

# CHAPTER 5

---

## Managing Change

### KEY TOPICS

- ◆ Classic Scope Control
- ◆ Managing Scope Change

The *scope* of a project is the set of affordable systems and software that the project team has agreed to deliver.

Defining and managing scope is one of your most important responsibilities. By the end of this chapter, you will have the answers to these three important questions about scope:

1. What are the standard techniques for defining and controlling scope?
2. Why do the standard techniques seem to fail for most Web projects?
3. What are the latest best practices that seem to work for the Web?

### A New Perspective on Scope

---

Most standard models for software development assume a fixed scope that can be clearly defined at the outset of a project before any work begins. This presents a problem, since Web sites constantly evolve in a changing technological and business environment. In this chapter you'll learn to look at scope organically, as a collection of requirements that need to be described,

documented, and managed as they grow. We'll also examine processes that embrace change through the use of *iteration*. These newer approaches allow the requirements to evolve through a series of small releases, showing early results and incorporating client feedback during production.

---

G *Iteration* The process of making incremental refinements to software. The product gradually evolves in a series of working prototypes. These prototypes incorporate client feedback into each release cycle.

---

During its initial stages, scope definition is analogous to a legal agreement between two parties. As you draft the first round of requirements documents, your goal is to create a “contract” between yourself and the project stakeholders, in which you agree on the features of the site you are about to build. The key to success is defining what you intend to deliver and how you plan to deliver it while obtaining the client’s consent before implementation begins. Once work has begun, the project documentation will grow and change with each iteration of the product. Project management theorists have come up with a standard methodology to guide you through this process.

## Classic Scope Control

---

Standard software development models are requirements-driven: They assume that the project team will be delivering a final product with characteristics that can be clearly defined up front. These methodologies provide a disciplined, sequential process for defining the schedule, budget, resources, risks, and scope. For the sake of simplicity, let’s assume a representative process with four stages: Define, Implement, Control, and End (DICE). The hypothetical “DICE” approach has a few distinguishing characteristics.

- ◆ *Define*. All project requirements should be captured in the Define stage. The output of the Define stage is the project plan. The approved plan includes the scope, budget, and schedule. Project work is broken down into a hierarchy of phases, activities, and tasks. The plan serves as a contract between the project manager and the project sponsor. The purpose of this contract is to commit the project team to the terms of the plan. The plan should capture as much detail as possible, and it must be complete before implementation begins.
- ◆ *Implement*. Once the schedule of deliverables has been set, Implementation begins. During Implementation, the project manager assembles and deploys the resources (people, hardware, and soft-

ware) that are needed to deliver what was agreed on during the Define stage.

- ◆ *Control.* During the Control stage, the project manager monitors and reviews the project team's progress against the schedule of deliverables. The project manager also resists the introduction of changes into the plan.
- ◆ *End.* At the End, the product is released. Project success is measured by comparing the final results to the original set of requirements. If the project team delivered against the original specifications on time and within budget, the project is deemed a success. Project failure is usually attributed to scope changes on the part of the client or inaccurate effort estimates up front. This analysis is used to make more accurate effort estimates in the future.

Figure 5.1 illustrates this generic software development process. This diagram is a simplified representation of the "classic" approach, which has been adapted to the Web.

Classic software development processes were designed to hit a fixed target. Unfortunately, the Web presents a moving target, as business models

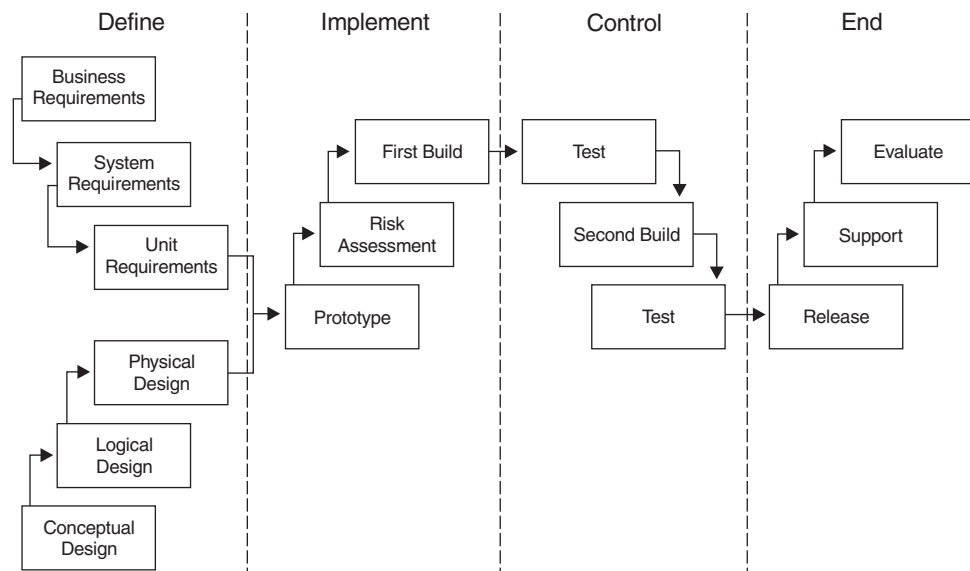


FIGURE 5.1  
DICE Software Development Lifecycle

### ◆ The Standard

It is difficult to identify a “canon” of software development methodology, but a few heavy hitters set the bar for industry standards:

McConnell, Steve. *Software Project Survival Guide: How to Be Sure Your First Important Project Isn't Your Last*. Redmond, WA: Microsoft Press, 1997.

