UDC 004.4

No Cure No Pay: How to Contract for Software Services

Tom Gilb

Independent Consultant
Tom@Gilb.com

Abstract. Contractual motivation is needed to avoid costly project failures and improve the delivery of stakeholder value. Only if the supplier management is made to feel the pain of project failure will it strive to avoid it. The current culture of rewarding failure, by paying for systems development work regardless of the product delivered, must be altered. Such contractual motivation must be supported by quantitative requirements and evolutionary delivery. Quantitative requirements allow project progress and success to be measured enabling monitoring and testing for contractual compliance. Evolutionary delivery (that is, delivering early high value in small increments and using feedback from deliverables to determine future increments) allows early reporting of the ability of systems development to deliver and so enables any required corrective actions. *Note: This paper specifically addresses the software problem, but the ideas most likely apply to the wider systems engineering problem to some interesting degree as well.*

1. Introduction

At least 10% of all software projects are 'failures' (cancelled before completion) and another 50% are 'challenged' (completed and operational, but over-budget, over the time estimate and with fewer features and functions than initially specified) according to widely quoted surveys in the UK and USA (See [1] quoting a March 2003 press release on the Chaos Report from the Standish Group, and an Oxford University – Computer Weekly study of project management by Chris Sauer and Christine Cuthbertson). Several large UK government projects have reported spectacular failure at great expense to taxpayers [1,2]. What is the problem? Most discussions have centered on improving the software engineering process itself: better estimation, better requirements, better reuse and better testing. No doubt all these can be improved. However, I suggest the motivation to improve them needs to be put in place first. Think about it. Most project failures have been fully paid for! We not only pay well for failure, but the bigger the failure (in terms of project size and project duration), the more people get paid!

My suggestion is simple. Pay only when defined results are provably delivered. This requires several things:

Contracts that release payment only for meaningful results

The ability to define those results quantitatively, particularly the requirements currently defined as qualitative, and particularly the organizational benefits

The ability to deliver the results incrementally, thus proving capability at early and then on-going stages.

2. Defining the 'cure'

We write contracts, and we write requirements for projects, but these are often useless because we define the wrong things: that is 'valueless' things! Such as the following:

- Qualitative critical benefits: that is, words such as 'higher productivity' (we do not define quality measurably)
- Functions and use cases (not the improvements needed to them and the resulting benefits)
- Designs, architectures, technologies (not the results required of them)
- Hours of effort required (not value delivered).

This means we fail to understand the 'value' a project is to deliver. We need to define:

- The different stakeholder groups ('stakeholders') involved
- The *results* that the different stakeholders value
- The *quality* levels, numerically and measurably
- The stakeholder value that the new or improved system will enable to be achieved
- An evolutionary series of early, short and frequent delivery stages with high-value delivered at earliest opportunities.

Of course much of this is *management* work (not software engineering), and management consistently fails to manage properly. They make consistently bad assumptions that the software or IT system will deliver benefits. But they do not control the delivery of those benefits. Why is management so bad at this? It must be because they have not received training (at any level of management training) on how to control IT project delivery. This is the fault of senior management. It is senior management who should take action to ensure that money is not wasted on failed IT projects by ensuring appropriate management training and reinforcing this training with a 'Pay for Results' policy for IT projects.

Senior management is also responsible for forcing decisions to be made on too large a scale and too early. We define too-large pieces of work: we define work delivery packages in terms of years. We should define them instead in terms of quarters, months and weeks. We would then see early if any planned value can be delivered to stakeholders, and if not, we would have chance to correct the situation or remove the incompetent supplier.

We also choose contractors too early, based on the wrong criteria. We choose based on size and 'reputation' (note, not the reputation for delivering successfully!). We should instead choose suppliers based on proven ability in the past, and on our project to deliver. One way of improving contractor selection would be to allow three competing contractors to start work in parallel on complimentary aspects of a system, and move work towards the contractors that prove their ability to deliver value. Move work away from those that do not deliver value as promised. If all three perform well, that is fine, keep them going!

