Software Engineering Is Not Enough

James A. Whittaker, *Florida Institute of Technology*
Steven Atkin, *IBM*

Much software engineering literature begins with the admonition that practitioners aren't doing enough—that the state of the practice is creating bad software. We do not dispute this fact. However, we believe that what software engineering literature offers as solutions are 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 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 be only partially effective.

How can we create better software? The challenge is to broaden the software engineering discourse from just management techniques to include the real technology that software development requires.

In this article, we describe a basic set of technologies that real software developers apply before, during, and after designing real software, often against unrealistic schedule and budgetary constraints. We hope this discussion encourages dialog and debate between the people who write software and the people who write about software.

Looking for answers in the software engineering literature

Imagine that you know nothing about software development. So, to learn about it, you pick up a book with "Software Engineering," or something similar, in the title. Certainly, you might expect that software engineering texts would be about engineering software. Can you imagine drawing the conclusion that writing code is simple—that code is just a translation of a design into a language that the computer can understand? Well, this conclusion might not seem so far-fetched when it has support from an authority:

> *The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.*[1]

Really? How many times does the design of a nontrivial system translate into a programming language without some trouble? The reason we call them *designs* in the first place is that they are *not* programs. The nature of designs is that they abstract many details that must eventually be coded. Many notorious software bugs are the result of

0740-7459/02/$17.00 © 2002 IEEE

overlooked or ill-understood details.[2] If details really were unimportant, designs would be compilable and little programming by humans would be necessary.

However, automatic code generation from designs is possible only in limited application domains (for example, databases); for most problem domains, human programmers are the norm. Programming remains monstrously complicated for the vast majority of applications. It is so complicated that developers can work for years in a single language and still not learn all its nuances. Often, developers don't have time 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, that a good design does not ensure a good program.

The problem with current software engineering texts that focus only on design is that they encourage software developers to gravitate to the latest design fads, assuring us that good code will follow. This drives developers further away from their work's technical foundations. We lament the preference for the latest fashionable design technologies and the seeming aversion to the fundamental technical skills such as systems programming and debugging that form the core of software development. How can you expect practitioners to take these fundamentals seriously when they fail to get more than cursory treatment in the literature?

Can you also imagine concluding 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.***[3]

In our experience, the only programs that are simple and clear are the ones you write yourself. When you have written a program in its entirety, you have forced yourself to understand every aspect of the problem and the solution. At that point, 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 operating system, or 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 having an allergic reaction to complexity, true software engineers react with suspicion. Perhaps we can apply simplification or refactoring techniques, but then, some software is simply complicated because the problem it automates is complicated. We might be frustrated, but we must also be realistic. (For more on simplicity—or the lack of it—in software, see the sidebar "Simple Code Is Not So Simple.")

You might 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.***[4]

Imagine concluding that documentation can be excessive or even unnecessary! We repeat: real programs are complex by nature. (For an example of complexity, see the sidebar "Programming Is Easy, Isn't It?") 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 successfully change the code. Imagine having to modify

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 have only 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 Figure A in the sidebar, "The Making of a Maintenance Nightmare" (see p. 114). That code involves at least 29 arguments. You might 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 might also contain pointer fields, thereby increasing the count even further. In addition, `HANDLE` is often used as a pointer, so we would need to consider its fields as well.

This function can be highly destructive. All the fields of `LPDCB` are accessible to it whether or not it requires them. The function is free to change any fields. The moral:

> *You can't judge a function by its prototype or its listed parameters; a careful examination is always required.*

Programming Is Easy, Isn't It?

The Windows 2000 kernel contains over 1,200 calls.

The Win32 application programming interface contains over 20,000 calls. The printed reference manual for just the calls and their parameters is over four inches thick.

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

Applications routinely use dozens of other libraries of similar complexity. Each of these libraries likely has thousands of bugs.

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

We wish you well.

an encryption-key management program without 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 descriptions 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 decide that you simply read the wrong section of the software engineering book, so you try to find the sections that cover coding. A glance at the table of contents, however, shows few other places to look. For example, *Software Engineering: A Practitioner's Approach,* McGraw-Hill's best-selling software engineering text, does not have a single program listing.[1] Neither does it have a design that is translated into a program. Instead, the book is replete with project management, cost estimation, and design concepts. *Software Engineering: Theory and Practice*, Prentice Hall's best-seller, does dedicate 22 pages to coding.[5] However, this is only slightly more than four percent of the book's 543 pages.

You could conclude that the act of coding is a very small part of software engineering. Search all you want; you will not find the answers to your coding problems in the pages of most software engineering books.

