# Introduction

Requirements engineering continues to be a hot topic in the software industry. More and more software development organizations realize that they cannot succeed unless they get the requirements right. Too often, the people responsible for leading the requirements process are ill-equipped for this challenging role. They do the best they can, but without adequate training, coaching, resources, and experience it's an uphill climb.

As a consultant, trainer, and author, I receive many questions from practitioners about how to handle difficult requirements issues. Certain questions come up over and over again. Many of these issues haven't been well addressed in the many published books on requirements development and management. In this handbook I've collected information on some of these recurrent topics. The handbook addresses such common questions as:

1. How do I keep too much design from being embedded in the requirements? (I heard this question again just yesterday, literally.)
2. When should I baseline my requirements? (Ditto.)
3. How can I convince my managers that we need to get better at handling our project requirements?
4. What are some good questions to ask in requirements interviews?
5. How can I use requirements to estimate how long it will take to finish the project?
6. How can I write better requirements?

I've addressed some other topics in this handbook simply because little is written about them. For instance, everyone talks about project scope, but I've seen little written about how to actually define scope. Everyone complains about scope creep, but how do you know you're experiencing scope creep unless the stakeholders have agreed upon and documented the project's scope?

Still other topics are included because I don't see practitioners using some of the established techniques that can help them do a better job. As an example, nearly all requirements specifications I see consist entirely of written text—there's not a picture to be found. However, the skilled analyst should have a rich tool kit of techniques for representing requirements information. Text is fine in many cases, but other requirements "views" are more valuable in others. The chapter titled "The Six Blind Men and the Requirements" addresses this topic. I've included numerous true stories from real, personal experiences. These are highlighted with a newspaper icon in the margin.

I hope you'll find this handbook to be a valuable supplement to your other sources of knowledge on software requirements engineering. You can read the chapters in any sequence that interests you. But don't just read the chapters and say, "That's interesting." For maximum benefit, set yourself a personal goal of finding at least three new practices that you want to try the next few times you put on your requirement analyst hat.

Copyright © 2013 Karl E. Wiegers. All rights reserved.

# Defining Project Scope

Every software team talks about scope and most project team members complain about unending scope creep. Unfortunately, the software industry lacks a uniform definition of these terms and the requirements engineering literature lacks clear guidance on how to represent scope. In this chapter I present some definitions, describe three techniques for defining the scope boundary—the context diagram, the use case diagram, and feature levels—and offer some tips on managing scope creep.

## Vision and Scope

The vision and scope document is a key software project deliverable (Wiegers 2003a). Other terms for this high-level guiding document are a project charter, marketing requirements document, and business case. Vision and scope are two related concepts. I think in terms of the *product vision* and the *project scope*. The product vision is:

> *A long-term strategic concept of the ultimate purpose and form of a new system.*

Chapter 5 of *Software Requirements, 2nd Edition* describes how to write a concise and focused vision statement using a keyword template (Wiegers 2003a). We can define the project scope as:

> *The portion of the ultimate product vision that the current project will address. The scope draws the boundary between what's in and what's out for the project.*

The latter part of the project scope definition is the most important. The scope defines what the product is and what it is not, what it will and won't do, what it will and won't contain. A well-defined scope boundary sets expectations among the project stakeholders. It defines the external interfaces between the rest of the world and the system, which could be just a software application or could be a combination of automation elements and manual processes. The scope definition helps the project manager assess the resources needed to implement the project and it helps him make realistic commitments. In essence, the scope statement defines the boundary of what the project manager is responsible for.

Your scope definition should also include a list of specific limitations or exclusions—what's out. Obviously, you can't list *everything* that's out of scope, as that would be the entire known universe except for the tiny sliver that is in scope for this project. Instead, the limitations should identify capabilities that a reader might expect to be included in the project but which are not. One project that was building a Web site for a national rugby team included the following scope exclusions for the initial release:

- There will be no virtual or fantasy rugby 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 in this Web site project might have expected these capabilities to be included with the first release. Itemizing them as exclusions makes it clear that they will not be. This is a form of expectation management.

