
Approaches to Software Development

Concepts and Ideas

Thomas E. Besemer

Copyright © 1998 -The Besemer Corporation. All rights reserved. This document may be reproduced and distributed within corporations and engineering environments freely, but not published, intact or modified, without prior written consent of The Besemer Corporation. This is a preliminary draft, and should be treated as such.

1.0 Introduction

After more than ten years of consulting, working with dozens of companies, and then this past year traveling 65,000 miles in six months delivering technical training seminars to engineers and managers, I have many war stories to tell. I have seen many things, and have gathered a unique education through the wandering that a consultant tends to do over the years. This article is designed to introduce the reader to some of the consistent traits I have seen for software projects which both succeed and fail.

It is important that the reader understand that I am only trying to be a reporter; although without doubt my own personal opinions will creep in from time to time, it is truly my hope to present the consistencies and behavior patterns for both successful and disastrous software projects as I have witnessed them over these past ten plus years. It is left fully to the reader to determine if they wish to work with these ideas, recognize any of the described characteristics in their own software projects or make changes based on any discussions or reflections which may follow their reading this.

1.1 What's Contained

It is my hope that I have clearly reported both what I have witnessed which does work well in software project execution, as well as what does not work. Notably, successful projects have many consistent characteristics and traits, just as projects do which don't reach their fullest potential.

It is easy to generate criticism, state what isn't going to work, and people in general don't like a complainer such as this - they like people who offer solutions. In writing this, I felt it important to present what does not work with equal fairness as what does, as often you cannot appreciate solid approaches without being able to look honestly at

unhealthy approaches (some of which may exist in your own software projects and engineering environments). I have tried to outline the consistent traits of failed projects, as well as the consistent traits of highly successful projects (and possibly the people behind both types of projects).

1.2 Target Audience

It is designed to provide insights for both managers and engineers. Managers may be able to recognize patterns in their staff and make improvements based on these observations, while engineers may also be able to recognize consistent patterns in their managers and learn to react better, or provide them additional information to improve their environment.

I have seen two things which are sure killers of software projects: *arrogance* and *lack of communication*. I would be arrogant if I tried to state that this article is the definitive text on how to execute software projects. However I may be able to help prevent problems by providing the readers with this article, perhaps presenting an opportunity, or starting point, for improved communications. In this, it is my hope that managers and engineers will use the information contained in this article to start discussions, improve discussions, adopt new approaches or discard of existing unhealthy approaches.

In addition to these two groups, Venture Capitalists may find this article useful when working on forming or funding a start-up.

1.3 Scope

The scope of this article, from a technical perspective, is geared towards people doing embedded, possibly real-time, systems. The readers should have been through one or more significant embedded design and implementations, and through their own experiences, be able to relate to the information provided within this article.

1.4 Article Goals

This article is primarily designed to be a starting point for discussion between the reader and the readers peers. The goals are simple:

- Help identify what does not work.
- Provides some ideas and concepts on how to proceed with a large software project in a fashion which results in project execution which is a success for both management and engineering.
- Improve communications between management and engineering, and among a team in general: one of the keys to success is open minds and open communications.

The reader is encouraged to evaluate what is contained in this article and try to apply it to their own environments, and is reminded that one of the keys to failure, both in life and in projects, is denial.

2.0 Success Versus Failure

Software projects often tend to be peaceful and full of lofty goals and dreams at the start, slowly turning into frantic, buggy, erratic, late and frustrating experiences.

Not all end this way, but many do. Understanding how this happens may help the reader avoid these frantic endings through recognizing early on characteristics which may have limited successes.

A few things that consistently do not work:

2.1 Lack of a Solid Schedule

This is a big killer. The lack of a solid schedule reflects the lack of a strong understanding of the projects needs and goals. It prevents maintaining accountability, and helps contribute to the “it’s almost done” answer from engineers.

All poorly executed software projects are late. Some don’t work at all. I have seen people fall into delusions towards the end of projects like this, making plans and commitments based on hopes, not realities. Without a solid schedule, neither the managers or the engineers understand the status of their software project at any time, and fall in to the trap of stating what they hope is the truth, which is often very far from reality.

