Software Engineering Is Not Enough

James A. Whittaker and Steven Atkin

Abstract—Much of the software engineering literature begins with an admonition that what practitioners are doing isn't enough—that the state-of-the-practice is creating bad software. This paper does not dispute this fact. However, we believe that what the software engineering literature offers as solutions is also not enough. Books on the subject favor the "light" side of the discipline: project management, software process improvement, schedule and cost estimation and so forth. The real technology necessary to actually build software is often described abstractly, given as obvious or ignored altogether. But software development is a fundamentally *technical* problem for which management solutions can only be partially effective. This paper argues this point and then describes a basic set of technology that real software developers apply before, during and after designing real software, often against unrealistic schedule and budgetary constraints.

Introduction

Imagine for a moment that you know nothing about software development and to learn about it you pick up a book with the words "Software Engineering," or something similar, in the title. Certainly one might expect that software engineering texts would be about engineering software. Could you imagine drawing the conclusion that writing code is simple? That it is nothing more than translation of a design into a language that the computer can understand? Well, this conclusion might not seem so far-fetched when it is supported by an authority:

> "The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded." [6, p. 346]

Really? How many times does the design of a non-trivial system translate into a programming language without some trouble? The reason we call them *designs* in the first place is because they are *not* programs. The nature of designs is that they abstract many of the details that must eventually be coded. Many notorious software bugs are the result of overlooked or ill-understood details [3]. If details really were unimportant, designs would be compilable and very little programming by humans would be necessary.

However, automatic code generation from designs is possible only in very limited application domains (e.g., databases); for most problem domains human programmers are the norm. Programming remains monstrously complex for the vast majority of applications. Programming is so complex that developers can work for years in a single language and still not learn all of its nuances. But it is often the case that developers don't get the time it takes to master a language because they must learn a new language every time technology and platforms change. Such change ensures that mastery of language and platform details is *not* a given and, therefore, a good design does not ensure a good program.

The problem with the design-only focus of modern software engineering texts is that they encourage software developers to gravitate to the latest design fads, assuring us that good code will follow. Our concern is that this drives developers further away from the technical foundations of their work. We lament the sad preference for the latest fashionable design technologies and the seeming aversion to the fundamental technical skills like systems programming and debugging that form the core of software development. How can practitioners be expected to take these fundamentals seriously when they fail to get more than cursory treatment of them in the literature?

Could you imagine drawing the conclusion that the only good program is a simple program?

> "Simplicity, clarity and elegance are the hallmarks of good programs; obscurity, cleverness, and complexity are indications of inadequate design and misdirected thinking." [2, p. 192]

In our experience, the only programs that are simple and clear are the ones we write ourselves. When you have written a program in its entirety you have forced yourself to understand every aspect of the problem and the solution. Of course the program will seem simple and easy to read. But unless the problem being solved is so simple and well understood that it cannot be obfuscated, programs are complex! Have you ever seen a simple compiler? A simple

J. Whittaker is an associate professor of computer science at the Florida Institute of Technology (jw@cs.fit.edu), S. Atkin is a staff software engineer with the IBM Corporation, Austin, Texas (atkin@us.ibm.com).

operating system? A simple 64 bit encryption algorithm? Just because code is complex does not mean it is wrong. Complex problems often demand complex solutions.

Instead of allergic reactions to complexity, true software engineers react instead with suspicion. Perhaps simplification or refactoring techniques can be applied, but then again, some software is simply complicated because the problem it automates is complicated. We may be frustrated but we must also be realistic.

You may then decide to consult some older references to look for more timeless wisdom.

> "Many programs don't need flow charts at all; few programs need more than a one page flow chart." [1, p. 167]

Could you imagine drawing the conclusion that documentation can be excessive or even unnecessary? As we just stated, real programs are complex by nature. There is rarely such a thing as too much documentation. When you must maintain code you did not write, the documentation is often your only chance to make successful changes to the code. Imagine having to modify an encryption key management program without any documentation! Designs do not and cannot cover all the details. The only hope for understanding programs is good documentation of control structure blocks and detailed description of the purpose and use of data structures. Moreover, effective documentation often must go beyond this detail and include design rationale and, even more important for maintainability, the reasons against alternative designs. Documentation—often exceeding the source code in size—is a requirement, not an option.