*Guide to the Project Management Body of Knowledge (PMBOK®)*, 2000 Edition. Newtown Square, PA: Project Management Institute, 2000.

Rational Unified Process®: The RUP is a Web-enabled set of software engineering best practices. RUP is used today by many organizations that develop e-business applications. See <http://www.rational.com>.

and implementation technologies are reinvented at an exhausting pace. In spite of this mismatch, the majority of current best practices in use by Web project managers today are inherited from this classic model.

### ***The Project Web Site—Getting Everyone on the Same (Home) Page***

Document and version control is essential to scope management. If your team can't easily access the latest project documentation, they cannot know the scope of your project. Once your requirements documentation has been created, it needs a home. Important documents should be easily accessible to all members of the project team. As the scope of the project evolves, “Who has the latest version?” becomes a very popular question! A project Web site should contain links to all of the latest project documentation, upcoming milestones, and the date of the most recent updates. As the project manager, you should maintain the project Web site, which will solidify your role as the “chief librarian” for all project documentation. Here are a few tips.

- ◆ Use a file-naming convention that includes author, version, and date.
- ◆ Only one person should be responsible for maintaining the most recent copy of a document and uploading it to the site. (That means *you!*)
- ◆ Do not be lazy by simply overwriting older files with new ones. Be disciplined about keeping backup copies of older versions.
- ◆ Provide links to an archive of previous versions as well as the most recent version.

### ◆ Project Web Sites Made Easy

Maintaining and updating the content on your project Web site can turn you into an HTML slave. Avoid the headache by using Web logs. “Blogs” are free, easy-to-use, customizable publishing tools that can be placed directly on your own Web server or used as a third-party service (<http://www.blogger.com>).

Several service providers offer customizable project extranets at low cost or even free of charge! Two of them are:

- ◆ eProject Express (<http://www.eproject.com>)
- ◆ Intranets.com (<http://www.intranets.com>)

- ◆ Provide multiple file formats for project team members who do not have Visio, MS Project, or other software. For example, export Visio diagrams as .GIF files. Export MS Project documents in Excel. Specify the file format and file size alongside the hyperlink.
- ◆ Upload meeting notes and status reports in addition to the usual project documentation.
- ◆ Include a project team roster with contact information (including instant messenger handles), titles, and a brief description of each person’s roles and responsibilities on the team.
- ◆ Use a Web log system like Blogger ([www.blogger.com](http://www.blogger.com)) to make the job of updating content on the site easier.
- ◆ Include links to the various rounds of graphic design mockups, prototypes, and demos in addition to technical documentation.

## Managing Scope Change

There are several change management techniques you can employ to avoid an adversarial relationship with the project sponsor.

### ***The Project Triangle—Scope, Schedule, Resources***

A cornerstone of the project manager’s role is to ensure that stakeholders understand the tradeoff between scope, schedule, and resources. Tact, diplomacy, and good negotiating skills will go to waste if the client does not understand where their project sits along these three axes. The customer must be able to articulate which legs of the constraint triangle she is willing to change

### ◆ Point–Counterpoint: Stakeholder Tradeoffs

When confronting the unforgiving tradeoffs of the project Triangle, always be a polite problem solver. You should be assisting stakeholders to navigate through difficult choices. Compare these two reactions to a change order that was issued late in the project.

“As the project manager it is my duty to inform you that we just can’t cram this feature in and still make the deadline. It’s entirely your choice, but if we issue this change order, there is no way my team can hit the deadline. We can’t commit to the date if you really want to have this new feature. I must be frank and up front about this so your expectations will be in line with reality. The other option is to add this feature to Phase 2, which I strongly recommend. We can also hire additional consultants, but you must decide that you’re willing to waste

\$20,000. We’ll need a decision within two days so we can revise the project plan.”

“Yes, we certainly can build this feature! I’ve read the description, and I think I understand what you’re asking for. In order to build it we’ll all have to make some tough decisions. The work that this new feature requires would push us over the deadline, given our current staffing. One solution is to bring additional consultants on board. I ran this scenario on the project plan, and this option adds about \$20,000 to the development fees. Optionally, we can move this feature into Phase 2. I suggest that we all sit down and make a list of the consequences that might result from pushing back the deadline. Then we’ll measure these against the benefits of having this feature in Phase 1 and consider the budgetary impact of the \$20,000.”

and understand their interdependence. This is a communication challenge that must be met with careful education and reinforced at every opportunity. Rather than being the “project policeman,” a more effective approach is to assume a helpful “consulting” role that lasts throughout the duration of the project. As the helpful consultant, you assist the project stakeholders in navigating all sides of the project triangle.

