Closing the Barn Door: Re-Prioritizing Safety, Security, and Reliability

Richard J. Sutcliffe
Trinity Western University
7600 Glover Rd.,
Langley BC Canada
1-604-8887511
rsutc@twu.ca

Benjamin Kowarsch
Project Management Consultant
Sihlpost Postlager, Kasernenstr. 95/97
CH-8004 Zürich, Switzerland

trijezdci@NoSpam.gmail.com

ABSTRACT

Past generations of software developers were well on the way to building a software engineering mindset/gestalt, preferring tools and techniques that concentrated on safety, security, reliability, and code re-usability. Computing education reflected these priorities and was, to a great extent organized around these themes, providing beginning software developers a basis for professional practice. In more recent times, economic and deadline pressures and the de-professionalism of practitioners have combined to drive a development agenda that retains little respect for quality considerations. As a result, we are now deep into a new and severe software crisis.

Scarcely a day passes without news of either a debilitating data or website hack, or the failure of a mega-software project. Vendors, individual developers, and possibly educators can anticipate an equally destructive flood of malpractice litigation, for the argument that they systematically and recklessly ignored known best development practice of long standing is irrefutable. Yet we continue to instruct using methods and to employ development tools we know, or ought to know, are inherently insecure, unreliable, and unsafe, and that produce software of like ilk.

The authors call for a renewed professional and educational focus on software quality, focusing on redesigned tools that enable and encourage known best practice, combined with reformed educational practices that emphasize writing human readable, safe, secure, and reliable software. Practitioners can only deploy sound management techniques, appropriate tool choice, and best practice development methodologies such as thorough planning and specification, scope management, factorization, modularity, safety, appropriate team and testing strategies, if those ideas and techniques are embedded in the curriculum from the beginning.

The authors have instantiated their ideas in the form of their highly disciplined new version of Niklaus Wirth's 1980s Modula-2 programming notation under the working moniker Modula-2 R10. They are now working on an implementation that will be released under a liberal open source license in the hope that it will assist in reforming the CS curriculum around a best practices core so as to empower would-be professionals with the intellectual and practical mindset to begin resolving the software crisis.

They acknowledge there is no single software engineering silver bullet, but assert that professional techniques can be inculcated throughout a student's four-year university tenure, and if implemented in the workplace, these can greatly reduce the likelihood of multiplied IT failures at the hands of our graduates. The authors maintain that professional excellence is a necessary mindset, a habit of self-discipline that must be intentionally embedded in all aspects of one's education, and subsequently drive all aspects of one's practice, including, but by no means limited to, the choice and use of programming tools.

WCCCE '16, May 06 - 07, 2016, Kamloops, BC, Canada

ACM 978-1-4503-4355-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2910925.2910938

CCS Concepts

• Software and its engineering • Software and its engineering~Software reliability • Software and its engineering~Software safety • Software and its engineering~Imperative languages • Software and its engineering~Software design engineering

Keywords

Software Engineering, Safety, Security, Reliability, Education, Modula-2

1. The Purpose of Computing Software

1.1 Problem Solving

We produce software systems for the purpose of solving problems. At the highest level of abstraction, a problem has a start or input state, a state transition called a solution, and an end or output state. At a lower level of abstraction, the transition is a machine itself, and may have many internal states. A definition such as this is important for students because:

- they need to realize that a problem has a solution. Otherwise it would be called something else;

- software functions as part of an intentionally designed system, whose "intelligence" or meaning is supplied by the programmer, it is not self-generated by the system;

- issues of planning, appropriate tool choice, disciplined best-practices methodology, and a suitable testing regimen are therefore critical to success.

2. Successful Software

For marketing purposes, computer systems (hardware and software together) are sometimes thought of as commodities, in the same category as toasters, juice blenders, and ranges. Even if one confines oneself to discussing the hardware alone, this is misleading. They are not toasters, they are compound sliding

mitre saws, shapers, sanders, and drill-drivers--general tools for problem solving, for completing a variety of tasks.

However, software is more specialized to particular tasks--word processing, number crunching, data base management, data mining, numerical or statistical analysis, accounting, payroll and other business solutions, teaching, expert systems, school or hospital management, portfolio management, stock and bond trading, correspondence, record keeping, and the like.

Software is human purpose driven. Its meaning is imbued by the programmer. And, it is more specialized than the system as a whole. An individual program is a scroll saw, a chuck key, a number three Robertson hex driver bit, a tire or oil filter wrench, a flush cut saw, or a wire stripper--always a tool, and taking its purpose from the tool designer on the one hand and the tool wielder on the other. One needs far more than just a hammer, because not every problem involves driving a nail.

It takes the right tool in the right hands to produce a work of high craftsmanship, something that has real value. But there is more to it than this.

2.1 Epistemology

It is a truism that a student does not begin to comprehend an academic discipline until she knows what it means to say that some proposition is true in that discipline. After all, the answer to such questions differs among the fine arts, the arts, professional schools, and science, and within these in turn among individual departments. The epistemology of lab sciences differs markedly from that of mathematics, for instance.

In computing science "truth" metamorphoses to a constellation of

- correctness, that is, conformance to the specifications;

- readability, because humans determine problem solution semantics, compilers only deal with pre-defined static and dynamic language semantics, which is very narrowly limited by comparison;

- safety, which usually translates to the ability to run integrally, without mishandling or damaging its own resources, or those belonging to other components of the system;

- security, that is, the ability to prevent external malicious code units from interfering with its own resources, whether by intent or not;

- reliability, or the ability to continue operating correctly when the environment is stressed by scaling up the scope of the input or output data set or the number of concurrent processes and/or users;

- modifiability, which itself is a constellation around writability, readability, testability, modularity (factoring) and portability.

The bottom line is that software is "true" if it works as we intended and correctly specified, all the time.

3. What Do We Often Get?

3.1 Failed Projects

The software development landscape is littered with high-profile failures. A small sample follows:

The Denver Airport Baggage System, expected to be the most advanced of its kind, instead resulted in the airport sitting idle for more than a year while futile attempt were made to finish the project. what little ever became functional was scrapped when it became clear that a manual system was more cost effective [3].

British food retailer J Sainsbury PLC wrote off a US $526 million investment in an automated supply-chain management system, and returned to manual handling [4].

One of the classic failures is the US IRS Modernization Project that evaporated more than 3 billion dollars over the span of a decade [5].

Various authors estimate between twenty and thirty-one percent of all software projects fail to complete and that over half exceed cost estimates by two hundred percent or more [6]. It would be nearly impossible to accurately tally the monetary cost of failed software projects, but the general consensus is that it runs to the billions in the United States alone [7].

These dismal figures only scratch the surface of the software crisis. Far more projects achieve only a portion of their goals. Lidberg [8] took this into consideration when estimating the true software project failure rate at over eighty-four percent. More still are deemed unacceptable by their users because of poor functionality, inappropriate interface design, slow access, failure to scale up, or the inability to be adapted to changing workflow circumstances.

In recent years, in British Columbia alone, numerous government-sponsored IT projects have been cancelled unfinished or met with disapprobation by users because of interface, usability, and performance issues. [9-15]

Nor is there any lack of literature on the prevention of software failures--the observations made here are widely reported and quantified. [16; 17]

3.2 Software Crashes and Freezes

Some software that does get completed, even if with only partial functionality, may function more or less as specified, but only for a limited time, then crashes or becomes unresponsive. In the best case, no data is lost, and the only cost is an annoying reboot. In the worst, buffers were overflowed into memory belonging to other programs, the operating system, or their data, files were not properly closed, or critical data the program was using is damaged. As a result, customers may loose confidence, the client's reputation be damaged, even the future of the client business imperilled.

We will discuss software safety and reliability later from a positive and preventive point of view, so suffice it to say for now that crashes are rarely due to hardware faults, and that sound programming practice, especially when assisted by appropriate tools, can always at least ameliorate if not eliminate serious consequences, and ought to prevent such errors altogether.

3.3 Inconsistent or Unreliable Solutions

Commercial software releases, even those from large houses, often contain so many bugs that a discipline of statistical analysis arose in an attempt to (a) determine the probable number of total bugs from a sample, and (b) the optimal point at which to cease looking for more, in view of diminishing returns. [18] Some of these attempted to compare languages on this score. [19] Others tried to measure open source software quality. [20]

The situation has demonstrably improved, but too many developers release product that belongs more properly in an alpha or beta testing environment than in the marketplace. Whether due

to ignorance, lack of ability to test, or market pressures, the result is that vendors often rely heavily on end customers to perform their testing and bug locating, and some metrics have been attempted to determine the effects of "early release" on software quality. [21]

What ought to be the astonishing thing about such practices is that the marketplace endures them. Though end users complain about lost time, money and data, they let it slide, and there are therefore few or no consequences for behaviour and performance that in almost any other context would be viewed as unacceptable, unethical, and actionable.

