jama build great products™

FOUR TECHNIQUES FOR DEFINING PROJECT SCOPE

Every software team talks about project scope and team members often complain about unending scope creep. I'll present some definitions, describe four techniques for defining project scope, and offer some tips for managing scope creep.

Unfortunately, the software industry lacks uniform definitions of these terms, and the requirements literature is short on clear guidance regarding how to even represent scope. In this whitepaper, adapted from my book More about Software Requirements1, I confront scope head on.

VISION AND SCOPE

I regard the vision and scope document as a key software project deliverable. You can find a suggested template for this document at the Process Impact website. Other terms for this type of guiding document are a project charter, market (or marketing) requirements document, and business case. You don't necessarily need a standalone vision and scope document for a small project. Any project of any size, though, will benefit from such strategic guidance, even if it's just a paragraph or two at the beginning of the software requirements specification.

Both the vision and the scope are components of the project's business requirements. I think in terms of the product vision and the project scope. I define the product vision as: "A long-term strategic concept of the ultimate purpose and form of a new system." The product vision could also describe the product's positioning among its competition and in its market or operating environment. Chapter 5 of my book, Software Requirements, 2nd Edition, describes how to write a concise vision statement using a simple keyword template.

I'll define project scope as: "The portion of the ultimate product vision that the current project or iteration will address. The scope draws the boundary between what's in and what's out for the project." The second part of the project scope definition is most important. The scope identifies what the product is and is not, what it will and won't do, what it will and won't contain.

A well-defined scope sets expectations among the project stakeholders. It identifies the external interfaces between the system and the rest of the world. The scope definition helps the project manager assess the resources needed to implement the project and make realistic commitments. In essence, the scope statement defines the boundary of the project manager's responsibilities.

[1]Adapted from Karl E. Wiegers, **Testing the Requirements,** Microsoft Press, 2006.

Your scope definition also should include a list of specific limitations or exclusions—what's out. Obviously, you can't list everything that's out of scope because that would include every detail in the universe except for the tiny sliver that is in scope for your project. Instead, the limitations should identify capabilities that a reader might expect to be included in the project but which are not included. I know of a project to build a Web site for a national sports team that included the following exclusions for the initial release:

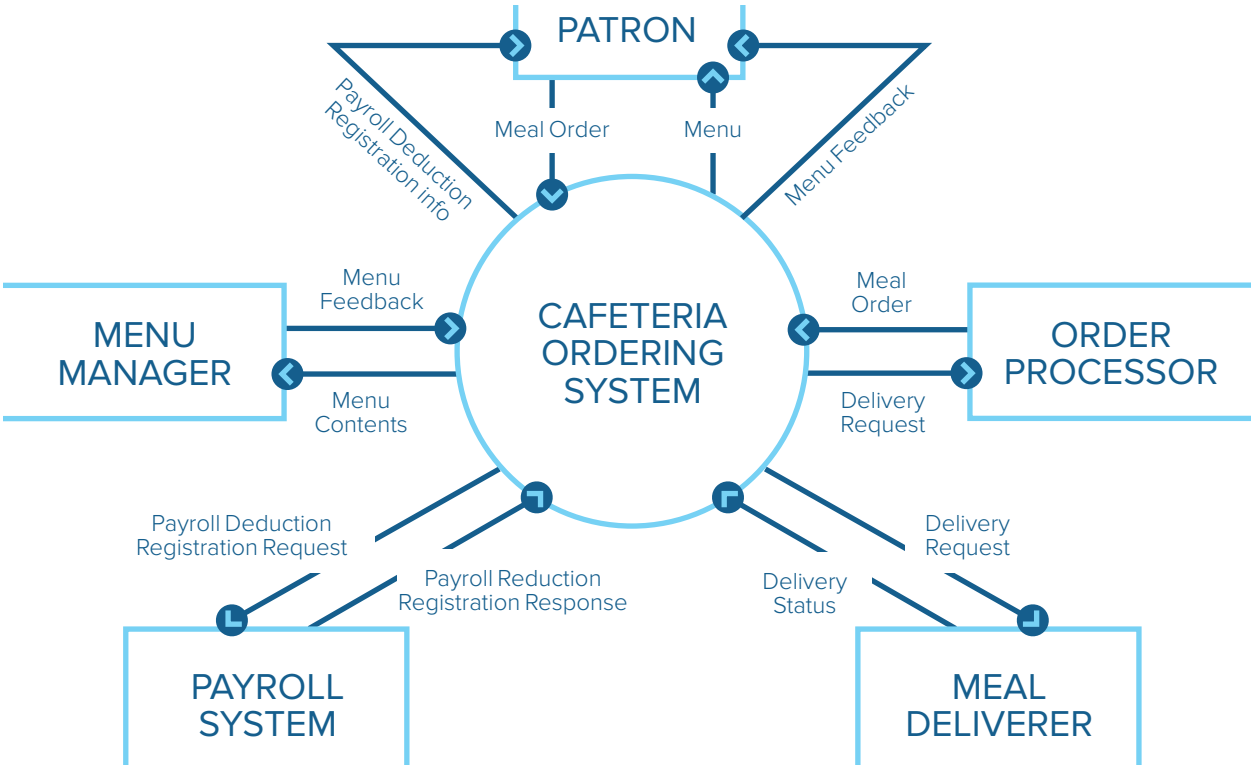
- There will be no virtual or fantasy games via the Web.
- There will be no ticketing facilities on the site.
- There will be no betting facilities available.
- The demographic details for newsletters will not be collected.
- Message boards are out of scope for phase 1.

