RAD RAPIDS

Meaningful Metaphors for Software Development

Gerhard Beck

1 The River Runs

Water flows downstream in its journey to the ocean. Both time and water move only in a single direction. Unlike time, water can slow down, speed up, pool, whirl, mist, and evaporate. Water has its own magic cycle which time does not. Along its journey to the lakes and oceans of our Earth water changes. It evaporates and travels in all directions. Eventually, it finds its way back to the Earth to start its downhill journey anew.

So it is with developing software systems. The classic metaphor harkens back to the waterfall, but we can use water to provide a richer set of metaphors by which to govern our software development efforts.

Methodologies are not a cookbook which can simply be followed to get the desired result. They serve as guideposts to a journey. The journey is made by programmers and other related artisans at the behest of their business benefactors. Therein lies the first conflict: for the programmer the joy is in the journey but for the sponsor the joy is in the destination.

Yet both are bound by their disparate talents. The programmer can not take the journey without the businessman's sponsorship and the businessman can not reach his destination without the programmers travails. If the journey is jointly taken by both parties, they may be able to achieve their respective goals in a mutually advantageous way.

However, there is a delicate balance between them. Without the balance, the journey will enter a swamp and may die stuck hopeless in quick-sand. Even with a balance, marauding influences may yet stop the journey, but that can not be helped.

Here I set out my guideposts for the journey of software development.

2 The Rapids

Let us start the journey back in our software development 101 course. There we were introduced to the classic waterfall paradigm for software engineering:

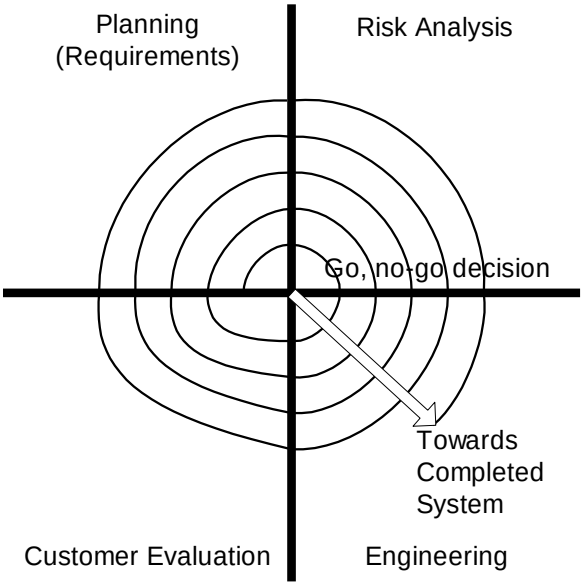
Analyze – determine the requirements
Architect – determine the environment the system must run in
Design – determine how to build the system
Code – build it
Test – see if it works
Deliver – deliver it

Maintain – keep it working

In a perfect world where nothing changes and all parties know exactly the desired results, this is a good model. You can view something akin to this in an automobile assembly plant. There the assembly line flows through the channel of a vast building on a journey to an end which both workers and their sponsors agree on: a car to be sold.

Our journey does not start with such detailed knowledge of the end. Our journey is more one of setting sail across an ocean to a distant place which can only be recognized once it is seen.

One could view our craft as building custom armor to be used in the wars of business. To do this, we build the armor piece by piece carefully checking the fit and finish before moving on to the next piece. This is the basic truth to Barry Boehm's spiral development model which uses a type of iterative waterfall with feedback to build the armor. The effort turns into a whirlpool from which the system will finally emerge.



In the spiral model, initial requirements are evaluated against risks to determine what will be implemented in an initial prototype of the system. The prototype of the system is then evaluated by the users. User evaluation is used to refine and expand the requirements. Those requirements begin a new cycle adding refined and expanded functionality to the prototype.

A major advantage of the spiral model is that user input and evaluation is spread over the life of the system development effort. This allows the system to be evaluated by the users at an earlier time, allows the system to be changed based on user feedback of an actual product, and allows the system to evolve as the user's business problems evolve and change.