Copyright © 2013 Karl E. Wiegers. All rights reserved.

## Context Diagram

The *context diagram* is a venerable analysis model dating from the structured analysis revolution of the 1970s (DeMarco 1979; Wiegers 2003a). Despite its age, the context diagram remains a useful way to depict the environment in which a software system exists. Figure 5 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 circle represents the system boundary. The rectangles outside the circle represent *external entities*, also called *terminators*. External entities could be different user classes, actors, organizations, other systems to which this one connects, or hardware devices that interface to the system. The interfaces between the system and these external entities are shown with the labeled arrows, called *flows*. Flows represent the movement of data, control signals, or physical objects between the system and the external entities.[5]

The context diagram shows the project scope at a high level of abstraction. This diagram deliberately reveals nothing about the system internals and it doesn't identify exactly what features or

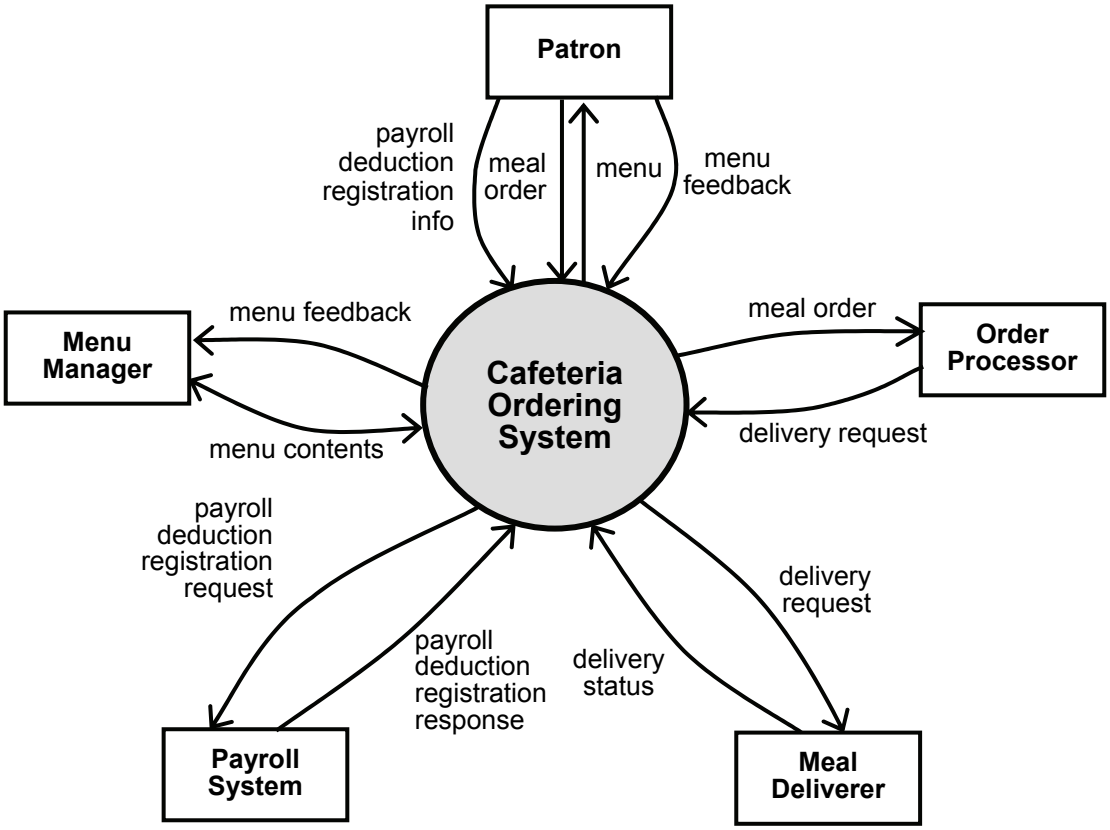


**Figure 5: A sample context diagram.**

[5] If a "system" is considered to include both a software application and manual operations, then flows could represent the movement of physical objects. However, if the "system" is strictly an automated system, flows should represent data or control signals.

