A technical discussion on Use Case Best Practices
06/11/03

Rational. software

IBM

Use Cases: Best Practices

By Ellen Gottesdiener

Table of Contents

Introduction

As an analyst, you have the crucial task of defining the requirements for software that is to be built or acquired. Your task is crucial for a number of reasons. If software teams fail to define excellent requirements, projects suffer from a variety of problems, including quality shortfalls, failure to meet schedules, ever-expanding user requirements, and, in the end, customer dissatisfaction. The financial costs are enormous. Depending on which study you read, typical software projects spend roughly one-third of their overall budget correcting errors that originate in requirements.

Whether you are defining requirements for new software, software that will be purchased, or existing software to be enhanced or maintained, it's easy to see why a clear understanding of requirements is one of the most important determiners of project success. Moreover, the task of defining high-quality requirements is crucial to all the project stakeholders: clients, end users, developers, testers, and managers.

Years of experience in defining requirements have led to the development of a number of techniques and models to assist in the process. Among these, perhaps the most well-known model is the use case, the focus of this paper. If you have experience with use cases, you know how pivotal they are for supporting many project activities, and you may be wondering how to improve your use case modeling to save time and energy. If you are new to use cases, you want to know the bottom line best practices for getting started. This paper's goal is to provide practical advice to both novice and experienced use case modelers.

How Use Cases Help You Define Software Requirements

To understand how use case modeling helps you define excellent requirements, let's first take a quick look at requirements. *Requirements* are the defined operational capabilities of a system or process that must exist to satisfy a business need. Requirements are the underpinning of all software development.

The generic term *requirement* covers both functional requirements and nonfunctional requirements. *Functional* requirements evolve from *user requirements*—tasks that users need to achieve using the software. *Nonfunctional* requirements, on the other hand, cover quality attributes of the software such as performance, system needs such as security and archiving, and technical constraints such as language and database. In the end, both functional and nonfunctional requirements must be clear to the development and client communities, who may not have a easy means of communicating with one another.

Requirements don't come out of thin air. They are the product of a systematic discovery and definition process in which analysts play a key role. Software requirements derive from a process of thinking through three perspectives of requirements: the business level, the user level, and the technical level.

The Business Level

At the highest (or business) level, you begin by understanding and clarifying the project's business goals and objectives, and you define a vision for how the product will achieve those outcomes. Your purpose is to ensure that you are going to build the right software. In addition to articulating a project vision, this involves defining all the project stakeholders, including *direct users,* or *actors,* of the system. You record these findings in a document such as the Rational Unified Process®'s Vision and/or Business Case template.

The User Level

Next, your focus turns to the direct users of the system to define the user requirements. This is where use cases come in. Use cases are the best modeling construct for defining user tasks. Put simply, a *use case* is describes an interaction between an external actor and the system, thereby documenting a major function that the system will perform.

At minimum, a use case has a name (more about use case names later) and a step-by-step description of a basic course of action. Some use cases also describe exception conditions and variant paths. Each use case should describe an action that is necessary for the user to achieve a project goal or objective. A useful tool for writing use cases is RUP®'s Use Case Specification template.

The Technical Level

The technical requirements include functional requirements based on user requirements and nonfunctional requirements. Traditionally, projects have used text-based functional hierarchies to describe the user requirements in declarative statements ("The system shall provide the capability to…"). Because use cases have become the de facto standard for describing interactions between actors and the system, many projects now employ use cases to augment these functional hierarchies. If you use detailed use cases to cover all the software's behavior, they can substitute for functional hierarchies.

Documenting Requirements

You document your project's software requirements in a Software Requirements Specification. For those projects that employ use cases to represent functional requirements, you can combine your Use Case Specification with RUP's Supplementary Specification template for documenting your nonfunctional requirements. Use cases, then, lie in-between the business and technical perspectives and provide the basis on which all development and testing is based.

Listening to the "Voice of the Customer"

Functional requirements are typically written from the point of view of the software, but use cases are written from the "voice of the customer." This expression, which comes from the quality movement, refers to discovering the stated and unstated requirements of product customers and users. Building a software product without understanding their needs is a sure path to failure. Use cases are arguably the best requirements technique we have for describing "the voice of the customer" in software products.

Getting Started with Use Cases

Following are seven key best practices to use as you embark on modeling use cases in your project:

- **Scope the domain.**
- **Scope your use cases.**
- **Validate use cases as they emerge.**
- **Define the requirements models you'll need.**
- **Determine the strategy you'll use to elicit requirements.**
- **Settle on a standard format for your use cases.**
- **Develop a project glossary.**

Let's look at each of these in detail.

Scope the Domain

When the scope of your project expands as the work proceeds, the project is experiencing *scope creep*, often cited as the highest risk for any software development project[1]. Requirements may change because of changing market and business conditions; that kind of change is largely unavoidable, and may even be welcomed in products where requirements are emergent. Your problem as an analyst is to manage avoidable scope creep, which happens when you haven't clarified and prioritized requirements and established agreement between customers and product developers.

You must tackle scope creep early in requirements development, and continue to do so throughout the project. Your project's use cases are an effective mechanism for scope management. Define which use cases are in and out of the scope of your project before detailing them. After initially defining use cases in scope, you will continue to discover use cases and use case alternative or exception paths. When that happens, decide if they belong in the project or the current or next iteration. Continually delineating your project's scope saves time you would otherwise waste in defining unnecessary requirements. It also accelerates use case modeling and goes a long way toward establishing productive communications with your customers.

