

# The Programmer at Work

---

## Approaches, Methodologies, Languages

When we looked at a one bit program being written, we saw the need to find a mapping between the problem domain and the system semantics that fulfills the desire. Obviously, the less rich the possible set of mappings is, the easier it will be to find a useful one, assuming it exists. Any given problem domain will have its own inherent complexity, and every instance of a problem within it will have its own unique complexities. When we have a problem however, it is what it is. We can rarely change its definition to control its complexity (although sometimes it is both possible and desirable, and thus A Good Thing). So in search of leverage, the most effective way to get the job done, all we can play with are the system semantics.

At one end of this spectrum is the COTS product. Load it, run it, job done. At the other is the processor's instruction set, which allows us to organise any behaviour the hardware is physically capable of. Between these extremes are a variety of layered semantics that simplify the mapping by restricting the semantics.

In these terms, a language is any kitbag of semantics. C is a language, but so is *Excel* and so are GUI builders. The kitbag sits there but doesn't give you any clue how to use its contents. Languages are specialised by problem domain to offer greater chances of achieving simpler mappings to any given problem within the chosen domain. To decide if one wishes to make a choice of one semantics (language) over another, the criterion is usually to ask which requires the simpler mapping (the simpler program) to get the job done. Beyond the most trivial cases, this requires familiarity with both kitbags in use.

Although we can get a clear understanding of what a language is, a methodology is harder to pin down in these terms. We suggest that the reason for this is that the idea of a 'methodology', as it is commonly encountered, includes the default assumption that it is a procedural approach to solving programming problems, and we know there is no such thing. What we can describe instead for now, is something slightly different - an approach.

An approach consists of advice, given by one experienced mapper to another, about how best to tackle a kind of problem. It is an invitation to see the world in a certain way, even if it is phrased as procedural guidance. The injunction 'Draw a Data Flow Diagram showing weekly inputs' in a book called *How To Build A Payroll System*, is actually saying, 'Constrain your world-view to a weekly batch input system, and list the batches the world throws at you.'

This is sound advice for the builder of a payroll system, provided the work patterns it is to reward can fit into weekly batches. Like a language, the approach gets simpler the more it is specialised to a given domain. Also like a language, the approach is hard to select appropriately without an understanding of the 'currencies' of the available approaches, and the problem. With more clever developers writing COTS products every year, which automate an approach to form a highly domain specific language that any idiot can work, the likely future leverage for good programmers is going to be in familiarity with deep, profound approaches, and deriving new approaches in the face of new problems. There will likely always be hordes of people using the same approach to ritualise the production of the same billing system for another client. They may well be 'always retraining to new methodologies'. But they are and will remain, clerical workers, and the gap in performance and rewards between clerical workers and programmers is going to widen. This is what it means to be a player in the information age.

There are some languages that are specialised to particular approaches. Smalltalk requires the user to see the world as objects. Lisp requires an unhealthy relationship with the lambda calculus that leads to proposing the dog of food instead of feeding the dog.

The point that languages are real, and approaches are real, but methodologies are a figment of our collective imaginations and do not exist must be emphasised. Confusion on this point and an unfortunate choice of approach can lead to situations where critical parts of the problem are not addressed because the approach happens not to speak of them, while those who attempt to deal with the issues are hampered by their colleagues who feel that they are acting 'unprofessionally' by not 'applying the methodology'. This is an example of the mapper/packer communication barrier.

Interesting methodologies consist of part approach, and part language. Jackson Structured Design (JSD) constrains its domain of applicability to problems with clearly identifiable features, and is then able to offer quite detailed guidance on how to address instances of those kinds of problems. Keep your eyes open and JSD will serve you well in its domain. Outside its domain however, it can cause problems because if the problem doesn't have Jackson's features, no end of kludging will make a good system out of a bad understanding. This is not Jackson's fault, as he never said that JSD was a ritualised panacea for solving all computer problems.

