Notes on Collaborative Software Development

Ryan Renn

There are a number of reasons that software development (henceforth "development") in group environments is significantly different, and should be analyzed significantly differently than, individual software development. In this article, I hope, among other things, to (1) explain why we need to be thinking about beyond the obvious, sometimes using counterintuitive reasoning, and (2) analyze a number basic techniques of collaborative development. Some of these particular practices relate to the debates over "Extreme Programming" and are geared towards gauging its appropriateness. Along the way, this discussion will delve into some interpersonal dynamics in collaborative development situations, which certainly would not have to be confronted in individual software development contexts.

Before I begin, I hope to put this in a historical perspective. Software development is a relatively new discipline. As an infant discipline, it grows rapidly, in terms of both the relevant pure theory, and in its applications to other disciplines. On the one hand, the novelty of the discipline provides a generous playground for promulgators of hype and computing quacks. On the other hand, there is often room for genuine pioneering beyond the status quo. It should not surprise us if some of the analyses we encounter in this discipline are conveniently geared towards prominent and common practical application of software engineering, yet perhaps overlook several relevant factors that should be evaluated when tailoring the application of development principles to uncommon situations.

I begin by shortly bringing up some basic terminology that will be used throughout the article, and then introduce the discussion by giving some examples of commonly stated development-related fallacies, all of which is geared to demonstrate the type of misleading thinking common to those involved in software development which I hope to sweep up in my discussions.

Rules in Software Development

In discussing software development, I would like to briefly mention the role that rules play in it. First, let us lay down some basic properties of different kinds of rules:
(1) Rule R is a *good rule* for agent A if it is usually (i.e., in most contexts or cases expected to be encountered by A) better for agent A to obey rule R than to violate R;
(2) R is a good *rule of thumb* (or, a good *soft rule*) if it is often helpful for A to obey R, but ordinary judgment can easily decide in what cases A should violate the rule;
(3) R is a good *hard rule* if the cases in which R should be violated by A are especially difficult to discover.
To give a couple examples: in the context of operating a specific business in which corporate profits and employee earnings are known to be desirable, a *hard rule* of development would be, "If profits are good, then, all else remaining equal, software

development should be done so as to maximize profits." An example of a *soft rule* might be, "Employees should practice XP (Extreme Programming)."

Unfortunately, many software developers hold development philosophies that are either too rigid or too lax. Soft rules (e.g., "Assembly code is faster than Java, so developers should write in Assembly when optimization is critical") are often treated excessively rigidly. Similarly, many important hard, near-axiomatic rules (e.g., "A business should not write 100% bug-free code instead of 95% bug-free code if time is better spent developing 95% bug-free code,") are ignored. Typical software development misconceptions and fallacies will be treated below.

Selected Software Development Fallacies

Common Types of Development Fallacies

1. *False monism:* Oversimplifying all software development problems or solutions to a single problem or solution. Example: "Everyone needs Linux."
2. *Bad metric:* Measuring something the wrong way. Example: "This code is harder to understand than it used to be: it is longer than it used to be."
3. *Solving the wrong problem:* Solving a problem other than the one that should be solved. Example: "Forget the bottom line – I'm a professional, and this code isn't optimal yet."

A Small Catalogue of Development Fallacies

*Fallacy: "*Companies should not release code that they know has bugs."
Type: False monism / solving the wrong problem.
Explanation: Writing code that is mostly bug-free is often cheaper than writing code that is entirely bug-free. Writing 100% secure code may be a laudable if neurotic goal in the university, but in businesses, the bottom line does and generally should supercede almost everything else. (Many young, idealistic developers overlook the fact that money is a necessary and positive aspect of society reaching many healthy goals.)

Fallacy: "That doesn't simplify your code. It just makes it longer."
Type: Bad metric
Explanation: There is an element of truth to the fallacy: all else equal, code that is more concise is usually easier to understand. But imagine the implications of taking this fallacy to its extreme: we could then "simplify" all of our code by putting leaving source files in a zip file. When measuring the complexity of software, there are better metrics than code length. *Readability* is better way to gauge how simple code is to modify, debug, develop, or understand.

This fallacy is often an objection raised when one is modularizing code: modular code, with function headings or object definitions, is sometimes longer (at least at first) than non-modular code. When replacing a long blurb of code with a descriptive method call or an object definition, one shortens the piece of code where the method body is modified, but may lengthen the code overall. That said, such a change may nonetheless make the code much more readable and better-organized.

Fallacy: "Information should be free to everyone!" (taken from the movie *Antitrust*)
Type: False monism
Explanation: The idea that only open source code benefits the world is essentially just one branch of the naive belief that "only volunteer work benefits the world" and that no one should be paid money for anything (specifically, information). While open source code does benefit the world in some ways, so does paid work. Intellectual property rights are foundational to capitalism and technological innovation. It may sound like a conscientious thing to say, but the idea of banning money-for-software transactions is not harmless and is certainly opposed to capitalism and the U.S. Constitution.

