The Case for Doing a Project by the Book

by Sheldon Linker

Linker Systems, Inc.

800-315-1174 — www.linkersystems.com — linker@linker.com

I make most of my living by fixing projects that have gone wrong, rather than controlling and performing on projects that are going well. I'd rather be doing the latter, because my customers are a lot calmer and happier when they're not in trouble. To that end, I present the case for doing a project by the book, rather than by cutting corners.

Why do we need it?

The first step in doing a project is not what you think. The first step is knowing why you need the new system, or modifications to the system in the first place. Some good examples of why you need a project done are:

- The current system will come to a screeching halt at the end of the year unless we do something about the problem
- Something happening manually in the business could be automated for better performance at the same or better cost, or could be automated for lower cost at the same or better performance
- There's an untapped profitable opportunity
- It will allow customers to…

There are also bad examples of why you need a project done:

- It would be really cool to…
- Everyone else is porting to… (or rewriting in…)
- Bob says we should.
- Let's put a GUI (or web UI) on it!

It will only take you a little while to figure out why you need something. But, you must also take the time to be Devil's Advocate, and try to show that you don't need it. If you can show that you don't need it, you've saved a pile of money and time. If you take the time to show that you do need it, you've protected your job. If you skip this step, you've only saved an hour or two.

What do we need?

The next step is determining what you need. Very often, in attempting to form a business solution, those deciding what to do either lose track of the objective, or start adding more features that are needed to the system. The latter is called "feature creep". Very often, you'll find that there are several types of features you'd like:

- a required set of features,
- a set of features which are not absolutely necessary, but would enhance the project's usefulness or speed, and
- a set of features that would be nice, but for which there is no business justification.

Obviously, you'll need to make the required set of features a part of your specification.

The set of features which would constitute valuable enhancements should generally be held off and made a second phase. By using a two-phase approach, you can have the product up and running and in the hands of the users, and getting value from it as soon as possible. By including in the specifications the future requirement for the enhancements you'd like, but annotated to show that they are not to be implemented now, you pay for a single unified design (and save the cost of redesigning to add the features later), but don't pay up front for every feature.

By leaving out the features you don't need, things may not be as "cool" as you would like, but they will be less expensive and on time.

Here's a case in point: An insurance company we helped had designed a computerized system to handle just about everything in it's business. They were replacing a system that did some of the work, but required that much of the work be done manually. The legacy system used character-mode terminals. The new system was to have a GUI (graphical user interface). Eventually, they wanted to have the customers be able to reach some of the information by phone or web. Here are the phases of the project as they planned it:

- Design the GUI front-end
- Design the back-end processing to match
- Implement the front end and the back end at the same time
- *Roll the product out*
- Design the customer web UI
- Figure out how security will work
- Build the web UI and at the same time, build a web back-end
- Design the interface between the web back end and the existing processing back end
- Implement the interface
- Add security to the system
- *Roll the web product out*
- Design the voice processing UI
- Build the voice UI with back end
- Design the interface between the voice front end and the existing processing
- Implement the interface
- *Roll the voice product out*

If we had been there from the inception of the process, or if they had done their work by the book, the phases would have been more integrated:

- Design a character-mode terminal interface
- Design the GUI front end
- Design the web UI
- Design the voice UI
- Design a common back end to interface to all front ends, with required security
- Implement the character-mode front end and the back end at the same time
- *Roll the product out (quickly, since no GUI was involved)*

- Add the GUI
- *Roll the product out*
- Add the web UI
- Add security
- *Roll the web product out*
- Add the voice UI
- *Roll the product out*

As you can see, there are more steps in the second approach, but the steps happen much more quickly and at a lower cost.

Preliminary Design

A preliminary design is a design which conveys all of the requirements and features, but does not get more specific than it needs to. An example of the difference between preliminary design and (complete) design is Customer address. In the preliminary design, it is sufficient to say that there is a customer address. We don't care if it's part of the customer record or a separate address record. We generally don't care what the distinct fields are, nor their sizes or even types are. We know that there will be a dialog (screen, page, conversation, or whatever) about it, but we don't need the specific form.

Send the preliminary documentation to the proposed users of the system. Ask what they think. Here's the hard part: Make changes as required and repeat the process.

At first glance, it might seem that stopping the design process here, talking with the users, and making changes (more than once) is an added time and expense. Actually, the opposite is true. The changes at the insistence of the users will occur (sometimes after years of griping and works-around). Might as well do them now, while they're still cheap (almost free, at this stage).

