

**Building
Blocks:
The First
Fifty Posts**

Paul W. Homer

Copyright © 2007 Paul W. Homer

All rights reserved. No part of this publication may be reproduced stored in a retrieval system, or transmitted in any form or by any means electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher and author.

1st Edition

July 28th, 2007

Contents

1	INTRODUCTION	6
2	QUALITY	7
3	MORE QUALITY RAMBLINGS	8
4	ARTIFICIAL COMPLEXITY.....	9
5	TURBULENCE	10
6	COMMUNICATIONS.....	11
7	SYSTEM DESIGN	12
8	GOOD PROGRAMMERS	13
9	DISORGANIZATION	15
10	CANNED GOODS.....	16
11	INDUSTRIAL STRENGTH.....	17
12	POTENTIAL.....	18
13	UNDERSTANDING DATA.....	19
14	SOURCE CODE.....	20
15	NOUNS AND VERBS	22
16	A QUIET REVOLUTION	24
17	INCONSISTENCIES	26
18	PHILOSOPHY	28
19	DEGREES OF FREEDOM	30
20	ABSTRACTIONS.....	31

21	FLAWED INTELLIGENCE	33
22	USER EXPOSURE	34
23	YOUR AUDIENCE.....	36
24	DEVELOPMENT PLANNING	38
25	BUILD INVERSIONS	40
26	TECHNICAL RENAISSANCE.....	42
27	DATA DIRECTED DESIGN	44
28	ITERATIVE DEVELOPMENT	46
29	EXTENDING THE FOUNDATION.....	48
30	PERSISTENT STORAGE ARCHITECTURE.....	50
31	PARTIAL ENCAPSULATION	53
32	HOUSE OF CARDS	54
33	ATTRIBUTES OF CODE.....	56
34	THE MYTH OF RELATIONAL DATABASES	58
35	HORIZONTAL VS. VERTICAL TOOLS	60
36	LEARNING FROM HISTORY	61
37	BEST DEVELOPMENT PRACTICES	62
38	THE TROUBLE WITH DECLARATIVE.....	66
39	INDUSTRIAL STRENGTH LANGUAGE	68
40	RUST AND BLOAT	71
41	QUICK AND CLEVER.....	72
42	FUGLY AND THE ART OF PROCESS	73

43	EVALUATING CODE	75
44	SOFTWARE DEVELOPMENT	77
45	EFFECTIVE PROGRAMMING.....	79
46	SOME THOUGHTS ON TESTING.....	88
47	INDEPENDENT TESTING	90
48	WHEN IDEAS BECOME PRODUCTS.....	91
49	CONTAINING COMPLEXITY GROWTH	94
50	WEAKER THAN THE WHOLE	97
51	THE FIVE PILLARS OF SOFTWARE	99
52	COMPLEXITY UNDERTOW.....	104
53	WHY IT ALL MATTERS.....	106
54	PROCESS BLUES.....	109
55	FINALLY.....	112

1 Introduction

A few years back I looked up from my keyboard and realized that Computer Science was – putting it nicely – not going too well. Some things were better, but overall progress has been painfully slow and minimal. It is very disappointing.

I've built a number of complex software systems, and I've learned a lot along the way. I felt that it was important for me to, at least try to put that knowledge and understanding back into the community. The big problem: I'm not a particularly gifted writer. In fact, I barely have a concept of grammar at all, and if it weren't for automated spell checking, most of what I've written over the years would be completely illegible. Still, one has to try.

I wrote the Programmer's Paradox immediately after leaving a very tough job that I had been struggling with for years. I like most of the ideas in the book, but the writing leaves a lot to be desired. Later, someone suggested that I get more experience by writing a blog. I named it Building Blocks and proceeded to fill it with long unreadable essays. I was going to keep it light, but my own nature kept getting the better of me. Now, I'm going to try again with a new blog for a lighter feel, but there was enough written in Building Blocks that I wanted to save it. Even if I'm the only person to ever reread any of the entries, I think it is worth saving, if only to see many years from now where I came from. Hopefully I improve!

These entries come directly from the web site, in the order they were written. I decided against including the posting dates, figuring that it was the writing that I most wanted to preserve, not the way or frequency that the work was done. It is worth noting that the very first post -- which wasn't include cause all it said was "Comments and/or questions about my opinions or writings? Please be nice..." -- was written on the 3rd of November 2005. The first real post, "Quality" showed up several months later on the 21st of February, 2006. The frequency over the years was inconsistent, and sometimes depended on what I needed to say. The last post was July 13th, 2007, roughly about a year and a half later. The dates actually correspond to my working on a couple of contracts. I didn't edit the writing other than by fixing obvious spelling mistakes. The grammar is, as it always has been, if that is a reasonable only way to express it.

For anyone other than myself -- in reading this -- if its any consultation, the writing probably gets a bit better as you read on... (although I wouldn't hold my breath if I were you).

2 Quality

I've been thinking a lot about the quality of code lately. To me, it is an independent attribute from how much testing the code has received. By that, I mean that you can have heavily tested, but very poor quality code, or poorly tested good quality code. The difference doesn't come from the runtime behavior of the code; it comes from the ability to enhance the code in the long term.

3 More Quality Ramblings

The issue is not necessarily one about whether or not the underlying code is or should be elegant. More often than not, it is really a business issue about the cost to bring a line of code to market and the cost to keep it there.

Programming is about building things, which is more satisfying if people are actually using what you have built. A key problem in any type of building is to make effective use of the resources. There is never enough time to get it created the way you want it. The more effective the development, the more time there is to spend on things like making sure the functionality is what the user's need to complete the tasks.

Ultimately, a software development project comes down to finding the most reasonable way to leverage the available resources. Simplicity and elegance within the code base are not exercises to complete for fun, but are really ways in which to get the most from the limited resources by reducing the amount of time spent testing, bug chasing and refactoring the code. Better quality code requires less testing. For many of the huge development projects, testing is the single most significant usage of resources, so cutting back on it is a significant cost reduction.

4 Artificial Complexity

It could just be age, or possibly a by-product of having lived through the repeating five-year cycle of computer software technology a few times, but I have that feeling that most of the modern incarnation of the technology is overly complex given what it is able to accomplish. Certainly, there have been big improvements in the size and scope of the available data, and the applications themselves have gotten prettier, but the fundamentals remain largely unchanged. In the meantime, the sheer amount of complexity involved in utilizing these new technologies seems to have increased in an exponential type of growth.

Trying to keep up with all of the required knowledge has become a significantly complex problem all on its own. Many of the newer tools are poorly designed compared to some of their earlier predecessors, as each new set of developers is clearly intent on reinventing the wheel. The growth of esoteric 'little languages' to solve mini-problems within the systems has become endemic and clearly a cause of a lot of artificial complexity. It is as if developers have been expending extra effort to try to avoid using any standard type of set up or interface in their designs. The normal course one would expect for an engineering discipline is that each new generation is more complex, but also more dependable, more usable and less subject to unexplainable problems. Clearly, that is not the case with most of the modern versions of our software. They have become difficult, hard to master and untrustworthy, as what seems to be a by-product of a fervent desire not to learn from the past mistakes and design choices.

5 Turbulence

Complexity can build up in a software project in the same way that turbulence can build up within a stream. As the bed twists and turns and there are obstacles to maneuver around the water becomes more and more turbulent. It builds and feeds on its self until the water has worked its way into achieving the state of being rapids. Individually any of the obstacles would not have caused the water to become so rough, but as a collection, they leave the water no choice as it tries to flow through the streambed.

In many ways, simple bug patches, workarounds and those little bits of inconsistency on their own do not cause significant increases in complexity. However, if enough of these exist, they build on each other to become significantly larger than the whole. Consider that at some time in the very distance past, a programmer ran into a problem with the handling of new lines while storing text files on a hard disk. To get around the problem, a choice was made to differentiate between text files and binary files, thus allowing the text files to be manipulated, as they were stored on the disk and undone when they were read into the computer again. That choice was arbitrary and a poorly thought out work around intended to quickly deal with the problem. Only one operating system needed this hack, but eventually as more and more systems were interconnected, the hack propagated. First in the original OS, and then in the programs to copy from files different OSs, and then into the protocols that allow all of the machines to interconnect with each other. The little fix has been the cause of countless debugging problems and a massive amount of code. The sheer amount of work that emanated from this little fix is staggering and particularly so when one realizes that the distinction between binary and text is unnecessary.

Modern software is so filled with this type of turbulence, that it is a true wonder that new programmers can understand how any of the existing code works. Between the inconsistencies, hacks and the all of the other artificial complexities, software is rapidly approaching some threshold where it will be so complex that not even the best and the brightest will be able to consistency make things work.

6 Communications

It is important for programmers to be able to communicate the knowledge that is building up inside of them. Software development needs more conversations about how to enhance the process of building systems. Too much gets lost as each new generation reinvents the wheels in the belief that they are forging ahead into new territory. As a discipline, software development is constantly taking 2 steps forward and then 1.8 steps back. By now, a significant amount of the thinking and development effort on the technical fundamentals has already accomplished long ago. Our greatest challenges come from making the tools easier to learn and building systems reliability so that they last, not in trying to find algorithms for faster sorting.

While writing is an effective way of sharing the knowledge gained from experience, it is difficult to find a voice that is neither boring nor arrogant. The reader should feel like the conversation is personal in order to maintain their interest, but the work also needs to get straight to the point. With so much to learn, the reader's time is not a luxury to be taken lightly.

7 System Design

While a software program is often viewed as a set of steps taken by the computer to perform a specific task, this type of viewpoint complicates the understanding of the underlying mechanics. The steps for modern software systems are many and conditional based on a huge number of different variables. The more dynamic the software, the more confusing the sequence-of-steps viewpoint becomes. This leads to confusion that causes overly complicated implementations that are impossible to maintain.

Visualizing software as a series of transitions to data based on user-initiated functionality is a more abstract model that actually simplifies the implementations. Data flows into the application, gets stored for long periods, manipulated and then flows out. All applications are essentially the same, just the underlying data changes. The frequency and quality of the data do have a direct impact on the types of technological problems that are solved along side of the data manipulation, but it is likely that the technological problems can be completely separated from the data-related (business logic) ones.

Under this perspective, design is entirely focused on determining what data needs to be available in the system and what underlying data-structure can correctly hold that data. The system needs to import the data structure, manipulate it, store it for a long time, and then export it. There could be some complicated algorithms required for manipulating the data internally, but beyond that, once the data-structures are known, the design for the system is a direct consequence of the data that it will be working with.

8 Good Programmers

Almost anyone can write a computer program. The essence of assembling a series of instructions into a set of files for the computer to iterate through is not particularly complicated. However, in practice, there are many reasons why the actual number of good programmers is a small one. By this, I am referring to programmers that have the knowledge and experience to deliver working systems in a timely basis repeatedly without serious issues. Achieving that goal is sufficiently complex enough that it is not done as nearly as often as it should be for the software industry. Companies get around quality control problems by applying excessive testing, delays and strong support structures to keep the clients placated, but these are band-aides to cover over a shortage of good well-trained experienced programmers.

Many programmers fail at producing good software because they cannot simplify the underlying problems. Writing 50X as much code as necessary in a field where time is always scarce will always lead to definite problems. For some people, over complication seems to be an essential attribute of their thinking and is unlikely to be corrected by experience or training. The excellent developers naturally zero in on the simple solutions because they are driven by an internal sense of elegance. Most developers are somewhere in the middle, so they often have a sense of simplification, but do not necessarily implement their code with that in mind. Good programmers consistently produce simplified solutions to the problems within their development project.

Over complication can be inherent in the design but it can also creep into the development over time. Failing to keep a body of code clean and organized can lead to exponential complexity growth. This can derail programmers that might have ordinarily been able to simply the problem — the disorganization and lack of standards create a vortex for wasting time. These types of problems become vicious circles as the more time that gets spent flailing at the code, which is then taken away from the time available for general maintenance and cleanup. This causes the code to get messier and the problems become harder to diagnose so they eat up more time. The fix, to break this cycle, is in not allowing the code to get disorganized and messy. Programmers require the discipline to continuously cleanup the code in their projects to insure that it is ‘industrial strength’. This is easy to say, but rarely done in practice. This problem is extremely common for most software, particularly if it has been in development for a long time. It is a very solvable problem, but rarely gets the attention or mandate it requires to be fixed. Instead, most development struggles on with lots and lots of source files that are ‘too messy’ to fix. The time to correct this situation would, in the long term be less than the amount of time that is wasted in dealing with issues caused

by the mess. Good programmers find the initiative needed to cleanup the code and keep it that way, because they do not like wasting time.

9 Disorganization

Complexity more often comes from disorganization than from over-complicating the design. This type of problem comes from making continual changes to a body of code without applying reasonable housecleaning in between the iterations. Common problems of this type include code changes layered on top of each other, duplicated code with minor variances, differences in standards that are not consistently applied and half-implemented changes to the underlying architecture. These inconsistencies make understanding and modifying the code increasingly difficult.

Software development is a ‘stitch in time saves nine’ type of discipline where it is crucial to keep up on the consistency and organization of the source code in order to insure that the continued development of the system is possible. No software system exists statically in time. The code can become so messy and disorganized that it becomes too much of an effort to continue to extend it. Finding the time and the discipline to keep the code clean are core skills that you need to learn in order to enhance your programming abilities.

Disorganization problems that can be a minor irritant for small projects can become fatal for large ones. Even if you are comfortable now with the degree of organization in your development environment, learning to keep the code clean and organized can allow you move forward and build larger systems.

For large development projects, I generally try to spend 10% to 20% of the each development cycle on implementing housekeeping duties like deleting unused code, updating code with old standards and refactoring convoluted segments of code to make them more readable. These types of changes should not alter the functionality of the system, so a quick and simple regression test is acceptable afterwards just to validate that the core behaviors remain unchanged. Starting with this type of clean foundation makes extending the code faster, more accurate and less painful.

(Many thanks to [Andy Hunt](#) for helping me clarify my ideas :-)

10 Canned Goods

Spaghetti code is a common problem with many programming languages – it gets its name from the fact that the code is so convoluted that following it is like tracing strands of spaghetti through a pasta dish. The logic weaves and winds around the many different functions and procedures, following no clear or understandable pattern. To completely understand the code is nearly impossible and to make fixes to it is exceptionally dangerous.

A lack of structure imprinted over the code is a primary contributing factor in creating spaghetti code. This common problem was certainly one of the reasons behind creating higher-level abstractions such as Object Oriented programming. The semantics of an Object Oriented language try to force the programmers to adopt a structural approach to developing their code. This reduces the likelihood of the programmers building up essentially random sets of instructions, but it does not eliminate it. Techniques such as design patterns further help to apply consistent structures over the code to insure that it is readily understandable.

It is a little surprising then, to find a variation of the problem comfortably existing within the confines of the Object Oriented realm, even when there is an application of structuring techniques such as design patterns. Despite all intentions to the contrary, objects can be turned into what is essentially a variation of the original problem – a playful name for this might be something like ‘Spaghetti-Os’. This Object Oriented version of the traditional problem is easily created by failing to keep the code focused, clear and understandable.

There are a huge number of techniques available for preventing or fixing this type of problem and all of them revolve around the same key principle – your code should be easily understandable by other programmers. Source code moves towards disorganization naturally all on its own. Your alterations to the source should insure that the results have become more readable than the previous version, not worse. Each iteration should continue to further clarify the code, the conventions, the style and/or the architecture.

Whenever I see a problem with structure or conventions, I try to fix it immediately. If I’ve open a source file and it does not match the conventions, I always try to bring it up-to-date first before starting with any new set of changes. In my experience, the time spent updating the code is actually less than the time wasted debugging some problem in the middle of sloppy code. For all construction projects, the results are always better if they are built on a solid foundation, the same principle holds true for software development.

11 Industrial Strength

Source code comes in different grades – ranging from being very fragile to being Industrial Strength. To get the highest grade, the code must at least adhere to a defined set of standards, not contain weak constructs, be easily readable and have a clean obvious structure. An experienced programmer with no prior exposure to the code should be able to get a reasonable understanding of what the code will do when executed based solely on its structure and its comments.

Following a coding standard is significant because it reduces the complexity of the system. If each block of code is laid out differently, the code looks messy and it makes it very hard to read. This artificial complexity can easily be avoided by adhering to a set of standards and conventions. Weak constructs come from choosing a method of achieving some goal that works, but can easily fail at some point in the future. An example is relying on lexical comparisons for determining order in version number strings. The code works correctly if the numbers remain single digit, but parsing and numerical comparisons are necessary if the code is to support any larger version numbers.

You should always strive to make the code readable. The lifespan of code is long enough that this type of effort always pays for itself. The functions and variables of the code should match the nouns and verbs of the description of the code. It should read as if it were its own description. Overlaying a simple and consistent structure is important in controlling complexity. Well-structured code minimizes the artificial complexities. Code has a normalized form and refactoring it can push and pull the code into this form.

Software development can be extremely difficult. Working with Industrial Strength code eliminates a huge number of simple problems. The code is easier to search, it is faster to understand and it allows for more automated operations. Bring new team members up-to-speed is significantly faster and easier, and the code requires less testing to assure that it is ready for releasing. Bugs are easier to find and faster to patch. There are fewer problems with extending the functionality and it is less likely that you will misinterpret the behavior. It makes your job simpler and allows you to spend more time focusing on building new functionality, instead of trying to maintain it.

12 Potential

It does not take long when playing around with a computer to realize the potential for these machines to have a positive impact on our lives. It also does not take long before the frustration starts to build. For novices, many of these easy-to-use systems are anything but, while for the more experienced there is a constant disappointment caused by the continuous failures that one so frequently encounters with modern software.

All software does is collect and manipulate data, but as we collect more and more of it, we can tie this data back to the real world. Finding ways to simplify and leverage this collection and processing of data is the key to adding value to a software tool. We can apply computers to automate tasks that were faulty or too cumbersome to complete by hand. As more people connect to the machines, we produce larger and larger value from the sharing and interchange of information. We live in an age where there is an abundance of data, and we have the ability to build tools to help us make sense of all of it.

It is frustrating to work with software that falls short of this potential. Perhaps more so, when one realizes that producing better software is not necessarily a factor of more time and effort. Many programs would be significantly improved if the developers spent less time pounding out new functionality and more effort in cleaning up their existing code. Cleaning up and simplifying the foundation would significantly reduce the effort required to build on it, resulting in huge cost savings for the next implementation cycle. We do understand how to build simple and elegant works, but these are rarely produced by our methodologies these days. Our processes for building software focus on heavily on documentation or features, not on the key objectives – simplifying the problems and creating better source code.

It is time for us to acquire a new understanding of how to build systems. There are better methods of working that are less time intensive and will produce consistent results. The focus in the past was either on obsessively adding new features to the products or in creating massive documents that cover every possible detail – the focus for the future should be on simplifying the exist foundations to allow for a more stable building environment. We need to do less work, but get more results. We do not need more new features on a bad foundation; we need to simplify the foundation first. Only when we have come to this understanding, will our processes change enough for us to be able to leverage the existing technological foundations to their full potential. Until then we will remain lost in a fog of overcomplicated technical implementations, features and bugs.