3.4 Poor Documentation

Documentation is created in several flavours, each for its own purpose and audience.

1. Project specifications

These define what the software is supposed to do from a user and management point of view. They specify scope, and requirements, outline necessary resources, library and change management, timelines and procedures for completing the project. Quality assurance against specifications, including those for user acceptance is a crucial part of the latter. Success can only be measured against documented acceptance criteria using pre-defined processes.

These specifications should then be regarded as unalterable. In our experience, the most common reason for project failure is poor or lacking scope management that permits scope drift and feature creep. The most important skill for a project manager is the ability to extract a very painful price for any change of agreed scope as a matter of habitual principle. The most important skill for a project engineer is to say "no" when asked to do more than what the agreed scope says they should deliver. There simply is no change, however tiny it may seem that does not have an impact. Managers and customers alike must learn that once they agreed on a scope, changing that scope means severe pain that they will do everything to avoid. If they want an additional feature, they should be prepared and willing to give up something else in return. Developers' ability to deliver reliable and safe software is severely impaired by ever-changing specifications.

The timeline must include reasonable amounts of buffer time to accommodate emergent problems without jeopardizing the schedule. This buffer time may allow accommodation of minor changes without impacting the schedule. However, all change requests should come with a specified price tag, or demands for them will overwhelm the supply of buffer time, often to the point of failure.

Project managers and developers must also learn to be realistic when negotiating resources against project scope. They should not sign off on assignments they know cannot be completed in the specified time, assuming that accomplishing some portion will be adequate, nor should they agree to design-build methodologies, where specifications are created on the fly under the assumption that the final project can be created in chunks, without first knowing how they will work together. Unfortunately, this is precisely the basis for the now-favoured software delivery methodology known as "Agile", which institutionalizes not having to limit scope by avoiding commitments to defining what the final product should do.

2. Developer Documentation

These define the program units, libraries, interfaces, team membership and responsibilities, code review processes, testing and consulting regimens, and generally refine the specifications to the point where each team and member has a detailed list of responsibilities for code production. Reasons for decisions and change logs for these organizational plans and procedures are supplements or appendices. These are supposed to prevent endless revisiting of already settled arguments.

3. Internal Documentation

This consists of informative commentary within the code body, providing headings, explanations of algorithms, cross references into other documentation, and change log, with reasons for decisions. Unlike the above, the later are on the code level. Their purpose is to make the code not only readable but understandable by code reviewers looking for logical and semantic errors and by later teams tasked with modifications.

4. User Documentation

The provides either or both of built in help via a user interface menu and a manual with step-by-step operational methods for every aspect of the software. Usually written last, the manual would be better written as part of the specifications, revised when mockups are shown for test-of-concept to users for approval, and a draft finalized before code is written, or at least before it is tested--any change to which must be approved before being incorporated.

The sad truth is that few projects are documented in this manner. Project specifications are loosely written and ambiguous enough to allow of many interpretations, all of which may vary from what the users asked for, even more from what they need. Developer documentation may be no more than a whiteboard sketch of the factored functionality, code may be cryptic and entirely lacking in any explanation, decision logs may be nonexistent, bug fixes not recorded, the help menu explanations so brief as to be useless, and manuals incomplete or wrong because of last minute changes no one thought about including.

Document creation and management is not rocket science, and has been the subject of numerous papers in itself. [22-24] Computing science is manifestly harder than rocket science, but software developers ought to be good at tracking what they themselves do--there is even software for this purpose, [25] some of it for open source code that has many widely distributed contributors. [26] Code check-in and -out, while not documentation itself, must be managed and this management documented.

It is impossible to know if a software project is on track to complete with the resources provided unless the team knows what success means and can test it at every step. It is impossible to know if a project is finished unless "finished" has been clearly defined and is testable. It is impossible to maintain or modify software without all relevant internal and developer documentation available to testers, including prior decision tracking.

3.5 Easily Compromised Systems

It is scarcely possible to open a newspaper these days without reading of some well-known institution, manufacturer, or retailer having suffered a security breach that exposed personal and credit card information to a hacker. The "information is beautiful" site [27] lists a few hundred of the larger, that is where the number of

records compromised exceeds 30 000, and only since 2005. This runs to over fifty million in several cases, and the sites themselves are a catalogue of well known names such as Adobe, Apple, eBay, JP Morgan Chase, Home Depot, and many others. Most fall into one of two categories:

1. Misappropriation of passwords due to poor employee practice--leaving the password lying around, giving it to an unauthorized person, or being finessed out of it by someone masquerading as an executive of the firm, a bank account officer, or a law enforcement officer.

2. Taking advantage of a well-known security hole in the system hosting the entity's web site or other software, exploiting the insecure nature of the language in which the package is written, or using a brute force attack on passwords, relying on the programmer not having bothered to enforce password security. Because operating system exploits are often easier to close, and that closure affects many applications, the focus of attackers, and therefore of prevention, has in recent years shifted to the application level. [28]

The former are social exploits, which our students will be unable to influence except by education. The latter category is entirely a developer responsibility, first and foremost to be sufficiently thorough and meticulous to produce sound code, then to test it thoroughly. There is plenty of literature on testing techniques, [29-32] however testing should be viewed as quality assurance, that is, verification of correctness, not as a means to find mistakes resulting from sloppy habits.

4. Influences

4.1 Planning to Fail

Software is intentional. Developers should not start a project whose goals are not clearly laid out in advance, whose resources are not specified in detail, whose top down plan and initial factoring are not already complete.

In author Sutcliffe's forty-six year experience, the single most common reason for students being unable to successfully complete software projects is their failure to adequately limit their scope.

It is far better to correctly plan and meet modest goals than to produce little or nothing from unrealistic or ill-defined ones. Instructors need to pay close attention to this issue, and explain carefully why they are pruning the scope of a student software requirements document, or forbidding the dynamic addition of bells and whistles not part of those specifications.

Planning issues have been broadly identified in the literature as one of the most important issues in software failures. [16; 33; 34]

The issue once again is discipline. Students who fall into bad planning habits or are permitted to make undisciplined changes to previously agreed-upon plans will continue with the same habits after graduation, helping to doom yet more software projects and waste yet more billions.

Failure to plan appropriately and in detail or failure to stick to agreed plans is the same thing as planning to fail. To put it more positively, any software project that is more than a toy is complex in itself, thus the development process is also complex. Our students must learn early to manage that complexity. Compulsory courses in project management, or at least inclusion of a substantial management component in the standard software engineering course should be mandatory. The literature on this point is extensive. [35-37]

Since nearly all real-world projects are not constructed by programmers working solo, but require teams of developers, software engineering should last at least two semesters, with the basic theory up front, followed by a team-based project effort on a project of sufficient substance to mimic in a small way the workplace students will eventually face. [38-40]

The material on project management should address the construction and internal management of said teams, the choice and deployment of tools, version control, code standards, software metrics, and quality assurance, both on the code and the design.

Further, the software requirements document must include from the beginning how the issues of safety, security, and reliability will be addressed, how that part of the plan will be enforced during development, and how those measures will be tested (on non-live data). Ignoring these issues is the same as guaranteeing that the resulting software will be seriously flawed in all three respects.

Coding for readability, modifiability, and portability must also be addressed in the initial plan. Even if the overall model does not include porting to other platforms or envision later versions, the necessary code factoring should still be undertaken--not only because the initial plan may change, but also because it embodies correct discipline, which is worthless if not consistently and universally applied.

4.2 "Quick and Dirty" Does Not Scale

The knock against the small traffic circle intersections many jurisdictions are building is that although they reduce costs and increase traffic throughput at low volumes, they impede traffic flow at high volumes.

Something similar can be observed in, say, a university calculus course. If all the instructor does is present the theory and one or two trivial examples of the relevant application technique, students will not begin to appreciate the essential idea, the scope of the application domain, or the synthesis with other techniques to tackle even modestly difficult problems, let alone the complexity of real world engineering.

Likewise, amateur programmers, and therefore many students, get into the habit of knocking off a few hundred lines of code for quickly programmed solutions to simple problems to satisfy immediate needs. Classroom examples and assigned homework reinforce such habits, as they tend to be short enough to lend themselves to this kind of brute force frontal attack, with its production of throwaway code that illustrates nothing about the real world.

Such may be adequate to teach and illustrate basic principles of program constructs, but unless students quickly move beyond mere code snippets not exceeding a few hundred lines, they will be all but ignorant of any practical discipline of software engineering.