Some stakeholders involved with this project might have expected these capabilities to be included. Itemizing them as exclusions makes it clear that they won't be. This is a form of expectation management, an important contributor to project success.

CONTEXT DIAGRAM

The venerable context diagram dates from the structured analysis revolution of the 1970s. Despite its antiquity, the context diagram remains a useful way to depict the environment in which a software system exists. On the next page, figure 1 illustrates a partial context diagram for a hypothetical corporate cafeteria ordering system. The context diagram shows the name of the system or product of interest in a circle. The circumference of the circle represents the system boundary. Rectangles outside the circle represent external entities, also called terminators. External entities could be user classes, actors, organizations, other software systems to which this one connects, or hardware devices that interface to the system.

Figure 1. A sample context diagram.



The interfaces between the system and these external entities are shown with labeled arrows, called flows. If the "system" is strictly an electronic system involving software and perhaps hardware components, flows will represent data or control signals. However, if the "system" includes both a software application and manual operations, flows could also represent the movement of physical objects. Two-headed flows indicate update operations involving the data object on the flow.

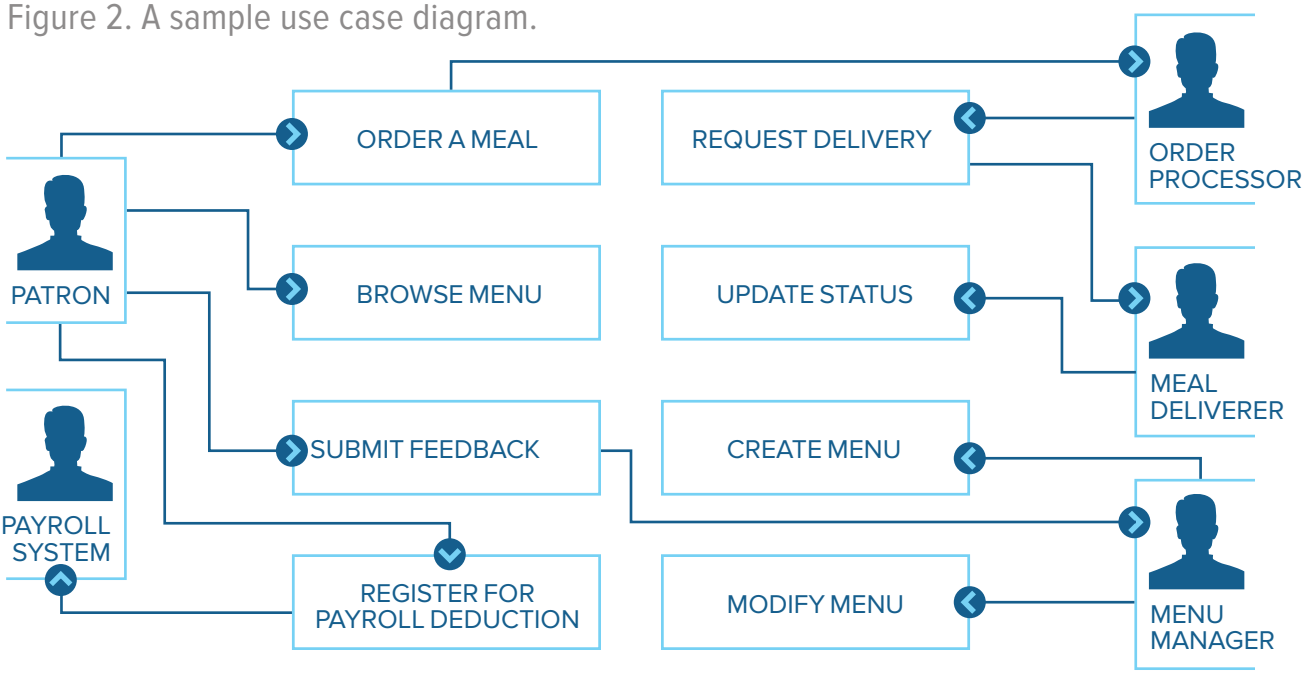
The context diagram depicts the project scope at a high level of abstraction. This diagram deliberately reveals nothing about the system internals: no information about functionality, architecture, or look-and-feel. Nor does it explicitly identify which features or functionality are in scope and which are not. The functional behavior of the system is merely implied by the labeled flows that connect the system to the external entities. Even the flows are labeled at a high level of abstraction, just to keep the diagram's complexity manageable. The business analyst can decompose these data flows into individual data elements in the project's data dictionary or data model. Corresponding data inputs and outputs imply the types of operations the system will perform, but these aren't shown explicitly in the context diagram.