Our sponsor may approve of a drawing of the suit of armor (or a storyboard of the software system), but the real feedback comes when the suit is tried on. Then we find the rough spots, weaknesses, and kinks which must be corrected before the battle is fought. With each fitting, the suit becomes more complete, more comfortable, and more of a weapon for its wearer.

If we had but a single armorer, this model alone would be sufficient. But we typically have several working together to build the suit. Here is where we get to the rapids.

Once each armorer has a common understanding of what the overall suite of armor should be, each may begin working on a component part. The leggings may be completed even before the chest measurements (requirements) are taken.

More formally, an overall architecture with general requirements is established. A core is created and then the system emerges from as a continues asynchronous series of miniature spiral development projects.

When viewed from afar, the general course of the river will be obvious. The details will emerge from an ever changing forth as the whole effort flows over the jumbled rocks of evolving business requirements. Each participant will guide his own kayak though the rapids towards journeys end. Certainly, software development managers can relate to the effort required to keep a flotilla of kayaking programmers together in a journey over the rapids.

Come, ride the kayak with me through the software development rapids.

3 RAD Rapids

Rapid Application Development ("RAD") describes what the project sponsor truly wants: to complete the journey as fast as possible. RAD is a loose term vaguely describing how to speed things up, typically through reducing the feedback cycle by producing demonstrable code at the earliest possible time. A classic RAD approach is the GUI building IDEs which can almost magically create simple database input and reporting applications.

It has been my experience that minimizing the feedback cycle is absolutely critical to a project's success, so I happily take on the RAD moniker. To that, I add the wisdom that a project is but a thousand small projects resulting in the metaphorical rapids. Hence, I refer to my methodology as "RAD Rapids".

4 Core Tenants

The core values for implementing RAD Rapids are: Honesty, Understanding, and Flexible Simplicity. Passion will hopefully be the key by-product.

4.1 Honesty

Honesty is the cornerstone to successful planning and execution. All parties need to be as honest as possible. The biggest areas where there is often less than honest communication I have seen are: (i) development effort projections, (ii) progress reports, and (iii) deadlines.

Projections

Projections as to when the product will be done or when certain features will be ready. Often this starts with the sales cycle. Projects are often sold with overly optimistic projections which often amount to wishful thinking. The initial estimates often do not include time for the inevitable

changes and project growth. It is vital that projections be made as honestly as possible and that they be updated as often as possible. Projections should be used as a tool for deciding how to proceed not as a guarantee of performance.

To honestly guarantee a certain delivery on a certain date, either the system must be a simple one-off of something already done (which I've never been involved in as an IT professional) or a huge fudge factor must be built in. In that case, an absolute minimum performance should be guaranteed, with a best-efforts to add additional features. Ideally, the minimum will be hit early and the large fudge time will be used to add features.

Construction provides some apt analogies. Start with an automobile assembly line. Once it is in place, cars can be built on an extremely predictable schedule unless some significant outside force intervenes. Home construction tends to take a reasonably predictable time to build – once the plans are complete. However, because of the numerous scheduling and environmental factors, home are rarely completed based on projecting the time required.

Software projects do not resemble an automotive construction line. Software projects more closely resemble construction a custom home. But the custom home process starts with a detailed set of plans where as software projects tend to start with only a general description. No wonder its so hard to project software construction efforts.

In fact, it is theoretically impossible[1] to create accurate estimates at a projects early stages. As Steve McConnell says "Early in the project you can have firm cost and schedule targets or a firm feature set, but not both."[2]

Progress

Once projections are made, they are often used as performance guarantees, not as planning guidelines. This results in a lot of pressure to meet the projections. This will often cause progress reports to be less than fully honest, to be filled with puffery, and occasionally to be outright deceitful. Often, number will be fudged in the hope that they can be caught up at a later time either through future long hours or by magically going faster in later stages. I have yet to see either strategy work so I don't recommend them.