What suffices to hack together a program that occupies a screen of code does not work for one with ten thousand, a hundred thousand, or several million lines of code distributed among hundreds, thousands, or hundreds of thousands of discrete procedures. Indeed, the quick and dirty approach is inimical to sound software development, and guarantees failure in anything larger than a one-off toy or merely illustrative code fragment.

Students need to work with ever larger projects at an early stage of their education so as to have a realistic view of the market in which they will labour as professionals. They need to learn early on that software is complex, large, and therefore requires strict attention to discipline.

4.3 The Dilbert Principle

An old adage infamously has it: "Those who can do, those who can't teach." This saying is so bad it isn't even wrong. Rather, one cannot truly say they understand an idea or technique until they have successfully communicated it to another person. This was the basis of the Greek notion of λογος. Something apprehended from the divine realm, then successfully communicated using the rules of discourse and rhetoric was λογος--a word that can therefore be translated as either "spiritual", or "logical" but has both aspects.

Our point is that it would be far better to assert that one must successfully do before being able to teach.

One could make a case for an equally strong relationship between doing and managing. True, management can be thought of, studied, and analyzed as a discrete discipline in itself, with general principles applicable to all types of management. Yet every endeavour has its unique characteristics, work flow, tools, ideal team mix of personality types, knowledge base, and metrics. This is clearly understood in almost every sector. Whatever mechanism for senior management is in place, day-to-day oversight must be by a knowledgeable member of the profession being managed.

Principals are already teachers, university deans are faculty, shop managers have worked on the floor, senior officers have worked their way up the ranks, deputy minsters are already civil servants, an auditor must know accounting, and so on. Yet despite literature on team programming psychology [41] and success of chief programmer and diversified internal management models that directly involve the experts in organizing the workflow [42], many large software projects are contracted, planned, and supervised from outside the operational team by people with little or no knowledge of the discipline or of the kind of team personality mix needed to empower success. Sound decision making and best practices work are all but impossible in such an environment, making project success more a matter of luck than intentionality.

True, our students may not be able to influence this aspect of success promotion in their first few years after graduation, but they do need to learn software management as part of their undergraduate curriculum, as the deck may well be stacked against acquiring such skills on the job.

4.4 Tool Choice

Following the introduction of FORTRAN for numeric applications and COBOL for business use at the beginning of the modern software era, numerous programming languages were developed--some adaptations of those two, others specialized for particular problem domains, some functional, some logical, some general purpose, and with each generation, new ideas on sound programming practice were incorporated. Those designed by large committees such as PL/1 and Algol had brief periods of popularity for their broad generality and versatile control and data structures, but died under their own weight. Others were too cryptic or too specialized to reach the mainstream.

Some provisions that are today acknowledged as part of the canon are separate compilation of libraries with type checking, top down programming to go along with top down design, data hiding to avoid unwanted side effects, dynamic data types, generic programming, and object oriented programming. All were at one time introduced by one or more notations and found their way into others. Practitioners came to appreciate orthogonality, simplicity, readability, expressiveness, safety features, garbage collection, the ability to combine languages on one project using a programming environment.

It remains the case that some notations are more robustly suited to particular application domains than others, and all students need to take a course in programming language theory, history, and design so that they can make appropriate tool choices for the problem solution at hand.

4.5 Modula-2

One of the pre-eminent language designers of the last four decades has been Niklaus Wirth, who built ALGOL-W, designed Pascal, then Modula-2 as a corrective for the former and an alternative to Mesa, and from Modula-2 derived Oberon. [43-45]

At each step of this sequence Wirth introduced new ideas, removed ones he felt had been tried and found wanting, and aimed for simple but expressible languages. [46] Perhaps the most influential of these is Modula-2, defined in [47] subsequently refined in [48-50] and that enjoyed great popularity in the 1980s.

At the instigation of the British Standards Institute, ISO undertook a standardization of Modula-2, participated in by both the present authors, and the results of which were published in [51-53]. Numerous introductory and advanced programming texts were published during this decade, including those oriented to Wirth's original (classic) dialect [54-58; 58] and the later ISO dialect. [59]

Some of the Generic and Object Oriented extensions to the base language were explicated in [60-63] .

However the length of time taken by the ISO group to complete its work, and the additional complexity of the standardized library meant that only a few compilers were ever created for the ISO version, and the notation quickly became a niche language, often used by programmers as their pseudo code and proof of concept before translating into the better promoted (to managers), but less safe and more cryptic C++ for final distribution.

Starting in 2009, and with the essential design finalized in 2010, but under further refinement to 2015, the authors [1; 2] produced a modernized dialect of Modula-2 in an attempt to leverage the strength of Wirth's fourth edition, apply his design principles to new concepts and methods to improve safety, security, and reliability in final products, yet at the same time retain the original simplicity. Although the authors broadened the language's scope, they did so cautiously, retaining the simplicity of the original.

This paper is not a specific explication of the Modula-2 R10 design goals, decisions, or grammar, but the revised language is employed or referred to in portions of the following chapter to illustrate selected examples of this project's solutions to some coding aspects of the software crisis.

5. Solutions

It's an old saying that there is no silver bullet for software engineering. That "truism" is misleading. As already asserted, there are many techniques well-known since the 1970s that, if

applied correctly, greatly improve the chances of completing software development with a correctly working product and satisfied users.

True, some of the issues, such as unrealistic expectations, improperly limited scope, uninformed and even hostile management practice, are beyond the ability of our new graduates to immediately impact. They may not even be given input into the choice of tools. The knowledge and habits we teach in these areas can only be properly applied if they achieve management status themselves without forgetting the principles they learned from us.

However, there are practices that we can teach, habits we can inculcate that can at the very least partially ameliorate many of these problems. Indeed, sound programming practice, though best implemented with cooperating tools, should even overcome many of the pitfalls that bad tools present.

We discuss some of the issues under classic headings.

5.1 Readability vs Writability

Every piece of code presents the developer with a tug-of-war between these two. One naturally wants to get the job done with the least amount of writing, but need not go far in this direction before forgetting that an important part of the "job" is to produce code that can be read, understood, corrected, and modified by others--even by oneself at a later date. A code base that lacks this quality may require later re-development to start from scratch, wasting vast resources that could have been put to better use.

Our students need to buy thoroughly into the proposition that it is more important for human beings to be able to read their code and understand its semantics than it is for the compiler to accept it as syntactically correct. Code expresses intentionality that a computer cannot have. The absence of patent intentionality to others reading said code is indistinguishable from meaninglessness, for without such clear understanding, the code is frozen, forever unmodifiable for correctness, porting, or new functionality--all three of which are guaranteed will be required.

What is easier to read?

```
for (i=1; i<s; i++) for (j=1;j<t; j++)
get a[i][j];
```

or

```
FOR rowCount IN RowRange DO
  FOR colCount IN ColRange DO
    CaptureSpectralData ( SpectralArray
[rowCount, colCount] )
  END
END;
```

Both tool choice and coding standards are in play here, but the first example could have been made nearly as readable as the second despite the chosen language inherently lending itself to writing cryptic code. Layout (pretty-printing) helps to highlight nesting and explicate the logic. Choosing descriptive identifiers that explain what the programming entities are or do makes code clearer, often without requiring the addition of comments. If an algorithm is in play however, especially if it has been tweaked for performance, every step must be explained in clear and cogent internal comments. The last thing would be modifiers of the code need is a cleverly tangled and indecipherable starting point neither they nor anyone else can understand.

Now, consider the case of an abstract data type based on an opaque type. Often, such ADTs are implemented as public pointer types to hidden record/struct types. In C this will look like:

```
/* header file */
typedef struct adt_s *adt_t;
/* implementation file */
struct adt_s {  /* members */ };
```

In classic Modula-2 this will look like:

```
(* definition module *)
TYPE ADT;
(* implementation module *)
TYPE ADT = POINTER TO RECORD (* fields *)
END;
```

Neither is very intuitive. A practitioner unaware of the concept, will be very unlikely to guess what the definition in the header or definition module is all about. In the C version they will likely wonder where that adt_s type is defined and go looking for it. In the Modula-2 version they will wonder why the type has nothing but a name. In either case the meaning has to be learned. It is not self-evident from the syntax.

One of our design principles was to make such meaning self-evident in the syntax so that even a practitioner of another language who is unfamiliar with Modula-2 R10 will immediately be able to deduce its meaning:

```
(* definition module *)
TYPE ADT = OPAQUE;
```

Not only is this self-explanatory, but more important, it is search engine optimized. What would a person search for if they didn't understand the C or classic Modula-2 version?

Consider derived types. A syntax like:

```
typedef foo bar;
```

or

```
TYPE Foo = Bar;
```

could mean type equivalence, or derivation without equivalence.