More so, late projects are often late because of all the “one day projects” which got tossed on the back-burner through the course of the projects execution. So many times I have seen projects get down to the final stretch called “integration,” only to find that with one week left on the schedule, there are twenty, thirty, fifty or maybe even more, one day projects left to be accomplished. Thirty days of unresolved issues is thirty days that were not on the schedule, and thirty days is thirty days, period.

2.2 Over Complicating the System Requirements

This is most often done by in-experienced designers or technical leaders: making the system way too complicated. Lack of experience means they have not been burned before, while over-complicating the requirements is merely an exercise in ego.

Software is complicated by nature. Multi-programmer projects, or projects with complicated integration needs, tend of be difficult to execute on schedule while maintaining quality. Projects which are overly complicated simply tend to add to what is already, by nature, a very difficult process.

Over-complication is not always an engineering problem; it could also represent a significant management problem. If an engineering manager does not recognize the need to keep the approaches as simple as possible, it may recognize a lack of experience that will contribute to failure over the execution of the project.

Two key things contribute to over-complication of design:

1. Ego. An inexperienced engineer or group of engineers who has a strong need to prove how smart they are (experienced software people demonstrate how smart they

are by actually executing the project on time). It's often easy to recognize these people: keeping your eyes open for arrogance, through the consistent use of the "I" word ("I designed this", "I manage a large group" or "I want it done this way") may help. Winners tend to say "ya, we did get it going", and indicate that they appreciate their good fortune - the "we" word is all powerful, and all telling.

2. Lack of understanding of system needs. Engineers or managers who do not understand their goals, their customers or the system needs tend to hide behind statements of over-complication (designed to warn people, in the end, that things may be late because it is extremely complicated), and also tend to try and design complicated machines which will do anything (therefor, after they figure out what they need to do, they will hopefully have done something that can be massaged into functionality).

I do not wish to imply that all software is simple; very seldom is software simple. I am only trying to report that when software is made more complicated *than it needs to be*, the projects tend to be less successful than they could or should be.

2.3 Staffing - The Critical Component

A company's staff defines a company's culture and beliefs. Companies that founder seldom do for lack of a good concept or idea. They tend to wash up on the reef sooner or later for one simple reason: they hired the wrong people.

This is perhaps the most difficult area to cope with, and the one with the most significant long-term effects. Hiring of the wrong staff happens due to a few consistent reasons, not all of which exist at the same time:

- The engineering manager hires before they system requirements are understood, therefore possibly hiring the wrong people or persons. As the requirements are understood, a company may find itself saddled with a good engineering group - good for another project with another scope.
- The engineering manager is not qualified to evaluate and make recommendations on staffing needs. This brings up a very big question - who hired the engineering manager? Software project execution may not be the real problem in a situation such as this.
- The conceptual mind behind a project (the dreamer, or idea person) is given ownership of the project ("it's their baby, they will make it work"). This is not always a reason for problems, as there are many brilliant and creative thinkers who also have pragmatic management approaches and understandings, yet I have seen this more than a few times - the creative mind hires and manages, and ends up with a group of dreamers and theorists, not a group of people who can execute. Perhaps it is best stated that there are dreamers and designers, and there are mechanics, builders, and ideas need to be built by people who know how to construct software.

A big side effect of hiring the wrong people tends to be acceptance of them as time goes on; although they may not be the right people for the project or company, they tend to stick around, defining it's culture, and defining how well the company will be able to execute their projects.

Pragmatic managers are able to evaluate situations like this periodically, and make the difficult but necessary decisions for the organization, which often needs to be helping the problem staff understand that they would be better suited in another organization.

2.4 Management by Waiting

A common problem among managers who are not as qualified as the organization requires; they may not understand the problems, in through this, cannot offer solutions. They simply wait, hoping that somebody will figure it out and solve the problems.

This type situation reflects some deep seated problems in the entire executive staff of an organization; project managers need guidance as much as engineers do. An executive staff which cannot manage their managers often has more to worry about then the execution of a software project.

2.5 The Definition of Done

This clearly one of the most entertaining aspects of watching a software project develop and succeed (or otherwise): defining done. The software people think it's done when they can integrate and pass it to a quality assurance and testing group, while this group thinks the software is done when they can't find any more problems or bugs. Production people think it is done when they can ship, and the financiers call it done when they start seeing revenue.

This leads back to problematic schedules; a schedule should help define milestones, but in doing such, there must be a definition of done, and an accounting means to verify that a component, set of components or entire projects are in fact done.

