is like inspecting a Rolls Royce and polishing off a tiny smear on the bonnet with a soft cloth - it can't turn a soap-box cart into a Rolls Royce.

In the software industry, we have processes that put the conventional emphasis on review, which is either trivial or a police action, but do nothing to pull the team's focus and group experience into finding the best way to do the job. This leads to reviews where individuals or subgroups produce their best efforts, which frankly are often not very good, especially if the programmers concerned are inexperienced. The time for doing the work has elapsed, so if a baroque collection of application layer processes with complex instantiation rules have been used to kludge up something the OS does for free, it is too late to redesign. The review group members have to collude to not notice the ropey logic, and concentrate their efforts onto ritualised objections to trivial matters of style. None of this does anything for quality.

The solution of course, is a mapper one. We have to accept that most programmers would love to do a really good job, given half a chance, and invest in helping them rather than threatening them. Instead of spending our half-day or whatever in a review when it is too late, we should spend it in a preview, where we can evaluate the options and agree the general direction, before the work is done. Then the situation where the reviewers pass the work with regret can be avoided, because with the workpiece already Rolls Royce shaped, the final review just need to check for trivial smears on the bonnet.

Code Inspections and Step Checks

Code inspections form an important part of many organisations processes. The primary reason they are done is to fulfill the common sense, mapper, TQM dictum: 'When you have done the job, look at what you have done, and check that bit is OK.' But there is something funny about code inspections, that often deflects their purpose. This is because code inspections are like Christmas. They are an old festival, older than the structure they currently have a place in. And like Christmas, there's still a lot of holly and mistletoe hanging around from the Old Religion.

Once upon a time, programs had to b written down on coding sheets, which were given to typists to key onto punch cards. These were rekeyed for check, and then the cards were fed into the computer, which would be used for an incredibly expensive compiler run. It wasn't just the labour and capitalisation costs - the cycle time for a single attempt could be a week. So the wise developed the habit of sitting down together and examining each others' coding sheets in minute detail before they were sent off for punching.

Today we have spiffy editors, and processors in our toasters, so the original motives no longer apply, but we need to continue to inspect our code so that logical errors can be identified before they cause a fault in service. This is where the confusion sets in. Few organisations are as confused as the IT manager who recently said that the staff should perform code inspections on their listings before compiling them. For pities sake, we have our design phase to get the big picture right, and the compiler to check syntax errors. Manually poring through the code looking for syntax errors is not going to impose any useful discipline, and is a very high cost and unreliable method of finding syntax errors, even though it is the way it has always been done! The balance has changed, and we need to consult our mental maps, as with everything we do in this information intensive game.

Although most organisations let people compile and test the code before inspecting it, the holly is still up over the fireplace. Why do half a dozen people suddenly get up, abandon their whizz-bang class browsing, abstract modelled, GUI development environments, and huddle away for an afternoon with paper listing, in organisation after organisation, day after day? At least get a workstation in there, so that people can ask searching questions and get hard answers to questions like, will that value always be positive by searching the source.

Code inspections are very costly, and we should aim to get the best out of them. A very good way to do this where there is a symbolic graphical debugger available is to break the job into two parts. First the individual author, who is intimately familiar with the structure and intentionality of the code uses the debugger to single step every single line and test in the program. This may sound labour intensive and low benefit, but the effects are wonderful. The program itself naturally keeps track of exactly what the designer is checking at any time, and the designer's eye is in turn focussed on each step of the logic. One doesn't have to traverse the buggy side of a test to see it, because the mind runs more quickly than the finger on the mouse button, and picks things up so long as it is pointed in the right direction. It can be useful to print a listing, chunk it with horizontal lines between functions and code blocks, and draw verticals down the side as the sections are verified. This uses the designer's knowledge, machine assist, and takes one person, while picking up a lot of problems. The full group code inspection can then focus on things that a fresh viewpoint can bring, like identifying implicit assumptions that may not be valid, as the designer explains the logic.

Code inspections structured in this way in conjunction with individual step checks have a purpose, and are less likely to degenerate into holier than thou religious wars over comment style, as too many expensive programmers currently spend time.

Coding Standards and Style Guides

Coding standards and style guides have come up several times in this course, and what has been said may well be at variance with what is commonly believed. It will be useful to explore these differences, and understand exactly what we are talking about.

