



ARTICLES

www.requirementsdevelopment.com

The Practice of Use Case Driven Development page 2

The Essential Unified Process successfully integrates practices of the three leading process camps: unified process, agile methods, and process maturity.

By Ivar Jacobson and Eric Naiburg

When Requirements Go Bad: Part II page 6

Errors of conception are the most significant kind of requirements errors, and they are the hardest to detect. In this final installment, Kurt considers errors resulting from inadequate specification or implementation of requirements.

By Kurt Bittner

Capturing Requirements with Use Cases page 11

Requirements are essential for quality, but few people know how to create them well. Todd explains how you can turn use cases into system requirements that are as thorough and precise as you need them to be.

By Todd Wyder

Describing Your Software for Outsourcing page 13

What if you could outsource your software development and have a guarantee that the results would be successful? That would be very powerful indeed.

By Steve Mezak

Capturing Business Rules page 16

Business rules provide the knowledge behind every business structure or process. They are, therefore, at the core of functional requirements.

By Ellen Gottesdiener

An Introduction to the Business Analysis Body of Knowledge page 19

The business analyst understands business problems and opportunities in the context of the requirements and recommends solutions that enable the organization to achieve its goals.

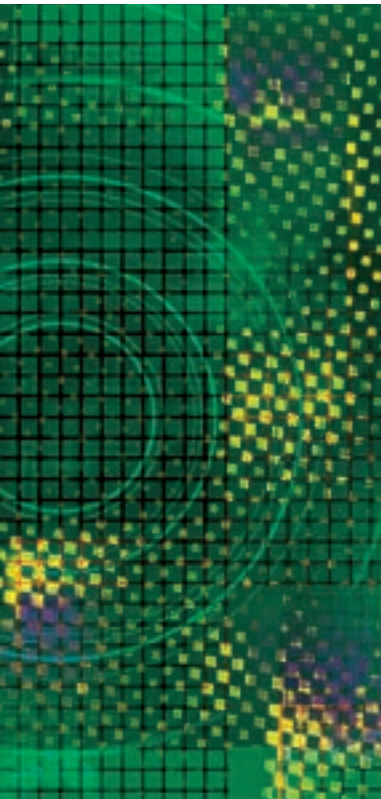
By Kevin Brennan

sponsored by

RAVENFLOW

The Practice of Use Case Driven Development

An agile, scalable approach to requirements management, development, and system testing



Ivar Jacobson invented use cases as a way to specify functional software requirements. He is the author of *Object-Oriented Software Engineering, and Aspect-Oriented Software Development With Use Cases*. Eric J. Naiburg is the author of *UML for Mere Mortals and UML for Database Design*.

THE PRACTICE OF USE CASE driven development offers a simple way of describing, developing, and testing software and software intensive systems. Its value lies in its ability to help both customers and team members better understand the opportunity at hand and work together to pursue it more effectively. Use cases, as the key elements for planning and progress tracking, play an important role in this development process.

Use cases describe ways of using a system without referring to technical details, such as database design and communication protocols. This makes them especially useful when describing interactive systems that respond to external events. Use cases have gained worldwide acceptance in a wide variety of industries as the best way to organize and think about requirements for both small and large systems.

This article discusses the works of Dr. Ivar Jacobson and his “Next Generation Process,” called the Essential Unified Process. We describe the essence of use case driven development in a concise, accessible, and—most importantly—usable manner. We hope this discussion will allow teams to benefit from the practice of use case driven development without feeling encumbered with a heavyweight, complex process.

Essential Unified Process

The Essential Unified Process (EssUP), a vision of the father of the Unified Process, Dr. Ivar Jacobson, stands on the shoulders of modern software development practices. It is a fresh start, integrating successful practices of the three leading process camps: unified process, agile methods, and process maturity. Each one of these possesses distinct strengths, whether structure, agility, or process improvement. EssUP is the first of a new generation

of processes that embody each of the following key characteristics; see Figure 1.

EssUP is unique in its presentation. It relies in many ways on a new concept called “Separation of Concerns” (SOC, or aspect-oriented thinking). SOC refers to the identification and consideration of specific concerns in order of priority. The application of SOC to process adoption greatly simplifies and focuses the approach. It is much easier and more intuitive to select your tailor-made software process to meet your specific concerns.

In our many years of experience in helping organizations, we have learned that few people read process material, whether it is from a book or website.

What Forms the Basis of the Process?

As Figure 2 illustrates, the Essential Unified Process contains five foundation practices, as well as three supporting practices. Each practice is separately defined, and these systematic divisions allow us to simplify the process description dramatically. Each practice can be developed, adopted, and applied independently from each of the other practices. This is very different from the Unified Process, in which the development practices are tightly integrated.

The independence of EssUP’s eight practices allows you to choose only those practices you think you need without having to deselect activities and artifacts. This offers the freedom to select the practice(s) you want and compose them with your existing process. This article will focus on the Use Case Practice.

Use Case Practice

The goal of the Use Case Practice is to capture requirements in an accessible form that can be

Requirements Development

used to drive development. This practice allows teams to:

- Work with customers to capture essential requirements.
- Work together more effectively to quickly develop a useable solution.
- Identify and deliver the value expected from the system.
- Establish an appropriate level of requirements detail to support their needs and the needs of their customers.
- Prioritize and subset requirements to identify a minimal solution and drive iterative development.
- Use a systematic approach to facilitate the correct design, implementation, and verification of system requirements.

Any action directed at a goal, whether it is software development or washing the car, can be bro-

ken into three areas: what we want to accomplish, how we can accomplish it, and the core competencies required to do so. In the following sections, we will discuss what is produced, the process for producing these artifacts, and who is responsible for the work to be done.

Things to Produce

In practice, use case driven development requires that a set of artifacts be created throughout the software development lifecycle (SDLC). These artifacts are used to document the requirements and relate them to different parts of the lifecycle. Figure 3 shows the artifacts to be created and where they play a role in the SDLC.

The opportunity box in Figure 3 represents the shared conception of the customer's real needs and is used to show that the specified system should capture those needs.

The backlog is a prioritized list of work items, such as requirements, defects, and changes. This list is used to plan and track progress.

The use case model contains the system's actors and use cases, where:

- An actor is somebody or something that interacts with the system, and
- A use case specifies desired system behaviors by describing how actors interact with the system. A use case consists in a sequence of events that describes how an actor uses the system, as well as how the system responds to the actor, in order to achieve a goal.

A use case specification describes a use case's flow of events and any other requirements associated with the use case.

A use case module contains all the information related to a particular use case. It is used to organize and relate the specified system, the implemented system, and the executable system, allowing the use cases to drive the development from specification to execution.

A supplementary requirement is a requirement of the specified system (typically a global or non-functional characteristic) that cannot be effectively captured within the use case model.

A test case describes how a specific scenario (covering one or more use case flows) will be verified. Test cases are used to determine whether the release meets the specification.

A use case realization describes how the implemented system realizes a specific use



Figure 1: Characteristics of next-generation processes.

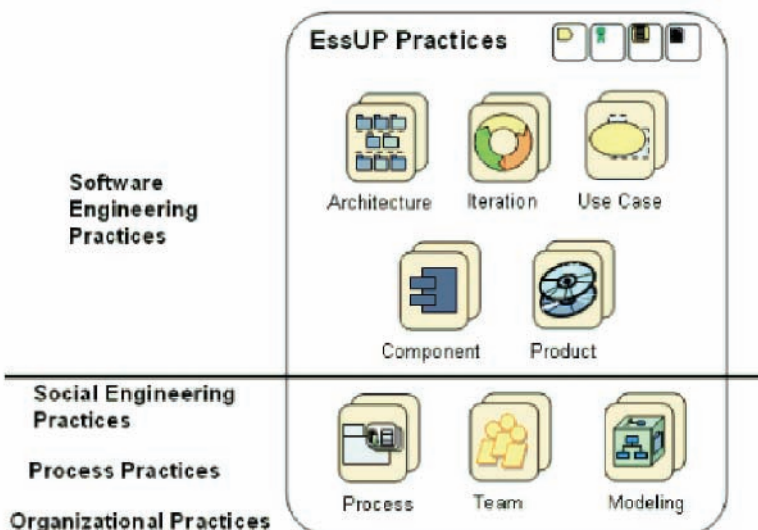


Figure 2: The eight practices of the Essential Unified Process.

