

Design Principles

Simple and Robust Environments

The development environment consists of all the tools (including word processing packages) used by the programmers, and the machine and network infrastructure they run on. A beneficial environment will be as simple as possible. Just like the software you are developing, the more complexity you let creep in, the more room for problems there will be, and the higher the maintenance costs will be.

A good general rule is to keep all your own work, including configuration stuff (in script files if need be) as plaintext. Be able to clean everything but your raw source out of the way whenever necessary, and be able to rebuild automatically.

Your repository and configuration management system's most important job is to give you security. Every bit of complexity you add increases the danger that the system will fail. The team that abdicates control to a whizz-bang client-server architecture configuration management system risks corruption due to loss of referential integrity within the system. The resultant chaos, as all work stops, efforts to find some way to have confidence in backups commence, people start to identify what they must rework, and morale disappears, shouldn't happen to a dog. It's not hard to build a good configuration management system out of system scripts using shopping lists, with something as simple as SCCS or RCS providing underlying version control.

The same logic, that the objective is security, so simplicity increases reliability and confidence when system and users are under stress, applies to backup as well. When something goes wrong, the most important thing is to know that you have rolled the system back to its exact state at a given time in the past. Incremental backups with convoluted strategies for handling changes to the directory tree can reduce confidence that they've worked OK even when they do. Or rather, they should. The team that automatically cleans down to plaintext, streams everything off to tape, and rebuilds every night knows where it is, and on this solid foundation can spend its time making real progress is actually smarter than the sophisticated team that spends a month trying to baseline itself.

Keep it simple. Never fix anything - you never know if you found all the problems. Clean down and reload instead. Always be able to reformat your disk, reinstall your tools, retrieve your plaintext repository, reconfigure and rebuild. This brings total security and saves all that time you have to spend worrying about viruses. Who cares?

System Types

One of the most important things to ask about a new system, or even an old one you have to do some work on, is what sort of system is it? One would not attempt to lay out a hippy colony around a parade ground, or a military training camp as a loosely coupled collection of teepees bound together by communal walkways! Any system may have more than one of the attributes listed below, although some are mutually exclusive. The attributes are possibly useful crude categories encountered in practical experience. They are not derived from any underlying theory. Examples of kinds of system yours might be are:

Monolithic

Centralised processing and either offline connections to users or pretty dumb terminals with all the work done in the monolith. Users are either in the same building or have a lot of leased line capacity. This is great way to do enormous amounts of commercial processing with industrial strength paper handling facilities adjacent to huge disk stores and laser printers chucking their output several feet into the air. Monoliths have their own problems - users are often loathe to go unavailable for backups for instance! These are mitigated by the high degree of control one can exert over what happens on the site. Fault tolerant hardware (including RAID and power management) and sophisticated database engine

technology are areas where competition has produced real benefits as well as hype.

Client-Server

Distributes processing upfront near the user, with work that needs centralising in one place. Provides good layering, by separating storage, processing and HCI. Allows local specialisation of functionality and multiple vendors, and hence future proof to an extent. Requires (and therefore exploits) smarter networking than monoliths, but better suited to interactive operation by allowing as much as possible to be done on processing local to the user, using processing appropriate to the user's needs. All client-server architectures actually have a monolith in the background.

Interactive

Allows the user to engage in dialogue with the system. Essential when working, say, in the near term or with the general public. These days usually means graphical. The system state evolves continuously so journalling or periods of exposure to data loss are an issue. Sizing is an issue with interactive systems because as soon as they get them, the users' use patterns change, and the use patterns can vary a long way between peak and normal.

Batch

Often thought of as an old fashioned strategy using punch cards, batch systems are simple, reliable, wonderful if communication links are unreliable, and more readily scalable than interactive systems.

Event Driven

Respond to events in the outside world. Applications with GUI s spend most of their time waiting for the user to click somewhere on the desktop, and respond to the event. One armed bandits are event driven, as are burglar alarms. Event driven systems have complex state spaces and significant danger of feature interaction. They often have an obligation to respond within a certain time limit to an event. If a problem can be represented as a non-event driven system, it is probably better to do so.

