Optional Scope Contracts

Kent Beck, First Class Software
Dave Cleal

> "Customers can now have what they want at the project end, after they've learned, instead of getting what they wanted at the project start."

Introduction

"Please propose a system to satisfy the attached requirements specification. Specify date of delivery and total cost." These words are enough to put lust and fear in the heart of programmers. Lust, because if you have a technical edge and experience in the area of the proposed system, you can make money far in excess of any conceivable daily billing rate. Fear, because when a fixed price contract goes bad, it goes really bad. Tense negotiations are the best you can hope for; unrecoverable costs, loss of reputation, or a lawsuit are the worst.

Kent digresses:

> In the heady early days of the commercialization of Smalltalk, I took a contract to deliver a spreadsheet engine to a Wall Street investment bank. I wasn't sure exactly what was going to be involved, but I was sure I could do it in six weeks. I hired my long-time collaborator Ward Cunningham to hole up with me for two days to get a good start. At the end of the two days, we were basically finished. I spent another two or three days polishing, loafed for four weeks, then shipped the software off a week early. The customer was happy enough to sign a contract for extensions.
>
> This time I had it wired. I would get another programmer to implement the changes, which weren't all that difficult. I stayed in touch with him as time went on. He seemed to be making good progress. However, a week before the deadline an ominous silence descended. Then, two days before the deadline, I got a strange call. "Ummm…. I'm in the hospital. My appendix burst. I'm so full of drugs I can't see straight. I hope it's okay."
>
> Aaaaarrrgggghhhh!!! Panic. I took the code I had received so far, stocked up on Jolt (I reserve Jolt for special panics), and in a marathon 48-hour programming session managed to make the deadline. Needless to say, the quality of the second delivery was not up to the quality of the first, nor was the customer happy.

Fixed Scope Contracts

The story above illustrates some of the advantages and disadvantages of fixed scope contracts. Overall, a fixed price/date/scope contract appears to shift the project risk from the customer to the supplier. The biggest risk is that the software will be harder to implement than expected. If this risk doesn't come to pass, the supplier's bet pays off, and such contracts can be very lucrative. If the worst happens, however, the supplier is in a position to lose a lot of money, either by canceling the contract or by finishing it at a loss.

Fixed price/date/scope contracts exist because they seem to serve the needs of both customers and suppliers. It looks as though customers get:

- Predictable cost
- Predictable deliverables
- Predictable schedule

And suppliers should get:

- Predictable revenue
- Predictable demands

What's wrong with this picture? For a start, the only predictable part of the customer's costs is the monies they pay the supplier. But what happens when a feature turns out to be harder to implement than expected?

The customer might not have to pay any more money to the supplier, but if the work is completed late or to a low quality standard, then the indirect costs to the customer can be huge. Even worse, what happens when someone realizes that a feature is less valuable than expected, or circumstances change the priorities that determined the original scope? It's too complicated to keep renegotiating the contract, so the issue typically gets shelved until version 2.0 of the system, while version 1.0 delivers software that everyone knows doesn't address the real requirements.

The supplier might have predictable demands, but if their understanding of the requirements turns out to be wrong, they face an unenviable choice. They either put in extra unpaid work or start an acrimonious argument about the detailed meaning of the contract. At best the argument leaves them with one less customer for the future. At worst it's time to crank up the lawyers.

The basic problem is that a fixed price/date/scope contract pits the interests of the customer and supplier directly against each other. Consider the selfish (read "under stress") positions of both parties:

Customer	Supplier
Interprets requirements as broadly as possible, to get as much for their money as possible.	Interprets requirements as narrowly as possible, to reduce resources.
Wants the work done as quickly as possible.	Wants to finish the work on the date, to have time to get the next contract lined up.
Wants superlative quality.	Wants to invest in just enough quality so that the customer will pay.
Burned out programmers are not the customer's problem, unless they threaten delivery[1].	Wants the team members to be successful on this project, and to stick around for the next one.

What we need is a different kind of contract that aligns the positions of the parties. The four variables of project management are:

- Time
- Cost
- Scope
- Quality.

The fixed scope contract specifies values for three of these variables:

- Time
- Cost
- Scope.

The other variable, quality, is in many real contracts left to flap freely in the breeze. But this doesn't make any sense for the customer, in either the short or long term. Just talk to the 'real' users at the sharp end of the business. In the short term, they want something that works, and they want it soon. They'll find a way to live without that complex report or the once-a-month function. It's not as if they have those today. In the longer term, though, they will need changes. The software had better be maintainable.