But a syntax like

```
TYPE Foo = ALIAS OF Bar;
```

cannot possibly misinterpreted.

5.2 Safety

The above code also illustrates one potential safety issue. In C/C++, and some other languages, array index bounds are unchecked, so that indexing past the end of an array references memory not belonging to that array. Other languages require the program to raise an index out of bounds run time error (exception). The notation shown can be employed to remove both defects. If the specified range is used to define the array type in the first place, the iteration is restricted to that range. Moreover, if the iteration can be specified in the definition of the data type (possible in and encouraged by Modula-2 R10) to be over rows first, then columns, the following code is safer still:

```
FOR item IN SpectralArray
  CaptureSpectralData (item)
END;
```

Other safety questions that can be asked about professional practice and/or compiler error detection (enforcement):

- are actual procedure and function parameters type checked against the formal ones in the definitions (headers)? Best solution: language requires this.

- Can code be made generic as to: the size of an array in a record field or procedure parameter, without sacrificing type checking, or the target type for a data structure or algorithm so as to allow reusable code? Best solution: language requires both. Several now do include generic library templates, few but Modula-2 R10 offer both.

- Are all variables declared before use? Best solution: language requires this.

- Are all variables initialized before being used in a declaration, either automatically by the compiler, or manually by the programmer? Best solution: compiler generates code to set pointer variables to NIL, but for all others, detects and reports whether the program initializes them before use. Why the difference? Because any dereferencing of a NIL pointer should cause a run time fault, whereas auto-initializing other variables might mask a logical error of failing to initialize.

- Does the program rely on the value of a variable used in a loop after the loop concludes? This varies from one language to another, so instructors should not accept student programs depending on specific semantics. Best solution: The language explicitly defines the loop body as the scope of any control variable defined in its header so the value of this variable cannot be accessed outside it.

- Does the meaning of a procedure or function depend on the order of evaluation of the parameters? This too varies, with many languages not requiring the implementor to adopt a particular semantics. Best solution: tell students that programs depending on a particular parameter order for its meaning is wrong.

- Are all branches of program logic executable and in particular are all loops exitable? Partial solution: the compiler is required to check whether all branches of code are reachable. Students must adopt the habit of checking that in all run time scenarios their program will remain live and will not loop indefinitely.

- Are all data accesses guarded against such things as null pointer references, divide by zero attempts, value and bounds overflows, and the like? Note: A student program that employs exceptions as general error handlers should be returned unmarked to underscore that no shipping code should contain such workarounds. Run time faults must be prevented from happening in the first place, and such handlers used only as part of the debugging process.

- Are the language's control structures adequate to express a programmer's algorithms? If not, the solution is not to subvert them, but to employ a more appropriate tool.

- Is all dynamically allocated memory disposed of when no longer needed, either automatically, by garbage collection, or manually in the program). Best solution: If such memory is not automatically dynamically deallocated when the entity goes out of scope (garbage collected), programmers must develop a strictly disciplined habit of ensuring they do this. One habit is to ensure that the action of obtaining such memory is clearly marked, either by language syntax such as `NEW item := bat;` or if through a user-defined procedure, by including the word "new" in the procedure name: `newItemWith (bat)`.

In languages requiring manual deallocation, memory leaks are the number one cause of program crashes, and can be the hardest bugs to locate and repair.

- The above question may need to be asked and solutions applied separately to object entities, as the semantics may differ, even though object entities are almost certainly implemented with pointer semantics.

All these language-related points are addressed in Modula-2 R10 and are here illustrated with some examples:

Example 1:

Older languages such as FORTRAN, BASIC, and Pascal used GOTO as an emergency exit to control structures. Few do today, but if some form of GOTO is permitted in a language, the transfer of control should always be forward, never backward.

Example 2:

Variant records raise yet another safety issue. These contain a tagged field followed by two or more variants, depending on the value of the tag. Syntax such as

```
CASE tag OF
  value 1:  field : expression
```

is itself not very self evident. Moreover, changes to this variant part of the record definition can break all client code. This is similar to the fragile base class problem in object oriented programming. Moreover, if the language does not guarantee run time consistency checks on the values of the tag versus field references, the code is unsafe, because a value could be stored under one type and read back under another. Having extensible record types as in Oberon, or ones that can optionally specify their extensibility as in Modula-2 R10 provides convenience and efficiency without sacrificing safety.

Example 3:

One often wishes to create a record type with an array field generic as to size. In C:

```
struct vla_t { unsigned size;
  /* possibly other members */
 payload_t buffer[];
}
```

In Modula-2 R10:

```
TYPE VLA = VAR RECORD
  size : CARDINAL;
  (* possibly other fields here *)
VAR
  buffer : ARRAY size OF PayloadType
END;
```

The difference is that in C, the programmer is responsible for calculating the required size for a given number of elements for buffer, allocating it accordingly, then storing the size value manually in the size field. Further, the runtime system will not bounds check the buffer even though its size is stored in the struct. This too is the responsibility of the programmer.

In Modula-2 R10, allocating an instance merely requires an additional argument that provides the number of elements for the buffer.

```
VAR vla : VLA;
...
NEW vla OF 100;
```

which automatically calculates the required allocation size and stores the number of elements in the size field, which is henceforth treated as immutable. Further, any access to the buffer is automatically bounds checked against the size field.

The benefit of this kind of allocation is that it requires only one call to the heap allocator which is an expensive operation and it also avoids cache misses since the fields in the record are stored in close proximity to the buffer, which may not be the case when the buffer is a pointer that must be allocated separately. It is also more convenient to address buffer elements using array notation instead of using pointer arithmetic, further pointer arithmetic is less readable and represents opportunity for error. Hence, the Modula-2 R10 solution delivers convenience plus efficiency plus type safety all in one, without contradiction.

Example 4:

Many practitioners today no longer know the difference between safe and unsafe operations, particularly safe and unsafe type changes. First, there is a problem with terminology. Universities no longer teach a consistent terminology that aims to distinguish between safe and type safe type transfer. Then there is a problem with programming languages not distinguishing between the two operations either, whether by terminology or syntax. Every unsafe type transfer is a code loophole that elevates the risk of a program malfunction. They should therefore be restricted to the absolute minimum and clearly marked so they stand out.

It is thus worth mentioning that in Wirthian languages there has always been a clear distinction between the two, where cast has traditionally been used to mean the unsafe coercion operation and conversion has been used to mean the safe operation. ISO Modula-2 highlighted this by putting the unsafe operation into a pseudo module and requiring it be imported:

```
FROM SYSTEM IMPORT CAST;
...
 real := CAST ( REAL, card );
```

but it also introduced a multiplicity of safe type conversion functions with a variety of names and syntaxes.

The authors renamed this module in question to UNSAFE, so there can be no doubt as to its contents, and they encourage only qualified import, thus

```
 real := UNSAFE.CAST ( REAL, card );
```

and for the sake of consistency, readability and orthogonality, they removed all the separate type conversion functions and introduced a universal syntax for safe conversions:

```
real := cardinal :: REAL;
cardinal := real :: CARDINAL;
```

rather than

```
real := FLOAT (cardinal);
cardinal := TRUNC (real);
```

Even in classical and ISO Modula-2 any type could be auto-cast to an ARRAY OF BYTE parameter, and it was not obvious that this was a cast. In Modula-2 R10 we mark these as unsafe, ensuring there are no hidden unsafe features.

Example 5:

Yet another safety issue emerges in languages that employ variadic formal parameters, where the number of items passed has to be determined at run time. Modula-2 has safe variadic formal parameters. For instance, representations of polynomials of the form:

$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ...$

can be created by a variadic function of the form:

```
PROCEDURE newPoly ( factors : ARGLIST OF
REAL ) : Polynomial;
```

subsequent to which a function call

```
p := newPoly( 15.0, 3.75, 0.0, -5.0);
```

creates a representation of

```
f(x) = 15*x^3 + 3.75*x^2 - 5
```

The number of arguments is variable, but the argument type is defined and can therefore be compile time checked.

