

# White Box Software Development

Dewi Daniels, Richard Myers and Adrian Hilton  
Praxis Critical Systems  
20 Manvers Street, Bath BA1 1PX, England

## Abstract

This article attempts to debunk the populist view that building high quality software is difficult and costly, and that having software systems that crash is an acceptable state of affairs.

The technology to build predictable reliable software systems exists today. Principled engineering judgment can be used to tailor software development so that quality can be built in with cost in mind – this is particularly the case with safety critical systems, where the application of standards can force an unnecessarily rigorous approach for little proven benefit.

This article explores the general poor quality of software “in the large”, the public’s (and the industry’s) view that this is in some way acceptable, and then presents some real case studies which show how quality can be built in without the need to invest in overweight tools and technologies.

## 1 Introduction

*The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive world-wide distribution of bug-ridden software for which we should be deeply ashamed.*

Edsger Dijkstra, Communications of the ACM, vol. 44 no. 3, March 2001

### 1.1 Poor Quality Is The Norm

Software is generally sold with little or no meaningful warranty. The media is often the only part of a package with a guarantee. The contrast between the end-user licence agreements for software and hardware is shown in the program licence agreement from Compaq (Compaq 1999):

Compaq Computer Corporation warrants the media on which the programs are furnished, to be free from defects in materials and workmanship under normal use for a period of one year from the date of delivery to you [...]  
YOU ASSUME THE ENTIRE RISK AS TO THE QUALITY, USE AND PERFORMANCE OF THE PROGRAMS. SHOULD THE PROGRAMS

PROVE DEFECTIVE, YOU – AND NOT COMPAQ OR ITS SUPPLIERS  
OR AN AUTHORISED RESELLER – ASSUME THE ENTIRE COST OF  
NECESSARY SERVICING, REPAIR OR CORRECTION.

## **1.2 The Industry Has Accepted This Situation**

Software projects are frequently too late, too expensive and unable to meet the customer's real needs. In the three-way trade-off between time, cost and functionality, software often fails to deliver any of the three.

The natural result of this is that the industry no longer believes that it is possible to deliver adequate software on time and to budget. The consequence is a widespread practice of attempting to improve software quality solely by finding and fixing faults in finished artefacts, whether units or systems.

*I've said before that the best way to find bugs is to execute the code and then somehow spot them...*

Steve Maguire, "Writing Solid Code", Chapter 4, Microsoft Press, 1993

## **1.3 White Box Software Is Different**

We might call this generate, test and rework paradigm "black-box software development" because it is fundamentally unconcerned with the internals of the software development process. Building-blocks are assembled or re-assembled, and getting it out of the door is all that matters. A "one size fits all" approach is taken to both the product and process; no assessment is made of whether some parts of the product may be better served by a different process.

By contrast, an approach in which specific attention is paid to the product being built and the development process being used might be dubbed "white-box". This approach tailors the process and methods used to the success criteria of the project and the heterogeneity of the product.

## **1.4 Structure**

In this paper we review the current state of the industry, and then argue that the low expectations of software are the result of three fallacies:

- that software is inherently complex and error-prone;
- that tools or methods will fix this problem; and
- that Software Engineering cannot be a true engineering discipline.

We then show that it *is* possible to produce high quality software on time and on budget by applying sound engineering principles throughout the life-cycle. There is no single method for doing this – there is still no silver bullet (Brooks 1995). However there are common themes:

- deploy small, highly-skilled teams;
- use tailored, focused processes; and
- choose the appropriate tools for the job.

## 2 State of The Practice

*It's hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn't a book on software design.*

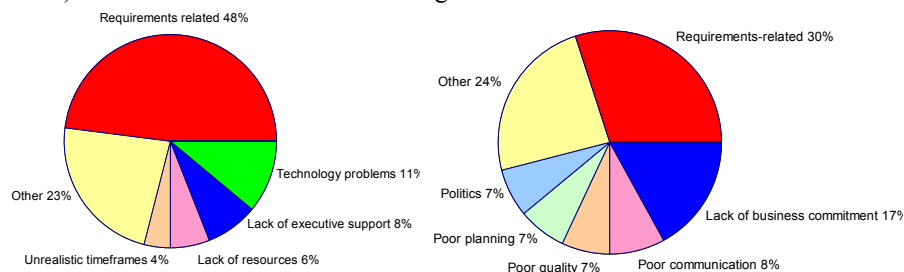
Bruce Tognazzini, "Principles, Techniques and Ethics of Stage Magic and their Application to Human Interface Design", Proceedings of INTERCHI, April 1993

### 2.1 The Costs of Failure Are High

A recent report from the American National Institute of Standards and Technology suggests that software faults cost the US economy some \$40bn to \$60bn annually (Tassey 2002). The costs to the European Union will be of a similar order. About 60% of these costs are borne by the customer. The rest are borne by the vendor. The report surveyed software developers and users in the automotive, aerospace and financial services sectors. The cost estimates did not include "mission critical" software.