How do you define the scope of your project? One way is to start top-down by creating several modeling constructs: your list of stakeholders (as defined in your vision statement), a context diagram, a list of events, and a definition of your domain. (For more on these models, see "Define the Requirements Models You'll Need.") From those documents, you generate a list of use cases in scope. Another approach is to generate a list of candidate use cases and associate them with actors.

At this point, your list of use cases will contain only the *names* of the use cases and not the details. A later section of the paper ("Determine Your Elicitation Strategy") offers information about techniques you can adopt to do this defining, creating, and generating with members of your project team. For now, let's focus on understanding what these names will look like.

Generating Use Case Names

The best way to generate use case names is either to start with the actors or to list the use cases and then name each use case's initiating actor. Well-named use cases often enable a business customer to easily infer who the actor is, and that is more than half the battle of defining clear, in-scope use cases.

When you're naming use cases, follow these best practices:

- **Name your use cases using this format: action verb + [qualified] object. Examples are "Place order," "Request product or service," "Monitor network usage, "Assign resources to project".**
- **Avoid vague verbs such as *do* or *process*. Instead, use precise verbs such as "query", "approve", "notify", "monitor", "generate" and so on.**
- **Avoid low-level, database-oriented verbs such as *create*, *read*, *update*, *delete* (known collectively by the acronym CRUD), *get*, or *insert*.**
- **The "object" part of the use case name can be a noun (such as *inventory*) or a qualified noun (such as *in-stock inventory*).**
- **Make sure that the project glossary defines each object in the use case name (more on the glossary later).**
- **Add each object to the domain model (as a class, entity, or attribute).**
- **Elicit actors and use cases concurrently, associating actors with use cases as you name each one.**

[1] See Capers Jones, *Patterns of Software Systems Failure and Success*. Thomson Computer Press, 1996.

You name use cases during the Inception phase. Your purpose is to make project scope decisions and perhaps to get a rough estimate of project size. Compiling this kind of survey list of use cases before you detail them is particularly helpful if you have a large project with multiple teams working concurrently on the system.

Scope Your Use Cases

Just as you must carefully define the scope of your project, you must carefully define the scope of each use case. To ensure that each use case stays in scope, make sure that it addresses a single actor goal and is not overly complex. At the same time, however, avoid the temptation to create small, piecemeal use cases that handle functions or partial processes in the business. A common mistake is to create CRUD (create-read-update-delete) use cases or to create separate use cases to describe alternative courses through a use case.

One way to define well-scoped use cases is to frame each use case with triggering events and necessary event responses. In this way, you'll know when the use case starts and ends. *Events* are what cause actors to initiate use cases. The use case finishes when the actors goals are satisfied, in other words, when the *event response* is achieved, the use case is finished. For example, consider a use case named "Place order." A customer wanting to place an order is the triggering event that motivates the customer actor to initiate the use case, and the event response would include "stores order information" and "send confirmation to customer".

Defining Events

Events for scoping use cases come in two flavors: business and temporal.

***Business* events are high-level occurrences that happen at unpredictable times. Although you can estimate, for example, how many widget requests or product searches might occur, you can't specifically say *when* they will occur or *how often*. A simple and direct way to name business events is to use a "subject + verb + object" format—for example, "Customer requests book." In this example, one event response might be that book information is provided to the customer, which is the desired outcome of the use case. Using business events as a starting point gives you obvious use case names. The subject part of the business event turns out to be an initiating actor ("customer"), and the verb part ("requests") gives you clues for naming one or more use cases.**

***Temporal* events, the other type of framing event, are clock-driven, predictable occurrences. For example, you may have scheduled requirements such as replenishing inventory levels, publishing schedules, backing up key databases, or posting bills. Temporal events should be named using the format "time to <verb + object>"—for example, "Time to publish schedules." The initiating "actor" for these temporal events is something like Clock or a pseudo actor name you choose, such as Inventory Controller or Schedule Manager. Event responses to temporal events can be *custodial*—for example, cleaning up data inside the system by refreshing information—or they can generate tangible artifacts to actors, as business events often do.**

Defining events can also help you eliminate use cases that don't belong in your project's scope. One way for everyone to "see" the system's scope is to draw a *context diagram* while simultaneously naming business and temporal events. A context diagram is a simple diagram that represents the system as a single "black box" surrounded by its major interfaces, thus showing the system in its environment. The system is depicted as a circle surrounded by its interfaces points which can be drawn as boxes or with actor icons. Actors "give" or "get" something to or from the system. This simple visual diagram goes a long way toward visually describing all the participating actors.

You can supplement that picture with an *event table* (a table with one column for events and another for the corresponding event responses). For example, an event might be "customer places order" and an event response might be "receipt transmitted to customer"; "pick list sent to distribution center".

The context diagram (possibly supplemented with an event table) acts as an initial high-level blueprint for the scope of your project. As you generate your use cases throughout the project and understand more paths through those use cases, you can refer to and revise this blueprint to avoid specifying use cases that aren't in scope. Creating these scope-level requirements models in collaboration with your customers allows you to quickly and effectively grasp the scope of your application.

Validate Use Cases as They Emerge

Whether you've named your use cases in scope by using a top-down approach or simply by listing them and associating them with actors, it is essential to *validate* the use cases: ensure that each one is necessary to meet the business opportunities in your product vision. Validation ensures that you are defining the right requirements.

