#

#

#

09 February 2016

**How Can Software Be So Hard?**

Professor Martyn Thomas

In my first lecture, *Should We Trust Computers?,* I quoted some research results that showed that software is typically put into service with more than ten defects in every thousand lines of program source code (KLoC). Then last time, in *A Brief History of Computing 1948 – 2015,* we saw that many software development projects have to be cancelled and that few deliver all the features that were promised, on time and within budget. The transcripts of those lectures provide the details and references, and all transcripts slides and videos are on line, on the Gresham College (www.gresham.ac.uk) page for each lecture.

This lecture explores why software development is so difficult.

A major problem we face is complexity. Most useful software is very complex, for two main reasons. The first reason is that, in a complex system, it is almost always the right decision to put the complexity into the software because special-purpose hardware can be very expensive and hardware built from a lot of general-purpose components is likely to be both expensive and unreliable. The second reason why software is usually complex is the constant temptation to add “nice to have” features, because it is so easy to add them to the specification and the consequences of the added complexity only show up later.

Complexity makes every software development task harder. The statement of requirements will be more likely to contain errors, omissions, conflicts and contradictions, and it will be far harder to review and to analyse. Complex requirements lead to complex designs and to complex programs so, when requirement change which they usually do, it is far harder to see the overall impact on the project and to accommodate the changes without making errors or having to make extensive changes to work that has already been completed.

Software projects are usually important; why else would a company be willing to spend the money? So the developers need to accept responsibility for getting the software right and for providing enough evidence that it is safe to put it into service. That is why Dijkstra said:

*“It is not only the programmer's responsibility to produce a correct program but also to demonstrate its correctness in a convincing manner”[[1]](#endnote-1)*

For obvious reasons, companies need to have confidence in a new software system before they put it into service if the system is critical to the business. They will need to be sure that the software can deliver the required functions reliably, and they should also expect assurance about their important *non-functional requirements,* such as safety, security, usability, maintainability, and legality[[2]](#endnote-2). The requirements specification for software should therefore be clear about these properties and the software developers must organise their work so that they can provide adequate evidence that the essential requirements have been met.

And, as we shall see, getting strong evidence that software is fit for purpose is difficult unless you have planned how you will do it at the start of the project.

**A Simple example of Complexity**

Consider the apparently simple task of writing a software controller to provide central locking for a modern automobile[[3]](#endnote-3).

The requirements may include at least these:

• The system shall provide a convenient way to lock and unlock all the doors and boot (trunk) after leaving the car.

• The system should indicate clearly that the car has been locked or unlocked, by flashing the turn indicator lights.

• Any childproof lock settings must remain in effect when the car is unlocked.

• All the doors must be locked whilst the car is in motion or the engine is running, for safety.

• There should be a way to keep the boot locked when giving someone the ability to open and drive the car (for example, to permit valet parking).

• All doors should automatically unlock after an accident, to facilitate rapid escape, and this should override any childproof lock settings.

• The system should be secure against theft and against ‘carjacking’.

• There should be an acceptably low risk of unintentionally locking oneself out, or of becoming trapped inside the vehicle.

These requirements involve at least all the door latches, the window controllers, the indicator lights, a motion sensor, the boot catch, and some impact sensor, and the requirements interact in ways that raise questions and may conflict. Should the boot be locked when the car is stationary but the engine is running (to prevent thefts in traffic jams, for example)? What constitutes an accident that should unlock the car (should an impact to a stationary and unoccupied car in a car park unlock the doors[[4]](#endnote-4))? What should happen if the car is commanded to lock with one of the windows open (should the window close? Should the system sound a warning alarm? Should the system refuse to lock the car?). What should happen if a door is not properly closed? Or if the engine is running? These are just some of the issues that the software developers must address.