Requirements Development

case and is used to enable a systematic implementation based on a use case specification. It is also used to maintain traceability between the use case and various elements of the implemented system (i.e., source code, unit tests, project files).

A test specification tells how to execute a test in order to meet a specific test objective.

A test is the execution of a set of test cases to verify that the system executes as specified.

The results of the test are captured as a set of test results.

Things To Do

Figure 4 illustrates a work flow for the Use Case Practice.

Understand the need. Reaching agreement with customers on the real problem to be solved is critical, but should be achieved prior to this practice.

Find actors and use cases. Establish the system boundary and scope by identifying actors and use cases and capturing any supplementary requirements that are needed. This process yields both the use case model and the supplementary specification.

Select and prioritize use cases. Plan development activities by selecting and prioritizing use case flows and the scenarios they define. Add each use case to a use case module that corresponds to the aspect of the system to be developed. This results in requirements prioritization as well as backlog population.

Specify use case module. Reach a shared understanding of the desired system's behavior by specifying use case flows, test cases, and supplementary requirements for a particular aspect of the system. This yields the use case specification and the test cases for the use case module.

Identify Tests. Identify the set of tests that will be used to verify that the executable system delivers the specified use cases. This results in the definition of a set of tests and their objectives.

Realize use case. Understand how adding use cases will impact the system by distributing their behavior to the implementation elements. This will add use case realizations to the use case module.

Build test. Decide on how to implement a specific test by selecting a minimally sufficient set of test cases based on the use cases and their flows of events. This completes the test specifications for the desired tests.

Implement software. The design, code, and unit test of the implementation elements occurs beyond the scope of the Use Case Practice, but these should be based on the distribution of the use case behavior to various implementation elements. This step yields working, unit tested software.

Execute Test. Verify that a release is correct by executing a specific set of use case based test cases. This will yield test results indicating how well the software executes the use cases. Progress can be assessed by

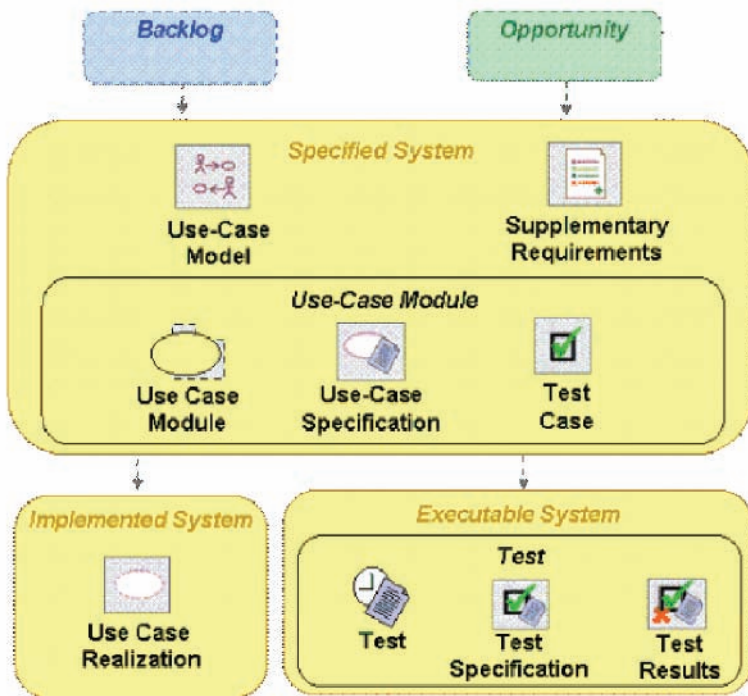


Figure 3: Artifacts produced in Use Case Driven Development.

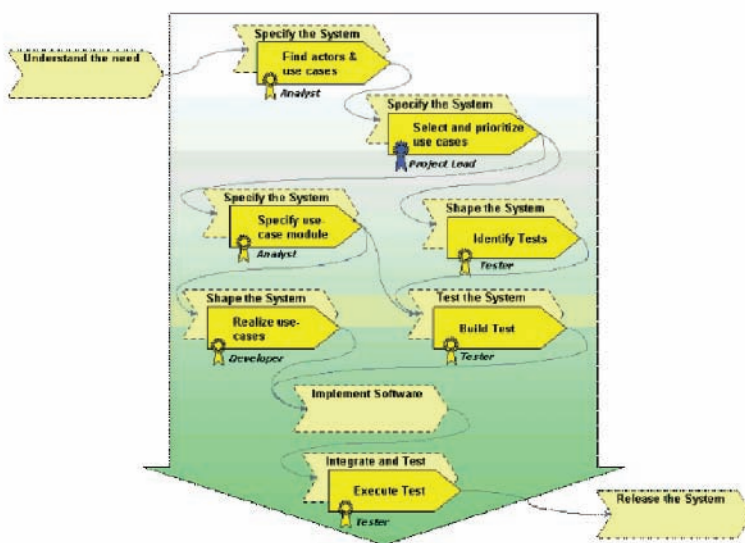


Figure 4: A work flow for the Use Case Practice.

counting the number of use cases and use case flows (weighted by size and complexity) that have been implemented and successfully verified. Compare the completed items with the plan to gain confirmation of the team's progress.

Release the System. When the use case driven development is complete, the system can be

the most valuable functionality to the business and its users.

The Analyst competency is required when eliciting opportunities, needs, and change requests. Turning these into agreed elements of the specified system, often actors and use cases, is accomplished by the analyst.

The Developer competency is required when designing, implementing, and unit testing software that meets the requirements.

The Tester competency is required when verifying that the executable system behaves as specified and satisfies the use cases and supplementary requirements.

The Project Lead competency is required to coordinate and lead the team in the specification, implementation, and verification of a solution that meets the needs of the customers.

Use cases have gained worldwide acceptance in a wide variety of industries as the best way to organize and think about requirements for both small and large systems

released. The release of the software occurs outside this practice.

Core Competencies

Figure 5 shows the core competency roles required by the Use Case Practice.

The Customer Representative competency is required to ensure proper representation of the project's customer and user community. This competency plays an important role in shaping the use case model and prioritizing the requirements (the use cases, flows of events, and supplementary requirements) to ensure that the software produced delivers

When To Use

As Terry Quatrani wrote in *Dr. Dobb's Requirements Development*, Issue #3, "Use cases capture the functional requirements of your system and they drive the entire software development life-cycle...This means that in order to be successful, you must get your use cases right." Every software development project aims to succeed by meeting the needs of the customer, whether internal or external. For this reason, it is crucial that a clear understanding of what is to be developed both exists and is documented. Use cases are a great way to standardize the language and format requirements definition, description, and modeling. The Use Case Essentials practice should be used when the team faces challenges in:

- Creating a solution that meets both the customers' and the users' real needs.
- Meeting time and cost expectations.
- Delivering defect-free solutions.
- Capturing stakeholder requirements in an agile and efficient manner.
- Prioritizing and subsetting requirements to identify a minimal solution.
- Identifying meaningful sets of requirements that can be built and improved upon in an iterative and incremental fashion.