13 Understanding Data

The key to a simple software design is to focus entirely on the data within the system. All software is composed of a set of data structures that undergo a series of transformations. In some cases, these translations have a degree of complexity that requires a complex algorithm for implementation. In most cases however, the translations are very simple operations and often nothing more complex than just a series of copies. Breaking down the mechanics of the system in this manner provides a clean and simple perspective on the code and is should be built.

Software developers frequently get lost in a tangle of features and functions, many of which become increasingly more complex than necessary. The same data is copied repeatedly throughout the system because the developer's focus is on other issues. Inconsistencies in the copies and scoping problems are direct results of this type of design. The CPU burns through redundant or overly complex steps that are not necessary. The sheer size of all of the duplicated data in memory bloats the software and causes performance problems. All of these problems are unnecessary and often a direct consequence of over-complicating the design of the code.

Software as a tool that manipulates data structures with transformations is a very simple concept. It allows you to focus on the key aspect of the code which is generally is providing access or manipulation of data. Your design deals with importing, exporting and storing data for the long term. The system provides the users with the ability to display data in various ways and to apply transformations to their data for short or long-term persistence. All other aspects of the design fall easily in place either as a new data structure within the system or as a transformation on an existing data structure. This viewpoint applies equally to all software and forms a fundamental base on which to design anything from an operating system to a graphical end-user application. It is an exceptionally powerful way of being able to simplify a design and break down complex systems into smaller tangible pieces.

14 Source Code

Software is the end product of a very long and often complex process. It is built from a collection of information stored in files that is fondly known as the ‘source code’. This term has its roots in the history of computer language development and essentially identifies any information that is assembled and manipulated by the programmer, and so differentiating it from what is maintained, generated or altered by some automated computer interface.

From the source code, most systems have complex build procedures that bring together the many pieces in the system and link them together. This process normally builds all of the code into a final ready-to-use software package suitable for running and testing. Commercial products often have another step whereby the software is ‘packaged’ in some form that is ready for shipping directly to the customers.

The source code includes far more than just the primary programming files for the system. It also includes any configuration, initial data, language translations and help files. Documentation is part of the source, although rarely recognized as such and so it is often split off from the base code so that it becomes the source of many problems. Source code is any and everything that is needed for the building and packaging steps to complete the product. That even includes the instructions on how to build and package the project itself.

For a developer, understanding what you are building is important. Your source code is the most critically important part of the project because it is what you manipulate on a daily basis. The build and packaging steps may produce the final outputs, but the work, testing and analysis are all directed at the code. There are many misconceptions about source code, such as more is better, or that it is wrong to delete older code. Some believe that warnings during the builds are unimportant or that formatting and style are meaningless if the code actually runs correctly. These ideas fail to account for the fact that all source code is actively being changed to continuously keep up with the surrounding environment. If the code ever becomes truly static, it is nearly impossible to get it into active development again and it must be thrown away, which can be expensive and wasteful. Another important attribute of source code is that it is tied heavily to the structure of the development teams. The architectural lines of the system match the social and working arrangements of the developers. Failure to account for this arrangement leads to sever complications within the development process.

When assessing the technical merits of a development project, the state of the

source code is the primary focus. If the source is inconsistent or overly complex, that is generally a by-product of problems with the developers or their processes for developing the system. Source code problems are good leading indicators of instabilities within the software itself. The problems propagate throughout the code, build and packaging stages and they generally affect the software grade, quality and required support. The modern accessibility of the source code for a huge number of software products shows quite strongly that there is a clear correlation between the state of the source code and the usability of the final product. Our practices as software developers should be focused on insuring that the source code is organized and presentable – that will allow us to spend more time to build better cleaner systems.

(Special thanks to [Stan Yack](#) for suggesting this topic :-)

15 Nouns and Verbs

There are nearly an infinite number of ways to deconstruct software functionality into byte-sized pieces. In the process of building a system, the software developers need to acquire a specific enough understanding of the domain's problems to be able to communicate with the end-users about their requirements. At a higher level, the developers communicate these issues directly with the end-users. This communication centers on the functionality that the users need, but it also deals heavily with the data that the users will access. This higher-level discussion is then decomposed into lower level concepts, algorithms and is eventually added to the system.

In understanding and discussing the problem domain, you need to be able to put terminology to the various data and functions within the system. To talk directly with the end-users you need to understand all of their processes and terms. To be able to explain the code to a fellow developer you will need terminology for all of the component pieces right down to the lowest level within the system. For example, if the system were processing a set of financial instruments, specifically bonds and performing a set of calculations on them to help people trading them to value them correctly, the data in the system would be something like bond, value (yield), sequence, etc. The functionality for this part of the system is to retrieve the bonds from storage, add them to a sequence, calculate their valuations, display the calculations, and then save the results. You may notice that the data corresponds in an almost one-to-one fashion with the nouns that are used in the description of the system. The functionality as applied to the data corresponds to the verbs used to describe the system. Quite simply the nouns and verbs used in describing the problem domain map directly onto the data and functionality of the system. If you can describe the system to someone, the words that you are using form the specification of the system.

There are different levels within the system itself. At the highest level, the code deals directly with the business logic. This clearly maps back to the language used in the description with the end-users. At lower regions in the code, the problems become more technical in nature and thus map back to descriptions of the underlying software or onto some Computer Science theory. In the previous example with the bonds, within the lower parts of the sequencing portion, the language will generally switch to use terms such as linked lists or arrays because they are the underlying physical representation of the data. The change in language reflects the programmers intended use of the data. This helps to document the system as it was built.

Being able to describe a system verbally is the first key part of producing a reasonable architecture for the system. If you have having trouble putting terms to the pieces, it generally means that there is more investigation to accomplish before you can complete the design. This becomes a simple verification step that will highlite any unknowns in the design.

16 A Quiet Revolution

Fundamentally, building software is not a complex task although it can require a significant amount of work to get it done properly. Gathering the requirements, putting the design together, building and testing the code and then deploying it are all well-known, well-understood tasks that have been successfully completed by many people over many years. The vast majority of the work involved in software development relies on existing knowledge, such as algorithms, theory, data, etc. that has been available for decades and already exists in multiple live systems. Despite the fact that none of the development work is truly original, few development projects incorporate this past knowledge to any significant degree. Most resources are expended in either maintaining the process of development itself, or in the unnecessary reinvention of some existing technology piece.

It can be frustrating to see the results of such a massive amount of effort and to realize that the resources should have been more effective. Over time, software should be improving in quality and usability, but this is clearly not the case. Instabilities in our systems are greater than ever before, and while the amount of data and access has been widened, the overall quality and trustworthiness of the tools we produce are nowhere near where they should be at this point. Unlike most other knowledge-based professions, the longer software is worked on, the more disorganized and dysfunctional it gets.

Existing processes for development may vary, but they almost never focus on the output of development itself, and as such, more often than not they only insure that even if the process is followed, the quality of the results will still be highly variable. There are many new ideas about how to control development with less process, but most of these are geared towards smaller projects. By focusing on less organization, and on building the code on the fly, these low-process ideas work on thin-layer projects, but fail to scale because they ignore the need and importance of structure. Too little or too much of anything is trouble, and this is certainly true within development processes.

The software developers themselves need to come together to create new processes based on our experiences. We need a revolution in the way we visualize and build software, not just another methodology that avoids the real problems. We need to identify the ways to make things simpler, and to choose the paths that most easily allow us to avoid complexity. We need to change the way we organize our development projects and teams to take advantage of our understanding. We need structure and we need to insure that the process itself forces quality workmanship, not just blind obedience. We need to change our

mind-set from one where we ignore the existing problems in the name of issues like backwards compatibility to one where we are always looking to improve upon the existing foundations. Clearly, software should get better over time, not worse. Our problems come primarily from our work habits, culture, processes and organization, not from any technical challenges or theoretical problems.

17 Inconsistencies

Even the smallest of inconsistencies in a graphical user interface can combine to give the tool an awkward or amateurish feel. The users may not have explicitly noticed the problems, but the feeling of the tool is heavily affected by this type of issue.

As a user negotiates the different functionality, they are very sensitive to even the smallest differences because it forces them to concentrate harder while using the software. Working with a product that requires a lot of concentration just to get the basic functionality leaves you feeling prematurely tied and drained. Dialogs that are not obvious or functionality that is hard to find in the menus adds to the effort required to get the program to even accomplish the simplest of tasks, but the problems can be as simple as inconsistent writing styles in the dialogs or menu choices.

Over time and exposure, the users are less affected by the inconsistencies and in some cases they may even grow attached to them, but the damage has already been done. Their opinion of the software is more likely formed from the intangibles and their own emotional response to the program, than it is from the base functionality or the dependability of the code.

If you consider your own favorite and most hated software programs, you'll likely find there is a pattern where programs that were easy to use are more beloved. Those programs where nothing makes sense, the functionality is all over the map and you end up having to search every menu item just to find the basic functionality are most likely on your hated list. If you've had some fear of pressing the wrong button or you mistrust the program in any way to do the right thing, than it most certainly ended up on your hated list. For some programs, even if the initial response was negative, but the usage was enough that over time you forgave the program its faults and got used to it. But likely that was because you were forced to continue to use the program, and had you had any other reasonable choice you would have abandon the software before reaching that point.

When you are building an interface, every inconsistency, no matter how small counts as a negative towards the tool. If there are just enough negatives, the user reaction toward the program will be negative — a truth that exists whether or not your users have given you that explicitly in their feedback. When developing, because of repetition, the inconsistencies will not be noticed, or even if they are there is a tendency to understate their importance with phrases like 'that is not important' or 'the users will never notice that'. Rest assured, that

even if the users don't notice it directly it will have an impact on their overall opinion of the software. The ultimate goal for software developers is to build usable tools that will meet the needs of the users. Tools that are frustrating or awkward to use do not fulfill this requirement.

18 Philosophy

Early Unix systems were an absolute joy to work with because they had this all-encompassing simple philosophy in their construction. It took a while, but once you groked the design, getting around the system and making it work for you was extraordinarily easy. It is that uniformity and consistency that when fit into an overall design plan makes something both elegant and a pleasure to use. It allows the user to only browse a fraction of the manual but in doing so, acquire a real sense of how the whole system actually works.

Building things that make intrinsic sense seems to be a goal that has gotten lost as each new wave of developers extends out the existing types of systems in unique ways. The absence of an overall structure makes it inherently more difficult to understand the system because each part has to be understood completely within its own context. Understanding one part of the system sheds no light on any other part. This is generally poor engineering; the pieces should come together and should fit into the picture as a complete whole. Ironically, in many cases making all of pieces more consistent would have actually reduced the amount of work required to build the system.

A single overall philosophy in the design can produce something of absolute minimal complexity. This attribute is important for both the building of the tool and for its usage. When spiraling out of control, complexity is the root cause of many development failures and disappointments. Overly complex tools hinder the users ability to complete their work. If you reduce the complexity in the design, it both reduces your workload and makes for a better product.

Generally, the best philosophies are those based on abstract concepts that were assembled by a very small group of people. Abstraction helps to allow a broad understanding of the problems, and to be able to conceptually manipulate them. The more people involved in a design, the more likely that the design will become a by-product of compromising between many different beliefs. Compromises work fine for some aspects of society, but they clearly increase the complexity, which in engineering can be problematic. Simple is best, and its unlikely that a committee will produce a simple design.

In your development, it is important to look for that consistent uniform overall philosophy and to be able to articulate it in all aspects of the design such as the architecture, the interface and the configuration. A lack of this attribute shows up in the added complexity of the system. In interfaces, you can see the overly complicated portions based on where the users ask the most questions, and where they have the most trouble. In your code, returning to the same section

to fix yet another bug always highlights a problem area. In the architecture, if each new iteration requires a major architectural change to implement, then it is not a good design. A sign that you have gotten it correct is that you only ever need to revisit the structure, specification or implementation to update it or to make obvious new extensions. Another indication that it is simple is that you can easily explain the philosophy to an outsider with a reasonable background.

19 Degrees of Freedom

At times, it can seem like there is an almost over-whelming number of choices to be made when assembling a new design for software. This sense of a large number of degrees of freedom in the design is misleading because a well-built tool fits snugly into its functionality, which in turn, dictates how the tool is built. As you fully understand the tool and its requirements, there are less and less possibilities for implementation. Once the primary purpose of the tool is chosen and the big technological choices are made, building a good tool is less about creatively making up options and more about finding and harmonizing them into the design.

All things in the development limit your degrees of freedom. The time and resources available for the development make a significant impact into the amount and type of functionality that will be built. The choices made in implementing iterative portions of the design will likewise play a core part in shaping the final design. The data that will be handled within the tool and the underlying business logic, that will be encoded within the functionality put a considerable set of constraints onto the design, often made more intense by also having to support some technological based problems such as performance or compatibility.

These requirements significantly limit the final design, and when factored in with any standards and conventions that the code, interfaces, connections, design etc. should follow, there are very few degrees of freedom involved in the design of a software tool and none at all in the implementation. As part of the culture, architects and programmers often choose to ignore this and proceed with designs that are non-standard and often not particularly functional. The idea that the programmer can do almost anything at all is particularly strong in development projects – an idea that is generally followed by a re-write and some kludges to reduce the impact of the inherited problems. There is still a huge amount of creativity and skill involved in building a great software tool, you don't need to extend that by making arbitrary choices that compromise the functionality, look or feel of the program.

20 Abstractions

A seemingly consistent way to get software implemented is by using brute force. This popular technique has the programmers explicitly iterate out all of the possible lines of code to accomplish the required functionality. The strengths of this method include making it fairly easier to write the code and making the time for implementation consistent. Initially bug fixing is faster as well, but over time, this fades quickly. An added bonus is that the programmers don't have to be particularly well trained as the technique does not rely on skill and training as much as it does on just decomposing the problem as quickly as possible and then typing in all of the instructions.

Abstraction is the opposite technique from brute force. It involves conceptualizing something, generalizing it or producing models to build the implementation at a higher level, but much simpler overall. An abstract model is always simpler than the messy details of the real thing. While it may require a greater degree of understanding and a lot more thought, the gains from applying abstract outweigh any extra effort by a huge margin. Once the abstraction is understood, working with the code is much easier and extending it should be quite natural. If the abstraction is general, then the same code can be redeployed for many uses, further leveraging the initial effort. The key plus for abstraction is that the code is smaller, and not just by a little amount, but by a huge amount depending on how generalized the abstraction is. Abstract implementations can be a mere fraction of the size of brute force ones. Less code means less of everything else including maintenance and testing. Abstraction can take a complex brute force implementation that needs a small army to keep it going and put it into the hands of the a tiny team of people.

The fear with abstraction is that the code will become so complex that it will never work correctly, or that it will eat up more time in debugging than the brute force code would in implementation. There is clearly some extra effort required initially to properly set up an abstraction, but the benefits are quickly apparent once it is implemented. By the second or third iteration of development, an abstraction easily pays for itself because it is an order of magnitude less work and another order of magnitude less testing. Certainly, most good abstractions can be based on already well documented foundations, such as data structures, design patterns or other abstract representations of code or functionality. Some thinking is required, but it shouldn't be a grueling exercise. It is possible to over-complicate an abstraction and by doing so remove many of the initial benefits.

You can simplify your code, reduce the overall amount of work and make your

life easier by applying abstraction to the key areas of the system. When correctly implemented, your system will be less complex, more stable and easier to manipulate. If you feel like your are continuously writing the same piece of code over and over again, then you are probably employing way too much brute force to build the system. A smaller, tighter system is a more effective solution and more enjoyable to work on.

21 Flawed Intelligence

At times, very clever ideas have been added to software in an attempt to mimic intelligence. Some developers have gone to great lengths to add these types of specific behaviors into their products. Unfortunately, more often than not, that type of functionality stands in the way of utilizing the tool completely. The key problem is generally caused by the programmers focusing too hard on a specific part of the functionality while missing some key aspect of the overall usage of the tool.

Tools are always most useful when they don't make any assumptions. The perfect tool is the most general one that completely covers any work required. Whenever the programmer feels that they understand the user's problems better than the user does, they come off being high-handed in their development work. This generally means that they will embed somewhat intelligent behavior into the code in order to guide or shelter the users from some perceived problem. The justification for this type of functionality is always that the users are not technically sophisticated and need significant help in order to complete their tasks. Massively underestimating the abilities and experience of the users is a common programmer myth, which often leads to this type of problem.

Beyond that, computers are particularly dumb machines, so that embedding partial isolated bits of intelligence into their behavior is inconsistent and more likely to annoy and confuse people. A good tool does the job its inventor envisioned, but it also finds itself used in places that were completely unforeseen. Less intelligence and fewer restrictions will allow you users to leverage your tools for purposes that are completely unexpected but still entirely valid. If they aren't frustrated by your attempts to think for them, you may find that they appreciate using your creations a lot more.

22 User Exposure

There was definitely a time in the past when the majority of software users had very limited exposure to computer environments. This experience level needed to be taken into account for the design and building of the interfaces, particularly if the target audience was expected to be general and large. Now, however, software has become so ubiquitous in so many areas of our lives that it is getting harder and harder to make a strong case for the existence of what are essentially, under-exposed users. That is, users with little or no basic understanding of the elementary software interface concepts, such as windows, buttons, upgrading, etc.

Certainly, email has become an indispensable part of corporate life for most office-based professions — PCs, old and new, are the least expensive way to bring that medium to the masses. Within a typical office environment, it is increasingly difficult to find desktop space that is not dominated by a computer. With the machines on every desk, access to technologies like the World Wide Web are easier to set up and more prevalent. Even if access to software is tightly controlled and the hardware is very old, the user is still getting exposure to the basics of application design and system functionality.

A significant number of households have their own PCs with Internet connections so it becomes more difficult to even say that even non-office based careers would leave one without exposure to modern technologies like browsers. While not every one is a power user, it is no longer safe or reasonable to assume that the users will not or cannot understand basic interface paradigms. Even if they have somehow managed to dodge the Internet for the last decade, users all over the world have become significantly exposed to similar interface driven technologies such as cell phones, which may be less sophisticated, but certainly are fine example of common software interface paradigms.

Beyond that, interfaces such as bank machines and phone menu systems are frequent interactions for a huge segment of the world's population. We are forced to interact with many different computers systems on an almost daily basis. It would be difficult, particularly in any of the worlds big cities for a person to continuously dodge getting some type of exposure to computers. Even things like Internet Caf e's have been available in every major city on the planet for nearly a decade. So for instance, staying in touch by email when travelling the world is not a problem – finding an available wireless connection still might be.

When you are considering your users, you should try to avoid easily dismissing their exposure to technology. Technically hopeless users are a common myth in the programming community. However, in this day and age, you may find that many of your users have a greater exposure to the different technological paradigms than many of the programmers do. It is often easier for users to keep abreast of the current technologies and some of them do so with far more enthusiasm than the professionals. Be honest in how you measure them. The entirely hopeless user, the one that can't even work a mouse to access a basic web page is fast becoming more of a legend, than an actual person. Software developers need a full and honest understanding of their real audience to allow them to design and build the correct tools. Underestimating the users leads to weak functionality, which effectively cripples the software.