Despite the limited view that the high level of abstraction imposes, the context diagram is a helpful representation of scope. It serves as a tool to help the project stakeholders communicate about what lies outside the system boundary. A business analyst in a requirements class I once taught showed me a context diagram for her current project. She had shown this diagram to the project manager. The manager had pointed out that one of the external entities shown on the context diagram, another information system, was now going to be part of the new system. With respect to Figure 1, this would be like moving the Payroll System inside the project circle. That is, the scope of the project just got larger than the BA expected. She had expected that external system to be someone else's responsibility, but now it was her problem.

USE CASE DIAGRAM

Use cases are a powerful technique for exploring user requirements. The Unified Modeling Language (UML) includes a use case diagram notation. Figure 2 shows a partial use case diagram for our cafeteria ordering system. The rectangular box represents the system boundary, analogous to the circle in a context diagram. The stick figures outside the box represent actors, entities that reside outside the system's context but interact with the system in some way. The actors correspond approximately (exactly, in this example) to the external entities shown in rectangles on the context diagram.

Figure 2. A sample use case diagram.



Unlike the context diagram, the use case diagram does provide some visibility into the system. Each oval inside the system boundary box represents a use case. The use case diagram shows the interactions of the system with its users and some connections between internal system operations, albeit at a high level of abstraction.

The arrows on the use case diagram indicate which actors participate in each use case. In Figure 2, the arrow from the Patron to the oval labeled "Submit Feedback" means that the patron actor can initiate the Submit Feedback use case. The arrow from Submit Feedback to the Menu Manager actor indicates that the Menu Manager participates somehow in the execution of Submit Feedback. Arrows on the use case diagram do not indicate data flows as they do on the context diagram. Some analysts simply draw lines instead of arrows on the use case diagram to avoid any confusion with data flow. In addition to showing these connections to external actors, a use case diagram could depict logical relationships and dependencies between use cases.

