Software Enginnering Notes

Matthew Johnson

February 22, 2002

1 Resources

- newsgroup comp.risks
- R S Pressman *Software Engineering*
- N Leveson *Safeware* - what goes wrong!
- www.cs.ucl.ac.uk/staff/A.Finkelstein/

2 The Software Crisis

Earlier in the computer lifetime, it was very difficult to build hardware, and we knew what we were doing with it. Now, the harware has vastly outstripped software development. Numerically, we can't create enough programmers to write for the new hardware capacity.

Another kind of crisis is the individual project crisis. Most large software projects fail - either by abandoning it entirey, or it doesn't deliver the expected benefits, at the price expected. eg.

- Denver Taurus
- LSE Taurus
- Swanwick

There have even been projects that not only cause fiscal damage, but cause later failures, and even loss of life

- Therac 25
- Ariane
- Pentium
- London Ambulance Service

2.1 London Ambulance Service

The existing system was manual, 999 call details were all rwitten down & looked up by hand, the sorted and sent out by hand. It was quite slow, and resource intensive (3mins, 200/2700 staff), and there were problems with duplication errors and queues/call-backs. 3 mins is very fast for manual operation, but the process was well designed for automation. Correlation ofnew calls about old events was also very difficult.

In the 1980s there had been an attempt to automate it all, but it didnlt scale, and failed a load test. There were pressures to cut costs, so they decided to go for a fully automated system, who could effectively email the ambulences direct, the computer would select the most appropriate ambulence to end etc.

A consultancy service gave a deadline & costing, but said that they would need a pre-packaged system, and AVLS could not be provided.

The ambulance service, however, specified that the 1.5m, but also specified that they would need an AVLS, and an entirely new system. They also specified 11 months deadline, not 19. Out of 35 firms, most who replied said 'this cannot be done in the time', but it was awarded, to someone who said they could do it, in 1/2 the time the consultants said, and at 1/2 the price, with more features. In fact, a lot of the money went to the hardware companies.

The first problem is that the contract went to a nieve company, who had probably made a big error by taking it at that price. By 1/2 way through they realised they could not do it all in time, and had to compromise. The software managers noted by half time, that they had no formal, clear methodology, and that the LAS had no full-time representative. Software Options, the Systems Integrator, were relying on 'cozy assurances' from subcontractors

Problems arose as soon as they started testing it. There were client/server lockups, the radios have blackspots, and messages didn't get through. During shift changes, radios got overloaded. Various working practices that were in place in the manual system had not got transferred to the system spec.

In 1992 the system never got stable, but management pressured them to go live.One review at the time said there was no evidence that "...it would not be reliable". Another review, though, noted that volume testing was needed, there was no specific implementation strategy, there was no change control, and no user training. This report was ignored.

26th October 1992 - the system went live. The people were seperated out, and the people responsible for exceptions weren't doing it any more. They didn't leave a backup system, nor any network managers.

As soon as this happenned, a lot of failures caused other things to happen, that were never meant to. Error messages scolled off the screen, incidents were being delayed, which caused more phone load. Data was being congested, users weren't properly trained to use it. Most of the incidents on these 2 days either had no ambulences or multiple ones. On the 27th they switch to semi-manual mode, and on the 4th Nov, a memory leak killed and corrupted the whole system.

It was decided that they programming and testing models were fine, but it was a management, and a software engineering problem.

3 The Software Crisis

People suggested that large software projects followed the same patterns of development as large engineering projects. However, what are the differences?

Writing software is fun, and is always different. The development cycle is much shorter etc.

Writing software is very hard, however. Any small defect can pull down the who system. Large software systems are more complex than engineering projects, because the number of interations is so much greater. The ease of change make users demand more flexibility and structure changes. Software is far more abstract than physical objects, and complex systems get very hard to understand. The end, testing, phases are the most demanding and boring, and it can be tempting to skimp on them.