**An example of conflicting requirements and the consequences**

*Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993[[5]](#endnote-5)*

*DLH 2904 flight from Frankfurt to Warsaw progressed normally until Warsaw Okecie airport control tower warned the crew that windshear existed on approach to RWY 11, as reported by DLH 5764, that had just landed. Following the Flight Manual instructions, the Pilot used an increased approach speed and with this speed touched down on runway 11 in Okecie aerodrome. Very light touch of the runway surface with the landing gear and lack of compression of the left landing gear leg to the extent understood by the aircraft computer as the actual landing resulted in delayed deployment of the spoilers and thrust reversers. The delay was about 9 seconds. Thus the braking commenced with delay and owing to heavy rain and a strong tailwind (a storm front passed through the aerodrome area at that time) the aircraft did not stop on the runway.*

As a result of the crash, one crew member and one of the passengers lost their lives. The aircraft sustained damage caused by fire.

For more details of the accident and the factors that were determined to have contributed to it, please refer to the transcription that I have referenced or to the full accident report. For the purposes of this lecture, I shall just draw attention to the description of the braking system logic

***1.6.3 Structure and operation of braking system***

 *The Braking system consists of*

1. *Ground spoilers.*
* *If selected "ON", the ground spoilers will extend if the following "on ground" conditions are met:*
	+ *either oleo struts (shock absorbers) are compressed at both main landing gears (the minimum load to compress one shock absorber being 6300 kg), or*
	+ *wheel speed is above 72 knots at both main landing gears.*

 *2. Engine reversers.*

* *If selected "ON", the engine reversers will deploy if the following "on ground" condition is met:*
	+ *shock absorbers are compressed at both main landing gears.*

 *3. Wheel brakes.*

*The above mentioned conditions (wheel speed above 72 knots and both shock absorbers compressed) are not used to activate the brakes. With the primary mode of the braking system, the brakes may be used as soon as wheel speed at both landing gears is above 0.8 V\_0 where V\_0 is a reference speed computed by BCSU. With the alternate mode of the braking system, the brakes may be used as soon as the A/SKID-NOSE WHEEL STEERING switch has been selected to the OFF position by the crew.*

We see that the aircraft had three ways to slow down:

• the spoilers (flaps that rise on the top of the wings to disrupt the airflow over the aerofoil to stop the lift);

• reverse thrust (mechanisms that move over the engine to deflect the engine thrust forwards), and

• wheel brakes.

The spoilers and reverse thrust must not be deployed in the air, because to do so would cause the aircraft to crash. (A crash a few years ago was attributed to reverse thrust having been engaged in the air somehow). This is an essential safety requirement. The aircraft systems therefore need to detect that the aircraft has landed, and the quoted extract explains that this was done by detecting compression of both main landing gear struts, and by wheel sensors that detect that the wheels are rotating at 72 knots or more.

In this accident, it seems that the pilot banked (tilted) the aircraft into the crosswind (a normal practice to keep the aircraft lined up on the runway and therefore landed with almost all the weight on one set of wheels. The conditions to allow the braking system to have full effect were therefore delayed for several seconds until both sets of main wheels were firmly on the runway, by which time it was too late to prevent the aircraft from going off the runway[[6]](#endnote-6).

The requirement to maximise safety in the air interacts with the requirement to maximise safety on landing, so the developers have to make design choices that must allow for a wide range of circumstances. Such requirements analysis is complex even in what appear to be relatively simple systems, as we shall see with other examples in my lecture on safety-critical systems, on 10 January next year (2017).

**Requirements should be expected to change**

The requirements for a software system often change. Changes occur whilst the software is being developed, when the end users first see a version, during system integration and testing, and after it has been put into service.

The software development process has often been described using the “Waterfall model”:

 

This is often represented as a V model to show where the faults that are introduced in each step are often found.

 

*Notice the overemphasis on testing as the way to validate requirements and verify that they have been correctly implemented.*

The V model and its variants are a very unsafe and inefficient approach to developing software, as we shall see in more detail when we look at the Correct by Construction method in a later lecture. The C-by-C methods demonstrate (and ideally prove) that each development stage is fully consistent with previous stages, so that errors are detected and corrected immediately, before they can lead to erroneous further work that will have to be corrected and repeated, whereas the V model and its agile variants often find errors very late, when they are much more difficult and expensive to change.

**Why and when do requirements change?**

Changes may occur at any stage of the development or after the system has been put into service.

• If the developers analyse the stated requirements, they may need clarifications or find omissions and contradictions. Finding these changes early is good because it minimises the rework required, though it often causes problems in a commercial development where a fixed-price contract has already been agreed.