Finally, you might decide that you are simply reading the wrong section of the book and try to find the sections that cover coding. A glance at the table of contents, however, shows that there are few other places to look. For example, in McGraw Hill's best selling software engineering text [6] there is not a single program listing. Nor is there a design that is translated into a program. On the other hand, the book is replete with project management, cost estimation and design concepts. Prentice-Hall's best seller [5] goes further by dedicating 22 pages to coding. This is slightly more than 4% of the book's 543 pages.

One might draw the conclusion that the act of coding is a very small part of the software engineering process. Search all you want but you will not find the answers to your coding problems in the pages of most software engineering books.

We believe that the reason software development gets so little attention in the software engineering literature is that it is the hardest and least understood part of the software engineering lifecycle. Coding is immensely difficult without a good design and extremely difficult with one. Maintaining code is next to impossible without good documentation and formidable with it. But there are some things that designers can do to ease development and maintenance that are *technical* in nature. This paper discusses a few of these and it is our hope to encourage dialog and debate between the people who write software and the people who write about software.

Considerations Before Design

Practitioners understand the importance of software engineering methodologies but recognize that there is no magic method that will guarantee a well-engineered product. This is especially true before design begins when no methodology is available to help.

There are two activities that developers must pursue before design: familiarizing themselves with the problems they are to solve (we'll call this problem domain expertise) and studying the tools they will use to solve them (we'll call this solution domain expertise). Expertise in both domains is crucial for project success.

Understanding the problem domain is a very difficult task and often having problem domain experts on a project is the difference between success and failure. Imagine writing a matrix algebra library without expertise in mathematics, coding an ARP cache without detailed knowledge of routing protocols or building a flight simulator without understanding how to fly an airplane.

Learning the problem domain means more than simply talking to users and gathering requirements. Problem domain expertise is intensely technical in nature and requires significant study of software environments, low-level protocols and domain conventions. Software engineering methodologies do not and cannot teach this art. It takes hard work, lots of study and experimentation, and usually years of experience. Seeding your team with problem domain experts

is one of the best things you can do to increase your chances of success.

The solution domain is more focused and essentially consists of the tools that a team of developers and testers will employ in building the software product. The barest minimum set of tools is one or more editors, compilers, linkers and debuggers (symbolic and kernel), the fewer in number the better. Add to this set make-utilities, run-time libraries, development environments, version control managers and, of course, the operating system and you have quite a number of complex tools that a developer must master before a software engineering methodology can even be used.

Software engineering doesn't help here either. Developers must be masters of their programming language and their operating system, methodology alone is useless. It is easy to misuse a programming language by selecting the wrong data structure or by using an unsafe built-in function. It is likewise easy to overlook features of the operating system that are not well-understood by participating developers. Understanding when to refresh a window, how to handle certain error codes or when to launch independent execution threads has nothing to do with design and everything to do with detailed knowledge of the solution domain. Developers who are inexperienced in a language or an operating environment have little hope of engineering good code regardless of their methodology and management practices.

Selection and proper use of tools is another issue that gets light treatment in the software engineering literature. Before design even starts, it is a good idea to settle on as few tools as possible. Everyone should use the same editor, the same compiler and linker (with the same settings and environment variables) so that code can be properly integrated and maintained. "Integration purgatory" is the cost of not doing so. If each module checked into a build was compiled with a different compiler using any settings and flags the developer desired, they may not integrate smoothly and costly rework and retesting must ensue. Even worse is the maintenance problem where one-line changes can take hours to recompile because the original settings used for the previous compile were lost or forgotten. The tools that were once employed to solve a problem now *become* the problem.