The use case diagram provides a richer scope representation than the context diagram because it provides a high-level look at the system's capabilities, not just at its external interfaces. There is a practical limitation, though. Any sizeable software system will have dozens of use cases, with many connections among them and between use cases and actors. Attempting to show all those objects inside a single system boundary box quickly becomes unwieldy. Therefore, the analyst needs to model groups of related use cases as packages or to create multiple use case diagrams.

Many analysts have found the context diagram and use case diagram to be helpful ways to represent and communicate a shared understanding of a project's scope.

FEATURE LEVELS

Customers, marketers, and developers often talk about product features, but the software industry doesn't have a standard definition of this term. I define a feature as: "A set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business objective." Features are product capabilities that a user can recognize, as opposed to capabilities that the product needs to have "under the hood" but aren't visible to end users. Marketing materials often state the features that the new (or improved) product will offer to the customer. Therefore, feature lists help customers make purchase decisions.

You can think of each product feature as having a series of levels that represent increasing degrees of capability or feature enrichment. Each iteration or product release implements a certain set of new features and perhaps enhances certain features that were partially implemented in earlier releases. One way to describe the scope of a particular product release, then, is to identify the specific levels for each feature that the team will implement in that release. A sequence of releases represents increasing levels of capability—and hence user value—delivered over a period of time. During requirements analysis, the analyst determines just which functional requirements must be implemented in a particular release to deliver the planned feature levels, beginning with the top-priority or foundational levels of the top-priority features.

To illustrate this approach to scope definition, consider the following set of features from our hypothetical cafeteria ordering system:

FE-1: Create and modify cafeteria menus
FE-2: Order meals from the cafeteria menu to be picked up or delivered
FE-3: Order meals from local restaurants to be delivered
FE-4: Register for meal payment options
FE-5: Request meal delivery
FE-6: Establish, modify, and cancel meal service subscriptions
FE-7: Produce recipes and ingredient lists for custom meals from the cafeteria

Table 1 illustrates a feature roadmap, which depicts the various levels for each of these features that are planned for implementation in forthcoming releases. FE-2 and FE-5 in Table 1 represent features that each have three enrichment levels, but there's nothing special about three levels. Some product features may have four or five levels of increasing functionality enrichment. Others might just have a single level that's implemented all at once, such as FE-1, Create and modify cafeteria menus, which the team will fully implement in the first release.

The full functionality for each of these features is delivered incrementally across the three planned releases. The analyst can also indicate dependencies between features or feature levels. As an illustration, the level of FE-2 scheduled for release 1 will let users pay for meals by payroll deduction. Therefore, the capability of FE-4 that lets users register for payroll deduction payments cannot be deferred to a later release, because the first level of FE-2 depends on the first feature level of FE-4. Understanding these dependencies is essential for effective release planning. Notice that this sample feature roadmap indicates that the team won't work on feature FE-3 at all until the third planned release. Also, feature FE-6 is of lower priority than the others. We'd like to get it out in the first release if we can, but it's okay to defer it to release 2 if necessary.

Figure 1. A sample Feature Roadmap.

FEATURE	RELEASE 1	RELEASE 2	RELEASE 3
FE-1	**Fully Implemented**		
FE-2	**Standard individual meals from lunch only; delivery orders may be paid for only by payroll deduction (depends on FE-4)**	**Accept orders for breakfasts & dinners, in addition to lunches; accept credit & debit card payments**	**Accept group meal orders for meetings & events**
FE-3	**Not implemented**	**Not implemented**	**Fully implemented**
FE-4	**Register for payroll deduction payments only**	**Register for credit card and debit card payments**	
FE-5	**Meals will be delivered only to company campus**	**Add delivery from cafeteria to selected off-site locations**	**Add delivery from restaurants to all current delivery locations**
FE-6	**Implemented if time permis**	**Fully implemented**	
FE-7	**Not implemented**	**Not implemented**	**Fully implemented**

The feature level approach is the most descriptive of the four techniques I'm presenting for defining the project scope. The farther into the future you look, the less certain the scope plans become and the more you can expect to adjust the scope as project and business realities change. Nonetheless, defining the product vision and project scope lays a solid foundation for the rest of the project work. This helps keep the team on track toward maximizing stakeholder satisfaction.