Without a definition of done, people work with their own definitions. I have literally seen programmers who think it is done (from a reporting basis in status meetings) after they have written the code and possibly gotten it to compile. On the same vein, projects which have been executed without exacting unit testing; as they reach integration, all these so called done modules end up being debugged after integrated - a timely process that is plagued with problems.

Again, this is tightly tied to scheduling, but without a definition of done, a schedule which shows project status and module completion may not reflect reality.

2.6 Design and Execution Philosophies

Design philosophies are really a summary of each of the above sections. A lack of consistent, solid and agreed upon design and execution philosophies results in a random project implementation which is subject to each members own inner definitions and expectations.

Lack of a solid schedule is a lack of a fundamental design philosophy. Hiring or keeping the wrong staff can result in unworkable design philosophies. An overly complicated design reflects a design and execution philosophy which may result in many side effects and problems.

2.7 Clever Programmers

Clever programmers are the downfall of any project. Too many times I have had to “pre-process” a code module through the C pre-processor to find out exactly what is happening. Clever programmers are dealing only with their own egos, with hope that as time goes on the rest of the world will come back and say “wow, you are so smart.” They never seem to recognize the fact that their work is late, buggy and flawed by design, and that the rest of the world is not looking at them with envy, but with concern.

Clever programming is about a state of mind which makes a simple solution complicated. One of the most worrisome traits of clever programmers is how proud they are of what they have done.

I recently witnessed a design review by a clever and complicated programmer. The audience was awe-struck. They were captivated. And they were full of fear; the project had not even begun yet it was already out of hand.

And the designer? Well, he missed this point. He isolated himself even further; clearly these people do not understand or appreciate genius at work.

Avoid clever people; they speak in tongues which none of us understand, or need to understand. If you can't understand their approaches, listen to your instincts; their approaches probably aren't even understood by them.

2.8 Section Summary

Reading the above sections could be seen as simply an exercise in listing of problems without a lot of offered solutions. The reader may be inclined so say to themselves “anybody can be critical.” Yet a reader with an open mind may be able to look at these issues and honestly evaluate their situations, their approaches and their own needs, recognizing some of these traits and taking the step of professional growth by tossing around ideas as a team and making changes.

In this, the “we” word surfaces again; a group, operating as a group and even as individuals, should be able to distance themselves from their egos and say “hey, what are we doing, and what can we do to improve our approaches.”

The team which does not operate as a team does not appreciate the absolute fact that we cannot succeed at anything easily without the help and support of our peers.

3.0 The Keys to Success

Just as the above section reported on some consistencies seen in projects which did not reach their full potential, this section is designed to report on what has been demonstrated as workable. The readers participation is essential; after, or even perhaps while, reading this, the reader may be able to work with others in their group to toss these ideas around.

In these type discussions, some of the above may be discarded, just as some of the reports in this section may be discarded. The important thing is to not keep what does not work because it is comfortable territory, and discard what in the back of the readers mind may make sense simply because it is not familiar and would require change.

3.1 Staff - The Illusive Component

This is an illusive component because it can be very subjective to opinion and judge of character. Although specific engineering managers must spearhead staffing, it's extremely important that checkpoints are built into the process. An executive staff, collectively, should be able to define before starting the hiring and staffing process just what they wish to achieve (this process itself should be a design, and executed by design).

What is important? I have seen the following workable concepts over the years:

3.1.1 Skills are Important, but Character Should Rule

It is extremely important to hire people who are technically qualified to perform their assigned tasks. However, definitions of a project change, so hiring people who can demonstrate a willingness and ability to learn is by far more important. A staff of educated specialists works great as long as the projects needs don't change, and projects are about managing change.

What to look for:

- A strong team approach, devoid of arrogance or ego.
- Honest resumes. If a resume looks suspiciously like it was written for the advertised position, it probably was.
- Offered references. A person with a strong track record has a list of references prepared; they have nothing to hide.
- Demonstrated personal growth, both technically and as a person. When interviewing, asking what mistakes a person has made is very illuminating - people who laugh at their mistakes and talk about how they improved their approaches are confident people who are able to grow.

I think it's important to understand that not everybody is able to take tests well; many people, such as myself, can "freeze" when "tested." I've learned to say "well, I'll try to answer this right now, but I may not understand the issues well enough to give the correct answer." Tough problems require research and bouncing ideas off of peers.