The concept of use case driven development was initially presented by Ivar Jacobson at OOPSLA in 1987. The concept of use cases has become so popular that it is now the basis of the standard approach for modeling and capturing requirements for object-oriented development, as well as a core part of the Unified Modeling Language. Use case driven development itself has led to many popular iterative and incremental development methods. □

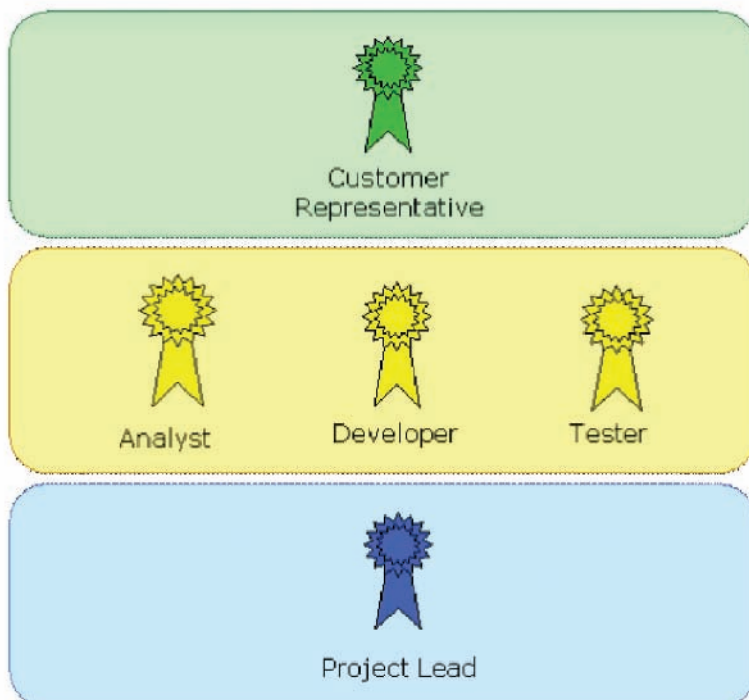
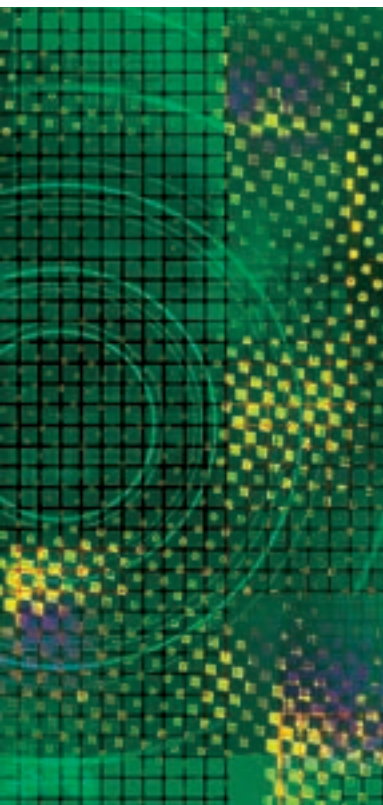


Figure 5: The core competency roles required by the Use Case Practice.

When Requirements Go Bad: Part II

Errors of specification and implementation



Kurt Bittner works for IBM on software development product strategy. In a career spanning 24 years, he has successfully applied iterative approaches to delivering software solutions in a number of industries and problem domains. He is a co-author, with Ian Spence, of *Use Case Modeling* (Addison-Wesley, 2002) and *Managing Iterative Software Development Projects*, (Addison-Wesley, 2006).

IN PART I OF THIS two-part series (*Dr. Dobb's Requirements Development*, Issue #4), I discussed requirements errors arising from misconceptions about the problem being solved or by the goals of a development project. In this final installment, I will consider errors resulting from inadequate specification or implementation of requirements.

Errors of Specification

Errors of specification, or errors arising from the way a requirement is described, are the most common requirement errors. For the purposes of this discussion, it is useful to divide these errors into the following subcategories:

- Under-specification (resulting from incomplete or vague descriptions).
- Inaccurate specification.
- Inappropriate method of specification.
- Over-specification.

Each type of specification error has different root causes and requires different avoidance strategies.

Under-Specification

Under-specification is the most frequent type of specification error, and it takes a variety of forms. For example, when it comes to projects that employ scenario-based requirements description approaches, such as use-case modeling, a common error is the omission of important alternative flows. Alternative flows describe a system's response to errors and offer alternate behavior paths. Failing to identify these may result in an implemented system that either fails to handle important exceptions and error conditions or fails to offer functionality that is important to the solution's stakeholders. Hence, it is important to pay careful attention during reviews and walkthroughs to those system behaviors that have not yet been specified.

Another common error is when requirements take the form of "lazy descriptions," which merely point to an example and expect the reader's imagination to generalize and fill in the details. Such a description may look something like the following:

The system must support the translation of foreign currency transactions, e.g. U.S. dollars to Euros.

Here, the offending phrase has been underlined, and its problems are two-fold. Firstly, the algorithms used to translate currency need to be specified. That is, each step required to perform a currency translation needs to be made clear; one cannot assume that the software developer will know how to handle these calculations. Secondly, the particular currencies to be handled must be enumerated. After all, if each currency is not specified, one cannot be sure they will all be supported by the system.

Omitting requirements, whether they be entire alternative flows or important details, amounts to under-specifying the project at hand. These types of errors typically result in delivered systems that fail to meet the needs of their users. After all, unless the desired system behaviors are identified and communicated, it is unlikely that they will be delivered. It may be helpful to use a tool that can generate visualizations of flows, as these often draw attention to areas of under-specification.

A somewhat less frequent problem is failing to include a glossary of common terms. Sometimes this occurs when a term is defined in one context (such as "in-line" in a use case) but not in a common place where a definition can be found easily when needed. Keeping a glossary of common terms can help, but you should also have links from where the term is used to the glossary in order for this to be useful. Linking does not require special tooling other than the ability to embed hyperlinks or URLs in the descriptive text.

A final cause of under-specification is vague description. When writing requirements, it is absolutely essential that language be used precisely. I have had conversations about tools that analyze text and identify ambiguous descriptions, but what you really want to ensure is that everyone on your team has reached a common understanding. The tools can certainly be helpful, but only active discussions about the requirements can reveal whether or not a shared understanding has been

When writing requirements, it is absolutely essential that language be used precisely

achieved. We need to banish the practice of writing requirements that we can “throw over the wall” to developers or testers and instead pursue a more open, communicative approach. What is important to realize is that requirements are what motivate discussions, but it is the discussion that matters most.

Inaccurate Specification

Sometimes the specification is complete, but incorrect. Often, these errors are not obvious, and it can take some careful review to uncover them. Just as a misplaced semi-colon can completely change the meaning of a statement in a computer program, a mistake in the way a requirement is described can be detrimental. A common technique in safety-critical software development is to perform code reviews, and a similar technique can be applied to requirements. In both cases the effort is labor-intensive and tedious, but it is often the only way to avoid errors.

A frequent contributor to inaccurate specifications is that most teams barely have enough time to specify requirements in the first place, let alone adequately review them. A strategy for working within such constraints is to focus only on the most important requirements (the “must do” requirements), rather than documenting and reviewing all of them. To the purist, this may sound like heresy, but here is the rationale. If you really can’t review all of the requirements for correctness, some is better than none, and most would agree that it makes sense to focus on the most important requirements.

If you have time left, you can move on to the “should do” requirements, but if you can’t afford the time to review these, it may not be worth it to write them down. Instead, stay focused on the most important requirements with the idea that if the team is really that constrained, the “should” requirements will probably not get implemented anyway. In other words, start managing scope early, and only describe the requirements that you know are going to get implemented.

The key to uncovering incorrect specifications is to make time for walkthroughs and discussions. I find reading most requirements specifications to be mind-numbingly dull, and I expect that I am not alone. Subject matter experts from the sponsoring business organization are probably even less receptive to reviewing long specifications. In today’s world, most people have the attention span of a gnat (present company excluded, of course), so sending around requirement specification documents for review is usually too passive an approach to get good results.