Within the procedure body, the number of arguments can be obtained with the built-in COUNT function.

```
argCount := COUNT(factors);
```

and arguments can either be addressed using subscript notation

```
arg := factors[n];
```

or via iteration in a FOR IN loop

```
FOR item IN factors DO
  WRITE(item);
  WRITE("\n");
END;
```

Other use cases may require mixed type arguments. Consider, storing key/value pairs in a collection.

```
PROCEDURE newDict
   ( pairs : ARGLIST OF {key : KeyType;
value : ValueType} ) : Dict;
```

a call

```
dict := newDict(key1, val1, key2, val2,
key3, val3);
```

creates a new collection with key/value pairs

```
{ {key1, val1}, {key2, val2}, {key3, val3} }
```

Here again, the number of pairs is variable, but both key and value type are defined and can therefore be type checked at compile time. As before, within the procedure body, the number of pairs passed in can be obtained with the built-in COUNT function. Individual keys and values can be addressed using array subscript notation:

```
key := pairs[n].key;
val := pairs[n].value;
```

or via iteration in a FOR IN loop

```
FOR key, value IN pairs DO
  WRITE(key); WRITE(" = ");
  WRITE(value); WRITE("\n");
END;
```

Safe type extension, safe variable length array fields and safe variadic functions combine convenience, efficiency and safety. These facilities could be provided in other languages, but there are few that do so. The authors express the hope that their ideas will be more widely adopted.

Examples could be multiplied, but the authors most important point is that although a language tool may encourage safety, good habits are irrespective of language. Deeply ingrained sound programming methodology will overcome many if not most limitations of a poor language, but the strictest of language safety features cannot overcome bad habits. Good habits are best taught in a disciplined environment using a disciplined language, because although a good programmer may produce sound code using any language, good or bad, the overwhelming majority of graduates will only ever become disciplined if required to use disciplined languages in their formative years, if not longer.

Poor programmers daily wield a loaded automatic assault rifle aimed at the integrity of their code (their foot, in older literature). That the gun may have its safety engaged only enforces a brief pause before the destruction begins; it does not prevent it. Murphy says that in modern computer software everything goes horribly terribly wrong in nanoseconds.

5.3 Correctness

We say that a program is correct if it implements the desired problem solution, that its it performs according to its specifications. But this requires us to ask what were those specifications? Were they sufficiently specific to tell whether the end product meets them, or were they too ambiguous? Did the users get what they wanted, what the developers thought they asked for, or what management decided they would get?

Again, our graduates might toil years in the trenches before being able to appreciably influence management thinking, but this is the one realm where a line should be drawn in the sand--no meetings between developers and end users to build their requirements into a firm and scope-delimited specification means no starting the project. The message has to get upstairs that it is impossible to develop software with incomplete, improperly delimited, fluid, or ambiguous expectations. While "terminated" may look bad on a resume, repeatedly being part of failed projects ought to look far worse. To be worthy of being thought a professional, a software developer needs to stand up and say "this methodology not only can't work, it can't even come close to working."

Moreover, since extraordinary claims require extraordinary evidence, it must be possible to demonstrate correctness, and this can only be approximated (it can seldom be proven) by extensive testing. Students need to be taught, then experience, software reviews for code quality and specification compliance, as well as regimens such as limit and bounds, logic branch, and scalability testing. In turn, code standards need to address such issues as appropriate program factoring, separation of model, view and control, business logic from system logic, and the quarantining of both unsafe code and platform specific logic, the latter so as to make portability at least distantly feasible.

Yes, these topics cover more than compliance with user specifications, because without diligent attention to quality requirements, producing correct solutions for all but the most trivial problems is essentially impossible.

5.4 Security

Few software solutions are for one-off calculations. The vast majority involve the manipulation of data to generate information. Electronic files may contain confidential personal data on employees, customers, or test subjects. Others may contain critical business information, including patents, plans, strategies, proposals, bids, finances--the mission critical life blood of the organization. Software systems are secure if:

(1) they do not themselves damage the integrity of the data they manipulate, and

(2) they do not, via some program flaw, reveal or permit access to that data to unauthorized persons.

Making hourly, daily, weekly, monthly, and annual backups to secondary storage, some of them off-site, is one aspect of securing data. But this can hardly help if a program with access to it has insecure passwords for its users, input buffers lacking bounds control and thus vulnerable to overflow hacks, resides on systems that run unsecured operating systems, or that co-exist or communicate with other programs or users that are easily compromised.

One of the authors owns a small web hosting company whose products and pages reside on a sleepy, low-use server situated in a large data centre. On a normal day, its software firewall reports an average of three hundred or so knock-on- the-door tests of its security by hackers' automated systems probing for takeover possibilities. During times of more intense activity, e-mail bombs, denial of service attacks, password compromise attempts and challenges to known loopholes in specific services or packages may increase to multiple ten thousands per hour. Better known companies may see that much malware traffic at normal times, orders of magnitude more in the periodic storm seasons to which the Internet is prone.

Vanilla host servers have many well-known vulnerabilities that must be secured against before being deployed on the net. A short list of solutions includes: deploying hardware and software fire walls, packet inspection, black lists of malicious sites, firewalling users from each other, scanning emails, attachments and uploads for malicious content, limiting SSH access, turning off root access and relying on a secondary account or an SSH key, turning off dangerous PHP functions, denying user access to compilers, monitoring for port scans, not allowing packages or extensions known to be vulnerable, intercepting code injection attempts and other common exploits, keeping the OS and service programs up to date with patches, filtering email, blocking access for too many password attempt failures or suspicious events, detecting and rejecting fake return mail addresses by requiring reverse DNA or other security credentials, and turning off some vulnerable services. Potential vulnerabilities are too numerous, and too informative, to list exhaustively.

Users also have to be taught not to post their passwords on the edge of their screens, not to click on links or reply to information requests in unsolicited emails, and not to surf pirate or porn sites, which often double as malware injectors. All the black hats need is a single point of entry to compromise everything.

Unless the entire system is locked down with multiple security measures it makes little difference whether a specific package is secure.

But for those, sound development practice means not only bullet proofing each individual package but gaming the consequences of an intruder gaining access and destroying or changing data. The programmer then takes steps to ensure that such a scenario will not render the enterprise incapable of continuing business. In other words, software needs a fall back position so that when Murphy strikes anyway, despite taking precautions against all known problems, and everything goes horribly terribly wrong, the failure in and of itself does not destroy mission critical functionality or data.

Once more the choice of tools impacts these considerations, for some languages are inherently insecure. Web hosting servers, for example, must configure PHP to deny the use of certain functions or they will see their machines quickly compromised. This may also mean restricting the use of certain common web portal packages or extensions thereto. As previously noted, buffer overflows are a notorious source of trouble for C/C++ code.

However, security precautions, like correctness, involve a mindset tuned to the issue from the beginning of instruction. Best of class tools will help, but cannot prevent the destruction inevitably flowing from sloppy security habits.

For instance, a program that enters the termination chain either normally or due to a run time fault should at least execute termination code that flush-closes all files and releases locks on any other resources. Do all languages or implementations thereof provide the means to attach such code to the termination chain? No, and when not, all files are put at risk for data corruption, meaning provisions to revert to earlier backups become critical.

Other helpful habits include: modularizing and otherwise employing data hiding to keep access to critical data, system functionality, or dangerous techniques all quarantined from parts of the program that do not require them, at the same time limiting possible namespace corruption, being careful to flush data buffers before closing files (not all languages or systems specify this behaviour) controlling the scope and visibility of records and individual fields, blocking access to data implementation details by using visibility controls such protected fields, opaque types and enforce strict procedural interface to data fields so it cannot be altered by any but programmer-supplied means.

5.5 Reliability

This quality attribute refers to the correctness and security of code over the long haul. Does it work in the prescribed manner for all inputs, or does it break under some? Does it work for a period of time and then fail? The former is likely due to incomplete testing at the data type bounds, and the latter to memory leaks, inappropriate storage techniques, or the inability to scale up for larger or more complex data sets. Some problems arise from operators becoming sufficiently used to the program that they enter the data faster than it can be handled. Others arise from the inability to scale the software from the handful of users who tested it in the lab to the hundreds or thousands that want to use it simultaneously in the wild. What appears to be reliable software in the microcosm becomes unusable in real life. This is a particular problem with governments and other large institutions.

Consider software that runs a telecommunications switch. It must be hit many times a second, continuously for years, and never fail. Is this possible? It must be. Many critical processes depend utterly on one hundred percent reliability at high speeds over extended periods of time. Again, examples multiply. Banks, manufacturers, shippers such as large on-line emporiums, postal services, insurance companies, school systems and many others have mission critical, very large volume software and user requirements that cannot be permitted to fail under load or become fragile over time.

Once more, the issue is mindset, though the measures taken to ensure correctness, safety, and security work together with scalability to ensure long-term reliability. Question for an aspiring professional: "will this piece of software you've written function under ever increasing stress for at least a decade?"

5.6 Technical Debt

No discussion of software quality and software reliability would be complete without examining what is now called "technical debt", a term coined to describe the accumulation of issues within software that are acknowledged to require attention and remedial work but are set aside for "later", often indefinitely. [64] The debt metaphor was deliberately chosen to communicate the fact that much like financial debt, technical debt accrues interest over time and it must eventually be paid down. If one fails to pay down the debt, there comes a point when it can no longer be managed at all.