### 2.2 The Causes of Failure Are Well Understood

Most IT projects fail (i.e. are cancelled before completion) because problems which should have been dealt with at early stages are not detected until it is too late to correct them. The common causes of IT project failure are revealed by two studies. The first was conducted by the Standish Group in the USA (Standish 1995). The second, smaller, study was conducted by Taylor in the UK (Taylor 2001). The results are summarised in Figure 1.



**Figure 1. Undetected faults in early life-cycle stages cause project failure.**

*The reported causes of failure are most often requirements-related and yet these projects often fail after considerable time and expense. The truth is that requirements problems are left undetected until late in the life-cycle at which point they cannot be rectified.*

*Based on data from (a) (Standish 1995) and (b) (Taylor 2001).*

The 1994 Standish Group CHAOS report found that:

- Some 31% of projects failed at a cost of \$81bn.
- An additional 53% were troubled.
- Only 16% were successful.

The mean time and budgetary overruns were both around 200%, with just over half the original functionality actually delivered. Almost all of the perceived causes were upstream of implementation.

Taylor's study was based on detailed questioning of 38 members of the British Computer Society, The Association of Project Managers and The Institute of Management. Of 1027 projects studied, only 130 were successful, of which:

- 2.3% were development projects;
- 18.2% were maintenance projects; and
- 79.5% were data conversion projects.

Out of over 500 development projects in the sample, only 3 were successful. About half of the projects which failed were cancelled prior to implementation.

A recent update to the CHAOS report (Standish 1999) indicates that by 1996 the proportion of successful projects had increased to 27%, but that this had stayed the same up to 1998, suggesting that the project success rate had reached some form of limit which left 3 out of 4 projects still not delivered to acceptable levels of quality within timescale and budgetary constraints.

## **2.3 But The Failures Keep Coming**

### *2.3.1 Delivered Systems*

Expensive software errors in delivered systems are not new. The first famous case was the loss of the Mariner 1 probe to Venus in 1962 as a result of an error in a FORTRAN statement: the programmer had typed a “.” instead of a “;”.<sup>1</sup>

In the mid-1980s at least six people were exposed to massive doses of radiation as a result of race conditions in concurrent software for the Therac-25 radiation therapy machine (Leveson 1995). The software had been re-used from an earlier model, but the errors had not been detected because the earlier model had hardware interlocks to prevent overdose.

In 1991, 28 soldiers died at Dhahran following the failure of a Patriot missile to intercept an incoming Scud. The software had never been designed to run continuously for long periods and had a cumulative rounding error in it. At Dhahran the Patriot system had been left on for 100 hours and the cumulative error was enough to cause the tracking system to fail.

---

<sup>1</sup> The use of a programming language where a single-character error can change the semantics of the program is clearly inappropriate for such a high-cost enterprise. In 1962 they probably had nothing better. However, today we find C and C++ used in a variety of mission-critical settings, where confusing = and == may have similar results.

In 1996 the Ariane 501 launcher was lost with payload at an estimated cost of \$1bn. The cause of the loss was a run-time exception in a software unit which had been re-used from Ariane 4 but was, ironically, not needed in Ariane 5 (Lions 1996).

Most recently, in 1999 the \$130m Mars Climate Orbiter collided with Mars due to an embarrassing confusion between metric and imperial units (Stephenson et al. 1999).

The apparent simplicity of these errors belies the complexity of the underlying causes. None of these accidents could truly be described as the result of a simple programming error, although software failure was the proximate cause of each.

### *2.3.2 Cancelled and Late Projects*

Cancelled projects, or those which are delivered late at excessive cost, are more significant in terms of cost because they are more common. One example was the 1990 Wessex Health Authority Regional Information Systems Plan, which cost the Authority £43m (Committee of Public Accounts 1993) before being abandoned. Poor project management exacerbated the result of the software failings, perhaps by up to £20m.