SYSTEM EVENT

The final scoping technique I'm going to describe focuses on external events the system must detect. Each event causes the system to produce a particular response. There are three types of system events to consider:

1. Business events, which are user actions that cause the system to respond in some way. The trigger that initiates the execution of a use case typically is a business event.

2. Temporal or time-based events, such as scheduled program executions. Examples are automatically producing a database extract at the same time every night or having an antivirus program check for virus signature updates once an hour.

3. Signal events, which could be input signals received from sensors or switches in a system that contains both software and hardware components.

Event analysis works well for real-time systems. Consider a complex highway intersection. It might include road sensors and cameras to detect vehicles (though sometimes they don't seem to spot me when I'm on my motorcycle), traffic lights, buttons pedestrians can press to cross the street, pedestrian walk signals, and so forth. This system has to deal with a variety of events, including these:

- A sensor detects a car approaching in one of the through lanes.
- A camera detects a car approaching in a left-turn lane.
- A pedestrian presses a button to request to cross a street.
- One of several timers counts down to zero.

At the scope level, you can just list the external events to which the system must respond without worrying about additional details. For an iterative development approach, you can allocate the handling of specific events to different iterations or releases. This initial events list also leads nicely into more detailed functional requirements specifications, expressed in the form of an event-response table. Exactly what the system does in response to an external event depends on the state of the system at the time it detects the event. On the next page, table 2 presents a fragment of what an event-response table might look like for such a system. This sort of layout is an effective alternative to the traditional list of natural-language functional requirements, which can be tedious to read and review.

Table 2. Partial Event-Response Table for a Highway Intersection. (See next page).

EVENT	SYSTEM STATE	RELEASE 3
Road sensor detects vehicle entering left-turn lane.	Left-turn signal is red. Cross-traffic signal is green.	Start green-to-amber countdown timer for cross-traffic signal.
Road sensor detects vehicle entering left-turn lane.	Left-turn signal is green.	Do nothing.
Green-to-amber countdown timer reaches zero.	Cross-traffic signal is green.	1. Turn cross-traffic signal amber. 2. Start amber-to-red countdown timer.
Amber-to-red countdown timer reaches zero.	Cross-traffic signal is amber.	1. Turn cross-traffic signal red. 2. Wait 1 second. 3. Turn left-turn signal green. 4. Start left-turn-signal countdown timer.

MANAGING SCOPE CREEP

Requirements will change and grow over the course of any software project. This is a natural aspect of software development. In fact, if a project doesn't experience some requirements evolution, the team likely is ignoring reality and risks releasing an irrelevant product. The prudent project manager anticipates and plans to accommodate some requirements growth.

Scope creep (also known as feature creep, requirements creep, featuritis, and creeping featurism), however, refers to the uncontrolled growth of functionality that the team attempts to stuff into an already-full project box. It doesn't all fit. The continuing churn and expansion of the requirements, coupled with inadequate prioritization, makes it difficult to deliver the most important functionality on schedule. This demand for ever-increasing functionality leads to delays, quality problems, and misdirected energy.

The first step in controlling scope creep is to document a clearly stated and agreed upon scope for the project. Without a scope definition, how can you even tell if you're experiencing scope creep? Project teams following an agile development life cycle should write a brief scope statement for each iteration. This helps make sure that everyone understands the goal of the iteration and what functionality the team will—and won't—implement during that iteration.

An ill-defined scope boundary can have serious consequences. In one situation I know of, a customer had hired a package-solution vendor to migrate three sets of existing data into the new package. Partway through the project, the customer concluded that six additional data conversions were required. The customer felt that this additional work fell within the agreed-upon scope, but the vendor maintained that it was out of scope and demanded additional payment. This scope ambiguity, and the resulting conflict, was one factor that led to the project being canceled and a lawsuit being filed. Not the desired outcome.