Data Driven

Similar to event driven and batch systems, but with a clear data flow through each sub-system, where the availability of the input data is the triggering event that causes each sub-system to perform its duty cycle. Data driven systems are more flexible than batch systems, because batch size can be varied dynamically, but they have the reliability of batch systems because we can always know how far we have got in processing each batch. Each subsystem can arrange an atomic operation that makes its output available, and removes its input data, so that even a powerfail at any time is immediately recoverable because system state is never ambiguous. Email systems are data driven.

Opportunistic

These kinds of systems never suffer from communications failures because they only ever use channels when they can. In fact, most business offices are opportunistic because this is how the underlying Ethernet works. Data are buffered until the local transceiver can transmit them without a collision.

Dead Reckoning

Attempt to track each step of the analyst's reality as it evolves. Often seen as desirable because they allow strong validation of user data, they can be brittle in use, leading to the famous joke, 'It's no use pointing to it - the computer says it isn't there!'

Convergent

These relax the interest in tracking each step of a perceived real world process, and focus on gathering changes of state at key monitoring points and subsequently integrating the data to form an accurate picture of the real world at some point in the past, and progressively poorer approximations as the point of interest nears the present. Mobile users who move data from their laptops to the corporate net are an example. We know exactly how many sales we made last week, nearly how many we made yesterday, but we haven't got Jack or Jill's figures yet, and so far we know of three sales today.

Wavefront

Systems that deal with things as they happen. Lighting rig control or telecommunications switching are examples. In cases of failure we are usually more interested in fast recovery than data loss.

Retrospective

Concerned with maintaining an accurate record of the past. Avoiding data loss is usually very important. Examples are accounting systems.

Error Handling - a Program's Lymphatic System

It has often been noted that one should trap one's error returns, but there is not much point in this if one does not know what one is going to do with them. Error handling is as much of your program's structure as the successful logic flows, but not as celebrated. The relationship is rather like that between the circulatory and lymphatic systems in the body. You need to consider error handling at each stage of design. For example, there is no point in capturing an error return from failing to write to the error log in an error handling routine!

Conceptual integrity requires that you define an overall approach to error handling to use throughout your design and stick to it. How will you indicate failures? What idioms will programmers use to elegantly test for errors without breaking the main flow? It should not be necessary to make a second call to find out either whether an error occurred or what it was - otherwise code will bloat. Errors should ideally be testable for on the function return to allow for terse idioms, which make for apprehensible code and beyond that, the quality plateau:

```
if((fp = fopen(...)) == NULL)
{
    // Error
}
```

or

```
if(!DoTheBusiness())
{
    // Error
}
```

One hears many excuses for bloating up the error handling logic to the point where a rigorously coded function call can take so many lines of code it isn't possible to see the big picture any more. As mappers, we know that accreting so many Worthy coding standards you can't write a beautiful program is just missing the point.

In procedural (and some object) code, there is a debate about error handling strategy that it is worth addressing. There are two approaches addressed in the debate. The first says that the called subroutine should not return until it has done what it as asked, and we might call it 'total delegation'. The other says that the called subroutine should proceed until encounters a problem, and then tidy up any mess it has made and bale out telling the caller what has gone wrong - we'll call it 'ready failure'.

The attractions of total delegation are that it makes for very clean code in the caller, it can be very efficient, and it devolves responsibility for maintaining the state of lower levels to the levels themselves. The disadvantage is that it only works if the caller doesn't need to actually handle the consequences of error using its own context. This limits it to systems programming type situations, where the application really can have no knowledge of the goings on below, and if the lower layer really can't sort the problem out, any kind of exit niceties above would be inappropriate because the OS is going to sling the process out or panic trap.

Ready failure always allows the caller to make a response to problems, and nested calls can pop their stack until they reach a layer that can deal with the problem. The error handling logic is threaded through every layer, but can be minimised with careful coding, and both author and maintainer know how to operate within the scheme. In addition, one can ensure that a call trace showing how the process got into trouble so that the situation can be reproduced is always produced.