4 The Software Lifecycle

The cost of owning software is not just original costs, but mostly maintainance, particularly for bespoke software. In general only 10% of programming timeis spent on implementing new systems. There are different models for critical software and packaged software.

4.1 Common Difficulties

Code doesn't 'wear out' like mechanical systems, but the code can need updating for various reasons. As you 'bolt on' the code, the system gets more ocmplex, and documentation of changes is lacking. These changes can have side-effects that cause other bugs. After a while, the number of bugs start to rise, and people make the decision to recode entirely.

The price drops of hardware make it seem that the high price of software is unreasonabe, but software still costs the same to create. The two main causes of project failure are bad requirements specifications, and insufficient time or money. You can see that any individual problem is solvable, but when you combine many parts, the complexity becomes unmanageable

4.2 Reducing Costs

By the late 60's, the industry was realising that well-built software costs a lot less to maintain, and that spending time getting the specification correct, before any code is written, makes both the implementation easier, and the cost of maintainance lower.

What is the 'cost' of code? The common measure of productivity is KLOC (thousnd lines of code). IBM found that a team member would produce about 2-10 KLOC / year. This seems very low, since people can write 2K lines in a few days, but the other things that must be done take the rest of the time. A man year costs =40k salary, and another =60k on associated costs. This means it might cost 100/line of code!

There are other measures of productivity, but KLOC is resonably good, averaged over a project. However, things like high level languages and IDEs change how much each 'line' can do.

There have complained that 'men' and 'months' are not interchangable. More people imply communications complexity. In fact, adding people can reduce productivity while new people need training. Unfortunately, 3people*3months work, can't be done by 9people in 1month.

Brooks law: "Adding man power to a late project makes it later"

According to Boehm, the optimal time to release (T) is $2.5 * \sqrt[3]{M}$ where M is man months. Very fewprojects can be completed in $< 0.75T$

5 Structured Design

The only practical way of writing large systems it to break the problem up into smaller problems with independent teams running each. This enables the project managers to control the complexity.

Deciding how to subdivide the problem into modules can be very difficult, oftem there are resonably obvious hig-level components, but not always. There are various design methodologies, looking at function orientation, or data separation.

5.1 The Waterfall Model

This is the US DoD standard. *c.f. diagram*

- *Requirements* - Done by 2 different groups of people who speack different languages, from different disciplines.
- *Spec, Design, Impl.* - Done by a groups of single discipline professionals who can usually communicate.
- *Installation* - Done by people who don't undrestand the issues or problems
- *Operation* - left to people with no guarenteed understanding
- *Maintainance* - Often the original implementers have left, and possibly done by relatively inexperienced people.

5.2 Feedback

None of the steps in the model will be trouble free, so there must be feedback, and verification and validation. Validation checks that we are building the system that we are meant to be. Verification make sure what we write, does work. The waterfall method doesn't allow feedback for validation, which can leave the possibility that the wrong spec gets implemented. The waterfall method does allow for easier management, and deadlines. It also means that you can more easily charge a cient for specification changes. It is ideal where appropriate - the condition is how easily the requirements can be defined beforehand.

5.3 Pros & Cons of Waterfall Model

The waterfall method makes it a lot easier to give the project discrete milestones, which you can work towards, and can give you a more structured development. It also enables you to more easily charge for changes to the speification after it has initially been agreed, because you have had a (often contractual) milestone where you finalised the specification. There are a lot of design tools that work well with the waterfall method, and since you can clarify the goals and interfaces together it is condusive to a good programming style.

The waterfall model is ideal when it is appropriate. The critical factor that makes it appropriate is whether or not you can define the requirements to a sufficient level of detail in advance. There are some systems this works well for, but others, like HCI development, the set of requirements is not well defined in advance of proto-typing.