• Ambiguities, omissions and contradictions may be found at any stage of the planning, design, programming and integration.

• The customer and users may require changes when they see early versions of the system, because they see things they do not like or because they recognise an opportunity to have something that seems better.

• Changes often arise during the testing phases, either because a problem becomes apparent when the tests are being designed, or because they fail and the software has to be changed to make the tests succeed.

• Changes often arise when the system is used for real work and problems are encountered, or when a new group of users start to use it for the first time and do things differently.

• Changes will be needed when new versions of other software are implemented and interfaces change.

• Changes will certainly be needed when the business needs change, perhaps because of a reorganisation, or to provide new services, or because legislation or other external constraints have changed. When designing a software-based system it is important to foresee as many possible business changes as possible and to ensure that changing the software is no more arduous than changing the business because, if it is, the software becomes a brake on business agility and competitiveness rather than a facilitator of both of these.

In my experience of a number of failed projects and lawsuits, disputes about changes are often at the heart of the delays, overruns and technical problems that have led to the project becoming late, too expensive, and being cancelled. I have analysed several change logs for failed software projects and found that few of the changes are things that were unknown or not foreseen at the time the project started.

Let me emphasise this. The problems that often cause a project manager to lose control, and cause their project to overrun, to escalate in cost, and to be cancelled could usually be avoided by better requirements analysis.

Agile software development methods such as Scrum and Extreme Programming aim to welcome changes and to avoid them causing problems by building working software as soon as possible, with minimal functionality, and by adding new functions in a series of short developments that deliver working software frequently and involve the users in agreeing whether each new feature is fit for purpose. This approach can be very successful, so long as the new features do not require radical reworking of the software that has already been written. Agile methods therefore work best where the system is not fundamentally different from successful systems that the development team has built before, so that their early decisions about the system architecture and design turn out to be adequate to support all the features that the users require. Many web-sites fall into this category.

But agile methods fail where it becomes apparent late in the project that the architecture or design cannot support the functions that are required and that extensive rework will be necessary. Then time and cost pressures are likely to lead to compromises: either the desired functions will not be provided, or the project cost and timescales will increase substantially to allow for a full reworking of the system architecture and design, or they will be implemented with technical compromises that make the system less reliable and less maintainable.

Agile methods are also generally poor at providing strong evidence that the system has all the necessary properties – such as safety and security and reliability – because these are system level properties[[7]](#endnote-7) and cannot be demonstrated adequately through testing. The system must be designed to have these properties, and they need to be specified unambiguously at the start of the project and to guide and constrain decisions on system architecture and design, on the choice of commercial or open-source components, and the on choice of programming languages and tools.

**Software developers are optimists**

A professional should know the limitations of their expertise. Surgeons specialise, so gynaecologists do not do heart surgery, for example. Engineers mostly specialise too: naval architects don’t design aircraft and electrical engineers do not build bridges (though everyone seems to write software).

As a matter of professional standards and ethics, civil engineers and mechanical engineers will refuse to accept a contract that they believe is impossible to deliver. Chartered engineers may also incur liability for their failures.

But, through optimism or lack of experience, software developers seem happy to take on applications where they have little experience, and to accept unrealistic targets. With software, everything seems possible at the start of a new project. We like to say “yes”, hoping that things will turn out well, even though things rarely do.

**Software developers often use inappropriate standards and tools**

In every law-suit where I have been an expert witness, I have found serious weaknesses in the software developers’ standards and in the tools that they use.

Safety critical projects are the most rigorous, as one would hope and expect, but software teams that are developing business software often follow remarkably few standards in a disciplined way. Documents often lack even basic version control, and it can be difficult to establish what is the latest specification, design or plan, or whether it has been reviewed and agreed. Plans are often incomplete or out of date.

Most project risk management is almost worthless. If the project maintains a risk register at all, it is likely to omit most of the major risks or to claim mitigating actions that do not address the real risk and that have not been included as activities in the project plan.

Most programming languages contain serious weaknesses that make software development unnecessarily error-prone. Common language defects include weak type-checking, complex automatic type-conversions, lack of array-bound checking, and complex interactions between language features.