We don't think there should be a debate because when total delegation is appropriate it is at the lowest level. Mixing the two is a nightmare in code because it breaks conceptual integrity.

Some object languages provide exceptions, which allow the automatic collapse of the call stack to a layer qualified to handle the error. These are a great way to decouple the main flow from error handling. Important points to remember are that an exception can sometimes be thrown be lot higher if you just want to handle it than if you want to know how it got into trouble. An error message from a low level saying

```
Could not write() datafile ftell() = 246810
```

followed by another saying

```
Could not Save World
```

just doesn't help with debug. You can throw exceptions up a layer at a time without compromising the main flow, and should think about doing this.

Do not abuse exceptions to create weird control flow in company time. In particular do not hide `long jmp()`s in macros and call them from handlers. If you wish to experiment with the Powers of Darkness, do it at home. We all have to do it, but it is madness after all, and your colleagues might get the wrong idea and start rationalising it. Isn't it strange that we are producing languages today that are becoming anally retentive to the point where it takes ages just to get the `const` declarations set up right in function prototypes, but allow us to pull stunts with control flow that we'd never have tried to pull off in assembler after we got just a few K to play with?

Try to avoid leaving `assert()`s and conditional compilations of debug macros littering your code. You cannot achieve the necessary sufficiency of the quality plateau with all that junk lying around.

Modalism and Combinatorial Explosion

For some reason, there is an assumption going around that in order to be robust, systems need normal modes, failure modes that they enter when they fail, and recovery modes that they get into between being in failure mode and going back to normal mode. Part of this is certainly encouraged by misguided users who are attempting to describe objectives in cases

of failure, but do so by talking about system 'modes'. It is a ticklish area, because when discussing failure users must think about the bits of a real system that can fail, and they must discuss failure early on if they have to submit a URD that can later be used as a stick to beat them. This means they must attempt to learn more about the final implementation than the designers themselves know, so that they can specify what to do when the components fail.

As well as emphasising the importance of dialogue, this points out an often overlooked point. Does the user really want you to implement the failure mode described in such detail in the URD? Might a system that just works be acceptable? Of course it would be, but many teams just go ahead and implement the failure just like the URD said.

A modern legend at ICL tells that when they bought their first load of boards from Fujitsu, they specified that there would be a 1% rejection rate. So just before the first batch of 100 shipped, a senior Fujitsu executive picked the top board out of the crate and smashed it with a hammer before repacking it.

Apart from the need to manage state transitions and execute rarely exercised code, often across distributed platforms during conditions of failure which is just asking for trouble, there is always a deeper problem with systems of this kind.

First we are in normal running. Then we enter failure mode. Then recovery mode. What happens if we now fail? Do we have failure during recovery from failure mode? Recovery from failure during recovery from failure mode? It is so easy to introduce a need for this kind of infinite regress into modal systems, and not even recognise it. Of course, if in your design every level of failure and recovery in the regress is identical then you are OK - all you have to do is prove this is the case.

If you can collapse the infinite regress then you can probably take the next step - eliminate the normal and recovery modes altogether and stay in failure mode! (Or eliminate the normal and failure modes and stay in recovery mode if you'd rather see it that way.) Then there are no co-ordinated state transitions to manage across multiple platforms while the gremlins are shoving the power leads in and out. The system need not ever know that it is under sustained real world attack and this is the fourth time it has tried to process a bunch of transactions. It need not know its context to do The Right Thing, if you have defined The Right Thing carefully enough.

Having multiple modes for handling failure really is much less necessary than most people think, and avoiding them makes huge gains in controlling complexity. If we wish to keep control and understanding of our designs, we must minimise complexity everywhere we can. On the win side of this equation is the quality plateau. On the lose side is interaction between complexity with other complexity to produce vast growth in the system's state space called 'combinatorial explosion'.

Avoid Representative Redundancy

Every database designer knows about normal forms. It ends up as a complicated area if one wants to do a thorough treatment of the subject in real world conditions, but the basic idea is very simple. Avoid redundancy in representation. If you need an order record and an invoice record, both of which need the customer's name, store the customer record in one table, and use a unique index into the customer table in the order record. Then index into the order table in the invoice record. Then things never get into a confused tangle where you end up having to remember to check loads of little things every time you want to change a datum.

