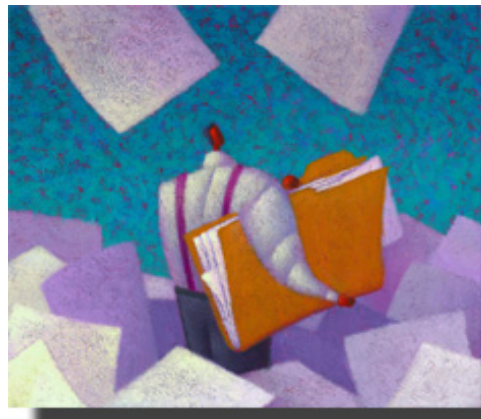


▶ **Project Scope Management: Effectively Negotiating Change**

by [Ben Lieberman](#)

Software Architect

All projects begin with the definition of needs. Needs of the client, needs of the development organization, needs of project managers, programmers, architects, and more. These needs cannot all be met in a reasonable time frame. In order to make progress, these needs must be prioritized in some way. Often, the project manager is responsible for establishing the base project priorities.



A good first step is for the project manager to prioritize all the identified needs, which we refer to as establishing the "project scope." If the project scope is too large, then the development organization will be overwhelmed, leading to chaos and project failure. If the scope is too narrow, then the resulting system will be unfit for the desired purpose, and the customer will likely refuse payment. Effectively balancing these varied needs requires all parties to negotiate in good faith and adopt a cooperative model between the client and the software developer. This article presents techniques for effectively categorizing, organizing, and prioritizing requirement changes over the course of a project; it also suggests ways to use this information to conduct effective negotiation between the vendor and the client.

Software programs are conceived to solve specific problems. An accounting package may provide the ability for businesses to perform complex accounting practices. A scientific software package may allow a researcher to conduct a sophisticated analysis of experimental data. A sufficiently full-featured software system can fly a spacecraft. In order to create a software application, a development organization needs to define the problem, interview users to discover their specific needs, and establish a project scope for development. While determining the system scope, the project manager must walk a fine line between the client, who desires the system to be as useful as possible, and the developer, who must create

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

the system within a fixed timeframe and budget.

All truly successful projects are collaborative efforts between the customer and the development organization. Collaboration allows for increased efficiency, accuracy, and comfort for all participants. Unfortunately, most projects operate using a competitive model that creates an unhealthy tension between the client and the development organization; this leads to disagreement, argument, even lawsuits. Along the way, the original goal of creating a software system that meets the true needs of the user is lost.

Below, I will first examine a couple of key factors in the client/vendor relationship, taking into account human nature and the effect the desire to "have one's way" can have on the relationship. Then I will present some techniques for analyzing the impact of scope changes on the project schedule and prioritizing requirements based on cost, risk, benefit, and effort.

Cost of Features

All software features cost a significant amount of money to develop, and controlling cost is one of the jobs of project management. Often, development teams respond to pressure to please a critical client (who is, after all, the source of the money) by allowing the introduction of new requirements into the project. Unless project scope can be effectively managed, such additional requirements may flow into the project in a haphazard manner, leading to the inclusion of features that were neither discussed nor planned. It is likely that these features will not be properly tested (if they even get mentioned to the test team), and the addition of this functionality will likely come at the expense of the overall system architecture. Of course, some changes will be necessary and beneficial, but many will be detrimental.¹ In addition, the haphazard introduction of new features will destroy a project schedule more rapidly than any other single cause.

As illustrated in Figure 1, satisfying 1% additional needs can easily translate into 10-50% additional development work. This dramatic increase is due to the work needed to correctly capture and communicate the desired features, translate these features into a workable computer architecture, and finally create complex programming code.