Software development gets so little attention in the software engineering literature because it is the hardest and least understood part of the software engineering life cycle. Coding is immensely difficult without a good design but still very difficult with one. Maintaining code is next to impossible without good documentation and formidable with it. So, we now look at the *technical* things designers can do to ease development and maintenance.

Considerations before design

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

Before design, developers must pursue two activities: 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 difficult. Imagine writing a matrix-algebra library without expertise in mathematics, coding an address resolution protocol 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 and requires significant study of software environments, low-level protocols, and domain conventions. Software engineering methodologies do not and cannot teach this art. Learning it takes hard work, much 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 employ to build the software product. The minimum set of tools is one or more editors, compilers, linkers, and debuggers (symbolic and kernel)—the fewer, the better. To that set, add make-utilities, runtime libraries, development environments, version-control managers, and, of course, the operating system. So, developers must master many complex tools before they can even use a software engineering methodology.

Software engineering doesn't help with this domain, either. Developers must be

masters of their programming language and their OS; 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 OS features that participating developers do not understand well. 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.

Selecting and properly using tools is another issue that gets light treatment in the software engineering literature. Everyone should use the same editor, 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 is compiled with a different compiler using any settings and flags the developer desires, it might not integrate smoothly. Costly rework and retesting then follows. Even worse is the maintenance problem where one-line changes can take hours to recompile because the original settings for the previous compile are lost or forgotten. The tools that once were 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. Unfortunately, because these tools are software, they can 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 costly.

Simply put, predesign activity defies methodology. However, the technical work performed in understanding the problem and solution domains can be the difference between project success and failure.

Considerations during design and coding

New problems surface once design and coding begins. The first hard lesson that developers learn is to not trust their environment. 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 OS 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 OSs never run out of resources will write code that works only when everything goes as planned. This results in 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 the original developers established.

Selecting and properly using tools is another issue that gets light treatment in the software engineering literature.

Validation

Good developers understand that they cannot trust user inputs. 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 passes 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 the appropriate type and that the value is within an acceptable range. Anytime we fail to perform such validation, we risk 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; they write their programs to balance speed versus risk.

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. Validating a and b is necessary to avoid overflow; however, we must also constrain their combination. Suppose a user enters the values 32,000 and 1,000 for a and b, respectively. If we sum the two and try to store the result in another short signed integer, the result overflows because 33,000 is larger than the maximum signed two-byte integer value 32,767—valid data, invalid result.

Plenty of Opportunities to Fail

Often we think of program inputs only in terms of our interaction with an application through its GUI. However, system inputs are much more pervasive and provide a much larger input validation problem.[1]

For example, Microsoft PowerPoint, a large, complex application for making presentations and slide shows, makes nearly 800 calls to 30 different functions of the Windows kernel upon invocation. This means a single input from a human user (invoking the application) causes a flurry of undercover communication 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. For example, when PowerPoint opens a file, 13 kernel functions are called nearly 600 times; when it changes a font, two kernel functions are called a total of 10 times.

These calls are only to the OS kernel. PowerPoint also uses many other external resources (dynamic-linked libraries) in the same manner as the kernel, including mso9.dll, gdi32.dll, user32.dll, advapi32.dll, comctl32.dll, and ole32.dll. Clearly, the amount of communication between an application and the operating environment dwarfs GUI input.

Unless all these calls are validated, 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.

Reference

1. J.A. Whittaker, "Software's Invisible Users," *IEEE Software*, vol. 18, no. 3, May/June 2001, pp. 84–88.

Handling failure

Developers quickly learn that every program has two parts: the code that performs the desired function and the code that handles failure. We do well at coding the main functional code; handling failure is the thing we do poorly.

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

This isn't enough, as we said; software can also fail internally. Inside a program are essentially two things: data and computation. Both can fail and therefore require constraints, just as inputs and outputs do.

Constraints on input and output are programmed in a number of ways. For graphical user-interfaces, interface controls carry much of the workload by filtering out incorrectly typed data and data that is out of the acceptable range. However, if you are coding a solution with a programmatic interface, all the error checking is your responsibility. You must painstakingly validate each parameter of each call, often meaning a lot of If statements and calls to validation routines. Check the sidebar "Plenty of Opportunities to Fail" for another view of the validation problem.

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, most modern programming languages provide little support for programming constraints beyond If statements and access routines. So, 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, let the failure occur and then trap it. The trapped failure can then be handled by a special routine that the developer provides.