Many software engineering books highlight a different type of tool that most practitioners shy away from: CASE tools. The problem with such tools is that since they are software, they have the potential to produce buggy output. Writing your own bugs is bad enough, inheriting bugs from your tools is the epitome of frustration. Rework and work-arounds are very costly.

Simply put, pre-design activity defies methodology. However, the technical work performed in understanding one's problem and solution domains can be the difference between project success and failure.

Programming is Easy, Isn't It?

There are over 1200 calls in the Windows® 2000 kernel.

There are over 20,000 calls in the Win32 API. The printed reference manual for just the calls and their parameters is over 4 inches thick.

A typical runtime library for C has over 10,000 functions. C itself has hundreds of built-in functions, operators, and reserved words.

Besides the above libraries, applications routinely use dozens of other libraries of similar complexity.

It is very likely that thousands of bugs exist in each of these libraries.

Your mission, should you choose to accept it, is to go out and write a decent program.

We wish you well.

Considerations During Design and Coding

New problems surface once design and coding begins. The first hard lesson that developers learn is not to trust their environment. A software's environment consists of users that provide input and process output. For example, humans are users who provide keystrokes and mouse clicks and process screen output, the operating system and external code libraries are users that provide system resources via function calls, and so forth. Developers who trust that humans will always enter the input they expect, and that operating systems never run out of resources, write code that only works when everything goes as planned. The result is software that works well for only the most careful and deliberate users in the most perfect system configuration. The rest of us are stuck with buggy behavior whenever we stray from the well-beaten path established by the original developers.

Good developers understand that inputs from users cannot be trusted. Each time an input enters the system it must be validated to prevent failure or corruption of internal data (which will eventually lead to failure). This means each time data is passed from an interface control to the main functional code, it must be checked for validity before it is stored or used in computation. Each time a field is read from a file we must ensure it is of the appropriate type and that the value is within an acceptable range. Anytime we fail to perform such validation we are risking program failure.

Deciding which inputs to trust and which to validate is a constant trade-off. Input validation costs valuable CPU cycles and can slow down an application. Experienced developers have a good feel for which system calls rarely fail and which interface controls provide reliable data and write their programs to balance speed vs. risk.

But even if all input is valid, internal data can still get corrupted. Consider, for example, the addition of two short (two byte) signed integers *a* and *b*. Validation of *a* and *b* is necessary in order to avoid overflow, however, we must also constrain their combination. Suppose a user enter the values 32000 and 1000 for *a* and *b*, respectively. If we sum the two and try to store the result in another short signed integer then the result overflows because 33000 is larger than the maximum signed two byte integer value 32767. Valid data, invalid result.

Developers quickly learn that there are two parts to any program. The code that performs the desired function and the code that handles failure. We are pretty good at coding the main functional code, handling failure is the thing we do poorly.

How do we handle failure? A good place to begin to answer this question is to ask ourselves where can a program fail? Certainly any interaction with our environment could be risky, humans are unpredictable, system resources can fluctuate and files can be corrupt. Thus, we need to program constraints into our code that ensure every input is expected and every output can be processed by our users.

But this isn't enough as we discussed above, software can fail internally also. Inside a program is essentially two things: data and computation. Both of these things can fail and therefore require constraints just like inputs and outputs.

Constraints on input and output are programmed in a number of ways. For GUIs, interface controls take much of the workload by filtering out incorrectly typed data and data out of the acceptable range. But if you are coding a solution with a programmatic interface, all the error checking is your own responsibility. Each parameter of each call must be painstakingly validated, often meaning lots of "if" statements and calls to validation routines.

Constraints on data and computation usually take the form of "wrappers"—access routines (or methods) that prevent bad data from being stored or used and ensure that all programs modify data through a single, common interface.

Unfortunately, there is little support for programming constraints beyond "if" statements and access routines in most modern programming languages. Thus, most developers rely on exception handlers. Raising exceptions is not the same thing as programming constraints. Constraints actually prevent failure, exceptions, on the other hand, allow the failure to occur and then trap it so that it can be handled by a special routine provided by the developer.