23 Your Audience

The advice given to writers is to know your audience before sitting down to write. The work is always tailored to an intended audience, so that it is more likely to be read. Writing differs from programming in that, the ultimate goal of programming is for the code to be used, not read. Because of that, most programmers consider the only audience for their work to be the computer itself, and so they focus on making the code run correctly, while downplaying or ignoring any other possible attributes of the code.

Maybe it is just a by-product of the current industry, but at this moment, software rarely remains static. There are constant changes to the underlying dependencies and quite often, a need to expand the existing functionality of the code. These pressures insure that most software is activity under development. There are of course lots of examples of software that has been entirely static and unchanging for decades, the y2k bug sheds light on many of these types of systems, but that is probably only a fraction of all of the software we activity use. The rest is changing, or heavily dependent on pieces that are changing such as the operating system frequently.

What change really means to a programmer is that they are very unlikely to be the one and only person to ever have to look at and work with their code. It is much more likely that many, many programmers will contribute to the source during the entire life of the code. The audience for any code written is far greater than just the computer. The life span of the code may ultimately be much longer than the programmer expects as well, although it is very hard to predict how long code will really stay in active use.

As a programmer, you need to assume that their audience is both the computer and a large number of programmers. You need to structure their code in a way that makes it easier for your peers to fully understand all of the proper workings of your code. You need to contribute any extra information to the documentation, comments or source code that helps to clarify the methods behind the madness. There is a great deal of similarity between a writer sitting down and considering the audience, and a programmer envisioning the way that someone will see their code at some point in the future.

These other programmers in your audience are equally as important as the computer is itself. You write code because you want to build something, so it is important to ensure that what you have built lasts as long as possible and is used by as many people as possible. The secret to achieving that goal is to ensure that any future programmers would prefer to build on top of the quality of your

work, instead of plotting to remove and rewrite it. They would make that choice only if building on top of your work was clearly the easier and less painful route to take. If the design was general enough to be extended, and the code was clear and easy to understand the course of any future programmers is almost certain.

24 Development Planning

Software developers often throw themselves at a project with a great deal of urgency and pound out as much code as possible. Culturally, this is the most common approach taken to meeting critical deadlines, but rarely is it an effective way to make use of the available resources. A little planning can go a long way and the ability to look out into the future, see a clear path and then stage the direction of development onto that pathway can save on the endless tasks of reworking the same basic underlying foundation. Software development shares a great deal with the other engineering based disciplines, despite looking malleable; once the code for a system is laid down, it becomes increasingly static. An undeniable requirement for backwards compatibility further helps to enshrine early design mistakes into costly architectural flaws. A controlled approach to building systems that covers the requirement influences, design focus, tools usage and development process can significantly reduce the amount of resources needed to initially build a system and to keep it in active development.

Understanding this lack of flexibility, early in the process allows you to extend the design of the architecture around not only the technical and user-domain problems, but also the around development environment problems as well. All three need consideration and proper handling to avoid allowing the problems to escalate and spiral out of control. You should not overlook the influence of the development environment in the design, it is better to deal with it explicitly then try to ignore it. The available resources, required delivery dates, internal processes and organizational culture all have significant impact on the outcome of the project, so they should be accounted for within the design and architecture.

Software is just a tool to manipulate data, so a prerequisite to any good design is a through understanding of the required manipulations and their underlying data. Often however, only a small part of the real usage of the software is truly understood in the beginning of the project. You can use an iterative approach to the development to prevent costly mistakes in building awkward or unused functionality. Short development cycles become important, but only if they are explicitly directed by quality feedback from your users. Using shorter development cycles may seem less efficient, but not if you compare it to backtracking in the development and redoing, large portions of the system because of series design limitations.

Focusing the development on providing clear and simple abstractions for the design can help you to reduce the overall amount of code involved in building a

complex system. The first pillar of a good design is to validate that all of the required data by the tool will make it into the system. Focusing on the data long before concentrating on any ‘functionality’ of the system saves in not redoing redundant storage, copying or handling on the internal data. Once the data can be imported, stored, accessed and exported, extending the functionality available to meet the user domain requirements is clean and simple. Your coverage of the basic handling of the data should be as complete as possible. It forms a critical part of the foundation on which the rest of the system is built.

Setting standards, using the appropriate tools and defining a development process are also ways in which your resources can be used more effectively. Standards help to reduce the overall work and complexity by making it easier to understand or automate the underlying processes. Tools like source code control and bug tracking automate important parts of the development process and help to keep it organized. In the beginning, they may be seen to slow you down a bit, but in the crunch, they ensure that your development efforts are not lost or duplicated. Doing the right work at the right time is critical, so automated storage, tracking and planning tools help to insure that this happens correctly.

The most popular technique for building software is to get a small army of smart but not heavily trained programmers to belt out a massive amount of code as fast as possible, while another army of testers is continuously beating on the code to find and document its weaknesses. This brute force method clearly works and the low quality output it produces has been accepted by the public for many pieces of commercial software. However, software can be build to operate at a higher standard and it can be built with significantly less resources. To accomplish this, planning, understanding the environments, applying significant abstractions and making good use of the available developments tools help not only to keep a project on track, but also to do it with significantly less resources. Good quality software doesn’t require an army to build it, just a development team with an understanding of how to make effect use their resources.

25 Build Inversions

Encapsulation is a standard design technique to hide the internal details. Software components encapsulate their complexity allowing them to act as an opaque ‘black box’ around a specific set of functionality – preventing others from being able to see the details. With the details safely hidden, the only access is thru the interface, so the box must be treated as a single atomic element. This concept and practice exist to reduce the development complexity by decoupling independent solutions. The purpose of encapsulation is not to hid things for the sake of hiding them, but to take that complexity, resolve it and then keep it away from another other set of problems. In this way, the details are handled at multiple levels, and can be distributed to different teams of developers.

Complexity is the single most significant problem in software development — finding techniques to control it is important in being able to complete development in a reasonable time with good quality. Encapsulating the details is an important part of controlling complexity. A basic software system is huge, so that it would be extremely rare for just one team of developers to write all of the code involved in delivering a complete solution. Because of this, an important part of development is working with pieces that where created by other developers. Relying on pieces built by other programmers helps to control complexity, allows the development to work within a compatible set of programs and can fit a complex development effort into a smaller timeframe. An important attribute for these pieces is the encapsulation of the details, so that the programmer can use the component, but not have to acquire significant knowledge to do so.

The culture of programming likes to provide the greatest possible degrees of freedom. As part of this, programmers building code tend to add a great deal of flexibility into their designs. So much, that sometimes it is difficult to understand all of the possible flexibility, let alone even try using a small portion of it. Overkill in exposing too much variability is a common problem. This type of ‘build inversion’ acts as the opposite of encapsulation – instead of hiding the details it pushes them back upwards. Over time, these interfaces grow in complexity and often get extremely quirky. Instead of reducing the complexity within the system, they actually manage to increase it.

If you are working on code that will be used by other programmers it is best to understand that they would most appreciate your work if it truly simplified the problem for them. Computer Science is strange in that the longer it has been practiced, the more complicated things have become. You would think that after almost half a century it would be easier to build complex systems, not

harder. When technologies come along that really encapsulate the details, they generally win great favor amount the programmers. Unfortunately, at present this does not happen often enough.

If we are building what we should, in time our jobs should be getting simpler as more and more solutions exist to solve more and more of the complex problems. As our knowledge increases, we should be able to build on these earlier works allowing us to produce larger more stable systems. Currently, this is not exactly what appears to be happening – our systems are larger and more complex, but they are also less stable than their predecessors. The bulk of the mission critical systems, particularly financial and accounting systems, still run on the older but more reliable mainframe technologies – ignoring decades of new but unreliable technologies. This seems to indicate that we are still in the very early stages of assembling our understanding of how to properly build complex systems.

26 Technical Renaissance

The software dark ages — a time when bugs proliferate, projects fail, software crashes, and complexity spirals out of control. A time when the underlying technologies fall dramatically short of their surrounding hype and promises. A time when the most popular approach to building software is to flail away at the keyboard until all hours in the faint hope that producing enough code that mostly works will be considered acceptable. For nearly all of the world's computer programmers, this is our reality every day we sit down to build systems.

The many myths and misdirections that make up the programming culture may have helped in the past to move things forward, but they are clearly holding back all of the software science and engineering disciplines. Objectively, it would be difficult for anyone to claim great successes over the last couple of decades in increasing development techniques, it would also be difficult for anyone to claim that the current process of building software is anything but haphazard. Given that we rely so heavily now on our technology to remember data that is critical to our societies, it becomes even more concerning that this cornerstone is so carelessly handled.

Certainly, the knowledge and understanding to build sophisticated systems is available – building software is not in itself a particularly complex problem. Over the years bits and pieces that work have been better understood, yet there is no cohesive understanding that pulls together the working techniques and discards the worst industry practices. The focus in building software too often falls away from the code itself and lands on secondary factors that fail to properly control the complexities involved in development.

Our understanding of software development should allow us to overcome these types of problems, yet while the knowledge exists, it does not come together as a usable methodology for development. The more classical approaches drive brute force implementations, while some of the newer agile ideas tend to counter decades of understanding gained from building complex systems. Each in its own way bends towards the extremes, each focuses on tangible aspects of the work without getting overly concerned about the one truly important issue – the code itself. Culturally we generally avoid saying what the code should be because this is perceived as a critical lack of freedom for the programmers. Pragmatically, this is the source of a great deal of problems.

While the current development culture doesn't change, the by-product of this culture will not change either. Every year the technologies grow more complex

and more cumbersome. We are gradually losing control of our ability to build functioning systems – the foundations are increasingly fragile – the resulting systems are also increasingly fragile. We are in need of an inspired leap forward that would put software development back on track for providing positive improvements in peoples lives.

27 Data Directed Design

Software, while complex can always be decomposed into sets of functionality that is directly applied against various collections of data. In its essence, software is nothing more than a tool to manipulate data with specific functions. As such, the core of any design for software need only address those significant issues involved in making the software work. Many people believe this to be the final high-level functionality of the software, but that approach tends to draw the developers into building code that becomes redundantly long sequences of steps taken to implement specific functions. That produces fragile and static systems. Instead, focusing primarily on the data and ignoring the bulk of the functionality until later dramatically reduces the complexities within the development while providing a simple architectural model for the design that is considerably stronger and easier to deal with.

Software applications are simple tools that need to get data into the running system, by either importing it in some fashion or retrieving it from a known persistent storage location. Most of the core functionality for applications involves viewing or editing the data once it is within the system. Completed data is saved persistently for later or exported out of the system in a variety of different formats depending on the needs of the users. For the vast bulk of functionality that exists in applications software, the above operations are the only ones needed for implementation. For large volumes of data, there may also be navigational issues – including various methods of searching to bring the user to the correct data much faster. Some applications have at least one large self-contained computational piece of functionality, but this can be wrapped into a ‘black box’ with all of the details encapsulated and called something conceptual like ‘an engine’. Sometimes the display of the viewing or editing of the data is complex as in the case of WYSIWYG editors where the internal system data structure is quite different from the visual presentation of the data. This type of problem can be contained in its own independent presentation logic.

The above very simple model covers all of the base functionality for most applications. Specific functions can be added on top of the base to give it more depth and to make some tasks easier for the users to complete. With this perspective of software, it makes it easy to see the base foundation necessary for an application and to build the functionality on top of this base to make sure it does not become too redundant or complex. This data directed approach is not only simpler to design, but also faster and more likely to be complete. By constructing a foundation that controls the data and makes it available to the rest of the application we have created a solid base on which we can build in all

of the remaining functionality as either simple access to the underlying foundation primitives or as encapsulated engines that are feed specific inputs and return completed outputs.

We can call this Data Directed Design – an approach to software design where you focus primarily on the data that needs to be manipulated within the system. The remaining functionality is secondary and can be composed of sets of primitive operations acting on the data contained in the foundation. This greatly simplifies the design of your software and helps to break the problem down into easily manageable tangible chunks. You need only to find out all of the different types of data that are necessary in the system that you can then add into the design. Suggested functionality acts as indicators to help find and add in the different types of data that you need in the system. There may also be architectural features that come from environmental or technical issues, but these can be added into the design at a later stage.

This general design can be extended to handle any problem domain and extra technical requirements. It is an extremely good starting place for any large architecture. The key is to focus on and build a foundation with complete base access to all of the data within the system, long before even considering what functionality is necessary to work for the specific problem domain. This approach can be done as one big shot, or iteratively using or extending each individual data set at a time. So long as the data is unique and independent within the system, it can be handled as separate components within the foundation making it easy to distribute the work to different groups. Data Directed Design reduces the complexity of designing systems, reduces the work in building them, allows the work to be easily partitioned and helps to insure that all of the resource will be effectively leveraged.

28 Iterative Development

In any development, there are some things are within the purview of the workers, while others are beyond their control. For software, the development process drives the priority and allocation of the work in a project. The organizational environment sets the philosophical goals for the work and further helps to define the way resources are utilized. Both have a massive impact on the outcome of software development – often it is more significant than the individual developers. Environments are generally slow to change and create issues that just must be dealt with – there is limited or zero flexibility. Process, however is usually within the scope of the senior developers influence, even if they are not explicit aware of it. The selection and application of any process methodology for development will have a huge impact on the final outcome of the project.

A full encompassing methodology for software covers more than just the design and development – it includes aspirations, requirements, deployment and feedback as crucial parts of the process. The focus is in minimizing the resources necessary to produce quality software. Development resources are always scarce – prioritizing them and making more effect use of them is an important attribute. Software is always in continuous development, so time should be an important element in any process. Development continues on a slow but steady path over a large number of releases. Frequent changes to direction, back and forth, both decrease the quality of the results but also fail to utilize the development resources in the most effective manner. Accepting this, and integrating the speed of development into the process allows for more effective strategies over the long run. Changes can be clumped together in one cycle that are not fully utilized until a later cycle. The most important attribute for a methodology is that it needs to focus primarily on the core output of the development – the code itself. While it may be easier to frame a methodology as deliverables based on documentation, choosing to focus on anything other than the final source code produced can lead to chaos and easily misinterpreted results.

An ideal methodology would define the process of development as a sequence of cycles of development. While a broad stroke outline should exist for the long-term development goals, each iteration in the development should focus on a more tangible set of code, testing and documentation changes. Working backwards from the code, only the minimal amount of information required should be collected and processed. Within a data directed design, gathering the data and all of its attributes becomes the prime material for the code production. On top of that foundation, the higher-level functionality gets built

on top of the base, requiring it to be outlined in terms of the primitives from the system. Iterative development fits well with applying a data directed design approach to handling the architecture. Many new feature requirements represent incomplete systems, as the last iteration may have started specific functionality groups but did not fully complete them.

The focus of iterative development is to find the requirements, model them in terms of the underlying data and architecture, extend the base foundation, add in the higher order code, minimize the testing, deploy the code and collect the feedback to merge into a new set of requirements. This cycle is repeated until interest is lost in maintaining the software. Feedback drives many of the new requirements, which in turn drives the new underlying data to get added into the system. Reorganization, which includes refactoring, scripting, setup and cleanup is always the first stage of any new development cycle, without it, the code base degenerates quickly. There is always some place or construct in the code that is not right, was hastily implemented or has drifted away from the conventions. Multiple conventions, half finished changes, old code, and inconsistencies all increase the complexity of the code. Addressing at least a small cleanup in each cycle helps contain the complexity of the code. Automation for common tasks is also important to add in here as well. Scripting is an important tool in automating the common development tasks, which both helps to bring in consistency to the process and saves time in not having developers find individual ways to accomplish common goals. Issues such as testing and language translations come at the end of each cycle, often as late as possible to prevent binding the code to incomplete design issues. The final steps are always the building, packaging, delivery and deployment of the final code release.

The size of the iterations drives the efficiency of the project, and inversely the degree of risk to which the project can get off course. Smaller iterations are less efficient for resources, but also less risky. Generally, the environment and resources dictate the iteration size, however, in some cases the architecture can play a significant role — some architectures are better handled in smaller iterations, while others need long iterations because the changes they require are huge. Iterative development allows for a tighter integration between the feedback and the next set of development changes. If you control this process, it will allow for a roadmap for the implementation over the next set of releases. Any reasonable methodology for producing computer software systems, should strive to insure that the results of the process are really successful, not just from a resource organization standpoint, but also from a functionality and usability standpoint for the final tool created.

29 Extending the Foundation

Resources are always limited within software development. Usually even the simplest of goals requires significant work. When you think about the teams of hundreds of people who spend years and years building most of the large commercial software products it becomes easier to understand how it you can invest a massive amount of time in designing, implementing and testing code.

It is very important, given the resource restrictions, to insure that any work completed is as efficient as possible – not just over the next set of changes, but throughout the entire lifespan of the software. Some inefficiencies, however, just have to be accepted – true optimization comes not from trying to force an ideal, but from accepting the reality of the underlying domain. In this case, rewriting code multiple times is part of software development. Many software products are the outcome of no less than two or three total rewrites of the code – most software is rewritten far more often than that.

As the code grows, the initial infrastructure becomes quickly strained and needs significant refactoring or rewriting work. The 1.0 architecture rarely survives to the 2.0 version – 3.0 replaces large parts of 2.0 – this continues for each major version. Over time, as the code stabilizes the changes generally become smaller and more focused, but with each new plateau, large portions of the initial code must change to make way for the new structure and functionality.

Rather than fight this problem for as long as possible, forcing even more code to be written as temporary patches, developers need to accept that the next major version will likely out-grow the current infrastructure. This isn't an issue to avoid — it is an important part of choosing the architecture for the system. Picking an abstract component architecture that leaves room for entirely rewriting segments of the code on a component-by-component basis is a very critical attribute for a good architecture to have. This plus good clean interfaces and reasonable encapsulation can significantly minimize the impact of the major rewrites on the rest of the system.

Drawing lines around the major components helps to contain major changes to specific parts of the system. The Speculative Generality code smell suggests that programmers avoid adding in functionality that might be used in the future, but given that this is based on the definite knowledge that the code will need to be replaced as the system grows, it is not speculation. The extra work in implementing this type of compartmentalization is quickly paid for with the first set of rewrites. By the mid-point of the lifespan of the development, systems built with this type of architecture easily account for only a portion of the same

development effort. As development progresses, the existing code should be leveraged to make adding in new functionality easier with each iteration of the system. This is another important attribute for a good architecture, one that will survive the iterative process of development.

If you can admit that you will end up rewriting the code at least once if not multiple times during the lifetime of the project, you will be able to understand how to better structure the code so that rewriting it is an easier problem. In this way, a bit of work now, will save a lot of work on a future rewrite. Also, as an added benefit, choosing to rewrite a section of code becomes an easier decision to make while scheduling the next set of development. This added flexibility decreases some of the pressure that comes from repeated continual development on a system.