At the output end of JSD we see something quite unusual, an artifact of its time. Jackson describes how to transliterate from his diagrams into code, by hand! He is clear that this is what he is doing, and explains that the automatism of this task allows us to break the rules of structured programming and use gotos. Today he would not do this - he'd just hand the diagrams over to a code generator as many others do. The point is, the diagrams of the JSD notation are best considered a programming language! Jackson has created a language that is specialised for an approach to a problem domain.

The same is true of the Booch, Rumbaugh and Unified Modelling Language approaches and languages. In fact, every interesting methodology. In Booch and Rumbaugh's earlier publications, they did not hand the diagrams over to code generators, but showed that the translation of most of the diagrams was largely mechanical. Don't worry too much for now about the methods one fills in by hand - the whole point about these is they are not complicated!

The creation of a language and approach, more or less specialised for a domain, is a great achievement. In doing so, the authors must have dwelt long on how best to navigate about problems, chunk them, explore them, see them in different ways, and designed their approach and language accordingly. But many seem to get confused by the mapper/packer language barrier, and feel the need to omit the emphasis on creative thinking needed to find the mapping between the problem and their language. Instead of presenting their approach as a structure, and suggesting some heuristics for seeing a problem in terms of it, they feel the need to use a procedural language, and describe actions to be taken, in the imperative voice. If someone hasn't been encouraged to think creatively, *ie*, construct a mental map of their problem through daydreaming and then explore it, what choice do they have but to follow this procedural misdirection, and their results will inevitably depend on luck. Jackson is good here. He specifically limits his domain and tells the reader what features to look for. The reader starts by searching the problem and looking for clues. Booch includes an interesting section on finding the objects, which if only it had gone deep and wide enough would have rendered this course unnecessary because it address exactly the right mapper issues. Finally, Stroustrup's book describing the C++ object approach and language is a celebration of style, insight, structure, depth and creativity. It is a hard book describing a complex programming language, but it is written by a great mapper at play, who seems to have no internal confusion about these issues.

## How to Write Documents

In many software engineers' view, much of their lives consist of writing documents. From the perspective of this course, we would prefer to say that their lives consist of performing work to gain understanding, which will be delivered to their colleagues according to the protocol specified in their process. Hence we are aware that the work is always understanding, and the process tells what understanding we need to convey to them. It therefore indicates the suitable language for each document. These considerations can inform a description of the actual job to be done with each of the documents; User Requirement, Software Requirement, Architectural Design, Detailed Design and Test Specification, that an engineer produces.

There are two more general points that should be made. Firstly, the job does not consist of producing reams of unintelligible gobbledegook that no-one will ever read, that look like 'engineering documents'. The first person to quote a reference, full of slashes and decimal points, in the main text when it should have been in an appendix if anywhere, wasn't just being rude to his or her readers, they were setting a trend that has devalued the whole of our art. Use simple, normal language (including specialist terminology where necessary, but not made up for the sake of it), to tell the reader what they need to know.

The second point regards format. In any stage of the software engineering process, people use understanding to find and propose pattern. If they knew what they were going to find, they wouldn't have the job, because somebody would be setting up a COTS product instead. So we don't necessarily know what the worker will need to present, so how can we tell them how to present it? Standard formats in processes should not be taken as exclusive. All decent ISO 9001 processes have provisions to tailor the required sections of a document where appropriate. Make proper use of these, and if the

structure of the document emerges during the writing, you can still put an insertion in the Project Management Plan to describe your chosen format. This is what ISO 9001 is all about.

## User Requirements Document

There has been much interest recently in 'Business Process Re-Engineering', (BPR). This is the practice of examining one's business processes to determine if they can be improved, and it often has to be done simply because the passage of time has altered the nature of the organisation's businesses. It is sometimes overlooked that software engineering has always included a significant component of BPR, because otherwise a customer will find that a computer system that automates an outmoded business process will not include the workarounds that staff will have implemented to handle change, and the system will fail. The first duty of the software engineer is therefore to help the customer understand the nature of their own requirement. In the example of the one bit program, it is the crystallisation of the desire from general discomfort to a specific need for more light. The software engineer is aided in this task by the discipline of having to write a computer program. It isn't possible to hide ambiguities in flowery code, as one can in a text report. A useful URD therefore captures as clear an understanding the user's needs as can be had at the beginning of a project, as understood by user and engineer, in the user's language. The URD will almost certainly need clarification later as the programming discipline identifies ambiguities, whether the amendments are tracked as part of the document or not.