The Bowman project to provide the British Army with a secure communications system has suffered over 75 months of delays and went £200m over budget (Hansard 2000). The lack of this capability could leave forces vulnerable to electronic counter-measures and unable to use secure lines of communication.

## **2.4 And We Just Accept It**

The most outrageous aspect of this situation is that it has been accepted as the industry norm. The difficulty of producing a high-quality product (and it is difficult to engineer good software) should inspire us to strive harder rather than just give up. Unfortunately the latter is what appears to have happened. We believe that tolerance of this sorry state is the result of the three widespread fallacies which we discuss in the next section.

## **3 Three Widespread Fallacies about Software Development**

This section describes and refutes the three fallacies which we believe lead to unjustifiably low expectations of software development:

- that software is inherently complex and error-prone;
- that tools or methods will fix the problem; and
- that Software Engineering is not a true engineering discipline.

### 3.1 Fallacy: Software Is Inherently Complex

*There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other is to make it so complicated that there are no obvious deficiencies.*

Tony Hoare, ACM Turing Award Lecture, 1980

The first fallacy is that we should expect software to be unreliable because it is complicated. The truth is that software is complex mainly because we choose to make it so. The industry has become used to creating complex software solutions where simple ones would do. Complex software arises because:

- software construction begins prematurely;
- it is easy to achieve partial success at the expense of complexity; and
- writing simple software is difficult.

Premature commitment to software creation is a pervasive problem in the industry. The pressure to be seen to be making progress, coupled with the desire to get coding means that software artefacts are created before the problems they address are adequately understood (Leveson 1995, McConnell 1999). This tendency contributes to the large number of failures attributed to “requirements-related” causes. If software has been created before the requirements are well understood, requirements changes will be difficult to accommodate and the final product will be expensive, late, incomplete and/or defective.

It is easy to create complex software which works most of the time, but this is at the expense of unmanageable complexity which renders further improvement impossible (Leveson 1995). Repeated updates to software to fix bugs will tend to increase the software’s complexity (the principle of “software entropy”), eventually to a point where the software is too complex to change without breaking existing functionality. Such software is often termed “fragile”.

The difficulty of writing simple software is not immediately obvious. In the physical world, the properties of raw materials provide natural constraints on modifications. Building a physical machine involves purchasing and assembling components and possibly tooling-up a factory. There are few such obvious constraints on software: changing it appears to be a simple matter of rewriting some text. The traditional constraints of memory, processor speed and turnaround time no longer apply, so programmers are free to create code in an ad-hoc manner. The only requirement is that the code compiles.

Such an approach ensures that only the software’s author can understand or fix it, which carries a certain attraction for some programmers. The fact that *no-one* understands the code becomes apparent as attempts to fix the remaining defects prove futile. Unfortunately for the customer, the “heroes” who create dazzlingly complex software so quickly and then fight so valiantly to fix the bugs are often better rewarded than those who take the time to get things right.

The result of the fallacy of complexity is that customers have come to regard rapid fixes as the pinnacle of quality; not introducing the fault in the first place is

less well regarded (Stålhane et al. 1997, Chulani et al. 2001). In fact, in software development as in so many other activities, it is the mark of a skilful practitioner that they make things look simple. Elegance and simplicity are hard-won but the end result looks trivial to the inexperienced eye.

### **3.2 Fallacy: Tools and Methods Will Fix The Problem**

*It is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.*

Edsger W. Dijkstra, “How do we tell truths that might hurt?”, June 1975

The second fallacy is that tools and methods alone will improve software quality. We are not arguing that tools and methods are useless, but that they are no substitute for skill and experience. Software tools and methods are often expensive to deploy and require specialist knowledge to use and support. The inappropriate use of a method may hinder rather than assist a project. Several of our clients in aerospace, rail and telecommunications have experienced significant difficulties stemming from the inappropriate use of methods and tools.

#### *3.2.1 No Silver Bullet*

Brooks’s 1986 prediction of “no silver bullet”, that no single Software Engineering development would yield an order of magnitude improvement in programming productivity in ten years, was borne out by events (Brooks 1986, Brooks 1995). Indeed, many commentators questioned whether all the developments taken together over that period had led to an order of magnitude improvement.

No language is a substitute for programming skill. Similarly, no requirements, design or testing tool can eliminate the need for engineering skill and experience.