The thing is, database normalisation concepts apply everywhere, for the same reasons. Never store a thing, and another unconnected thing somewhere else that asserts that the thing exists. Let data control its own structure, and it won't get tangled. Ritualistic use of data structures, often includes an aspect of pretending to be in control by picking up one's toothbrush with chopsticks. If we create the financial accounts structure in Box A, and a complicated description of Box A in Box B, we can spend a long time thrashing about in Box B and never have to address the fact that we really don't understand what is happening in Box A at all!

Don't fall into this trap. Let data represent themselves, or as Laurie Anderson said in *Big Science*,

Let X = X

Look at the State of That!

In the same way that it is important to avoid representational redundancy of data in the context of your system, it is important to avoid representational redundancy of your system in the context of the platform. This is true because global resources can be left in ambiguous states due to failure. The design should always consider cleanup of all system resources, particularly partially written files that can eat space even if they don't confuse processing.

Be aware of which system resources clean themselves up (such as semaphores) when the owning process dies and use them preferably.

Avoid 'cleanup processes' that run around non-deterministically on the system clock with slash and burn rights against all your system's resources. Try to use initialisation protocols that start by determining a known state and moving forward instead. An example might be,

1. Find an input file
2. If the output file already exists, delete input file and exit.
3. Open the input file
4. Open a temporary output file with a standard name in truncate mode.
5. Process from input to output file.
6. When output file is complete, change its name atomically to the output name.
7. Delete the input file.

Or, grasp branch firmly with left paw before releasing right paw!

The Reality of the System as an Object

This section is primarily intended for designers of object systems, because the problem it addresses primarily appears in the object approach. This is because of the rigorous encapsulation the object model affords. We have already discussed the two approaches to designing object systems that mappers and packers prefer. The mapper approach entails understanding the nature of the desire, and then as an iterated activity, identifying appropriate system dynamics and producing an optimal mapping between problem dynamics and system semantics.

Object designs are intended to create a formalised Knight's Fork, by providing an approach which explicitly relates real world objects to viable system semantics via object programming languages (be they Eiffel or UML code generators). When producing these designs their creators tend to represent everything that is in the real world today, rather than tomorrow, when the system will be in use. The major difference is that tomorrow the system will exist in user's world - today it does not. So analysts regularly produce little pictures of the user's world in the future that contains everything but the very computer system that is central to the whole scenario.

Meanwhile, the internal design of the system is also hampered by the lack of representation of the system itself. One might say that the real world system and the internal system are the same thing in both real and abstract worlds, and hence this identity forms the appearance of the Knight's Fork in its most basic form in object designs.

The two deep questions when finding objects and how they link together are:

1. Who instantiates who?
2. Who exerts who's methods?

With a clear System class in the design it's a lot easier to draw out the instantiation hierarchy, as well as see where things like GUI and tape I/O come from, let alone use cases that are triggered by wall-clock time! This doesn't mean that functionality can't be moved out into specialised classes later in the design, but it does give the user world reality an equal footing with the system reality in the design, so the result will satisfy both criteria.

Getting to grips with an abstract set of classes floating around in the air with no rigorous way to get a reality check can be as painful as anything else when you don't know what you are doing.

Of course, the need for a System class disappears if one is interesting in simply modelling, rather than automating business flows where the control systems are not represented. What would be the point in that? Here we stress the point that engineering informed by the mapping cognitive strategy involves more than a set of procedural actions. It means bounding your own problem, clarifying your own desires, and finding the optimal point of leverage between problem dynamics and system semantics. If your design won't benefit from having a System class, don't use one!

Memory Leak Detectors

There are a number of products on the market that by a variety of strategies, detect memory leaks in your application. A memory leak is what happens when a program requests some memory from the heap (using, say, `malloc()` in C under UNIX or DOS, or the operator `new` in C++), and then forgets to give it back when it is finished with it. This can sometimes damage other processes on the same platform, because some OSes will allow one process to gobble up all available system memory and swap!