Fixed scope and quality

Why not specify all four variables? Actually, that's the way many projects start. Programmers actually prefer to do high quality work, so they tend to set some – usually pretty high – standards. The first thing that happens as the inevitable conflicts arise is that the programmers start to work harder. Poor managers applaud programmers that do this. Smart managers treat this behavior as an early warning of trouble ahead. Eventually the limits of human stamina come to bear, and something has to give. Does the individual

[1] Actually, of course, they are the customer's problem in the long-term, because their unmaintainable output is the customer's problem. It's just that when you're looking at a contract, it's hard to think beyond the contract's end.

programmer tell the manager that they're going to miss Friday's deadline? No, because there are lots of way to hit a deadline that's too tight:

- pronounce the software complete even when you know it doesn't handle every case
- stop wasting time (sic) testing
- ship the first version that works instead of refactoring to make it maintainable or even comprehensible
- stay up all night until you can't see straight, and then hope for the best.

When all four variables are specified, quality is the first one to go, because it's least visible – or, it's least visible before the software is delivered.

Optional Scope Contracts

If fixing time, scope, and cost doesn't work, and fixing all four variables doesn't work, how about another combination? What if we fixed:

- Time
- Cost
- Quality

and let scope absorb the uncertainty? And what if we kept the contract really short?

The contract would look like this:

> We will pay the team of six $75,000/month for the next 2 months. Whatever software they do deliver, will meet the quality standards in the small print below. There are some initial estimates in appendix A, but they are just for fun.

This can't possibly work. The customer doesn't know what they are going to get for their $150,000. The suppliers could goof off for 2 months and then turn in a very high quality login screen.

First, that isn't going to happen. The supplier wants repeat business. If the whole project is going to last a year, the supplier only has 1/6 of the money after the first contract. In any case, we've kept the contract short, so the customer is only risking 2 months' money before they find out whether progress seems reasonably quick.[2] The initial estimates on the features and a little math will give the customer a pretty good idea what they could get in the next few contracts.

Second, what happens when the current understanding of the requirements does turn out to be wrong? Both parties just change direction. The supplier has no motivation not to respond, because scrapping existing software doesn't affect their ability to meet the contract.

Let's revisit those misaligned interests.

Customer	Supplier
Interprets requirements as broadly as possible, to get as much for their money as possible.	Happy to accept customer's changing interpretation of the requirements
Wants the work done as quickly as possible.	Wants to deliver on time. Happy to get through as much functionality as possible, without risk to quality.
Wants superlative quality.	Wants quality before all else.
Customer want programmers to want to come back for the next 2 months.	Wants the team members to be successful on this project, and to stick around for the next one. If this

[2]You're actually risking less. The fixed scope contract could leave you with a huge but unusable (and therefore ultimately worthless) system. At least this login screen will have the quality attributes to be a potential basis for a valuable system

	is a year-long project, the supplier only has 1/6 of the money after the first contract.

We see more common purposes. The most likely conflict now is the one where the supplier insists on keeping the quality up when the eager customer is trying to cut corners to squeeze in one more feature.

The fixed scope contract had the following advantages:
Customers get:
- Predictable cost
- Predictable deliverables
- Predictable schedule

Suppliers get:
- Predictable revenue
- Predictable demands

The optional scope contract retains all of these advantages, except for "predictable deliverables". The customer has a reasonable idea of what they may get at the beginning, but reality rapidly intrudes. What they will get is inevitably different [from] what they imagined at the beginning. By giving up the illusion of control over scope at the beginning of the contract, they gain something much more valuable. Instead of getting what they wanted at the project start, they can now have what they want at the project end, after they've completed that learning process. What they learn to want is inevitably different from what they *could* have imagined at the beginning of the contract.

More importantly, however, the optional scope contract makes quality a constant. The team will work, for a fixed time and price, at the quality level demanded by the customer. Nothing will be "80% completed". 80% of the originally envisioned features might be completed, but each of them will be 100% done, as demonstrated by automated tests.

One important feature of fixed scope contracts is not duplicated by optional scope contracts. The supplier can't finish a job in a third the time and bill for the whole value (Kent's four weeks of loafing in the opening story). If the supplier finishes early, they ask for more work. However, optional scope contracts are more valuable for customers. When we've signed optional scope contracts, we have been able to charge enough of a premium because of the extra value that we were willing to forego the possibility of windfall profits.

Recording requirements

OK, so let's imagine that we've sold this approach to our customers and suppliers. They're now locked into a contract that excludes functional requirements. So what are the suppliers actually going to develop? We obviously still need some way to record requirements. What's the appropriate form?

One constraint unique to this contract is that we don't expect the customer to ever deliver "the requirements". We'll get enough requirements to start, and more later as we learn. So, we need our requirements to be recorded in bite-sized chunks and we need the customer to prioritize those chunks.

The prioritized list might be recorded in various forms, but each entry needs to describe something that the customer is willing to pay for. Some people call this a "use case"; some call it a "set of use cases"; other call it a "story". We'll use the word "story" here. What's important is that:

- The programmers must be able to imagine how they can verify that the system actually implements the story. Automated test cases are a simple and objective mechanism for this verification.
- The programmers must be able to estimate how much effort this story will require, relative to all the other stories. Without estimates, the customer cannot make informed priority decisions.