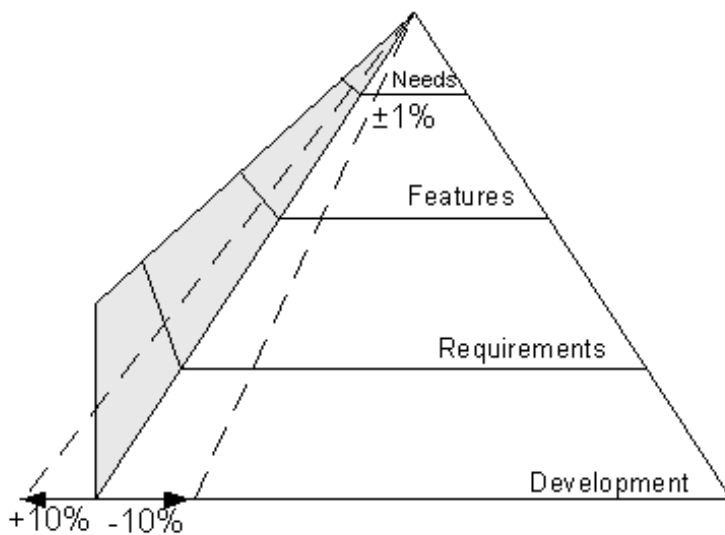


Figure 1: Requirements Cost Pyramid

On the other hand, good scope management can significantly reduce the workload by eliminating or postponing a very small percentage of "needs." A project manager can avoid success-threatening work by reaching an agreement with the customer on the true importance of a requested change. Naturally, clients seek to maximize delivered functionality for minimum cost, just as development organizations seek to minimize risk and maximize return on investment. What's needed to resolve this dilemma is the ability to *negotiate*. To negotiate effectively, we need to understand the likely impact of each feature in terms of risk, effort, and benefit.

Effective Negotiation

The goal of negotiation is to resolve competing forces, bringing parties into agreement on a reasonable solution to the conflict. Most often a conflict arises because of the need for each party to look for the most advantageous personal position, the standard driving force of competition. Although competition between groups can lead to positive gains between two organizations, competition within a group is almost always detrimental. A software project requires a close, cooperative alignment between the client and the vendor, so an understanding of the forces that will affect this alignment is beneficial.

Understanding the Game

Many of the forces driving individual or group actions during conflict can be modeled through "Game Theory," which uses mathematics to help competing parties choose an optimum strategy.² A classic example of the conflict inherent between cooperation and competition is evidenced in the simple but profound game called "The Prisoner's Dilemma," invented at the Princeton Institute of Advanced Science in the 1950s.³

The game begins when you and your accomplice are captured following the commission of a crime. The police separate you from your partner and offer you the chance to "give evidence" against your partner. You are told

that she will be given the exact same deal in a separate room. Figure 2 shows the matrix describing the result of your actions and those of your partner (the numbers represent years in prison):

		You	
		Cooperate (Stay Silent)	Compete (Give Evidence)
Your Partner	Cooperate (Stay Silent)	1 Year	Get Out of Jail Free
	Compete (Give Evidence)	5 Years	3 Years

Figure 2: The Prisoner's Dilemma Matrix

If you "stay silent," then you are attempting to *cooperate* with your partner to avoid a long prison term. If you refuse to give evidence against your partner, and she does the same, then you both receive one year in prison (Cooperative Payoff). If, however, you keep quiet and she gives evidence, or *competes* with you, then you go to jail for five years and she is set free (the so-called Sucker's Payoff). Finally, if you both "give evidence" against one another, then the judge sends both of you to jail for three years each (Competitive Payoff). The dilemma resides in the fact that each prisoner has a choice between only two options, but cannot make a good decision without knowing what the other one will do. So what choice should you make? When should you cooperate, and when should you compete?

Conflicts of this nature often occur when two groups are attempting to work together. In software development the two groups are the vendor and the client. The relationship often starts out with good feelings and high expectations from both parties. However, as the project progresses and problems are encountered, communications between the client and the vendor tend to deteriorate. This leads to the situation outlined above: both sides have the option of continuing to cooperate or compete, but neither can make a decision without being able to trust the other party.

Software development is often a case of multiple rounds of The Prisoner's Dilemma. To map this game into "The Software Developer's Dilemma" (Figure 3) we can reassign each sector in the payoff matrix:

- The development organization often wishes to solve the problem in a particular way, regardless of what the customer thinks (**Dictate**).

- Alternatively, the customer may choose to force a pet feature down development's throat or threaten to pull out of the project (**Extort**).
- If both parties argue, then very little progress is made (**Impasse**).
- Only through cooperation is a significant benefit for both parties guaranteed (**Support**).

		Development	
		Cooperate	Compete
Customer	Cooperate	Support	Dictate
	Compete	Extort	Impasse

Figure 3: The Software Developer's Dilemma Matrix

What occurs as the level of trust in the relationship breaks down, most often as a result of a crisis, is that one side or the other begins to seek greater personal advantage. Then, to avoid the "Sucker's Payoff," the other side will move to a competitive stance. This tit-for-tat response leads inevitably to Impasse, until one side gains sufficient power to force its position (i.e., Dictate, Extort). The game is played repeatedly during the life of the project until very little trust exists and all decisions require a large expenditure of time, effort, and emotional capital.

