Managing the Application Development Process

Whil Hentzen

Hentzenwerke Corporation

980 East Circle Drive

Milwaukee, WI 53217-5361

USA

Voice: 414.332.9876

Fax: 414.332.9463

www.hentzenwerke.com

whil@hentzenwerke.com

Introduction

Let's meander from the nuts and bolts of writing code and designing objects for a few minutes, and discuss "the other 90%" of the application development process – the work involved in determining what it is that we're going to build, how to estimate what it's going to take to do so (in terms of time and money), and what we might do in order to make sure that "we dun whut we sed we wuz gonna do."

An Argument for Fixed Price Work

Highly skilled professionals in our line of work are typically underpaid when compared to professionals in other lines of business. It's likely that you could find a 25 year old accountant or lawyer working for a no-name firm above a shopping center nearby who's charging well more than most anyone here. And to add insult to injury, that 25 year old is charging those rates for the equivalent of adding up columns of numbers or standing in line to have documents filed.

Exaggerating? Perhaps a bit. But no one here will be argue with the experience of having a customer scream bloody murder at your hourly rate, yet not blink at paying significantly more for any number of mundane types of tasks by other professions.

And yet, in many cases, the work we are performing has a significant effect on the way the company runs their business. It is time that we get compensated for the value we are delivering.

Second, skill levels of developers run the gamut from highly skilled to completely incompetent. Customers have no way to determine the difference, and, as a result, highly skilled developers can't charge proportionately more. Suppose you're ten times more productive than the guy down the road - not an unreasonable assumption. Now suppose that developer charges, say, $40 an hour. Can you charge $400 an hour? Not likely. So, given your higher skill level, how do you make more money? It follows that if you posses a higher skill level, you can deliver the same product for less money, or deliver more product for the less money, or deliver more product for the same money as the other guy. As a result, if you quote applications with a fixed price, you can potentially make more money per unit of time than if you bill hourly.

Doing work on a fixed price basis requires four components. First, a detailed specification that defines exactly what will be delivered. Second, a mechanism for determining the cost for producing the application. Third, a set of tools for efficiently producing it. And fourth, a high level of skill with the tools to be used.

An Argument Against Fixed Price Work

Obviously, if you're missing any of the four components, a fixed price is not going to be to your advantage.

If you don't have a detailed specification, you'll fall prey to the scenario where the customer is expecting to receive one thing and you're expecting to deliver something else. "I'm sure we discussed the need for a variable, multi-dimensional amortization interpolation routine to be run every time the user blinks." "Yes, we discussed it, but we decided to postpone that until Version 2 – when the ReadYourMind ActiveX control has finished beta testing." Someone is going to end up unhappy.

Suppose you have a specification. If you're still estimating using one of the traditional techniques (e.g. guess at a number of hours, double it, and then double it again), then you may be able to deliver what the customer wants, but you may lose your shirt. And, contrary to what most programmers think, you can't make it up in volume.

Third, obviously, if you're still using outdated tools, you're not going to be as efficient. The guy next door with a snow blower may have had to spend some time learning how to use the machine, but while you're still digging out your car, he's back inside, watching All-Pro Wrassling.

And if you've just picked up a copy of "The Complete Idiot's Guide to Misusing WonderCalc" and are starting to sweat as you're paging through the installation instructions, then perhaps a fixed price isn't for you either.

Finally, consider the bigger picture. Frequently we're asked to use components or work on projects that are still under development or haven't been tested yet. Fixed Price work has no place when the project requires R&D or "proof of concept" work as part of the deal.

The Formal Specifications Process

This session covers two major topics – first, how to produce a detailed specification that will define what will be delivered, and how to work with the customer to create the spec, and second, a methodology to determine your cost for the application that you have designed.

These components are required for being able to deliver fixed price jobs profitably, but, as you'll see, they're very valuable even if you end up using time and materials billing.

Preparing the Customer for the Specification Process

Some customers are reluctant to pay for the design of an application, arguing that (1) the guy down the road will do it for free, or (2) that the design is part of the sales process and thus the cost should be born by the vendor. How do you counter these arguments?

The Phone Call

First, the relationship with a customer begins when they first ask for work to be performed. This often happens when the phone rings and the voice on the other end mumbles "Kin y'all write up this lil ol' computer system fer me?"

Your time is your only asset – and the voice on the other end is woefully unaware of this. They're all too willing to have you come out for a half-day visit in order to look at their system, without ever asking whether it's worth their time – or yours. Thus, it behooves you to manage the relationship so as to optimize the use of your time. An effective five or ten minute phone call can save you a great deal of time later, by allowing you to determine, roughly, if there's a match, and, if so, to continue.

At this point, it's time to do two things. First, determine what they want done, and second, to define the terms under which you will work. If you begin the entire relationship with an explicit understanding of how the relationship will work, it's less likely to become difficult later on. And one of the prime expectations to set is that you will be paid for all intellectual work performed.

Typical questions we ask include the size and scope of the application, the purpose and intended use, any environmental or language concerns, and, if the application is a

replacement or add-on, why the original developer isn't being called in, and who else is being considered.