### ***Getting Project Documents Approved by the Client***

As you collaborate with the project team to draft all of the requirements, you will also have to get those documents approved by your client, who will try to squeeze as much work as possible out of your team. The tug-of-war begins as soon as you post the first page mockups to the project Web site. Your new client may drag your team toward defeat by adding features while the deadline remains fixed. Other clients try to cut costs by haggling over every detail as they try to wring a few extra hours out of your overworked developers, or perhaps the first draft page mockups have launched the client into a brainstorming hurricane that spins aimlessly with no end in sight. Threatening to

quit seems like a good option, but you'll have better luck scheduling a priorities meeting and applying a few time-tested techniques.

### **The Chinese Take-Out Menu Approach**

A common approach to obtaining sign-off is to help the project stakeholders prioritize features. Once you have all of the decision makers in the room, pull out a complete "wish list" summarizing all of the features that have been suggested. Ask everyone to prioritize the list. If the room explodes into a frenzied debate, add structure by having each stakeholder work independently. After everyone has ordered his or her own list, ask them to present their list to the group with the reasons for their selections. As the group moves toward consensus, priority conflicts will emerge. The debate over some features will become deadlocked, and stakeholders will ask for more information about cost. When you hear "I can't decide—it depends which one is cheaper," then it's time to add the price column to your Chinese take-out menu.

Now that you have the prioritized list and you've taken note of the deadlocked or "tied" combinations, you can bring on the implementation team. Use a flip chart or white board and create two columns, one for time and the other for money. Don't provide actual figures in terms of days or dollars, since this information could compromise your negotiations with the client. Simply assign a scale of 1 to 5, from low cost to high cost. Determine which lower-priority features may be bundled with higher-priority features—for example, "If we decide to provide online music samples, then we can also include the film clips at minimal extra cost because they both use the same streaming media server."

During this discussion, be careful to recognize the intrinsic value of software that you may be repurposing, as well as the "sunk costs," R&D, and other up-front investments that were poured into your technology infrastructure. Just because you've built a highly scalable solution that only takes an hour to adapt, it does not mean that the software is only "worth" an hour of a developer's time.

As you make progress, take a moment to introduce the concept of "Phase 2." Some of the riskier features may require a proof-of-concept or prototype. Consider developing "pilots" to test the business validity of a feature. Take baby steps. Create an e-mail newsletter campaign as a one-off, and send it to a small target list of customers before investing in the million-dollar publishing interface. Build a long-term relationship with your client through a series of small successes, and avoid the disastrously expensive "white elephant" or "Maginot Line" projects that become legendary failures.

### **The “Surgeon” Analogy**

Cost is naturally a major roadblock to getting a client to approve a set of features. The “surgeon” analogy is a technique for helping the project stakeholder understand the hidden costs of software development (“Why are you billing him so much for tasks that appear to be no big deal?”). When discussing cost tradeoffs, some project stakeholders may try to reduce the terms of the discussion into pure man-hours. They will assume that the faster an application can be built, the less expensive it should be. These clients are computing the value of your work based on man-hours alone. Remind the client that graphic design and software development are not like automobile repair, factory production, or bricklaying. A better analogy is the complicated surgical procedure. In the case of laser eye surgery, the procedure takes only a few minutes, but the doctor is being paid for years of knowledge and expertise. The cost of a medical procedure reflects the thousands of hours of training and preparation that qualify the surgeon to provide a service that is both technologically complex and customized to the patient’s needs.

Furthermore, rapid deployment and decreased time-to-market provide significant competitive advantages. The scalability of your team’s software is a valuable asset. Your team should be rewarded for developing its expertise and infrastructure to the point that you can produce a quality product in a short amount of time.

### ***Playing Defense***

Project managers seek to define the project and then build a protective wall around the initial set of features. Scope “creep” usually manifests as a gradual erosion of this wall. Most project stakeholders are sensible enough to quell major changes, but the cumulative effect of tiny incremental changes takes its toll. How do you prevent new requests from coming in over the wall? Formalize the process. Establish a change committee and a formalized change procedure so that you will not find yourself desperately plugging leaks as small requests pour in from all directions.

### **Change Orders**

Use a written change request document whenever there is a change in scope or requirements. The change request form should include mention of the relative priority of the change and the importance of the change in terms of time and resources. Be consistent about requiring the use of change orders, or the client may assume that a change has no cost impact. The purpose of this protocol is to manage expectations and costs, not to discourage change or to appear inflexible in the face of a dynamic business environment.