When developers use exceptions (which most do because there is little choice in the matter) they must determine *how to fail*. Recovery from a failure is often very difficult. The state of the application at the time of the failure must be known so that the program can properly react. If files are open, the program needs to be aware of this so that it doesn't try to re-open a file. If an operation didn't complete when the failure occurred, we need to be able to figure out how much of the operation succeeded and how much is left to do. The exception handler must be able to take stock of the application and recover appropriately, otherwise it may fail too: it may write to a file that doesn't exist, use data that didn't get initialized, etc. And if the exception handler is sloppy, what is going to handle the exception when the exception handler crashes?

Design and coding present extreme technical challenges that defy methodology and stretch the capabilities of modern programming languages. In addition to designing the main functional code, developers must write constraint code and consider how their program can fail gracefully. Failures will occur, how developers design and code against them is, unfortunately, not a part of mainstream software engineering.

Plenty of Opportunities to Fail

Too often we think of program inputs only in terms of our interaction with an application through its GUI. But sys-

tem inputs are much more pervasive and provide a much larger input validation problem [1].

For example, Microsoft® PowerPoint®, a complex and large application for making presentations and slide shows makes nearly 800 calls to 30 different functions of the Windows® kernel upon invocation. That means a single input from a human user (invoking the application) causes a flurry of undercover communication to be transferred to and from the operating system kernel.

Certainly, invocation is a special input and requires a great deal of setup and data initialization. But other operations also are demanding on low-level resources: when PowerPoint opens a file, 13 kernel functions are called nearly 600 times; when PowerPoint changes a font, 2 kernel functions are called a total of 10 times.

And these are only calls to the operating system kernel. PowerPoint also uses a number of other external resources (dynamic linked libraries), including mso9.dll, gdi32.dll, user32.dll, advapi32.dll, comctl32.dll and ole32.dll, in the same manner as the kernel. It is easy to see that the amount of communication between an application and the operating environment dwarfs GUI input.

Unless all of these calls are validated then a single bad return code can bring the application to its knees. Perhaps the next time you use desktop software, you might consider the complexity of the operations the product is performing.

1. J. A. Whittaker, "Software's Invisible Users, *IEEE Software*, Vol. 18, No. 3, pp.84-88, 2001.

Considerations After Design and Coding

After design and coding the software is built, debugged and executed. Modifying an existing system is different than "clean sheet" development. The code is already written and chances are it is buggy. Furthermore, unless you are the original developer, it is also hard to read and understand. It is in this very situation that developers are handed problem reports and asked to diagnose possible bugs and fix them.

The tools to help with this task are source debuggers and other low-level system tools. All good developers depend heavily on these tools. Indeed, it is impossible to get along without them for many reasons.

The most important of these is in failure reproduction, called the "repro problem" because it is not the no-brainer it seems to be. The situation is that a tester (or user) finds a problem and reports it. When the developer gets it, he or she can't reproduce it in their environment. Why? Well, because lots of things are happening under the hood of software that aren't visible to humans who are watching the software through a GUI. For each input a human user applies, dozens of system calls and calls to reused components are being made. Usually, users only report their own input, i.e., "it broke when I did such-and-such." Developers need their system tools to tell them exactly what was happening when those inputs were applied. Which system calls got executed? Which ones failed? What were the return codes? Without this information the hope of diagnosing non-reproducing problems is slim. Developers must learn to effectively use their system tools or they are operating with blinders on—seeing only the GUI and not the system interfaces.

But system tools don't help much when code has to be modified to add new functionality. Adding new behavior to existing code is difficult to say the least. Two pieces of advice that experienced developers share when they mentor novices: don't trust comments and be wary of side effects when you modify code.