Not meeting the projected schedule is bad news. Bad news does not age well. Regular honest reports of progress are crucial to a project's success and survival. The projection should be used as a yard stick against which progress is measured. If the progress reported is too slow, then some corrective action should be taken (preferably other than telling the programmers to give up their personal lives).

One key item in measuring progress is to break the project down into small enough bites so that progress can be measured on a weekly basis. The bites must be Boolean in nature. Saying that a feature is 90% complete is not helpful – especially if the part left undone is the nearly impossible part. All progress reports should indicate one of three things: not started, in process, completed. What's in process this week should be completed next week. Of course, weekly deadlines for new projects are hard to determine at the beginning of the project – which is why scheduling should be an on-going negotiation not a one-time event.

---

[1] Software Project Survival Guide, Steve McConnel, Microsoft Press, 1998. P 32.
[2] id. p33.

Deadlines

Projections and progress are typically places where the programming folks are less than honest. This is often driven by the deadlines imposed on them by the sponsor of the development effort. Of course the sponsor want the system sooner rather than later. Sponsors often try to impose artificial deadlines to try to speed things up or to see if they can get at least something working (since so many projects are failures).

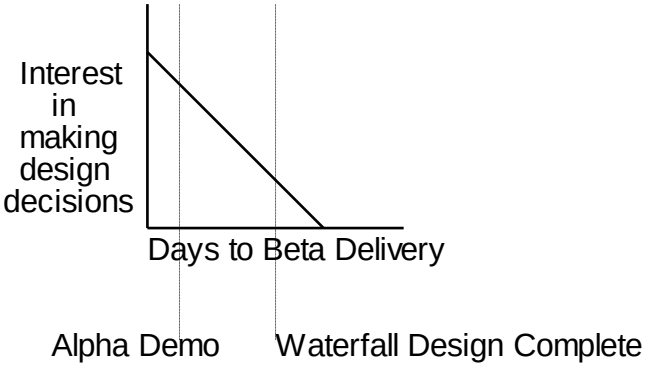
The two best ways to encourage honest deadlines is with honest projections and progress reports. Being able to demonstrate weekly progress on the customer's key concerns goes a long way towards getting the customer to be realistic in setting deadlines.

4.2 Understanding

Understanding requires communication and feedback. Communication has not occurred until there has been feedback. I would rather get a document back with lots of changes than no changes.

Getting good feedback requires some enthusiasm on the part of the folks providing the feedback. Reading long boring documents is a real enthusiasm killer. Playing with an initial prototype or reviewing storyboards engenders much more enthusiasm. In general, a mix of techniques is required for full communication because the needs of the client's business and technical folks diverge. But the simplest advice is to provide lots of small opportunities for feedback instead of a few big opportunities.

To me, there is an underlying human equation driving RAD. The further the user is from seeing or playing with the system, the less interested the user is in making real decisions. Consider a project where there is a six-week design effort. If the user has to wait six weeks to see the results, the user will not be keenly interested in making decisions. If the user's decisions today will be reflected in a document and storyboard distributed next Monday with a mockup next Friday, the user become much more animated and involved. Once shown the results, the user becomes even more involved in correcting and amplifying the design.



Note that true understanding can only be based on open and honest relationships. Dishonesty and closed environments can quickly destroy the mutual understanding between programmers and users which is essential for successful projects.

4.3 Flexible Simplicity

Honesty and understanding are human values necessary for rapid development. Flexible simplicity is the technical virtue required. I have found projects work the best if they are designed as simple as possible and as flexible as possible.

Simple in this case means that the software does the absolute minimum required in the easiest way possible. Flexible means that the software can be rearranged and extended easily. Often the two go hand-in-hand because striving for flexibility will force the project to be simpler.