Final Design and Specification

We both know that no design is ever final, but it's still called "Final design".

At this stage, you should know all about the final data format and screen layout.or specifications, all items should be shown as a "meet or exceed" standard. For instance, If I decide that zip codes will be in the Zip+4 9-digit format, I should not say that the zip code must be stored as 9 digits. If I'm using the meet or exceed standard, the developers could instead store the 12-character bar-code data, allowing me to print Zip+4 and/or bar codes.

The specification is required whether the company is having the work done by an outside firm, or doing the work in-house. The specification consists of at least these three items:

- Documentation of the data the system must keep
- Documentation of the user-interface(s) and/or machine-interface(s) the system must support
- Documentation of any required methods of processing (such as formulae)
- Documentation of timing standards (such as respond in 2 seconds)

The data base documentation can be in the form of a list of data requirements, a final file design in SQL, Cobol, or other formal language, or anything in between,

The documentation of the user interface should be in the form of the final user manual. You're going to pay for a user manual anyway; might as well do it now. Manuals are far easier to change than code is, and both will change before we're done.

The documentation of the machine interface, if any, should be as detailed as possible.

Documentation of timing standards is especially important if the work is to be done by an outside company. For instance, you can's afford a quick inquiry that takes 10 minutes. You can't afford an overnight batch process that runs until noon, and you can't afford a monthly process that takes 32 days.

Again, send the design and specifications to the users and engineers involved before proceeding to the next step. Make your changes now, because you may not be able to make them effectively in future steps.

Get Bids and Evaluate Them

Always get bids and evaluate them, even if you're the only one at the company and you're going to do the job yourself. Here's an example. Let's say the company has a customer service and tracking system in mind. To get just what we want, it would cost us $10,000 worth of our time. But, we find that we can buy off-the-shelf software for $195 that does 95% of what we want. Is that last 5% worth $9,805?

If you're a larger company, and have your own in-house IT department, you still need large projects bid on internally and externally. However, in this circumstance, you need to figure incremental cost of the internal team versus cost of an external solution. The reason you figure incremental cost is that some of the costs of the internal team are fixed, and you are going to pay those costs whether the internal team does the project or not.

If you're at a very large company, the IT department is big enough that attrition counts. In this case, you're comparing real internal cost against real external cost.

When evaluating bids, don't just figure direct dollar costs. Other things to bring into the mix are the quality of the final product, delivery schedule versus need, and level of confidence in the bidder.

Enter into a Contract

If the internal IT department won the bid, then the contract is simple and one-sided. Basically, "You will produce This by Then." If you're contracting with an outside firm, there are several items to include in the contract. The following is a quick list of what our lawyer put into our contract:

- List of parties
- Date
- Recital (basically, an abstract of the intent of the contract)
- Paragraph covering sale of hardware
- Paragraph covering license of software delivered off-the-shelf
- Paragraph covering license of software to be produced
- Specifications (often a reference to an appendix) including:
    - Inputs
    - Algorithms
    - Outputs
    - Performance
    - Physical characteristics
    - Environment
- Delivery schedule
- Cost
- Payment schedule
- Customer obligations
- Stop-work clause

Acceptance criteria and schedule
Warrantee
Confidentiality/trade secrets terms
Maintenance and support terms
Limitations of liabilities (if any)
Security interest
Termination
Export assurances
Compliance with laws
Governing law & arbitration

Here's the short version: Make sure you don't have to pay for any item if they fail to deliver it, and make sure you don't have to pay for their legal mistakes. Your contract should make mistakes cost them, or they'll make mistakes cost you. Your contract will force them to manage their development well

Managing internal projects

There are several areas your costs will come from:

- Design
- Implementation
- QA (testing)
- Problems resulting from bugs
- Redesign
- Recoding
- Retesting
- Maintenance

Design definitely follows the old adage of "you can pay me now or pay me later". However, most companies use the philosophy of "everyone knows what they're doing" or "there's not time to do it right, but there will be time later to do it over." A sound design follows from the specifications. The specifications are actually the first round of design if complete. Before anything goes on, finish the user manual. You'll be making changes and adjustments later, but it's your best blueprint. The design should also include the complete data base layout (if any), list of modules, how they interconnect and react, the list of common data, including format, exactly how the modules or objects communicate with each other (down to the interface file, called the Macro Definition file, the H file, the Copy Book, and a host of other names by various languages), electrical connections, and anything else any two people might need to decide on later. Very often, each module or object will be left as a "black box". Very seldom does internal design of a small piece of the project matter to anyone but the one programmer who will produce that section.