An issue which causes a great deal of confusion here is a joint purpose that the URD has developed. From an engineering perspective, the URD must be a living document, but from the commercial and legal perspective it takes the place of a reference document for the duration of the project. The two objectives are quite distinct. When they are confused, we get the spectacle of engineers, unfamiliar with legal knowledge (such as it is) trying to write clauses out of 'A Day at the Races', while crucial issues of the business process go unexamined.

Sometimes the only way out of this is to have two documents. One specifies the contractual minimum, and may well be written solely by the customer, as some methodologies suggest. The other is a living, internal document that tells us what would 'delight the customer'. It is what we are trying to aim for on his behalf. How can we delight the customer if the only clue we have to how to do this is a something that will serve our commercial colleagues well in a court of law? The extent to which the customer should have visibility of the 'real URD' depends on commercial circumstances.

Be very careful of 'integrated feature tracking environments' that purport to capture your URD and track its clauses through design, into code, and through to test case. Such environments often forget that requirements can be met by not doing something, that several requirements may be implemented across several code segments, without any direct mapping between requirement and segment, and that it is hard to test for perfectly reasonable general requirements with specific test cases. This is not to say such tools have no use - for tasks like configuration and datafill they work perfectly. One could even track the features of a specified group of classes for GUI manipulation. But for general 'user level' black-box requirements they either distort what can be expressed in the URD, or impose a style of development that encourages long hand coding of individual features instead of performing abstractions wherever possible.

## Software Requirements Document

Where the URD describes the needed system in the user's language, the SRD describes it in the engineer's. It is in this document that system sizing calculations can first appear. Particularly with modern object methodologies, the need for an SRD has been reduced, because the architecture will consist of pragmatic classes that have a clear relationship with the language of the URD. In this situation, the SRD and ADD can be combined.

Sculptors are told to think of the completed work as residing within the block of stone or wood they are carving. It helps. In the same way, we can imagine ourselves looking over our user's shoulder, one day in the future, when our design has been delivered. As we watch them using the features of the system, we can ask ourselves, 'How must that have been implemented?' A software engineer's description of the user's needs is then easy to capture.

## Architectural Design Document

It is in doing the work captured in the ADD that the hard work of a design has to be done. It is also in the ADD that the greatest opportunity to fudge it exists. While we deliberately omit design detail from the ADD, sometimes so as to retain portability, sometimes just to avoid clouding the big picture, we must still be convinced that our design is in fact implementable. The engineer should know of at least one acceptable way to implement each feature before calling for it, and should have thought about the conceptual integrity of the collection of all the code required to implement the features.

The proposition that architectural design should not consider detailed design, we suggest is misguided. If we cannot consider implementation, we can't be very good engineers, because any fool can design the unbuildable. It is by considering implementation that we discover the limitations of our designs and learn the difference between good and bad. We are able to see alternatives, compare them and select the best. If we cannot consider implementational reality, one design is as good as another, and this critical stage of cognition becomes a typing exercise to see how fast one can 'write the document', and never mind what is written!

The ADD is a didactic document. It teaches the reader how to see the problem and solution in the way that the author sees them.

## Detailed Design Document

The DDD is a message in a bottle. It tells the reader about how the original author planned the implementation, so that the code is intelligible. The detail of exposition must take over where the ADD leaves off, and take the reader to the point where the code can stand for itself. Sometimes, this explanation can be assisted by pseudo-code, but this need not be the case. The DDD should be regarded as amendable. During implementation, design details like the organisation of code into modules will emerge. If these details are not captured for our colleagues in the DDD, where will they be captured? This simple omission causes far too much unnecessary trouble, as engineers pick up parts of systems that they can see is well documented, if only they knew where to start! Your final DDD should tell your successor whatever they need to know in order to pick up the system and change it.