Even if the OS is sophisticated enough to limit the amount of real memory it will allocate to a single process on a multiprocessor, the application can soon gobble up its own quota, which usually ends up failing in a user-visible fashion, at the very least by having the application lobbed out of the system by the OS!

So memory leaks are A Bad Thing.

This is why people sell, and buy, memory leak detectors. The trouble is, memory leaks are a symptom of problems, not a cause. It isn't hard to call `free()` or delete objects that are no longer needed. Using `delete` on collections of pointers to active objects is just plain sloppy, as is over-writing their addresses. What if these objects have, say, callbacks registered with the GUI? How will you get rid of them? Destructors out of control mean programmer out of control. If a programmer can't show control over objects enough to avoid memory leaks, how can we know that anything else is right?

Conceptual integrity is one of the strongest supports in keeping control of objects. A useful general rule (although like all rules it is not always appropriate) is to say that the layer that constructs a module should also be responsible for destroying it. This at least focuses attention on the objects lifecycle, and not just a few aspects of its behaviour that might be indicated in use case diagrams.

Timeouts

One of the most effective ways of getting a seed for a random number generator is to look at the system clock. Similarly, if two processes are running on the same multiprocessor, we can never predict just how much wall-clock time will have elapsed between their starting execution, and a given point in the program being reached. We can't even predict exactly how much processor time each will have been allocated.

Therefore timeouts are A Bad Thing. In deliverables they make the system's state space vastly bigger, so making its behaviour much harder for the designer to predict. In debugging, they can make the conditions under which the fault occurred impossible to reproduce. Don't use them unless you absolutely must.

Communications layers are often obliged to use timeouts, because when it comes down to it, the only way to find out if a remote box wants to play is to send it a message, and wait to see if it sends one back. How long should one wait? The 'Byzantine Generals' Problem' illustrates this. So most modern systems have timeouts in the comms layer, but this is not an excuse to use them all over the place, and where they must be used, they should be hidden within an encapsulated object that can be replaced by a deterministic event generator (such as a key press) for debug.

Design for Test

It is rarely enough that our systems are correct. Usually we need to know that they are correct as well. This point may sound trivial, but it has consequences for how we set about our work.

At a requirements elicitation level, we can wander around the particular part of the problem domain we are supposed to be tackling, without ever knowing if we have yet captured all the issues that are relevant. To gain confidence that we have not missed anything, we need to widen our gaze, so that we can see where our outputs go to, and where our inputs come from. We need to find a way to present these flows so that we can see the big picture at a glance. Reams of prose or fat folders full of Data Flow Diagrams are of no help at all here (although they may well be needed elsewhere on the project), because they will not allow us to see at a glance that there are no loose ends. If we can see that there are no loose ends, we can be reasonably confident that there are no hidden horrors that we will discover during implementation. This is an example of the mapper technique of problem bounding.

At an architectural and detailed design level, the same idea applies. During our contemplation of our design we represent our ideas to ourselves in as many ways as we can, and challenge them to see if we can break them. It is important that we use some feature of the design, such as the number of possible input states, to show that the system we design will be robust in all cases, by showing that we have considered all cases. This does not mean that we attempt to enumerate all cases - instead we find a means to group them, and show that we have considered all groups.

When single-stepping code with a graphical symbolic debugger, at each decision we should consider all the circumstances under which the path we are taking would be followed, and all cases where the other path would be followed.

In all these situations, design for test begins by laying our work out so that its correctness is visible to inspection. In this light it is interesting to consider what we mean by a mathematical proof. The purpose of proof is usually described as being to show that a proposition is the case. That is a very packer, activity-centred way of seeing things. The mapper description of the purpose of proof is that it shows us the proposition in a new light, in which the truth of the proposition is obvious to inspection. For mappers, a proof doesn't just establish a fact, it increases our understanding as well. We have recently seen computer-assisted proofs that fulfill the packer purpose, but do nothing for the mapper purpose. Because they do not exploit the leverage which comes from understanding, these proofs are also weaker. Is it necessarily the case that the correctness of the code (let alone the architecture of the computer) that is going to perform the search is obvious to inspection?