The second step for managing scope creep is for the business analyst or project manager to ask "Is this in scope?" whenever someone proposes some additional product capability. This could be a new or extended use case, a set of functional requirements, another user story, or a new feature or report. Note that the project scope could also encompass activities and deliverables besides the software products themselves. Perhaps your customers suddenly decide they want an online tutorial to help users learn the new system. This doesn't change the software deliverables, but it certainly expands the scope of the overall project.

There are three possible answers to the question, "Is this in scope?" If the new capability is clearly in scope for the next release, the team needs to address it. If it's clearly out of scope, the team does not need to address it, at least not now. They might schedule the new capability for a later release.

Sometimes, though, the requested functionality lies outside the scope as it's currently defined, but it's such a good idea that the management sponsor should expand the project scope to accommodate it. Less frequently, a proposed change could even modify the strategic product vision in some fundamental way. The question to consider is whether the proposal to expand the scope of the current release will significantly enhance the success of that release.

Electing to increase project scope is a business decision. When weighing whether to increase project scope, the key stakeholders need to consider cost, risk, schedule, and market or business implications. This demands negotiation between the project manager, management sponsor, and other key stakeholders to determine how best to handle the scope alteration. Referring to my three-level model of business, user, and functional requirements, the stakeholders must decide whether proposed changes in user or functional requirements will become the project manager's responsibility through a scope expansion (Figure 1).