We have argued that the software industry often finds itself trying to sit between two very different views of the world. The packers have been trained to structure reasoning and discourse around identifying the situation in terms of learned responses, and then applying the action that is indicated. One's conduct in the world is basically about doing the right thing. At work, one is told what to do. The mappers seek to obtain a general understanding of the situation, reusing known patterns where they are appropriate, and making up new ones where they are not. Mappers expect to act guided by their map, given an objective. At work, one understands the problem and finds an optimal solution. We have also seen that the mapping approach is what making new software is all about, and packing can't do it.

So coding standards are part mapper motivated, part packer motivated, and have their purposes confused. The mapper/packer communication barrier applies, so mappers tear their hair out as packers smear the complexity around until it is invisible on any one page of listing, citing Knowledge Packet 47684 - Clarity Is Good, while removing all clarity from the work.

If we accept that mapping and TQM are at the heart of good programming and that mapping and TQM are all about understanding and control, we can look at the goals, and see what we can do to make better coding standards and style guides.

The first point is about clarity. There is an idea around that using the syntactic richness of one's language is A Bad Thing, because it is 'complex'. Not when the compound syntax is an idiom. Not even when the idiom has been introduced and discussed in documentation. And not necessarily in any circumstances. When Newton wrote *Principia*, he wrote it all out in words, even though he could have used algebraic symbols, being the co-discoverer of the fluxions or calculus. Today we have decided that it is better to do maths with algebraic notation, even in exposition. Now it takes several pages of text waffle to say what can be said succinctly in algebraic notation in one page, and although the reading speed per page goes down, the total reading time goes up, because it's much harder to keep track of the text waffle. So should our programs be more like prose in their density of complexity per page or expression, or should they be more like maths? We suggest that if a person is new to a language, it is better that they can see the overall structure succinctly and struggle with each page for a few minutes than that they can read each idiot line perfectly and not have the faintest idea what it's intentionality is! How often have we seen keen people, sitting in front of reams of code, not having the faintest idea where to start, and feeling it must be their fault.

The second point is about conventions. Before adopting a convention, make sure it is going to gain you more than it will cost. We had an example earlier where if one didn't know what a variable was for, there didn't seem much point in it announcing it's type, but it's not just adding to the convention overhead that 'good' programmers are supposed to store as knowledge packets that is the problem. The fact is, too many conventions are just plain ugly. If one is aiming for one's minimal code to be beautiful, it is harder with great blots of ugly `gzw_upSaDaisies` littering the code. Never do anything to inhibit your team's aspirations to make a great product. There was a site where they thought that conventions were A Good Thing. Several people were instructed to Make Rules, which they duly did. One of them announced that variable names were to be limited to 31 characters. Very sensible - many compilers could only differentiate that many characters. Another announced that the sub-system the variable was declared in should be indicated by a three character alpha code at the start. Every variable, just in case anyone ever used a global. (Another announced that global variables were banned.) Another produced a baroque typing scheme that paralleled the languages own compound types, and announced that this must be included in each name. Quite why we didn't know. Another published a list of abbreviations of the names of code modules known to the configuration manager and said that this must be included in each variable name. By now they were getting 'duck's in a row crazy'. The fun really started when we realised that the total length of the obligatory stuff plus the typing of 'pointer to function returning pointer to record' exceeded 31 characters. The Law Givers simply informed us that the construct was too complex and we weren't allowed to do that, although it was integral to the architecture, and messing around declaring intermediate variables and type casting while assigning to them wasn't exactly going to help clarity or efficiency. So finally the bubble burst and we adopted some pragmatic standards that looked good and told us how to derive names quickly by establishing a project vocabulary and some acceptable abbreviations. From the mapper/packer viewpoint, we can see that the situation developed because the Law Givers were Announcing Worthy

Rules, which is a A Good Thing, but the cost, for pitie's sake, the cost...

The third point is about the nature of building bricks. A style guide that contains a multi-levelled hodge podge of naming conventions, return capture, idiom and an example module for designers to emulate will serve you better in the task of keeping good order than a collection of imperative instructions that restrict the programmer's ability to seek elegance off their own back. If you can't trust the team to know when to explicitly delimit a block and when not to, how can you trust them to write your MIS?