To quickly and easily validate the use cases you've defined early in the Inception phase, answer the following questions:

- **How does this use case help us achieve our goals and vision?**
- **Does this use case address some aspect of the problem in our problem statement?**
- **Does this use case differentiate our product in some way?**
- **Do we have use cases to address all the stakeholder and user groups we identified in our vision statement?**
- **Which use cases will be implemented in our initial release?**

Any use case that does not align with the vision is potentially out of scope or lower in priority. In the end, this process can help you avoid requirements scope creep and optimize the time you devote to requirements development.

Define the Requirements Models You'll Need

A *requirements model* is a set of models which acts as a blueprint for a software product. Each individual model within the overall requirements model takes the form of a diagram, a list, or a table, often supplemented with text that depicts a user's needs from a particular point of view. Examples of user requirements models include event lists, use cases, context diagrams, data models, class models, business rules, actor maps, prototypes, and use case storyboards.

Whatever its form, the primary purpose of a requirements model is to *communicate*. Defining requirements is a discovery process for users and customers. User requirements evolve from the process of users trying out their requirements through models. Requirements models speak to users as well as to software people. As users create and modify the models, their understanding of their requirements changes and evolves—in other words, the requirements become clearer.

Models can depict multiple views but primarily emphasize one view: that of behavior, structure, dynamics, or control. For example, use cases are primarily *behavioral*, although you can infer necessary business rules and structure to support those behaviors. A class diagram is primarily *structural*, although you can infer behavior from it when it depicts operations. A statechart diagram is primarily *dynamic*, because it communicates allowable lifecycles for a domain—although you can infer behavior from its state transitions. And of course a statement of business rules, which are at the heart of all these models, communicates *control,* but you can infer necessary structure, behavior, and dynamics from some business rules statements.

Using multiple views gives you a rich context for eliciting user requirements and aligns with *separation of concerns,* a key principle and best practice in software engineering. Separation of concerns involves identifying, isolating, and encapsulating concepts from one another. This separation allows you to simplify complexity while facilitating reuse, change, and adaptability. The practice of defining multiple views to represent requirements exploits separation of concerns. Each view describes a specific aspect of your software and omits extraneous information. This means, for example, that your use cases don't include details found in other views such as data attributes and business rules. Instead, these related views—whether defined with a diagram or text—should be traced to your use cases, as we'll discuss later.

From the point of view of a person or system interacting with your software, a use case naturally describes an aspect of its behavior. But no single user requirements view and fully express all of the software's functional requirements: its behavior, structure, dynamics, and control mechanisms. These are interrelated views of the functional requirements, and they give you complementary mechanisms for analyzing your business domain and modeling it accurately and completely.

As an analyst, you need to decide early on which view you want to use to augment your use cases and define user requirements. It's best to use a variety of starting models, combining text and

diagrams if possible. For example, during Inception, you might start with a use case list and an actor map, a context diagram, a list of stakeholder classes, and a list of domains in scope. Alternatively, you might choose to create an analysis class model, use cases, and descriptions of some use case scenarios. Pick views that fit the business problem domain. If your business domain is rich in dynamics, with numerous events that change the state of business objects or data, then a coarse-grained statechart diagram might be useful. If your domain has many processes and tasks, it might be useful to employ activity diagrams for visualizing use cases. If multiple concurrent teams are working on your project, the teams should work through the use cases and related requirements at roughly the same level of precision, periodically regrouping to review each other's requirements to find shared requirements and avoid duplication of effort.

In sum, let your use cases do what use cases do well: describe actor-system interactions. Supplement them with other, well-chosen user requirements models to give you and your customers a richer understanding of the project requirements.

Determine Your Elicitation Strategy

You can use a variety of techniques for eliciting use cases and related requirements:

- Conducting one-on-one or group interviews
- Holding facilitated workshops
- Generating lists of scenarios
- Reusing existing requirements
- Prototyping the user interface
- Observing end users in their work environment
- Conducting focus groups
- Sending out questionnaires or market surveys
- Reviewing regulations, procedures, and guidelines
- Mining customer complaints and help desk logs
- Conducting competitive analyses
- Using a combination of these techniques

In most cases, a combination of elicitation techniques is best. Select techniques that fit how much user access you will have, and exploit what you can learn about their existing and potential needs.

For example, if you are developing commercial software, a good combination is to review market surveys, to conduct on-site visits to observe users interacting with your current software product line, and then to hold facilitated workshops with product development and marketing reps (who act as surrogate end users). If you are building enhancements to an existing in-house business system that has a large user base, it might work best to employ a combination of reviewing help desk logs, reusing existing requirements, and holding workshops with representative users from different user groups.

If you have a smaller user base, choose techniques that take advantage of direct customer collaboration, such as facilitated workshops and observation, in conjunction with techniques that provide more precise information, such as scenarios and prototypes. As these examples imply, you should consider your elicitation strategy very early in Inception as you are defining the project stakeholders and specifying all the groups of end users.

Be sure that the right people are working with your user community for your chosen elicitation approaches. As an analyst, you know that requirements work is difficult and takes certain skills and proclivities, including the ability to listen, question, and abstract, as well as a genuine interest in people's work life, a sense of curiosity, and a tolerance for ambiguity. When you and your colleagues are working with representative users, seek people who not only have deep domain knowledge but also share the ability to work with abstraction, play with models, and relentlessly define details when necessary.

Settle on a Standard Format for Your Use Cases

You can specify use cases by using various forms and formats and with varying degrees of precision.