30 Persistent Storage Architecture

Software systems need to keep data around for a long time – far longer than any program is expected to be continuously running. For this reason, most of the data within an application needs to be stored in a long-term storage mechanism, usually called a database. The storage of data from a program is a huge issue because it must exist in virtually every system to some degree or another; it accounts for a huge amount of code and has a big impact on the overall complexity of the system. The synchronization of the data between the internal representation and the persistent one is a key architectural problem.

Although there are many different solutions to this problem, a popular choice is to use a relational database to hold any persistent data. The relational model offer a great deal of functionality including data consistency, locking, sophisticated reporting access and an ad-hoc scripting language that allows for many complex customized tasks to be executed directly against the data. Developers like the relational model because it is simple, easy to use and easy to manage. Companies like it because the same database software can be utilized consistently across an enterprise, reducing some of the operational problems.

Utilizing a relational database in a system's architecture can lead to a number of very difficult choices. The relational model and many of its implementations offer a variety of features for the programmers, but they do so at a high cost. A software architect must make a choice when designing a system – should the runtime version of the code be the primary source of data or should the relational database be considered the primary source of data. Mixing the precedence from one source and another within the same data is likely to massively increase complexity – consistency with this issue is critical for simplifying a design. Explicitly or not the programmers must choose which data has precedence over the other data – consistent rules result in significantly less complexity. Ultimately it comes down to a software designer choosing between utilizing the all of the functionality of the database or utilizing all of the functionality of the development language. This choice affects both the complexity of the system and its overall extendibility.

Most relational database implementations offer excellent data consistency tools, extensions to the basic model and ways around the inherent limitations. For simple data with simple functionality the model is sophisticated enough to allow for many applications. If an architecture were to utilize the relational model, then the internal elements of the code should closely mirror the arrangement of the data in the database. The cleanest code would model the internal data as a sequence of table data-structures and allow the rest of the program to interact

directly with that data. Attempting to convert the data to another format or applying synchronization rules beyond those available in the database tends to lead towards extra complexity.

For anything other than simple data with simple operations, this design approach will run into trouble quite quickly. The relational model is very simple, but this came at the cost of making a large number of trade-offs. Because of the underlying set-based mechanics, the expressive power of the relational model is significantly less than the expressive power of any standard programming language. Although many databases now have specialized approaches for handling them, the relational model does not work well for common data types such as hierarchical or time-series. Large featureless data often has problems as well, and the relational model cannot handle graph theory or matrices. Because of these limitations, mathematical, geographical, financial and historical databases all fit poorly into the relational model.

In most cases, to really utilize the relational capabilities, the programmers must make a Faustian choice and tie the program heavily to one specific software vendor — for most software this is a poor choice as it limits the options of growth in the future. Companies arbitrarily change technology. Software needs to be as agnostic as possible to allow it to appeal to the largest possible audience. For many development projects this can lead to serious problems with vendor dependencies, in the worst cases all or most of the software code could be abandoned. Minimizing these types of dependencies within an architecture, helps to insure that the work in building the system will last for as long as possible.

The other approach to persistent storage architecture is to utilize the full functionality of the programming language to implement complex data structures. In this case, the internal representation of the data is too sophisticated to be adequately represented by the relational database. As just a third party system that is responsible for storing the data, the database should be used to its maximum potential, but it should not drive the internal architecture. Reasonable usage still allows for direct ad-hoc scripting and reporting access to for most of the data, which reduces the amount of code necessary for the system and provides better operational access when there are problems. However, this should not affect the rest of the architecture of the system. The database schema is only a secondary representation of the data used to hold it for long periods of time. The internal representation should be the primary representation and should be structured to reduce the complexity of implementing the required functionality on that specific data set. That code in any system represents a significantly large percentage of the actual work to build the system; big reductions in complexity have a direct effect in reducing the overall amount of resources needed to build the system.

For a small system, utilizing the database as the relational model for maintaining the data makes sense because it is likely less work. The developers can leverage the capabilities of the database and use it to avoid writing their own code. As the size and functionality of the system grows, the complexity from trying to get around the limits of the relational model quickly exceeds the complexity from mapping an extended model back to a relational one. This occurs because the mapping backwards to a relational model is a one-time consistent problem. If the full model of the data exists, the same underlying consistent techniques can be applied to the work. An abstraction applied to this problem can keep it quite manageable. On the other hand, each fiddle in the code to make a new way around the limits of the relational model will likely be unique. Over time as these fiddles grow they can create a code base that is increasingly impossible to understand. The mechanics of the code quickly grow towards being spaghetti code, and as such the complexity will increase exponentially causing it to outpace any other complexities.

Pushing the bounds of a limited model will generate more complexity than mapping a complex model onto a more limited one. At some point the benefits gained from utilizing a relational database are not enough to counter balance the extra code that needs to be maintained to keep the system running. Clearly the intended size of the system is an important consideration. While it may be quick and easy to take pre-existing pieces and combine them in a way that utilizes their functionality, ultimately that will always be the more limited approach. There is a cost associated with everything, picking specific pieces of technology is no different.

31 Partial Encapsulation

The central idea behind applying encapsulation in software is to hide the details so that anyone outside of the implementation is free from having to understand the details. When handled correctly, the user of the encapsulation is given an abstract interface that hides something more complex underneath. The only reason for applying encapsulation is to reduce the complexity of the overall problem – the interface must be significantly simpler and provide access to the same functionality.

There are many reasons why an attempt to encapsulate may not meet the required attributes to be useful. A very common one is where the developer both provides an abstract interface but allows a lot of the underlying complexity to get pushed upwards. This partial encapsulation is particularly troublesome as it eliminates the usefulness of encapsulating the problem, causing it to be more complex, and it also doesn't properly hide any of the existing complexity – allowing it to nearly double the complexity.

A common cause for partial encapsulation is the developer wanting the interface to have access to the full underlying functionality. The culture of development often dictates that any options from below are made available to the higher levels. In that way, a new interface is created, but none of the key degrees of freedom are properly contained or eliminated. Instead the developer simply pushes most or all of the complexity back upwards towards the user.

A strong indicator of this problem in a module or library comes from examining how the interface is used. If the possibility exists to abuse aspects of the interface, then the abstraction is not really containing the basis of the problem. If the knowledge required to utilize the interface is equal to or larger than the knowledge required to utilize the underlying functionality they there clearly is a problem. There are so many examples of both of these cases in practice. The problems are so widespread that a large number of programmers frequently mistake poor encapsulation implementations for software best practices.

The absolute rule of thumb should be “does using this implementation make the code simpler?” Building on top of well-encapsulated functionality will require less time, less resources and less concentration. The encapsulation should allow the programmer to spend more time focusing on other problems. If this is not the case, if the implementation is requiring significant resources, then the likely cause is partial encapsulation.

32 House of Cards

A foundation should form a solid layer on which to continue building. In software, we have chosen to factor our software architecture into well-defined technologies that combined to form the foundations for our user applications. As such, we generally choose a programming language with some set of libraries that interacts with a specific operating system. To this we add a database implementation and often some GUI framework to handle a significant amount of the technical issues.

In a paradigm like Java, when building a web based app we may also add in other well-defined pieces such J2EE containers and our web servers. Each of these different technologies is aimed at encapsulating some aspect of the technical problems involved with building applications. Adding to this are the secondary construction technologies such as ant for build scripts, CVS for source code control and some type of editing tool such as NetBeans.

A lot of trouble comes from the idea that these technologies are correctly encapsulating our technical problems. Experience with many generations of technologies makes one sensitive to the increasing complexity of many of these pieces. While they purport to encapsulate the technologies and make the problems simpler, they often require more knowledge in order to properly take advantage of their features. As such, over the decades, the process of software development has massively increased in complexity. It is true, that the by-product of development, the software itself, has also increased significantly, but the key question comes down to whether or not the underlying technologies have increased to a greater extent?

In the past, it was entirely possible for a programmer to have a nearly complete understanding of the development platform at multiple levels. Answers could be achieved by guessing the correct answer because the knowledge base was detailed enough to understand not only how the technology worked, but also how it evolved over time. Within most of the popular modern development environments, the sheer volume of knowledge required to solve even the simplest of problems quickly removes any potential to fully understanding the underlying technology. In some cases it is because of the inherently increased complexity of the problem domain; modern fonts for example, are very sophisticated so very few people really understand how they work anymore. However, in many more cases, it seems as if the underlying designs for the component pieces are both inordinately over-complicated, and only partially encapsulated.

Two very different powers are at work here. One is that it is common to make technology more complex than is necessary. In software it is also very common, once this happens, to not ever revisit the problem with the intent to fix or simplify it. The second problem is the cultural need to not restrict another programmer's freedoms. Even if you are writing an interface to hide the details, you still want to give anyone using it the option to change the settings and apply it badly. This behavior creates new complexity from the encapsulation and then adds to it by exposing too much of the underlying complexity. The net results are technologies that not only fail to hide the details, but also managed to make the original problem worse.

Certainly in the past, specific branches of technology became extended to the point where the increases in complexity overshadowed the usability. The developers essentially get bogged down in the details and the technology stops growing, and starts to die. There are a number of popular technologies that have clearly reached or exceed this point, but now the problem has become greater than any one specific branch. We now desire functionality that is breaking the paradigm of many of our core foundation technologies — this problem goes far deeper than most people expect. If we want to move forward to any degree and build the types of systems that our users would like, then we will have to entirely refactor our base building blocks. To be successful, we'll need to hold nothing as sacred — specifically we will have to change our current model — we should not be afraid to rearchitect base technologies including ones like operating systems, databases, applications, libraries, processes, threads, machines, etc. The new paradigm we are looking for is one that makes effective use out of the resources, solving most of the technological problems, while making it easy for use to implement business domain problems. When we get a foundation that really encapsulates the technical problems, we will be free to apply our energy to building better tools for the users.

33 Attributes of Code

Focus in development is often concentrated on the issue of whether or not the code is working correctly. Working, however is just one particular attribute that can be ascribed to a block of code. There are many others – essentially they are properties of the code above and beyond basic existence. Some like conformance to standards are objective and easily verified. Others, such as elegance, tend towards the subjective and can differ somewhat between developers. Readability, sometimes described as the code being well written is another important one. Following a lot of other similar definitions, there are an infinite number of different attributes that can be defined and attached to any specific block of code. One of the key components of controlling complexity is understanding which attributes are the most significant in ensuring that the development project reaches its goals.

While there is still time, the working attribute is not necessarily important. Well written, well-structured code – two very critical attributes – can easily be fixed to remove any known bugs. Overly complex and confusing code on the other hand, even if it works, will always be a struggle to fix, maintain or extend. Cascading bugs are commonplace in problematic code, so many more rounds of bug fixing are required to stabilize the operations – frequently exceeding the scheduled debugging times. With limited resources, producing a clean well-structured implementation is far more important than grinding out a large amount of poor quality code. Simple, elegant and readable code — particularly code that is generalized and can solve a wider problem domain — is far more valuable than a mass of poorly thought out, brute force code that is fragile and hard to extend.

Development, particularly in the early stages of a new iterative cycle, should concentrate heavily on getting the underlying attributes in code to be correct. Refactoring early to make the code consistent and understandable is an optimization in the use of development resources. Waiting only makes the problems worse. No matter how the code is optimized towards its different attributes, the bottom line is always the same. The code should be as simple, short and consistent as possible. Clever, intricate or overloaded code is easy to misunderstand and should be avoided. Multiple different blocks that essentially do the same underlying work are examples of really bad code – being both repetitive and making the competing calls inconsistent. Reducing the amount of code, especially if the reductions are large and ultimately simply the original code, help to insure smaller, tighter source code. The true skill in programming is to take an extremely complex problem and implement it with easily

understandable code. Making hard problems look simple is the essence of elegance. Elegance in code is an attribute that always pays for itself.

34 The Myth of Relational Databases

Lots of people believe that relational databases are the perfect, one-size-fits-all solution to both act as the persistent storage method and to anchor the system's architecture. This view, often so strongly entrenched in myth causes developers to ignore the obvious problems, go to extra lengths in their implementations, implement fragile solutions and often start death marches all because they can't admit that the underlying technology is not suitable for their specific type of problem.

It is not to say that Relational Databases are bad or should not be used for many different systems. They are an excellent technological abstraction that can contain specific types of data into a reasonably encapsulated package. Along with persistent storage, they offer great ad-hoc querying capabilities, an amazing array of reporting connectivity and sophisticated general-purpose access to the data. For many systems they are the perfect repository for the persistent data, however, they are certainly not universal. They can be large, slow, bloated containers that force bad design onto the data representation to get around their extremely limited expressibility. Their many quarks are reminiscent of software twenty years ago — they have missed most of the modern advances in dynamic development practices. Sure, some vendors have fancier features and capabilities than others, but should you really use them and tie yourself to only one vendor? Is that safe, given the past history of the dominant vendor changing every five or ten years? Nothing last forever – and there are still at least two major database vendors fighting it out for market supremacy.

Newer features offer nifty things like automatic 'triggers' to allow code to essentially execute directly inside of the database, but the dangers of distributing a system's core logic is rarely understood. In a big system, it can be hard enough to find the 'place' where the problem originates, we don't need to have it originating out of many different places all at the same time. And so, it is often suggested that good practice is to bring together the related processing elements of the system into the same location – a technique that makes it significantly easier to diagnose problems, thus giving more resources to designing and developing the code. Utilizing capabilities such as triggers can be the exact opposite behavior, worthwhile if and only if there is huge benefit to offset those very large and troublesome deficiencies.

Without vendor specific extensions, the base features of Relational Databases are geared towards non-hierarchical, non-historical, flat collections of data. It is a limited problem domain. To get around that, support for jamming in large chunks of complex data using things like blobs exists, but only as a work-

around to dump something ‘inside’ of the database, when it is still clearly ‘outside’. The full functionality of SQL is not available for special data types.

A strong, well-defined and practical amount of knowledge exists to set up a new database with a forth-normal form schema, providing general access to the data. Despite this body of knowledge having been around for many decades, most schemas are denormalized – badly – by their authors who claim their specific deviation from normal is better, but is often based more on instinct and guess work, than on sound engineering principles. Many serious problems start with the initial design. Every experienced software developer has a story about a clever schema gone horribly wrong. It is a common experience.

We can’t move forward to bigger and better systems if we blindly leave our faith in the underlying technologies. The longer you develop, the more you come to understand that a great deal of the current technical underpinnings are more complex, more fragile and certainly more volatile than necessary. Overall we need to refactor our existing foundations because it has become increasingly responsible for the exponential increase in artificial complexity that is making our jobs more difficult than necessary and is making our solutions more fragile than acceptable. Too many operations people and developers have adopted a mantra: ‘we have to have our data in a relational database’, in the hopes that that will somehow magically erase all of the technological problems. Even when a relational database is a good fit, it will not come without a significant large cost to development and maintenance. When it is a poor fit, it might well be fatal.

35 Horizontal vs. Vertical Tools

We build tools for the users based on our decomposition of the problems. We tend towards a functionality related bias, and as such our tools collect together many similar functions and package them as a complete tool. The effect of this design tends to be vertical solutions – tools that solve with great deal a specific subset of the problem domain.

An example is a word processor. The user's problem in its full scope might be to have several people collect together documentation on a specific set of issues. In perusing this work, they might use a word processor to create and edit the document. Other tools, such as email may be involved, but the word processor itself is a limited vertical solution to solving the problem of collecting and modifying large sets of text into a single document.

The horizontal solution would provide a place for the users to congregate and share their information. The final output to a printer might still be a word processor document, but the entire horizontal flow of the work is the true problem domain. Any issue encountered by the people in combining their efforts, and ultimately achieving some goal with the work like publishing it on the web, or using it as the basis for some new project, becomes more significant than whether or not the font used for the text was Arial or Helvetica.

Vertical solutions tend to dominate our industry because they are easier to produce, but the users often crave true horizontal tools, particularly if they repeat the same tasks on a daily basis. When we advance our understanding of software construction, we can move towards building better and wider tailored horizontal solutions so that the users can really utilize the power of a computer instead of just pounding at it.

36 Learning from History

So much has been written about building good software, and so much of it has been completely ignored. Left in the dust. Not relevant. Unimportant. It is unlikely that any other single field has resisted maturing as much as Computer Science. The ideology seems to be based around going your own way, rebuilding the wheel and not wanting to seek out better ways of developing. We foolish few, who as an industry have such aptly named books as “Death March” and “The Mythical Man Month”, yet they go unread, and their warnings unheeded.

We are brutalized in the press from outside, but also from within. Popular opinion tends towards everyone being suspicious of software, with the most general opinions believing that it never works as advertised. While software has been weeding its way into our infrastructure, since at least the 60s sentinels have been warning us about a looming software crisis. Is this all just nay saying or is there really something behind the notion that a lot of software is badly written? In a way, it all hedges on whether one believes that the current fare represents the best that we can achieve. If true, for me I find that particularly depressing, since the bar has been set extremely low, and seems to be on a downward trend. If false, then what are we waiting for? Why is it that there seems to be so little in the way of substantive discussion about improving our works? Does anyone care anymore, or are they just happy that they can take home their pay and afford a larger TV set?

37 Best Development Practices

The same types of problems bubble to the surface over and over. With enough development experience you start to see the common ideas that work and ones that don't. Everyone has their own personal list; this one contains the most frequent issues that I have encountered over the years while developing software.

1. Manage Complexity

To everything there is a great deal of complexity. Some of it is necessary, but lots of it is not. The business and technical problems are inherently complex, so nothing can be done about that. It is critical to fully understand what is real complexity that is really part of the problem and what complexity is just artificial. If you can prevent too much artificial complexity from creeping into the project, then you have a chance of actually getting it to work. If not, an exponential growth in complexity is probably what will kill you.

2. Look Past the Short Term

The favorite excuse is that it was just a 'quick fix'. If only we had known, if only there was time. The truth is, that there is never enough time — development is a slow beast to turn — so you have to make it count. Once you understand the long-term direction for the project, build for that over the next set of iterations. A five-year horizon, if dealt with realistically can always be used to optimize the development effort. Also, programming is a “stitch in time, saves nine” type of profession, so if you fix it now it will save you lots of work in the future.

3. Work Smarter, Not Harder

You can pound at the keyboard with all of the speed and fury in the world, but that won't make the work go any faster, and it won't eliminate the problems, just create more. Stop, think, and find a better way to get the same results. Utilize the existing tools. Apply abstraction to the code. Do more with less. Automate the common tasks. Compact the code, and make it more consistent. Spent some time to see that all of the tools are being used correctly. If you put in the effort to find a better way, the payoffs will allow you to capitalize on the savings.

4. Don't be Clever

A fellow programmer once admonished my clever bit of code with the

statement “Monkey’s are Clever”. He was absolutely right. Clever always causes problems later. It causes misunderstandings. It wastes time. It may seem like a smarter way to tackle the problems, but it will always cost in the end. Never confuse smart with clever. If you fill the code with clever tricks, you’ll quickly wish you were working on some other code base.