In general, I have found the projects are fairly stable at a big picture architectural view and are fairly stable in the low level routines. All the routines in between, the user interface in particular, change constantly. Building the system to a general architecture based on lots of well-built small components which can be re-arranged at will provide for the flexibility to change as needed.

4.4 Passion

A group will become passionate on a project if they share the same vision, if they believe in each other, and if they believe they can be do it. Building a common understanding among all parties, clients and artisans, will supply the vision. Honest communication will help the group to believe in each other. A simple flexible design and architecture will convince the team that they can do the project.

Passion should provide both determination and diligence. Determination is the ability to take on difficult tasks while diligence is the ability to do the mundane tasks. The key to both is the belief that the task is necessary and that the effort will eventually succeed and be useful.

5 Key Practices

This section covers the practices which I have found most helpful in my projects.

5.1 Build to Budget

Build-to-Budget is the most fundamental precept I have. Basically, the customer comes to you with a concept for a system and a budget. This reflects the reality that projects are usually approved at the concept stage, not after the detailed requirements and architecture have been established. The team (client and artisans) must then determine the basic requirements and delineate an acceptable architecture. Once that process is complete, construction can commence.

Because there are innumerable variables in going from concept to reality, the initial amount budgeted only loosely fits the reality of what the system will take to build. When the budget runs out, either there will be something of value or there won't and the project will either receive more funding or it won't. If there is nothing of value when the budget runs out, the project is not likely to get additional funding and will be a failure. If there is something of value when the budget runs out, then the project can be deemed successful and additional funding for enhancements is much more likely to be forthcoming.

Project Parameters

The key to providing something of value is to keep the concept in the contract as simple as possible by outlining general objectives. The tactical implementation of the concept will be done by estimating all the things which could be done and then selecting those which will allow the project to be built within the budget.

This is akin to building a house for a certain budget, then determining between simple light fixtures vs. chandeliers and hard-wood floors vs. linoleum to get the house completed within that budget.

To successfully build to budget, you need:

- A complete list of requirements (items needed in this version) and desirements (items which the user would like in some version). Initially there is only one lists since only the budget and schedule separate requirements and desirements.
- Priority of each requirement
- A task list
- Cost estimates for each task
- A dependency table

The complete list of everything the user could ever want the system to do helps to ensure that all fundamental requirements are known up front and what the shape of future enhancements might take. For consultants, this also provides opportunities for up-sales (contract for future enhancements) and value-adds (items delivered beyond the original contractual obligation).

Note that the basic difference between a requirement and a desirement is what can be afforded in this version. Obviously, this can only be determined after the complete estimating cycle. Hence, its hard to know if any specific item will be when gathering the requirements/desirements from the user community.

The requirements list forms the basis for the task list. The task lists convert the requirements into technical items needed to realize the requirements. For example, interfacing with a database isn't likely be on the user's requirements list, but the system won't work without it.

Requirements drive the system value. Understanding the priorities of various requirements drives the entire build-to-budget process.

Tasks are needed because that is how the system is built and tasks drive the cost equation.

The purpose of the cost and value is to provide a ranking so that the most valuable features are added first. The dependency table indicates what the order of build must be – it may not be possible to implement a crucial feature until other features and tasks which it depends on have been implemented.

Understanding a project's cost/value parameters allow for a determination of the what the minimum to provide a core system should be and how that core can be enhanced with additional functionality.

Constant Improvement

Build-to-budget requires a constant fine tuning of what will be delivered and when. The process of refining requirements as the schedule advances is greatly helped if the client can see meaningful progress each week.

Obviously, meaningful progress differs in each stage. Early on, progress is meetings and papers. Soon that should turn into storyboards and GUI demos. Technical experiments can be demonstrated. Once an operational core is completed, that can be demonstrated and then enhanced in a thousand different ways. There must be progress to be reported and demonstrated weekly.

One way to show constant progress is to break the project up into lots of little sub-projects. Then there can be significant progress shown on the various sub-projects while the overall project inches along to completion.