Following are some sample use case forms, in order of increasing complexity:

- **Use case name only ("verb + [qualified] object")**
- **Use case name and a single-sentence goal statement**
- **Use case brief description (two to five sentences explaining what the use case does)**
- **Use case outline (bulleted or numbered list of use case steps, with alternative flows outlined separately or not listed at all)**
- **Use case conversational format, also called "essential" form (use case header information plus two columns—one for actors and one for system responses—written in a conversational style)**
- **Use case detailed description[2] (a sequential, conversational, or narrative text description that includes not only the normal path but also sections for all alternatives, exceptions, and extensions)**

During Inception, stay with the simplest forms, such as one of the first four forms listed. As an analyst, you have the tasks of clarifying the goal and expected outcome of each use case, ensuring that it is in scope, and helping customers understand their importance to the project. During Elaboration, you will add details to in-scope use cases to create an architectural prototype of the software as a tool to reduce project risk. To do that, start with existing high-level use cases and then use an iterative approach to add details, such as steps for the usual (or *happy*) case followed by alternatives and exceptions (*unhappy* cases).

Document each use case by using a *use case template* to standardize its format. One example is the use case specification template provided in RUP. You can modify or tailor the template to meet your project needs. For example, you might add information such as reference documents that an actor must access while performing a use case, or you might remove sections that you don't need, such as extension points or alternative flows.

Calibrate the level of desired detail for documenting your use cases according to the project's needs. Projects with a large number of team members producing mission-critical software with nondiscretionary funds will need more detailed documentation. Software governing such systems as airline cockpit controls, missile guidance, or human clinical trials must be precise and well documented. On the other hand, if you have a small, co-located team of developers who are familiar with the domain, your users are fairly accessible, and speed is of the essence, a simple format of use case names, each with a brief description, will do.

Stylistically, use cases should be understandable to any stakeholder. Remember that not all use cases are created equal; in any domain, some actions are inherently more complex than others. Thus, you should be ready to mix styles as needed. Some use cases can be aptly described in an outline, but other, more complex ones need explicit documentation of alternative paths and exception paths.

The writing style of your use cases should be direct and straightforward. It's best to write simple declarative sentences in the active voice using present tense (like the sentence you're reading now). Some users prefer bulleted lists, whereas others like a more narrative style. Try each style in one or two use cases early in the requirements elicitation process, and solicit user input on which is a better fit. After all, the primary customers of your use cases are the end users, so they should be involved in choosing the most appropriate documentation style.

Develop a Glossary to Define a Common Language

It has always been difficult to uncover and clarify user requirements. Indeed, many industry experts assert that defining user requirements is the most difficult task in software development. One of the

[2] See Kurt Bittner and Ian Spence, *Use Case Modeling*, Addison-Wesley, 2003.

biggest sources of difficulty is the gaps in communication between software people and business people—gaps that result in less than stellar customer-supplier relationships. Each side uses its own acronyms and jargon. Unless people on both sides have experienced each other's roles, they have trouble truly appreciating each other's legitimate concerns. This problem exists for both external software development organizations and software groups that support internal business organizations.

The different languages spoken by each side compound the problem of communication. For one thing, language is the primary means by which most knowledge workers experience and share their work. Furthermore, terms, especially business terms, and their meanings are perhaps the most fundamental construct in defining requirements. In the business community, people may use the same word in different ways or use different words to mean the same thing. When this happens, communication falters, and time and energy are wasted.

Thus, the first and most fundamental best practice is to agree on what the words you use really mean. The best way to do that is to start building a *glossary* right away and keep it a living, vital part of the requirements process. All your key business terms should be defined there, and you should reference it regularly as you define your use cases. Although the glossary will evolve as you progress through your project, most key terms will be identified while you are creating your first-cut list of use cases and use case descriptions.

Seven Tips for Writing Successful Use Cases

Tip 1: Develop Your Use Cases Iteratively

Use cases evolve and vary in format and complexity as you move through your project. Don't get caught up during Inception in "analysis paralysis" by trying to define your use cases in great detail. Early in your project, use cases should be lightweight and informal, providing you and your customers with enough information to understand their overall scope, complexity, and priority.

During Inception, it can take minutes or hours to define a single use case, depending on its complexity and the knowledge of the people doing the work. To keep your momentum going, decide ahead of time how you will reach closure on each use case, and bite off the most important use cases first. Reaching closure can be as simple as having a customer or representative user review and agree to the use case, or as involved as creating and walking through sample scenarios to ensure that the use case covers them.

During Elaboration, use cases are the central mechanism for reducing risk. You choose architecturally significant scenarios (paths through use cases that explore architecturally "interesting" behavior) as the basis for building an architectural prototype, thereby building one or more slices through the application. You might also choose to elaborate on those use case scenarios that are most unclear and therefore pose special risks. You will add more detail to each use case that is part of the architectural prototype, or enough detail to enable architects and developers to design, build, and test those use case scenarios. You continue through each iteration of Elaboration to slice through use cases in this manner.

During Construction, the remaining flows of use cases are detailed as necessary to fill-in the remaining behavior of the system. The detail in the descriptions of this behavior will vary – complex behavior will need more precise description than behavior that everyone collectively understands. With an eye always toward delivering software, you should exploit your use case documentation to create end-user documentation, training, and help manuals during the Transition phase.