Congruent Negotiation

It should be clear from this model that cooperation between parties is the best choice. So how do we avoid the feedback loop that leads to competition and sub-optimal returns? The key here is good and regular communication between the vendor and the client. Remember, the "dilemma" in both The Prisoner's Dilemma and The Software Developer's Dilemma is caused by the inability of one party to know what the other party is choosing to do. To ensure that mutual benefit remains the choice, it is necessary for both parties to congruently (openly and honestly) negotiate all desired changes to the system.⁴

As opposed to competition, congruent negotiation is seeking a win-win for both parties. No side should seek unfair advantage over the other, because that will lead to sub-optimal long-term gains. In fact, both parties should be clear on the cost of non-cooperation and the negative impact competition will have on the probability of project success.

Here are some guidelines on conducting effective congruent negotiations:

- Establish mutual benefit as a motivational force.
- Negotiate from a position of openness.
- Stay congruent, express disagreement honestly.
- Avoid pointless argument.
- Be willing to compromise, but don't placate.
- Seek a neutral mediator when faced with an impasse.

Business relationships being what they are, during negotiations it is very good practice to establish protective measures for each party to ensure compliance to the contracted agreement. This can be thought of as ensuring the relationship and maintaining cooperative behavior for the best possible outcome. In his poem "Mending Wall," Robert Frost expresses this another way: "Good fences make good neighbors."⁵

Here are some suggested practices that can be used to protect both the vendor and the customer. This list is only a sampling of the possible mechanisms:

To protect the developer:⁶

- State in the contract that the development house owns the code until final payment.
- Agree on clear and reasonable acceptance criteria (given that no software is "defect-free").
- Include a software key that will operate after the date of contracted software acceptance. (This must be clearly disclosed in advance to avoid legal issues.)

To protect the client:

- Clearly state that payment will be provided only for software that meets the agreed upon functionality (e.g., based on use cases).
- Require milestone presentations of progress for continued funding (i.e., functioning executable releases, not just documentation).
- Provide a realistic expectation of software reliability and functionality.

Scope Impact Analysis

As noted earlier, changes to the project scope can and will have a large impact on the success of the project. There will always be the need to accommodate change, but without analyzing the cost and benefits of the desired change, rational decisions cannot be reached. Effective negotiations require not only an understanding between the parties, but also accurate, focused information about the impact the changes will have on the project scope.

Scope is established early in the Inception phase of the project. In order to effectively negotiate the scope, it is necessary to have accurate and useful information on the relative costs and benefits of a requirement. During the project it is of even greater importance to determine the effect of a requirement change on the system under development. To determine the likely impact of additional work, it is necessary to understand the risks, benefits, costs, and effort associated with the change(s). When dealing with requirements it is beneficial to:

- **Categorize** -- to group requirements, permitting a higher-level understanding of relationships and dependencies; use cases are the preferred mechanism.
- **Organize** -- using automated tools (e.g., Rational RequisitePro®) to assist in understanding and tracking of the requirements from introduction to acceptance to implementation.
- **Prioritize** -- to determine the order of consideration based on criticality of need and level of associated risk.

According to the Rational Unified Process®, a project consists of four primary phases, Inception, Elaboration, Construction, and Transition. Inception defines the project goal and initial scope; Elaboration is primarily concerned with system architecture; Construction is focused on creating the final product; Transition is moving the product into the hands of the customer. As can be seen in Figure 4, the bulk of the requirements will be gathered in the Inception and Elaboration phases. However, it is during Construction when changing or additional requirements are most likely to have a large system impact and must be carefully studied for risk, particularly to the system architecture.

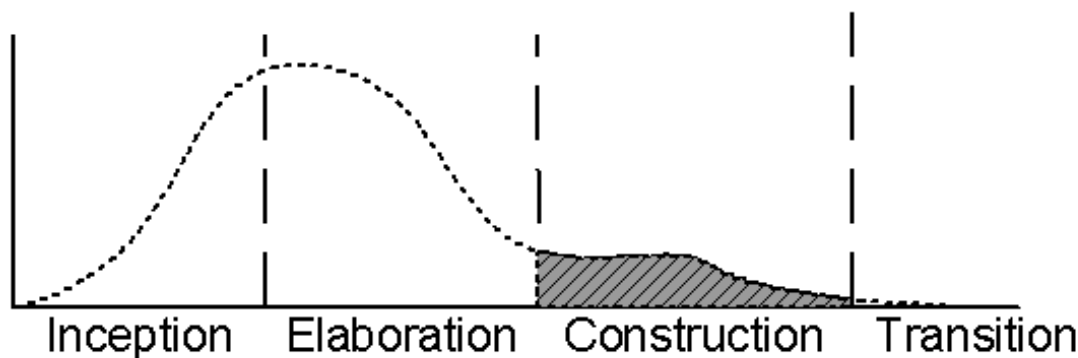


Figure 4: Project Phases and Requirements Load

Requirements are categorized according to Risk, Effort, Priority, and Cost.