The fourth point is about those imperatives. There are some facilities that should never have been invented in the first place, such as UNIX `scanf()` and `gets()`. The imperatives to never use them in deliverable code are reasonable. But there are some things that you can't just get safely another, better way. And there is always the balance of clarity issue. We'll look at two concrete examples where we argue that there is a good case for using C `goto` - something you may have been told doesn't exist.

In the first, there is no other way. Imagine a recursive walk of a binary tree:

```
void Walk(NODE *Node)
{
    // Do whatever we came here to do...

    // Shall we recurse left?

    if(Node->Left) Walk(Node->Left);

    // Shall we recurse right?

    if(Node->Right) Walk(Node->Right);
}
```

So as we walk the tree, we start by making calls that lead us left, left, left, left... until the bottom. We then wander about in the tree, visiting every combination of left, left, right, right, left... and so on, until we finish our walk by doing a whole bunch of returns from our final visit that was right, right, right, right...

Every step left or right entails opening a new stack frame, copying the argument into the stack frame, performing the call and returning. On some jobs with a lot of navigation but not a lot of per node processing, this overhead can mount up. But look at this powerful idiom, known as tail recursion elimination:

```
void Walk(NODE *Node)
{
Label:

    // Do whatever we came here to do...

    // Shall we recurse left?

    if(Node->Left) Walk(Node->Left);

    // Shall we recurse right?

    if(Node->Right)
    {
        // Tail recursion elimination used for efficiency

        Node = Node->Right;

        goto Label;
    }
}
```



```

Start:  if(!OpenPort())goto Start;
        if(!InitPort())goto Start;
        if(!InitModem())goto Start;
        if(!SetupConnection())goto Start;
        if(!Logon())goto Start;
        if(!Fetch())goto Start;

```

Which is exactly what specialist scripting languages designed for this kind of job allow us to do!

Don't forget, if you want your SO to understand your love letter, you won't allow pedantries of spelling and grammar to distort the letter, and if you want your colleague to understand your program, don't twist the structure out of all recognition in the name of `clarity'.

Meaningful Metrics

We can turn the lens of practical understanding of purpose on the collection and interpretation of metrics, which many sites spend a lot of money on, so it behoves us to get them right.

There are three kinds of motives for going out and collecting numbers. All are valuable, but it is always important to understand what our motive is. The three motives are:

Descriptive Science

This involves going out and collecting data about an area to see if one can find any interesting features in the data. One needn't know what one is expecting to find; that is the point. Uncritically collected raw source data are the roots of everything. Modern entymology owes a great debt to the Victorian and Edwardian ladies that spent their time producing perfectly detailed watercolours of every butterfly and stick insect they could find. It has become something of a tradition that really interesting comets are simultaneously discovered by a professional and an amateur astronomer. Our discipline has suffered from a crude transfer of `metrics' from mass production to an intellectual, labour intensive activity. If we want to draw analogies with factories, we need to ask about what optimises the complicated human elements of our production facility. We need to spend more time on our descriptive roots. For example, do test case fails go up with code that was written in summer, when there are so many other things one could be doing instead of taking the time to check one's logic? One could brick up the windows, or factor seasonality into the site's work plan to maximise the chance of quality. What are useful indicators of quality? Internal and external fault reports per function point? KLOCS per function point?

Experimental Science

This involves making a change to an otherwise controlled environment, and seeing if the result is what we expected. It enables us to validate and improve our mental map of the workplace. It's fairly easy to do in a mass production environment, very hard in software engineering, where cycle times can be months, team members change, and no two jobs are exactly alike. One can either hire a good statistician with a real understanding of the art of programming, or look for really big wins that drown out the noise. We know there are big wins out there, because the hackers exist. This course is designed to point professionals towards under-exploited areas where big wins lurk.

Cybernetic Technology

This is where we really know what we are doing. Before we take a measurement, we know how we will interpret it, and what variable we will adjust given the value recorded. If we really had software engineering down pat, then this is what we would do. Unfortunately we don't. The field is so complex that we probably never will, but we can develop some very good heuristics. We must take care that a packer culture's need to pretend that we already have total control does not prevent us from achieving better partial control by mystifying our actions and the interpretation of such statistics as we can

get.