A more promising strategy is to use storyboards (informal sketches of the user interface—not full-fidelity prototypes) to walkthrough use case scenarios while highlighting specific areas for discussion. This approach will be more engaging for everyone, but I do raise a couple of cautions:

- Don’t use high-fidelity prototypes that look and behave like the real application. You want to get rapid, early feedback, and if you spend too much time developing a high-fidelity prototype, you will delay getting feedback unnecessarily. In addition, if the prototype looks “too good,” both you and the stakeholder may be reluctant to make changes. Rough sketches, like those used in the animation industry to sketch out the plot line for a film, are sufficient for having useful discussions about the desired behavior without requiring much investment or attracting commitment.
- Don’t wait to get feedback. Start these sessions early, and have a lot of them. A number of short, informal 10-minute sessions are better than one huge 2-hour session. They can be held sooner and more frequently, and the quality of feedback will be better. Your goal should be to have continuous (or nearly continuous) communication about the project.

Over-Specification

Sometimes the specification is over-done, including information that is inappropriate or excessive. I once encountered a project that had 18,000 requirements for a customer-service system. No one could comprehend the specification, and, not

surprisingly, its implementation had failed several times.

Over-specification is usually grounded in good intentions, often as a response to prior failures from under-specification. But over-specification brings its own problems, usually presenting as requirements things that are not really “required.” They may, for example, be things that should be left to the creativity of the developer. Over-specifications tend to treat

Use case description can capture flow of events and user context, but use of storyboards or prototyping tools better capture the UI.

Over-specification can also be a problem when describing implementation details. Consider this fragment from a use case:

“...the user enters their password. The system then goes to the User Authentication System and retrieves the stored password using the user ID. The system then validates the entered password...”

What’s wrong with this? The author has over-specified the process. First, there may or may not be a User Authentication System, and whether or not there is should not matter from the perspective of the use case. The user’s ID and password do need to get validated, but it really is up to the developer to decide how to do this. The use case fragment should read:

“...the user enters their password. The system then attempts to match the entered password with the password on record for the entered user ID. If there is a match, the user is granted access...”

If there actually is a User Authentication System and the developer must use this to validate the password, this can be stated in a constraint requirement:

“Constraint: The User Authentication System is used to validate all user information”

The benefit of expressing the constraint in this way, rather than in the use case itself, is that if the authentication mechanism changes, you only have to change it in one place. The use of the User Authentication System will be important when designing and implementing the solution, but the requirements don’t have to describe how this will be done. It is sufficient to state that it needs to be done and leave it at that.

Another area of over-specification may more precisely be termed “misplaced specification.” This occurs when there is a need to describe the information captured or manipulated in a use case. Putting the information “in-line,” in the text of the use case, can quickly become burdensome. Consider this fragment:

“...The user then enters the customer’s first name, their last name, their street address, their city, their state, their home phone number...”

A better practice is to call all of this the “customer information.” Rather than repeating each piece of information every time we need to reference it, we can create a glossary term called customer information and use this more concise reference when

We need to banish the practice of writing requirements that we can throw over the wall to developers or testers and instead pursue a more open, communicative approach

symptoms (and lots of them) without getting to the root causes of the problems to be solved.

Over-specification often occurs in the user interface, where users may have grand ideas about how things should look and feel. User interface design is a very challenging thing, however, and the best solutions are usually simple and intuitive, utilizing effective metaphors and organization for information. Most users are not very experienced with this—they don’t know what they want, but they know what they don’t want when they see it.

As a result, it’s best to employ people with experience in human factors and user-interface design, engaging them as part of a team that includes the users and developers. The team can then work through the best way to represent information, the best way to provide the most natural flow of information and function, and iteratively come up with a design for the user interface. This process usually starts with storyboards and outlines for the use cases, evolving the two in parallel with storyboards. Later, the process may involve prototypes that describe the look and feel of the system, and use cases that describe the flow of the system.

Just like the old saying, “if all you have is a hammer, everything looks like a nail,” people working with use cases often fall into the habit of putting details about the UI into their descriptions. The practice initially seems harmless, but it often becomes unwieldy and impractical, as use case descriptions are typically unable to capture the dynamic nature of the UI. Prototypes and storyboards are usually better tools for describing the UI, but they lack the facility to describe flows of behavior effectively. Keeping these two things separate yet adequately related is ideal.

needed. This saves a lot of typing and allows us to change the information in one place if an update is required.

It is an extension of this idea to consider that there are sometimes relationships between information. For example, customers of an insurance company may have lots of different bits of information, such as billing addresses, insured property addresses, coverages for the insured properties, names and

Don't use high-fidelity prototypes that look and behave like the real application

addresses for beneficiaries, and so forth. Keeping all this information in a glossary is possible but this makes it hard to see how the pieces of information are related.

What we really need is what I call a “domain model” (or a model of the concepts used in the problem domain), though different terms such as “business entity model” can be used as well. It is helpful to have a way to describe the concepts embedded in the requirements. By having a richer set of conceptual tools for describing the requirements (including use cases, prototypes or storyboards, and glossaries and domain models), the requirements actually become easier to understand and easier to manage.

Errors of Implementation, or, The Importance of Testing

No discussion of requirements errors would be complete without some consideration of implementation errors. These show up when the concept is right, the specification is right, but the requirement is either not implemented correctly or not implemented at all. It is easy to say that this is not a requirements problem but rather a project management problem or a testing problem, but I view such distinctions as artificial and ultimately not very useful. There is no point in writing requirements if you are not going to implement them, and the only way to know if they are implemented correctly is to test them. The functional areas of project management and testing are inextricably linked with requirements.

Let me restate something I just said a little differently: Every requirement should have one or

more associated tests that verify that the requirement was implemented correctly. If you're not going to test it, don't bother with writing the requirement because you'll have no way to know if it ever got done.

It is possible to combine test cases and requirements (as some approaches do) by writing all requirements in the form of a test case; this saves the “overhead” of a separate requirement. I think this works fairly well in some cases and not so well in others. Consider the case where we want to make sure that the system is able to support 1000 concurrent users. We could, with some work, define a set of tests that simulate the load created by 1000 concurrent users, but the tests are likely to have a lot of implementation details that obscure the actual requirement. This makes it hard to review the requirements with people who can tell you whether the 1000 users is the right number or not. As a result, I generally prefer to have a separate statement of what we're trying to achieve and let the test cases focus on how to measure whether the requirement was met.

Failure to test requirements usually arises from a resource problem—there may not be enough people to test. This usually results from what I call the “dysfunctional functional organization” mindset that puts people into narrow functional silos. Developers need to test, but so do analysts, users, and anyone else who might have “tester” somewhere in their role description. There are specialized skills needed to deliver successful results, but there is no reason for rigid barriers between team members. Just as there is no reason to write requirements for things you're not going to test, there is no reason to develop things that you are not going to test. If you're getting behind on testing, someone is not pitching in and is potentially doing valueless work.

Assuming that you have tests for every requirement, then figuring out what did or did not get implemented becomes much simpler—either the test ran and it passed, or it ran and it failed. Failure may occur because the requirement was not implemented, or because it was not implemented correctly. Either way, the desired result of the requirement is not being achieved.

Another requirements implementation issue relates to the availability of test data to support whether a requirement was implemented. I sometimes work with project teams who have an overall mandate that every requirement must be traced down to code, usually to ensure that a particular require-

ment was implemented. I find this practice to be costly to the point of impracticality, and ultimately it is ineffective.

Many requirements, such as the “support 1000 concurrent users” requirement, are satisfied by architectural considerations that do not have a single locus in the code—they emerge from the overall capability of the system. Requiring that these kind of requirements be traced down to code is pointless. In addition, just because the developer says that a particular section of code was designed to satisfy the requirement does not mean that it does. As the old saying goes, “the proof of the pudding is in the eating,” meaning that only actual experience will tell you if it is good. Only testing under real-life conditions will tell you whether the requirement is really met.

Conclusion

Errors in the specification of requirements can take several forms. Requirements may contain too little information, leaving them vague or incomplete. They may also be over-specified, leaving them full of irrelevant or extraneous detail that makes the important parts difficult to see. Striking the right balance often requires knowing which details must be conveyed and which can be left to the developer to decide. Making the right choices usually involves having open and meaningful conversations between team members to agree on the appropriate level of detail for the project at hand.