So what other actions should senior management take to correct all these problems (not defining results, too-large increments and selecting contractors based on the wrong criteria)? We shall outline some suggestions in the following sections.

3. Motivating to contract for results

Senior management must first motivate the cultural change to contracting for results. Here are some ideas outlining the required high-level policy:

- It is our professional/ethical/legal duty towards shareholders/ customers/taxpayers to make absolutely sure we do not waste money on projects that do not deliver expected results
- The supplier/contractor has the technical/managerial/economic/ experiential competence to control the results/cost ratio for us. That is why we are contracting with them. If they cannot contract for results, we should not do business with them
- It is our professional obligation to always clearly, measurably and testably define what we expect to get for our money, in advance. Anything less is incompetence and unprofessional. If we cannot do that, we should not have the job of commissioning any use of organizational resources
- We do not intend to *ever* be a party to failure. If our contracting process fails to deliver, it is *our* fault, *not* the supplier's fault. Any failure will be *structurally* (step size one week, for example) kept to a minimum, and failure will result in reconsidering the supplier's continued participation
- We need to prioritize high value results early one, and lock them in. We cannot get involved in primarily, and initially, building superstructures that fail
- Maybe we, the customer, should make this even clearer by bonus or rewards for employees and teams that manage value deliveries successfully.

Senior management is going to have to ensure that staff have adequate guidance and training by supplying the following:

- A clear policy (see above)
- Training in quantification of values delivered
- Training in measurement and testing of value delivered
- Training in requirements specification
- Useful templates for specifying requirements and designs [3]
- Access to experts in-house or externally.

- **The request for proposal (RFP)**

The request for proposal must be along the following lines:

"We invite you to tender for a contract to <build software/deliver an IT system>. The contract will be based on a 'value payment' system. This means that we will define what *we* expect in terms of measurable and testable values from the system. We will pay only when that value is satisfactorily and provably delivered. We will not pay for effort put in, and we will not pay for sub-specification results. If you are focused on delivering us the results we agree on, then you can earn money independently of the costs to you. Efficient suppliers can earn more than usual. Inefficient suppliers would not. We hope you will get rich by helping us to get what we expect for our money."

4. Specifying the contract

A No Cure No Pay contract must be put in place. Such a contract must suggest the use of evolutionary project management (which we shall discuss later in this paper). The essential ideas in such a 'No Cure, No Pay' Contract are as follows:

- Payment is totally dependent on proven delivery of the defined value
- Estimates for delivering the value will be made by the supplier in advance
- We will accept some level of cost overrun compared to the estimates, when actual costs exceed the estimate (for example, 100%). Above that, the supplier pays any excess costs
- We will allow invoicing to be triggered based on a simple test of delivery
- Actual payment of the invoice is dependent on a trial period with continued success (for example, 30 days).

A sample contract is shown in Figure 1.

A Sample 'No Cure No Pay' Contract for Evolutionary Project Management

Intent. The intent of the evolutionary project management method is that the customer shall gain several benefits: earlier delivery of prioritized system components, limited risk, ability to improve specification after gaining experience, incremental learning of use of the new system, better visibility of project progress, and many other benefits. This method is the best-known way to control software projects [4].

Precedence. This contract has precedence over any conflicting paragraphs in previous contracts. (This sample contract is designed to work within the scope of a present contract with minimum modification. You can choose to declare this contract has priority over conflicting statements in the initial contract – the alternative is to 'clean up' any conflicting statements.)

Steps of a phase. For the defined phase, the customer may optionally undertake to specify, accept and pay for evolutionary usable increments of delivery of any size. These are hereafter called 'steps'.

Step size. Step size can vary as needed and desired by the customer, but is assumed to usually be of one-week duration.

Specification improvement. All specification of requirements and design for a phase will be considered a framework for planning, not a frozen definition. The customer shall be free to improve upon such specification in any way that suits their interests, at any time. This includes any extension, change or retraction of framework specification, which the customer needs.