## Test Plan

Test is the most context sensitive of the document types, but the following observations are useful guides within the imperatives of the situation. The test strategy aims to stress the system. It will not get much leverage out of doing this entirely at random, so the issue is to find one or more models of the system, that can give us an indication of likely typical and stress conditions. A useful structure is therefore to describe the model, derive the stress conditions, and then list them.

## The Knight's Fork

Over and over again in this course we see the echos of a deep pattern that we exposed in the writing of a one bit program. We have the problem domain, the system semantics, and a mapping between the two created by the programmer in the light of the desire. This pattern is the central act of computer programming. It may not be understanding in itself, but the ability to do this is the only evidence that one can have that one has actually understood a problem in the terms of a given semantics. If the semantics are rigorous and testable like those of a digital computer, one might claim a 'deep' or 'true' understanding, but this is suspect, because someone can always pop over the horizon and say, 'See it this way!'

This pattern is so important we want to focus attention on it. Although we have avoided fatuous jargon without any real meaning behind it, we want to introduce a term, 'The Knight's Fork', to tag this pattern. We've borrowed the term from chess. In it, a Knight sits on the board and can make a number of L-shaped moves. The other pieces are all constrained to move on diagonals or orthogonals, but the Knight's L shapes allow it to threaten two pieces, each themselves constrained to their own worlds, and thus accomplish something useful in any case.

This kind of pattern occurs over and over again, but everywhere we can track it down to the writing of the one bit program. A computer system can be in many states and evolve according to its own internal logic. the reality the computer is

following can also be in many states, and itself evolve. Because of the designer's insight, a critical aspect of the problem can be abstracted and captured in the computer, the same pattern in both cases, such that in any case, computer and reality will conform. The test cases, informed by a model of the problem and of the system, will cover the permissible (and possibly impermissible) state space of the inputs according to insight of the writer, such that in any case, the system's state evolution will be verified. The designer, looking at a need to perform data manipulation, will exploit features of the data that genuinely indicate structure in the data, and map this to features of the language, as in the canonical:

```
while((c = getchar()) != EOF)
    putchar(f(c));
```

All architectural design involves teasing apart a problem by looking at the needs from as many directions as possible, until it reveals the structure within itself that the system designer can use to defeat it..

The Knight's Fork always uses an inherent deep structure of the problem domain. Checking that a proposed deep structure is real and not just a coincidence is very important. If a designer exploits a coincidence, the result will be `clever' rather than `elegant', and it will be fragile, liable to explode into special cases provisions all over the resulting system code, with all design integrity lost. Weinberg gives the example of a programmer writing an assembler. He discovered that he could do table lookups based on the opcode number and so designed his program. But the hardware people did not hold the opcode numbering scheme sacrosanct, and when they made a valid change, the program design broke.

## The Personal Layered Process

A Zen koan tells of a wise monk who visited a great teacher. He entered the teacher's room and sat before him. `When you came in' asked the teacher, `which side of the door did you leave your stick?' The monk did not know. `In that case, you have lost your Zen'.

After you have seen the structure of your program and are ready to implement it, there is still a great deal to keep control of. Even if you can see the critical lines of code there are still a great many others to type in. The discipline required is far greater than any formal process could control, and must be applied intelligently in each new situation.

Your process will break a task down so far, and then you must take over. Like a track-laying vehicle, you must structure your work as it develops. After a while you get to the point where you can do this in your head, very quickly indeed, because you can get leverage out of two techniques.

You can only expand the part of your plan that you are working on. At one point in an activity to add a change to some source might be held in your mind as:

1. Identify all files that include functions:

```
ModelOpen( ),
ModelRead( ),
ModelWrite( ),
ModelClose( ).
```

2. Book all files out of version control.

3. Hack.

- 3.1. Change modread.c

- 3.1.1. Hack ModelOpen( )

- 3.1.2 Hack ModelRead( )

- 3.1.3. Hack ModelWrite( )

- 3.1.4. Hack ModelClose( )

- 3.2. Change appfile1.c

### 3.3. Change applile2.c

4. Book files back in.

5. Update conman system.

The fact that the process definition can't spell out every little step and so doesn't insult your intelligence in a futile attempt to do so, doesn't absolve you from the duty to do the job for yourself. And it's quite proper to leave how this is done up to you - it allows you to do the necessary organisation in your head, or any other way that pleases you. Some people like to write down little lists of files to modify on scraps of paper and cross them off as they do them, but leave the rest of the process in their heads. They can remember where they are in the big picture, but if they're interrupted in the middle of a big list, they might get confused.