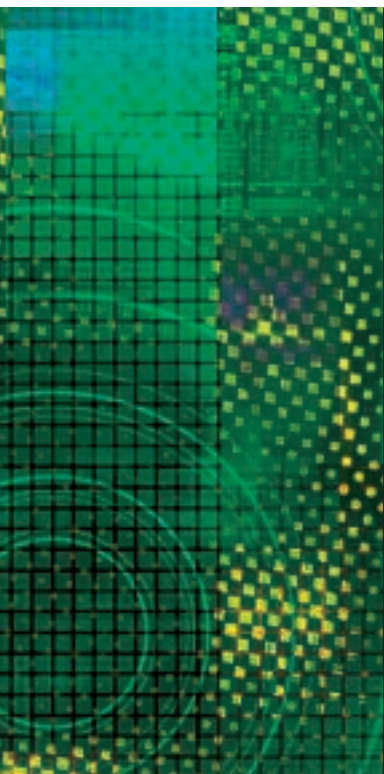
Using appropriate techniques for different aspects of the description can help to simplify the description while making it more complete. Using storyboarding and UI prototyping tools can supplement the requirements specification with additional, essential information; and domain models and glossaries can augment the requirements specification while making it easier to understand and manage.

Once the appropriate level of description is agreed upon, it becomes easier to spot remaining requirements problems; for example, cases where the specification is simply wrong. These usually arise from miscommunication, but they are easily remedied through frequent and open feedback between members of the extended team (including subject matter experts).

Finally, requirements need to be tested. And testing is the only reliable way to know if a requirement was actually satisfied. Every requirement should have one or more tests, and there should be no tests that do not derive from requirements. □

Capturing Requirements with Use Cases

Requirements are essential for quality, but few people know how to create them well. Todd Wyder explains how you can turn use cases into system requirements that are as thorough and precise as you need them to be.



USE CASES ARE BECOMING more mainstream as a method for capturing requirements, as evidenced by the endorsement of big companies and methodologists such as Booch, Rumbaugh, and Coad. One benefit of use cases is that each one encapsulates a set of requirements. This encapsulation lets you easily manage and track the use cases individually and provides a better alternative to prose requirements.

There is more to effectively using use cases than just capturing them and putting them into diagrams. As you implement use cases, you need to validate them, determine their size, and establish a plan for implementation. Then, you need to incorporate the use cases into your system design and turn them into code and documentation. Throughout this process, you must also be aware of the status of each use case. This article will discuss ways to do all these things.

Validating Requirements With Use Cases

Once you've captured a use case, you need to confirm whether it accurately describes the system and is truly needed by the system's users. Sometimes, in the course of development, you realize some of the requirements you've created are unnecessary or peripheral to the main purpose of the system. You must identify these as soon as possible, so you can work on the functionality that is the most valuable to the customer. But how do you determine which use cases are most important to your users? You use a method called "Quality Function Deployment" (QFD).

QFD helps you weigh use cases to determine which ones are important and which are discardable. To use QFD, users representing each group in the actor catalog are given a list of the abstract use

cases and \$100 in virtual cash to spend on the ones they think are the most important. The amounts are then tallied to determine which features are the most desired.

When using QFD, it's important to keep one caveat in mind: Make sure you include on the list all the obvious features the users will expect in the system, because it's very likely these features will not come up while you're gathering use cases. For example, in the banking program, users would obviously require a "transfer funds" use case. However, because they expect this feature to exist, they might not consider it important and not allocate funds for it. It's important to allow for this and include the basic functions of the system into the QFD.

Sizing

Because use cases describe functionality from the user's point of view, they can be directly converted to function points. Assigning function points to use cases helps us understand how large a use case is and the associated effort needed to produce it.

We can use this knowledge in iterative development to divide the iterations into roughly equal sizes and determine earned value. Some companies' use earned value to recognize revenue. You can do this by dividing the estimated cost of a program by the number of function points, which yields a cost per function point. Each use case then becomes part of the executable. Multiply the number of function points associated with each use case by the cost per function point. The result is the amount of revenue recognized. You can use this same technique for project tracking by using the number of function points delivered to determine the progress on the project.

Todd Wyder is Senior Vice President and Chief Information Officer for SmithBucklin, where his primary responsibilities involve the strategic and operational application of technology to achieve client and company objectives.

Iterative Development

Determining the implementation order of requirements involves several conflicting factors: the needs of the customer, the needs of the development team, and the needs of management. The customers would like to see the parts of the program they want implemented first. The

Use cases are becoming more mainstream as a method for capturing requirements

development team needs to work on the most complex parts of the program first, so it can move from high complexity to low complexity. Management would like to work on the parts with the highest risk, so they can move from high risk to low risk.

So, how do you balance all three concerns? The customer's needs are known through QFD. If you deliver the iterations based on the QFD scores, you ensure customer satisfaction. But to also satisfy the needs of the development team and management, it's best to rank each use case for complexity and risk as well. Here's one way to do this: Rank complexity on a scale from one to five, with one being very simple, four being very complex, and five unknown. Rank risk similarly, with one being marginal, four being high risk, and five unknown. Once you've rated all the use cases, multiply complexity, risk, and the QFD percentage to get a weighted value that takes into account customer satisfaction, complexity, and risk.

Design

Use cases can help you create your system design and also serve as the foundation of design reviews. Use cases can be readily converted to object, interaction, and event diagrams and used as a basis for CRC cards. In a design review, use cases force designers to show how each use case is enabled by the design and which things in the design are not part of any use case. This ensures all requirements are implemented and no unnecessary work is done.

Testing and Documentation

Use cases are the backbone of testing and documentation. If the use cases are clearly stated and testable, they form the basic system test plan. They are also well suited for acceptance testing, in

which users test all use cases on the system and approve the system's performance of each one. Because use cases represent the user's perspective, they can form the initial user manual, online documentation, or help file. Some divisions of Microsoft use a similar technique; they write the user manual first and it becomes the specification for the program.

Use Case Tracking

To make sure you deliver what the customer has asked for, you need to know the status of each use case. Knowing where each use case is within the software lifecycle is valuable for managing the project and determining status. You can accomplish this goal by assigning each use case a Work Breakdown Structure (WBS) in the project plan. Then, as you track the project, you also track each use case. This also yields a method for determining WBS's.

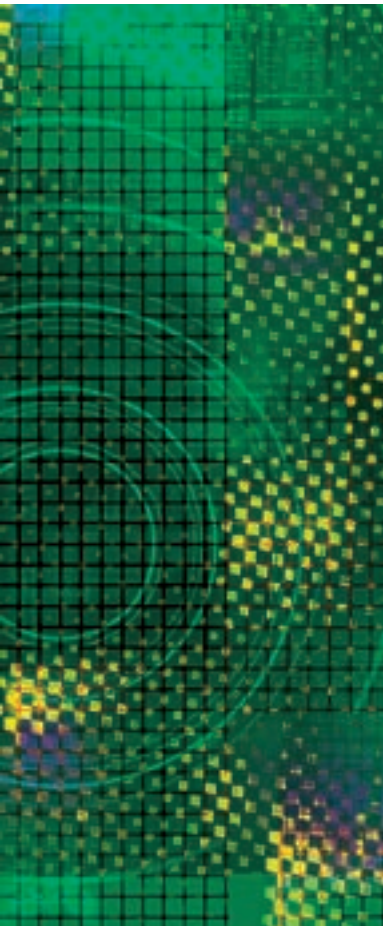
As your project progresses, you need to manage and control your use cases. A repository makes this possible. At my company, we keep the data in a groupware database product such as Lotus Notes and include the following information on each use case:

- A short descriptive name of the use case in the form of an action verb and noun, such as "export to spreadsheet."
- A detailed description of the action the use case performs.
- The preconditions: other activities that must take place before this use case executes.
- The post conditions: the actions that will happen to the use case after it executes.
- The exceptions: what happens if the use case fails.
- The pattern name from Coad, Gamma, and so on.
- The Work Breakdown Structure (WBS).
- The number of function points required.
- The location of the design files for the use case.
- The location of code files for the use case.
- The modifications made to the system, the date the modifications were made, and the name of the person who made them.