Payment for acceptable results. Estimates given in proposals are based on initial requirements, and are for budgeting and planning purposes. Actual payment will be based on successful acceptable delivery to the customer in evolutionary step deliveries, fully under customer control. The customer is not obliged to pay for results, which do not conform to the customer-agreed step requirements specification.

Payment mechanism. Invoicing will be on the basis of evolutionary steps triggered by end of step (same day) signed preliminary acceptance that the step is apparently as defined in step requirements specification. If customer experience during the 30-day payment due period demonstrates that there is a breach of the specified step requirements, and this is not satisfactorily resolved by the supplier, then a 'stop payment' signal for that step can be sent and will be respected until the problem is resolved to meet specified step requirements specification.

Invoicing basis. The documented time and materials will be the basis for invoicing a step. An estimate of the step costs will be made by the supplier in advance and form a part of the step plan, approved by the customer.

Deviation. Deviation plus or minus of up to 100% from step cost and time estimates will normally be acceptable (because they are small in absolute terms), as long as the step requirements are met. (The assumption is that the customer prioritizes quality above cost). Larger deviations must be approved by the customer in writing before proceeding with the step or its invoicing.

Scope. This project management and payment method can include any aspect of work, which the supplier delivers including software, documentation and training, maintenance, testing and any requested form of assistance.

Fig 1. A sample No Cure No Pay contract

5. Motivating suppliers to contract for results

Suppliers can be expected to initially resist such contracts, for all the obvious reasons:

- They are unaccustomed to this way of contracting
- They will probably require permission to do so from their own senior management
- They are not sure what they will be getting into. They have no such experience
- They like the old idea of getting paid millions, even if the result is useless for the customer.

So, we are going to have to motivate them!

- Refuse to do business on any other terms
- Make sure they know you are offering this business to their competitors, and that their competitors have indicated willingness to do it
- Employ multiple competing suppliers at the same time. Let them know that the biggest share of the business will go to the best value provider
- Get your top management to make it clear to their top management, that this is the new way of doing business
- Get some early examples of supplier success using this method to tell hesitant suppliers about
- Reduce and eliminate business flow from the suppliers who do not actually deliver results, and redirect the flow to those who do.

Having looked at the policy and contractual aspects, let's now discuss some more technical specifics: how to quantify requirements and a brief description of evolutionary steps (steps).

6. Quantifying qualitative results

We all know how to specify results about storage capacity, transaction throughput, and response time. The difficulty for most customers is

specification of *quality* ideas – popularly called the 'ilities' or non-functional requirements (NFR).

This is not difficult, *if* you have some training, and some quantification templates. We have found that absolutely all qualitative aspects of software can be expressed quantitatively in a practical way. We have also seen that once you find a quantification, there is always a practical way to test the level of that quality in practice. Such testing need not take vast amounts of time and effort: for example, one customer of mine reported for a website application that they needed between 30 minutes and 2 hours to carry out all their testing of the quantitative requirements [5].

Few professionals, managers or engineers, have any training in quantification of qualities. So, this is one of the real barriers to paying only for results: the quantification of the *qualitative* values. In fact, the process of quantification is simple in principle, and is mostly willpower and common sense. If it varies, it can be quantified, by definition.

Many qualities are 'complex'; meaning they actually involve a set of sub-quantities. It is most likely you will have to list the sub-qualities, and quantify them individually. Failing to create a hierarchy of sub-qualities is one of the most common reasons for failing to quantify a requirement (instead people are left struggling with a concept that is too high-level). If you fear too many sub-qualities, then you can always simply select the most important ones. Selecting the ones with the best means of measurement can be one way of determining the key sub-qualities (this is because ease of measurement often goes hand-in-hand with something the organization already deems as of value). Here's an example of a hierarchy for 'Maintainability', which tracks the process involved:

Maintainability

- Problem Recognition Time
- Administrative Delay Time
- Tool Collection Time
- Problem Analysis Time
- Correction Hypothesis Time
- And many more! (Fix, Check, Tests).

Having identified your key sub-qualities, you then have to specify the quality levels. To do this you need to determine a useful Scale and some levels on it. Note if a Scale is not immediately apparent then it could be a case of needing further hierarchical decomposition into even lower-level sub-qualities.