### ***Problems with Classic Approaches***

When your project feels like it is spinning out of control from scope “creep,” you may be tempted into a destructive battle with stakeholders. From your standpoint, the objective of this tug-of-war is to aggressively limit the scope of deliverables. After the scope has been “set” and the project is underway, you will be called upon to battle the client to prevent changes. “Scope management” becomes a reaction to changes initiated by the project stakeholders, who are laying siege to the integrity of the original plan. Caught in a destructive dynamic, your team will begin to feel like a defensive garrison as you fend off salvos of change requests that threaten to erode the fixed set of features. The traditional “siege mentality” creates an adversarial climate that quickly infects the entire project team. The fallout from this power struggle between the project manager and project sponsors sets a tone of conflict that can last for the duration of the project.

In this hostile environment, traditional project managers focus on the plan, tracking inputs, deliverables, and milestones against a fixed set of requirements. Unfortunately, the attempt to specify every feature ahead of time and draft a static plan has proved to be unrealistic for Web projects. During the lifetime of your project, stakeholders will learn about the technology as well as their own business needs. The overall business climate may change if the project lasts several months. In response to this reality, innovative approaches like Rapid Application Development (RAD) and Extreme Programming (XP) have taken the spotlight. These methodologies share an “iterative approach” to scope management.

### ***Iterative Approaches***


Given the limitations of the standard model when applied to the Web, you will be called upon to find creative ways of controlling requirements throughout the lifetime of the project. The details of implementation will cause the specifications to be modified on the fly. Several new methodologies have formalized this iterative process. RAD is one of the most commonly accepted iterative approaches.

The Rapid Application Development methodology was designed to build software with speed as the most important success criterion. The RAD approach is not appropriate for every Web project. RAD is best suited for projects that have a limited, well-defined scope and a measurable outcome. RAD works best with a small, tightly knit project team and an on-site stakeholder who is empowered to make quick decisions about functionality. The ideal team size is under ten people. RAD also requires a stable technical

architecture: It is not well suited for creating large, complex systems on top of a completely new or untested infrastructure.

The cornerstone of RAD is rapid prototyping, wherein developers try to create a small working product as soon as possible. The working prototype is then refined based on direct feedback from the client. Each refinement, or “iteration,” is a step closer to the finished product. RAD project managers use a technique called “timeboxing,” in which scope is allowed to change but the delivery date remains fixed for each iteration. The main advantages of RAD are that the client is able to see results right away and frequent scope changes are allowed. With conventional methods, there is often nothing delivered to the client until 100 percent of the process is finished and the completed software is unveiled.

Extreme programming represents one of the most notable applications of RAD techniques to the Web. This client-centric approach aims to deliver just enough software to meet the customers’ needs, incorporating the immediate feedback of clients during the development of a series of rapid prototypes. In fact, there is no distinction between the “prototypes” and the finished product. XP utilizes several innovative techniques like “pair programming” to encourage teamwork, creativity, and collaboration among developers. The interview section of this chapter describes how XP works in the real world.



**Extreme Programming (XP) Resources** See <http://www.extremeprogramming.org/>  
<http://groups.yahoo.com/group/extremeprogramming/links>.

## Common Scope Headaches

---

Most day-to-day problems with scope are caused by weaknesses in the project definition process. While you work to create a methodology for managing scope that is high on concept, keep an eye out for these nuts-and-bolts breakdowns. This section describes the most common headaches, along with the cures.

### ***Problem #1: I Sketched the Site Out on a Napkin— Is that Okay?***

It’s difficult to draft project specifications if the initial requirements were poorly defined by the customer in the first place. The “cocktail napkin blueprint” can be used to memorialize a moment of inspiration, but it is not a valid input for your specification. Sure, you can “wing it” and attempt to fill in the

blanks for your client, but this guessing game usually ends in dissatisfaction with an end product that doesn't match your customer's expectations.

### Symptoms

- ◆ There is no creative brief or written document signed off by the client that states the scope of the project and its features.
- ◆ There is a creative brief, but it is vague or poorly written.
- ◆ The project stakeholders are new to the Web or inexperienced with user interface design.
- ◆ The developer has numerous questions about the specifications.

### Solutions

- ◆ Build a feature inventory by asking the client to list all the items that appear on each page.
- ◆ Build out a use-case scenario by walking a hypothetical user through the application and asking the following questions.
  - What are the user inputs?
  - What sort of output should users receive?
  - What are all the possible actions a user might take?
- ◆ Find an example of a similar site that can be used as a model. Ask the customer how her vision differs from the example or how the model should be modified.
- ◆ When all else fails, take a stab at designing the application yourself. E-mail the specifications to the project owner with the alternate features and design choices presented as a series of questions highlighted in red. As the client answers the list of questions, he effectively builds the specifications.
- ◆ Circulate early versions of the spec to a tech lead for a brief review and obtain a list of questions for clarification.

### ***Problem #2: It's Nice, But It's Not What We Had in Mind.***

So you did a great job writing a 20-page specification, using plenty of technical jargon to impress the developers. Unfortunately, your project is in trouble because your nontechnical client didn't "get it" until round one of the design popped up on her monitor. The fateful words "We didn't know it was going to work like *this*" spell disaster for your deadline. In order to prevent drastic revisions late in the game, steps need to be taken right now during scope definition.