Demonstrating constant progress on the client's key requirements breeds trust and trust fuels the entire build-to-budget process.

Adroit Scheduling

I have never made a project plan which survived even a week, much less the term of the plan. Since project plans never seem to work, I tend to rank what must be done by importance and assign the most important things first. Sometimes less important things may be done in conjunction with an important thing because the features are expressed in an interrelated code base.

What is most important can also vary. The software environment into which the software will be loaded tends to change and evolve along with the software. The client's business priorities can also change as the project progresses. Thus, the list of important items is subject to as much change as a project plan (but is easier to maintain).

Since change is a given for most projects, build it in and schedule accordingly. The basis of adroit scheduling is a weekly project and schedule review. During the review, the following should be discussed:

- Risks – items which may adversely affect the project
- Open Items – items which are needed from the client
- Accomplishments for last Week – items delivered and functional items completed
- Goals for this Week – what the plans are for the next week

Adroit scheduling seeks to involve the client in the process of determining what is important. It also provides a forum for highlighting problems which need to be managed and provides for client input on how to manage the problem.

Risk identification and reduction is a key management task. When a project starts, there will be many risks. These should be identified. The initial items scheduled should be to reduce the areas of major risk. For example, if a new technology (or tool) is to be employed, an early task should be to see if the technology will actually do what it is supposed to do.

Whenever a new technology or technique is involved there is a risk and a need to experiment. Design is only possible once the components of the design have been used and tested. This accounts for one of major problems in software estimation – if you've never used a technology before, you don't know what problems you are likely to encounter and how long things will take. If I could amend the classic waterfall method, it would be to change it from analyze, architect, design, build, test, deliver to analyze, architect, experiment, design, build, test, deliver.

A basic wisdom is that there is no magic bullet. New technologies may speed a project up, but they rarely live up to the vendor's claims. Until a technology has been used, it hard to know how useful it will be.

Adroit scheduling requires an appropriate ramp-up for a project and a reasonable end time. The ramp-up means that programmers aren't brought on before there are any requirements or architecture. Once a core group determines the architecture, that same group can test the tools and technologies then build a basic deep-slice prototype to show that the architecture will indeed work. Then, more programmers can be brought on. Care must be taken that the learning curve for new programmers does not slow the overall project down.

Note that documentation can greatly assist in ramping up new programmers without slowing down the existing staff. If fact, that should be one of the major goals of the architecture and high-level design documents – to bring new programmers quickly up to speed on the overall project plan.

This if reflected in Brook's Law: "*Adding manpower to a late software project makes it later.*"[3] The basic premise is that a project can only accommodate a certain number of workers at any given state. Adding additional workers to try and meet a schedule will only slow the process down because of the added needs for communication among workers. This places a threshold on short a schedule can be made. This minimum time threshold is often expressed in the quip that nine women can't make a baby in a month. As Mr. Brooks states, "More software projects have gone awry for lack of calendar time than for all other reasons combined."[4]

5.2 Replaceable Service Driven

Visualize the application as a host of services which interact each other under the direction of the GUI. (Can we say "Model View Controller"?) Imagine that you compartmentalize the logic along functional lines (perhaps in Java packages) and then only provide access through a limited API. Further suppose that each package is designed so that its underlying technologies could be substituted at runtime. (Can you say "Factory" pattern?) Such is the nature replaceable service design.

One of my projects was an application which relied on a thesaurus. The thesaurus we were using came from Princeton University and there was no requirement or reason to believe we would ever use another thesaurus (the application was built around this particular thesaurus). However, rather than directly access the low-level APIs in the GUI, we created an internal thesaurus API. That API more directly serviced the GUI and made it possible to substitute a different thesaurus. Well, as I'm sure you've guessed by now, we were later asked to by a different client to use a different thesaurus. We adjusted our internal API so that it could simultaneously use both thesauri and picked up a rather large contract as a result.