Once the application expectations have been determined to some extent, I spell out, briefly, how the application development process works. I explain that a sales call for an hour or two is free, but any time thereafter is on the meter, and that we break systems into two pieces – the design and the production. There may be other factors or situations that come up later that will kill the deal, but you must do as much qualification as you can as soon as you can.

The Sales Call

The Sales Call is a physical visit where you get a chance to see what's really going on. I've found an on-site visit, even though we never do on-site work, to be extremely valuable. There are still a large number of folk who believe that since software development doesn't involve a product you can hold in your hand, the work involved in producing it isn't worth paying for. Thus, they're much more likely to engage someone for work either (1) without the intention of every paying for it, or (2) given the ethereal nature of the work, to misunderstand what is being done, and decide later to change their mind as to their needs and expectations.

An on-site visit can provide valuable information as to whether this is a firm you want to work with. Furthermore, a face to face meeting can help the customer become comfortable with you (or, unfortunately, convince them that they should have kept their fingers doing the walking <g>).

The Engagement Letter

Once the Sales Call is over, there will be one of four outcomes. The first is that the customer throws you out of the front window. This usually means you're not going to get the job. The second is that they make all sorts of promises, and that "they'll get back to you." I figure this is the coward's way of saying "No." If we were desperate for work, we'd probably chase these folk more, but I usually bail after a phone call or two results in additional "we'll get back to you" promises.

The third outcome is that you decide you don't want to do (or can't do) the work. In this case, the best thing to do is to explain that your schedule, skill set, or available personnel aren't a match for their expectations.

The fourth, and usually the desired outcome, is that the customer asks you to do the work. At this point, you need to put in writing what was discussed in the Sales Call. This typically can be done with an Engagement Letter that defines the process of developing a specification, the charges that the customer will incur, and describes what the specification will cover. At this point, the customer can decide whether they want to continue or not.

I've found that a written Engagement Letter also wards off those folk who would have you spend time doing design work that turns out to be more tentative than you had been led to believe. Without a written agreement, it's easy to end up in one of those "I thought this 40 hours of analysis was part of the sales process and that we weren't going to pay for it." This is all the easier to have happen if the customer doing some tire-kicking but isn't sure if they want a new app or not.

By describing the contents of a specification to the customer at this point, they will learn that what the guy down the road means by specification - three sheets of paper thrown together while watching Jay Leno the night before - and your specification - a complete design document that is sufficiently detailed to allow a programmer to produce and a tester to test with minimal follow-up questioning.

An analogy to blueprints for buildings or machinery is also useful in explaining why the specification carries a price and why it's not a trivial process.

In some cases, the customer will offer to provide their own specifications. We often let them do so. We've never run into a situation where their specs have been more than a bare starting point missing a great deal of information, but they're useful to point out where we can add value to the analysis process.

Components of a Specification

Cover Letter

- Description
- Price
- Delivery Timeframe
- Explanation of Fixed Price
- Terms and Conditions
- Acceptance

Executive Overview

- General Description
- Functionality
- Definitions and Processes

General Interface Notes

- Code Maintenance
- Maintenance Screens
- Buttons and Toolbars
- List boxes
- Pick lists
- Mover Boxes
- Notes button

Specific Interface Notes

- Specific Interface Notes

Functional Specifications

- Log On
- Application Launcher
- Main Menu

Menu Structure

- File
- Edit
- Operations
- Reports
- Tools
- Help

Description of a Typical Screen

- Purpose
- Access
- Usage
- Screen Objects
- Rules

Description of a Typical Process

- Purpose
- Access
- Usage
- File Formats
- Rules

Description of a Typical Report

- Purpose
- Detail Entity

- Filter
- Order/Group
- Fields/Objects
- Calculated Fields
- Additional Notes
- General Report Disclaimer

Typical Utilities

- User Preferences
- Data Sets
- Password Maintenance
- User Maintenance
- Data Maintenance
- System Maintenance

Technical Specifications

- Environment
- Operating System
- Hardware Requirements
- Third Party Software
- Interaction with Environment
- Directory Structure
- File Structures
- Table Summary
- Original Data
- Data Set Size and Throughput Analysis

Implementation

- Test Methodology
- Test Plan
- Test Data Set Requirements

- Deliverables
- Training
- Installation
- Milestones and Delivery Schedule
- Modifications
- Error Handling
- Application Feedback

Costing From a Specification - Overview

Accurately determining the cost of a piece of software has long been the bane of software developers everywhere. Accordingly, having a methodology to do so becomes a competitive advantage as well as a tool to help simply stay in business.

The Difference between Cost and Price

It's critical to realize that the cost for producing an application isn't necessarily the price. In fact, it probably shouldn't be, else, you won't make a profit, and most businesses are based on a profit motive.

The goal of determining your cost is two fold: first, you don't want to price your work below your cost, and, second, as mentioned in the first session, the way to maximize your revenue, and your profits, is to quote applications on a fixed price. The higher this price is, the higher your revenues and profits will be. The key, as discussed in the first session, is that you have to be explicit about what is being delivered for that price.