- The story must be comprehensible to the users of the system. If the customer is going to compare the priority of this story with others, they must have a feel for its value as well as its cost.
- We need to be able to code fewer or more stories to use up the time available. So smaller stories are better, because they let developers work in parallel, and we can use all of the time available. Our contract forces suppliers to deliver only software that is high quality. Partly completed stories won't be in the delivered software: so a very large story risks getting nothing for a large investment of effort.

There is a tendency in software engineering to try to fix as much information as possible before coding, to avoid large downstream costs. If we can somehow maintain the flexibility of software, however, we can afford a much quicker peek from 30,000 feet before parachuting into action.

In particular, the amount of information you need about our "chunks" of functionality before we can estimate and prioritize them is much less than the amount of information we need to actually implement them. This is because our contract doesn't encourage us to treat "wrong" estimates as a huge problem. We can expect the early part of development, before the customer has helped us narrow focus by setting priorities, to pass quickly.

Conclusion

Here are the characteristics that make an optional scope contract valuable:

- Customers can change their minds
- Suppliers aren't encouraged to sacrifice quality as soon as something goes wrong
- Customers' and suppliers' interests are contractually aligned
- The knowledge that both parties gain *during* the project can influence the finished product.

All of these revolve around the management of change. Imagine a project where the customer knows everything about the problem at the outset, the supplier has run a dozen similar projects before, and both sides are certain they will get no surprises during the project and are certain they won't learn anything during the project. Such a project won't gain anything from this form of organization, and the traditional models are fine.

But, if that isn't the case, then something unexpected is going to happen during the project. When that happens, you want the customer and supplier to work together to deal with the problem, and you want the ability to change the shape of the project to reflect the new reality. In that case, consider an optional scope contract.

And, even more radically, an optional scope contract means you don't even need to have an opinion about certain things. When will we be finished? After as many cycles as we choose to run. What do we ultimately want from the system? Not sure: we'll deal with that after the obvious things have been coded. The optional scope contract lets the team make useful progress whilst deferring decisions about things they don't yet understand very well.

The implications of developing an optional scope contract challenges conventional software development strategies.

- Evolutionary design- You don't want to invest in features you won't implement, so you can't analyze and design the system all at once. Instead, you have to be prepared to evolve the analysis and design throughout development.
- Evolutionary delivery- You want to make money with the features you've implemented while you are waiting for the "optional" features to clarify themselves. You must put a minimal system into production, and continue to implement features while you support the running system.
- Internal and external quality- you expect to change the system, so you have to invest in the quality that lets you radically rework the system throughout the project without prohibitive costs for reworking and retesting.

- Automated testing- When you are finished with a feature, you must demonstrate that it is finished, and you want to ensure it doesn't break in the future. Automated tests demonstrate features.

Perhaps the biggest barrier to the acceptance of optional scope contracts is that programmers can't believe that customers would accept such a contract. If the customers are happy with their relationship with their suppliers, this probably is, and certainly should be, true. However, if the customers are unhappy and the programmers are unhappy, something will have to change. Optional scope contracts gain strength precisely in those circumstances that are likeliest to give fixed scope contracts trouble: vague and changing requirements.

Contractual definitions of quality

Clearly this style of contract is only as effective as its definition of quality. A full review of how to define good measures of quality would be the subject of another, much larger, article. However, here are some pointers to ways that we believe quality can be controlled.

Most of the traditional measures of quality are based on external observation of the software. Is it sufficiently bug-free? Just count the bugs reported over three months and compare it to some target maximum bug count. The trouble with this kind of approach is that by the time you know the software is poor quality, the supplier has been paid, the individual developers have scattered to the four winds, and it's just too late to anything about it.

So, instead, we look for internal quality measures. As we've implied in this article, we believe that an essential element of quality software is automated test suites that test 100% of the system's functionality. (How do you know they cover 100%? Read the next article).

Another quality measure is how well factored the software is. The whole issue of *refactoring* is currently a hot topic at software development conferences. The idea is to change software purely to make it simpler, more readable and more maintainable, without changing what it actually does. It's perfectly possible to employ a third party to review representative parcels of code to determine whether every possible refactoring has been applied. The benefits of this approach are software that is orders of magnitude easier to understand and cheaper to change: and that's critical for the evolutionary development process we're proposing here. See [Fowler99] for much more information about the science of refactoring.

Both the above are quality measures that we'd apply to every project under the sun. Other traditional quality issues may be better handled within the stories. Performance is one: any hard performance constraints should be elements of the stories. An example might be "All end of day processing must be completed before the bank reopens for business the next morning". An appropriate test would be added to the automated test harness for the "end of day process" scenario: and the story would only be considered deliverable if the test succeeded.

In other environments, other softer, more system-wide qualities may be important, such as user-friendliness. In this case build some quality standard into the contract. The most important thing is to have an unarguable measure. Remember the measure doesn't have to be objective. For example, if you are concerned about user-friendliness, then it may be best to jointly agree to abide by the decision of a trusted third party ergonomist.