The second important technique is that you can change your plans. The core concept of TQM is that we must understand what we are setting out to achieve, if we are even going to know when we have got there. This means that we need to be able to say honestly what we think we are doing at any time, but does not stop us changing our minds! For example, we might add to the example above,

#### 3.1.5. Sort out all the headers :-(

at any time as we are changing the function definitions and our bored little minds are roving backwards and forwards and realise that the prototypes will be wrong too.

We do not need to remember which bin we threw our morning coffee beaker in to have total understanding of where we are in our organisable work. Instead we can take control of the spirit of TQM and organise ourselves with full consciousness of what we are doing. As we do this, all the usual benefits of thinking about what we are doing come about. We can see opportunities to automate the boring typing with scripts and macros, and within the PLP we can always ask the question 'How would I undo this action', which is what makes people who don't accidentally delete all their source, and have to wait two hours for the administrator to retrieve last night's tape backup.

As a final comment on this topic, we often need to use a PLP to control the complexity of even the simplest job in a professional engineering environment. The ritualisation of PLP can become hypnotic. To keep proportion, always ask yourself if there is a 30 second hack that would accomplish the task, and if you can just do it, don't waste time on elaborate self-created rituals. Always keep a backup!

## To See the World in a Line of Code

We've described the central problem of software design as finding the optimal mapping between the problem and system semantics. We've also discussed the activity usually referred to as 'writing documents' as doing the necessary work and capturing the results in a document. So what is involved in doing the work that does not show up in the document? It will have a lot to do with finding the optimal mapping.

The fact is, no-one ever picks up a job, sits down and rolls out the best solution as if they were doing some sort of exam questions. The designer of an effective solution will always look at the problem from several different directions, and will usually see several variations of possible solutions. The solutions must be challenged to ensure that they meet all the requirements, and that they are going to be practical to implement. Only the winner will be recorded in the document. Sadly, the usual convention is to omit the details of why the documented solution was chosen over other alternatives from the document.

This point is particularly important when our dominant approach, usually the one that provides the basic structure of our process, involves top-down design. The idea of top-down is that it enables us to see the wood for the trees. In the early stages, we can see the overall intent of the system. We can then concentrate on getting the details within each subsystem right, knowing that its general direction is correct. This is distinct from the approach of doing top down design to remain independent of the design details of the lower levels, although the two motivations are often found together.

In both cases, the design will actually have to be implemented, so the designer will have to convince him or her self that

the design is actually implementable. If the objective is seeing the wood for the trees, there will probably be an idea around of what the target language, operating system, or in management problems the team, actually is. A criterion for a successful design is then usually optimising the use of system resources. If the objective is independence, the criterion is to produce a design that is implementable in all of the possible targets. Ideally this is done by using a model explicitly common to all the targets.

This means that the designer must have considered implementation during design, even though usual practice is to lose the implementation considerations that caused the designer to prefer one design over another.

While thinking about design, it is quite common for designers to see in their minds a high level description of the outer parts of their system, perhaps the I/O, a more detailed description of the inner parts, perhaps a group of database table definitions, and right in the middle, at the point where the key processing of the system is done, they often know just what the critical line of code, which may be quite complex, actually says. From this line they can convince themselves that the details of the outer parts of the system will be OK without having to think them all through. It's not always at the core of a design that the tickleish bits exist - the designer might notice a critical part of a low level error recovery protocol, and feel the need to know that it can be implemented. There is no better way to feel secure with what your design calls for than to be able to state at least one practical way to do it.

We are not saying that it is imperative to see lines of code popping into your head during design. We are saying that it can be a very useful way to clarify your thinking about an area, and if your thoughts do turn to code, follow them. Don't cut off these considerations because your deliverable is a higher level document. That way, you get a design document that is effective in use, and people will call you a demon wizard of the design process. Remember holding your toothbrush with chopsticks? People that are into the habit will rather believe you have a really good chopstick technique than that you just grasped the toothbrush with your fist.