My neighbor, growing up, was director for The Boys Club of America. He stated once that the world is run by C students, and I think there is a lot of truth in that. Academics can often achieve great successes within the framework of school, and pass tests well, but sometimes these people have a hard time working within the real world, which is subject to changes, lack of clear focus and framework.

Most important on staffing is to think of it as a marriage; you may try to guide a marriage, but it is really the marriage which guides you, changes you. A staff is no different; schedules and design philosophies are not directives from an executive staff, but a collection of approaches and ideas from the group - the group guides you. It's extremely important to build a staff of team players who work well together, as any limitations in technical ability can be compensated for through help from peers.

Excellent reference on these approaches can be found in books published by the founders of Hewlett Packard. I was fortunate enough to consult twice at H.P., once for over a year. The lessons I learned about the power of empowering people and working as a team have proven to be the most powerful lessons I have ever learned; I walked into H.P. with a big ego, using the "I" word often, and left using the "we" word most of the time and finding myself achieving results which had always escaped me.

The "H.P. Way" is not for all companies, but every company and group of people can stand to learn a thing or two from how this company was built.

3.1.2 The Management Team

What is the purpose of a manager? Dilbert has provided a few definitions which are often very near the mark.

Seriously, what is the purpose of a manager? Perhaps one role I have seen work is that of a coach. They've been around, they know some good plays, some solid approaches. They are people who help build confidence, who give their team members a little more than they are ready for and then help them achieve these goals. They are capable of delegating (therefor not being a bottleneck), yet while delegating, capable of understanding the intricacies of the technical aspects of a project (through this, being able to spot problem situations before they blow out of proportion).

Since I am not a manager, I would be showing naivete and arrogance to write about a subject which more capable people have studied, executed and written numerous books on. However, as a consultant, I can state clearly that it is rarely a technical problem that causes a project long-term problems: it is always management. Hiring the wrong people, not understanding the technical needs of a project. Not delegating enough, or delegating too much (the "asleep at the wheel" syndrome).

My experience is that solid managers all show the following consistent traits:

- Excellent people skills. The ability to defuse problems, to get "buy-in" by of a group.
- Excellent technical skills. The ability to grasp the technical problems and requirements and guide the staff on design and execution requirements.

- Enough ego to be driven, but enough wisdom to keep their ego in check, and understand that a successful project stands on its own with a statement of how good the manager is.
- Having been through a few big projects from “soup to nuts” - this is the biggest failure I have seen when a management team is assembled; one or two weak ones slips into the system, and although they may be technically qualified, they have not been through enough big projects to understand the pitfalls which lay ahead.

Through the technical seminars I presented this past year, I was fortunate enough to work with many different people, both managers and engineers. I was inside the halls of companies like Lucent and Cisco, and recognized one consistent pattern: the managers of solid companies like Lucent and Cisco were practical. They were investing in education for their staff, yet they were attending the seminars themselves as well to improve their own understandings of the technical details of the project.

3.1.3 The Engineering Team

Successful engineering teams have always shown one consistent trait: they operate well as a team. They offer and accept criticism openly and without being driven by ego, or allowing their own egos to be damaged. They are technically qualified, but each team member is also humble - they tend to check their work twice before making statements of “fact,” and are very open to getting help from their peers.

Experience is always extremely important. In today’s tight job market, it is not always easy to hire experienced people, but hiring the wrong people simply for the sake of getting bodies on board is plagued with long-haul problems.

Balance is also extremely important; a few (and the right number depends on the project) technical leaders are important, but so are less visionary people. The technical leaders are important, but successful ones are interested in helping the less experienced learn and achieve. Less experienced team members can accomplish a great deal with solid guidance.

One of the most important characteristics of successful teams which I have witnessed is the ability and willingness to document their work. From a skill perspective, the ability to write a solid document is key; it shows the ability to understand and present a problem, and the ability and willingness to communicate. When interviewing, perhaps a solid thing to look for is the ability of the potential hire to show you some of their documentation; they may not be able to allow you to read it or keep it due to the proprietary nature of this business, but if you can peruse it briefly in the interview process you will be able to evaluate what level they are at.

Proficiency with the C language is extremely important. When it gets down to it, it’s the code that makes the difference in a software project. This concept may seem blatantly obvious, but so many times I have seen groups with weak coders; they can’t write it well, they don’t have good debugging approaches, the work is sloppy or they simply don’t understand the language well.