Fallacy: "That code is slower than it could be; therefore, it is worse than it could be."
Type: Solving the wrong problem
Explanation: Companies are only interested in speed of execution when it impacts the bottom line. Sometimes program speed is important (e.g., in high-end video games), and sometimes it is much less important than development time. There would no logic involved in paying someone $30 per hour for what is effectively three extra hours of work, to save their office one second of time per execution on a task that will be executed about five times in its life cycle. The most heavily optimized solution is not always the best solution.

Fallacy: "Two people programming at one computer always produce code more slowly than two people writing separate code on separate computers."
Type: Bad metric
Explanation: Could twelve NBA teams consisting of one player each combine for more wins than one NBA team consisting of twelve players? Clearly not. While there are good reasons that two workers working two separate tasks often achieve more than two workers working on a single task, there is no *a priori* truth to the assumption that groups always work faster separately than in collaboration. In reality, separateness of programmers is not the best measure of development time. However, it is true that two programmers on a single task must program at least twice as fast as their average individual programming rate to match the speed of coding separately, so teaming up had better have some significant advantages to be worthy of consideration.

Fallacy: "Development using object-oriented code costs more to develop than non-object-oriented code, because it takes more skill to be able to develop object-oriented code."
Type: Bad metric
Explanation: This measures skill level required to modify, use, or maintain code (part of the development process) by the skill required to write the code. If that were true, Java applets would be as hard to write as Java Virtual Machines, HTML web pages would be as hard to write as HTML editors, and on a more abstract level, writing a rock song would be as hard as inventing rock music. Object-oriented design is perhaps more difficult or rare of a talent, but programming using objects is not so rare – it is virtually performed unknowingly by any novice Visual Basic programmer, for instance.

Fallacy: "You can better optimize code with a lower-level language than you can with a higher-level language."
Type: Bad metric
Explanation: This is not always true, because there are often development-time constraints. Someone writing in a higher-level language may spend less time debugging and developing, and thus be able to devote more time to finding suboptimal code and selecting or refining algorithms. Of course, lacking this time constraint, we could write a 100% optimal program by directly reprogramming the CPU's circuitry.

The intention of this miniature catalogue of software development fallacies was mostly to establish the following point: thinking rationally about software development sometimes requires a counterintuitive approach. In general, the best software development arguments will follow from reliable, nearly-axiomatic hard rules, and not from soft rules. Yielding to rules of thumb is fine when little is at stake, but for a major decision, time spent in analysis can be easily dwarfed by that decision's consequences.

Elements of Development-Related Costs

I say "development related costs" rather than simply "development costs" because some costs that occur as a result of development decisions do not happen during the development process, or may not be accounted for explicitly as development costs. A prime example of such a cost would be the lost revenue due to clients' eroded confidence in a product released with bugs.

What follows is an enumeration of various costs associated with development. (The items on the list are not intended to partition development costs, but to organize the analysis of its costs.)

1. *Complexity and size of project:* Obviously, this influences the quality and quantity of labor-hours to be used in development, but more importantly, all else equal, this may also affect the number of bugs that will arise during the project. Analyzing the cost of developing a small, simple, hard-to-mess-up piece of software (e.g., a web page that lists a person's hobbies using plain HTML) is a much different ballgame than analyzing the cost of developing a highly complex software (e.g., software to control the flight of a 737).
2. *Bug release costs:* Sometimes the consequences of releasing a buggy piece of software are grand; at other times, they are not. Examples of such costs include the loss of customers due to poor software quality, consumption of critical computing resources due to poor software performance, or excessive maintenance costs due to the necessity of managing existing bugs. One paper cites the following: "IBM reported spending about $250 million repairing and reinstalling fixes to 30,000 customer-reported problems. That is over $8,000 for each defect!" (Laurie Williams and Alistair Cockburn, "The Costs and Benefits of Pair Programming.")
3. *Debugging costs:* The cost of fixing bugs should be weighed against the alternatives: releasing bugged software, or taking a more preventative approach to

bugs. A Computer Science professor of mine stated, "Coding is 90 percent debugging." Fixing bugs once they are spotted is not the most significant issue here: bugs are sometimes simply spotted, but are often notoriously difficult to diagnose. I have heard stories of people being derailed spending upwards to twenty hours searching for small, simple bugs in C code.