Tip 2: Involve Users Directly

User involvement (truly hearing the "voice of the customer") is critical to successfully developing use cases. After all, your job is to describe what users really need and not the project team's interpretation of possible needs. Strategize how to involve end users or *ambassador* users—people responsible for bringing user community domain knowledge into the project and coordinating information with their respective user communities. Involving users can be difficult, but it is essential

to plan for user participation as you enter each iteration of your project in which use cases are a key principle for the development work.

Users are often busy or noncommittal, or on the surface they appear to be inaccessible to the project. How much user participation do you need? The best situation is to hold facilitated workshops to develop your use cases. In a *facilitated workshop*, a group of carefully selected stakeholders and content experts (users or ambassador users), led by a neutral facilitator, collaborate to produce a predetermined set of deliverables.[3] Like any well-run workshop that is planned, focused, and highly productive, a use case workshop can deliver your use cases and related requirements much faster and with higher quality than more traditional approaches such as interviews, surveys, and questionnaires. Every analyst should be skilled in designing and leading such workshops.

Prototypes developed in the presence of users, or at least reviewed with them, bring your use cases to life. They describe requirements as viewed by the actors. Even a low-fidelity prototype, such as sketches of screen shots, can be useful. You can also integrate prototypes into your use case workshops. You might conduct a workshop to elicit a set of use cases, business rules, and scenarios in the morning; then in the afternoon or next day, participants return to review sketches of screens that navigate through the use cases they've just defined. Try eliciting and testing use cases by using simple screen navigation maps or mockup dialogs posted in sequence on a wall. These low-fidelity prototypes also help you manage user and customer expectations about the system's look and feel without locking anyone into specific interface designs. At the same time they clarify the flow of each use case and uncover business rules that the use cases must enforce.

Another way to involve users is to conduct *reviews,*[4] especially use case reviews. Use case reviews are short meetings focused on a set of use cases and related requirements, with the goal of improving clarity and removing errors. For these sessions to be successful, reviewers must prepare by individually checking the work product beforehand.

Depending on the type of review, you may want formal sign-off on use cases. In other, less formal reviews, you will note errors and correct them without requesting user sign-off or another review meeting.

At a minimum, end users should verify requirements by doing use case walkthroughs or reviews. Scenarios—sequences of behaviors or sample uses of a use case—are the best way to conduct a walkthrough, especially if end users have developed the scenarios. Even if users can't be involved in ongoing use case specification, they can help you fix and evolve the use cases during a one- or two-hour walkthrough.

If you're developing commercial software, it can be difficult to gain access to your true end users. Product and marketing managers can act as surrogate users, but they can't know everything. A big concern for requirements in these projects is a possible disconnect between what real end users need and want and what surrogate users *think* end users will need and want.

In general, it's not a good idea to develop detailed use cases before doing some reality checks with real end users. Try to conduct reviews or walkthroughs (an informal review that steps through a work product for educational and communications purposes) with them, or show them early prototypes.

Product managers, who are often stand-ins for end users, can conduct focus groups to get feedback on draft user requirements, prototypes, or some scenarios covered by your draft use cases. Or, in lieu of these approaches, you can ask knowledgeable business people in the marketing organization to role-play representative end users, inventing names and personal backgrounds for each person to make the role playing more realistic.

Ask your user community how to make the best use of their time while not skimping on defining requirements. As you attempt to involve them more fully, solicit their feedback periodically about how the process is working for them. Tell them what you need for developing quality use cases and successful requirements, explaining the risks associated with insufficient user involvement. This will allow both of you to adjust your interactions and build sound and trusting relationships.

[3] See Ellen Gottesdiener, *Requirements by Collaboration: Workshops for Defining Needs*, Addison-Wesley, 2002.
[4]See Karl Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2002.

Tip 3: Depict Your Use Cases Visually

Consider supplementing text use case descriptions with diagrams. Here are some examples:

- **A use case diagram (ovals with the use case name and actor icons, à la the UML)**
- **An activity diagram showing the logical flow of a set of use cases)**
- **A use case steps diagram (activity diagram with each step depicted as an activity)**
- **Use case packages (named packages of use cases, possibly with package dependencies shown using UML dependency notation among the use case packages)**

Just as with text descriptions, plan which visual diagrams, if any, you will use. You can adapt your documentation as you iterate. For example, during Inception you might find that listing well-named use cases and then providing a brief description is enough; then as you add detail during your Elaboration iterations, you may supplement more complex use cases with a use case step diagram to help customers see the steps.

Similarly, diagrams can help you see the larger context for a set of logically related use cases. In real projects, groups of use cases are often used in conjunction with each other. To see this bigger view, employ an activity diagram of use cases, a package diagram, or both to arrange use cases into groups of logical dependencies. Using scenarios, you can test your use case flow. Each might be further partitioned, particularly for large systems, into packages. Diagrams like these help everyone keep the big picture in mind.

Tip 4: Use Your Use Cases to Harvest Nonfunctional Requirements

Along with defining use cases, a critical part of your job is to describe nonfunctional requirements. They drive architectural decisions, govern user satisfaction with your final product, and provide competitive advantage in commercial software products. Yet nonfunctional requirements are notoriously difficult to define. Use cases are a pivotal tool for harvesting them.