When developers use exceptions (which most do because they have little choice in the matter), they must determine *how to fail*. Failure recovery is often difficult. The program must know the application's state at the time of the failure so that it can properly react. If files are open, the program needs to be aware of this so that it doesn't try to reopen a file. If an operation didn't complete when the failure occurred, the program needs to figure out how much of the operation succeeded and how much is left to do. The exception handler must take stock of the application and recover appropriately. Otherwise, it might fail too: it might write to a file that doesn't exist, use data that didn't get initialized, and so forth. If the exception handler is sloppy, what will handle the exception when the exception handler crashes?

In a nutshell

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 not, unfortunately, part of mainstream software engineering.

Considerations after design and coding

At this point, the software is built, debugged, and executed. Modifying an existing system is different from "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. This situation is where developers are handed problem reports and asked to diagnose possible bugs and fix them.

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

The most important reason occurs in *failure reproduction*—the "repro problem." Failure reproduction is not the no-brainer it seems. Often in this situation, a tester (or user) finds a problem and reports it, but when the developer gets the report, he or she can't reproduce the problem in his or her environment. Why? Well, many things happen 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 enters, dozens of system calls and calls to reused components occur. Normally, users report only their own input; that is, "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 irreproducible 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.

However, system tools don't help much when adding new functionality to code. Adding new behavior to existing code is difficult. When experienced developers mentor novices, they share two pieces of advice:

- Don't trust comments.
- Be wary of side effects when you modify code.

The first piece of advice might seem a contradiction; we debunked the idea earlier that all code was simple enough to do without comments. Comments are indeed important, and some complex programs are not maintainable without them. However, the advice still stands: you can't explicitly trust comments. Developers should use them to understand the code, but not as a surrogate for the code. Frequently, 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 comments 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.

Regarding side effects, if a change requires modification of data, we must ensure that all programs that rely on that data can still function properly after it changes. The worst maintenance sin is to break something that used to work. (For another example of a negative side effect, see the sidebar "The Making of a Maintenance Nightmare.")

Checking that a fix was successful and that no other dependent functionality was broken is a formidable technical task—one that modern software engineering addresses incompletely.

Constant considerations

If there were one thing that all developers would learn and never forget, we would want it to be this: eventually someone else will have to modify your code. Learning this would motivate them to

- Minimize the need for other programmers to modify their code
- 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. The latter motivation would ensure that the code is readable, that data is named appropriately, and that the code has informative, up-to-date comments.[6] If only developers never forgot about maintenance, we would all be better off.

We are not implying that what software engineering literature preaches is either incorrect or

Regarding side effects, if a change requires modification of data, we must ensure that all programs that rely on that data can still function properly after it changes.

The Making of a Maintenance Nightmare

In the frenzy of fixing bugs, maintenance developers often forget the design principles that they routinely apply when they create code from scratch. The code in Figure A, published on a popular Web repository, is one such example, we believe.

This code 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 to create a new function or fix a bug. 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.

```
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;
```

Figure A. Published source code for a DCB data type.

unimportant. We are convinced that these authors' intentions are 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 for doing 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 good technical people can create good products despite poor management. However, the reverse is not necessarily true—below-average developers are extremely unlikely to create a good product even under the best management and methodology. As we mentioned before, developers must master their programming language (and its compiler, debugger, and integrated develop-

ment environment), master their operating environment (the OS, runtime 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.

It is time to call a truce. Software developers do not deserve to be beaten up each time a new book or article on software engineering gets published. They 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.

Acknowledgments

We thank Nikhil Nilakantan of Texas Instruments for his review and helpful comments. In addition, the anonymous reviewers helped improve this article by giving insightful comments.

About the Authors

James A. Whittaker is a professor of computer science at the Florida Institute of Technology, Melbourne, where he leads a team of computer security and software-testing researchers. His current interests are in secure operating systems, digital rights management, reverse engineering, and anti-cyber warfare. He received his PhD in computer science from the University of Tennessee. Contact him at the Florida Inst. of Technology, 150 W. University Blvd., Melbourne, FL 32901-6975; jw@cs.fit.edu.

Steven Atkin is a software engineer at IBM in Austin, Texas. He is also 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. He received his PhD in computer science from the Florida Institute of Technology. Contact him at the Florida Inst. of Technology, 150 W. University Blvd., Melbourne, FL 32901-6975; atkin@us.ibm.com.

References

1. R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1997, p. 346.
2. R. Glass, *Software Runaways*, Prentice Hall, Upper Saddle River, N.J., 1998.
3. R. Fairley, *Software Engineering Concepts*, McGraw-Hill, New York, 1985, p. 192.
4. F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Boston, 1982, p. 167.
5. S. Pfleeger, *Software Engineering: Theory and Practice*, Prentice Hall, Upper Saddle River, N.J., 1998.
6. S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, Redmond, Wash., 1995.