Wise architects usually layer their designs so that there are discrete stages visible in the transition from end-user facing code and OS facing code. Every one of these layers provides an opportunity to write a little test application. These opportunities should usually be taken, because although it may seem like a high up-front cost, tracking down bugs that do not have well-defined test points in the layering can explode the final test phase and add enormous time costs just before delivery. To fully exploit these test opportunities, we should consider test when defining the API s for our layers. Is it possible to simplify the API definition so that we can reduce the proportion of all possible calls that are meaningless? Each layer must either validate its input, or if time is really critical, require prevalidated inputs. Test must ensure that this logic works properly as well as the job the layer is supposed to be doing. If the API can be simplified, the test requirement is automatically simplified at the same time.

Considerations that apply between layers also apply between runtime processes. Most non-trivial systems require several processes to co-operate either on a single platform or across a network. The functionality of these processes should be divided up so that it is possible to test them, ideally in isolation and from a command-line or script.

Sometimes we cannot avoid introducing discontinuities into the solution where none exists in the problem. For example, if our database is so big we must spread it across several machines (and our COTS RDBMS isn't managing this for us) we need to recognise the points where the logic of our programs must change to look on another system, and test that this change is negotiated correctly.

Designers of object systems have a particularly easy strategy available for automating test. Every class (or key classes at the discretion of the architect) can have a matching class defined that exercises the methods of the system class. This works so well because the class declaration forces the surface area of the class to be tested into a standardised, and well defined format (this is what objects are all about). So each class can carry with it its own test code, that just needs calling

itself from a little application wrapper to automate the test. These test classes are sometimes called `yang' classes (the deliverable classes are the `yin' classes).

There are two benefits that can be attained when automated test is in place. The first is that the tests can be run every night, as part of the build process. It does programmers no end of good to come in and find an email from the development environment saying that everything that the entire team has developed to date is still working properly. When the email says that something is broken, they don't waste days trying to find out what is wrong with their new layer when in fact the problem has appeared two levels down. The second benefit is that automated test code cannot slide out of date as documentation can. If the automated test compiles, links, and passes, than we know that the description of the behaviour of the tested code that it contains is true.

These ideas of the definition and execution of automated tests are especially important on very sophisticated projects where dynamic configuration management and incremental compilation tools out of science fiction books allow hundreds of developers to hack away like demented monkeys on cocaine without even stopping for sleep let alone thought. (Said the authorial voice rhetorically.) Checkpointing and running full tests from the ground up should not be considered an interruption of work - it is a very cheap way of buying confidence in the solidity of the ground. As an added benefit, such events can become team festivals as module after module, layer after layer, announces its own successful build and test on the configuration manager's workstation. It is at these festivals that the team can naturally reflect on all that they have achieved to date, because the first festival should be simply to compile and run a `Hello world!' program and prove that the compiler is working properly, while the last produces a working product that is deliverable to the customer with all objectives achieved.

Dates, Money, Units and the Year 2000

An area where test (and failure) can be massively reduced by reducing system complexity is by recognising discontinuities in the problem domain and avoiding their deep representation. What does this mean in practice? One example is time and daylight saving. Time actually proceeds at the rate of one second per second, and the planet does not do a little shimmy in its orbit every Spring and Autumn. So even though the requirements document may talk about switching in and out of daylight saving, there is no need to represent this within the system any lower than the user interface level, which just needs a function, method or whatever called `LocalAdjustTime()` or some such. UNIX has wonderful support for doing this stuff right, and sadly few sites ever use it properly.

The same thinking applies to time zones. Your users may well work all over the planet and want to talk in terms of their local times, but your network should use GMT (or UTC if you really mean UTC) throughout, and files should be so timestamped. Sequencing issues with files created on computers with different clock settings still absorb far too many programmer hours. One manager of an international network went to her local domestic furnishings store and bought forty hideous, 1950s pointy-style, identical clocks and a boxful of spare batteries. Over the following year she took a clock with her whenever she visited a remote office, set it to the correct GMT time and hung it on the wall. By the end of that year, the persistent undercurrent of difficulties that were ultimately caused by sequencing issues magically went away, because every operator had a very big reminder of what time to set the system clocks to when they rebooted them.