4. *Bug prevention costs:* Often, an ounce of prevention is worth a pound of cure. But that is not always true: an often efficient way to deal with bugs is to wait for them to arise visibly in testing. Bug prevention is useful when its cost is less than bug release costs. Realistically, any development environment will achieve some balance between prevention of bugs and fixing of existing bugs, but what balance is necessary may vary greatly from project to project.
5. *Testing costs:* Efficient testing is paramount when bugs are costly. When deciding on a testing scheme, common sense should be applied: the level of skill required to successfully test a piece of software will depend largely on the nature of the assignment. In some cases, verification is a task that can be carried out simply by someone other than the person who programs the software. Other aspects of testing may be too complicated to efficiently fall on the shoulders of anyone other than the person who wrote the software or who is most familiar with the modeling of the problem that the software is solving.
6. *Labor costs:* It goes without saying that the opportunity costs of the developers must be analyzed. Clearly, one should not want to pay a head software engineer's wages to get entry-level work done. But deciding upon an efficient division of labor is not always effortlessly achieved. Also, labor costs are affected when other decisions are made. To take an example raised by a friend of mine in the industry, Josh Fuhs, suppose a manager is interested in developing a piece of software to work at optimal speed. As such, he chooses a very low-level language like C for all his employees to work with. Additionally, suppose he decides not to write libraries to automate and secure hard-to-debug tasks. As a result, no one but high-end, debugging-skilled programmers can effectively contribute to his project, and he must pay a premium to have simple maintenance or development tasks accomplished. Perhaps the project will take longer to finish than necessary, and time-pressed developers will select bad algorithms throughout their code. As a result, the code is far under-optimized, and the strategy fails to optimize the code as desired. It turns out somewhat like a tricycle with racecar wheels.
7. *Communication / interaction costs:* Supposing that two groups decide to write separate but interdependent components of a software project independently, how do they communicate information necessary to integrate their project? Developers cannot ignore each other but by design and foresight, which is often costly in and of itself. Some texts (e.g., Eric Braude, *Software Engineering: An Object-Oriented Perspective*, p. 81) suggest that, from experience, programmers usually work most efficiently when interacting with three to seven people on a given project. A motivation for this figure is that when programmers work alone, misunderstandings of project requirements hamper productivity; when programmers must interact with too many people, there is little time left over for development itself. Generally speaking, we might say that the more interdependent the work of two groups, the more closely they will need to

communicate. This may speak strongly against excessively rigid, inflexible approaches to project management: if the components of the project cannot be separated into highly independent parts, then there is little sense in making programmers work independently. If, on the other hand, there are highly independent tasks to work on, it may not be a good idea to require that everyone be in the same room before someone proceeds with work: perhaps we should allow the worker to take some tasks home under such circumstances.

8. *Maintenance / usage costs:* Poorly documented or hard-to-understand software can be hard to maintain. The more skill or knowledge required to maintain a piece of software, the higher the costs.
9. *Modification costs:* In the same way that poorly documented or difficult to understand, it may be much more difficult to modify. Developers who idly do any task the first way that comes to mind, without questioning how their decision will affect their later efforts, are costing their managers a lot of money. Unless the incentive structure is set up so as to reward easily modifiable software, we cannot simply expect developers to cut down future modification or maintenance costs out of self-interest: the cost of paying for extra modification bubbles up to management and is paid to developers themselves, without a good incentive structure.
10. *Incentive structure:* An organization should give incentive to do those things which benefit it, and disincentives to those behaviors which harm the organization. The incentives can come in the form of recognition or money. But more important than the means of reward is deciding *which* behaviors to reward. Do we reward an employee only for direct, independent, individual accomplishments? What if the employee fails to make his code modifiable by anyone other than himself: Do we punish him for writing unmaintainable code, or do we reward him for (seemingly) being the only person up to the task of understanding his own code? Do we reward employees' code based on whether or not their software passes tests cases and is produced on time and within budget, or do we also check their code against modifiability standards before deciding on their reward? Are we too soft on those who refuse to meet standards? Are we too hard on those who miss a three-month deadline by one day? Do we punish people for being unnecessarily difficult to work with? Does our reward system reward ball-hogging and politics, or does it support team play? If we support team play, how do we measure individuals' responsibility for team accomplishments? Do we keep an "assists" statistic as well as a "points per game" stat? Are we rewarding production or character? If character, is it the kind of character that profits the company or the kind that is agreeable to the interests of managing agents? Are we too dictatorial, too lax? Do we implement oversimplified or soft rules, like "Keep the pressure on," (taken from Braude, p. 78) as hard office law? And so on.
11. *Risk management:* Are we developing in a language that will be supported in the future? If there are "upgrades" to the development programming language, will we have to rework 90% of source files, or just two? What if one of our key developers becomes temporarily unavailable: will we still meet our deadline? What if we don't: is all of our work for naught? What if requirements change? What if we misunderstood requirements? What if the project is scrapped? Such

> major risks should be addressed at the optimal time (usually early), with *reasonable* effort given to analyzing contingencies where desirable.

Again, at this point we are yet to define any kind of comprehensive, mathematical model for estimating development costs of a project. Instead, I have aimed merely to define the variables (some of which may be independent) to which development cost may be sensitive. These will serve as prying points in analyses of aspects of development that follow.

Analyzing Development Techniques

Here I simply hope to compare a number of development-related techniques, or aspects thereof, to determine how, why, and under what circumstances they are valuable.

Pair Programming vs. Solo Programming
This technique involves having two employees programming a single task, with only one of the two at a time typing at the computer. The most prominent methodology to employ pair programming is Extreme Programming. It works basically as follows:

> The best way to pair program is to just sit side by side in front of the monitor. Slide the key board and mouse back and forth. One person types and thinks tactically about the method being created, while the other thinks strategically about how that method fits into the class (from http://www.extremeprogramming.org/rules/pair.html).