5. Always Design First

You don’t need a 200-page full color technical specification with 50 diagrams to justify adding a new screen into the system. But you absolutely must have a design before you start to code. Just think about it and then document it to whatever degree is necessary to deal with your own environment. Some places need a couple of paragraphs; some need a bit more information. Not designing first, means rewriting later. Don’t confuse creativity with flailing around in the dark; they are very different things. If you think about it first, you’ll always save yourself a lot of wasted time and bad code.

6. Don’t be Afraid to Rewrite it.

Extending or refactoring will only take the code so far. If there are serious problems, people often pile more work into maintaining something that needs a rewrite, then it would have been to just fix it by redoing it. In any normal development, the bulk of the code manages to get rewritten at least several time, it is inevitable, so just accept it. If the code can’t be extended, rewriting it as early as possible requires the least resources. Patching it until it becomes nearly impossible to fix later, becomes a sink for wasted effort. Designing the architecture around the idea that the different pieces can be easily replaced with newer more functional ones

7. Only Use Industrial Strength Code

If the code is simple, easy to read, and consistent, fixing the bugs is work that can be schedules for almost any programmer. Messy code, even with few known bugs is a pending nightmare and will eventually cause major problems. It is not hard to apply standards and create industrial strength code and it is absolutely worth all of the extra effort to do so. Sounds easy, but so few programmers get this right in practice. Their chief miseries often are their own inconsistencies.

8. Don’t be Afraid to Talk about It

For some reason developers just want to flail away at the keyboard without talking about what they are doing. The culture shies away from a lot of interaction. That leads to all sorts of problems. Miscommunicated

specifications, design failures, etc. If a team can't get together and talk about the merits and disadvantages of the different approaches, then serious problems can and will occur. Developers need to get in sync with each other, and communication is the key to keeping everyone on the same page.

9. Go with the Flow

Too many times I've seen developers break from the norm and try to go at the problem from a different angle – often they stubbornly refuse to admit that it is not working. All of the bits and pieces on a computer work well only in specific circumstances. If you use the tools the way they work well, and don't try to force them to do things that they can't, then the development tasks get easier. Trying to bend a tool to fit in an awkward circumstance is an exercise in futility.

10. Be Flexible

Eventually, programming changes the way that programmers think. Everything becomes code; the world gets very black and white. We tend towards trying to use logic to solve everything, irrational or not. We become resistant to change. Programmers tend towards sticking with approaches they have used in the past, whether or not they really work. A large number of eccentric ideas build up and we tend to believe them. Just because a technique didn't fail, doesn't mean that it is the best way to accomplish the goal. The world around us does not fit into the software model, and often we must remind ourselves that this is true and that we need to be more open to alternatives.

11. Push the Hard Choices Back to the Users

Programmers want to believe that they really understand what their users are trying to do with the code. It is hubris to put one's own understanding above those who actually use the code for a living. The users may not be complaining, but that doesn't mean that it is right or working. Don't make hard choices for the users, or the administrators – build the solution to allow them to configure it to their specific needs. If you give them the freedom to arrange it a fashion that suits their needs, you'll be surprised as to how well the system will adapt to the users. Its just a tool, but it works better if it fits the problem.

12. Make The Interface Consistent

This seems to be an incredibly hard problem for most programmers. Consistency is boring, and doesn't involve solving any new problems. Despite their dedication to logic, its like pulling finger nails to get most programmers to implement interfaces with any degree of consistency. The users absolutely notice, whether they complain about it or not, and it definitely gives the

interface a 'sloppy' and 'hard-to-use' feel if its inconsistent. A simple consistent interface with limited functionality is worth more than a huge amount of badly packaged functionality with fancy graphics. If the users have a job to get done and they have a choice, they will always pick consistent. Also, don't forget that libraries, config files and command line applications are interfaces too and need to be consistent as well.

Understanding the user's needs enough to build simple tools that actually reduce the work is a very difficult problem. Software development has been around long enough that translating that knowledge into a reliable working system for the user shouldn't be difficult. Most problems with software development come from the environmental and human complexities surrounding the project, not from the technical or problem domains. If you use the language and the tools that way they were meant to be used, and follow all of the best practices, development of software should be reasonably straightforward. It's the personalities, political alliances, incomplete or incompetent designs and total lack of standards that will always bring down the project.

38 The Trouble with Declarative

One approach to development is to opt for a declarative type paradigm. This way, the interconnections between the various elements within the software are formed explicitly by declaring them within configuration files. An example of this is the struts framework. To define the system, the programmers create very large and complex XML configuration files that specify the relationships between the presentation elements and the functionality within the system. An alternative towards this type of explicit mechanism is to build in the inter-linkages implicitly into the definitions. Minimizing the attributes of the relationships between the various blocks of code becomes possible when the blocks are heavily typed and their behavior in the system is tied to directly to their type.

In either case, the end-goal is always the same. Tie together various blocks of code to an existing foundation of code so that the base technical problems, which are solved by the framework, do not have to be re-solved by the programmers. The key for this type of foundation is to allow for the largest number of technical problems to be solved with the minimum amount of complexity while not limiting the programmer to a restricted subset of development. With that in mind, the criteria for judging the quality of several frameworks would be to look directly at the code they force the programmer to implement in order to take reasonable advantage of their capabilities.

Using a declarative type interface pushes the programmers into specifying their logic in a) multiple places and b) repeatedly for each slightly different case. For the first issue, there is no real way around having some of the mechanics in the configuration file and some of it in the actual implementation. For a small system this may not seem an unreasonable burden, but as the size grows, this becomes a considerable problem. Building medium to large systems demands that the programmers get the maximum value for any complexity added towards the development – without it, the projects tend towards exponential complexity increases while derail the work. As the configuration files double in size they probably quadruple in complexity, if not more. They become unmanageable, with no reasonable way to fix the problem.

For the second issue, there may be short cuts within the framework to minimize the similarity between the different entries, but they come at a cost of significantly increasing the complexity. Packing together a large set of declarative statements with techniques like regular expressions is both costly and confusing. Just making them implicit in the first place quickly surpasses the other option. For both of these issues, the result for applying a declarative

approach is a significant increase in the overall complexity. As the system grows the declarative complexity will grow faster at an increasing rate. It will eventually become fatal.

This type of problem is common in software development. What works rather nicely in a small system, quickly becomes a nightmare in the medium and large ones. As the size and scope of the project change, the techniques and styles used for development need to change along with them. A little extra complexity with 20,000 lines of code becomes a critical design problem with 150,000 lines. Underlying this problem is the inability of the programmers to isolate a chunk of the system and encapsulate it cleanly into a subset. In a sense explicit declarative interfaces are the complete opposite of encapsulating the code – the complexity and mechanics gets turned inside out and becomes global to the system. Once this type of complexity is exposed it acts as a threshold to limit the overall size of the system. It is only by encapsulating the pieces of the system that the programmers can break down the development of a large system into pieces that are small enough to digest and make function correctly.

39 Industrial Strength Language

Computer Languages have matured a great deal since the early punch card based languages such as FORTRAN, BASIC and COBOL were invented. Still it is easy to see that there is considerably more room left to grow into before the underlying languages ease some of the aspects of software development. Strange syntax, excessive complexity and awkward mechanics shackle programmers to out-dated processes for complex application development. We need an "industrial strength language" that doesn't hamper development, but by design aides in completing increasingly complex software systems. The computer is a powerful tool that can automate a great deal of mundane work, but it is left to us to find ways to truly harness it.

With that in mind, these are my top 9 attributes that I would like to see embedded in a new language definition. The combination would truly make it industrial strength:

1. Only one way to do it.

Multiple different ways is nice for creativity, perhaps in an art project, or something. I don't want a paper mache sculpture, I want a software tool that I am sure works properly.

2. Enforces all syntax (and semantic) conventions.

The language may leave some wiggle room, but for a given project if the developers say that the lines of code must be indented by four spaces, then the language should refuse to compile any code that doesn't meet the standard. It can be configurable so that different projects have different conventions, but ultimately the computer is the right tool that should 'enforce' the standards (and there should only be one for the project).

3. Small language.

Large languages are hard to learn, so few programmer do. What results is programmers trying too hard to utilize only 70% of the language mechanics. Unwanted complexity. Lots of wasted effort.

4. Encapsulation.

Enforced encapsulation, with no reverse engineering. A real black box. The interface is properly spec'ed and documented or it is tossed.

5. Elegant.

The language relies on combining axioms in an elegant, but not clever manner. Falls back into there being only one real right way to implement the solution. Anything else is just hacking at it.

6. Simple, clear syntax.

Without being obscure or confusing, the syntax is as small as possible. COBOL is at one end of the extreme, APL at the other, an industrial strength language sits in the middle. Somewhere.

7. Lots of primary data structures build into the language.

Why re-invent hash tables, trees, arrays, linked lists, queues, stacks, pagodas, etc. over and over again. These should be build right into the basic definition of the language with the ability to 'statically' initialize all of the structures, no matter how complex.

8. Support for abstract programming.

Objected Oriented was a step in the right direction, but somehow it is frequently abused and a lot of "OO" code just isn't. Partitioning the code into sub-pieces is a critical problem in containing the complexity. Done properly, it helps minimize the amount of code required to implement a solution. Done poorly, its just an alternative to spaghetti code. A good language needs to provide a stronger approach for partitioning code in a reusable manner, something more data driven.

9. All rope needs to be isolated.

In C it was pointers, in Java it is thread concurrency. Language designers leave rope for the coders, which they subsequently use to hang themselves (or at least waste a lot of time in not coding properly). If it must exist, it needs to be set in a way that it can be properly encapsulated from the rest of the code. There should be a clear line drawn to differentiate the code used to solve technical problems from the code used to solve the tool's problem domain.

Summary

There are probably a lot more core issues that should be build into an industrial strength language. I could see the possibility of leveraging databases to help guide the construction of clean consistent code. A normalized form based on

data structures would help to overlay a consistent structure onto the partitioning of instructions. The most important attributes would be to utilize the computer itself to ensure that the code was more abstract and smaller. Software always lacks intelligence — which limits the types of algorithms and heuristics available — but there is plenty of CPU available for determining inconsistencies or for remembering specific settings for programmers. Tying the languages into a new generation of build and management tools based around simple abstract paradigms could result in giving programmers the ability to work with coding blocks at a significantly higher level, thus producing more complex systems, faster and more reliably.

UPDATE:

10. Strongly typed.

With the ability to apply loose typing in the language. From a language perspective all types whether primitive or not should be treated equally. At the lowest levels strong typing helps reduce bugs, but generates more code at the higher levels. Polymorphism is an important technique in code simplification and reduction. The language needs to inherently support it.

40 Rust and Bloat

The software industry changes very rapidly; code that is not under active development will quickly "rust" and become unusable. This happens most frequently when one or more of the underlying dependencies becomes unsupported; the trend in development is towards being more dependent on underlying technologies. Within a decade, an untouched body of code can easily become useless and obsolete.

This aging process pushes the managers and developers towards building very rapid but low quality systems. The constant pressure of keeping the code-base up-to-date outweighs the need for keeping it clean and organized. Intuitively most people focus on the short term goals; patching the code quickly to get it back into working order. This onion approach creates new layers that get slapped over the existing logic to fit it into a slightly revised problem space.

While this works in the short term, it causes a massive increase in complexity. Each new layer fails to encapsulate the underlying details, instead it acts as a multiplier. Little quirks in the code ripple through the implementation requiring increasingly significant effort to maintain. In the long term, this approach is deadly. More and more developers and time are required, the system becomes a sink hole for resources.

The old saying "A stitch in time saves nine" is a fundamental underlying principle of software development. It may not always be possible to get it right because of short-term pressures, but it is absolutely unwise not to revisit the problem as swiftly as possible to fix it properly. Growing a system through well thought out iterations is a fraction of the work required to peruse an onion-based approach. If followed, the extra time savings could be used for such luxuries as better quality or simpler interfaces. Backward compatibility is not a reasonable excuse for letting the problems fester.

41 Quick and Clever

Nothing feels worse than an "almost usable" piece of software. That is, code that is 95% usable, but contains a significant number of fatal flaws. Maybe some clever developer thinks that providing something mostly working is their way of letting us get some use out of their work, but I tend towards thinking its either laziness or arrogance. Laziness because the work just isn't completed. Arrogance because they think we somehow need their broken tool; that we can't live without it. Either way, getting sucked into some tool that is an "almost" is irritating, and reflects poorly on the software developers who spawned it.

To me, the only way to make that worse is for the tool to be an freely available OpenSource product. Given that the developers had all of the time in the world to go to market — there aren't by the very nature of OpenSource any compelling business reasons to rush into a poor release — my expectation is that they, above all else, would care enough to spend that extra time to get it right. Code monkeys being whipped by a soulless mega-corporation just seem pity-able in comparison; they have so little influence in the code's destiny.

For anything "professional", I expect that the standard amount of care and thoughtfulness is added to the work before it is released. That's just the definition of professional. As software developers we are skilled craftsmen, so it is no less embarrassing for us to release flaky code than it is for a rock band to play hopelessly out of tune. The measure of our work is the quality of our code. In the first few years of our career we might be forgiven for letting a few poorly though out lines of code go into production, but as our knowledge and experience progress, the stability of our work should increase as well. That alone, and certainly not the speed in which we code, is what sets apart senior developers from the youngins. Sloppy code usually comes from inexperience.

By now there is enough code available to solve most of the problems out there. The issue isn't creating something new, its building it better than before. We need to stop rushing into half-baked designs and start building full complete tools that truly learn and expand on all of the work that had preceded them. We really don't need more poorly thought-out flaky functionality, we need software that works correctly. That would be a huge improvement over what we have now.

42 Fugly and the Art of Process

There is an aesthetic, or what might also be called a design philosophy for which I have seen many real life examples. The most vivid was a soviet-build SUV; it was incredibly industrial looking; bolts visible; hard square lines; absolutely nothing was done to hide the details or make it look "prettier". It looked like someone deliberately went out of their way to make it as unappealing as possible while making it as functional as possible. This particular machine was painted flat military green and was christened with the slightly humorous but appropriate name: "fugly". That name fit the design to such a tee that whenever I see that functional, but completely unappealing aesthetic, whether it is in architecture, machinery or software, I think back to that machine and that name. There are lots of examples, particularly for architecture in a place like post-communist Berlin. Sputnik, the tower at Alexandar Platts, and several of the communist built vanilla plain apartment buildings stand out as exceptional implementations that share this aesthetic.

Years of experience has taught me that process leaves an indelible mark on its output. You can't divorce the two, one controls the actions in which the other one is fashioned. Experience has also shown that process can run amok. It is often applied to control the work, or just to ensure a minimal level of quality, but left on its own, process generates more process which gradually stifles progress and innovation. Left untreated, the excess grinds everything to a halt. Good processes guide work to completion; bad ones act as sinkholes for energy and resources. Bad ones are a common occurrence in software development, possibly because it is such a young discipline, maybe because the output is not tangible, so its hard to measure the quality.

Commonly, many big software companies rely on tying their bugs directly to code changes. So for instance, the source code repository does not accept updates unless there is a corresponding bug id. The bug must be found and logged first before the code is added. This particular process' intent is towards keeping the working code from getting modified unless necessary; deemed a good thing by management consultants and process-minded management. However, the actual effect of this process is to act as an incentive for coders not to fix the underlying problems in the code. Unless a bug is reported, the code must remain the same, but if a bug is reported, it reflects badly on the programmers. It goes beyond the all too familiar cop-out "If it ain't broke don't fix it", to act as a barrier to developers to correct long standing problems. No bug, no change. Why bother fixing things if it only makes you look like a worse developer. As the underlying software grows, refactoring the existing code to integrate into the project isn't just a "nice to have", it is a important and

necessary task to prevent the code from exploding in complexity. Failing at that, means a code base that is massive, redundant, inconsistent and exceptionally fragile.

Often the common software development processes like to combine multi-disciplinary teams in a silo structure. So that different teams — made up of many diverse occupations such as programmers and testers — deal with different subsystems. Not too sure why, but multi-disciplinary teams seem to foster poor morale. It is an issue that could use more study, but it seems as if the different skill sets get compared to each other to form a hierarchy. That causes people lower down in the rankings to feel that their skills are being underrated, even if their workload is greater. Testers, support staff, writers and programmers need to interact, but they shouldn't compete with each other. Along with the poor morale, siloing teams contributes to redundant work being reduplicated over and over. Often it is code, but also design, infrastructure, automation and testing are repeated as well. Silos are a natural arrangement for a large project; breaking down the overall team into smaller teams based on collections of functionality. Alternative arrangements would likely work better, but the processes are well known and large development efforts don't like to deviate from convention even if they are aware that it decreases their likelihood for success and quality.

Software development is at its infancy, this is shown most clearly by the processes that are frequently used to develop software; particularly in large development shops. The irrationality of many of the processes, particularly if examined carefully, really are unexplainable except to say that people will wear blinders and accept the flaws if they feel the system is somehow protecting them.

Failure in process is very visible in its output, although it may have elements of subjectivity in its measurement. For example, overweighted large scale waterfall processes stunk out initiative and tend towards producing the bare minimum functional output. As the processes grow out of control, they become more important than the output; a clear sign of trouble. People blindly follow the rules of each process, even when it makes no sense, and they willingly choose not to think about what they are actually doing or whether or not it will work. Subjective issues like design and aesthetics get less weight as the process gets worse, primarily because it is too hard to articulate their importance and the process is sapping all of the energy from the available resources. Work tends towards the easy parts. This trend continues until the output has achieved that "fugly" status. The output functions minimally; the process ensures that will happen; but it looks awful and is just barely acceptable. Strangely, those key to the process, who often know it is diminishing their ability to get real work done, are the last ones to admit how "fugly" their output truly is.

43 Evaluating Code

Its useful to have a rating system for software that is simple, easy to score and minimally subjective. My suggestion would be to evaluate three different perspectives of software: design, functionality and completeness. Rating each category from 1 to 10, where 5 is the bare minimum usable code produces a good set of metrics to asses the quality of the product. This is an external measure only, it has no baring on the underlying quality of implementation.

The importance of design in software cannot be overstated, it is equivalent to the presentation in cooking. Even if the food tastes great, a crappy presentation significantly diminishes the meal; great chefs make sure the food looks as good as its tastes. It matters. For software, design is more than just the colors or positioning of the text. It includes the placement of all functionality within the overall navigation and menuing structure of the application. Poorly structured programs were it is hard to find the desired functionality should be scored low. Having to spend lots of time in the help or get a course on how to use the product should also cause low scores. An ugly aesthetic, bad colors or still relying on widgets from the 80s, when its now the 21st century count heavily against the code. If its too painful or annoying to use the software, it should be scored below 5. If it fits like a glove, requires no digging through help or tutorials, and looks so appealing that you actually want to fiddle with it, then a 9 might be a valid score.

Functionality drives the application — it is nothing more than a tool to manipulate data — broken or awkward manipulations impair the ability of the tool to perform its designated function. Weak algorithms, strange bugs, annoying behaviors, and overly complex abstractions all count heavily against the functionality of the software. Low scoring, implies that existing functionality in the software is not up to the being used to perform the major tasks of the tool. High scores come from knowing that the users go to the tool first, because it has all the capabilities needed to get the work done.