Copyright © 2013 Karl E. Wiegers. All rights reserved.

functionality are in scope. 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. Data flows can be decomposed into individual data elements in the project's data dictionary or data model. Input/output pairs of data flows suggest the types of transactions or use cases that the system will perform, but these are not shown explicitly in the context diagram.

Despite the limitations that the high level of abstraction imposes, the context diagram is a helpful representation of scope. An analyst in a requirements seminar I once taught showed me a context diagram for her current project. She had just shown this to the project manager. The manager had pointed out that a decision had been made to make one of the external entities—another information system—a part of the new system being developed. If the diagram was that shown in Figure 5, this would be comparable to moving the Payroll System inside the circle. That is, the scope of the project just got larger. This analyst had expected that external system to be someone else's responsibility but now it was her problem. The context diagram provides a tool to help the project stakeholders communicate a common understanding of what lies outside the system boundary.

Note that a context diagram could potentially represent either the ultimate vision for the final product or the scope for a specific project that will develop just one of the releases planned for that product. Both representations are appropriate, but be sure to label your context diagram so readers know exactly what they're viewing.

## Use-Case Diagram

Use cases have become widely recognized as a powerful technique for exploring user requirements (Kulak and Guiney 2004; Wiegers 2003a). The Unified Modeling Language includes a

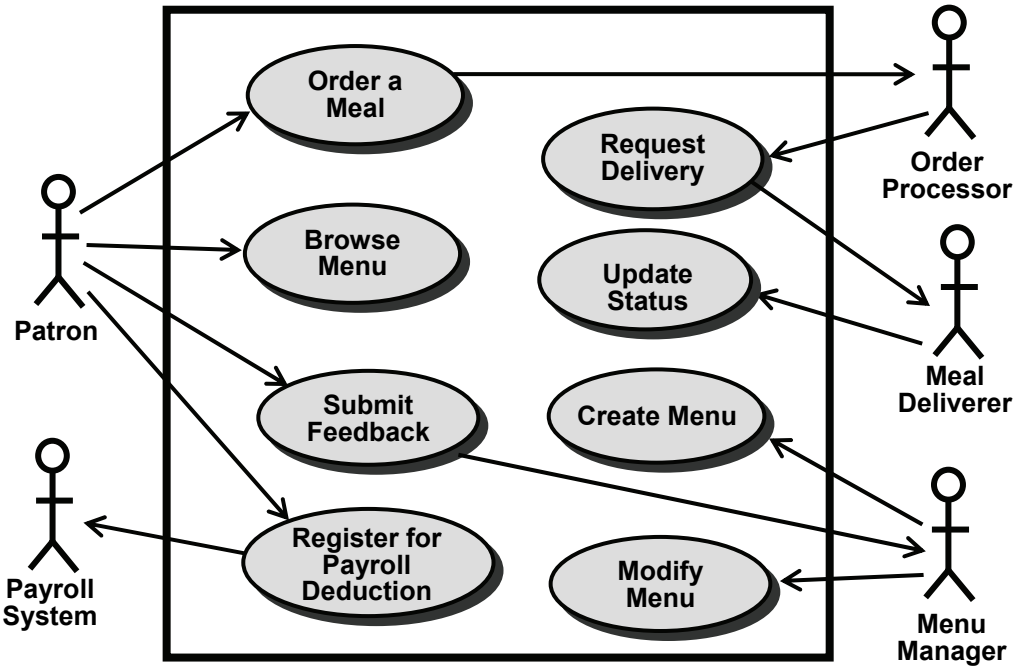


**Figure 6: A sample use-case diagram.**

Copyright © 2013 Karl E. Wiegers. All rights reserved.