A certain line of analysis argues that productivity per employee diminishes with each programmer added to any given computer screen. Isn't this having two people do the work of one, and thus, half as efficient?

This assumption, in fact, does not hold true empirically. Laurie Williams, in a paper co-written with Alistair Cockburn, "The Costs and Benefits of Pair Programming," concludes that "for a development-time cost of about 15%, pair programming improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications *and* is considered more enjoyable at statistically significant levels" (http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF; quoted June 18, 2003).

Why should we expect this be true, intuitively? First, as Eric Braude notes, pair programming is, "in effect, a form of continual inspection" (*Software Engineering: An Object-Oriented Perspective*, p. 116). If we expect that code inspected by someone other than its direct author will be better designed and less defective, this explains part of the conclusion. I have tried to establish this elsewhere in this work. Still, why should we expect development time to be only 15 percent more, rather than a full 100 percent more? ("Two doing the work of one," right?) The best explanation of this is the following: having a second person at the computer reduces the number of bugs that are corrected. In practice, when I have programmed with a partner, the partner not only points out the more obvious syntactical errors that I make, but also is unlikely to share my subtle conceptual mistakes. Two programmers will rarely make the same programming mistake

independently, and thus, when paired, one corrects the other, and neither of them spends all afternoon on a wild goose chase. Mistakes that are not made the same way by both persons are discovered essentially as soon as one of them says, "I don't see how that's going to work."

Another interesting point is that pair programming was suggested to be more enjoyable in this study. Over 90 percent of those who pair-programmed in Laurie Williams' experiment said they enjoyed programming more in pairs. The implication for management is a shift in the labor curve.

Code Review and Inspection

It is difficult to list *a priori* truths about inspected products beyond the fact that, all else equal, they should do a better job of passing an inspector's scrutiny. It is possible to have an incompetent inspector (or manager) view code and inspect the code in such a way that actually detriments the quality thereof. A near-categorical imperative of inspection should be to tailor the inspection of the code to whatever goals would benefit by inspection. This may sound tautological, but its application can be tricky. For instance, when might you want a less skilled programmer to inspect the code of a more talented programmer? Never? Or perhaps, when the code the more talented programmer writes will need to be modified, updated, reused, or maintained by less talented employees, and the code needs to be easily readable? (I will here be using concepts of "review" and "inspection" nearly interchangeably, both signifying the examination of existing software modules in order to find defects or room for improvement. By "defects," I am not limiting the term to functionality defects; defects can also be design or requirements specifications.)

Pair programming serves as one form of inspection – continual inspection. When two programmers sit at one computer, one programmer, typing at the console, is likely to think about those things a sole programmer ordinarily thinks about. In particular, the typing programmer is likely thinking actively about syntax and such immediate programming concerns. The second programmer, the one who sits nearby and inspects the code, does not have to do so, and feels free to think about design patterns, coding standards, or generic problem-solving concerns.

Code review meetings can contain a much different form of code inspection. Here, meetings are held in which some group of developers inspect what has been developed. What is inspected, and how it is inspected, should not be decided arbitrarily. If programmers are not meeting coding standards, inspect code against standards at group meetings. If programmers are in danger of straying from user requirements, inspect the software's function to ensure that it is serving valid purposes and implementing necessary functionality.

Code reviews that do not seem to have genuinely logical motivations are likely to be dismissed as time-wasting routines. In particular, if someone who is not directly involved in writing a given program offers critiques of the author's code, that reviewer needs to be making critiques that he or she is in a position to offer, given that the coder often has a

better vantage point with which to assess certain specific aspects of his or her code. Some code style defects are obvious, such as overly lengthy method bodies. Others are far from obvious: has the inspected code been pasted from elsewhere? A potential problem with code review meetings is that they tend not to be terribly enjoyable, and if the suggestions made are not worth the time spent, or the meetings do not stay on track and address necessary business, they will grow to be viewed as pointless rituals. Cynicism breeds quickly if all feedback is negative. To many in the computer industry, "No news is good news" seems to be an unfortunately dominant way of deciding what to talk about during meetings and reviews, and one can imagine what effect this approach has on morale. Also, it should be noted that the goal of code reviews is not to make up-and-coming programmers feel incompetent by bombarding them with criticism from all directions. Remember to give only so much advice as one can rationally be expected to find helpful.

Criticism, it *should* go with out saying, should only be directed towards reasonable goals. If one code reviewer looks at a piece of code and says, "You know, if you do XYZ, your code will run [insignificantly] faster," but the software being developed is anything but speed-critical, his optimization-obsessive criticism will be distracting and anything but helpful. Unacceptable levels of pointless criticism should be punished.