The pattern that emerges is, don't put the cart before the horse. If we run around collecting statistics without a clear understanding of what we are doing, an important tool is distorted into a bean counting exercise. Without wise interpretation, people become more concerned with creating rewardable artifacts in the statistics than getting the job done well and letting the statistics reflect this. This is not a vice of machine tools. Without a clear cybernetic model, 'bad' statistics become a stick to beat people with. they are bad people, and should consider their sins. This will surely produce improvement. People start having meetings where they try to count the bugs in different ways, to 'improve the situation', but the customer's program still doesn't work.

With metrics, like everything else, we can do nothing by abdicating our responsibility to a procedure.

Attitude to Tools

Whether a designer is using the socially normal packer strategy, or has got good at mapping will have a strong influence on that person's attitude to tools. A packer sees a tool as a machine that does a job, like the photocopier in the corner of the office. In fact, this is the way most of us use compilers - chuck the source in one end, and an executable pops out the other. This is usually OK, although an afternoon spent reading the compiler and linker manuals will pay for itself many times over.

Packers like big, expensive tools with complicated GUIs and incredibly complicated internal state. They promise to do everything, and take weeks to set up. There is lots of complicated terminology involved. All the razzamatazz has an unfortunate consequence. Amidst the noise, its easy to lose track of the fact that the premise of the glossy marketing brochures, that programming is a data processing operation that this product will automate for you, so you can wear a tie, smile a lot and be 'professional' is a load of baloney.

Mappers don't think of tools as photocopiers, they think of them as mind prosthetics. They are the mental equivalent of Ripley in the film Aliens, getting into the cargo handling exoskeleton to beat up the chief alien. Mappers retain responsibility for everything, and use their tools to extend their reach and awareness. Mappers don't like putting all their stuff into one tool where another can't get at it. They like their tools' repositories and I/O to be plaintext and parseable, so they can stick tools together.

Mappers think it is reasonable to write little programs on the fly to manipulate their source. They are aware of what they do well and what computers do well, using their own judgement when they examine each call to, say, a function whose definition they are changing, and the computer to ensure that they have examined every single instance.

There are some excellent mapper tools available - browsers, reverse engineering tools, even some 'integrated development environments' that are accessible from the outside. It is always worth bearing in mind however what can be achieved on most systems with just some scripts and the system editor. There was a team that became very excited when they were shown a tool that gave them all sorts of valuable browsing and cross-indexing facilities. The only differences between the tools and a bunch of scripts they already had, and which had taken a morning to type in were:

1. The tool was not modifiable.
2. The tool cost UKP 20,000 plus UKP 5,000 per seat.
3. The tool took several weeks to set up.
4. The tool had a GUI.

And when someone enthuses about what a new product can tell you, always pause to check if the right response should be, 'So what?'

Software Structures are Problem Structures

Can you imagine what Margot Fonteyn would have looked like with her arms and legs in plaster casts, and a pair of chain-gang style leg irons on her ankles? The result would not be very graceful.

One of the saddest things that able young designers do is start out with a step by step approach that enables them to start building software, and while they are doing it they get familiar with the idioms that enable them to map problem to language and approach. The whole point is that the languages and approaches they use are supposed to map to their problem domains. It is hardly surprising when they start to see and describe their problems in terms of software structures. The dividing line between a good way to describe a problem and a good way to solve it is blurred, and with object approaches and languages this intent is to maximise this blurring.

But just at the point where they could become skilled and insightful, these designers start to feel guilty, because they feel they should 'see the problem, not the solution'. So they start a peculiar performance where they pretend they can't see the very thing they are talking about. If they have a specific goal to fulfill, like maintaining implementational independence, this kind of manoeuvre can be accomplished adroitly, because they know exactly what they don't know and it becomes an exercise in rigour, but if they are just pretending to be dumber than they are, where are they supposed to stop?

If you are good, you are an asset to your organisation, because you can succinctly state that solution Y is good for problem X and why, next question please!

It is not a crime to be skilled and experienced.