#### *3.2.2 Everything Old Is New Again*

Many of the significant “developments” of the 1980s and 1990s were not actually new. Object oriented programming was first introduced in 1967, and program proof was first mooted in 1946. Many of the project life-cycle models used today date back to the 1970s. Fashions may change but fundamentals do not.

Worse, modern re-inventions of old concepts may not incorporate the lessons which were painfully learned by the adopters of the original idea. An example is the confusion between *abstraction*, which excludes unnecessary detail, and *information hiding*, which is the removal of state from visibility. The latter has been the focus of object-oriented programming, whereas the former should have been the focus (Amey 2001).

The effect of information hiding is that state and the way in which it changes can become complicated without the programmer realising. Abstraction makes things less complicated by hiding unnecessary detail, but information hiding

increases complexity by brushing it under the carpet. Devising a useful abstraction is a highly skilled activity; stuffing state into the private part of a class is not.

### 3.2.3 Hidden Costs

Software tools may appear to save effort at a particular life-cycle stage, but this may be at the expense of other life-cycle stages. For example, automatic code generators may reduce the effort required in the coding phase. However, the effort required for design may increase correspondingly because the designer must provide additional information to direct the code generation.

Code generators may also have a negative impact on the effort needed for verification and validation. Since this is often 60-70% of the total effort, code generation tools should be judged by how much they simplify verification and validation rather than by how much coding time they save.

The idea that tools somehow encapsulate prior knowledge is a common but dangerous misconception. Its implication is that the software process can be de-skilled and effectively turned into a production line staffed by large teams of relatively unskilled technicians. Programmers did not become less skilled with the advent of compilers. Compilers enable programmers to shift their attention from the machine domain to the problem domain where it is needed. The inappropriate use of tools causes attention to shift back from the problem domain to the tool domain.

## 3.3 Fallacy: Software Engineering Cannot Be A True Engineering Discipline

*Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built aeroplanes – build the whole thing, push it off the cliff, let it crash, and start over again.*

Professor R. M. Graham, Massachusetts Institute of Technology, from the debate “Software Engineering and Society”, NATO Science Committee conference on Software Engineering, October 1968

The third fallacy is that Software Engineering cannot be a true engineering discipline. It is certainly arguable that current practice in software development would not be acceptable in many engineering disciplines. Our position is that current practice should be unacceptable in the software industry also.

This fallacy is often expressed as “programming is an art, not a science”. However, there is much more to Software Engineering than programming. The fact that very few project failures are directly attributed to bad programming and relatively few to poor project management suggests that the problem lies in between these two activities. That is the place of Software Engineering.

There is a substantial body of knowledge in Software Engineering, dating back over thirty years. The first conference on the subject was in 1968, and the above quotation from that conference is still true thirty four years later.

Three examples distinguish Software Engineering from programming. The first is the 1996 Ariane 501 failure (Lions 1996). The immediate cause was a run-time exception generated by a piece of legacy software from Ariane 4. The software function was not actually needed in Ariane 5 and could therefore have been removed. The second is the loss of the Mars Climate Orbiter in 1999 due to a data file containing imperial rather than metric data (Stephenson et al. 1999). Both of these expensive and very embarrassing losses stemmed from software failures which were nevertheless *not* simple programming errors. They were, respectively, the failure to understand the impact of a domain change and the failure to understand an interface.

The third example is the Therac-25 overdosing failure (Leveson 1995). Here, the designer used software to maintain safety, instead of using the hardware interlocks present in the older Therac-20 machine; a software fault that killed and injured a number of patients treated by Therac-25 was also present in Therac-20, but the latter's hardware interlocks

Both the academic and professional worlds accept Software Engineering as a discipline. Many universities throughout the world offer undergraduate and postgraduate degrees in Software Engineering. Certification and licensing of software engineers is under study in the USA, notably in Texas and Illinois. Canada licenses software engineers, and the Canadian Council of Professional Engineers has advised Microsoft that unlicensed MCSE and MCPSE holders who call themselves "engineers" could face enforcement measures. The Engineering Council of the UK recognises that software engineers may be awarded Chartered Engineer status after appropriate training and experience.

We believe that software development processes are generally more mature than digital electronic design processes. Many functions which would once have required custom hardware are now implemented by programmed devices. The circuit design is expressed in VHDL source code. Leveson's "curse of flexibility" (Leveson 1995) now applies to hardware as well as to software. We have found on a number of recent projects that the software development has proceeded more smoothly than the hardware development. On one project, the software development was completed within a few months but hardware problems delayed the project by nearly a year.