There are many changing factors that make the waterfall model inappropriate. In the time between specification and release a lot of things may have changed and the waterfall model allows for very little change in the specifications during the project. It is often important in software development to persue a more 'iterative' method, where you can constantly revise and improve the specifications & requirements, particularly in critical software and packaged software

5.4 A Cautionary Tale

A banking system was replacing their system, and decided to buy in a solution. They then customised it. At the end of the precess it ended up being identical to their current system, and again non-upgradable to the next version of the package. The waterfall model didn't work here, because they were starting from an existing package, not from scratch.

6 Iterative development

The iterative model keeps building and updating the system until it fits OK. This was combined with watreflal develop ment by Boehm, to get the Spiral Model. This defines a series of increasingly functional prototypes, each time analysing the risk, and checking how close to the customers expectation you are. There are 4 quadrents, each traversed multiple times.

- Determine objective, alternative & constraints
- Evaluate alternatives & risks
-
-

It is a good idea to limit beforehand the number of iterations of this loop, but not neccessarily a small number. The whole model is based on risk analysis at each step.

7 Critical Systems

Many systems have the property that certain failures must be avoided at all cost. Safety critical systems, and security critical sytems are in this category, as are software systems on which the business model of a company rely. There are a lot of comparisons between these software systems and critcal engneering problems - flight controls, bridges etc.

7.1 How can systems fail

Eg. Patriot missiles. These were used to destroy incoming scuds, but were very unsuccessful. This was because of binary truncation of timings, not all of the accuracy upgrades propogated to all of the code. There were also accumulated rounding errors, resulting in 1/3s sync errors. These were not found because the testing scheme was to short (in time) - errors hadn't built up.

Many failing systems are multi-factorial - its a lot less likely that a simple system will fail.

8 Definitions

- *Error*: an actual design flaw, or programmer error.
- *Failure*: The non-performance of a system, given some subset of the conditions.
- *Fault*: error $\rightarrow$ fault $\rightarrow$ failure
- *Reliability*: how probable is a failure within a certain period of time (sometime expressed in "mean time to failure [mttf]")
- *Accident*: undesired event that results in a loss
- *Hazard*: set of conditions of a system, which together with conditions in the environment, will lead to a failure
- *Risk*: The hazard level combined with the danger (probability that hazard $\rightarrow$ accident), and the latency (hazard exposure or duration)
- *Safety*: freedom from accidents.

9 System Safety Process

To design a safe product, you need to have a safety program plan, and involve not only the programmers, but also management, and the users. You have to identify all the hazards and risks in the sytem, then for each one decide the strategy for handling it. This includes design contraints, avoidence of the situation. If you trace the hazards to a software/hardware interface, who takes responsibility?

Having done that all the coding, all the testing, coding and proceedures relate to the plan. The documentation system needs to also be integrated, supporting certification and training etc.

It is not possible to 'retrofit' safety - it should be designed in at the start.

10 Real-time systems

Typically, systems used in monitoring or control have a maximum timedefined for operations. These are a lot harder to design and setup, and have there own specific problems. Extensive knowledge of the application is neccessary, and it can be very hard to test the timing mechanisims, since generating all possible faults, in the time that the fault might occur. Exceptions can be problematic, since many things will have to keep running, while other parts of the code are waiting for user input to resolve the exception case.

For example, the Ariane 5 rocket, a float-int conversion error caused an exception that wasn't caught, and cause random data to be dumped to the flight control memory.

11 Hazard Analysis

Many industries (eg the motor indstry) categorise the hazards, from uncontrolability, to merely distracting or nuisances. These different categories need to have different failure rates, and differing amounts of investment in eleminating them.

In a more complex system, there are more structured techniques for analysing the hazards, looking at everything that could happen to cause a hazard throught the development and use or the system (cf US Navy nuclear missile program). In fact, they look at the same fault multiple times from different angles, looking at what things might create causes for it, then looking at the possible hazards directly, and working back to eliminate causes.