The culture of "release early, fix later" has led to an explosion of technical debt in our software. In a competitive race for ever more features, decision makers are unwilling to set resources aside to do remedial work on software to pay down technical debt. However, when technical debt accumulates in a software it reduces the ability of the developers to meet future demands. At some point, there will be so much technical debt that it is no longer possible to meet any change requirements. At that stage, a complete rewrite of the software is often cheaper than doing maintenance on the existing software.

More and more companies are beginning to recognize this problem but in many organizations technical debt is the elephant in the room. Everybody knows it is there, everybody knows it is a serious problem but it is not openly talked about and considered best ignored. Yet even organizations who have acknowledged the problem find it difficult to get a handle on their technical debt because methodologies and tools to deal with technical debt are not yet readily available, though attempts have been made to move it from theory to practice [65] and some case studies have been done [66].

The first step to deal with a problem is to acknowledge that the problem exists and warrants action. This is the realm of education. Managers and engineers alike need to be made aware of technical debt and the need to deal with it.

The next step is to model and quantify the problem [67]. To do so, experts in software quality assurance have developed proprietary tools. These tools aim to identify the issues that exist within the software by location with the source code, catalogue and rank them, then compile reports on a regular basis.

In a further step, management can then identify which of the reported issues promise to make the biggest impact if time was to be spent on fixing them, derive work schedules and assign resources to work down the debt while continuing to track it.

The key to success lies in the tracking and quantifying of the technical debt. Developers often add "TO DO" marked comments in their code to remind themselves that a certain task is yet incomplete or that an algorithm was implemented incorrectly just to pass a deadline. What is lacking is a systematic approach to

filter, catalogue, quantify and report such TO DO items to revisit them as a matter of process rather than by chance.

In Modula-2 R10 we have therefore chosen to replace the empty statement with a TO DO statement that is part of the language. A TO DO statement may include a reference number from an issue tracking system, an optional severity level and a mandatory non-empty list of TO DO tasks with a short description and an optional estimate of time assumed to be needed to complete the task.

Example:

```
TO DO ( "Issue #1234", Weight.Critical )
  "Document pre-, post-, error conditions", 2 * WorkHours;
  "Implement replacement procedures", 2 * WorkDays;
  "Implement unit tests (separate module)", 4 * WorkDays
END;
```

Formalizing TO DO items within the source code has the advantage that all the reminders can be reliably found, and regular reports can be generated automatically.

It further raises awareness amongst developers about the importance of technical debt and the need to reduce it. This in turn leads to better discipline and a more realistic assessment of the feasibility what can be done with given resources in a given time.

5.7 Modifiability and Portability

There is always a version 2.0. However, in the software industry, by the time a second version's corrections, refurbishing, and new features are scoped out and formally specified, possible new platforms decided upon, and the resources needed to implement everything documented, one of two scenarios may play out. Brooks suggests [68] that if the development team is intact, overconfidence may lead them to agree to a feature-heavy design they cannot possibly implement.

Alternatively, the original programmers may have moved on to greener pastures--two or three times. A new team is assembled, given the specifications, the timeline for roll out, and handed the passwords to the code vault. What do they find there? A carefully, clearly written and documented gem that's easy to understand on first reading, has the system specific, unsafe, and presentation (view) code sequestered in discrete modules distinct from those implementing the business model, easily extensible data types, reasons for coding decisions explained, guidance for future coders and the whole indexed, cross referenced and ready to roll?

More likely they inherit a cryptic spaghetti tangle of interwoven business model, view, control, system specific tricks, and clever but undecipherable algorithms, no internal and little external documentation, no explication of code decisions, and no way to find anything in particular, or tell what it does when they do. It is slightly possible someone once understood such a mess; it is impossible that even the original authors could now comprehend its meaning.

Such "code" is essentially worthless. It would be cheaper to update the original specifications, if there were any, then start over from scratch, than attempt either maintenance, extension, or porting (presented in increasing order of impossibility). Since even two years can be an eternity in hardware provisioning, a degree of porting is often necessary. Indeed, even if all that is required is adaptation to one new peripheral, a program written as an unmodularized monolith may have to be changed in hundreds of scattered places (if they can all be found) to achieve the goal. It's always better to develop correctly the first time, and that may well mean scaling back expectations.

Doing it right the first time in this instance includes: ensuring that the project is carefully designed from the top down, factored into discrete units, further modularized to keep related items together in separate modules with clean interfaces, and in particular ensuring that input/output, system specific code, and presentation code are grouped in their separate modules. Then

- the implementation of a code thunk can be fine-tuned for speed and reliability without affecting the correctness of any other part of the package, because everything else sees only the library interfaces;

- porting can take place by replacing a single library unit;

- peripheral upgrades can be achieved by modifying a single code unit without interfering with the business logic;

- presentation interfaces can be updated or modernized by changes in another single code unit.

Data type extensions to prior versions require special attention. If an existing record type is expanded in situ, all client code must be recompiled, and some of it may well break. On the other hand, if record types can be extended as they may be in Modula-2 R10, and the new fields manipulated in separately encapsulated code, the changes are non-fragile. The original code continues to use the original data types, and only the new code uses the extended type. This is no different from adding an entirely unrelated new type to an existing program.

As educators we need to ask whether embedding such thinking in future professionals is indeed part of our goals for our students and woven into our modus operandi when teaching. Again, only part of this comes down to tools. The right mindset, though helped by supportive tools, does not utterly depend on them for that assistance.

It is, however, worth asking whether the tools with which we teach support:

- extensible records;

- library defined types that can be extended by another library, and have their functionality automatically available to the type;

- An I/O system whose built-ins are themselves extensible to any new data type so that, for instance,

```
WRITE (myData);
```

automatically translates to

```
myDataType.Write (myData);
```

and the compiler will complain if it does not find the latter has been made visible in the current scope by the appropriate import;

- a FOR loop that will automatically iterate over a user data type according to the code it finds under MyDataTpye.For, (and likewise for other functionality that may be type specific such as allocation and deallocation of memory);

- library based generics for common data structures such as queues, stacks, arrays, assorted trees, and other common dynamic types, as well as for oft-used algorithms such as search and sort,

where all these can be refined for any data type with a simple macro, avoiding the necessity of rewriting code known to work with one data type and possibly introducing errors in the doing;

- a language specification requiring unsafe and system specific libraries be separated from all other functionality in supplied libraries.

All these, by the way are features of Modula-2 R10.

5.8 Having Our Cake and Eating it Too

Every intellectual pursuit, computing science as much as any, presents both theorists and practitioners with choices, "forks in the road." Two aspects of the discipline's epistemology or technique are presented by rival schools as mutually exclusive ways of encapsulating how to think or how to do, and a "go left" vs "go right" debate ensues--one that rarely settles anything except to create a belief that one must do one or the other, for they are mutually exclusive. Faith versus deeds, love versus truth, conservative versus progressive, teaching versus discovery learning, phonics versus whole language, wave versus particle, egalitarian versus complementarian--examples could be multiplied from every discipline.

These all have in common that two schools of thought or practice often arise around what are considered polar opposites, or mutually exclusive definitions of the (sub) discipline--each with its own list of advantages of their school and the disadvantages of any other. Some of this thinking is encouraged by the very compartmentalization of specialities. Students study an ever narrowing field until they produce a thesis so constrained in its vision that although few people will read it, and fewer understand it, the writer will forever after filter their entire discipline through the lens of the "big idea" they put forward in that thesis. All else is to an extent in the "other" camp.

This pervasive dualism can be summed up by either, "there are only two ways to think about this. Pick one.", or "You can't have your cake and eat it too," or in worse cases, "When I am in charge, not only will I do as I please, but you will also do as I please."

True innovation usually comes by following a new path, either "both-and" or "none of the above, but a third way, neither left nor right, but straight ahead."

For instance, conventional development wisdom claims that you can either do it safe or fast but not both; be safe or flexible but not both, safe or low-level but not both. The fork in the road tends to be convenience versus safety and the dominating school teaches that the advantages of convenience outweigh the disadvantages of deprecating or ignoring safety, especially when labouring under management's dictates for getting a product out the door.

The authors do not believe that convenience should come before safety but we know that without convenience, few people will follow a path towards safety, so we had to reject the notion that one can only have one or the other but not both.

In other words, a design, whether of a project or a tool, may well start with a clean sheet asking first "what is it that we really want" instead of asking first "what is it we can do that the unspoken rules of one of two schools of thought permit". We should draw up solution designs and refine them into working code according to what satisfies our true requirements, rather than what spoken or unspoken rules would have allowed us to do.