Write the word 'Scale:' and define the quality variation in such as way that you can put a number on it. Think of how you would measure the sub-quality. For example, for a sub-quality 'Intuitiveness':

Intuitiveness: Type: Quality Requirement.

Scale: Average percentage (%) of tasks where defined [Users] need no help from people,

manuals, help lines etc. to correctly and immediately carry out defined [Tasks].

Then decide two levels using the Scale, the current level (a Past level) and your target level (a Goal).

Past [At date measured]: 30%.

Goal [Next Version and/or date for achieving]: 80%.

(This is from a real example by one of our customers [5] – they reached their goal in 3 months.)

The value of reaching such a technical goal, as the 80% for Intuitiveness, can be understood, perhaps even roughly estimated, as follows:

- Estimating the time saving for an average task (say 3 minutes)
- Estimating the amount of such savings for a year: say 3 x 100 users X 10 times/day X 200 days = 600,000 minutes or 10,000 hours/year.

Table 1. A real example of weekly control of quantified quality results from Future Information Management Research (FIRM) [5]. In the 9th week, 95% of the planned improvement is delivered (2x planned result) using a design idea called 'Recoding'. Similar results were achieved every week in the first 24 weeks of using this method of project control

Curent Status	Improvements		Goals			Step 9 Design = Recording			
						Estimated impact		Actual impact	
Units	Units	%	Past	Tolerable	Goal	Units	%	Units	%
			Usability.Replaceability (feature count)						
1.00	1.00	50.0	2	1	0				
			Usability. Speed. New. Features Impact (%)						
5.00	5.0	100.0	0	15	5				
10.00	10.0	200.0	0	15	5				
0.00	0.0	0.0	0	30	10				
			Usability. Intuitiveness (%)						
0.00	0.0	0.0	0	60	80				
			Usability. Productivity (minutes)						
20.00	45.00	112.5	65	35	25	20.00	50.00	38.00	95.00
			Development resources						
	101.0	91.8	0		110	4.00	3.64	4.00	3.64

See Table 1, which shows metrics for some other sub-qualities in an impact estimation (IE) table produced by this customer. Notice under the heading 'Goals' that the Past benchmark levels and the future target Goal levels are given. There is also an indication of the levels that would be just 'Tolerable'. The right-hand side of the table shows the impact on the Productivity sub-quality level of implementing a solution (design idea) of

'Recoding'. The bottom two rows capture the development costs permitting the cost benefit ratios to be calculated.

If you are stuck for ideas for Scales, a search using an internet browser might help you (you can obtain maybe 20,000 to 60,000 hits for each of the more common -ilities!). If you would like more detail on the basics of finding Scales, see Chapter 5 in [3].

7. Direct and indirect results

There is a critical distinction between the performance characteristics of software or an IT system itself (system performance) and the organizational impacts that achieving the system performance characteristics are expected to deliver. For example, a system might be designed to a usability level so good that it takes only 10% of the effort to learn and to use that was needed for the older systems. This system performance characteristic has an *organizational* value in relation to:

- How many people it affects (savings population)
- How often they use it (savings frequency)
- The time period over which the value is derived (lifetime value).

As a practical matter we will probably pay the supplier for the system performance. It is then our problem to exploit that performance, and achieve the tangible savings. Such exploitation is not under the control of the supplier. However, maybe in some cases there is an option to contract for the *actual savings through time* as the product is used. This could hold for a limited time period (say for 60 days or for a year) as a kind of field trail acceptance test. It could conceivably be a lifetime of system royalty to be paid to the supplier. This is not unlike the periodic software-use fees, or licenses, in widespread practice, except they would instead be based on *value delivered*, rather than *product used*.

8. Decomposing projects into small deliverables

One useful way to make sure that value is *really* delivered is to make it happen early and often. In my opinion, that translates to deliveries occurring next week and every week. Many organizations from Hewlett Packard to the 70-person software product developer, Future Information Management Research (FIRM) do exactly that.