Why spend the time, effort, and money to do all this design effort now? Because sooner or later the design has to be done. If you don't do it now, it will be done by the programmers. As the design evolves, the code will have to change with it. If the design is done first, you minimize the amount of recoding and redesign later.

No design effort is ever complete until you have documented it. Actually, you can skip the documentation portion if you can guarantee that nobody will leave the project, nobody will be added to it, and that absolutely everyone on the project will remember each and every design point and interface item brought up or agreed by everyone.

Conduct a design review, with a designated Devil's Advocate. The DA's job is to point out flaws in the design, failure modes, and the like. The DA also challenges research. Have a strong-willed individual hold this spot, since many people will not understand the value of this position, and will take challenges personally. Tell them to remember that each successful challenge improves the design, and each unsuccessful challenge makes the defender look good. It's a no-lose situation all the way around. In aerospace, and other situations where design flaws lead to disaster, each person presents their portion of the design, and everyone challenges it where possible.

Implementation follows from the design. One of the big tricks here is to make sure that everyone involved understands the portion of the design that applies to them, can communicate well with those holding adjoining spots on the interface and data flow charts, and understands how to do the job you're asking them to do. At the start of this phase, if two people want to trade jobs, let them. There's likely a good reason for it. As always, use good coding practice, but don't be manic about it. Anyone should be free to violate any coding rule with good reason. Here is where mentoring comes in. People will run into snags. Help them out technically. If you can't, find someone in the organization who can. Comment what's not obvious, and put page-long comments at the front to explain the processing, if you need to. Documentation within the code will never be lost.

Code reviews help here too. If you ask "why is that here", very often someone will say "oops". Better now than in debugging. Here is where you may see that someone is missing a trick. Explain it to them privately if just that person needs help, or publicly if you think everyone might be weak on the topic.

Testing occurs in several phases. Which phases you use depends on just how solid your code needs to be. For instance, a game might fit into the category "if it looks right, it's close enough", while flight control and medical software fit into the category "it's never good enough". Here are the typical phases:

- Programmer's unit test: Required. Here, the programmer checks whether the module or object works.
- QA's black box unit test: Optional. Here, QA checks the module or object against the specifications and manual (if appropriate) to see whether it meets the specifications. Also, invalid (but possible) inputs are fed in to see whether the right results occur in those circumstances.
- QA's white box unit test: Optional. Here, QA reads the code, then devises tests to try to break the code.
- Integration test: Required. The application is assembled into its deliverable configuration, and is tested as it is supposed to be used. The time-consuming but straightforward method we use here on new products is to go through the specification and the manual sentence by sentence, and check the sentences off as tested one at a time.
- Integration white-box test: Optional. Here, QA uses what they know of boundary conditions and the code to see if the code can be made to fail.
- Limit testing: Optional. Under what stress load or other conditions will the item fail to perform. Everything has such a limit. It's nice to know what it is before customers call tech support. Two examples from projects here: Animation Stand is a graphics product. What resolution can it handle? The 1.0 version was limited to images 32,710 pixels across. Physical systems have their limit too. How much

stress can a space shuttle wing take? You can always build an extra one and rip a wing off and find out.

- Fool's test: Optional. Well, sort of optional, because someone working at your customer's office will do this one for you if you don't. Have some people who have not read the specifications or the manual (preferably people who don't know how to read yet) play (not work) with the product for a while. Can they get it to do anything stupid or dangerous? This is why some products require two people to turn two keys at the same time.

Whenever possible, all tests should be repeatable and automatic. The reason for this is that you need something called "regression testing". Let's say that you plan to test functional groups A, B, and C in your product. You test A, and find that all is well. You test B and find that it needs repair. While you're testing C, B is fixed. Then you test B again. How do you know that A and C are still working? You can generally guess that they do, but you can only know by testing A and CC again. Thus, in this scenario, A, B, and C all got tested twice. If you don't have automated testing, you can't automatically retest everything that you guess is still working.

Testing will elicit changes. Make sure the changes are documented. Some of the changes will be changes to the way the application is built or coded. Others will be changes to the specifications or manuals to deal with improvements in the program.

Conclusion

Each of these items comes with a current cost, and a future cost savings. Your job as project manager or purchaser is to decide at each step whether the trade-off should be taken or not.