Use cases are a valuable tool for capturing and managing requirements. You can use them in all facets of the software development life cycle. As you move throughout the different phases of your next project, think of how use cases could be involved and how you would manage them. If you manage and track use cases well, you'll be able to use them to their fullest. □

Describing Your Software for Outsourcing

Agile approaches to software development can ensure reliability



Steve Mezak is founder and CEO of Accelerance Inc. and author of *Software Without Borders* (www.SoftwareWithoutBordersBook.com) from which this article is adapted.

WHAT IF YOU COULD outsource your software development and have a guarantee that the results would be successful? That would be very powerful indeed.

A guarantee is one of the most attractive enticements you can offer your customers to help them decide to buy your product. Actually, marketing folks refer to such incentives by the fancier name of “risk reversal,” since they are meant to reverse the feeling of risk within your customers. Whatever you offer as a risk reversal should motivate the feeling that it is less risky to do business with you than it is to continue the present course. The customer should feel that they have a great deal to gain and very little to lose.

The most common risk reversal is a guarantee. (And the offer of a free trial follows close behind.) With a guarantee, not only will your customer get the benefits of your product or service for a reasonable price, but they are also promised to see results.

Why don't most software outsourcing service firms offer a guarantee? Because developing custom software on time is notoriously difficult to do. An on-time delivery guarantee is virtually never offered for software development, and it would seem too good to be true if it were.

This is primarily because the end goal of a software development project is difficult to define and is not often completely known at the outset. The competence and ability of software engineers is rarely the issue, especially if you have carefully selected your vendor.

The key, you might think, is in defining your software as carefully as possible. If you do this well, it seems you may reverse your risk in outsourcing software development. But this is not always so simple.

The Old (and Obsolete?) Waterfall Method

In the past, efforts to make the software development process more productive focused on creating better specifications. As the theory went, you

should first try to do a better job of capturing the requirements for what your software is supposed to do, including both features and responsiveness.

Second, you should create a good design and description of how the requirements are to be achieved. Then you can start coding. With enough detail in your specifications and design documents, you expect to get guaranteed results.

The waterfall approach is still the recommended way to communicate the desired behavior of your software when you are outsourcing individual modules or reimplementation of an existing system. But it is rare to know all the details of what your software should do when you are at the beginning of a new project. It seems crazy to expect guaranteed results in this kind of situation.

And in fact it is.

Observant software engineers have realized that they jump back and forth between the different waterfall boxes during the course of developing new software. Enforcing a sequential process of requirements specification, design, and development is artificial and counterproductive for new software projects.

The Whole Enchilada

What if you decide to outsource the creation of your entire software application? Can you really specify your software application completely? Will you know all the details of what you and your users want the software to do? Sometimes you might.

If you are rewriting an existing application—perhaps converting a client/server application to a web application—then you have a well-defined goal. You want to make the web application behave as much like the client/server application as possible. Your existing software functionality, user documentation, and old specifications are a great start for defining the new system. In such a case, you have a pretty good chance of defining the software completely.

However, if you have only broadly defined goals with limited details about what your software will

PRECISION			
Low		Medium	
High			
Use Cases	List of actors and goals	Use cases with success scenarios	Use cases with success and error scenarios
User Interface Design	Screen flow diagram	Screens with data entry and display items and actions defined	Screens defined in HTML or other useful code

Table 1: Amount of precision needed in use cases and user interface design.

do, the old waterfall techniques are not the best approach.

Using Use Cases

The use case is the main tool of the business analyst, marketing professional, and software designer for describing how your software is used. Written use cases are an extremely useful tool for describing your software. They explain how users interact with your software to achieve their goals. Use cases should be a major part of the requirements of your product.

Use cases are part of the Unified Modeling Language, or UML. UML is a set of tools and techniques for the description and definition of a system's behavior and architecture. UML is popularly used in the design of software applications and systems, and the role it gives to use cases is that each should include both a high-level diagram and a detailed description in text.

The length and quantity of your use cases depends on two primary factors: your software's complexity and the expertise of your development team in the application area.

For example, I once managed an outsourced development team that had previously developed a software application very similar to what we needed but with a different style of user interface. All the major use cases were already well understood by the developers, and we needed only brief descriptions of use cases for a few new features.

However, a significant amount of time was spent designing the new user interface. As it turns out, use case writing and user interface design are separate tasks, and we will discuss this point in more detail later.

When you are creating a new software application, rather than adding to an existing one, you'll need to create your use cases from scratch. The best way to start is to identify the kinds of users, or actors, that will interact with your software.

What Do You Mean, Precisely?

Use cases tell your developers what they cannot discover for themselves, especially if they are several

oceans away. Use cases tell them what your users want, need, and expect your software to do. If you can explain that, then good engineers can figure out how to build the software.

But how much detail do you need to provide? If you are spending more time describing your software than it would take to write it, something is wrong. You should start out with a low level of precision in your use cases and then increase the amount of detail needed to communicate with your engineers effectively. There is no point in providing more detail than is needed.

In Table 1, the amount of precision you need in your use cases is described as low, medium, or high. Low levels of precision are okay for a development team familiar with the application area, or to get started quickly on a prototype when details are not yet known. For example, in the case I described earlier, where the software project needed just a new user interface, I was able to use low precision for use cases and medium precision for the user interface.

In another project, which involved the creation of a web application product, we used medium precision for the use cases and high precision for the user interface. Error scenarios were straightforward and were added later, as they were encountered in development. The user interface was created in HTML before any coding was done. Then the HTML pages were linked together as a demo for early customers and investors.

Medium or high levels of precision are often required to effectively outsource software development offshore. The lowest level of precision for use cases involves listing the actors and their goals in using your software product. If you provide only low precision use cases, you will likely need more frequent iterations and a higher level of collaboration with the offshore software development team.

Get a Handle on Your User Interface

Writing use cases is different from designing the user interface for your software. If you have a user interface designed, it is best not to let your use cases degrade into descriptions of how to work the user interface (click this, then that, etc.).

This is because a use case is supposed to describe the goals and intentions of your software's users at a much higher level. Creating use cases is a way to draw out desired software behavior from the users and stakeholders; it should not be a lecture on how the system will work.

This is not to say that the two processes should not be related. Designing the user interface at the same time that you are writing the use cases will provide valuable insight into the behavior of your software. New functions will be discovered, and others may be found to be impractical. Furthermore, as the old saying goes, “a picture is worth a thousand words.”

The use case is the main tool of the business analyst, marketing professional, and software designer for describing how your software is used

Providing screen shots or screen sequences is a very effective technique for communicating the desired behavior of your software to an outsourced development team.

I recommend developing the user interface while capturing the use cases as a way to confirm what you are hearing from users and stakeholders. The user interface design is a common device for both confirming system function and specifying the software to developers.

Designing a Completely New Software Application

We have covered several techniques for specifying your software and ways to capture your software’s requirements. Unfortunately, such techniques are often not enough to ensure success when you are creating a completely new software application. Specifying every detail ahead of time is difficult, even bordering on impossible.

It requires tremendous forethought to create a specification that covers the details of use cases, exceptions and error handling, performance requirements, and the many other details that are often not yet known at the beginning of software development. And the specifications created are typically ponderous documents that few will read before they become out of sync with the software under development.

Most software engineers are not dummies, and you don’t need to specify every single detail. Doing so often takes the fun out of writing software anyway. Use the innate intelligence and creativity of your engineering team, either in-house or offshore, to your advantage. The ability to create new software applications is a critical criterion for hiring employ-

ees and it’s the same for selecting an offshore outsourcing vendor.

Be More Agile

It is rare to see software accurately specified the first time. It nearly always changes because new ideas arise as the software takes shape. Else, customers and users may change their minds about what the software should do.