For instance, in revisiting the language Modula-2, the authors:

- knew that conventional wisdom said you can either have variadic parameters or safety but not both. We have shown that this is false as our design provides both.

- knew that conventional wisdom was you can either have variable length array members but not bounds checking, thus such dynamic structures were always unsafe. We have shown that this too is false. In Modula-2 R10, type safe variable length array fields can be used in record type definitions.

- knew that conventional wisdom said you can either have safe generic IO built-in but not library defined (Pascal), or safe library defined IO, but not generic (classic M2), or generic library defined IO but not safe (C). Modula-2 R10 has type safe generic IO, yet it is library defined.

- knew that conventional wisdom said data types built in to the language had to be treated fundamentally differently than ones defined by the user. Modula-2 R10 blurs the line between the two almost to the point of nonexistence.

- knew that conventional wisdom said that I/O functionality could either be in the language proper and restricted to built-in types, or placed in libraries and imported. Again, we differ. Our I/O is triggered by versatile READ and WRITE statements that are built in macros, whose functionality is picked up automatically by the compiler from the library where the data type and its I/O are defined. Moreover, they default to the standard input and output channels if none are specified. We have separated the visibility of I/O from its functionality, thereby combining versatility, convenience, simplicity and safety.

- knew that conventional wisdom said that opaque data types had to be pointers, target types whose details hidden in an implementation because run time memory allocation requires the compiler know the size of the target. We continue Wirth's use of such types, but instead of simply providing an empty definition, we use OPAQUE as a visibility marker for readability. However, in addition, we also permit the use of an asterisk as an opaque flag on individual record fields in a definition module. This allows the compiler to compute the allocation sizes of those fields while hiding them from clients. We have therefore completely overcome this limitation.

Many more examples can be found by scanning our language report. The point is that students also need to be taught (and allowed to practice the discipline with) a mindset that is open to finding the "way straight ahead" even when faced with schools of thought that collectively say there's no road there. We have shown that it is possible to have safety, reliability, and efficiency in a single language.

5.9 A New Mindset for a New Reality

Another truism the authors challenge is the idea that fallacy that software is different from other products and simply cannot be built reliably.

Sooner or later legislators, regulatory bodies, and the legal system are going to realize the falsity of this, and both developers and their managers will discover they cannot get away with bad planning, poor design, inappropriate choice of tools and methods, sloppy coding, worse documentation, little or no testing, in short, all-round incompetence.

The authors have attempted to attack some of the issues with tools with a focus on safety, security, and reliability. Another important piece of work of late in this area is Tannenbaum's redesign of the

Minix operating system [69] which is designed to be fault tolerant and secure. But tools alone are far from the whole picture--they do address the "how" but not the "why". Tools will still be abused if the mindset is not re-landscaped for the new realities of enforced professionalism backed up by required credentials and enforced with the threat of malpractice suits when faulty software is delivered despite there being well-known and thoroughly documented means of doing it correctly. Among other things we believe that:

- if software is infrastructure, it ought to be designed, built, managed and regulated as such;

- this requires a change of mindset from "software is different, can't be done properly" to "software is not different, classical engineering can be applied" in education, industry, law, and politics;

Projects like ours aim to (a) show how this can be accomplished technically, (b) cultivate the mindset in science education and industry and (c) inform the public (including policy makers).

5.10 Development as Craft

To the extent that software is a tool for problem solving, using it becomes a craft, the opposite end of the theory-to-practice spectrum. Craft work takes planning, time, effort, diligence, skill, dedication, the best tools, testing, and polishing. Michelangelo didn't take a six week course in painting sculpting, then "hack together" a quick and dirty Sistine Chapel Ceiling or statue of David. Machinists hand turn precision parts for precision purposes with precision tools. There is almost no tolerance for error, and removing just a little too much material can waste days of work, forcing a restart of the project. The same is true of a chef in a large establishment, where waste avoidance requires project management skills.

Trade craft requires a combination of education, apprenticeship under a master or journeyman, licensing, periodic inspection against standards, and ongoing training. Mistakes have a cost, at first just warnings by the master, but eventually deductions from pay. Ruined materials must be paid for by someone. This applies to metal workers, electricians, mechanics, plumbers, chefs, bakers, butchers, and so on. They all have one thing in common. They are subject to a world where physics, legal regulations, and economics directly impact their working habits. Waste has a price, so one must both plan and act to minimize it.

By contrast, in the world of IT, both prerequisites for and the consequences of practice are generally ignored. The waste of hundreds of millions to billions of dollars and years of human resources is tolerated, excused, even expected. Cronyism being what it is, contracts to replace failed software are often let to the same people, the practice slightly obscured by a change of company name. But if software development were approached as real world tradecraft, its practitioners would work with an entirely different mindset--one conscious that failure to meet specifications has personal and corporate consequences.

One possible approach to education and training for software developers would be to admit only those who already had a trade involving making things with their hands. Another would be require the same kind of regimen as for an electrician, a chef, a carpenter, or a machinist--classroom education paralleling closely supervised apprenticeship, licensing and periodic inspection and upgrading. Yes, computing science is worthy of study as an academic discipline, but there is a strong case to be made that it ought not be practiced with either an academic or a hobbiest mindset. Likewise, a masters designation, unless intended to lead to a research career, ought to be reserved for those with years of successful practice.

5.11 Development as a profession

We may need to go farther.

Modern buildings and the multiple complex systems they contain are not just thrown together. Neither are automobiles or airplanes. There are no amateur physicians, dentists, police, physics teachers, or investment bankers. Professional work has two things in common: It requires training, and, once more, there are consequences to making mistakes. Firefighters, ambulance attendants, and lifeguards, may have lower hurdles to pass in order to work, but also require training, and in most jurisdictions, licensing.

Engineers are not permitted to design, nor are contractors permitted to construct buildings, roads, railways, pipelines, vehicles, airplanes, tunnels, sewers, and bridges as they wish. They must follow a stringent regime of regulations based on known best practises. Failing to do so constitutes at best actionable malpractice, and at worst criminal negligence. There are standards for work in forestry, mining, law, medicine, education, research, money management, first aid, consumer goods, machine parts, electrical components, and for many other professions and commodities. Why? Because we deem the physical or human infrastructure they create and maintain too important to fail.

Why is software development thought of so differently? Why is it that software failure is considered an acceptable risk of doing business? [7] More to the point, for how long will this continue to be the case? Computer software is a tool, but has become much more than that--all the above professions depend on it. Software has come to dominate almost every aspect of human endeavour. Life, health, transportation, business and nearly every human endeavour and institution depends on it. It is now infrastructure, like buildings, roads and bridges.

Consequently, it is too important to be allowed to fail. Thus, we have no choice but to begin treating software development as we do any other infrastructure development and maintenance. It has to be professionalized, that profession regulated, and its practitioners not only properly trained but insured against malpractice.

If the percentage of airplanes that crashed, bridges that collapsed, surgery patients that died because of surgeon errors, money managers that absconded with the funds, food that was contaminated or spoiled, machine parts that didn't fit, and/or buildings that were either never completed or uninhabitable when they were, was even close to the twenty or thirty percent earlier mentioned for failed software projects, our very civilization would be imperilled.

Given that all these and more now depend on fully functioning software, the days in which software development failures are excused, the waste written off without consequence, must end or we will face a breakdown of civilization. Software production has to be held to higher standards.

Since our graduates have minimal initial influence on the larger environment in which they work, and may not even be able to choose their own tools, our emphasis must first and foremost be

on instilling an appropriate discipline--a trade craft mentality further restrained by a professional regimen and regulation. That will have the most significant impact.

Thus, for our part of the solution at the educational level, there is indeed a silver bullet: teach classic professional engineering principles and tradecraft, and instill the self-discipline to apply them habitually.

Unfortunately for the large number of code slingers out there who don't have this kind of education, there is no silver bullet. They are a lost generation. The only thing that could help them is a complete re-education, but unlearning bad habits is far more difficult than developing good ones in the first place.

We can do better. We must. And it starts in the classroom.

References

[1] Kowarsch, B. and Sutcliffe, R. J. 2016. Modula-2 R10 Repository. https://bitbucket.org/trijezdci/m2r10/.

[2] Kowarsch, B. and Sutcliffe, R. J. 2016. Modula-2 R10 Information Site. http://www.modula-2.info.

[3] Chua, A. Y. K. 2009. Exhuming IT projects from their graves: an analysis of eight failure cases and their risk factors. The Journal of Computer Information Systems. 49, 3, 31-39.

[4] Charette, R. N. 2005. Why software fails. IEEE spectrum. http://www.rose-hulman.edu/Users/faculty/young/OldFiles/CS-Classes/csse372/201310/Readings/WhySWFails-Charette.pdf.