*use-case diagram* notation (Schneider and Winters 2001). See Figure 6 for 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 to the external entities shown in rectangles on the context diagram. Unlike the context diagram, the use case diagram does provide limited visibility into the system internals. Each oval inside the system boundary represents an individual use case—literally, a case of usage—in which actors interact with the system to achieve a specific goal.

The arrows on the use-case diagram indicate which actors participate in each use case. Arrows do not indicate flows, as they do on the context diagram. An arrow from an actor to a use case indicates that that actor (called a *primary actor*) can initiate the use case. An arrow pointing from a use case to an actor means that that *secondary* actor participates in the successful completion of the use case. 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 internals, not just its external interfaces. There is a practical limitation, though. Any sizeable software system will have dozens of use cases with many connections between actors and each other. It quickly becomes unwieldy to show all those objects inside a single system boundary box. Therefore, the analyst needs to model groups of related use cases or create multiple use-case diagrams at various levels of detail (Armour and Miller 2001). This gets away from the simplicity of having a single diagram that shows the objects on both sides of the system boundary.

## 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 consider a *feature* to be:

> *A set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business objective.*

We can think of each product feature as having a series of levels that represent increasing degrees of capability or feature enrichment.[6] Each release of the product implements a certain set of new features and perhaps enhances features that were partially implemented in earlier releases, beginning with the top-priority levels of the top-priority features. One way to describe the scope of a particular product release, then, is to identify the specific level of 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.

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

[6] Nejmeh and Thomas (2002) use the term *feature vector* to refer to a single feature and the various functional levels that a feature implementation can achieve.

Copyright © 2013 Karl E. Wiegers. All rights reserved.

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

Figure 7 illustrates a feature roadmap, the various levels for these features that are planned for implementation in forthcoming releases. FE-2 and FE-5 in Figure 7 represent features having three enrichment levels each. The full functionality for each of these features is delivered incrementally across the three planned releases. The feature level approach is the most descriptive of these three techniques for defining the project scope. The analyst can also indicate dependencies between features or feature levels. For example, the level of FE-2 scheduled for Release 1 is expected to 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.

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.

| *Feature* | *Release 1* | *Release 2* | *Release 3* |
|---|---|---|---|
| FE-1 | Fully implemented | | |
| FE-2 | Standard individual meals from lunch menu only; delivery orders may be paid for only by payroll deduction (depends on FE-4) | Accept orders for breakfasts and dinners, in addition to lunches; accept credit and debit card payments | Accept group meal orders for meetings and 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 sites | Add delivery from cafeteria to selected off-site locations | Add delivery from restaurants to all current delivery locations |
| FE-6 | Implemented if time permits | Fully implemented | |
| FE-7 | Not implemented | Not implemented | Fully implemented |

**Figure 7: Sample feature roadmap.**

## Managing Scope Creep

Requirements will change and grow over the course of any software project. This is a natural aspect of software development and the project manager needs to anticipate and plan for it. Scope creep (also known as feature creep or requirements creep), however, refers to the uncontrolled growth of features that the team attempts to stuff into an already-full project box. It doesn't all fit.

Copyright © 2013 Karl E. Wiegers. All rights reserved.

The continuing churn and expansion of the requirements make it difficult to deliver the top-priority functionality on schedule. This demand for ever-increasing functionality leads to delays, quality problems, and misdirected energy. Scope creep is one of the most pervasive problems in software development.

The first step in controlling scope creep is to document a clearly stated—and agreed to—scope for the project. Without such a scope definition, how can you even tell you're experiencing scope creep? The techniques described earlier in this chapter provide several ways to define a project's scope. Projects following an agile development life cycle should write a brief scope statement for every iteration cycle to make sure everyone understands what will and will not be implemented during that iteration. Alternatively, give each iteration a short but telling name that conveys the goal of the iteration.

The second step for managing scope creep is to ask "Is this in scope?" whenever someone proposes some additional product capability, such as a use case, functional requirement, product feature, or output. The project scope could also encompass activities and deliverables besides the delivered software products. Perhaps your customers request an on-line tutorial to help their users learn the new system. This doesn't change the software product itself, but it certainly expands the scope of the overall project.