Engineering team members who rely heavily on tools often are using the tools as a crutch. Programmers who write “clever code” are often plagued with other technical

problems. When it comes down to it, software projects are successful due to one simple equation: each team member is able to write code in C as though it is second nature. They are able to produce phenomenal amounts of solid code in short order.

Sometimes it is true that behind every great structured designer is a very good hacker. There is nothing like terminal time, and a person who can write solid code quickly is also able to throw away bad code just as quickly - so many times I have seen people struggle for weeks with a module, and in the end, it really needs a re-write. And "re-write" is the surest way to see management tremble.

Everybody makes mistakes. Keeping a bad software module around because it's already been written is the biggest mistake of all. Too many times I have debugged a very difficult problem only to find that the culprit was some terrible module written a year ago, and it's problems were not apparent at the time of generation (it slipped through unit testing).

3.1.4 Consultants

Consultants may or may not have a place within an organization. Over the years I have been hired as a consultant many times for one consistently wrong reason: to fix a project that has taken an undesirable course. How I am hired has taken many forms, from "we just need a little help" to "we are a little behind schedule and need to pick up the pace a bit" when the realities have been that the project is in terrible shape and needs a minor miracle (and I'm a consultant, not a miracle worker).

I've also been hired at the beginning of a project to work on specific areas where I am an expert. This is a good use of my time, and an example of how to best use consultants: hire them to help put in solid approaches up front (especially in this tight job market where it is difficult hire period).

At times, judicious usage of consultants can result in getting a solid contribution to a project. My rates are high, but so is my productivity and ability to understand how to put software together.

Most important is to be able to differentiate between consultants and contractors; consultants have specific skills and experiences which can greatly speed a project along. For example, I am extremely fluent with VxWorks, device drivers and putting in base-lines which large groups can work from, and from this, feel as though I bring a great deal of value. Contractors are workers; give them a task, they should be able to get it done, but may require extensive management.

When evaluating whether to use consultants, some of the following characteristics should be taken into account:

- Budget versus time to market. Consultants may appear to cost more than regular employees (this is debatable, as amount of work produced may actually make a consultant more economical), but if time to market is the critical need, consultants can probably get you their faster and more reliably.
- Daily support needs. Consultants tend to be able to work very independently, not requiring extensive management. Again, if time to market is essential, the less effort required to manage a group helps everybody.

If you do decide to use one more consultants, it's wise to evaluate them on the following:

- What are their documentation skills. This is more important than with regular employees as consultants tend to move around. You want to ensure that your investment is well protected in the form of a lot of documentation. Remember, you are hiring them because they are experts in certain areas - you want to retain that expertise long after the consultant is gone, and a good way to do this is to make sure everything is written down.
- What are their people skills? What good is an expert if you can't communicate with them, or if they are unwilling to work with other members of your staff?
- What does their background look like? A good consultant should be skilled in many different areas, and able to work well in each of these areas and switch gears without problems.
- How long have they been consulting? Consulting is an art form. Good consultants try to avoid getting politically involved in a company unless specifically asked to, and learning this characteristic takes a few years (I know first hand). In today's market, I have seen people quit their regular jobs and hang up the shingle of "I'm a consultant" - they are not; they are contractors.
- What are their rates? If they are afraid to ask for significant amounts, then they are not sure of themselves; you are paying for their education, which they will take with them. A good consultant continues to be educated, but a true consultant comes to an organization to quietly educate (without appearing to) while performing their tasks.

Pitfalls to avoid:

- Don't let them get you into a tough situation where they hold your projects knowledge, as well as the keys to completion. A skilled and experienced consultant walks away each week as though it's their last (all documented, all work clean and ready to use by others), while a game player attempts to manipulate the situation to where they have a political edge over you. The professional stays on a project because the work is so good the organization wants to keep them, not because they have to.
- Don't wait until it is too late. If you see a problem coming up, or you need some specific skills on the project, sooner is always better than later (I was once hired a month before a organization was planing to ship. After two weeks I said "January 15th, not July 15th." They thought I was crazy, and the manager called me on January 6th and said "you were wrong - we shipped today, sooner than you said.").
- Arrange financial terms that are in your best interest; a consultant who is allowed to bill once a week, and is paid within a week, seldom has much to lose by walking out, while the consultant who is required to bill in monthly intervals with 30 day terms generally has a better incentive to exit cleanly, as they can have up to two months money riding on their decisions on how to exit. Let them carry some risk; after all, you are taking a risk that they are truly worth the money they ask for.