It may seem a contradiction that comments can't be trusted because we debunked the idea earlier that all code was simple enough to do without them. Comments are, indeed, important and some complex programs are not maintainable without them. However, the advice still stands. Comments are not to be explicitly trusted, developers should use them to understand the code but not as a surrogate for the code. All-too-often comments are not updated as the software is modified. It is too easy for developers to update code and not bother updating the comments. The result is that the comments are describing code that no longer exists. So trust the comments only as an aid to understanding the code; but never forget that only the code has to be up-to-date, the comments do not.

Side effects are the other main risk when modifying code. Maintenance developers must be wary of possible side-effects when changing code. If a change requires modification of data, then one must ensure that all programs that rely on that data can still function properly after it is changed. The worst maintenance sin is to break something that used to work.

Checking not only that the fix was successful but also that no other dependent functionality was broken is a formida-

ble technical task. One that modern software engineering addresses incompletely.

The Making of a Maintenance Nightmare

Another technical hazard during maintenance is that in the frenzy of fixing bugs, maintenance developers often forget the design principles that they routinely apply as they create code from scratch. The following code published on a popular web repository is what we believe to be one such example.

```
typedef struct _DCB {
   DWORD DCBlength;              /* sizeof(DCB)                */
   DWORD BaudRate;               /* Baudrate at which running  */
   DWORD fBinary: 1;             /* Binary Mode (skip EOF check) */
   DWORD fParity: 1;             /* Enable parity checking     */
   DWORD fOutxCtsFlow:1;         /* CTS handshaking on output  */
   DWORD fOutxDsrFlow:1;         /* DSR handshaking on output  */
   DWORD fDtrControl:2;          /* DTR Flow control           */
   DWORD fDsrSensitivity:1;      /* DSR Sensitivity            */
   DWORD fTXContinueOnXoff: 1;   /* Continue TX when Xoff sent */
   DWORD fOutX: 1;               /* Enable output X-ON/X-OFF   */
   DWORD fInX: 1;                /* Enable input X-ON/X-OFF    */
   DWORD fErrorChar: 1;          /* Enable Err Replacement     */
   DWORD fNull: 1;               /* Enable Null stripping      */
   DWORD fRtsControl:2;          /* Rts Flow control           */
   DWORD fAbortOnError:1;        /* Abort all reads and writes on Error */
   DWORD fDummy2:17;                 /* Reserved               */
   WORD wReserved;               /* Not currently used         */
   WORD XonLim;                  /* Transmit X-ON threshold    */
   WORD XoffLim;                 /* Transmit X-OFF threshold   */
   BYTE ByteSize;                    /* Number of bits/byte, 4-8   */
   BYTE Parity;                  /* 0-4=None,Odd,Even,Mark,Space */
   BYTE StopBits;                /* 0,1,2 = 1, 1.5, 2          */
   char XonChar;                 /* Tx and Rx X-ON character   */
   char XoffChar;                /* Tx and Rx X-OFF character  */
   char ErrorChar;               /* Error replacement char     */
   char EofChar;                 /* End of Input character     */
   char EvtChar;                 /* Received Event character   */
   WORD wReserved1;              /* Fill for now.              */
} DCB, *LPDCB;[1]
```

This is an example of the all-inclusive type. This type most likely came about incrementally. Originally, maybe only four fields were needed. As time went on developers needed this type along with some other data for some new function or bug fix. So rather than introduce a new type they just appended their fields onto the existing type, opening the door to possible side effects (because there is more shared data to be misused) and security issues (because the entire structure is exposed to users whether they need the data or not). It is highly unlikely that all the fields are related and are always required by functions using this type, however, the pressure to quickly fix a bug often corners maintenance developers into such “quick” fixes.

Constant Considerations

If there was one thing that all developers would learn and never forget, we would want it to be the following: eventually someone else is going to have to modify their code. If this were the case then developers would be guided by 1) the motivation to minimize other programmers needing to modify their code and 2) a desire to make their programs as maintainable as possible. The former motivation would mean that programmers would endeavor to write code that is easy to use and hard to break. Both of these have the effect of reducing the need for rework. The latter motivation would ensure that the code is readable, that data is named appropriately and that the code has informative, up-to-date

[1] This code is copied verbatim from a nameless vendor’s web site.

comments [4].