Many of the errors that these features induce can be detected through the use of an analysis tool such as Coverity[[8]](#endnote-8) but such tools may miss errors or give false warnings (although they can certainly be very useful), and most development groups still do not use them.

**Planning a software development**

There are many books that describe how to plan and manage a software development. One of the best that I know is Martyn Ould’s Strategies for Software Development[[9]](#endnote-9), which describes the approach that we taught and used in my company Praxis in the 1990s.

Firstly, understand the requirements, both functional and non-functional. Write them down, analyse them for ambiguities, inconsistencies and contradictions, review them with users and get answers to all the questions raised by your analysis. Repeat until the requirements are stable[[10]](#endnote-10).

Then, list the risks and uncertainties. What do we still not know? What could go wrong? What would be the impact if a risk materialised? What can we do now to reduce the probability or impact?

Then design the project approach to minimise the major risks. If the requirements are straightforward and the system is very similar to something that the team has done before successfully, then a waterfall or V-model or agile structure, may be adequate. There may be a need to build prototypes, perhaps to explore performance issues or to try out different user interfaces or alternative hardware solutions.

Then select the development methods and tools that suit the problem and the skills available to the development team. This *Technical Planning* should be reviewed from the viewpoints of the users, the system architect, the developers, and the staff who will have to maintain, manage and modify the software after delivery.

Next, draw up a *Quality Plan*. The Quality Plan should contain a description of everything that the project will acquire, develop or deliver, including all the software that will be delivered to the customer and all the documents and other artefacts that will be needed to manage the project, such as the project plans, progress reports, the risk register, change logs, models, designs, prototypes, tests and results etc.

Decide on the quality measures for each deliverable, and how each one will be assessed, perhaps through reviews, inspections, conformance to standards, analyses by tools, tests, or proofs. How will you know that the deliverable is fit for purpose?

The description of each deliverable and each quality control is the Quality Plan, and all the activities to produce each deliverable and to carry out quality control should be incorporated in the *Resource Plan*.

The Resource Plan should contains a hierarchic work breakdown structure that identifies the activities that must be performed with estimates of the resources that will be needed, such a programmer time, reviewer time, or access to any specialist tools and hardware. These activities can then be arranged in a network diagram that shows the constraints and interactions (for example, that one activity cannot start until another has completed).

These three plans – the technical plan, quality plan and resource plan – will usually need to be reviewed and adjusted together until an acceptable project cost and duration has been achieved (perhaps by reducing the requirements or incorporating different commercial products).

**Every Project has Risks**

The project should maintain a register of risks – typically things that are not yet well enough known (such as the delivery date for the target hardware) and known threats to success (such as the project turning out to be more difficult than estimated, or the productivity of the team being lower than had been assumed). Tasks should be included in the project plan to do work to reduce uncertainty, where this is possible and cost-effective (for example to verify essential performance assumptions), and to manage the consequences if risks materialise.

Such *contingency time* in the plan should be allocated and managed against identified risks. It is never good enough just to add a percentage to the estimates to allow for problems, though this is often done.

The project costs and timescales can now be estimated as three values: the cost and duration if *none* of the risks materialise, the costs and duration if *all* the risks materialise, and the current best estimate (which will lie between the other two values). These estimates should be kept up to date as the project progresses and the spread should narrow because uncertainty should reduce – if it does not, the project is probably out of control.

**Staying in Control**

*“As a project manager, at all times, make sure that you know what you know, you know what you don’t know, you know how you will find out the things you don’t currently know, and you know what the team will do next”*[[11]](#endnote-11)*.*

Plans should be living documents, kept current as the project proceeds, with rigorous version control.

There are five main ways that software development goes wrong …

• Ambiguous, incomplete or contradictory requirements

• Underestimated duration or budget

• Inadequate management of changes

• Incompetence, either management incompetence or technical incompetence (or both)

• Complexity – which makes everything else worse

… and there are no easy ways to recover.

Frederick P Brooks, in his classic book The Mythical Man Month[[12]](#endnote-12), observed that *“Adding people to a late project makes it later”* because the new people have to be trained and to learn about the project, and more staff inevitably increases the number of interactions between staff members which reduces productivity.