12 Ways of avoiding these problems

12.1 FEMA - Failure modes & Effects Analysis

This was originally desinged for NASA and the DoD, but has been expanded for software. You start by listing all the functional modes of the system, and see what can go wrong, then quantify the worst case effects of that failure. Secondly, you look at the systems that connect the modes and transitions.

An alternative, is fault tree analysis, which starts by looking at each hazard, and trace every possible way it could fail, then eliminate all the causes. This is more appropriate for software systems, particularly real time, critical systems.

12.2 Redundancy

Some systems have multiple redundent hardware, including CPUs doing proccessing in parrallel, and they have to agree before anything is done. This is a lot more expensive, for a limited amount of increase in reliability. This is fine if the hardware is the fault location, however, software is often where things break. You can use multi-version programming, where you keep older (working) versions running, while you test new ones to ensure there are no faults in the new version, the systems may even be written completely independently, with the same spec. This is good, but the same errors can still come up, and in fact the main cause of failure is now inadequate specifications, and mistakes in that.

12.3 Kegworth Crash

A BM 737-400, one of the engines failed, but the crew shutdown the wrong one. This caused crash on landing. Initially people thought the sensors had been cross-wired, so the thought that number 2 engine *was* faultly, but infact they discovered it was probably a problem with the crew reading the new instruments.

The conclusion from this, was that fault analysis in software design should include human factors, and that should also be desinged throughout. There were also reommendations about how clear the error display was, and about having non-software problem evaluation systems (external vid cams etc)

13 Human Error Probabilities

14 Modes of Automation

There are varying levels of automation possible, from merely having the computer give advice, to 'fly-by-wire' to where the computer manages the process, and the operator just gives general limits, or high level commands. sectionMyths of Software Saftey

Extrodinary Errors	$10^{-}5$
Errors in simple tasks	$10^{-}4$
Press wrong button / read wrong display	$10^{-}3$
Dependence on situation & memory	$10^{-}2$
Highly complex task	$10^{-}1$
Unfamiliar and complex operations	$O(10^0)$

- *Computers are cheaper* - the original outlay may be less, but if you count the maintainance costs of the software, it is often the case that an electtro-mechanical system is not only cheaper, but more reliable
- *Software is easy to change* - this may be true in the abstract sense, but it is not easy to do it in a safe, reliable way - and often very expensive
- *Computers are more reliable* - unless you have an extremely extensive and expensive review procedure, any maintainance of the system, can introduce more bugs as time goes on.
- *Increasing software reliability increases safety* - even if the software does exactly what its designed too, there is a high probability that the design will have a safety critical bug.
- *Testing / Fornal verification cn remove all errors* - exhaustive testing is usually impossible, and exhaustive proofs normally take too long, and can have their own errors.
- *Reuse of software is safer* - the original specification will not match the new environment, and differing conditions uncover more errors
- *Automation can reduce risks* - there will always need to be human intervention, because the software designers will not have coded every possible outcome.

15 Tools

We use tools whenever we can't do things ourselves - cf. levers, axes. In software engineering we need tools to deal with the complexity, because we can't cope ourselves. There are 2 kinds of complexity to manage. Incidental complexity, which dominated early computing, describes when we are doing seemingy simple or trivial tasks, but the specific implementation is complex, ie coding machine code. This can be solved using tools such as high-level languages.

Nowadays the main problem is intrinsic complexity - wher the problem itself is complex. This is a lot harder to find ways of eliminating this complexity. Particularly when you need a large team to manage the project. The closest we have got to a solution are human management applications, development models and so on. We can aim to eliminate incidental compleity, but we must manage intrinsic complexity.

Incidental complexity has been managed in many ways, starting from the introduction of high evel languages, that relate more closely to the system, not to the machine implementation. This is particularly true of object oriented languges. The tedious corssrefencing for finding erros has been managed by IDEs and other such programs. The other advances that have been made have been in defining formal methods and good programming techniques, to help prgrammers manage their code.