If only developers never forgot about maintenance, we would all be better off.

Simple Code Is Not So Simple

Here is a "simple" function prototype:

```
WINBASEAPI BOOL WINAPI GetCommState(HANDLE hFile, LPDCB lpDCB);
```

There is little reason here to do much verification, right? Wrong. Simple programs are often deceiving. Look into this function a little more carefully and you can begin to appreciate the complexity of many programs that developers must deal with every day.

This function prototype appears to only have two arguments. At first this seems like a well-defined function with simple types that just checks the communication state and returns a Boolean value. However, on close examination the LPDCB type is not a simple type at all. Look at the previous sidebar. There are *at least* 29 arguments. One may question why we say "at least." The answer is that we cannot be sure how many arguments a function actually uses. Simply counting the number of fields in LPDCB and adding one for the HANDLE type only sets the minimum. The LPDCB type may contain pointer fields as well thereby increasing the count even further. Also, HANDLE is often used as a pointer, so we would need to take its fields into account as well. This function has the potential to be highly destructive. All of the fields of LPDCB are accessible to this function whether or not they are actually required by the function. The function is free to change any and all fields. The moral: *you can't judge a function by its prototype or its listed parameters*, a careful examination is always required.

Conclusion

There is no intention in this article to imply that what the software engineering literature preaches is either incorrect or unimportant. We are convinced that the intention of these authors is good, they are concerned about software quality and are trying to help. Indeed they have helped: sound management practices are beneficial to any software development project. However, we are concerned over the lack of substantive treatment of the technical aspects of software development. Ultimately, all software development projects concern designing, writing and maintaining a code base. The techniques used to do these things correctly are at the heart of the discipline we call "software engineering."

Attention to the technical details often defines a successful product. Solid technical processes are responsible for quality and it is possible that good technical people can create good products despite poor management. However, the reverse is not necessarily true—it is extremely unlikely that below average developers will create a good product even under the best management and methodology. Developers must master their programming language (and its compiler, debugger and IDE), their operating environment (the OS, run-time libraries and APIs) and be willing to become problem domain experts to have even a chance at being effective. Then and only then can they use software engineering effectively.

We believe it is time to call a truce. Developers do not deserve to be beaten up each time a new book or article on software engineering gets published. Software developers face enormous technical challenges whose solutions are poorly addressed in the software engineering literature, yet they manage to create some of the most complex systems known to man. Software engineering advocates face the equally enormous challenge of helping them to do this better. It's about time we all started working together.

Acknowledgements

The authors thank N. Nilakantan of Texas Instruments for his review and helpful comments. Also, the anonymous reviewers gave some very insightful comments that helped improve this paper.

Author Bios

James A. Whittaker is an associate professor of computer science at the Florida Institute of Technology, Melbourne. He founded the Center for Software Engineering Research with grants from a number of industry and government funding agencies. The Center performs contract software development and testing and research focusing on the rea-

sons that software fails and what can be done to make software safer and more reliable. The idea is that by breaking software, we learn about what it takes to fix it.

Steve Atkin is a Software Engineer at IBM in Austin, Texas. Currently he is a member of the IBM Globalization Center of Competency. He was the development lead for Universal Language Support and the Graphical Locale Builder for OS/2. His research interests include: character coding systems, bidirectional text processing and software globalization.

References

1. F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1982.
2. R. Fairley, *Software Engineering Concepts*, McGraw Hill, 1985.
3. R. Glass, *Software Runaways*, Addison-Wesley, 1999.
4. S. McConnell, *Code Complete*, Microsoft Press, 1995.
5. S. Pfleeger, *Software Engineering Theory and Practice*, Prentice Hall, 1998.
6. R. Pressman, *Software Engineering A Practioner's Approach*, McGraw Hill, 1997.