The key to uncovering nonfunctional requirements is to ask good questions as you elicit your use cases. As you begin to specify the functionality needed to achieve the goal of each use case, ask questions of your users to derive associated nonfunctional requirements, such as response time, throughput, and usability. Here are examples:

- **Will the people playing the role of the actor for this use case be in the same location, or in different ones? How many people will use the system concurrently? How many locations? Where are the locations?**
- **What type of knowledge or experience will the users require when interacting with the system? How familiar will they be with the tasks they need to perform?**
- **Will experienced users need to learn to use this functionality differently from the way new users learn it?**
- **What is the maximum acceptable time for getting responses from the system during the course of this use case? Are there certain critical steps that need to be performed in a more constrained time period?**
- **How many times will this use case be used during a day, an hour, or a week? Are there times when it will be used more often? How many instances of the use case are likely to be performed concurrently?**

The answers will help you begin to nail down the nonfunctional requirements for your use cases. You will also begin to see patterns whereby certain nonfunctional requirements are associated with groups of use cases. Often you need define them only once, and they don't belong in your use case text.

Separate nonfunctional requirements from use cases, and store them in your Supplementary Specification document or the "Special Requirements" section of your Software Requirements Specification document. Other nonfunctional requirements—such as backup, recovery, security, and

audits—relate to multiple use cases. All these will help architects get a comprehensive overview of the technical issues that will drive important design considerations.

Tip 5: Prioritize Your Use Cases and Use Case Scenarios

Not all use cases or use case scenarios are created equal. Some are more crucial to achieving business goals than others, providing different levels of business value to the customer. Having customers prioritize use cases or scenarios within them will help you decide when to define and analyze them. For example, it might be wise to delay lower-priority use cases until future releases or put off working on those with less architectural significance until Elaboration.

Of course, you must balance customer priorities with other factors, including architectural dependency (wherein one use case must be implemented before others) and risks that can arise from new or untried technology or business processes.

Beginning in Inception, after you and your stakeholders understand the purpose of the use cases in scope and the characteristics of their initiating actors, ask key customers to prioritize them. These priorities will drive the content and sequence of your iterations in Elaboration and will become the basis for determining your release strategy.

Assessing use cases and use case scenarios by themselves is usually not enough to prioritize them; you must also include their associated nonfunctional requirements as the basis for determining your release schedule. For example, for a Web-based order system, software that implements the "Place order" use case will not be acceptable if end users experience long response times, locked screens, or error messages. In other words, certain nonfunctional requirements must be included in your use case prioritization process.

You must also prioritize any new or modified use cases that emerge later in your project. At any point, requirements discovery can change the complexion of the set of use cases you are working on. Prioritization helps you manage that work and, of course, is essential for planning all the work in the project.

Consult with your customers on the prioritization scheme you plan to use. The classic way to rank your use cases is to group them into categories. One approach is suggested in RUP. Called MoSCoW, this strategy prioritizes use cases based on four categories: must, should, could, won't. Other ranking schemes use different terms (essential, conditional, optional; high, medium, low; urgent, desired, useful). You can also use analytic techniques that factor in the relative risk of building or not building each use case, the cost to implement each one, and the business value of each one, assigning a numerical value to each use case.[5] Whichever approach you choose, gain agreement from your customers and consistently use it throughout requirements development.

Tip 6: Trace Your Use Cases

***Tracing* allows you to track the lifecycle of each use case from its origin to its implementation. You establish a baseline, or starting point, for each use case (and its related requirements) that your project is committed to implementing. Without such a tracing scheme, you will never know whether and how you satisfied the original requirement. In addition, tracing lets you efficiently understand the impact of changing requirements as you move through your project.**

Tracing can go backward and forward. Backward tracing of a use case means that you link a use case to a customer need such as business goal, objective, or product vision. Forward tracing of a use case means you associate a use case with its test cases, design elements, and implementation artifacts. Trace matrices allow you to track the progress of your requirements.

Tracing can also go horizontally. As you define use cases, you trace their connection to other requirements documents such as business rules, actors, analysis classes, nonfunctional requirements, and functional requirements statements (brief text statements of a requirement). For example, a use case to place an order via a Web-based shopping cart may have a nonfunctional performance requirement such as "New customers (ones who have not visited the Web site before) shall pay for the items in their shopping cart in 15 seconds or less." As you might imagine, a single use case will trace to multiple other requirements. This tracing enables you to conduct impact

[5] See Karl Wiegers, *Software Requirements: Second Edition*, Microsoft Press, 2003.

analysis, finding which requirements are affected by a change in any single requirement. It is also useful in preventing problems downstream with missing or conflicting requirements.

Determine attributes you want to associate with your use cases, such as owner, status, source, and rationale. Requirements management tools such as RequistePro have numerous built-in attributes you can use. At first you may not know the values to assign to each of the attributes you choose to capture, but determining those that are important will have at least three benefits. It will remind you to ask the necessary questions to complete the attributes' values, help you respond to changes in the project, and enable you to understand the state of your requirements at any point.

Tip 7: Verify Your Use Cases

Verification **determines that your requirements were correctly defined. It involves testing or demonstrating that your use cases are correct, complete, clear, and consistent. You can verify your use cases by testing them with a variety of tools such as matrices, formal or informal reviews, quality checklists, and scenarios or test cases. Verifying a use case means challenging it by using other requirements models, such as scenarios, or by using walkthroughs and reviews.**

Reviews are a low-tech, high-efficiency way to test requirements. To ratchet up the effectiveness of your reviews, use quality assurance (QA) checklists or questions that might help the reviewers find errors. For example:

- **Is this use case in scope?**
- **Do the initiating actors exist on our context diagram? Is the success outcome of the use case visible on the context diagram?**
- **Are all the business rules that must be enforced in this use case accounted for?**
- **Have all the business terms mentioned in the use case been defined in our glossary?**
- **Has the domain model been updated to support the data referenced or stored in each use case?**
- **Is this use case testable?**

The very act of defining your QA checklist tends to make you build use cases that are complete and correct in the first place.