**Symptoms**

- ◆ The client signed off on a feature summary but never reviewed the detailed specifications.
- ◆ The client has very few questions about the specifications.
- ◆ The client makes radical scope change requests late in the design phase.

**Solutions**

- ◆ Conduct a face-to-face review of the spec where each feature is discussed in detail. Many project stakeholders do not read through e-mail attachments.
- ◆ Be aware of different comprehension styles when you present the final specifications to your client for sign-off. Convey your concepts visually, orally, and in writing.
- ◆ Include visual mockups in your specs in addition to text explanations.
- ◆ Identify all the project stakeholders and have them sign off on the specifications.
- ◆ If your project owners are inexperienced, do not sign off on the specs until there is a demo/front-end prototype or finished design mock-ups. Build an HTML skeleton or “shell” of the application and walk stakeholders through the demo. Many inexperienced stakeholders don’t understand what they are getting until they see it on their monitors.

***Problem #3: Just One More Tiny Little Change . . .***

The cost of changes tends to increase as the launch date approaches. However, your client may not realize that creative brainstorming and “tweaking” just isn’t appropriate during system testing!

**Symptoms**

- ◆ The project deadline is repeatedly pushed back.
- ◆ The client can submit feature changes without incurring any cost.

**Solutions**

- ◆ Draft a detailed scope document or specification and require a formal sign-off before any work begins.

- ◆ Break payment schedules into a series of installments for each itemized deliverable rather than a single payment for the finished product.
- ◆ Create a formal change order procedure and assign a cost to change orders by making arrangements for additional fees or deadline extensions. Require change orders to be authorized by the ultimate project sponsor.
- ◆ When drafting your documentation, include features that are *not* required. Ask stakeholders to identify categories of requirements that will not be included, like credit card transactions or personalization features.
- ◆ Circulate updates to the project plan and specifications after each change order.
- ◆ Try an approach that employs rapid prototyping techniques. Shorten the release cycle. Establish an open-ended “burn rate” fee structure, and invite the client to provide continuous creative input.

## Summary

---

Your job as a project manager is not to prevent change but to help your stakeholders navigate through the inevitable tradeoffs between scope, schedule, and resources. Avoid assuming the role of the “scope police” and become your client’s trusted partner, educating stakeholders so that they can make intelligent choices. The inevitability of change may cause you to throw up your hands in despair, but keep in mind that you are managing a creative process in a dynamic communication medium, not building a suspension bridge. Attitudes and methodologies inherited from civil engineering, the military, and other project management cultures may not fit the open-ended, creative world of Web site design.

Although they represent an exciting development, iterative models are not a panacea to the problem of scope change. Many prototype-centric methodologies require unique conditions for success. For example, rapid prototyping requires a client who is knowledgeable about the Web, responsive, creative, and willing to participate actively in the ongoing design process. Such clients are a very rare breed! The techniques of pair programming advocated by XP enthusiasts often require significant changes in organizational culture and work habits, as well as a suitable personality! While it is true that many new approaches promise to be better suited to the Web, it is advisable to first acquire a firm foundation in commonly accepted practices. Once you

have mastered the fundamentals of scope management, don't be afraid to experiment on small projects. As you work toward mastery of the basics, stay abreast of novel approaches that offer a treasure trove of techniques that will assist you in confronting change.

In the end, change can make your project better. Often the best ideas emerge in the late stages of the graphic design phase, originating as an unwelcome "suggestion" from a client that threatens your stranglehold on the scope. As you prepare to embrace change, be forewarned that its inevitability is not an excuse for vague or shabby specifications. A clear and well-defined set of requirements will go a long way toward managing the client's expectations. Solid specifications will provide a baseline against which you can measure the costs of future enhancements and a valuable addition to your company's knowledge base. If you've done a thorough job of defining the product, you will enjoy a smooth transition into the next phase—creating the plan.

## EXTREME PROGRAMMING

*Alex Cone, CEO of CodeFab Enterprise Development (<http://www.codefab.com>), talks about using Extreme Programming to move beyond the traditional methods for managing scope.*

This interview was conducted in October 2001 over a few margaritas at Tortilla Flats restaurant in New York City's West Village. The restaurant served up a spicy mix of retro 1950s paraphernalia and Mexican home cooking to local patrons. The booths were packed with art gallery owners, bikers staggering in from meat packing district bars, and Silicon Alley computer geeks taking a break from their labors in the raw industrial loft spaces that litter the neighborhood.

After a distinguished career developing back-end applications for Wall Street trading systems, Alex teamed up with some of the top developers in town and set up shop in a warehouse space in 1997. Since then, his team has abandoned traditional software development methodologies to produce innovative applications for clients like Apple Computer and Standard & Poor's.