When a project overruns, you may reduce the functionality, work harder and longer hours, cut out some planned activities (often people reduce testing!), or start again. None of these actions will return the project to its original schedule and budget and deliver the planned functionality; indeed, reducing testing (or any other quality control) is highly likely to lead to much greater problems later. Nevertheless, this is the easiest thing to do and it is still very commonly proposed, as this extract from a recent US Pentagon memorandum published in the USA about the software for the F-35 aircraft shows. The memo contains this paragraph

*The current "official schedule" to complete full development and testing of all Block 3F capabilities by 31 July 2017, is not realistic. It could be achieved only by eliminating a significant number of currently planned test points, tripling the rate at which weapons delivery events have historically been conducted, and deferring resolution of significant operational deficiencies to Block 4. In fact, I learned very recently that the program is currently considering reducing by two thirds the number of planned weapons delivery events (per the approved Test and Evaluation Master Plan) for weapons certification. This course of action, if followed, constitutes a very high risk of failing Initial Operational Test and Evaluation (IOT&E).*

**So how do teenagers write software that makes them rich?**

The answer is simple.

• They have a great idea.

• They move very quickly to write the software.

• The programming team is very small (perhaps one or two people) and they really understand the target market.

• The requirements are initially quite simple, and well understood by the programmers.

• They can just get on and create the software: no-one else has to be consulted and there are few constraints.

• They can distribute the software widely through social media, because they do not need to charge for it: the value will come later from the number of active users.

• They can release buggy software because it’s free and the users will tolerate errors if the software offers them something useful enough.

• They can ignore issues such as security because the software is free and they carry no liability.

• Most importantly, they can fail, because they have no reputation to lose and (just as with lottery tickets) it is the very few successes that make news, not the very many more failures.

This is *innovation* not *engineering,* but it can be very successful in the few cases that achieve major success before a competitor can steal the market. It works because the developers do not face the complexities, constraints, costs and liabilities that are inherent in software developments that are business critical, safety-critical, or security critical. Such critical systems need rigorous systems engineering and software engineering. Unfortunately, most of the software industry does not use rigorous engineering, and this is a problem that we need to solve urgently.

© Martyn Thomas CBE FREng, 2016

1. Edsger Dijkstra (EWD249) [↑](#endnote-ref-1)
2. Legality will include issues of copyright, patents, data protection and licensing, but there are other considerations that depend on the application. [↑](#endnote-ref-2)
3. I am indebted to Michael Jackson, FREng, for this example.

M A Jackson *Where, Exactly, Is Software Development?* <http://www.ifi.uzh.ch/seal/teaching/courses/archive/fs14-2/ase14/Jackson2006.pdf> [↑](#endnote-ref-3)
4. This has been used by thieves as a way to unlock a locked vehicle – with a well-placed kick on the bumper (fender) triggering the airbags and opening the doors. [↑](#endnote-ref-4)
5. Extracted with light editing from a transcript of the accident report. See <http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html> [↑](#endnote-ref-5)
6. This is an incomplete and inaccurate summary of a complex accident investigation. [↑](#endnote-ref-6)
7. Sometimes called *emergent properties* because they emerge from the overall system and may not be properties of the individual components. John Conway’s *Game of Life* is a good example of emergent properties. See http://www.bitstorm.org/gameoflife/ [↑](#endnote-ref-7)
8. http://www.coverity.com/products/ [↑](#endnote-ref-8)
9. Martyn A Ould, *Strategies for Software Engineering, the Management of Risk and Quality,* John Wiley & Sons, 1990 [↑](#endnote-ref-9)
10. Michael Jackson has written two excellent books on requirements: *Software Requirements and Specifications* (1995) and *Problem Frames* 2001). Both books are published by Addison-Wesley. [↑](#endnote-ref-10)
11. Martyn A Ould, *Strategies for Software Engineering* Chapter 7. [↑](#endnote-ref-11)
12. Frederick P Brooks Jr, *The Mythical Man-month,* Addison-Wesley 1975 and (Anniversary Edition with four additional chapters) Addison-Wesley 1995 ISBN 0-201-83595-9

Gresham College

Barnard’s Inn Hall

Holborn

London

EC1N 2HH

www.gresham.ac.uk [↑](#endnote-ref-12)