Nothing is worse then getting half way through a task, only to find that there isn't a way to complete it. You could add something for example, but there is no way to modify it or delete it. In its own "relative" terms, completeness comes from there being all of the complete functions on a set of data for any given set of functionality. For such an easy idea it is stunning how many common software tools are actually incomplete over their whole range of features. Half implemented ideas are the norm, but they also impair the use of the tool. This score should reflect the amount the usable functionality that is incomplete. If lots of features from the fact sheet are missing parts, then the score should be

low. If the software is well rounded and complete for the all of its functionality then it should get a high score.

From a usability perspective these three attributes describe the basic elements of software in a simple fashion. Averaged, they can provide a single metric to a body of code that allows it to stand along side of others for judgment. A simple scoring measure that is minimally subjective allows for this rating to be easily implemented. If people examine the state of existing software out there they will probably be shocked as to how low so much of our commonly used software will actually score. This rating works well for applications, but it also can be applied to non-visual things like libraries. Design becomes the qualities (consistency, understandability, intuitiveness) of the API, while functionality and consistency stay more or less the same.

44 Software Development

Fundamentally, the problem with software development is that it is not rigorous. Most of the work is driven by instinct and guesswork, there are few theoretical foundations or even well established rules of thumb. What happens in this climate generally is that small groups of people go off in various directions, solving and resolving the same basic problems. Facts, results, and understanding get lost as each new group simply reinvents the same mechanics. Sometimes a bit better, sometimes a bit worse.

Software is essentially a solution to a problem. The variables that make up the requirements define the scope of the development; the implementation should fit within those variables. Systems exist that work, but little seems done to validate why or to build up a foundation. Knowledge gained from these experiences is quickly lost.

Economically, as the need for the industry has grown, the most effective way to meet the needs is to assemble large teams of young, smart, but relatively inexperienced programmers to pound out the maximal amount of code. Quality is unnecessary in absence of any real competition. The first to market wins — the problems could be fixed later (if necessary). The brute force approach dominates the industry, and has become more acceptable and mainstream. Poor quality has yet to be a serious market liability.

Oddly, most real mission critical systems rely on the very old generations of software and hardware; crude and simple, but far more trustworthy than any of the latest technologies. They are difficult to change, which turns out to be a good thing. The newer systems have become so bogged down in unnecessary complexities — mostly caused initially by poor programming and then compounded by bad patches — they function, but not predictably and not for sustained periods. Quality issues in hardware have become significant as well, as that industry adopts lesser standards in order to produce cheaper products.

The problems with software go beyond the development processes. The science seems derailed with little real progress being made on actually trying to move it forward again. There is little innovative research, and very little discussion into the underlying problems. Even most of the industries writings and newer standards and processes aren't even being adopted by the masses. Old habits die hard. Computer Science is moving backwards as it degenerates into smaller and smaller groups of people pursuing their own similar directions. Brute force produces low quality software, but in absence of any other demand that seems to sustain the industry. The industry will stagnate. The science won't make any

progress until the consumers start demanding software that actually works and does what is advertised. Software developers will adopt the basic brute force approach to developing software, even codifying the approach into the process itself, until there is some real pressure to find ulterior methods for building software tools. What irks is that there are so many problems left to solve and so many good uses left for software, but its unlikely that we will see any real progress for quite some time. We are in a period of rewriting the same old broken solutions again and again.

45 Effective Programming

INTRODUCTION

Modern societies build many different types of complex items: cars, computers, houses, skyscrapers, oil tankers, software, etc. Each different medium for building has its own unique problems and challenges. These issues change as the scale of the work grows, so that small projects rarely resemble big ones in process. Experimental work rarely resembles continuous production. The differences are important, but underneath that goal of constructing something new is always the same; the process needs to be repeatable. On top of that, because consumerism drives our motivation, most of what we build is done so to be profitable. The costs of the work must be less than the amount of revenue generated by the work. This added financial constraint pushes us to find better, more consistent and more productive approaches to building. It pushes us to be more effective with our available resources.

Software is still a very young discipline. It differs from other types of building, but more because of perception and bias than because of any real underlying constraints. Intuition, guesswork and luck still form the basis of most software development processes. A staggeringly large number of software development projects fail, even after more than 50 years of experience and learning. The lack of progress can be very frustrating, and it is hard to get to the source of the problems.

Two interesting points are true for most software projects: 1) there are not enough resources to get the work done correctly, and 2) the project never ends. While contradictory, understanding the effects of these two opposing issues can be key to understanding how to optimize resources for effective software development. They represent the root cause for most of the different short verses long trade-offs that must be addressed while developing. Choosing the short-term answer too frequently leads to a mess, which causes an exponential increase in complexity that eventually makes the system unstable. The programmers cannot fix it fast enough to get it to work correctly. Choosing all long-term trade-offs leads to a super-design for the perfect system, that still won't be built in the next 100 years. Generally the project runs out of funding or gets cancelled. The idea process comes from making a series of difficult trade-offs; from balancing the two different objectives.

PROGRAMMING GOALS

The goal for any reasonable development process should be to make the best

overall use of the available resources. Being effective in software development means finding the shortest path such that in the end there is an active running software tool with minimal support requirements implemented in a number of sites. Effectiveness comes from completing the minimal amount of work on a predictable schedule to get the tool finished. It does not come from creating mass amounts of code, or spending huge hours over-testing the functionality of the system. The amount and quality of the documentation is only secondary to the goal of producing software. A working process must be evaluated within the whole context of the software development including testing, deployment and support. If it only works for part of the puzzle, the side effects can be devastating.

A usable software development process needs to be predictable. Developers need to be able to say with reasonable certainty when the work will be completed and ready to go. This is essential in making effective decisions about resources. It is also important for providing confidence in the work. Predictability comes from not only correctly understanding the initial amount of work, but also from preventing or constraining new work, such as scope creep. Uncertainty, which exists within most building disciplines, is hard to guess, but still needs to be accounted for in the scheduling.

The degree of uncertainty in programming is not as significant as most programmers want to believe. It can come from technical problems, missing design elements, or from not fully understanding the total amount of work. Despite never having built something exactly the same, the bulk of the programming in most systems is substantially similar and can be reasonably estimated. There are parts that are widely variable, but that can be handled by adding slack time. The more difficult problem is that programmers work inconsistently, speeding up and slowing down at unexpected moments in the schedule. The morale issues are hard to predict.

A good process should encourage developers to fully utilize all of their efforts. Because there are never enough resources, it is critical that all of the existing work that is going into a project gets fully leveraged. Getting the intended use from any work is important, but being able to get as many extra ‘uses’ out of the work is key. For instance, code to solve a very specific problem is better leveraged if it is generalized and used to solve many similar problems within the same development. The small extra cost of generalizing is offset by the huge extra savings of repeatedly using the same block of code again over and over. As well, any automation that saves time in completing repetitive tasks is hugely beneficial to the project, particularly as time progressing and the savings get larger and larger. The initial investment of time pays ever-increasing dividends. Documentation needs to serve a specific purpose, and must be easily usable. Putting a lot of work into documents that are for “show” is not effective. Doing

that little extra now to allow the work to be leveraged later pays huge dividends as the project progresses.

For some developers, using brute force to explicitly write out all of the individual steps for functionality may seem like the fastest, most direct way to get the code finished. This may help in the short-term, but managing code is expensive, so minimizing the code base is a more effective strategy. Small amounts of poor code can be refactored, but as the code grows, the amount of work to fix it will increase dramatically. Minor changes to a huge code base of badly written code quickly become the primary development problem, eating up the bulk of the resources. Making it even worse, existing behaviors often need to be preserved even if they are wrong or don't make sense. This combination can easily stagnate development. Existing code can become a serious hazard.

Similarly, a popular way to program is by quickly cutting and pasting sections of code, over and over again. Cut-and-paste experts may generate large volumes of code, but it is unstable and problematic to maintain. What seem like big gains in the short-term, quickly turn into organizational nightmares. The problems compound as changes make the difference instances fall out of sequence with each other. Sloppy inconsistent interfaces look unprofessional and detract from the usability. In a well-designed system, the opportunities to cut and paste are minimal, so frequent cutting and pasting point to architectural problems and unnecessary redundant code.

The process should keep the effort from getting too far off course. Short iterations can be very useful when the overall direction of the development is vague. They do come with a hefty price but they help to mitigate the risk of developing useless functionality. For a reasonable project, the iteration size should vary, depending on the business needs and how easily the developers can visualize the upcoming work in the future. If there is a huge queue of pending changes, the iterations should be expanded to optimize the implementation of that work. If the direction is less concrete, the iterations should be shortened so that the updates are more experimental in nature. As well as functionality, iterations should be used to refactor and cleanup code. Starting each new cycle with cleanup work helps to push the development work into becoming easier over time, instead of more difficult. Consistency within the code is as important as consistency within the user interface.

An easy way to optimize virtually any work is to perform it in batches. Doing the same thing over and over again may be boring, but allows for finding optimizations. A good process bundles together the work, but makes sure it is completed in time for any dependencies.

Predictability mostly comes from the process, but because of the number of

external dependences for most modern development projects, the underlying layers provide the most instability. A strong development team is more likely to be delayed because of some external problem, than because of a coding one. That means for some programmers, an external system bug that is hard to diagnose is more likely to delay the release than a problem with the architecture being bad and not workable enough to get the code running. These external problems come into the project as various complexities that need to be understood, tracked and managed.

MEASURING SUCCESS

Developers want continuing success with their releases. Practically, the project never ends, so just getting one version out the door is not really the problem. Eventually it will happen. It is being able to deliver again and again on a continuous basis that is difficult. Over time, the development should get easier as the earlier work is leveraged and the difficult problems are all encapsulated in completed solutions. In practice however, most programmers sacrifice so many long-term issues just to make it to the next release that the project degenerates as time goes on; sometimes to a point where further development is no longer possible or worthwhile. Stealing from the future to patch the current happens often.

A good development process should be the path to achieving success. Most development is not pushing the technological bounds, so the core uncertainties come from organizational or informational problems. If you can get all of the facts together, get them organized correctly, and get the programmers working on the final code, the remaining big problem should only be the amount of time it will take to complete the development. For most processes however, the true problems come from failing to notice or stop exponential growths in new and unknown complexity. The facts never come together, and the costs of that quickly becomes an endless cycle of patching one part of the system while breaking another.

An effective process comes both from optimizing the details, but also from seeing the problems as a whole. Software development goes beyond design and coding, and includes all of the testing, documenting, distributing, operational and support problems that come from trying to actually make use of the software. Writing a bit of code is only solving a small part of the problem. Its not even that hard. A successful project not only develops code, but it also insures that the users want to utilize the tool to complete their work. That means that the right functionality is easily available and stable enough that the users aren't just using the system until something better comes along.

DESIGN IS CRITICAL

A design is a simplified abstract representation of the final object that can be easily manipulated to test its correctness. Blueprints are an example frequently used in the construction industry. They are a simplified 2-D rendering of the work, but exact enough to allow for construction workers to use them directly. The only point of producing a design for software projects is as a resource to make the building of the code simpler and more consistent. The key is that the design exists to validate the work and make the building easier. Anything beyond that is wasted effort. Building takes time and effort, which is lost if the work is incorrect. The code can be changed or refactored, but that costs more effort. Problems can be ignored, but they quickly compound in complexity if they are not dealt with.

A design should exist to prevent wasted effort. The more direct the route for building, the more effective the overall work. The design should be no larger than necessary or that contributes to the wasted effort. Assuming the design is good, its goal is to guide the building to the final product as quickly as possible. Techniques like refactoring are useful, but they represent course changes and as such are expensive.

Any project that is larger than 'small' in software is not malleable. Once going it can not be easily changed. It takes a considerable amount of time to create and maintain it. Course changes are dangerous and expensive. To be effective, the design must be useable by the developers and contain only the relevant information needed to complete the code. Simple detail need not be redundant; the design is an abstract representation so the conventions could account for some of the design elements.

Because the project goes forever, its obviously not possible to know exactly what should be built for the future. That's probably why the project goes on forever, that and the fact that all of the other dependencies are constantly being fixed and altered. Because of this the design will not be correct; there will always be issues in the future that should be handled differently, either because of changes in the technology, the business domain or the architecture. Refactoring is a great skill and needs to be used frequently in development, but it really should be limited to the start of each new iteration. The first thing the developers should do is correct the mistakes from the past. Non-destructive refactorings are best, as they reduce the instabilities from the applying work. Each iteration that starts with a cleanup phase can optimize the next set of work to be based on a stronger foundation.

ORDER IS CRITICAL

In most projects various 'non-technical' people have an impact on what ends up

in the final development. Any established profession should leave the ultimate decision making to a qualified professional. It is their role to take on the understanding and responsibility for making it happen. In software, people often talk about meeting the needs of the “stakeholders” in a way that sounds like their needs supercede the needs of the actual development. They used this to justify their desire for control. That’s crazy because neither management nor the users are experts on software development, therefore any decision they make in regards to what should be done, will not be based on fact. To build software effectively, the order of development is important. If the project is segmented by iterations, then what ends up in each phase is an important development problem. The users may hold the key to understanding what needs to be built, but the developers understand the order. Management may hold the purse strings, no amount of money can fix a project on a death march.

System architects and/or technical leads need to know more than just how to write code. They need to understand how to optimize a development process to make the best use of their available resources. They need to be the experts in developing their specific tool. They need to understand their users and how and why they use the tool. They need to be responsible for the way and order in which the code is implemented, it is part of the scope of their position. If they shy away from their responsibility, they have little cause to blame their failings on management or the users. The order and priority of the development work is primarily a development issue. The main functionality should come first, and the iterations of the software should be complete and should be useful to the users. Implementing software in a ‘technically’ confused order wastes development effort.

HOW VS. WHAT

A lot has been written about the ‘how’ to build systems, but little of it seems to change the actual success rate. The most likely reason for this is that ‘how’ the system is built is still less critical right now that ‘what’ is being built. Most developers employ a significant amount of brute force in their development. That generates fragile code. Developers don’t take the time to fix the small problems. These compound into larger issues within the system. They don’t clean up the work as they go. The problems get worse with each new release. As such most long running projects don’t end up with partially nice or stable code. Over time, the problems only get worse. The mistakes start to weight against the development, slowing it down to almost nothing. The problem comes from ‘what’ is being written and maintained, not from the ‘how’.

Changing ‘what’ is being built can significantly alter the development cycles. What comes from both the underlying architecture of the system, but also the ongoing extension of the system. Generally, programmers think about code in

terms of the way they interact with it: the functionality. Concentrating on functionality leads to heavily redundant architectures that require more code to complete. Mostly because functionality represents a horizontal view of the system, where the underlying manipulated data is frequently copied and repeated. The code ends up in silos, where the data is moved into some set of functionality, operated on and then moved to another set of functionality. The handling code is identical for many different functions. This large amount of duplicate code makes the system more fragile and over time less consistent.

Alternative approaches to structuring code exist. For instance, before object oriented became popular, using abstract data types (ADT) was emerging as the best paradigm for coding. This set of ideas consisted of well-known structures like lists, arrays, trees, stacks, graphs and hash tables. Most university programs still teach ADT courses for good reason; ADTs are expressive enough to be able to represent all known real world data. As such, they represent a foundation on which to build fully functioning systems, and are roots of object-oriented concepts.

The interest in this older perspective comes not from the ADTs themselves, but the motivations behind their existence. Strictly applying data structures to systems means that the code is either from a data structure, some algorithm (or heuristic) that works directly on the data structures or it is code that binds some functionality to the interface. All of the systems code fits into the three categories. While similar to objects, the simplicity of this type of development makes it much easier to debug than a heavily ‘design patterned’ object oriented system. There can be a one-to-one correspondence between a data structure and a “set” of related objects, such as a List or a Map, however ADT implementations are simpler. ADTs optimize the simplicity, readability and debugability of the code, which can be quite a comfort when fixing bugs at 4am.

As an abstraction, seeing programs as a series of data structures, keeps the mechanics of the system within a consistent framework. An abstract view can be mentally manipulated to validate its design. On the other hand, seeing programs as sets of functionality pushes the developer into decomposing the steps into smaller sets of brute force operations. Brute force programming produces orders of magnitude more lines of code than abstract implementations. As such, the simpler brute force viewpoint leads the developers to brute force their implementations, leading to extremely large masses of brittle code.

Object-oriented languages do not protect programmers from creating brute force implementations. In several cases, the language culture itself tends towards pounding out brute force code and then just trying to patch it to make it work. Some object-oriented patterns, such as adaptors are used horribly to

justify patching the code instead of fixing it. Implementations become rapidly increasing ‘onion’ architectures with each new layer badly slapped over the old one to hopefully hide the defects. Just arbitrarily chopping up code and then jamming it into a ‘pattern’ does not make it abstract or even object-oriented.

The key lesson from ADTs is that viewing the program as a series of data structures is a far simpler abstraction than viewing the program as a collection of functionality. In a typical system, there is considerably more work to listing out all of the functionality than there is to listing out the key types of data. Given the data, most of the applied functionality is obvious, so it can be implicit. There are often a couple of very complex algorithms that form the heart of the code. Listing out the data also leads to easily being able to see key optimizations. Data forms a better basis for design ‘blueprints’, because for all software it is consistently less complex than the functionality used to manipulate it. It is also the key to creating reusable components, abstract architectures and to structuring large development teams.

Brute force attracts programmers because it appears to be the fastest most direct path to getting the code completed. It may seem that way, but only in the short-term. Coding is only a small part of system development, and brute force code quickly creates its own make-work.

SUMMARY

Software development can be very frustrating because there has been so little progress in the way software is built. The industry was quick to determine how to profit from software, but that seemed to stagnate any further progress in pursuing better development techniques. The results: software has become increasingly unstable, even while becoming visually more attractive.

Software development is not complex, but the culture of development makes it taboo to curtail the freedoms for developers. Sadly, many developers would like their jobs better if their work were more successful, and giving up some of their freedoms would probably result in an improvement, but its difficult for people to see outside of their current situations or to change their habits.

Lots of different software development processes exist, but none have really taken over as being the correct way to build software, because they have so many intrinsic problems. The only real issue that counts is that most of the resources are used effectively in some predicable manner. Getting over that hurdle changes the probability of success. Conceptually, the best way to visualize the shortest, and most optimal development path is by working backwards from the finished product. It is possible to find the minimal amount of work that is needed to deliver the minimal amount of stuff. Users need a

system to be running, and they need documentation, installs, upgrades and patches. Just providing the minimum, while leveraging as much of the work as possible to make the future easier cuts the standard amount of work in half. Changing the focus to data, from functionality further reduces the amount of code needed for a delivery. Combining all of these elements with a constant eye towards managing the complexity and eliminating uncertainty takes development from being a very complex endeavor to being a rather simple one. It just doesn't have to be that difficult.