Nobody is actually against early and frequent delivery of stakeholder value. The problem is that people have not been trained, or led, to decompose their long-term visions of improvement, into a series of smaller incremental tasks.

Table 2. This is a conceptual example [3]. Three goals (performance targets) and two resource targets are having the real impacts on them tracked, as steps are delivered. The same IE table is also being used to specify the impact estimates for the future planned steps. So at each step, the project can learn from the reality of the step's deviation from its estimates. Plans and estimates can then be adjusted and improved from an early stage of the project.

Step / Target Requirement	Step 1			Step 2 to Step 20		Step 21 [CA, NV, WA]		Step 22 [and all others]	
	Plan % (of Target)	Actual %	Deviation %	Plan %	Plan % cumul-ated to here	Plan %	Plan % cumul-ated to here	Plan %	Plan % cumul-ated to here
Performance 1	5	3	-2	40	43	40	83	-20	63
Performance 2	10	12	+2	50	62	30	92	60	152
Performance 3	20	13	-7	20	33	20	53	30	83
Cost A	1	3	+2	25	28	10	38	20	58
Cost B	4	6	+2	38	44	0	44	5	49

We are not simply talking about building systems incrementally. That is unavoidable. We are talking about actually delivering value (for example, time savings, money savings and/or making life better for people). We are talking about the very thing that the failed IT projects forgot to do. Give results! See Table 2, which shows how sequential steps can deliver a series of system improvements (incrementally improving the levels for various sub-qualities (shown here as performance target requirements).

Technical 'experts' will come up with 100 excuses as to why things need to take a whole year (actually often then about 3 years' project duration elapses before they admit failure). They cannot conceive of next week, and every week. It is outside their culture and their training, and also outside that of their management. So, they need encouragement, and training!

Management has to declare that hereafter things will be done in early, short, frequent increments. Those who can deliver this – will get work. Those who cannot – are not useful – and not valuable.

Then the technological planners need to be taught the art of decomposing their visions into incremental value deliveries. There are 20 key principles for such decomposition (see Figure 2). The main 'trick' is to focus on one value aspect at a time, and ask the simple question, "What can we do, in a week or so, to actually move measurably forward towards our Goal?"

How to decompose systems into small evolutionary steps: (a list of practical tips)

1. Believe there is a way to do it, you just have not found it yet![1]
2. Identify obstacles, but don't use them as excuses: use your imagination to get rid of them!
3. Focus on some usefulness for the stakeholders: users, salesperson, installer, testers or customer. However small the positive contribution, something is better than nothing.
4. Do not focus on the design ideas themselves, they are distracting, especially for small initial cycles. Sometimes you have to ignore them entirely in the short term!
5. Think one stakeholder. Think 'tomorrow' or 'next week.' Think of one interesting improvement.
6. Focus on the results (You should have them defined in your targets. Focus on moving *towards* the goal and budget levels).
7. Don't be afraid to use temporary-scaffolding designs. Their cost must be seen in the light of the value of making some progress, and getting practical experience.
8. Don't be worried that your design is inelegant; it is results that count, not style.
9. Don't be afraid that the stakeholders won't like it. If you are focusing on the results[2] they want, then by definition, they should like it. If you are not, then do!
10. Don't get so worried about "what might happen afterwards" that you can make no practical progress.
11. You cannot foresee everything. Don't even think about it!
12. If you focus on helping your stakeholder in practice, now, where they really need it, you will be forgiven a lot of 'sins'!
13. You can understand things much better, by getting some practical experience (and removing some of your fears).
14. Do early cycles, on *willing local mature* parts of your user/stakeholder community.
15. When some cycles, like a purchase-order cycle, take a long time, initiate them early (in the 'Backroom'), and do other useful cycles while you wait.
16. If something seems to need to wait for 'the big new system', ask if you cannot usefully do it with the 'awful old system', so as to pilot it realistically, and perhaps alleviate some 'pain' in the old system.
17. If something seems too costly to buy, for limited initial use, see if you can negotiate some kind of 'pay as you really use' contract. Most suppliers would like to do this to get your patronage, and to avoid competitors making the same deal.
18. If you can't think of some useful small cycles, then talk directly with the real 'customer', stakeholders, or end user. They probably have dozens of suggestions.
19. Talk with end users and other stakeholders in any case, they have insights you need.
20. Don't be afraid to use the old system and the old 'culture' as a launching platform for the radical new system. There is a lot of merit in this, and many people overlook it.