**So we're writing a book about how to get Web projects done. Any advice?**

When we started CodeFab, the first thing we did was look back at most of the software development projects that we'd done over the last 10, 15 years, and conclude that most of them had failed.

**You mean failed from a process point of view?**

They failed from the point of view that they didn't get finished in the timeframe or within the budget, and we generally weren't happy with the end results. We wanted to know why. We wanted to actually finish projects and do them the right way. By and large we've been successful over the last four and a half years that we've had this company. At the two-year point, I had already finished more software projects than I had in the previous 15 years. I actually finished them and delivered them to a satisfied client: They were a success.

In most of my experiences on Wall Street, you started some grandiose project, people worked on it for a while, and then all the significant players left. It was never finished, things got changed along the way, and you never saw a completed piece of software that matched what you set out to do.

We started examining the current wisdom on software development and why software projects failed. At that time the point person was Steve McConnell, who had written a number of good books, including *Code Complete*, *Rapid Development*, and *The Software Project Survival Guide*. There was some really good thinking in there, but much of it focused around the cost of introducing changes into your specifications and controlling the scope of your project over time, as well as management issues in terms of keeping on track with what you're doing and the importance of having more or less complete requirements up front. This works very well if your focus is to be able to do fixed-scope projects.

The traditional approach was to do a whole bunch of work against a set of specifications and deliver what was in the specs before moving onward. It seemed like a good way to go, and it certainly provided good backup in terms of a contract with the client. When you're two weeks into the project and the client wants to change something significant, you can say, "Look at the cost associated with making a change at this point."

However, by and large this didn't really work for us. We had some notable failures, and part of it was our fault and some of it was the fact that this development model really didn't fit with the Web. One of the primary tenets of the standard process was that you can know at the start of your project what it is that you want to do, what you'll need over the next 6 or 12 months, and that those needs will remain consistent over that time period. And this has turned out to be completely unrealistic. It was also founded on the idea that you could induce a client to actually describe in sufficient depth and detail exactly what they wanted you to do up front, before any work had been done yet. And that also turned out to be impossible. You go out of your way to discourage the client to change the project in midstream. You needed a change control committee, you billed them extra fees for changes, and so on. That tended to foster a very bad working relationship with the client.

In fact, this adversarial relationship was the source of our biggest problems. Basically, it forces you to have a huge fight up front with the client, wherein you wrestle over the features and the specs and the cost and so on and so forth. The client is trying to get the cost down as far as possible, while you're trying to limit the features. You're trying to get detail out of them when they don't want to be detailed, and then you have this huge fight before you even start the project.

Once the project gets going, you have another big crisis every time the client has a brilliant idea about how they would like to do something. This is because the client wants you to accommodate the change at no cost. Naturally, you show the client your documentation, which proves that the new feature is going to cost a lot of time, energy, and money—so you think you should be paid for it. The brawling continues.

Finally, when you finish the project, you have a big fight at the end. "We're done." "No, you're not." "Yes, we are—see, here are the specifications showing that we did exactly what we said we were going to do." The client is only trying to be financially prudent. This is their opportunity to get you to do more work for free by saying, "Finish this, change this slightly, try and do this," but you're trying to be financially prudent by saying, "Look, if we do another development hour, that is costing me money, and you're not paying me anymore for doing that, so we have to draw the line." So after you wrestle each other to a total standstill, you're probably not in a good head for doing the next project with this client, and he's not psyched to work with you, either.

**So the main problem you've identified with the standard methodology is that the customer does not really know what they want, and, to make matters worse, you're working with a new technology that is changing rapidly along with the business environment.**

Right, and you're starting off with some assumptions that are fundamentally unreasonable. You can't know exactly where you want to land. This isn't some kind of a

ballistic missile. This is like trying to drive to Boston by pointing the front wheels of your car exactly in the direction of your destination address in Cambridge. Then you take your hands off the wheel and just press on the accelerator and hope that you'll get there. That isn't how you really drive to Cambridge. Making such complicated journeys is about midcourse corrections and about being adaptable. The traditional process fundamentally rejects change rather than embracing change.

The only advantage of the standard methodology is that it fits well within the corporate consulting mindset. The corporate client says, "We want to have this e-commerce Web site up by June. I've got to go back to the budget people and get a check for this piece of development. How much will it cost me?" The only people who really succeeded at this were the overpriced consulting agencies who would say, "Okay, then, I'll pull a number out of my butt and completely pad it with a \$2 million markup." For a while, clients actually said okay to this, but now we're back to square one.

**So if the situation is unworkable unless you pad your estimates by some ridiculous amount, then what do we do?**

We really needed to start this whole process off on a completely different foot. One of my developers came to me with this new methodology, started by this guy Kent Beck, who was one of the original developers of Perl. It's called extreme programming, or XP. We started working with this, and we basically came up with a CodeFab version of XP. We worked on a couple of projects with this and found that it addresses pretty much all of our concerns.