Many XP enthusiasts have seized upon the negative effects of code reviews on morale, as pair programming instead does away with this. When two programmers code, there is a sense of collective accomplishment, rather than being bombarded with criticism from all angles, which results from pairing a less-talented programmer with a more talented mentor-programmer. That does not imply that a competitive coding environment is bad. If code reviews are to be held, (1) not everyone necessarily needs to attend each review (to avoid the damning presence of those who are overly critical), allowing reviews to be organized by skill level or along project organizational lines, and (2) negative comments can be hedged with offsetting comments wherever appropriate. "You need to replace that with a something more modular," can be more tactfully stated as, "I like the way this code of yours is modular and easy to modify here. [Explanation of principle in non-condescending terms, if it is new to them.] Do you think you can go ahead and do that with this code right here too?"

In spite of the criticisms, group inspections of code for bugs have been shown in studies to be worth the time spent (Braude, p. 44).

Coding Standards and Rules vs. Developer Freedom

There is nothing inherently laudable about a standard or rule. Enforcing a programming standard is, to some degree, a restriction of the programmer's freedom and creativity, or at the least, the programmer's ability to work outside those standards. In the office as with running a government, there is a tradeoff between freedom and control. Neither is a golden hammer.

Thus, motivating any standard or rule is a matter of demonstrating that the benefits of the standard outweigh the costs; in particular, it must be shown that the expected result of

enforcing standards is better than the result expected when programmers are endowed with the freedom to work in more utilitarian fashions, disregarding those standards.

An example of a very elementary rule worth following is that copy-and-paste reuse of code segments should be avoided. Why? Copied code is intended to reuse functionality from its source. But if there is a better way to reuse functionality, why not use it instead? Let's examine copy-and-past coding and its alternative: Pasting copied code segments has the same effect on functionality as replacing the copied block with a method call, and then duplicating the method call into the segment of code where the copied code block would have been pasted. So if the functionality produced by both methods is the same, we proceed to evaluate the costs of doing it one way or the other. When using the copy-and-paste method, only one copy and one paste is needed. As such, the immediate time for the code to be produced is short. When using the method extraction technique, a cut and paste is needed, but then again, so is defining a method body, and possibly doing extra work to make sure both execution points have access to the method definition. So, why all the extra work? The reason is that when bugs are found with that code segment, the method-extracting programmer must (1) debug the original code, and then (2) must also find any places where that code was pasted and perform the same debugging operations (not always photographically remembered) on it. Or suppose that it is not even the original coder that finds the bug: how is this coder to know where the buggy code was duplicated? This could be especially tricky if the copied code was duplicated throughout several files. The costs of locating all the duplicated bugs usually greatly outweigh the cost of writing a method body. In addition, a descriptive method call, like "foodDepartment.addToInventory(InventoryItem.FROZEN_PIZZA)" is usually more easily understood in context than the functionally equivalent code segment. The intended purpose of the code segment itself is also more easily discerned within a well-named function definition, making it easier to assess whether or not it does what it is intended to do. Similar arguments apply to situations in which code segments are modified for non-debugging reasons, and are particularly applicable when attempting to design large-scale, scalable, flexible systems. If the code is never to be debugged, maintained, or modified, however, the rule is unlikely to be necessary. As such, "Avoid copy-and-paste coding" is a good hard rule for situations where reuse and expansion are anticipated, or bugs are highly expected, but can likely work only as a softer rule otherwise, e.g., in many not-quite-programming situations (for instance, copying a chart once and modifying it in a word processor document for one-time use).

As the example above illustrates, development standards (1) are not always best decided on the basis of immediate, direct concerns, and (2) rules apply to varying degrees in varying development contexts. That said, while it certainly helps to be able to flexibly replace rigid, cookie-cutter development methodologies with more sensible alternative approaches, discerning the overall impact of implementing any given methodology can be a tricky practice that requires considerable professionalism or good judgment to achieve effectively. Blind adherence to the standards of the industry is, as always, a safe, conservative route, one that wastes little time barking up the wrong tree with untested methodologies. Analyzing the accepted rules and standards is costly in itself, and businesses prefer not to experiment too carelessly when money is at stake. Unusual minds

with abilities to analyze pioneering or customized development techniques certainly help, though I would not be surprised if few organizations had them. While broad industry experience and empirical observation usually form a safe pier from which to fish for sane decisions, this is no substitute for real analytical talent. Also, in an industry that has been rapidly developing and techniques grow quickly obsolete, industry experience may count for little. Some experiences teach timeless lessons. However, yesterday's "Internet prodigy" is today's unemployed college-dropout former JavaScript teen.

Maintainability-/Reusability-Numb Incentive Structures

As complicated as writing good software is, setting up an incentive structure to foster good software can be even trickier. In many situations, direct, obvious types of incentives, such as paying someone by their competitive ability to meet deadlines and user requirements, or paying someone per line of code written, or by whether or not they seem to sound intelligent, and promoting those who do the best work that is rewarded by these incentives, will suffice. In other situations, this strategy could have less-than-stellar results.