We have found that projects in more traditional engineering enterprises such as the railway industry suffer from requirements-related problems. It has been our experience that, far from not being a true engineering discipline, Software Engineering has lessons to teach other disciplines.

### **3.4 Summary**

These fallacies about software development are understandable, but demonstrably incorrect. Software need not be complex and unreliable; however, to make it

reliable tools and methods are not enough. We need to treat Software Engineering as the engineering discipline that it should be.

Now that we have examined and refuted these fallacies we provide practical suggestions for improving software quality, based on our own experience of software development projects.

## 4 White Box Software Development

*The advantage of being correct is that you do not need to change your mind.*

J. K. Galbraith, BBC interview, 1996

Our fundamental philosophy is that it is quite possible to produce high-quality software. Our experience has been that producing software with very few defects is not only possible, but highly desirable: we spend less time and money fixing faults. As a result, our systems have low defect rates and deliver the required functionality on time and to budget. We achieve this by:

- deploying small, highly-skilled teams;
- using tailored, focused processes; and
- choosing the right tools for the job.

### 4.1 Deploy Small, Highly-Skilled Teams

*I'm very impressed by the calibre of your people.*

Andy Calvert, Security Development Manager, MONDEX International

*Agencies frequently send out lists of ... skills [such] as Visual Basic, C++ etc. These are not skills, they are programming languages... As an aeronautical engineer I am amused by the idea of applying for a job at Boeing ... quoting my "skills" as: screwdriver, metric open-ended spanners and medium-sized hammers!*

Peter Amey, "Logic versus Magic in Critical Systems", keynote address at Ada Europe, May 2001

For example, we employ about 100 engineers of whom 60% are software engineers; the others are safety and systems engineers. All of our engineers are degree-educated and many hold post-graduate qualifications; 25% hold doctorates and another 25% hold masters degrees. We have been delivering software successfully across the certification range for 20 years.

Our engineers are actively encouraged to achieve national and international recognition through gaining Chartered Engineer status with a recognised industrial body, for example the British Computer Society or the Institution of Electrical Engineers. A third of our engineers have Chartered Engineer status.

As professional engineers our staff continually seek to learn from others' experience, as well as improving wider industry practice through sharing our experiences. Paradoxically, it is because we recognise that engineering skill and experience are more valuable than intimate knowledge of the latest fad that we have knowledge of a variety of methods, tools and programming languages.

## 4.2 Use Tailored, Focused Processes

*If you do not actively attack the risks, they will attack you.*

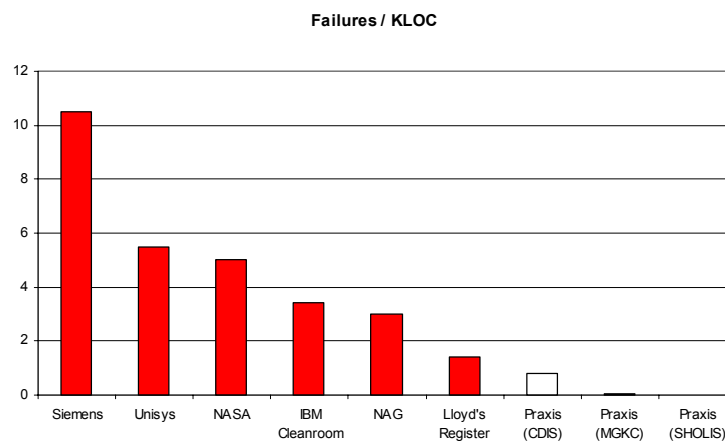
Tom Gilb, "Principles of Software Engineering Management", pub. Addison Wesley, 1988

The key tenet of White Box Software Development is to tailor the development process to fit the project. We do not mandate any particular process model, requirements engineering method, specification technique, programming language or test and integration strategy. Rather we select and tailor processes and tools depending on the particular job.

We use risk-based planning (Ould 1999) for every project. We start by identifying the project risks in order to determine what process model to adopt and what risk reduction measures to put in place. We identify the key attributes of the project and client expectations. This enables us to define or select appropriate processes, methods and tools.

Believing that it is more efficient to prevent errors rather than try to detect and fix them, we have developed the REVEAL method for requirements engineering. REVEAL is based on sound engineering principles, a practical and clear method for eliciting and defining requirements; for improving specification clarity; and for managing change to those requirements.