3.1.5 Technical Lead versus Group Management

TBD.

3.1.6 Education and Training

Education and training can be essential tools for an organization. If you have hired people who are capable of learning, this is a great place to apply this. When evaluating training, look for areas that complement your staff current skills. For example, if you are going to be using a microprocessor or configuration management tool which your staff has not used before, getting some solid training can be a win.

As I traveled this country and went to Japan doing my technical seminars this past year, I started to see that successful companies such as Cisco are successful because not only are the organizations open to training, but so are the team members. In contrast, when working with clients who have had many problems, their team members were plagued with the arrogance of “I don’t need training - I’m a very qualified engineer.”

This brings us back to staffing and hiring; are you hiring arrogant people who don’t feel they need to continue their education, or are you hiring people who know that they have good skills but are open to learn. Skills are an evolution, and it’s better to get training when available versus to learn simply through “screwing it up” a few times.

3.1.7 Dealing With Unproductive Team Members

This is an extremely sensitive and difficult area for any organization. I tend to think that the issues should not be about dealing with an unproductive team member, but in ensuring that they are not hired in the first place.

However, there are people who slip through the checkpoints. They are often political players with huge egos, and their true skill is marketing (you did hire them, right?). Marketing people don’t belong in engineering, and engineering people belong in marketing only after years of evolution.

The biggest pitfall I have seen over the years when dealing with team members who present problems is that they are not dealt with. They are allowed to stay involved in a project, and through this, they bring everybody down. Yearly reviews can help get the message across to these employees, but often a year is a long time to wait for an aggressive start-up with significant “time to market” issues.

Additionally, this is the “age of litigation” so the wise manager moves with caution. No matter how you deal with difficult situations like this, I can only state with clarity that one bad apple can bring the entire group down. What I have seen which works is a repositioning of these people, giving them a title and job description of “researcher for the next generation” and then isolating them. Ideally, they will figure things out and move on without much fanfare.

Most important is simple: deal with it.

3.2 Requirements and Engineering

All good projects come down to understanding the path to follow, and the goals you are trying to reach. All good engineers need to understand where they are going, and the fundamental basis for this is a solid understanding of a project’s requirements.

3.2.1 Having a Clear Path

Having a clear path is about having a documented set of directions which clearly state not only to the engineers, but anybody else who might read the documents, where the project is going, what the project's objectives are.

I have noted the following key components of a clear path:

- A set of clearly written requirements. These requirements state more than the obvious: the end goal. They state that the group understands clearly what they are doing.
- A detailed and agreed upon schedule. A schedule represents that a design and set of requirements has been broken down into components, and that each of these components has either an owner, or represents the need to bring on staff.

Mostly, a clear path avoids the traps that come from the “let’s just start, we will figure this out as we go approach,” an approach that only works well on research projects, not in production engineering. Remember, if you plan on shipping on time, then you are already in production, and should be well past the research phase.

3.2.2 Having a Clear Design Philosophy

Clear design philosophies can be extensive, and vary greatly from group to group (such as the differences between the “object oriented” camp versus the “simple state-driven” camp). Most important is to have consistency and clarity. No matter what your group's philosophical views, I have always noted the following elements of a successful group:

- Documentation standards that are agreed upon, and adhered to.
- Coding standards that are agreed upon and adhered to.
- An absolute belief in unit testing of modules, and a verification means for this.
- Code reviews. A walk through of all modules as they pass through the unit testing phase. These reviews really do catch some very obvious errors which are made by even the most experienced of programmers.
- Keep it simple. The nature of the software business is that unexpected complexities arise, and if your design is overly complicated from the start, it's only going to get worse.

In general terms on philosophy, I can state with great clarity that it's not what we understand which is the problem, it's what we don't understand. Software projects tend to be projects where education is ongoing, and with each milestone, we start to understand the complexities even more so; if we start with a complicated design, then the complexities we encounter during implementation are only going to be much more complicated.