And yet many companies treat offshore outsourcing strictly as a business decision. They give a general description of what the software should do and then ask for the lowest possible price. If you are developing a new software application, this can be a recipe for disaster.

An alternative is to spend months debating what your new software application should do and what should be in the specification. In any case, you are better off selecting an offshore vendor with experience developing new applications using agile software development methods and with whom you can collaborate.

Agile software development methods and user stories are recommended for creating new software applications because they embrace change. Agile methods get your offshore vendor developing software as soon as possible. Frequent releases—every three to four weeks—provide you with steadily increasing amounts of high-quality software. An amusing way to think of it is that agile methods let your programming team do something useful while you figure out what the software is supposed to do.

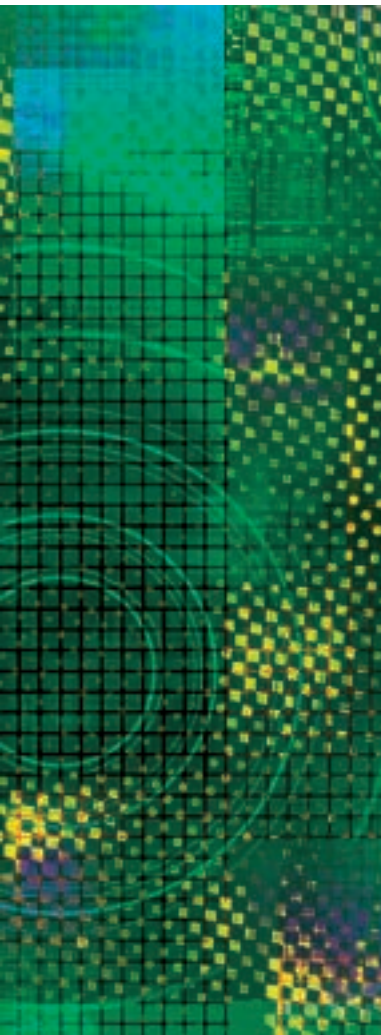
Your Outsourcing Guarantee

In the beginning of this article, I wrote that you are unlikely to find any guarantees in outsourcing your software development. Hopefully, you can now see that it is nonetheless possible to significantly reduce the risk involved. You cannot guarantee that your original software idea (which may be inaccurate or incomplete) will be implemented in a fixed period of time. However, you can be assured that you will get a predictable amount of software in a predictable amount of time, and this type of process is likely to deliver a product that your customers will actually want to buy.

The point of a guarantee is not to get something for nothing; rather, it is to remove risk from the development process. Whether you develop your software with offshore outsourcing or your own development team, you can use the principles of agile software development to ensure reliability in your own results. □

Capturing Business Rules

Business rules provide the knowledge behind every business structure or process. They are, therefore, at the core of functional requirements.



Ellen Gottesdiener is principal consultant and founder of EBG Consulting, Inc. Ellen has authored two books and numerous articles and has contributed chapters to several books. She can be contacted at ellen@ebgconsulting.com.

AN INEPT, INADEQUATE or inefficient requirements analysis starts a project on the wrong foot. Time is wasted at a point in development when time is of the essence, and developers will find it hard to produce a good software product based on ill-defined requirements. Capturing, validating, and verifying functional requirements are major challenges, not only for clients and business partners, but also for managers and software developers. We need clear and usable requirements that can guide the development of a quality software product, and a poor requirements process will not lead to a good requirements product.

The requirements process includes what techniques are used to capture the requirements (interviews, facilitated workshops, prototyping, focus groups, or the like), what tools are used for capturing and tracing information (text, diagrams, narratives, or formal models), and how customers and users are actively involved throughout the process. When it comes to the requirements product, management concerns include the testability of functional requirements, the ability to find and resolve conflicts in requirements, the ability to link requirements back to business goals (backwards traceability), and the ability to track functional requirements throughout the lifecycle from design to code to test and deployment (forward traceability).

As mature managers, we no longer expect silver bullets for requirements engineering. However, in my experience there is at least one “secret weapon.” Whatever functional models you wish to use, whether use cases, CRC cards, or some proprietary technique championed by your boss, the important thing is to focus on the true essence of the functional requirements: the business rules.

Rules Rule

Business rules are the policies and constraints of the business, whether the “business” is banking, software development, or automation engineering. The Object Management Group simply defines them as “declarations of policy or conditions that must be satisfied.” They are usually expressed in ordinary, natural language and are “owned” by the business. Your business may be commercial, not-for-profit, or part of a government, but it still has business rules. They provide the knowledge behind any and every

business structure or process. Business rules are also, therefore, at the core of functional requirements. You may use various functional requirements models—structural (data, class); control-oriented (events, state-charts); object-oriented (classes/objects, object interactions); or process-oriented (functional decomposition, process threads, use cases, or the like), but within the heart of all functional requirements are business rules.

Business rules are what a functional requirement “knows”—the decisions, guidelines, and controls that are behind that functionality. For example, functional requirements or models that include processes such as “determine product” or “offer discount” imply performance of actions, but they also embody knowledge—the underlying business rules—needed to perform those actions. An insurance claim in a “pending” stage means that certain things (variables, links to other things) must be true (invariants, or business rules). The behavior of a claim in a pending lifecycle stage is, thus, dependent upon business rules that govern and guide behaviors.

As the essential ingredient of functional requirements, business rules deserve direct, explicit attention. Since business rules lurk behind functional requirements, they are easily missed and may not be captured explicitly. Without explicit guidance, software developers will simply make whatever assumptions are needed to write the code, thus building their assumptions about conditions, policies, and constraints into the software with little regard for business consequences. Rules that are not explicit and are not encoded in software through the guesses of developers may not be discovered as missing or wrong until later phases. This results in defects in those later phases that could have been avoided if the rules had been elicited, validated, and baselined during requirements analysis. In the end, the lack of explicit focus on capturing the business rules creates rework and other inefficiencies.

Rather than just mentioning business rules as “notes” in your models, you should capture and trace them as requirements in and of themselves. Focusing on business rules as the core functional requirements not only promotes validation and verification, but it can speed the requirements analysis process.

Top-Down, Again

In working on a business-rule approach to software development, I have come to realize that such an approach needs to be driven from the top down, like the traditional top-down methods of information engineering. Unfortunately, in the modern world, with business moving at the speed of a mouse click, a systematic, top-down analysis is often an unaffordable luxury. Consequently, I've become more practical; a business-rules approach doesn't need to be a method or methodology in and of itself. Rather, it is just a necessary part of requirements engineering. It includes a process (with phases, stages, tasks, roles, techniques, etc.), a business rule meta-model, and business rule templates.

Advocates of the recently standardized Unified Modeling Language (UML) and the accompanying all-purpose "Unified Process" toss around the term "business rules" in presentations and conversations, but neither UML nor UP offers much guidance for business rules. UML has an elaborate meta-model and meta-meta-model to support its language. One of the classes at the meta-model level is called "Rules." But the UML has given business rules short shrift. The only modeling element that is practically usable is the "constraint" element, which can be attached to any modeling element. Furthermore, although the UML's OCL (Object Constraint Language) is a fine specification-level language to attach business rules to structural models, it is not a language to use during requirements analysis when working directly with business customers to elicit and validate business rules. Business rules need to be treated as first-class citizens (no pun intended).

Use Cases and Business Rules

At the center of the UML are use cases, which are viewed by some analysts as business requirement models, while others call them "user-requirement models." Use cases are a functional (process) model that can be expressed as words using a natural language; some templates also may include special sections, such as pre- and post-conditions, goal name, and the like. Use cases have proven to be an important and useful model for capturing requirements. Recent work has evolved use cases from an often vague requirement deliverable into something specific, focused, and usable. Alistar Cockburn has directed attention to the goals of use cases, and Larry Constantine and Lucy Lockwood have developed essential use cases for usage-centered design. However, use case narratives—the flow of events, as the UML people would say—are all too often missing

the very essence of the use case, because behind every use case are business rules at work.