Another approach to verification is to conduct ***perspective-based*** **reviews.[6] In this approach, you invite various concerned parties—perhaps specific users who will play the role of specific actors, a tester, a designer, a help desk technician—to examine the use cases from their unique perspectives. Testers and quality analysts should also be active participants in your use case modeling process to ensure that various stakeholder voices are represented.**

Scenarios **describe typical uses of the system as narratives or stories. Each narrative can be a few sentences or paragraphs. Scenarios are played out in the context of a path through a use case. You can think of a use case as an abstraction of a set of related scenarios. Scenarios are one of the most effective ways to both elicit and test use cases. As you iterate through your use cases, try to test them with scenarios. Walk through each use case, beginning with the happy case scenarios. Then move on to the unhappy case (error and exception) scenarios involving business rules violations.**

Because scenarios represent real life to your users, they are often willing and able to generate realistic scenarios to bring your use case testing process to life. Better yet, having users supply scenarios tends to promote their commitment to the use case modeling process. Even if they can't be involved in the ongoing effort, they can help you fix and evolve the use cases during a one- or two-hour walkthrough with scenarios. Because scenarios are the basis for building test cases, testers should also be involved in building or at least reviewing your working list of scenarios.

Do your best to get business users to participate in your walkthroughs and reviews. In their absence, surrogate users, such as product development managers or business-savvy developers, can role-play being end users and uncover important defects.

Using any or all of these verification techniques will help you find errors in your requirements that you might otherwise detect much later—when they cost much more to correct.

[6] See Forrest Schull, Ioana Rus, and Victor Basili, "How Perspective-Based Reading Can Improve Requirements Inspections," *IEEE Computer* 33, 7:73-79. 2000.

Conclusion

I hope this paper has given you practical advice for increasing your project's success by using use cases. Each topic is based on a real-world application of use cases. Following these practices will set the stage for using use cases as your core requirements model.

The time you spend developing and managing requirements and use cases is only a small part of your overall development effort, but it can have a huge impact on the quality of your end product. By using proven practices, you can make your use cases a powerful means for delivering what your user community and business customers really need.

Glossary

actor an external role, human or devise, that interacts with the system.

actor map [also called actor hierarchy] A *requirements model* that defines the relationships among the *actors* in the *actor table* in terms of how their roles are shared and disparate. The map shows both human and nonhuman actors arranged in hierarchies.

actor table [also called actor catalog] A *requirements model* that defines the roles played by people and things that will interact directly with the system. The contents typically include names, brief descriptions, and other useful information such as physical locations, necessary job aids, knowledge or experience level.

business requirements High-level needs that, when addressed, will permit the organization to do things such as increase revenue, avoid costs, improve service, and meet regulatory requirements.

business policies The principles and regulations that influence the behavior and structure of the business being modeled. These policies provide the basis for decomposing *business rules*.

business rules Specific guidelines, regulations, and standards that must be adhered to. Attributes of business rules include its owner, its source, the source category (for example, human or document), its jurisdiction, and its relative complexity.

context diagram A diagram that shows the system as a whole in its environment.

direct user See *user*.

direct users list See *actor table*.

domain An area of study or concern, generally referring to a business domain at varying levels of granularity such as purchasing, banking, invoicing, manufacturing, etc.

domain model A *requirements model* that defines groups of information that must be stored in the system and the relationships among these groups.

end user See *user*.

event list See *event table*.

event table A table that defines the triggers of events to which the system responds.

facilitation The art of leading people through processes toward agreed-upon objectives in a manner that encourages participation, ownership, and productivity from all involved.

functional requirements *Requirements* that specify the functionality that users expect.

glossary A *requirements model* that defines the meanings of all business terms relevant to the system being built. These terms serve as the foundation for all requirements models and business rules; the goal of the glossary is to provide a common vocabulary on which all the stakeholders can agree.

low-fidelity prototype A representation of screens or screen flows created on whiteboards or posters or with posts on a wall.

nonfunctional requirements *Requirements* that specify the quality characteristics needed by the software (such as performance and response times), system needs (such as security and archiving), and technical constraints (such as language and database).

prototype Anything that captures the look and feel of the user interface to be built for the new system and allows users to play with it. This can take the form of a fully working, albeit bare-bones, "system," or something more low-tech such as screen shots and low-fidelity sketches.

QA checklist A series of questions, usually stated in binary format, about a *requirements model* or its elements, and also questions about how one model cross-checks another.

requirements The needs or conditions to be satisfied on behalf of *users* and *suppliers*.

requirements model A blueprint for a software product that takes the form of a diagram, list, or table, often supplemented with text, that depicts a user's needs from a particular point of view. Examples include *event lists*, *use cases*, data models, class models, *business rules*, *actor maps*, *prototypes*, and *user interface navigation diagrams*.

requirements workshop A structured meeting in which a carefully selected group of stakeholders and content experts work together to define, create, refine, and reach closure on deliverables that represent *user requirements*.

review The process of examining a work product, such as a model, a document, or a piece of code, so that people other than the one(s) who produced it can detect flaws.

scenarios Descriptions of typical uses of the system as narratives or stories. Each narrative can be a few sentences or a few paragraphs.