Root Cause Analysis

Root cause analysis is formalised as a part of many organisations process. In it, the organisation recognises situations where things have got screwed up, and looks at what happened to understand what happened and ensure that it does not happen again. Mappers do not have a problem with this - it's business as usual. But for packers its a pretty unusual thing to do at work.

To understand what is important about root cause analysis, we can look at how a mapper does the same thing, by a different name, in requirements elicitation.

Imagine a transport firm that because of its marshalling policy has traditionally categorised all jobs as rural or urban. The rural and urban split might easily work its way into every aspect of the business process. When the software engineer turns up to identify the needs for the new system, it is important that the engineer looks at the patterns of data flow, and does not get confused by the customer's continual talk of rural and urban, which has no bearing on most of the requirements, and will need big pairs of complexity piles to import into the design.

The lesson is to see what is there, not what you are told to see. This means that you must step outside the system seen by the customer.

When performing root cause analysis in the workplace, it is important to see what actually happened, rather than expressing the events in the language of the process. In a car factory, adding a rubber stop to a conveyor belt might stop damage to workpieces, but we rarely have all the elements of the situation in front of us like the parts of an assembly line. If the events are always expressed in terms of the process, the most likely conclusion is that a scapegoat has failed to follow the process, which serves the twin packer goals of allocating blame and celebrating the perfection of the process.

In fact, causes of problems can be classified in terms of the involvement of the process as:

Unconnected

The whole team went down with chicken pox for a month. We can't do anything about this by changing the process, but maybe we can gather some interesting observations to pass to senior management about the tradeoffs in risk management.

Operational

The order should have been entered into the system but wasn't. This is often taken as the only kind of problem, but actually is the most rare. Even when it occurs, the packer tendency to then assert that the sales clerk is morally wanting and regard the problem as solved is not good enough, because we have not in fact established a root cause. Perhaps the sales clerk has been poorly trained. Perhaps the process is ambiguous. The question is why the order didn't get entered. This kind of problem can sometimes be solved by messing with the process definition, but it usually comes down to issues of morale, or identification with the job. That is, sociological effects that are powerful in the workplace but outside the domain of the process.

Ergonomic

The process is OK in principle, but the implementation is not viable. The sales clerk has to do cash sales too, and the continual interruptions interfere with the accuracy of the data input. Leave the process definition as it is but apply a little common sense to the implementation.

Procedural

The process is badly defined. Customers are invoiced for parts according to the delivery notes we get from our supplier, so customers are charged for parts that haven't even been delivered to us yet when our suppliers make an error. Change the process.

Complexity Matching and Incremental Boildown

All interesting systems from a game of backgammon to calculus, have operations that make the state more complex, and others that simplify it. Most of the formal process and study of software engineering is focussed on growing the corpus. Opportunities to shrink it, with all the benefits that brings, we must take on our own initiative. We can be tasked to write a function, but we can't be tasked to realise that it can be generalised and wipe out three others.

Great lumps of obfuscation always come in pairs, one to convolute things and one to unconvolute them. This extends to requirements elicitation, where it is common for users to request the functionality of their existing system be reproduced, including procedural consequences of the existing systems limitations, and their workarounds! These are sometimes called 'degrading practices'.

A problem that is emerging with object libraries is that objects really do a very good job of encapsulating their implementation. This allows us to use class hierarchies that we really know nothing about the insides of. So we can end up calling through several layers of classes, each of which (to take an example from Geographical Information Systems) mess with the co-ordinate system on the way, just to place a marker on a map. The difficulty doesn't emerge until we attempt to set up several hundred markers for the first time, and we discover that the 'simple assignment' we are so pleased with is so computationally intensive it takes half an hour to draw one screen.

A project that does not regularly examine its class hierarchies to ensure that the internal currencies are natural and standard, and that the design is appropriate for the use cases found in practice, can suddenly find itself in very deep water with no warning because of this hidden cost of reuse.

As ever, the best weapon against bloat is conceptual integrity, achieved by comparing mental models of the project.