[3] The Mythical Man-Month, Frederick P. Brooks, Jr., Addison-Wesley Publishing Company, 1972 reprinted 1982. p 25
[4] id at 26.

Making your code into a set of services makes not only replacement feasible but also refactoring. Refactoring is simply recoding a module for whatever reason. With a limited, fixed API, it is possible to develop a test suite for the module. With a test suite, you can ensure that the refactored module works the same as the original module.

When would you want to refactor a module? Lots of reasons: speed, maintainability, code size, memory footprint, etc. But usually the problem is maintainability. Building something the first time is hard and tends to produce inelegant (i.e. bad) designs because the designer/builder does not know any better way to do the design. Come back to the code a month later and there will often be some obvious ways to improve the design. That is, design and coding is like a sport – you learn by doing. Having done something once, you can likely do it better the second time. Doing the same thing a second time is refactoring.

Besides replacement and refactoring, maintenance is a key to replaceable services. Simply put, a limited API to functionality aggregates the code so that the implementation changes are hidden from the rest of the program. (Can we say object oriented?) An API layer of interfaces and classes allows for greater flexibility in changing the lower level functions without affecting the rest of the program.

This notion of replaceable services also fits with the latest ideas for XML based services using SOAP, UDDI, etc.

5.3 Shared Code Base

A simple notion really. Once code has been checked in, the group owns it and anyone can change it. Of course the original coder would be ideal for making changes, but anyone should be able (and willing) to go in and fix any code.

Reaching such a blissful state requires some group think and a supportive atmosphere. Specific techniques on the road to group coding nirvana follow.

Concise Vision

For all the programmers to be able to work on any part of the code, it helps if they all share the same vision of what the system is. I use two techniques for this: the one picture architecture and the desktop ERD[5].

The one picture architecture is just what it says – the whole system reduced to a single diagram. I use this diagram for selling/explaining the system to the client's upper management and to quickly demonstrate what's involved to programmers. The single picture is important because people can remember a single picture fairly easily. That picture represents the forest.

The desktop ERD is a database design printed on 8 1/2 x 11 paper with a logically related group of tables on each page. This allows the programmers to understand the underlying data model and relationships upon which the system is built. This needs to be on paper so the programmers can keep it on top of their desks, highlight it, edit it, and use it for discussions.

[5] Entity Relationship Diagram which shows how the tables in a database are related to each other.

Version Control

Version control is an absolute must for any project with more than one programmer. For most groups, the version control should be simple enough that no one needs to become the version control expert. Version control should allow multiple programmers to work on the same file at the same time and deal with any conflict at the time of check in. The open-source CVS[6] is my recommended version control system.

Desktop Integration

Every programmer must be able to build and test the units the developer is working on. Desktop integration is something more: the ability to build and test the entire system from any developer's desktop. This allows the developer to ensure that his changes will not cause the system to fail to compile (a big no-no) and that his system will work in the context of the overall application as well as in its unit test harness.

For this to work, each developer should use the same techniques for building and starting the application. I typically create batch files or shell scripts for this purpose. These are maintained in the version control system so that each programmer always has the most up to date compile cycle. This makes the process easier to see and maintain than using the services of an IDE.

I have had two problems with an IDE based compile cycles. The first is getting a standard project definition which can be checked into and out of version control easily. The second is that I don't require developers to use the same IDE or editor.

Coding Standards

A group must have a common coding standard so that each programmer can easily understand the other programmers' code. Coding standards should strive for clarity, should be simple enough to be easily remembered, and should highlight potential problems. I tend to document coding standards in peer review checklists (see my accompanying papers).

Pair Problem Solving

Pair programming is having two programmers share as single computer to code. This is a controversial part of Kent Beck's Extreme Programming methodology. I don't like pair programming because I believe that programmers tend to work best and fastest on their own. In particular, I believe that programmers get a great deal of satisfaction from devising their own solutions.