It is incumbent upon you to determine the value of the application to the customer, so that you can price it accordingly. The only thing that this costing methodology will do is make sure that you don't price below your cost.

The Costing Methodology

The basic premise behind this costing methodology is to determine "how many things" are in your application, and to determine your cost for producing a "thing", and then to multiply the two numbers together.

Counting "Things"

An application can be broken down into five categories: Forms, Processes, Reports, Foundation, and Other Stuff. Each of these can be broken down further to determine the number of "things" in an application.

Because the different components of an application look and feel different, and have different degrees of difficulty of production, it is nearly impossible to assign standard costs for each part. Instead, what we do is categorize each type of component as granularly as possible, and then assign weights to each piece. Multiplying the number of

components times the weight of a specific component results in a number that relates to the total size or scope of the application.

The components are based on a unit that we call an Action Point. For example, a label on a form would be worth one Action Point, while a validation for a check box might be worth three Action Points. If the form had five labels and two check boxes with validation, the total number of Action Points would be 1*5+2*3=11. There isn't an absolute value for an Action Point – it is merely a baseline for comparing the relative costs for various components. It would be just as easy to assign five Action Points to a label, but then, correspondingly, a check box would be worth fifteen.

In this way, two completely different applications can be compared in terms of size (and, thus, cost) by counting the Action Points.

Forms

The types of "things" that can be found on a form can be categorized into five areas. First, there are the "dumb" things, such as labels, images and other "view only" projects. The second group of things includes controls that map to a field in a table, and include test boxes, check boxes, option groups, and so on. The third group includes complex objects like combo boxes, list boxes, grids, etc.

The fourth group of "things" are non-visual - the underlying rules behind controls, such as validation, and behind the entire form, such as form level rules, triggers, and so on. The final group of "things" is a set of weights for the form itself - what type of form is it (a simple maintenance form receives a lower weight than a complex form set) - and other environmental considerations such as user security and operating system requirements.

Processes

A process is an operation that runs without user intervention, and thus does not require an interface. Some process may require a form in order for the user to provide parameters to control the process, but once initiated, the process generally needs no further interaction.

Processes are tricky - they may seem like one of those "none of the above" types of categories. However, we've found that we can generally break a process down into the following operations: (1) match two records in a table, (2) look up a value in another table, (3) assign a value, (4) insert a record, (5) create or delete a table, and (6) write an exception

Reports

A report is any type of output requested by the user - be it a printed list or output to be merged with a word processor.

The types of "things" found on a report map to those on a form. First are the dumb objects like labels and boxes. Next are straightforward output from a table - fields. We create a denormalized cursor that is sent to a report form and so the relationship of fields in the cursor to output objects on the report is generally one-to-one. The third type of thing are calculated fields and expressions - including subtotals, totals, variables, and so on. The fourth type of thing are orders and grouping levels.

Finally, since we invariably use Foxfire! for reporting, we also count how many elements are in each of the metadata tables - data items and joins - that we have set up. The more of this that we have to do, generally the less work is needed in the actual report set up, so it evens out.

Foundation

At times, you will be putting together pieces that are going to go into you foundation. This includes routines or functions that your foundation already contains, or that you are going to use to extend your foundation. How do you account for these? The answer is that you determine the number of Action Point just like any other Form, Process or Report, but then provide a weight or factor that may discount the tool so that you can spread the cost out among several applications.

Other Stuff

There will be those instances where a component simply doesn't map to one of these predefined categories. In this case, instead of just guessing randomly (remember, that's a bad thing), you can still break the component into smaller pieces, and then make some sort of guess at how many "things" are in each of these pieces.

An example would be an OLE Automation process. Instead of just guessing "Well, I think that will take about two days" you can break out the module into functionality and interfaces, and further identify pieces of the interface like done with the Processes earlier.

Determining Your Production Cost

Now that we've got a count of Action Points for an application, we simply need to multiply it by the cost per Action Point and we've got the cost of the application. So how do we determine the cost per Action Point?

If you don't like the answer to this one very much, you're not alone. Most people don't. The answer is that you use your history of what it has cost you in the past - and most people don't have those records in sufficient detail. What we've done is take our time records - details of how long we've spent on each component of an application - and then analyzed, in retrospect, how many Action Points were in each application.

From these numbers, we've been able to empirically determine our cost per Action Point.

What do you do if you don't have a history already? The best time to start tracking these costs is now. We track time down to a fairly granular level. We break the work we do into four levels: Customers, Projects, Modules and Tasks. A Project is a unit of work that requires a separate PO or, if the customer doesn't require PO numbers, is broken out for purposes of separate costing by the customer.

A Module is a component of a Project that is a distinct deliverable. For example, a Project may consist of two sets of screens and a reporting section. These may make up three separate Modules - one can be delivered and signed off before another is finished.

A Task is one of those things - Forms, Process, Reports - that can be costed out by itself. We track time against Tasks, and then iterate after completion to tweak the weights we use and make them more accurate.

(Regularly updated versions of many of these documents are available on our web site: www.hentzenwerke.com.)