The Infinite Regress of `Software Architectures'

There was a team whose customer had asked them to build a runtime environment which could support little processes being glued together by their I/O, like a kind of graphical shell which could specify complicated pipelines. If we must classify it, it was a systems programming job. This team decided to be Professional, and Use Proper Methods. So they went and produced an object oriented model of the stuff in the URD, using a tool that automates the Shlear-Mellor approach. There was a bit of difficulty with this, because the problem was wrong in places (\fIsic), but eventually they kludged it into the Proper Methodology. Then they could use the code generator! They pressed the button, and what came out was a whole bunch of glue classes. Each one just called down to an API routine from the `Software Architecture', a layer that the approach requires, that allows the application to be mounted on a particular type of computer system. In this case the Software Architecture would be a system facility providing a graphical interface to a shell supporting complicated pipelines!

As mappers, we can see what happened with this rigorous piece of packer Professionalism and Following The Procedure. The team had fired the wrong knowledge packet, and used a well-thought out and excellently automated approach and language in quite the wrong context. Shlear-Mellor is intended to capture real world application level behaviour, not for doing systems programming. In old fashioned language, they couldn't see that the problem categorisation was wrong because they had no common sense, which is either a result of moral deficiency or congenital idiocy. We would prefer to say that they were conditioned to believe they weren't allowed to think about the job directly, but were obliged to rush off and find a crutch as the first order of business. Then they could only see the problem through the approach, even though it didn't fit well. Then an application layer approach represented their problem to them in application language. So they never tried to think about what sort of program they were writing, and notice that it was a system shell, and not a real world application. To packers, Professionalism means always picking up your toothbrush with chopsticks, and never mentioning that your colleague's toothbrush has ended up rammed up his nose, albeit with full ceremony!

As things became more absurd, the tension grew, and the team became very defensive about their Professionalism. At one point, someone tried to suggest a little pragmatism, and in passing referred to himself as a `computer programmer'. The result was remarkable. The whole team started looking at the floor, shuffling its feet, and muttering `Software Engineer... Computer Scientist...', as if someone had just committed a major social gaffe. Two of the members later explained that they did not find `mere code' particularly interesting, and that their Professional Interest was in The Application Of The Process.

It's hard doing systems programming if crafting up some instructions to the computer is beneath you. It leads to very high stress, because it's impossible. So it's very important that each individual voluably demonstrates their Professionalism to management, so that none of the crap flying around in the eternal chaos that is the human condition sticks to them.

You can't write computer programs if your strategy is playing an elaborate game of pass the parcel. Not even if you are following the sublime packer strategy of passing the document to the person on your left, `for review'. Project managers that write task descriptions showing the inputs and outputs of a task can often get this kind of thing under control, but it only works provided they have respected the cognitive atoms in the project and done the chunking well, and ensured that the outputs are actually nearer the processor, either directly or by informing another part of the project's activities, than the inputs.

There is an interesting lesson for mappers in tales of this kind, too. One has to be careful of getting into games of functional pass the parcel. Complexity doesn't just go away, and functionality doesn't appear out of nowhere. Complexity can be expressed more simply, with a deeper viewpoint. You know when you've done that. It can also be cancelled out by removing another load of it at the same time. You know when you've done that too. If in your musings a lump of horrible awkwardness just seems to go away, you've probably just moved it somewhere else in your structure. Now this can be really good when it happens, because when you find it again, you've got two views on the same problem, and that can lead to great insights. But don't develop an attachment to any solution that seems to have just lost complexity by magic - it will reappear eventually and spoil your whole day. This applies at every level from requirements elicitation to bug fixing. Bugs don't just go away. The ones that pop up and then go back into hiding are the most worrying ones of all. Get the old release out, find them, and make sure they are no longer in the latest one.

For a practical example of functionality not appearing by magic, consider the problem of atomicity. On multitasking systems, many processes run without being able to control the times when they are suspended to allow another to run.

Sometimes, two processes need to co-ordinate, perhaps to control a peripheral. The problem is that it always comes down to one of them being able to discover that the peripheral is free, and then mark it as owned. In the space between the two events, it is possible for the second process to make its own check, find that the peripheral is free, and start to mark it as owned. One process then just overwrites the other's marker, and both processes attempt to access the peripheral at the same time. No matter how one juggles things around within the user processes, you can't craft up the test and set as a single, `atomic' operation out of any amount of user process cleverness. You can encapsulate it in a `GetLock()` function and tidy it away, but `GetLock()` must get the atomicity from the operating system. If the system's CPU is designed to support multiprocessing in hardware, and most do today, we ultimately need an atomic operation implemented in the instruction set, such as the TAS instruction on Motorola 68000 series processors.