46 Some Thoughts on Testing

Realistically bugs should be defined as any behavior encoded into a system that isn't what is commonly expected, whether or not the programmer purposely choose that behavior. Using this wider definition for bugs encompasses more than just the standard crash seen so commonly in software, but also any type of strange or uncomfortable behavior that disrupts the usage of the tool. Bad interfaces, weird messages, poor algorithms, etc. are all seen as 'bugs' to the users, so they should be bugs to the software developers as well.

Bugs can be fixed, they can be worked-around, or they can be ignored and codified into the overall behavior of the system. The best example of the third case is the differentiation between text and binary files. The rumored cause of this was an incompatibility between DOS and an early hard-drive. The characters stored at the end of each line were slightly different on the device. The fix was to fiddle with the characters if the file was text, but not if the file was binary, thus creating the arbitrary difference. That distinction then became embedded into thousands of pieces of software; making it impossible to remove. What started as a bug quickly (and sadly) became part of the specification.

Testing should show up more than just the obvious coding problems. It is a bottomless pit of work, so it needs to be done in a way where it is the most effective. To understand this, we need to look deeper the nature of bugs. While its not absolutely correct, a useful way to model bugs is to start with the assumption that there are a fixed number of bugs that get created with any new piece of code. The number of bugs is somewhat arbitrary, so we could use some guesstimate like 10 bugs for every 200 lines of code. After it's written, code generally gets an intensive testing of some type, often removing less than 50% of the bugs. For our 200 lines, that leaves roughly 5 bugs remaining. Over time, more testing occurs, more bugs are found, as the code runs longer and longer in an production environment. Bugs 6 and 7, and possibly 8 get discovered while running in the early days. In practice, its not uncommon to find subtle bugs in code that has been running in production for decades, so its fairly safe to assume that bugs 9 and 10 get found somewhere in the long term. That behavior sets a pattern whereby roughly 50% of the bugs are found in the short-term, 30% in the middle-term and another 20% wait for the long-term or are never found. Mileage and actually numbers may vary, but the pattern is close enough to make this model useful.

We can draw some interesting conclusions from playing with this simple model:

- no system is entirely bug free.

- some bugs will be ignored.
- initial testing will not find all of the bugs.
- some bugs won't be found until the code is realistically exercised.
- some bugs won't be found until anomalous input is entered.
- some bugs won't ever be found.

To get effective testing we need to reduce the amount of testing down to the bare minimum to insure that the right bugs are being found at the right time. We also need to assume that there are bugs that will get released and there needs to be some support/release mechanism to quickly ensure that their effect is minimized. If we know that some blocks of code have not changed, then unless it's automated, retesting that code is using up valuable resources for no effect. For simple low-level tests we need to make sure that we are not over-doing it, creating more work for ourselves or wasting resources. Some algorithmic sections of the code fit well into automated regression testing, but for most interactive systems automating the bulk of the testing consumes too much time to be considered useful.

Not all 200 lines of code are created equal, so there are sections of the code that require way more attention and focus for testing. Understanding the audience and how they utilize the tool allows the developers to focus on insuring that some paths through the system are less likely to break than others. Testing is a lot of work, so distributing it evenly across all of the code doesn't make sense. Looking at all of the related issues, the strongest type of testing for new code comes from putting it forth to an entirely independent set of testers. Test cases written by the developers are tainted, so only the minimal amount of documentation should be included. If we give the testers the tool, tell them what it should be used for, and have them use it to be productive, we should get a better sense of what defects are really in the code. If they attempt to reach enough 'end-goal-points', and document it properly then we can gain confidence that the main pathways through the system are as clean as possible.

47 Independent Testing

To be effective, testing must squeeze every last ounce of quality from each drop of effort. As well, any quality control check where one person follows in the footsteps of another is inherently weak because both people can easily make the same mistakes. These two points mean that a good testing process should be very focused but also developed directly from first principles. This sounds great, but it can be seen as a contradiction.

To get around this, good testing should come from an entirely independent group of testers. They need to cover the code, but not waste time testing unnecessary parts of the system. True independence means that the testers do not have access to the original design specifications, instead they only know the end-goals for the tool. This might seem to leave a number of holes in the testing, but if this type of end-to-end testing crosses the majority of the pathways throughout the system, the coverage is going to be similar to most development test plans. This may sound over-stated, but most plans focus heavily on easy tests, while dodging the hard ones. Also, testing every permutation of behavior and options is too expensive, and not effective, so any application that is non-trivial is released with 'testing' blind spots. Often huge blind spots. Effectively even if an organization has a large number of tests, and run them often, the quality of the testing is not guaranteed. In fact, for most products it's abysmal.

If massive testing is not effective, and only marginally better than a little testing, then it is not a good use of resources to build large test plans and use the development resources to execute them. A set of completely independent testers, with a goal-oriented approach towards testing can achieve equal or better coverage, but at a fraction of the effort. The key for quality control is 'how' these testers are driven. They need to focus on utilizing the tool, while correctly documenting any deficiencies that slow or restrict their actions. Done well, the results from testing should be a list of issues and problems that need to be addressed, including interfaces, standards, typos, weird functionality, etc. A group that is detail-oriented enough to create this list while utilizing the tool for a wide enough range of goals would be worth more than a pack of thousands of tests.

48 When Ideas become Products

Progress is important. But what might seem like progress, can often turn out to be something very different. With new ideas, people become so invested in them that they avoid critical thinking. Any potential for improvement is quickly lost by this stance. In particular, with software development there has been a long history of programmers attaching themselves to ideas or techniques whose negative qualities outweigh the positive ones, but they refuse to see it. Hungarian notation always comes to mind as the penultimate example, because it was backed by serious people and there were so many people willing to defend it.

Hungarian notation, for those not familiar with it is a 'technique' for naming variables. The idea, was that because it is hard to name variables, using some small mnemonic attached to the variable name that contains general type information — later to become specific physical type information — will make it easier for programmers to create variable names. The problem with the technique was so elementary, that most people missed its significance: there is no problem with naming variables. Not really. Not, if you understand what you are writing and have done the analysis. If that is true, you know the data you are working with, and by virtue of that you know exactly what it is called. If you do not, then you have a design problem not a coding one. The problems with Hungarian notation stem from the fact that they are a solution to a non-existent problem.

People being who they are, that is a hard argument for many of them to accept. But it is a nearly impossible argument for anybody who has committed to Hungarian notation either in time or emotionally. And thus, for years I witnessed countless "it is right" versus "it is wrong" arguments, that ultimately don't convince anyone of anything. All that could really be done was to hunker down and wait until enough time had passed, that most people could let the idea die a natural death. A well-deserved one, I might add.

I often find that sad, because I think we should be able to think our way around this type of circumstance. So much time was lost because those invested wanted the positive aspects to be true, but didn't want to accept that negative. And, they couldn't weigh the relative merits.

So I've been thinking about this more lately, and I am starting to see the root of the problem. If there is a discussion such as:

"I think Coke is better than Pepsi because everybody says Pepsi is too sweet."

Someone will ask "Have you ever tried Pepsi?"

and if the answer is "No" then the inevitable retort is:

"You are wrong, you can't judge Pepsi without ever actually trying it."

The final statement seems more than reasonable, although it may be likely to spark a debate on the merits of being able to judge a soft drink without ever experiencing it personally. Why, without experience could you ever make a judgment call on something that you know nothing about. Except consider the following:

"I think there is a threshold to sweetness in soda pops. An inverted parabola where the the desirability of the drink is on a curve versus the amount of sugar. The tip of the curve is the most desirable amount of sugar. Some colas sit too far to the right on that curve. I only like those colas that are on the left, so I've never tried any of those on the right."

You can't easily say that this statement is wrong, and because of that it prevents polarized arguments. The "have you tried it" question probably even won't come up. Without being specific, it fixes an idea and a pattern, but it is one which many people could reach a similar conclusion to the above discussion. The two examples revolve around the same basis, cola and sweetness, but the second one does not open itself up to the same type of obvious counter-argument. That is something I've seen, but I think most people in trying to explain it would come to the wrong conclusion about the reasons. The obvious draw would be to say that the second statement was more generalized and harder to argue; that it doesn't single out a specific cola. That I heavily suspect misses the real issue.

The key difference, the one that I think so quickly binds this type of issue to a polarized argument, is that in the first case the things discussed are "products", while in the second case the thing discussed is an "idea". A product, is essentially an immutable concept owned by someone, marketed to someone and can contain a great degree of excess baggage. For instance, Coke and Pepsi have been at war with each other my entire life. Politics, lifestyle, and philosophy have all been wound up into my impression of these products. I can't be objective about cola, since I never have been. A product, then isn't just the thing itself, the underlying ideas or anything that simple. And you can't accept half the product, its an all or nothing type of deal.

With an "idea" though, it is a lot easier. It is just something abstract that does not get under most peoples skin in any way. Our attachments to it are purely

intellectual, and so we are willing to look at it objectively or critically. We can play with it, think about it, and discuss it in ways that are impossible with products. An idea is unencumbered by any emotional ties, products start 'religious' wars.

That, I think is where it all starts. But, then as we see the idea, and we buy into the idea, the idea starts to become a product. Once we've bought the books, spent the time to read them, tried it out in reality, etc. We have left our objective space and jumped on board the product train. At that point, even though we have experience, it doesn't help because it may be a long long long time because we can return to looking at the original idea objectivity, if ever. We no longer want to know about the problems with the product.

By product, I don't want you to think that I am only referring to those things directly for sale. I'm using it in a broader way to refer to anything that is essentially packaged and distributed. Way back, I think the endless wars over whether or not the vi editor was better than the emacs editor was yet another example of a polarized product argument. At the time, I don't believe there was revenue at stake, but there certainly was emotional attachment and lots of invested time.

That's what I think happened with Hungarian notation, and that's what I think happens again and again with many of the new things in Computer Science, whether they are technologies or techniques. They start as ideas, and unfortunately before they can progress and grow, then get adopted by people who turn them into products. At that point, they freeze. They become static. People no longer want to improve upon them, they are too busy defending them, twisting them, or trying to explain away the flaws. Objective discussions are the last thing desired. Sadly, where we might have created productive discussions intent on improving our lot, we instead have polarized arguments where everyone simply repeats back the canonical texts. Time it seems, is the only way to break this type of reoccurring deadlock. Once the product mentality starts to seep in, it takes a long time before any real improvements are allowed. Is it any wonder then, that we are improving so slowly?

49 Containing Complexity Growth

I'm willing to admit that it might just be because I am getting older. I am.

Age, it seems makes me less impressed with everything, particularly when I can point back to something — that in memory at least — was a far larger improvement than what exists now. It often feels like we are going backwards.

And so it seems as I am reading documentation on some of the newer technologies that the proverbial 'ball' has been dropped. The failing: an overwhelmingly large need to over-complicate the easy parts of the problem, while ignoring the hard parts. These newer solutions require too much new understanding, while not removing any of the original complexity. They love to make up new terms, yet the factoring of the ideas seems if anything, less. It might just be age that makes me think that, but then again, it might just be experience. It's hard to tell.

Not that the new goals or ideas are wrong. In many ways there have been improvements in the 'direction' that developers are headed. It's just that most of the implementations are so often a complete let down. Particularly when they come from lofty goals, only to so quickly contradict their intent. Do people miss this? Or do we choose not to examine the goals and the technology and compare the two? There is so much potential for software, and so little of it makes it into the implementation.

Lately, I've crossed an increasing number of these technologies that seem to be overly complicated, yet missing core pieces. There are usually co-mingled concepts that get attached to the problem but are nothing more than pure artificial complexity. They are solutions to non-existent problems, or at very least exaggerations of the problem, while at the same time the real issues — the complicated ones that really need solving — are ignored or lightly brushed over. It is easy to solve easy problems, and it is just as easy to make them more complex. Too complex. We build up these abstractions that don't add value to the solution, but instead force us into redefining — once again — basic concepts, with new terminology but not new ideas. Abstraction is great, but it must encapsulate enough of the detail to justify its usage. If not, then it is just contributing the the problem.

There are several things that can be done to reduce the frequency of this type of problem. The first is for developers to add new definitions or concepts only as a last resort. Developers need to seek out something similar, and use verbs to modify their terminology before they start creating brand new concepts. For

example, if something refers to a sub-section of a tree, then 'subtree' would be preferable over some artificial term like 'hive'; 'subsection' would be acceptable as well. Just not a term derived from nature, that is only partially correct (there are no bees in a hive, for instance). Especially if it is only referring to a partition of the original structure and nothing more.

If something acts as a template to be built upon, then 'template' would be a preferable description over using a new term like 'archetype'. Why go forward with a new abstraction, when it is not? There is a perfectly reasonable term that fits, and it is already in common usage. Is there not more confusion from arbitrarily tossing in new terms, then from mixing with old ones? At least with the old, there is a basis point for understanding. A slight variation on a template is still a template.

Creating new terms may be fun, but ultimately it acts as a negative against the work if the concept is only an existing one that has only been recycled. New terms just don't help.

Another thing to be done is to apply a simple test to the idea to help developers decide if it is truly new or not. It should be really simple. For example, the developers, on thinking that they have a potential new concept, should sit down and write out a description of it. I believe that as they sit and write, the words they use will help them focus on explaining their ideas. If they then discuss their ideas with other developers, and write some more they will find that as they narrow down on a better explanation, the terminology will converge onto a smaller and smaller number of base concepts. When the explanation is simple and straightforward, and easy to say, it is likely to be the most elegant version. If and only if they were unable to compact the explanation after many hard fought cycles, then they could consider creating a new definition. But that needs to be the very last alternative.

In general, discussing the solution as well as the problems is an excellent way to work on any development project. To discuss something, you have to have a better understanding of it. The more you sit and explain it to technical or non-technical people, the more you are forced to understand it. But in doing so, you will also find that your explanations get simpler. As well, the explanations map directly to the design. The verbs and nouns of the explanation should mirror the methods and data in the solution. The sentences should mirror the architecture. If you follow what you say, your design will improve. Playing with your descriptions is also playing with your design.

In the same way that a line of ants gradually improves and optimizes its scented path as each new ant follows it — slightly cutting around the curves — the more often you talk about your design, the simpler it will become. That's the

key to finding and removing unnecessary complexity.

Another approach that is similar, is to write the user manual before writing the software. You'll tend to want to cut corners in the manual and head for a smaller simpler explanation of the ideas — just to save time while writing — which again tends towards a more elegant description.

By repeatedly talking and writing about our ideas, we tend to come down towards better, cleaner, simpler explanations.

At the very least, software developers should always be conscious of the way a specific function or feature will play out in the user documentation. That little simple trick helps focus on creating a tighter piece of code, and makes sure that the functionality doesn't confuse the users. Super sophisticated functionality that isn't usable doesn't make for a good or lasting tool.

Computers are fascinating machines with an almost unlimited capacity to automate intellectual tasks. Like a bulldozer is to a mountain of dirt, a computer can leverage its user's abilities to perform greater and greater tasks. But, there is always a possibility for all of that potential functionality to get locked away; buried under a mass of unmanageable, unusable complexity. Programmers naturally want to add little complicated bits. It takes discipline not to. Or at very least, an understanding that if the system is too hard for people to extend and to use, then it can never fulfill its purpose. As I grow old, I keep hoping that the things we'll build will improve upon the existing. But if we allow ourselves to over complicate our problems, it will be way past my lifetime before we are able to distill that into something elegant. It will also be way past my lifetime before we are able to distill that into something dependable.

I could be wrong. I admit it. But from the way my computer's been working lately I'd say that is rather unlikely...

50 Weaker than the Whole

We developers like to speak about software in terms of 'products', 'applications' and 'systems'. But our users see software installation as adding some new 'functionality' to their machine. There is a serious disconnect here. The problem is in the way we factor this division, which is a by-product of history. Because of this each 'product' implements its own poor installation, upgrade process and resource management; often quite badly. While they may install (barely), few programs can even actually delete themselves properly. There is a huge number of bugs in this type of software management functionality.

For some reason, we see this as being part of the application sphere. Inside of that we add sub-components such as plugins, add-ons, etc. Realistically, that doesn't make sense, as all of these things are really just extensions to an extension of the operating system. It should be the OS that provides a mechanism for the various collections of functionality to be placed together. It should track them, and their relationships. The OS is the 'root' program, responsible for managing the resources, which to me implies any other sub dependent program and their resources.

Our current approach of constantly reinventing the wheel for adding new pieces not only wastes a tremendous amount of time, it also doesn't work. Developers, administrators and users all lost vast hours of their lives to tracking down avoidable installation problems. We have at our disposal this incredibly powerful tool to manage data, but it can't manage the programs on disk properly. How embarrassing is that?

This has to change, and I certainly know what I want. I want one consistent OS dependent way to install functionality. I want one way to delete it. I want to be able to list what I install, and I want a checklist of any sub-components that are included. There should be no way to hide this. If there is a resource collision (of any type), I want to know about it, and have to override it to continue. If I select something for delete, I want to know all of the sub-pieces that will be deleted, and I want to be warned about any dependences. I want to be able to disable pieces, if there is problem with the overall combination. I want to be able to turn on automatic updates and see a queue of changes waiting to happen, waiting for my permission. If it is my personal machine, I want control the updates, if the machine is centrally administered I want the administrator to control it all (once) for the entire site. If I am admin, I want to be sure that the same installation really does exist on all of the different machines. I want it to be easy; I want it to be simple. I don't want to have to think about it, that's what the computer is there to do for me.

I think I should be able to run a query on the box which lists all of the files that belong to any of the installations, and all of those that don't. I only want to backup the user's data, not my installed packages. I also think I should be able to run a query that tells me what changed, and why. If I need to disable any given set of functionality, it should be easy to do. If I delete it, it should go away. If someone else packages up a collection of functionality, I should be able to work with it as a collection. If I install a browser and some extra pieces for example, I should be able to declare that as a collection and easily send it to other people. Working with higher level collections should be easy. The OS should just be able to move these things around in the same way that it moves around files.

I honestly don't think I am asking for too much here. The point of having a powerful tool for data manipulation is to actually use it to perform data manipulations. We shouldn't be lost in notion that some data is different from other data. Our tools can be generalized, but only if we can see that. With all of these poor home-grown auto-update facilities, the machines have grown unstable. That's unnecessary.

There are some tools out there for specific OS versions or specific languages, but they've been build on top of the OS, not into the OS. Allowing another set of paths around an issue doesn't solve it. It just bumps up the complexity with yet another partial solution to the problem. An unfortunate approach that is all too common in software. Until we change our perspective, we'll keep building software that is defective. And we'll keep spending time trying to track things that the computer could more accurately track for us.

51 The Five Pillars of Software

As we expand our software construction knowledge, we continue to add new technologies and techniques. Some of these push our boundaries of understanding, but there are always aspects of development which remain fixed in stone. Constant and unchanging. These immutable pillars of the development process have been remarkably static with each new technology or technique that has ultimately proven to be of value. They do not change. Forgetting or ignoring them always causes difficulties and aside from people problems, is probably responsible for the largest number of software project failures.