A good rule of thumb is “design to debug.” When you use this philosophy, you find that you try to keep your designs simple, as each time you start to think about how you are going to debug the project, you step back and say “can we make this any simpler than what we are doing?”

3.3 Scheduling and Execution

Scheduling and execution is not always a management issue. The entire group needs to be involved, as the people who do the real work are omnipotent. What they see and start

to understand needs to be communicated clearly to others in the group and management, as any good software project takes on a life of its own.

3.3.1 A Living Schedule

A solid schedule to start a project is without question essential; it is a stake in the ground which states that great thought has gone into what the software project will require. But in a sense, a software project is about education, and it's important that the schedule be updated often to reflect the current understandings.

Remember, schedules are not about imposing directives; they are a management tool to evaluate the current status of a project. My favorite quote on scheduling comes from a status meeting I was in a few years ago. The project manager said the following: "well, I see we are in about the same place as we were when we made this schedule last week."

No progress. In this situation, the schedule in question did not reflect what needed to be done; it reflected what management felt should be done. In this situation, the schedules we generated did not have a great deal of use, as they were not accurate. The team members were giving answers they felt management wanted to hear, and management was unable to track the progress of the project because the schedule did not reflect the realities of what implementation was requiring.

There are two forces in scheduling; the dates that management feels need to be met to meet customer goals, and the dates engineering feels are a reality to actually do the job.

When engineering starts making up dates to keep management happy, management has lost control of the project. When management starts using dates which are not based in reality, customers get let down.

Ongoing and periodic review of the schedule can be a valuable tool; it gives management the ability to truly see where they are at. If schedules aren't being met, it's for two reasons:

1. Things are more complicated or involved than originally anticipated (lack of a clear understanding of requirements).
2. The engineering group is not being honest with themselves, or their managers, about what they are actually able to accomplish (perhaps rooted in originally hiring the wrong group).

What I have seen which can work, provided that there are open minds, is weekly reviews of schedule which are honest; if there is a problem, look at it, try to solve it. Hiding behind an unrealistic schedule is a problem that manifests itself in low moral among employees, lack of confidence by management, and most important, extremely irritated customers (and if you don't have customers, how important is a schedule?).

3.3.2 Unit Testing

Unit testing is one of the most important but most often neglected components of a successful software project. Every software module is a foundation on which the entire project is built. Build a weak foundation and you build in a recipe for disaster.

Unit testing is also one of the first things to go when working with tight and unrealistic schedules. I once was hired on job where they told me up front that they decided to take the risk of not doing unit testing in hopes that “things would come together” at the end. And in the end, things did come together. After they had gone back and done unit testing.

Pay now, pay later. With software, the price only goes up with time. It’s not initial generation of software which costs money - it’s maintenance. Without solid unit testing, the foundation is weak.

Unit testing must be a strong religious belief among all. Unit testing is about making sure that you never have to revisit this portion of the road again. It is about “the definition of done” - in the end, it’s “done” when all of the modules work. Ensuring that each module works along the way is just an insurance payment that integration is going to be about that final success story versus the start of the most frustrating experience you have ever been through.

There is nothing worse than, during integration, spending a week debugging a problem only to find it was a module that was done months ago which was weak. When this happens, you start to distrust the entire code base. You start to believe in voodoo and magic. You lose hope - if one little piece from months ago is not sound, how sound is the rest of the system?

Putting the modules aside during implementation as truly “done” ensures that integration is reaching that final goal, not about starting on a path of concern, worry, and inability to schedule and manage.

3.3.3 Integration

TBD

3.3.4 Configuration Management

TBD

3.4 Getting Started

Getting started is hard. I had thoughts about this article for months, and when I started it, I felt it was going to be a couple of hours of work. It has not been “a couple of hours” of work - it has been a big job, and at the time of this release, is not completed.

There is nothing better than getting started at implementation; it is the key to reality. As you start, you start to realize how short time is, how much work there is to do. It reinforces all which is written above, as it helps make schedules honest, helps solidify understandings.

Wallowing in design is a terrible thing; good design is important, but so is getting the job done. If you don’t start, you will not finish. And if you don’t start, but wallow fully in the design process, you will find yourself mis-understanding how complex your project is. Starting to implement is starting to simplify, as with each step towards implementation you truly understand how important it is to keep it simple.

4.0 About the Author

Information about the author can be found on the following web site:

<http://www.tbcorp.com>