Effort is the amount of manpower that must be expended to accomplish a task and can be estimated from a well-developed set of use-case activity diagrams (i.e., by totaling activities),⁷ or by counting the number of pages in a use case (not including title, header, etc.). Alternatively, a fast but qualitative estimate can be made directly by experienced personnel.

A *risk* -- any situation that can negatively impact the success of the

project -- is affected by numerous factors, including:

- Political sensitivity
- Technical difficulty
- Effect on team morale
- Client perception of progress
- Management commitment
- Impact on schedule
- Impact on cost of delivery
- Impact on quality of product
- Effect on system architecture

Of these, perhaps the most damaging is the effect on system architecture. As with smoking, the first short cut is unlikely to kill you, but multiple architectural compromises over the life of the project will inevitably cause serious damage.

Cost is determined by the amount of money/time that must be spent to produce the required functionality. This is therefore a combined factor of effort and schedule.

Priority is usually determined by the customer, but may be determined by the project development team for internally generated system changes.

Use cases (and requirement changes) can be rapidly rated on these categories using a simple scale (High, Medium, Low), as shown in Table 1.

Table 1: Use Case Categorization and Prioritization⁸

Use Case	Effort	Risk	Cost	Customer Priority
UC1	H	H	H	1
UC2	H	L	L	6
UC3	L	M	M	5
UC4	M	H	L	3
UC5	M	L	H	4
UC6	L	H	L	2

The determination of risk is based on the overall success of the project, as opposed to just one system feature. When determining the relative ranking, it is therefore useful to prioritize based on highest risk first, followed by effort and customer priority.⁹

Using this technique allows the project manager to negotiate with the customer based on a solid, qualitative analysis of the relative benefit and risk for a given requirement change. Attacking the high-risk elements early leaves more time in the schedule to deal with contingencies. Moreover, items that have high risk with little direct benefit can be eliminated from the project or delayed to later releases of the software.

Organizing use cases in this fashion will facilitate negotiations by allowing

tradeoffs among risk, effort, and cost.

Effective Scope Management

All software projects must deal with changing requirements. The most powerful tools available to the software organization are effective negotiation skills backed by organized and prioritized requirement sets. Establishing a cooperative environment with agreed-upon verification mechanisms protects both the client and development organization, thereby increasing the level of trust. The result is a project scope that will meet the most critical needs of the client, while permitting the development team sufficient flexibility to produce the software in a timely manner.

Notes

¹ T. F. Crum, *The Magic of Conflict*. Simon & Schuster, 1987.

² "Game theory is a branch of mathematical analysis developed to study decision making in conflict situations. Such a situation exists when two or more decision makers who have different objectives act on the same system or share the same resources. There are two person and multiperson games. Game theory provides a mathematical process for selecting an OPTIMUM STRATEGY (that is, an optimum decision or a sequence of decisions) in the face of an opponent who has a strategy of his own." from [PRINCIPIA CYBERNETICA WEB](http://pespmc1.vub.ac.be/ASC/GAME_THEOR.html), http://pespmc1.vub.ac.be/ASC/GAME_THEOR.html

³ Robert M. Axelrod, *The Evolution of Cooperation*. Basic Books, 1984.

⁴ Congruent communication involves expressing yourself honestly and directly, as opposed to non-congruent behavior, which includes blaming, placating, irrelevancy, etc. For more information on congruent communication, see G. M. Weinberg, *Quality Software Management, Volume 3* (Dorset House Publishing, 1994), and V. Satir, *The New People Making* (Science and Behavior Books, 1988).

⁵ <http://www.robertfrost.org/indexgood.html>

⁶ J. Blumen, "The Prisoner's Dilemma in Software Development." *Ethical Spectacle*, Vol. I, No. 9, September, 1995.

⁷ B.A. Lieberman, "UML Activity Diagrams: Versatile Roadmaps for Understanding System Behavior." *The Rational Edge*, April 2001 (http://www.therationaledge.com/content/apr_01/t_uml_bl.html)

⁸ For more information on prioritization and project scope, see Chapter 20 of D. Leffingwell and D. Widrig, *Managing Software Requirements*. Addison-Wesley, 2000.

⁹ A complementary approach to this technique is outlined in an article by Walker Royce in the May 2001 issue of *The Rational Edge*, in which he suggests conducting a cost/value analysis for added features based on cost of development and associated customer perceived value. (http://www.therationaledge.com/content/may_01/f_sdev2_wr.html)



For more information on the products or services discussed in this

**article, please click [here](#) and follow the instructions provided.
Thank you!**

Copyright [Rational Software 2001](#) | [Privacy/Legal Information](#)