Pair programming is a vital tool for solving difficult problems. If a programmer comes to me with something they are stumped on, I have them check in the code, bring it up on my machine, and then we work through it together. I have found this to be extremely effective, especially if I keep up a running banter of what I am looking for and why I am making the changes I am making.

In fact, I solve other problems the same way – by assigning one programmer to help another with a task. I have never actually called it pair programming, but it works out that way. On my last project, I'd estimate about 10% of the effort was expended this way – usually the hardest 10%.

---

[6] http://www.cvshome.org/

Peer review

Peer review is an excellent way to improve the quality of the code produced and to enhance overall system quality. Unfortunately, I have never yet come close to having even half the code on a project peer reviewed. Even if you can't hope to peer review everything, having some peer reviews will greatly increase the quality of your code.

I don't actually see the enhanced system quality as the main benefit of peer reviews. The main benefits I see are:

- Peer review trains new programmers on the group coding standards and customs.
- Peer review teaches new tricks to old programmers.
- Peer review enhances the feeling at the group owns the code.

All of which will eventually help the overall quality of the code.

The technique I use is to have the team lead (me) code some critical system function and make sure it completely follows the coding guidelines. That becomes the first code peer reviewed. It shows the group how I expect things to be coded and makes sure that the critical function is coded well. More important, it sends a signal that the code belongs to the group since no one's code (not even the team lead's code) is above the peer review process.

I then select a module from each programmer for peer review – hopefully soon after the programmer begins working on the project. This gets the programmer used to having the group inspect his code and with the idea that the group really owns the code.

Most importantly, because the programmer never knows that sample of code will actually be peer reviewed (and because the stated goal is to peer review all code), each programmer tends to code closer to the group standards than if there never were any peer reviews.

I actually prefer to start the entire process by passing out the peer review checklist (or other coding standard) at the first meeting with the programming team and then ask for input on how the checklist/standards should be changed to accommodate this group and project. Allowing programmer input helps to provide buy-in.[7]

5.4 Open Project

A project should have a web site which contains everything there is to know about a project. Specifically:

- The project's charter/mission
- All of the weekly status reports
- Minutes of all meetings
- Results of JAD sessions
- All documentation
- Storyboards/Screen demos
- Current prototype
- Installation instructions
- Links to get the source code

[7] I actually tend to be a consensus despot. I try to build a consensus, but where that fails or where it deviates from something I think is really important, I decree a solution. Just don't decree to often.

On my last project, the client insisted on beginning the source code transition in the second week of the project! The result was that we opened up our CVS repository so that the client maintenance programmers could download and play with our source code at any time during the development effort. While I had great trepidation about this, it worked out very well. Since the client's programmers could see where we were on any given day, they had more confidence in our status reports. Since our management knew the client was looking over our shoulder, they were more honest in their status reports. In the words of Ronald Reagan, "Trust, but verify".

In order to open the project up with a web-based bulletin board, you actually have to create documentation for the project to place on the web site. The technique we found most effective is to use Adobe Acrobat and print all documents (and reports from case tools) in PDF format. This makes the documents universally available.

5.5 Just in Time Requirements

One of the biggest problems in any software project is changing requirements. While programmers would like to have all requirements up front, that is a dream world.

A better idea is to get as many requirements up front as possible and use them to derive the fundamental requirements of the system. Don't worry that all the details have yet to be worked out. Use those fundamental requirements to create the system architecture and divide the system up into a set of smaller projects. Then get the detailed requirement for each smaller project just before starting that smaller project. This results in a lot more focus.

Requirements are not a one-time affair. Programmers will always have questions about requirements. It is key that a user representative who is seated with the programmers. That way the programmers can really get requirement just in time for their code.

I prefer a business analyst over a real user as the user representative. A single user will tend to give the requirements an idiosyncratic flavor. A business analyst will blend the requirements from different users and from different types of users.