All new technologies come with inherent weaknesses. If these are not understood they can cause significant problems in their use in development. No technology is perfect, and more often than not, most of our technologies are just barely functional. Relying on any new technology to solve our problems, without first prototyping their weaknesses, is always a risky endeavor. There are always limits. Often with the new technologies come new improved solutions to the problems, but more often comes the forgetting of the older solutions. We seem to go one step forward and then one step back quite frequently.

New techniques hold the most promise for reforming Computer Science. If we change our behavior we should be able to develop more effectively. But often newer techniques are based more on wishful thinking and less on the cold hard understandings that come from experience. Techniques that cannot justify their expenditure of resources are unlikely to remain of value for long. Limitations will always bound software development; resources (people and time) being one of the most sever. The difference between experience and failure comes from deploying the right resources at the right time to do the right work. How many projects fail with people saying 'if we'd only ...'.

Finding that core-of-truth that forms the foundation of software development is as important as expanding our knowledge to add in new effective techniques. When you factor in the base, distinguishing between new technologies and techniques that are practical and those that are wishful thinking becomes easier. If we had to try every permutation, clearly it will be quite sometime before we've managed to find effective development processes. If the pillars are understood and correct, we don't have to spend significant time and energy to determine if something new ultimately adds or subtracts value from the overall process. We can just test to see if it violates one of the foundations.

1. Less

Even the simplest software tool is a huge amount of work. So, wasting scarce resources should be considered a very serious problem in development. Time is unforgiving, and you get only one chance to use it properly or else it's gone. The final project deliverable is usually a fixed set of software tools and those few things that directly support the tools (help, tutorials, etc). If we work backwards to determine the exact minimum amount of work that was necessary to just complete the project with the minimum amount of work, we can often determine what was truly useful and what was not.

Analyzing the wastage leads to a very simple understanding that developers must strive to produce 'less' of everything. Less code, less documentation, less help files, less config files, etc. That doesn't mean not building tools, just that building more of them won't help. It's not the quantity of work you put in, it's the quality of the of work you put in that matters. In pushing for less, we hopefully come closer to meeting the minimal amount of work needed to actually solve our problems; allowing us to solve them more effectively. Less isn't always obvious, for instance less code doesn't necessary mean less functionality but instead it can be more generalize code. Less design doesn't mean no design, instead it means that the design should be as simple as possible (without extra or redundant elements) while still managing to contain all of the relevant information needed to build the software. Less is not an excuse to cut corners. Nor does it mean 'less thinking'; one of the few 'lesses' that are clearly not desirable.

For every decision made while developing, and for every bit of work to be done, it is critical for developers to be thinking 'how can I do *less* of this?', not more. That's the right perspective that leads to finding an optimal solution. Stand back from the abyss and ask yourself 'do I really need this or am I just wasting time?'

2. Leverage

The computer is a powerful tool. To get the most out of it, we need to consider carefully how to best utilize it. There is never enough time to 'do it right', so any work that is done should be leveraged as much as possible. Whatever code, configuration or documentation that gets created should be used again and again to solve related problems. If you can do it once and use it ten times, your on the road to a very successful project.

It stands to reason that anything that is time consuming or expensive should be leverage to its maximum. It is an easy type of optimization. Often the difference between solving one problem and a broader set of similar problems is just a small amount of additional work. From a short-term perspective this may seem like 'more', but when leveraged, the balance is tipped and what was extra work,

now amounts to a savings, and often a considerable one.

Getting the most out of what you've got is the key. But that also means leveraging the power of the computer itself. Automation is the key to repeatability, which is tied very closely to quality. Automating the entire release process for instance, makes it easy to test and to assure that it will work correctly when the time comes.

3. Consistency

Inconsistencies create their own special complexity. Where they are not absolutely necessary, the resulting complexity is entirely artificial and could have been avoided. This is true in the way the interface appears and works, in the way the documentation is written, and in the style and formatting of the code. All of these little inconsistencies work together to ramp up the complexity and make the project more of a mess than necessary. Making it more likely to fail.

Users hate inconsistent interfaces, even if they don't directly put their finger on why the interface annoys them. Programmers hate working on messy code. Nobody properly reads redundant or sloppy writing. The work done to produce a mess is never really properly leveraged. Skipping it probably would have been more cost effective. Time spent in cleaning up, is always time saved.

4. Planning

If you wouldn't start building a house without a plan, why would you try to do so with software? Even if you have to alter the plan as you go, it provides the basis for making good decisions. Not having a plan is extremely risky. As well, a long-term plan always helps in the short-term to better optimize the overall choices. A thorough plan should provide direction for all phases, including scheduling, architecture, design, distribution, shipping, etc. Any work that ultimately needs to get done.

Attempting any non-trivial work without some type of well thought-out plan is futile. Over-planning wastes time, but not having a plan at all is generally a lot more destructive. Nothing is worse than having to start over from scratch again. Returning to zero. Maximizing the existence of the plan comes from being sure that there is enough information in the plan to get the job done, but not enough to make the plan overly confusing. Threading the needle for the perfect plan is the job of methodology, but its worth nothing that the size of the project directly influences the required detail level of the perfect plan.

5. Feedback

Software developers can get too excited. There is a tendency for them to race off and develop a set of tools without thoroughly understanding the problem domain. This is inevitable, but it is made worse by developers who refuse to believe that their first instincts could be wrong. We all make mistakes and developers need to admit that they are better at it than most people.

Software tools should be built for people to use them. Climbing high into an ivory tower only works if you come down from time to time to see what is happening in the real world. Some developers get frustrated with their users when they don't understand the functionality, but that's a bad approach; for most users their failure to understand really high-lites problems within the system, not their own intellect. If it is too hard to easily explain, or people keep messing up the steps or terminology, these are signs of problems that should be noticed by the developers. Important notices. If the users doesn't read the documentation, could it be that it is just badly written? If they mess up using particular features, it could be that they are confusing or even just fugly? You have to let go of your ego, assumptions and prejudices in order to allow yourself to accept the problems. There is always feedback if you are willing to listen. Only when you understand, can you fix the problems.

Summary

Some things are just independent of the technology or techniques. Wasting time is always wasting time. Doing it ten times is redundant. A big mess is always a big mess. Doing it over again and again is just repeating the same mistakes, and building something that nobody wants is just not helpful. These things do not change; will not change; and no upcoming technology or technique can alter that. In our search for new ideas we will grasp onto lots of great and poor concepts. Judging them rationally, based on a consistent foundation sets things into perspective. People will choose to promote things into which they are invested, and to move forward we need to see beyond the claims and open our eyes the what is really there in the most honest fashion possible. A technology or technique that violates any one or more of these pillars will not stand the test of time; it will not survive for long, no matter how vocal the chorus.

Beyond their influence in helping us expand our skills, understanding these basic pillars helps us to make more effective choices when developing. If you are focused on producing less for example, that drives you to try to incorporate more abstraction within your design. Knowing that planning is necessary, makes it easier to try and get enough of the broad stokes down into graphics and text, while not worrying about the specific format of that information. Capturing the ideas is more important. Keeping the interface consistent often removes most of the variability of the screen appearance making it faster to implement. These five principles live deeply in all of the work that needs to be completed in

successful software development, and it is highly unlikely that their importance will ever diminish.

52 Complexity Undertow

I'm having trouble giving it a name. Certainly I've run into it many times, we all have. As the project grows, the ever increasing complexity starts to mount up. At some point, it becomes its own significant problem, a sort of complexity "undertow" that is moving in the opposite direction of the development current.

It becomes harder and harder to make changes; fix mistakes; right the initial wrongs. Each new change becomes a bigger issue, taking longer to implement. At first it's barely noticeable. Just a little delay here, and there. But as the months go by, then years, the problems just build up into this strong opposing current.

By now I've been on enough long-term projects to see this coming. After a while you start to realize that it is the small things that add up. The big things you often can't do much about. They just are. But the small things are often left to the direct control of the developers. Consistency, for example, in the code, the documentation, the interface, etc. is one of the many ways we have to avoid complexity. Keeping the code clean, neat and rust free won't stop the undertow, but it puts off its effects for as long as possible. It is always work worth pursuing.

Another example is not fixing problems when you first spot them. It is tempting to turn a blind eye, particularly at crunch time. We all do it. But many developers continue to ignore the problems long after the pressure has subsided. They willingly avoid the issue. One thing that is always true about development is that the longer you leave a problem to fester, the worse it becomes. Thus, once you are aware of it — not only will the problem not go away — it will just continue to get worse. The time to fix it was last week, but now is as good a time as any.

Eventually, there comes a time when the size, scope and backwards compatibility slow down development, but it doesn't have to choke off any new work. It doesn't have to make it impossible to add new features or to fix the existing ones. All we need do is be aware of, and avoid as much artificial complexity as possible. And contain, isolate and encapsulate into individual pieces any of the existing and similar complexities. If we manage complexity in the same way we manage the other resources in the project, it becomes a lot less likely that it will force development to halt (also known as painting oneself into a corner). The currents opposing development always exist, but if we understand what cause them we can minimize their effect.

Mini-thought: The farther away you can sit from the manual, the better the technology.

53 Why it all Matters

I suppose you should forgive a few flaws. Sometimes, particularly with high technology we need to show some patience. Sometimes not. The general assumption with software flaws is that they come from some part of the process where the developers missed noticing that there was a problem. A large number of inconsistencies or bugs then is an indication that their development process is not working. A good well-rounded process wouldn't perpetuate the problems. It would fix them rapidly, and by definition should prevent many new ones from appearing. After all, that is the whole point of having up'ed the complexity and work by applying a process to development in the first place. The process must pay for itself by helping to ensure that the quality of the results is better, either faster or better quality, but hopefully better quality. Without that, following the process is just a pointless waste of energy.

So, if you were to look closely at the output of a development project, and in that you were to see that there was a consistently high number of bugs not being caught, and continuing to exist, then you can infer that the process is absolutely under-performing. More importantly, if there is essentially nothing to 'tweak' in the process to change the results, then you can be sure that the problem lies with the process itself. The process is ineffective, or perhaps even useless.

At this point, there would be many that could easily point to the people in the project, and blame them. "It is not the process, it is just that specific team" might be the defense. However, if the people — no matter how lazy and undereducated — follow the process, according to the rules of the process itself, then while the people may not be ones you would like to employ, the fault still lies with the process. A good process must ensure some type of minimum result, no matter who is involved. If they do not follow the process, then you can be less sure of the cause of the problems. But, mostly the process should have some way of validating itself, so developers continuing to disregard the process can in many ways be attributed as a problem to the process.

There is a great deal of information to be learned from a web site. If you use the site frequently, you get a real sense of its underlying flaws. How stable is it? Are there lots of weird links? Are the functions consistent? Do people ask a lot of different strange support questions? You can easily tell whether the foundations are stable, or if they have severe problems. You do need to differentiate between coding and operational problems. Is the code weird, but the administrators are working around it. Do the bugs halt the system or are they just inconvenient. When tied to an organization, you can also find out about the

underlying development philosophies, particularly if you look at the company want ads or job descriptions. Often programmer job ads are very specific about the processes used for development. You can determine most of the technologies and any of the major techniques used for building their systems.

Mixing these two pieces of information can give you an interesting example system for a given methodology. What you can't tell is if the developers followed — correctly or not — the process, but you do end up with some sense of whether or not the development itself has problems. Beyond the base functionality, what becomes important is that sense of how well things are going over a period of time. Has the development been getting better, or worse? Do the old problems ever get fixed? Are the developers overwhelmed by adding functionality or have they kept pace with the technological challenges?

When we sense that a site is degenerating, we must come to understand that there is a mass amount of code involved. Over time that builds up and gets worse, it is inevitable. That momentum should be considered, but again the real issue is whether or not the development process is working. A relatively slow development, with a large number of reoccurring annoying problems and inconsistencies tends toward showing that the development process is severely broken and needs to be fixed.

We can forgive an off-day, or a few flaws, but I find it hard to be patience with an organization that sticks to a bad process that is repeatedly causing problems. I also find it hard to be patience with people that turn away from their own examples of a bad process. Things work, so you leave them, but if not you should choose to fix the problem. Not fixing defects with anything other than spin is a flawed approach. Sticking to a visibly broken process in light of enough public evidence that it is not working is tad amount to burying one's head in the sand to avoid any dangers.

What makes software so frustrating, is the sheer amount of publicly available proof that our development processes are dysfunctional. The web is the perfect example. The bigger the web gets, the more ASPs that come to market with big flaws in their systems. Yes, we do learn to live with the flaws, and even manage to work effectively around them, but in so many examples the flaws themselves just are not an acceptable part of the solution. They don't need to exist, and if the underlying process weren't broken, not only would then not exist, but the cost of building the software would be less as well. But, people being who they are, we so often choose to stick with the familiar process even when the results aren't impressive, or even obviously bad. So now — lucky us — we are surrounded by lots of interesting applications, but trusting them to work correctly is a problem. And, investing lots of time into them is wasted, as the likelihood that that effort is supported, even 4 years from now is slim at best.

The technology that we built to leverage our abilities instead burns through our resources. Those small software flaws are the symptoms of much larger problems.

54 Process Blues

A process is a set of rules and a sequence of steps used to guide some work to a final conclusion. It is not, by its very nature, necessarily good or bad; that distinction depends on the specifics of the process. Either explicitly or not, a process has one or more goals that are supposed to be achieved by following the steps and rules. Presumably, the goals of a process are positive. Things that in some way make sure the conclusion of the process is better than it might have been if the process were not used. For instance the process might be trying to organize the work and insure that it meets a certain level of quality. Or it may try to speed up the work. A process could also have a number of other positive effects that were not intentionally stated, but come as a side-effect for free. A process for quality, might also make the workers feel more secure about their output for instance. This reduces anxiety, and by doing so likely increases the effort put in by the workers.

Along with positive side effects come their opposite. Depending on the process, there can be a number of negative things that come directly from the way a process works. One example could be a process that is so onerous, that it literally sucks the will to get work done from its workers. This is a well known negative side effect associated with an overzealous bureaucracy for example. Things crawl along because there is no other alternative. Morale is dismal and people are cranky. Another example is a process that it is so strict that it forces the workers to find ways to avoid it or find a way around it whenever possible. This type of side effect usually comes with the excuse that it is somehow increasing quality, when in fact its true nature is to penalize official effort. If you can't get the work done within the process, then you either find a way around it, or more likely you just don't do it. Either circumstance is undesirable.

Interestingly enough there are also many examples of negative side effects in processes where the desire is to cross-reference different pieces of information to make sure that all of the work really gets completed. In this type of step there is a dependency on how the initial list of work gets created. The process acts as a multiplier on any work initially submitted. Everything submitted just causes more and more work. The effect is to make it extremely desirable to minimize the amount of initial work that is actually known and covered by the process. Again providing a very strong incentive to go around the process or not do the work in the first place; either way it is causing a huge negative result.

The effects of applying a process are always visible, so that when negative side effects do occur, everyone using the process is eventually aware. Certainly, one aspect of this, is that many people choose to voluntarily wear blinders so as to

ignore the negative aspects. A bad process in general causes severe morale problems making it harder and harder to want to participate. If you don't want to leave, then pretending that you don't see the problems is a common option.

Given that processes are just sets of rules and sequences of steps, they are always open to being changed at any point. Adapting the process to fix its glaring flaws is usually the initial direction that most people take when they first implement a new process. Counter to this, over time, the rules in the process multiply and morph into some else, such that the process becomes less and less positive. The negative aspects increasingly cancel out the positive ones. As well, it is easy for well meaning people to enhance the negative aspects accidentally without fully understanding the consequences of their changes. Overall, a process is never really stable, and there should always be some ongoing work to analyze and improve it. Over time all processes degenerate.

Recently I have had the chance to closely observe two very different software development processes at work. One is on the far right — a heavyweight process concerned about producing quality, and the other is on the far left — a lightweight process concerned about getting functionality out quickly to the clients. Retrospectively, both processes are extreme examples in their specific categories. Interestingly enough, both of them are broken due to excessive negative side effects; although it is very different problems they are similar in many ways.

The heavyweight process was intended to insure a very high quality output, but because it is so inflexible and rigid, examples of it working have shown that the actual quality of the output is abysmally low. Barely passable, and often not. The key problem is that the process is so strict and rigid, that there is a huge incentive to avoid it. The process then doesn't insure good work, but it does act to throttle any existing work and insure that any cleanup work that should have been done is definitely "not" done. Software gets messy and rusts over time, so cleanup isn't an optional part of the process it is mandatory. This particular process is a variation on many of the large heavyweight processes that are common in the larger development shops in the software industry. Generally they all suffer from serious defects because they reward counter-productive behavior; while fixing things and cleaning up poor work are highly penalized. This process in particular is exceptionally strict and is far worse than a normal bad example. Because of this, it is easy to observe how that increased rigidity is the cause of the marginal quality. It doesn't need an objective study when the process is followed so diligently but the output is so obviously substandard.

The lightweight process does not attach enough significance to finding and correcting problems before releases, so consequently the quality of the output is low. While it is lower than most of its heavyweight cousins, it is surprisingly not

that much lower than a marginal result, showing that intensive testing follows the law of diminishing returns. No testing is not that much worse than lots of strict, yet poor testing. The point of the process was to get out code quickly, but the low quality of the releases cause enough serious support problems that quickly eat up the resources. Ultimately the effect from this is that the functionality is not being released in a timely manner. In fact the delay between changes is nearly as long as a heavyweight process, mostly owing to the necessity to wait between releases for the furor from the last release to die down. Ultimately then, while it certainly is less work, and less rigorous, this example of a lightweight process is no better or worse than the heavyweight one, with the exception of allowing more bugs to be released.

Regardless of what people say, process is always within our scope to modify and fix. It is the people who use and rely on the process who should be interested in trying to fix it. If everyone pulls a 'not my problem', then the process gets locked in by external parties. If they pull together, then while it may be slow in some organizations, pressure can be brought to bear that the only answer is to correct the obvious problems. It may take a while, but any process can be changed, and every process can be fixed. There is no magic to it, but you do need to be realistic about the side-effects in order to properly evaluate their contributions. The goal, of course is a good working process that does what it intends and it does so with a minimal amount of negative side effects. For software, at either end of the extreme, left or right, the processes quickly run into trouble and fail to deliver. We have enough evidence by now that shows that my above observations are quite common in the software industry. There are always trade offs to be made, but that is not the same as ignoring the negative side effects. A balanced process — somewhere between light and heavy weight — is likely the only one that is well rounded enough to deliver on its promises. But irregardless of the starting point, if they are not carefully maintained all processes degenerate into ineffective swamps. That much is beyond certain.

55 Finally

That's it. I'm moving to blogger with a new blog called "The Programmer's Paradox". This next time around, I'd like to keep the writing a bit lighter. Getting the ideas out is important, but with all of the millions of blogs going on right now, I'm afraid that any good ideas that I might some day come up with get lost in obscurity and forgotten. The only way around that is to package the ideas in a way that is interesting enough to get and keep people reading. Sound easy enough. Wish me luck.