The basic idea is—to continue my driving analogy—you agree that we're heading toward Boston, but all we work out initially is how to get to I-95. We'll take the next step once we're on I-95. The client comes in, and you put together a "story" rather than trying to do functional specifications. By trying to write these highly detailed specs, you are struggling to do technological implementation during the wrong phase. You're focusing on the details instead of the goal.

My emphasis on stories is that we want a good customer experience. For example, one story says, "We want to build the application so that the next time the visitor comes to the site we should know what they wanted the last time they came to the site." As you start programming, you come back to this one-paragraph story that describes the idea behind the user experience. You're not focusing on the features. You don't care about specifying the details like "there should be buttons on the left side instead of the right side, and the user should be able to turn on one-click buying with a checkbox."

**So the specifications are experience-driven rather than widget-driven. The specifications are just stories about what the user gets out of the experience.**

True, but there are a couple of restrictions on the client. The client has to be willing to put somebody on site. The client has to be part of the development team because you are doing short iterations. I can never get more than two weeks away from a working version, but the client has to be able to fine-tune things all the time. For

example, there may be two different solutions to a problem, so we have to ask the customer, “Which one do you like?” Or we say, “We’re at this point, and we could do any one of these three stories next—which one is the most important?” It really requires close interaction on the client, and that person has to be somebody with authority to make a decision. It can’t just be somebody who makes a phone call, because we can make a phone call ourselves. This system requires some serious participation and accountability on the part of the client.

**The actual process involves a series of prototypes, developed rapidly with immediate feedback from the client.**

You’re doing just enough to get the story to work. And then you move on to the next story and get that to work. You build this stuff up. You don’t think to yourself, “Okay, we need a grand shopping cart infrastructure before we start anything else. We need a network communications layer before we start anything else.” You just do a little bit and then get back to work and do a little bit more. Everything is based on making the stories come true. You iterate in the direction that you want to go. When the client comes in on Monday and says, “I had a great idea over the weekend,” you can say, “Okay, great, let’s do that now.”

**How do you get scalability into your software design if you are doing a series of one-offs?**

They are not one-offs. You are just adding more features to the baseline. They are not prototypes. They are the real thing. You just make the specs into a story, like “We should be able to handle a thousand users shopping at the same time.” The trick is to make it work, make it work right, make it work fast.

The major tenet of the whole XP thing is this refactoring. As you get a new requirement, you are not afraid to rewrite the code to accomplish a new goal. It steers you away from a tendency to design some vast infrastructure that you may not need. You avoid maintaining all of this useless code just because you might need it. You use just enough. You refactor and optimize the stuff that isn’t good enough. You don’t work on any part that isn’t actually a problem.

This is a classic problem with all software development. People tend to optimize too early and optimize the wrong thing. If you find out that one spot is 80 percent of your problem, then you put your efforts there. Don’t try and solve a problem if it isn’t a problem yet. The client is right there, saying “It’s fast enough to go live with now.” If it’s not fast enough to go live with, let’s change this one piece. Let’s stop adding new features and make this one thing faster.

**If the product evolves on the fly, how do you handle billing?**

Basically the client hires a team on a burn rate—say, for \$150,000 you get these four guys and their project manager for a month. And we just continue iterating, executing stories, testing, rolling out a version for as long as the client wants. When are we done testing? When the client says we’re done testing. When have we got enough features built into the product? When the client says we have enough features.

**So the client has to manage their own scope!**

Absolutely. But they get lots of short-term feedback. They can take shortcuts if they want to. They can say, "We need to go live with this feature, so we'll just iterate in that direction until we get something that is good enough." They can say, "Now I want to work on the user interface. Now I want to work on the communication with the CRM software." The client can really drive the process.

Since the development team has been doing a series of short-term iterations, they are never more than two weeks away from a working version. The client always has something to play with. The client can always refer to a working version, which allows them to base their ideas and suggestions on a realistic model. Plus, we do other things. We integrate unit tests into everything right up front. You cope with the story, write your first test and run it, and it fails because you haven't written any software. You write the software until it passes the test. If you want to make some changes, you re-run all the tests. If the code passes the tests, then you have a piece of software that does all the stories you've defined so far. You get much higher quality software that way.

We also do a lot pair programming, which is very nice. One person types, and the other person navigates, and we keep rotating people around. This way everyone sort of takes ownership of all of the code. Everyone has a piece of developing the product. It's no longer "This is the JavaScript guy; this is the guy who does all of the database stuff." Everybody gets a piece of everything, and you don't have to study a bunch of documentation to figure out what the other guy did because you worked on developing it. You have the developers volunteer to take on certain tasks. You write them up on 3 × 5 cards like you saw on the wall, and people volunteer to do them. This way the client isn't paying for twice as many programmers as they need, and generally after a month or two they are completely sold on the process.

The client feels like part of a team is their team and that they've come up with an idea and have seen it turned into reality very quickly, as opposed to fighting through some change control committee and then six months later finding out whether the feature was useful or not. My experience is that clients change what they wanted, or even what they thought they wanted, all the time. The changes don't necessarily have anything to do with what they were originally thinking.