The result is that we produce software with very low defect rates as shown in Figure 2, but without having to compromise on features, time or cost.



**Figure 2. Our post-delivery defect rates are very low.**

*The use of tailored processes allows us to focus on the real problems.*

*The result is very low defect rates without compromising delivery.*

*Source data from (Pfleeger & Hatton 1997).*

### 4.3 Choose The Right Tools for The Job

*To a man with a hammer, everything looks like a nail.*

Mark Twain

We do not start a project with any particular set of methods and tools in mind. We defer such decisions until we have defined the technical approach. This means that not only do we select the most appropriate methods and tools, but also that we are free to use those methods and tools in ways that maximise benefits to the client.

For example, suppose our risk-based planning activity requires us to show that a piece of software meets certain requirements. Depending on the level of assurance needed, we would choose an approach somewhere on the spectrum from

- simply tracing the software units to the requirements; to
- formally specifying the requirements, proving that the formal specification is consistent, writing the code in a high-integrity language, and then proving that the code met the specification.

We might choose a formal language like Z or VDM to specify a whole system or just a critical sub-system, depending on the circumstances. We would then decide whether or not to prove the specification. It may well be that most of the benefit is gained from the act of writing the specification. For some applications, full proof might be essential; for others it might be gold-plating.

Where our chosen programming language is SPARK Ada, we still have degrees of flexibility in its use. We may only use it as a design language and employ the Examiner static analysis tool to check information flow between design units; we may write the software in the SPARK subset and choose whether or not to do information flow analysis; we may attempt to prove the code free of run-time exceptions, or we may undertake partial proof of correctness of selected parts of the program. These choices will be made according to the results of our planning.

We might use SPARK Ada rather than C++ as the implementation language for a safety-critical controller, but C++ rather than SPARK Ada for a non critical GUI. Our decision would be informed by expertise in *both* languages, rather than received ideas or prejudices.

## 5 Examples

This section contains recent examples of our work that illustrate the benefits of our White Box approach. With the exception of CDIS, all of these projects have been completed within the last 5 years.

## 5.1 Example: 10-Year Warranty and No Claims

*A key point in choosing Praxis was that we respected ... their software engineering processes and felt confident that they would be able to deliver the system. We regard our choice as vindicated by the trouble-free manner in which the system has been integrated.*

Derek McLauchlan, CEO responsible for the CAA Central Control Function programme

In the early 1990s, Praxis developed the SIL-2 Central Control Function Information Display System (CDIS) for the then Civil Aviation Authority, now National Air Traffic Services (Hall 1996). CDIS forms a vital component of the data entry and display equipment used by air traffic controllers at the London Air Traffic Control Centre (LATCC). The key characteristics of CDIS are:

- Very high availability (24/7) distributed real-time system
- Risk-based planning
- Prototyping and interviews to gather requirements
- Formal specifications using VDM and CSP
- Proof of critical parts only
- Implemented in C (200,000 lines)
- 0.81 defects per KLOC
- 10 year warranty
- No claims yet!

The most important overall success factor was the requirements engineering phase. Models and interviews were used to gather requirements from a variety of stakeholders. The requirements were then presented in 3 complementary views.

As in the requirements phase, the key to successful specification was to use the right combination of methods. User interfaces were specified by iterative prototyping with close end-user involvement. The data handled by the system was precisely described using VDM. Formal proof was reserved for the most critical concurrent elements.

The formal specification was refined until the final mapping to C was relatively straightforward and code could be produced using standard templates. This mitigated one of the risks of using C: that programs are poorly structured and hence that it is difficult to show compliance with the specification.

The results of our approach speak for themselves. Independent assessment (Pfleeger & Hatton 1997) has shown that the quality of the code is far better than the industry average for new developments. In the nine years since its commissioning, CDIS has since shown a very high level of operational availability. The robustness of CDIS and its conformance to its specification stands in great contrast to that of the other systems developed for LATCC.