Another example of an avoidable discontinuity of problem domain is the two kinds of money most countries like to maintain. There are always 100 pence in the pound or cents in the dollar, or just store the pence or the cents, and if the user really wants a decimal point printed out after the second digit, put a 2 in the database somewhere and use an output routine that does the database lookup. That way sensible currencies like pesetas and lira don't end up causing problems because one must remember not to print out the redundant decimal point...

The difficulties associated with the Year 2000 problem repeatedly reveal an issue that programmers seem to have to discover over and over again. Programming languages provide data types because within the type there are a permissible set of operations that one can either perform or not, and if one is reading code and the operations are being performed, one can see them. OO languages extend this facility to any kind of data we wish to so control by giving us Abstract Data Types (ADTs). The real difficulty with Year 2000 is not the way so many programmers coded the year into two digits - in days gone by that was a necessary storage saving and some Year 2000 prone software is quite old. The problem is the way some programmers chopped up their two digit dates and tucked them away all over the place, without using a consistent subroutine, macro or even code fragment to do it. That means that to dig out the Year 2000 problems one must read and

understand every single line of these horrible blathering old programs.

Security

Sites differ widely in their attitude to security. Some of this is an inevitable consequence of the nature of the business. Many military and commercial operations have a genuine need to prevent the competition from discovering what is going on. But many of the differences come from confusion as to the intent of security, and this is the topic of this section. As has often been the case in this course, situations and techniques for increasing security are given elsewhere. We shall here concentrate on appropriate relaxations of usual security.

First, one should distinguish between the security requirements of one's products, which come from the users' requirements, and the security needs of one's own development environment. These may be linked, where, for example, the security of the product depends on the confidentiality of the source code, but linkage is not equivalence. In products, don't just add `security' features by force or habit. Is it really necessary to associate a password with every user ID in your product? Do you need user IDs at all? Can non-contentious functionality be provided behind a `guest' ID that requires no password? Every password in your product must be remembered and maintained, reducing ergonomic viability and increasing cost of ownership, for the little darlings are sure to forget their passwords.

Next, there are two kinds of security threat, malicious and inadvertent. Your product may need to guard against malicious threats, but if you need to guard your own development environment against malicious threats from within (we assume you are a grown-up and have a firewall), you have bigger problems than tweaking a few file permissions will sort out for you. So give up on malicious threats at work. As for inadvertent threats, such as accidentally deleting the entire source tree, you have your backups don't you? Placing a high cost security overhead on every operation in a development environment to guard against `disasters' that are in fact low-cost when, if ever, they happen is misguided. As programmers become more familiar with the doctrine of the personal layered process, even these low cost errors reduce in number, and the development of shared mental models and mapper jargon within teams means that informal `etiquettes' develop readily, such as the cry `Reinitialising the test database - all OK?' before cleaning up trashed test data. These outcry etiquette elements are the only acceptable shouting in the hated open plan office, and just about the only valid reason for them. It is not a good enough reason however.

So don't lock up your development environment to the point where changing anything at all requires every team member present to type in their passwords. Don't create or adopt a configuration management system that stops a developer dead at eight o'clock at night when he or she is on a roll but can't even book out a hackable file for read to try something out. Not only does this directly impede your project: it is also an emotionally painful experience that you dump on the most highly motivated animal in the commercial world - a programmer in Deep Hack Mode. What has this person done to hurt you?

And finally, don't allow any element of the packer article of faith that we must know exactly who did exactly what at exactly what time with respect to absolutely everything to cloud your thinking. If your project is a team co-ordinated by etiquette and formalised as necessary you have a chance. If it is a bazaar regulated by detailed records you are doomed anyway.

This file last updated 9 November 1997

Copyright (c) Alan G Carter and Colston Sanger 1997

alan@melloworld.com

colston@shotters.dircon.co.uk