None of this should be seen as suggesting that one shouldn't use layering, or specialised languages. Indeed, if one is in fact relying on a specific opcode just to grab the printer, we need some major layering! It simply means that we use the optimal level of specialisation to achieve maximal leverage - we don't just delegate the impossible to miracle boxes and proceed with a design predicated on a falsehood.

The Quality Audit

To mappers, the process is a protocol for communicating with their colleagues through time and space, not a prescriptive set of rules that must be policed. It offers facilities rather than demanding dumb compliance. In order to keep this distinction straight, we must approach the quality audit in the correct spirit.

In the default packer model, the audit is a trial by ordeal. Managers work up apprehension in the time leading up to the audit, and staff respond by booking leave and customer visits, or planning to call in sick, in order to avoid being audited. When the auditors swoop, they approach team members adversarially, and coerce them to tacitly acknowledge that the process is perfect, and that any flaws they find must be offences committed by the individual. The individual becomes the patsy that takes the flak for systemic flaws of the organisation, over which they have no control. This is the canonical situation that starts staff chewing tranquilizers before they go to work. There is no doubt that this is the model, even though it is rarely expressed this way, because the standard manager's Bogeyman Briefing includes advice such as, `Do not volunteer information. Keep answers short. If you are asked where the project management plan is, say it is in the Registry'. A barrister would sound very similar briefing a client who was facing cross-examination by the prosecution!

There is absolutely nothing in ISO 9001 that requires this sorry ritual to get in the way of improvement. We have no issue with ISO 9001 at all. In fact, we are concerned that talking about `going beyond ISO 9001' while we are incapable of even applying ISO 9001 itself with a positive mental attitude will just shift the same old stupidity into yet another `radical new breakthrough in management science'. So how would a gang of happy mappers go about a quality audit?

Firstly, the thing under scrutiny must be clearly identified as the process itself. We can do our staff the courtesy of assuming that they are doing their very best, because it is nearly always true even when they are treated like idiots. We therefore assume that any problems are by default, systemic faults of the process. It's no good blaming repeated airline crashes on `pilot error' - obviously the cockpit ergonomics need redesign.

Secondly, the comparison made by the auditor must be between the facilities of the process, and the business need of the users. One of the worst things about an adversarial audit is that non-compliances can easily emerge as consequences of a general procedure in a specific circumstance. For example, most people's process contains a clause saying that personnel records should be updated with training records. This is entirely appropriate in a semi-skilled job where workers can pick up `tickets' allowing them to perform specific tasks. In a section like a transport department, there is often an added need to retain actual certificates for legal reasons, and the problem is slightly different. In a programming team, formal training courses comprise such a low proportion of the training done in the team that they are almost irrelevant. Most training will be either on the job, or in the worker's own time. Many programmers will spend several hundred hours per year in self-training at home. Authors of `or else' based processes should remember that the employer cannot even require the employee to divulge what they have been studying at home, so they'd better not get carried away with rudeness, because the project managers really need these data, and have to ask politely. So testing dumb compliance to a global mechanism is futile and will only lead to arguments about interpretation. Instead the auditor must evaluate the local business need, and examine the suitability of the process in the light of the business need.

Thirdly, the auditor must be recognised as a specialist business colleague with his or her own positive contribution to the work to make. These people see hundreds of business needs and filing systems, both automated and manual. If the silly war between auditor and auditee can turn into a joint criticism of the process, the auditee is free to be open about their problems instead of keeping silent as advised by most managers today. Only then, when they know what the actual problems are, can the auditors search that vast experience and suggest solutions that they have seen work for others.

Quality auditors should not be bean counters whose most positive contribution is proposing elaborate rituals for balancing inappropriate complexity in the process. Their role goes much further than that. Robert Heinlein said that civilisation is built on library science, and quality auditors are now the library scientists of engineering industry. They can tell us how to file data cost effectively so that we can find them later, given the sort of thing we'll be wanting to do.

This file last updated 1 November 1997

Copyright (c) Alan G Carter and Colston Sanger 1997

alan@melloworld.com

colston@shotters.dircon.co.uk