Fig 2. Twenty principles for decomposing a system into smaller evolutionary steps

[1] Working within many varied technical cultures, I have never found an exception to this since 1960 – there is always a way!

Strangely there always seems to be an answer in practice! If you don't get one, you either need to study decomposition methods more deeply, or listen to those who can find solutions better.

Basic Evolutionary Planning Policy

1. Financial Control: No project cycle can exceed 2% of total initial financial budget before delivering some measurable, required results to stakeholders.

2. Deadline Control: No project cycle can exceed 2% of total project time (For example, one week for a one year project) before delivering some measurable, required results to stakeholders.

3. Value Control: The next step should always be the one that delivers best stakeholder value for its costs.

Fig 3. A basic evolutionary planning policy statement for 'No Cure No Pay' project management

Figure 3 outlines the basic policy for evolutionary planning. However, don't panic if a certain step takes longer. Maybe do something useful to deliver other results in parallel, or just be patient for a week or two! Weekly or short increments are merely a useful discipline that makes us really focus on delivering value. The main point is that we focus on really satisfying stakeholders, and on keeping our promises to them. What is totally unacceptable is to plan a project of several year's duration that doesn't intend to deliver anything until at least a year in, then start the project and delay delivering anything for a few years until finally admitting failure. Anything is better than that old habit of IT people.

9. Summary

One way to avoid software project failure is to refuse to pay for failure. This will motivate software suppliers to make use of already well-known and well-practiced methods for successful IT and software projects [3, 4].

There are two key ideas that too many people do not practice (due to lack of training and/or poor management). The first is the quantification of the value expected by stakeholders of the system, especially the 'qualitative' aspects. This gives the basis for payment. The second idea is to divide all large projects into an incremental series of smaller projects. This means weekly increments (that is roughly 2% of duration time for a large project planned to be completed in about one year) of value delivery. Each increment must attempt to increase some aspect of stakeholder value, in the direction of the longer-term requirements. This 'small step' discipline makes sure that suppliers really know what they are doing, and are really focused on value delivery, rather than their traditional concern for technical construction.

Top management must lead this culture change: the software technologists have consistently failed for decades, and the problem has in the vast majority of cases never been the technology.

References

1. Parliamentary Office of Science and Technology (POST): Government IT Projects. Report 200. Available online at http://www.parliament.uk/post/pr200.pdf/ (July 2003)
2. Royal Academy of Engineering and British Computer Society: The Challenges of Complex IT Projects, ISBN 1-903496-15-2. Available online at http://www.raeng.org.uk/ (April 2004)
3. Gilb, T.: Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage. Elsevier Butterworth-Heinemann, ISBN 0750665076. (July 2005)
4. Larman, C., Basili, V. R.: Iterative and Incremental Development: A Brief History. IEEE Computer, 2-11. (June 2003)
5. Gilb, T., and Johansen, T.: From Waterfall to Evolutionary Development (Evo): How we rapidly created faster, more user-friendly, and more productive software products for a competitive multi-national market. INCOSE Proceedings. (July 2005) Also published in EuroSPI Proceedings. (November 2004)

Tom Gilb has been an independent consultant, teacher and author since 1960. He works mainly with multinational clients helping improve their organizations and their systems engineering methods. Tom's latest book is 'Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage' (2005). Other books include 'Software Inspection' (1993, co-authored with Dorothy Graham) and 'Principles of Software Engineering Management' (1988). His 'Software Metrics' book (1976, Out of Print) has been cited as the initial foundation for CMMI Level 4. Tom's key interests include business metrics, evolutionary delivery, and further development of his planning language, 'Planguage'.