**How do these clients manage the overall budget?**

They basically have to accept a degree of uncertainty. It usually takes them a month or two or three to realize that we've got such velocity on this project that it's worth the ambiguity.

In the traditional model, the guy who is writing the check never looked over a set of detailed requirements anyway. The budget guys are just trusting the client manager to pick the right features and do the right implementation, so we don't need to be that much more specific. Either they trust him or they don't. And if there are specific requirements that can be conveyed to us up front, so much the better.

Nonetheless, that person continues to be responsible all the way. And it eliminates the classic “It’s the consultant’s fault,” which generally was just a smokescreen for a person who could never make up his mind about what he wanted.

**I guess the client sells this model to their manager by virtue of its flexibility.**

Right. Basically the client adds features until they are happy with the feature set or until they can’t add more features within the budget. They’re deciding what the next feature will be and what features do not make the cut because they will take too long to develop.

We want to shift the burden of expectations and prioritization back to the client and make someone a proactive participant in the process rather than having a project manager who is placed in an adversarial relationship with the client. The client gets a lot more responsibility out of this process, and they also feel tremendously empowered.

**How does the project manager’s role shift in this relationship? They are not necessarily managing deliverables because deliverables are being established by the client in concert with the developer.**

The project manager sort of glues everything together. They are responsible for getting the stories and the tasks written up and estimated, coordinating who is doing what where, and also overseeing interaction with the QA people. Since you start with unit testing right away, QA people get very involved. And there is still the matter of putting together project documentation that has to go through the client’s approval process, so the project manager does all that. Plus the project manager is still the primary point of contact for the client. They make sure we have the right documentation and document status reports on what was actually accomplished. Yes, there is a board with a bunch of  $3 \times 5$  cards on it, but there needs to be more formal status reports conveyed.

I love this process because I know I am getting paid fair value for the developer. They work as long as the client wants them to work, and they are being paid for it. If the client wants more developers or less developers, we can adjust. I avoid the typical scenario wherein we’re working like mad for the last three months, for the final one-sixth of the overall project payment. And what’s more, you are in such a perfect state to go on to phase two.

Also, you’re in a perpetual brainstorming mode, a creative process as opposed to being on an assembly line process. Developers often feel like they are on the receiving end of a directive, and if you’re on the assembly line, you’ve got to churn out whatever the directive tells them to churn out. There is no creativity.

With this system the developers are empowered to volunteer to take on this or that story and estimate tasks. Then you reiterate, compare the estimates to the actual time, and calculate your velocity or your fudge factor and estimate accordingly. You get more and more accurate about estimating over the course of the project.

**Alex, what's your definition of a story, and how do you measure it?**

You're trying not to describe technological implementation so much as what the user does and what outputs he receives. No longer than a paragraph. Something that can be easily communicated so that we can say, "Can we make this come true? Do these features make this true?" There is a strong emphasis on using metaphors. You want to be able to give people a good sound bite so they really get a feeling for what this is supposed to do.

**Give me an example of a story from one of the cards that would appear up on the wall.**

We're doing this distributed publishing product for these Japanese guys. It's basically a Japanese car magazine, but they have thousands of distributors and editors who are on the road all over the place. They want to put a contributor on the road with his laptop. He should be able to upload an article, upload the images that go with the article, and do it all remotely from a laptop over a wireless connection. The users should be able to search for an article by title in English or in Japanese, and the user should be able to find articles pertaining to cars by the model year. That's not saying anything about what their search algorithm is or what the tool is that allows them to upload pictures pertaining to the article.

The stories can change over time. We started out with a Web-based solution. Then we came up with a solution that integrates directly into Microsoft Word, but the story continues to be true. The developer might break the story down into tasks and provide a broad effort estimate for the story. And, of course, everything continues to be recorded on 3 × 5 cards, and you work on things until they are done. Then you mark them down and put them up on the board. The project manager is measuring the velocity by saying, "You did that in half a week instead of two weeks."

**What makes a good project manager really great?**

Good people skills, good organizational skills, willingness to be perceptive about the client and the client's mindset, good ability to digest fairly complex technological issues and translate them into something that the client can really inhale, good ability to work with the employees and balance all the egos. There is still a lot of developer ego management. Making sure that two people don't pair with each other all the time or that nobody leaves anybody out in the cold, making sure that developers are not spinning their wheels for too long on a problem and that they are respecting decisions that were laid down about how we are going to do this or that or the other thing. You are still doing client management. You are still doing status reports. All the clients are different, and they all want weird stuff.

One of our top project managers is a nationally ranked pinball champion and often competes in championships. One of my developers brought us XP and insisted that we use it and is fanatical about it. He keeps sending e-mail messages with sound bites about why this is good or bad or evil or whatever else. But I am actually trying to encourage developers to be in this mode. Part of this is like trying to be a housing coordinator at a college, trying to match roommates.