## 5.2 Example: Zero Defects Post-Delivery

*The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong, it usually turns out to be impossible to get at and repair.*

Douglas Adams, "Mostly Harmless", October 1993

In the mid-1990s Praxis was subcontracted by Power Magnetics and Electronics Systems (now part of Ultra Electronics) to develop the application software for the UK MOD's Ship Helicopter Operating Limits Information System (SHOLIS). This is a SIL-4 system which aids the safe operation of helicopters on naval vessels (King et al. 2000). The main features of SHOLIS are:

- First software to UK MoD Interim Defence Standard 00-55
- SIL-4 embedded information system
- Small team
- Formal specification in Z
- Proof of specification most efficient means of eliminating faults
- Written in SPARK Ada (27,000 lines)
- Proof of exception freedom for most of the code
- 0 defects in sea trials

Defence Standard 00-55 (MOD 1991) is among the most stringent procurement standards in the world. It requires a formal specification and design, with formal arguments to link the specification, design and code. Many suppliers claimed that the level of formality required was unrealistic. Praxis was the first company to show that it is not only possible but desirable to construct software to such demanding standards.

Our approach included:

- A small team using maximal tool support
- The most powerful workstations available, because "a big computer is far cheaper than the time of the engineers using it"
- A simple system architecture which made the Z proof feasible
- Some 150 Z proofs carried out covering 500 pages, finding 16% of the faults identified pre-delivery for only 2.5% of the overall effort
- Selective partial code proof strategy, with additional manual analysis to show termination of all loops, functional separation and correctness of SIL-4 components
- 9000 verification conditions proven, most of them automatically

Most significantly, we found that Z proof was one of the cheaper stages at which to fix faults because it was early in the life-cycle, before any significant design activity had been undertaken. The next most efficient phase was the system validation test, which took place after integration, and was therefore one of the most expensive stages at which to fix faults.

Despite the unusually large amount of proof activity on this project, our approach was pragmatic, limiting the proofs to only those areas which were required to demonstrate compliance with Defence Standard 00-55.

### 5.3 Example: Halving The Cost of V&V

Praxis provided the majority of the software design team for a stores control unit for use on Royal Navy submarines. This is part of the UK's Submarine Acoustic Warfare Command System (SAWCS), which is to provide the Royal Navy with an effective defence against the latest generation of torpedoes. The main characteristics of this project are:

- Defence Standard 00-55 SIL-3 development
- Small team
- Innovative technical plan
- Use of INFORMED design methodology
- Half the normal effort for V&V

Traditional software developments view the testing phase as a means of finding faults. The problem is that testing comes too late: it can be orders of magnitude more expensive to fix faults late in the life-cycle (Boehm 1976, Baziuk 1995, Leffingwell 1997). Testing can thus be seen either as a very expensive way of eliminating faults, or as a relatively cheap way of confirming that the system has been built correctly.

Our experience on SHOLIS, where unit testing took 25% of the effort, but was one of the least cost-effective ways of finding faults, suggested that it might be better to avoid unit testing all together. *We were only able to do this because the software was specified in Z and implemented in SPARK Ada.*

We defined our development process to:

- produce software that would be easy to verify;
- produce certification evidence as a by-product;
- minimise reliance on unit testing;
- achieve structural test coverage by system testing, as recommended by DO-178B (RTCA-EUROCAE 1992); and
- take a system safety engineering approach, identifying the tension between maintenance and operational requirements.

The result was that the project expended 29% of total effort on V&V, compared with a norm of 40-60% for SIL-3 / SIL-4 systems.

### 5.4 Example: First Ever Certification to ITSEC-E6

*Your approach to requirements [REVEAL] was very effective.*

John Beric, Head of Security, Mondex International

*I've never seen such a clean hand-over of deliverables.*

Andy Calvert, Security Development Manager, Mondex International

Mondex International (MXI) invented the Mondex Purse, an electronic smartcard alternative to cash. Praxis worked with MXI to develop the MULTOS Global Key Centre (MGKC), which generates and manages the cryptographic keys used by the MULTOS operating system to maintain security.

MULTOS and the Mondex Purse were the first consumer products to be certified to the extremely demanding UK ITSEC-E6 standard. Three years on, only one other product has achieved this. The key features of MGKC are:

- Critical component of ITSEC-E6 system
- Capture user requirements
- Formally specified
- Mixed-language implementation - SPARK, Ada, C++, SQL - 100,000 lines total
- 0.04 defects per KLOC

The key points of our development process were:

- capturing the user requirements for the system using our requirements engineering method, REVEAL;
- use of our design method, INFORMED, for the security-critical parts of the software architecture and design;
- implementing the user interface in Visual C++, database queries in SQL and the critical components of the system in SPARK Ada; and
- controlling the risks introduced by a mixed development environment.

The vast majority of defects in the system were eliminated soon after introduction. Only 6% of the total project effort was spent fixing faults. Only 4 faults were found in the year post-delivery, a defect rate of 0.04 per KLOC (Hall & Chapman 2002) – lower than the Space Shuttle software (Joyce 1989), but much cheaper.