In one situation, a customer had contracted for a package-solution vendor to migrate three sets of historical 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 lay within the agreed-upon scope, but the vendor maintained that it was out of scope and demanded additional payment. This was one factor that led to a cancelled project and a lawsuit (Wiegers 2003b).

There are three possible answers to the question "Is this in scope?" If the new capability is clearly in scope, 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, though.

Sometimes, though, the requested functionality lies outside the scope as it's currently defined but it's such a good idea that the project scope should be expanded to accommodate it. Electing to increase project scope is a business decision that needs to consider cost, risk, schedule, and market implications. This requires negotiation between the project manager, the management sponsor, and key customers to determine how best to handle the scope change. That is, the owner of the business requirements—the management sponsor—has to decide whether proposed changes in user or functional requirements will become the project manager's responsibility through a scope expansion (Figure 8). Here are some possible strategies:

1. Defer or eliminate some other, lower-priority functionality that was planned for the current release.
2. Obtain additional development staff to handle the additional work.
3. Obtain additional funding, perhaps to pay overtime (okay, this is just a little joke), outsource some work, or purchase productivity tools.
4. Extend the schedule for the current release to accommodate the extra functionality (this is *not* a joke).
5. Compromise on quality by doing a hasty job that you'll need to repair later on (not your best option).

Increasing scope always has a price. The people who are paying for the project need to make a considered decision as to which scope-management strategy is most appropriate in each situation.

Copyright © 2013 Karl E. Wiegers. All rights reserved.

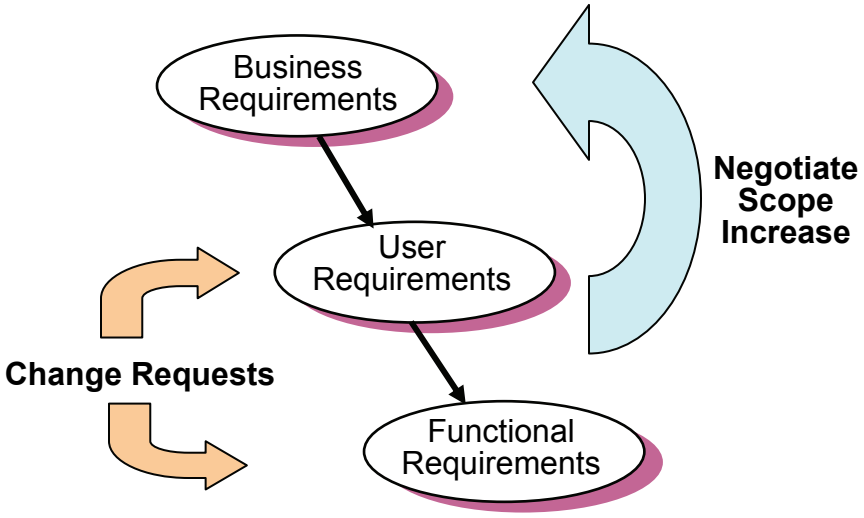


**Figure 8. Changes in user and functional requirements lead to negotiations about increasing project scope at the business requirements level.**

The objective is always to deliver the maximum customer value, aligned with achieving the defined business objectives and project 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. The savvy project manager will incorporate contingency buffers into project plans so the team can accommodate some scope growth without demolishing its schedule commitments (Wiegers 2002b).

Sensible project scope management requires several conditions:

- The requirements must be prioritized, so the decision makers can select the capabilities to include in the next release and can evaluate change requests against the priorities of requirements in the current baseline.
- The size of the requirements must be determined, so the team has an approximate idea of how much effort it will take to implement them.
- The team must know its average productivity, so it can judge how many requirements (measured in some size units) it can implement and verify per unit time.
- Change requests need to undergo impact analysis, so the team has a good understanding of what it will cost to implement each one and what the implications are likely to be for the project.
- 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 try 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 requirements evolution.

Copyright © 2013 Karl E. Wiegers. All rights reserved.