[5] Purao, S. and Desouza, K. 2011. Looking for clues to failures in large-scale, public sector projects: A case study. 44th Hawaii International Conference on System Sciences (HICSS).

[6] Goldstein, H. 2005. Who killed the virtual case file? [case management software]. Spectrum.

[7] Ewusi-Mensah, K. 2003 Software development failures. MIT Press.

[8] Linberg, K. R. 1999. Software developer perceptions about software project failure: a case study. Journal of Systems and Software. 49, 177-192.

[9] Shaw, R. and Culbert, L. 2015 12 14. Major B.C. government IT projects go over budget or end up missing key features. Vancouver Sun.

[10] Shaw, R. and Culbert, L. 2014 05 29. The B.C. government's $182-million computer system just won't work. Vancouver Sun.

[11] Sherlock, T. 2015 09 21. Another year, another computer foulup affecting B.C. teachers. Vancouver Sun.

[12] Sherlock, T. and Crawford, T. 2015 08 03. B.C.'s auditor general says technology used by public health system inefficient. Vancouver Sun.

[13] Shaw, R. and Culbert, L. 2015 12 15. The province and computers: Finding out what works and why. Vancouver Sun.

[14] Shaw, R. 2014 07 11. B.C. alone in using troubled software system to manage child welfare. Vancouver Sun.

[15] Shaw, R. and Culbert, L. 2015 12 15. B.C. not alone in bungling computer projects. Vancouver Sun.

[16] Verner, J., Sampson, J., and Cerpa, N. 2008. What factors lead to software project failure. Research Challenges in Information Science. RCIS 2008. Second international Conference on. 71-80.

[17] Williams, T. C. 2011 Rescue the Problem Project: A Complete Guide to Identifying, Preventing, and Recovering from Project Failure. American Management Association.

[18] Singpurwalla, N. D. and Wilson, S. P. 1994. Software reliability modeling. International Statistical Review. 62, 3, 289-317.

[19] Phipps, G. 1999. Comparing observed bug and productivity rates for Java and C Software. Software - Practice and Wxperience. 29, 4, 345-358.

[20] Tiwari, V. and Pandey, R. K. 2012. Some Observations on Bug Fixing Process and Defect Density of Open Source Software. International Journal of Advanced Research in Computer Science. 3, 1.

[21] Khomh, F., Dhaliwal, T., and Zou, Y. 2012. Do faster releases improve software quality? an empirical case study of mozilla firefox. Mining Software.

[22] Chomal, V. S. and Saini, J. R. 2014. Significance of Software Documentation in Software Development Process. International Journal of Engineering Innovations and Research. 3.4, 410-416.

[23] Zhi, J., Garousi-Yusifoğlu, V., Sun, B., and Garousi, G. 2015. Cost, benefits and quality of software development documentation: A systematic mapping. Journal of Systems and Software. 99, 175-198.

[24] van Loggem, B. 2014. Software documentation: a standard for the 21 st century. ISDOC '14 International Conference on Information Systems and Design of Communication. 149-154.

[25] Arraki, K. S. 2016. Source code management with version control software. American Astronomical Society Meeting Abstracts. http://adsabs.harvard.edu/abs/2016AAS.22712701A.

[26] Gajda, W. 2013 Working with Well-Known Repositories. In Git Recipes, Springer.

[27] Miriam Quick, Ella Hollowood, Christian Miles and Hampson, D. World's Biggest Data Breaches. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/.

[28] Gollmann, D. 2010. Computer Security. Wiley Interdisciplinary Reviews: Computational Statistics. 2, 5, 544-554. DOI=10.1002/wics.106.

[29] Tian-yang, G., Yin-sheng, S., and You-yuan, F. 2010. Research on software security testing. World Academy of Science, Engineering and Technology. 70.

[30] Chess, B. and Arkin, B. 2011. Software security in practice. IEEE Security & Privacy. March/April.

[31] Nunes, F. J. B. and Belchior…, A. D. 2010. Security engineering approach to support software security. Services (SERVICES-1) 2010 6th World Congress on. 48-55.

[32] Myers, G. J., Sandler, C., and Badgett, T. 2015 The art of software testing. Wiley.

[33] Al-Ahmad, W., Al-Fagih, K., Khanfar, K., Abuliel, S., and Abu-Salem, H. 2009. A taxonomy of an IT project failure: Root Causes. International Management Review. 5, http://search.proquest.com/openview/367d4e172fb56040e736cd2d5b6ae55b/1?pq-origsite=gscholar.

[34] Galorath, D. 2008. Software Project Failure Costs Billions… Better Estimation & Planning Can Help. Project Management.

[35] France, R. and Rumpe, B. 2007. Model-driven development of complex software: A research roadmap. Future of Software Engineering.

[36] Evans, E. 2004. Domain-driven design: tackling complexity in the heart of software. books.google.com. http://www-public.tem-tsp.eu/~gibson/Teaching/CSC7322/ReadingMaterial/Evans03.pdf.

[37] Jennings, N. R. 2001. An agent-based approach for building complex software systems. Communications of the ACM. 44.4).

[38] Kerzner, H. R. 2013 Project management: a systems approach to planning, scheduling, and controlling. Wiley.

[39] Royce, W. 1998. Software project management. Pearson Education India.

[40] Burke, R. 2013. Project management: planning and control techniques. cupa.ir.

[41] Curtis, B. and Walz, D. 2014 The Psychology of Programming in the Large: Team and Organizational. Academic Press.

[42] Dooley, J. 2011 Software development and professional practice. Springer.

[43] Wirth, N. 1996. Recollections about the development of Pascal. History of programming languages---II.

[44] Morris, R. 2009. Niklaus Wirth: Geek of the Week. http://fruttenboel.verhoeven272.nl/modula-2/data/NikGeek.pdf.

[45] Lindsey, C. H. 1996. A history of Algol 68. History of programming languages---II.

[46] Böszörményi, L., Gutknecht, J., and Pomberger, G. 2000. The school of Niklaus Wirth: the art of simplicity. books.google.com.

[47] Wirth, N. 1982 Programming in Modula-2. Springer-Verlag.

[48] Wirth, N. 1983 Programming in Modula-2. Springer-Verlag.

[49] Wirth, N. 1985 Programming in Modula-2. Springer-Verlag.

[50] Wirth, N. 1988 Programming in Modula-2. Springer-Verlag.

[51] Andrews, D. J., Cornelius, B.J., Henry, R.B., Sutcliffe, R.J.,Ward, D.P., Woodman, M. 1994 Information technology – Programming Languages – Modula-2 International Standard (ISO/IEC 10514-1). ISO.

[52] Sutcliffe, R. J. 1997 Information technology – Programming Languages – Generic Modula-2 (ISO/IEC 10514-2). ISO/IEC.

[53] Henne, E., Wiedemann, A., Woodman, M., Lancaster, J. 1996 Information technology – Programming Languages – Standard Object Oriented Modula-2 (ISO/IEC 10514-3). ISO/IEC.

[54] Sutcliffe, R. J. 1987 Introduction to programming using Modula-2. Merrill.

[55] Eisenbach, S. and Sadler, C. 1989 Program design with Modula-2. Addison-Wesley.

[56] Gabrini, P. J. and Kurtz, B. L. 1997 Data structures and algorithms with Modula-2. Jones and Bartlett Publishers.

[57] Helman, P. and Veroff, R. 1988 Walls and Mirrors : Intermediate Problem Solving and Data Structures. Benjamin/Cummings Pub. Co.

[58] Sincovec, R. and Wiener, R. 1986 Data structures using Modula--2. Wiley.

[59] Sutcliffe, R. J. 2005. Modula-2: Abstractions for Data and Programming Structures (Using ISO-Standard Modula-2). http://www.arjay.bc.ca/Modula-2/Text/index.html.

[60] Pronk, C. and Sutcliffe, R. J. 1997. Scalable Modules in Modula-2. Modular Programming Languages.

[61] Pronk, C. S., R. March 19-21, 1997. *Scalable Modules in Generic Modula-2*. Joint Modular Languages Conference at Johannes Kepler University, Linz, Austria. Lecture Notes Series in Computer Science #1204.

[62] Pronk, C. and Schönhacker, M. S., Richard J. & Wiedemann, Albert Nov 1997. Standardized Extensions to Modula-2. SIGPLAN Notices.

[63] Pronk, C., Schönhacker, M., and Sutcliffe, R. J. W., Albert Oct 2000. Extensions to the language Modula-2. Journal of Object Oriented Programming.

[64] Brooks, F. P. 1978. The Mythical Man-Month: Essays on Software. 1st ed. Addison-Wesley Longman.

[65] Tanenbaum, A. 2009 Modern operating systems. Pearson Education International.