An easy example of simple, direct incentives working properly goes as follows: programmers are paid by their ability to produce code to specification within deadlines. They work entirely independently of each other and have entirely individual responsibility for their own code. No one must use, maintain, modify, or build on any code written by anyone else. In this case, programmers who write unmaintainable code will be punished in that if their code is unmaintainable to anyone, it is unmaintainable to them alone. If a bug arises, they are the ones who lose their own time and, as a result, miss deadlines and lose out on pay. And so the incentive works.

In other circumstances, however, this incentive may run awry. Here is a simple example: Suppose John and Suzy, two hotshot programmers competing for a promotion, are assigned to two programming tasks, P_1 and P_2. Programming task P_1 is composed of two consecutive sub-tasks, X_1 and Y_1. Programming task P_2 is also composed of two consecutive sub-tasks, Y_2 and X_2. Suppose that John knows how to program the "X_i" tasks, and Suzy knows how to program the "Y_i" tasks. (Assume, for simplicity's sake, that all of these subtasks would be equally difficult, and would take an equal amount of time for a given programmer to complete; i.e., if a programmer worked on X_1 and Y_2, it would take him the same amount of time; if he then went on to code Y_1 and X_2, those two tasks would take an equal amount of time to complete. The argument would proceed similarly, in more complicated language, if this were not true.) If he acts out of self-interest and ultimately most want the promotion, then John will want the following to happen: that he completes his own code quickly and that Suzy will not complete her code as quickly. John, who, let us say, needs to pay off his kid's college expenses, has to finish his code quickly, and gives little attention to design or modifiability. In fact, he may be tempted to not to worry about certain "time bombs" in X_1 that will blow up on and burden Suzy when she has to program Y_1, which builds on John's programming in X_1. ("She's supposedly smart, and if she's as talented as they say she is, she can figure this out when she gets to it.") Suzy, let us say, is noble wants to optimize overall programming efficiency, and tries to leave an easy job for John, and hopes that John will

leave her an easy task, and that both of them will ultimately code with optimal efficiency. Thus, Suzy programs task Y_2 maintainably, taking a little extra time in order to save John more time in the long run when he has to build on Y_2 by coding X_2. Suzy, being a somewhat superior programmer, takes fewer shortcuts and still finishes her sub-task, Y_2, in the same amount of time as John. John now must code X_2, and Suzy must code Y_1. John gets to work with Suzy's maintainable code in Y_2, but Suzy must continually rework John's code in X_1, defusing his various time-bombs and unmaintainable code segments, in order to finish task Y_1. John, however, has been lobbed a beach ball by Suzy, who wrote him code that was easy to work with. As a result, John meets his deadline for X_2 faster than Suzy meets her deadline for Y_1. John now appears to management to be the more talented programmer, and Suzy appears to be a bit slower. While Suzy's behavior, if spread through to the rest of the programmers, would drastically increase overall efficiency, John's behavior was the sort that gets rewarded. As a result, John is promoted.

While such a case may sound like something that would arise only in the mind of a game theorist, it turns out to be reality. Actual developer attitudes towards programming modifiably can be quite appalling. There is a saying among many programmers that "Whoever makes his code reusable doesn't know the meaning of 'job security.'" Managers should be aware of attitudes such as the following, which can be found through a cursory survey of a *Slashdot* discussion board:

> What started off as a no pay internship turned into a $60/hour job, mainly because my code was valuable and it would have been impossible for someone else to figure out what I did and how it worked.
>
> Every time I mentioned other job offers, they just raised what they were paying me. It would have taken months for some fool to understand my code and do anything useful with it.
>
> I decided to continue in grad. school and I feel kinda bad leaving them with an expensive mess of spaghetti. If they pay $100/hour I'll spend my winter break making it understandable. (URL: http://slashdot.org/articles/99/11/17/2148249.shtml; quoted June 18, 2003)

A manager hoping to feel underestimated ought to have a look at the Slashdot discussion board. Articulate rebuttals to the notion that hairy-looking code is more valuable to a programmer are sparse. The incentives given for writing maintainable code are generally of the pathetic form, such as the idea that people should enjoy writing "neat" code for its own sake, for aesthetic appeal, or "just because," so to speak.

The problem is that programmers are given little real incentive from management to make their code reusable, unless it helps them complete their own program faster. Managers who know when, how, and to what extent to give incentives to their programmers to make code reusable and well-designed will certainly be at an advantage. A good start is to hold code reviews where a knowledgeable software engineer can assess code against reusability and maintainability standards. Hackers who deliberately obfuscate their code should be sent elsewhere.

In saying all of this, however, there is a key distinction to be made between *natural* complication and *artificial* complication. Some code seems complicated as a natural result of the fact that it *solves an inherently complicated problem* (such as multidimensional collision detection). Other code is complicated as a simple result of the some shortcoming brought on unnecessarily by its author (such as lack of modularity or object-oriented design). The best code is simple in appearance, yet solves a tricky, complex problem; the worst code is that which solves a simple or routine problem in an unmaintainable, esoteric fashion. However, managers should not let programmers off the hook just because they *claim* that their code solves an inherently complicated problem. Programmers often say such a thing, in earnest, but when pressed, will magically contradict their claim by producing a simpler, more modifiable solution.