Another area where little code fragments are really useful during high-level design is in getting a real sense of the system semantics that you are going to be using. We always have to learn new APIs, to our OSs, GUIs, libraries and so on. It takes years to become really fluent in all the ways we can properly use an API. So look in the books that discuss an API, and write little demo apps that demonstrate the features you think you'll be needing. This really helps concentrate your mind on what you need to keep track of from the bottom up, as your design progresses from the top down, and ensures that you don't attempt to use semantics that actually aren't there. It can be very embarrassing to produce a design that requires a different operating system design, but if you've spent a few minutes writing a little program that exercises a feature, you'll use it as it is, and never mind what the documentation claims. You win the minutes back during implementation, because you can copy bits of your doodles into your source, and hack them.

Spend a while looking at the design of the APIs you use. Look at their currencies - the values passed in and out of the API. How do the bits of the interface fit together? Are they well designed? What are the idioms that their designer was intending you to use? APIs are usually done by experienced designers, and they are like little messages from very bright people about how they see the world. The style of Ken Thompson's UNIX API has survived very well for nearly 30 years. He himself said of it that the only change he would make is 'I'd spell `creat()` with an e!'. There is something very close to the way computers work in the structure of the UNIX API

This section is all about the importance of being able to see one level below where one is working. This is true even though hiding the details of implementation is a permanent goal of our discipline. The better we get at this, the more we win, but we just aren't good enough at it yet to forget about the lower levels. Understanding where a compiler allocates heap and stack space enables you to handle scribble bugs, where we break the model of the language. Having a sense of how much physical memory (and swap) we have enables us to write programs that will work in real world situations. Even true virtual machines, such as the Java virtual machine, gives services so low level that we can trust the implementer to do it sensibly, so we can predict the efficiency of our operations.

## Conceptual Integrity

In *The Mythical Man Month* Fred Brooks emphasises the importance of conceptual integrity in design. Our deep view of programming suggests some practical ways to achieve conceptual integrity.

First, we know the importance of mental maps. If every member of the team shares a mutually-agreed mental map of the

system being constructed, then it is possible for everyone's contribution to be in the spirit of the overall design. If they don't, then it isn't, because a style guide detailed enough to allow someone to get everything right without knowing what they are doing would be much harder for the architect to write than the system itself would be.

Secondly, we have a picture of the programmer optimising a series of design choices to produce a minimal solution and control complexity. So we need to look at the kinds of constructs the programmers cook with, and ensure that they are shared. Such a project style guide indicates a coherent collection of variable naming conventions, error-handling strategies, example idioms for using the subsystems' APIs, even the comment style. One might say that by controlling the shape of the bricks, the architect can constrain the shape of the house, while leaving flexibility in the hands of the designer. The structure of the code then ensures that the code between the canonical examples is predictable and elegant. So code examples in style guides control structure, and structure controls code. Here we see another echo of the Knight's Fork - if we use the right structure, we can bring necessary and sufficient syntax into play and write minimal text. Conversely, the more twisted the stuff gets, the more of it you get, to be twisted up.

A final benefit of conceptual integrity that is very valuable to the professional programmer is very practical. Imagine you are on a roll. You've seen the way to divide up your functionality, you've got a really elegant way of catching all the odd ways the OS can signal failure, you're half way through coding up all the cases, and you need a new variable name. Your head locks up, overloaded by a triviality! The exponential benefits of getting focussed and staying that way are as great as the exponential benefits of minimising the code size, so every silly distraction you can get rid of is worth it. On sites where everyone stops work every ten minutes to argue about administration, the benefits of real focus can never emerge anyway, but where external conditions have been sorted out, having a style guide to let you pretty much derive this kind of stuff on the fly can dramatically improve effective productivity.

## Mood Control

Packers have rules of debate, that involve taking turns to score points of each other and demonstrating complete disinterest in the outcome by demeanour and language. Mappers have rules of debate too, but they are different.