The business analyst as user representative is the perfect person to set up the test scenarios and supervise the testing. After all, the ultimate test is how well the users like the system.

5.6 Integral Staffing

It is critical to have all the talents needed for the development effort on the team and to have them seated together. In particular, someone on the team needs to be able to perform system administration functions and DBA functions. These can be programmers with a particular interest since they can fall back on the full-time system administrators and DBAs. However, the team needs at least some integral representative to take care of the day-to-day problems which come up.

Apprenticeship is important in maintaining your integral staff levels. For every function on the project, you should have two persons who know the basics. If you have only one person who knows the function (or tool), then he is the master and someone else must be appointed as the apprentice/understudy to learn the function. That way the team has a backup for vacations, reassignments, time crunches, etc. This also applies to such people-oriented positions as

manager and team lead. The best success criteria for these positions is if the team has been trained well enough to carry one without them.

5.7 Work Space

There are many different opinions on appropriate workspace. One train of thought puts programmers in their own offices with doors. In these places, you often wind up with team meetings in the hall way (often described as fun sports events).

Clearly each programmer needs a minimum amount of space. This should be enough for the computer screen and keyboard, a separate writing area, and some drawers. Every programmer should also have a white board.

Having been in offices, full cubes, short cubes, and one big room, I find that the arrangement I prefer is a set of short cubes surrounding a conference table with a white board. Short cubes allow programmers to overhear each other. While this can be a distraction, it is more often a good way to encourage group involvement in whatever the latest problem is. Also, programming can be tedious so hearing other folks is not such a problem. Most of my programmers put on head-phones and listen to music when they need to concentrate.

The short cubes also tend to eliminate the need for most meetings. When an announcement is needed, I just holler for everyone to stand up and the announcement is made. Often a bit of socializing follows since concentration has been broken anyway. When a design meeting is needed, it's called the same way and is held at the common table with the white board. So the entire team gets to hear the meeting, not just those invited. Obviously, some private conference room must also be available for conference calls and private conversations such as reprimands (which are hopefully few and far between).

5.8 Success Driven

A project will run the best if it appears successful. A project will appear successful if it is going forward in a reasonable fashion.

Forward progress is generally measured by accomplishing various tasks. Breaking the project into lots of small tasks and working on the most important tasks first brings a feel of success to a project.

One quick way to lose the feel of success is to have to do lots of rework due to changes. While moving requirements closer to code will reduce this, there will always be changes especially around the user interface.

The best way I have found to minimize the impact of these changes it to focus on where the changes usually occur: in the middle. That is, the overall system architecture does not tend to change much over the project's life. Many of the sub-routines (fundamental classes) needed to actually drive the system also tend to be fairly stable. In between is the mass of the system which aggregates the fundamental classes into the system described in the architecture/high-level design. Those aggregation classes tend to change more than the rest.

So I challenge the programmers to design the fundamental classes in ways which will make it easy to rearrange in light of changing circumstances. In that light, having to rework a part of the

system for a changed requirement is a test of the code's flexibility, not a useless waste of resources.

Even if a project is going forward, it will not feel successful if that forward progress comes in an unreasonable fashion. Specifically, if forward progress is being made by heroic efforts (long hours) on the part of some or all of the programmers, then the project will feel out of control, not successful. Therefore, you should strive to maintain a 40 hour week for all involved. A week or two spurts before a major delivery may be needed, but deviating from a norm of a 40 hour week for more than a few days will begin to adversely affect the team and infect the project with a death march to failure mentality. (OK, we actually average between 40 and 45 hours a week. But that's still way lower than most other groups in our company.)

6 Journey's End

So how do you tell if the journey was successful? I suggest three criteria:

1. The system goes into production.
2. The client wants the team to do another project.
3. The members of the team want to do another project.

Remember that software development is primarily a creative process. Persons who are having fun and believe in what they are doing will be more creative and productive. They will work smarter not harder.