Of course, one cannot press one's workers to write reusable software if no one has a good concept of what reusable software is. There is no way to magically obviate this need. Management which does not understand these concepts fully would do well to consult with, read from, or otherwise employ the services of a talented software engineer.

Design by Foresight "versus" Evolving Design (Refactoring)

Whether design is better done on the fly or in advance is not given any obvious answer by intuition. While it is tempting to suggest that "An ounce of prevention is worth a pound of cure," my programming experience is that that is not always the case. It is sometimes much easier, and more enjoyable, to refactor design while writing the program, letting the design evolve into something of higher quality. (A refactoring is a design change to a program that does not change its functionality.) A necessity of good design is to have an idea what good design is. How to achieve good design is another question. Common sense will dictate that programmers who do not take minimal steps to plan out how their program will work will often lose time wandering in the dark and finding out that their design does not work. That said, some ideas about design changes are not worth thinking about, or are not visible, until some of the programming actually happens. In short, there are some things that should be planned ahead and some that should just be thought out as they arise.

Generic versus Minimal Design

Conventional software engineering attempts to take into account possible future modifications when designing software, thus tending to make their design generic and ready for future updates. Programmers from the more recently developed Extreme Programming school of thought often think of this generic design as "speculative generality," and have a phrase, "YAGNI," standing for, "You aren't gonna need it." Rather than anticipate future design changes, one can use the most simple, minimal design necessary, and refactor the design later when more generic functionality needs be added. While this can be very practical under most circumstances, there are some risks that can likely be seen from a mile away and would take a long time to rework from an existing minimal design. In such cases, clearly it may be much smarter to do it the first time. Obviously, if a change is 99% certain to happen, it may be smart to go ahead and do things right the first time, but if there is no reason to believe that chances of needing generic design later on are truly significant, it makes little sense to go to lengths to

prepare for it. In short, one should weigh the cost of implementing generic design against the risks taken, and cost of refactoring and rewriting the code, if one does not implement a more generic design. A more easily stated rule of thumb would be simply, "Ensure that the organization is *ready to efficiently accommodate* future changes *that are likely to happen*; when software would take much more time to refactor for generic use than to rewrite for a given change, write it generically the first time."

Infrequent vs. Continuous Integration
Extreme Programmers advocate continuous integration of software components; conventional software development does not necessarily advocate this. It should seem that, all else remaining equal, the more that each integration requires developers to change their existing software components, the more frequently integration should be performed so as to prevent programmers from building onto code that has integration-related bugs. If integration is a process that will not generate such bugs at significant levels, but is a time-consuming process, it would hardly seem necessary to integrate continuously. Of course, experience often proves that miscommunication and incompatible components often generate grand integration-related headaches.

Infrequent vs. Frequent Communication
This should depend on the nature of the project. If specifications are tricky or requirements are miscommunicated, then more frequent communication or more extensive documentation should be considered.

Writing Tests Before vs. After Writing Code
Extreme Programmers write a failing unit test before writing any segment of code. While this may seem a very costly thing to do, it is argued that writing tests right when code is written actually saves costs by finding bugs right where they start, and preventing code to be layered upon a faulty implementation. Searching for bugs that were written long ago, in other contexts, can be excruciatingly difficult, sometimes costing dozens of hours, or possibly weeks. Writing tests before code is written can greatly diminish the necessity of such efforts. The downside is that writing a comprehensive or near-comprehensive test suite can be an extraordinarily time-consuming task in itself that, in many contexts, can perhaps cost greatly more than the types of bugs that can rationally be expected to arise. One should, of course, be weary of analyses that never halt in their march to that conclusion.

Waterfall vs. Iterative Development
The *waterfall model* of the software development process is one where the development process is presented as roughly following a sequence of six steps, adapted from discussion in Braude, pp. 24-26, and "Rational Unified Process: Best Practices for Software Development Teams," p. 12. (This document is available at http://www.rational.com/media/whitepapers/rup_bestpractices.pdf as of June 20, 2003.)

1. *solution modeling*, modeling the solution to the problem being solved,
2. *requirements analysis*, where requirements specifications are determined,

3. *design*, where documents showing design of necessary software components are produced,
4. *implementation*, in which code and comments are written,
5. *integration*, during which one turns the separate components into a whole, and
6. *testing*, in which defects in the program and its units are sought out.

As was noted above, however, delaying searching for defects until long after code been built atop them has considerable disadvantages. It thus makes sense to test more continuously. Many developers, in fact, find the *iterative* model of software development better suited. An iterative process is essentially one which repeatedly follows some set of steps, such as the steps of the waterfall process listed above. The Rational Unified Process (RUP) and Extreme Programming are iterative processes of software development. The manners in which these iterations can proceed are varied, and as such, no brief analysis will extensively cover all the benefits or difficulties associated with all aspects of iterative development processes. One universal statement can be made, though: in an iterative process, one has whatever security comes with knowing that the last iteration has been performed. For instance, if each iteration includes testing all code written during that iteration, then in the event project deadlines were cut short, one at least knows that the existing software, up through the last iteration completed, has been through testing. This would certainly add some stability in the face of potentially changing deadlines.