Mappers are allowed to jump up and down and shout a lot. This does not mean that they are planning to murder each other, it means they are involved. They will likely go skipping off to lunch together, only to resume yelling on their return.

They will each have their own way of talking about the features of the problem, and will need to agree a common project jargon. Just doing this acknowledges the shared mental model, and focuses the group on creating and challenging a piece of group property, rather than throwing rocks at private sandcastles. Hate the sin and not the sinner!

If a colleague is saying something that you don't understand, or seems paradoxical or nonsensical, ask yourself if the person is trying to tell you about a part of the map that you are seeing in a very different way. Check what they mean by words that concern you. Start with the assumption that they have something interesting in their heads, and try to figure out what it is. This style of discussion has been thought about a lot by the Zetetics fans, that broke away from the Society for the Investigation of Claims of the Paranormal (SICOP), to investigate what the rules of evidence that would be able to test for genuine paranormal phenomena might be.

Just being a group of mappers with a shared mental model isn't enough to start modifying it together. Like everything else, we must become explicitly aware of what we are trying to do. At different times in the project, the team will need to do different things. Sometimes you will want to gather difficulties, and complicate the model. At other times organising and simplifying it will be strategic. Sometimes you will want to describe what is needed, at other times you will have to decide how to explain it to the customer.

If different members of the team have different objectives in a discussion, little will be accomplished. One member cannot construct a reasonable description of the technical issues if they are being interrupted by people who think that the goal is maximising customer acceptability.

This is not to say that all meetings without explicitly declared purposes must explode into name calling - that only happens when the randomly selected goals are mutually exclusive. But even discussions with multiple objectives can be clarified by first openly stating what the objectives are. Nor does the group focus have to be maintained with packer ritual obsessionism, because the idea is to clarify discussion, not prevent it. As ever, we must serve the objective, not

micro-police the procedure. If a team member sees something off-topic, that trashes the whole plan, they must speak up. Alternatively, if they see issues that need to be addressed but are not so critical, they can scribble them on a bit of scrap paper and raise them at an issue parade.

Mood control also extends to the overall phase of the project. By identifying particular moods and their changes, the team leader can provide structure to the teams activities, and avoid situations where everyone comes into work each day and wanders around sort of coding, without any clear understanding of what a good day would look like.

Beyond the project, the mood of the overall organisation can also have an effect on the project. A major threat can come from the way the organisation sees communication within itself. Some organisations have highly ritualised boundaries between groups, leading to a considerable amount of time being spent in self- administration. While there are plenty of forces that can grow the complexity, and hence reduce the effectiveness, of baroque administrative procedures, there are few that can simplify them. This is because only the people that connect with reality and actually do the deliverable work suffer the consequences, while others get progressively more convoluted hoops to jump through while telling themselves they are doing work.

The mapper/packer communication barrier often leads people to say that the effects of an intrusive administrative overhead are limited. There are three kinds of effect that ineffective admin can produce, at increasing levels of abstraction and hence as mappers know, power.

It takes actual work hours. Some organisations require people to fill in travel expenses forms so complex that people actually reserve a half-day a month just to fill in the forms. That's 10% of the salary and elapsed time budgets sacrificed to unchallengeable, ritual, administrative proceduralism! The data on the forms could be collected very simply, and the remainder of the clerical processing, if really necessary, could be done by clerical staff who cost less and are more abundant.

It breaks flow. It often takes several hours to actually get a problem into one's mind, and if one is constantly being interrupted by someone from Human Resources confused about their own filing system, one can work for days without ever getting to the few seconds it would take to sort things out. Pretty soon this develops into a kind of water torture, where the wretched programmer's mind veers away from thinking about the problem because every time he or she invests the emotional energy necessary to load up the hard, unstructured question to be considered, it gets blown away. This is a very unpleasant experience. People used to attach electrodes to alcoholics and give them shock when they touched a whisky bottle. It's the same thing.

It does your head in. Being a mapper involves seeking clarity and considering multiple issues. If intrusive and incompetent admin has turned the workplace into a surrealistic nightmare, keeping a focus on the high standards of clarity necessary to do programming becomes much harder, and if one can never predict how long it will take for Purchasing to acquire a software package, no planning based on it can be done.