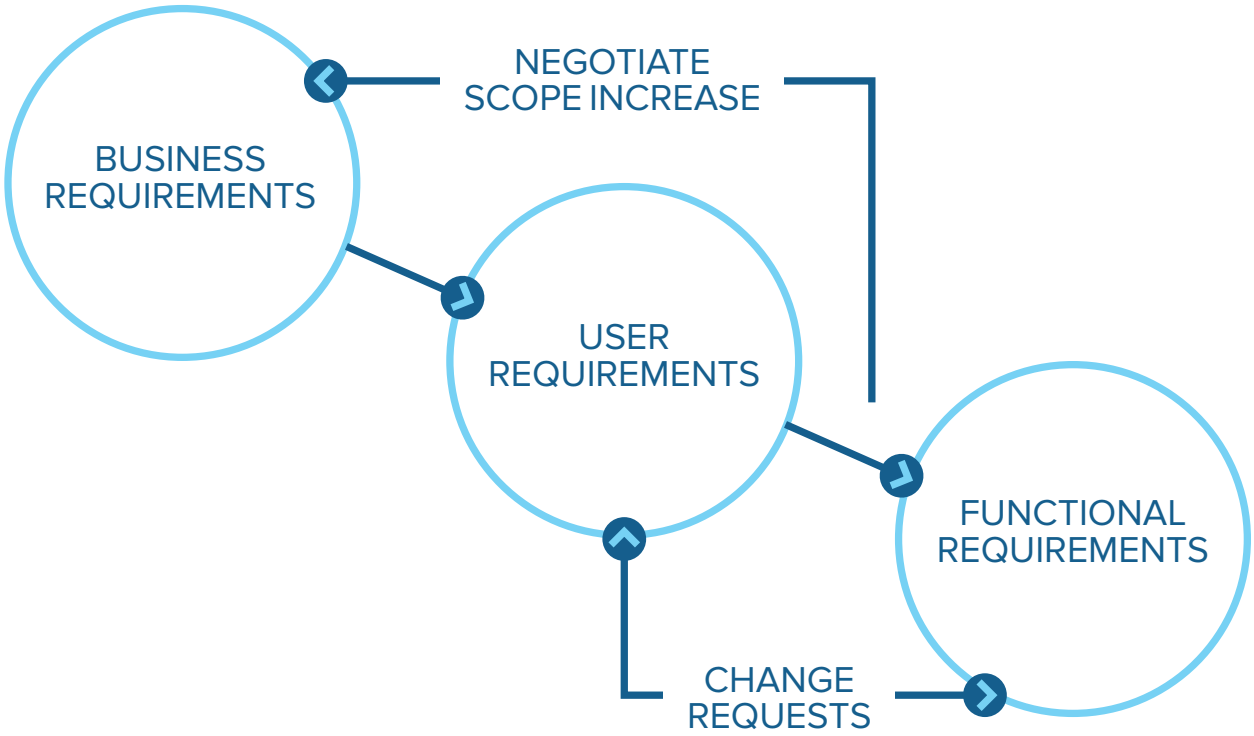


Figure 3. Changes in user & functional requirements lead to negotiations about increasing project scope at the business requirements level.

Following are some possible strategies for accommodating a scope increase:

- Defer some lower-priority or less time-critical functionality that was planned for the current release or iteration to make room for the proposed scope additions.
- Obtain additional development staff to handle the additional work.
- Obtain additional funding, perhaps to pay overtime (OK, this is just a little joke), outsource some work, or leverage productivity tools.
- Extend the schedule for the current release to accommodate the extra functionality. This is not a joke.
- Compromise on quality by doing a hasty job that you'll need to repair later on, which is not your best option.

Increasing the project's scope—or, even more profoundly, the product vision—always has a price. The people who are paying for the project must make a considered decision as to which scope management strategy is most appropriate in each situation. The objective is always to deliver the maximum customer value as soon as possible, while achieving the defined business objectives and success criteria, within the existing constraints.

There's no point in pretending the project team can implement an ever-increasing quantity of functionality without paying a price. In addition, it's always prudent to anticipate a certain amount of scope growth over the course of the project. A savvy project manager will incorporate contingency buffers into project plans so the team can accommodate some scope growth without demolishing its schedule commitments. This isn't simply padding or a fudge factor. Thoughtful contingency buffers are simply a sensible way to deal with the reality that things change and projects grow.
Effective scope management requires several conditions be met:

- The requirements must be prioritized so that the decision makers can agree upon the capabilities to include in each release or iteration, and so they can evaluate change requests against the priorities of unimplemented requirements in the backlog.
- The size of the requirements must be evaluated so that the team has an approximate idea of how much effort it will take to implement them.
- The team must know its average productivity (or velocity, in agile terms) so it can judge how many requirements or user stories it can implement and verify per unit time.
- The team must assess the impact of change requests they have a good understanding of what it will cost to implement each one.
- The decision makers need to be identified and their decision-making process established so they can efficiently decide to modify the scope when appropriate.

Don't be victimized by the specter of creeping scope or attempt in vain to suppress change. Instead, establish a clear scope definition early in the project and use a practical change control process to cope with the inevitable—and often beneficial—requirements evolution.

ABOUT KARL WIEGERS

Karl has provided training and consulting services worldwide on many aspects of software development, management and process improvement. He has authored 5 technical books, including Software Requirements, and written more than 175 articles. Prior to starting Process Impact in 1997, he spent 18 years at Eastman Kodak Company. His responsibilities there included experience as a photographic research scientist, software applications developer, software manager, and software process and quality improvement leader. Karl has led process improvement activities in small application development groups, Kodak's Internet development group, and a division of 500 software engineers developing embedded and host-based digital imaging software products. http://www.processimpact.com.

ABOUT JAMA SOFTWARE

From concept to launch, the Jama product delivery platform helps companies bring complex products to market. By involving every person invested in the organization's success, the Jama platform provides a structured collaboration environment, empowering everyone with instant and comprehensive insight into what they are building and why. Visionary organizations worldwide, including SpaceX, The Department of Defense, VW, Time Warner, GE, United Healthcare and Amazon.com use Jama to accelerate their R&D returns, out-innovate their competition and deliver business value. Jama is one of the fastest-growing enterprise software companies in the United States, having exceeded 100% growth in each of the past four years, during which time both Inc. and Forbes have repeatedly recognized the company as a model of responsible growth and innovation. For more information please visit http://www.jamasoftware.com.