Staffing

Various approaches to managing labor have been tried in recent years. I will make little in the way of suggestion here other than to advise avoiding overly simplistic solutions when it comes to staffing. Here is an example of a possibly misleading line of thinking (in deductive form):

1. The lower the developers' wages are, the lower the development costs will be.
2. Low-wage workers are capable of doing our work.
3. Therefore, to reduce development costs we should only hire workers at the lowest wages possible necessary to complete the task.

This may sound like a valid argument, but it can be astronomically off-target. A trivial counterexample to the reasoning follows: Company C Corporation needs to produce documentation for its Java code. Former JavaScript web programmer and current college undergraduate (off for the summer) Jim offers to document the code at a cost of $12 an hour. Software Engineer Sam offers to complete the same task for rival company D Corporation, and has a wage of $40 per hour. C Corporation uses a policy of hiring low-cost Jim to document Java code directly for $12 per hour, and never stops funding this "cost-efficient" program. D Corporation hires Sam, who points out the existence of javadoc as an automatic documentation utility, trains the current employees on how to use javadoc, and leaves after one week of work. From there out, the work required to document is far less costly than C Corporation's work.

The example above may seem too obvious for anyone to overlook, but it does indeed contradict the "reasoning" given in the deductive argument. Furthermore, the same type of thing is likely to happen in less obvious ways in reality. A company resists hiring a software engineer who could write generic, pluggable controls for database operations, when they know that their database selection is rapidly changing; it is not thought of as a "necessary" expense, and cheap workers can finish it anyhow.

At the other end of the spectrum, hiring overqualified workers is no good either. There is no need to pay off mere opportunity cost, and a product produced by someone with more overall talent is not necessarily a better product for the cost.

If the efficient market theory holds true, and all relevant information about a purchase is taken into account by the time market price is determined, then there is little need to analyze offers. Similarly, if the market were totally inefficient, then it *might* be true (though unlikely is) the case that workers at certain given wages are a better deal than others. The reality is probably somewhere in between: neither cheap workers nor expensive workers are automatically known to be superior deals, though, of course, cheaper or more expensive workers may be a better deal relative to that group's market price than the other, given the nature of the project. If there is a difference in price between two developers, there is likely a reason for it to be so. A guiding axiom for labor decisions should simply be to know the benefits of hiring a given laborer and to understand why the cost of hiring the worker is as it is. In practice, that is not so easily achieved as many of us might think, especially if management does not accurately understand the nature of the problem to be solved, and the principle is perhaps more commonly violated than we might initially imagine. My mom once said, "If I was a manager, I'd hire so-and-so over you in a second." My response was, "He's more talented than me at several things, but there are some things I do better than him, too. So who I'd hire depends on what type of work needs to be done." You get what you pay for, not necessarily something worth what you paid.

Customer On-Site vs. Not

Extreme Programming has the practice of having a customer or client on-site for the development of the given software. While this certainly helps to avoiding misunderstanding requirements, what if your customer is a specific brain surgeon, an attorney, or an actuary, yet the software requirements are clear? Having a customer constantly on-site may not be worth the cost.

Application: Extreme Programming Extremism

I am something of a fan of Extreme Programming in that it offers a somewhat refreshing methodology as compared to many types of formal, conventional thinking about software development. Extreme Programming emphasizes pair programming, an on-site customer, simple design, collective code ownership, testing before coding, and continuous integration. There is certainly much use for these practices. However, some Extreme Programming advocates suggest that anything less than adopting all of the practices of Extreme Programming "is not Extreme Programming," as though this is some sort of

indictment against the partial practice of XP. The question immediately should arise to this line of argument: so what if it is *not* "Extreme Programming"? What is in a name? All that matters is whether or not the practices adopted from Extreme Programming work as adopted. As shown above, the appropriateness of several of the practices of Extreme Programming depends on a number of variables. Where a rational analysis shows that some of those practices do not work, they should not be adopted, but the baby needs not be thrown out with the bathwater in such instances. Contrary to some thinking, it is possible to think of circumstances where partial practice of XP is more appropriate than total XP or no XP.

Conclusion

The analysis of software development is not an exercise achieved as simply as some gurus may suggest. Ultimately, the purposes and methods of developing software vary widely enough that a case-by-case analysis may be profitable under certain circumstances. Stiff adherence to soft rules will create undue confusion about development; adopted rules of development should be somehow related to the contexts in which they work if a rational party is to be convinced of their effectiveness. Lacking that, software developers may unnecessarily grow to view their practices as empty ritual. In the end, there is no substitute for good judgment about software development and careful weighing of costs and benefits of development techniques that extends beyond the most direct and obvious concerns.