scope A broad definition of the who, what, when, why, where, and how associated with project goals and objectives. Scope determines the context for the user requirements effort.

scope creep The condition in which the scope of the project continues to expand as development work proceeds.

separation of concerns A principle of software engineering that involves identifying, isolating, and encapsulating concepts for the purpose of simplification, reuse, and adaptability.

software requirements *Requirements* specifically associated with a software solution to a business problem and a user problem.

statechart diagrams Diagrams that define how time affects your *domain model*, in terms of the possible states that elements of that model can assume and the transitions between those states.

surrogate user A stand-in *user* who takes the place of an actual user.

use case A description of a major function that the system will perform for external actors, and also the goals that the system achieves for those actors along the way.

use case map A *requirements model* that illustrates the predecessor and successor relationships among *use cases*.

use case packages Cohesive groups of use cases that can form things such as a logical architecture, a test group, or a release of the system.

user Anyone who affects or is affected by the product: a person or thing (devices, databases, external systems) that interacts directly with the system being modeled, or a person or thing that receives system by-products (decisions, secondary reports, questions).

user requirements *Requirements* specifically associated with the user problem to be solved

user interface navigation diagram A diagram that shows the layout of the user interface (screens, windows, dialog boxes, HTML pages) and the navigation that is possible among the elements of the interface.

walkthrough A form of *review* in which the producers of a *product*—for example, the subgroup who created the steps for a given *use case*—describe the product and ask for comments, questions, and corrections.

voice of the customer An expression that evolved out of the quality movement (Quality Function Deployment) in which spoken and unspoken needs of customers are defined.

References

The references below will help you understand how to elicit and specify use cases:

- Bittner, Kurt and Ian Spence, *Use Case Modeling*.
  Addison-Wesley, 2003.
- Gottesdiener, Ellen, *Requirements by Collaboration: Workshops for Defining Needs.* Addison-Wesley, 2002.
- Jones, Capers, *Patterns of Software Systems Failure and Success*.
  Thomson Computer Press, 1996.
- Kulak, Daryl and Eamonn Guiney, *Use Case, Second Edition: Requirements in Context*. Addison Wesley, 2003.
- Leffingwell, Dean and Widrig, Don, *Managing Software Requirements: A Unified Approach*. Addison Wesley, 1999.
- Wiegers, Karl, *Peer Reviews in Software: A Practical Guide*.
  Addison-Wesley, 2002

Online articles:

- Gottesdiener, Ellen, "Top Ten Ways Project Teams Misuse Use Cases — and How to Correct Them
  Part 1: Content and Style Issues" available at:
  http://www.rational.net/main/cata.html?USE_CASE=viewcontent&SERVICE_NAME=Vignette&SERVICE_SPECIFIC_ID=2396&RESOURCE_ID=267267&SEARCH_KEYWORD=use+cases&SEARCH_KEYWORD_POSITION=19
- Oberg, Roger, "Applying Requirements Management with Use Cases", available at:
  http://www.rational.net/main/cata.html?USE_CASE=viewcontent&SERVICE_NAME=Vignette&SERVICE_SPECIFIC_ID=767&RESOURCE_ID=441193

- **Probasco, Leslee, "What Makes a Good Use Case Name?" available at: http://www.rational.net/main/cata.html?USE_CASE=viewcontent&SERVICE_NAME=Vignette&SERVICE_SPECIFIC_ID=589&RESOURCE_ID=34572**

- **Sturm, Jake, "Defining Goals and Creating Summaries, Use Cases, and Business Rules" available at:**

  htp://www.rational.net/main/cata.html?USE_CASE=viewcontent&SERVICE_NAME=Vignette&SERVICE_SPECIFIC_ID=500&RESOURCE_ID=33136&SEARCH_KEYWORD=use+cases&SEARCH_KEYWORD_POSITION=199

Author Bio

Ellen Gottesdiener, Principal of EBG Consulting, Inc. (www.ebgconsulting.com) helps people collaboratively define and verify business, user, and technical requirements. She works with teams to shape their development processes, collaboratively plan their work and facilitate workshops. Ellen presents at industry conferences, has written numerous articles and *Requirements by Collaboration: Workshops for Defining Needs* (Addison-Wesley, 2002).

IBM software integrated solutions

IBM Rational supports a wealth of other offerings from IBM software. IBM software solutions can give you the power to achieve your priority business and IT goals.

- *DB2® software helps you leverage information with solutions for data enablement, data management, and data distribution.*
- *Lotus® software helps your staff be productive with solutions for authoring, managing, communicating, and sharing knowledge.*
- *Tivoli® software helps you manage the technology that runs your e-business infrastructure.*
- *WebSphere® software helps you extend your existing business-critical processes to the Web.*
- *Rational® software helps you improve your software development capability with tools, services, and best practices.*

Rational software from IBM

Rational software from IBM helps organizations create business value by improving their software development capability. The Rational software development platform integrates software engineering best practices, tools, and services. With it, organizations thrive in an on demand world by being more responsive, resilient, and focused. Rational's standards-based, cross-platform solution helps software development teams create and extend business applications, embedded systems and software products. Ninety-eight of the Fortune 100 rely on Rational tools to build better software, faster. Additional information is available at www.rational.com and www.therationaledge.com, the monthly e-zine for the Rational community.

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America 05-03 All Rights Reserved. Made in the U.S.A.

The IBM home page on the Internet can be found at **ibm.com**

TP 204