Teams can do a lot to isolate themselves from admin chaos within their organisation, by allowing people that know the game to shield others. In the same way that a good manager shields the development team from external pressures and harrassment so that they can concentrate, a good administrator shields the team from lousy admin.

Remember that the packers in the organisation will not understand the effects described above, because they do not acknowledge the existence of the approach and state of mind with which we do programming. This is the open plan office problem!

## Situation Rehearsals

An effective way to maintain the shared mental map of the problem, the design and the group's activities is to hold regular situation rehearsals. These are short meeting where one person takes ten minutes to explain their current understanding of the group's situation. As with everything else, this is not a ritual that must be performed listlessly as part of the misery of work, it has a purpose. This means that it is worth doing a rehearsal even if not all of the team ate available, or calling impromptu ones just because some interesting people are around.

The *Sloane Ranger's Handbook* included a Sloane Ranger's map of the world. About 50% of the total surface area was

covered by Sloane Square itself, Scotland was connected to London by the thin causeway of the M1, and the major continents were squeezed into the sidelines. The point of the joke was that we all have our own distorted map of the world, but the Sloane Ranger's was particularly distorted with respect to geography. To a Sloane Ranger, it was not a joke - it was a fair representation of their world, and they argued that theirs was no more unrealistic than anyone else's. (Some of them bought the book so they could check it for accuracy. It passed muster.)

In the same way as we all have our own map of the world, we all have our own view of the problem and the group's activities. Hearing the differences in emphasis between different people's view of the problem brings more benefits to the team than just allowing the members to check that their vision at least maps to the speakers and identifying qualitative or factual differences (which the rehearsal also does). If looking at the problem from different directions brings understanding, hearing how the comms team describe the application can tell the application programmers things they never realised about their own task.

To understand why situation rehearsals are worth the disruption involved in getting some of the team together for a few minutes each day, it is useful to think about two different physical types of image storage systems. Traditional photographic plates store a different part of the total image in each part of the area of the plate. The mapping between image area and plate area is direct. Chip off a corner, and that corner of the image is lost. Holographic plates however store a transform of the whole image in each part of the surface of the plate. Chip off a corner and the image is still available, but at a lower resolution because the corner contained information about the distribution of a particular frequency component of the image.

The team need not concentrate knowledge about topics in individuals to the exclusion of all other knowledge. If it tries to do this, the results will be tragic because the team won't be able to communicate internally. The distribution of knowledge throughout the team must be more like a hologram than a photograph. I need to know a lot about my job, and a little about yours. The little I know must be true and fair, no matter how bizarrely I chose to express it from your point of view. Then you and I can talk to each other.

In situation rehearsals it is important to observe a strict time limit, or you will inevitably get bogged down. That means the speaker must have a few minutes to summarise What Really Matters, with the consequences being followed up off-line. These might be comparisons between views where team members actually disagree with the speaker, recognitions of opportunities for simplification where I learn that I'm doing something in my layer that you undo in yours, or offers of specialist knowledge.

Also remember that the model that everyone has a view of is the group model. If in the light of someone else's view of the model you can see a flaw in the shared model, attacking the model is not attacking the person whose novel approach has revealed the flaw you couldn't see on your own.

If the group can become comfortable with this Zetetic approach then an additional benefit is available from situation rehearsals. You can pick a speaker at random. This means that everyone will be motivated to run the whole project through their mind regularly, so they can be really elegant and insightful if they are picked. The effects of this can be astonishing.

When was the last time you had a job where you were required to think about your work, as you are required to make progress reports, fill in timesheets and sign off code review forms before passing them to the quality rep for initialing and filing in the project history cabinet unless it's one of Susan's projects in which case you file it under correspondence and record its existence in the annex to the project management plan to be found on Eric's hard disk?

*This file last updated 26 October 1997*

*Copyright (c) Alan G Carter and Colston Sanger 1997*

[alan@melloworld.com](mailto:alan@melloworld.com)

[colston@shotters.dircon.co.uk](mailto:colston@shotters.dircon.co.uk)