## **6 Conclusion**

We have debunked the common myths surrounding the difficulty of producing high-quality software at a reasonable cost. Although current industry practice is to produce complex and expensive projects prone to failure, and the common perception is that this is inevitable, this perception is unjustified. The key to success lies in treating Software Engineering as a true engineering discipline.

The white box software development approach described here has been applied to a range of commercial projects with successful commercial and technical results. There is no “silver bullet” in this approach; it concentrates on identifying the right people, processes and tools for the job. We spend less time and money fixing faults as a result of getting the software right in the first place.

## References

- Amev, P. 2001. "Logic versus magic in critical systems." In *Lecture Notes in Computer Science* vol. 2043, pub. Springer-Verlag.
- Baziuk, W. 1995. "BNR/NORTEL path to improve product quality, reliability and customer satisfaction." In *Proceedings of the 6th International Symposium on Software Reliability Engineering*.
- Boehm, B. W. 1976. "Software Engineering." In *IEEE Transactions on Computing and Software Engineering*, 1:1226–1241.
- Brooks, F. P. 1986. "No silver bullet – essence and accidents of software engineering." In *Information Processing* 86:1069–1076.
- Brooks, Frederick P. 1995. *The mythical man-month: essays on software engineering*. Anniversary (2nd) edition, pub. Addison Wesley Longman, Inc.
- Chulani, S. et al. 2001. "Deriving a Software Quality View from Customer Satisfaction and Service Data." In *Proceedings of ESCOM 2001*. Published on the internet, URL <http://www.escom.co.uk/conference2001>.
- Committee of Public Accounts. 1993. "Sixty third report: Wessex Regional Health Authority Regional Information Systems Plan". The House of Commons.
- Compaq. 1999. "Program license agreement for UK." Compaq Computer Corporation.
- Hall, Anthony & Roderick Chapman. 2002. "Correctness by Construction: Developing a Commercial Secure System." In *IEEE Software* Jan/Feb 2002, pp. 18–25.
- Hall, J. A. 1996. "Using formal methods to develop an ATC information system." In *IEEE Software* 130:66–76.
- Hansard 2000. Parliamentary debate, Monday 24th January. *Hansard*, 6th series, vol. 343, 5th volume of session 1999-2000
- Joyce, E. J. 1989. "Is error-free software achievable?" In *Datamation* 35(4):53–56.
- King, S. et al. 2000. "Is proof more cost-effective than testing?" In *IEEE Transactions on Software Engineering*, 26:675–686.
- Leffingwell, D. 1997. "Calculating your return on investment from more effective requirements management." Available from Rational, URL <http://www.rational.com/media/whitepapers/roi1.pdf>
- Leveson, Nancy G. 1995. *Safeware: system safety and computers*. Addison-Wesley.
- Lions, J. L. 1996. *Ariane 501 inquiry board report*. European Space Agency.
- McConnell, S. 1999. *After the gold rush: essays on the profession of software engineering*. Microsoft Press.
- MOD. 1991. "The procurement of safety critical software in defence equipment, INTERIM DEF STAN 00-55, Parts I and II". United Kingdom Ministry of Defence.

- Ould, Martyn A. 1999. *Managing Software Quality and Business Risk*. John Wiley & Sons.
- Pfleeger, S. L. & L. Hatton. 1997. "Investigating the influence of formal methods." In *IEEE Computer* 30:33–43.
- RTCA-EUROCAE. 1992. "DO-178B / ED-12B Software Considerations in Airborne Systems and Equipment Certification." RTCA-EUROCAE.
- Stålhane, T. et al. 1997. "In search of the customer's quality view." In *Journal of System Software* 38:85–93.
- Standish. 1995. "The CHAOS report." The Standish Group, URL <http://www.standishgroup.com/>
- Standish. 1999. "CHAOS: a recipe for success." The Standish Group, URL <http://www.standishgroup.com/>
- Stephenson, A. G. et al. 1999. "Mars climate orbiter mishap investigation board phase 1 report." NASA.
- Tassey, G. 2002. "The economic impacts of inadequate infrastructure for software testing." NIST. URL <http://www.nist.gov/director/prog-ofc/report02-3.pdf>
- Taylor, A. 2001. "IT projects sink or swim." In *BCS Review* pp. 61–64.