Failing to capture and verify the business rules that underlie the use-case models can lead to the delivery of a software product that fails to meet the business needs. Formalizing business-rule capture concurrently with use-case model development strengthens the delivered product through the synergy between use-case models and business rules.

Business rules come in many forms. I think of them as terms, facts, factor clauses, and action clauses, but there is no agreement on a standard taxonomy or set of categories for business rules nor should there be. The taxonomy should fit the problem. Some things in the "problem space" are more business-rule-based (e.g. underwriting, claim adjudication, financial-risk analysis, medical instruments monitoring). Other problems are more business-rule-constrained (payroll, expense tracking, ordering, etc.). This requires the selection or tailoring of a taxonomy and an accompanying business-rule template for any given business problem. The template provides a standard syntax for writing business rules in natural language. Such tailoring is beneficial because it requires us to understand the problem in greater depth by working directly with our business customers to perform this tailoring and to derive an appropriate business rule template.

Business rules can be linked to a use case or to steps within a use case. Business rules also can be linked to other models, depending on the depth of requirements traceability and the importance of specific attributes, such as source, verification process, documents, owner or risk. These other models can include glossaries, lifecycle models, activity models and class models. The business rules are thus reusable across multiple types of functional requirements and can be traced along with other requirements.

Besides use cases, other models can be useful for identifying and capturing business rules. For example, lifecycle models, such as a simple state-chart diagram, can be quite usable and understandable to business analysts, especially for problems in which a core business domain has many complex states. Even object-class or data models can be excellent starting points for problems that require understanding the domain structure, but which do not require a lot of "action" to take place.

Collaborate with Customers

Business-rule discovery and validation requires knowledge of the thinking processes of the business experts from whom we elicit functional requirements.

Collaborative work in groups that include customers is most effective in the early lifecycle phases of planning and requirements analysis, as well as for ongoing project process improvement. Such collaborative work patterns can be used effectively to model business rules and derive higher quality requirements, which include business-rule requirements.

In the collaborative approach to business rules that I prefer, requirements analysts and business customers collaborate to create a business-rule template that is expressed in natural language, based on common sense and directly relevant to the business customer. In modeling use cases and user interfaces, business rules are explicitly derived and captured using this natural language template. After all, what makes sense to our customers and users is what we express in their own language.

Collaborative modeling of business rules can be a powerful, eye-opening process. It is amazing to see customers realize that their own business rules are unclear, even to them. Often they come to realize the rules are not standardized and therefore may be inefficient or even risky. Considering regulatory exposure and potential for lawsuits, collaborative modeling can convincingly make the case for immediate clarification of business rules.

Eliciting business rules can be a real challenge, especially when business experts do not agree on the business rules or when the business rules are unknown or very complex. I find that facilitated requirement workshops in which business rules are explicitly discovered, capture and tested within the context of modeling other things, such as use cases, is the most direct and productive tool for eliciting and validating the rules of the business. These workshops are planned collaborative events in which participants deliver products in a short, concentrated period of time led by a neutral facilitator, whose role is to help the group manage the process. Prior to the workshops itself, the participants have agreed upon what will be delivered and the ground rules for group behavior. The workshop process exploits the power of diverse people joined together for a common goal.

A successful requirements workshop requires considerable pre-workshop effort. Such collaborative events, when woven throughout requirements analysis, tend to increase speed, promote mutual learning, and enhance the quality of the requirements themselves. In workshops, or even in interviews with business experts, if the problem domain is very much 'rules based' (vs. rule constrained as mentioned earlier), using cognitive patterns is extremely helpful to accelerate the group's ability to express the rules. Such

patterns, with their roots in knowledge engineering, model business expert's thinking process and enable the rules to emerge.

Some customers may wish to take on a business-rules approach because they want to uncover the real "dirt" of the business. They are ready to ask the "why" of the business rules. Actually, the question of larger purpose should be asked of any functional requirement. If the answer does not map to a business goal, objective or tactic, then the functional requirement is extraneous. Business rules exist only to support the goals of the business. Whereas good use cases represent the goals of external users or "actors," the business rules behind use cases are inextricably linked to the goals of the business itself. If not, the rules are extraneous and may even be in conflict with business goals. Thus, mapping business rules to business goals is a key step in validation and promotes traceability back from business-rule requirements to the business goals.

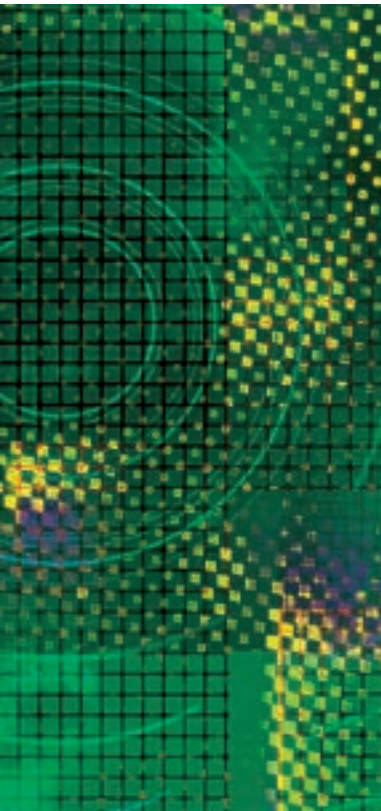
The Business-Rule Cure

Business rules are an ounce of prevention. Unless we get them, get them right and get them early, we are destined for problems in our projects and products. The project problems stemming from incomplete, erroneous or missing business rules include redefining requirements and re-testing results. The product problems are worse because, if the rules are wrong, in conflict or redundant, the users of the software product suffer. Unless we get to the very heart of the business/requirements analysis, with the business rule written in text by business people themselves, we are doomed to passing incomplete, inconsistent and conflicting business-goal requirements ahead to production. Thus, we must get to the very essence of requirements with business rules, written by business people.

To get to the heart of the matter, active business sponsorship is absolutely required. The process can be acutely uncomfortable because business-rule capture and validation exposes the "undiscussables"—the unclear and conflicting business policies and rules. It also begs for rethinking the sub-optimal rules and requires the realignment of the business rules with the business goals. To resolve such issues requires business sponsorship and leadership. To go forward with a business-rules approach in the absence of such sponsorship is treading on very thin ice. Only the collaborative efforts of IT professionals and their business partners with the active involvement of business management can yield the benefits of a business-rules approach in quickly and succinctly cutting to the core of the functional requirements. □

An Introduction to the Business Analysis Body of Knowledge

Eliciting, analyzing, communicating, and validating requirements



Kevin Brennan is the Vice-President, Body of Knowledge of the International Institute of Business Analysis. He can be reached through the IIBA website at www.theiiba.org.

REQUIREMENTS DEFINITION PROBLEMS are so well known that they hardly seem worth repeating. It sometimes seems like every other article on requirements analysis begins with the same grim set of statistics on project failure, reminding us that almost all projects run over time and budget and that a frightening percentage of these are eventually cancelled. If we were to sit down and calculate the ROI generated by most IT divisions, we may have a difficult time justifying our own existence. More and more, we see that the source of these project difficulties is related to the requirements. In particular, it seems that we don't know what we're going to build when we start, and the definition of what we're supposed to build keeps changing. So we end up with a system that falls well short of what was originally promised.

One reason we keep running into this problem is that many organizations place the responsibility for defining requirements on people with little to no prior experience. When we launch a new project, we pull a person out of an end-user group and assign him or her to work on an IT project as a full-time stakeholder. Eventually, the job they used to hold is filled by somebody else and they remain in IT, assigned the job title of "business analyst."

The problem our profession faces is that those people are rarely given the training or education necessary to be effective in developing requirements. Furthermore, business analysts have no way to know whether their own organization's processes reflect those of other companies, and this makes it difficult to figure out which additional skills they need to be most effective. Hence, over the years, they grow more distant from the work done by the

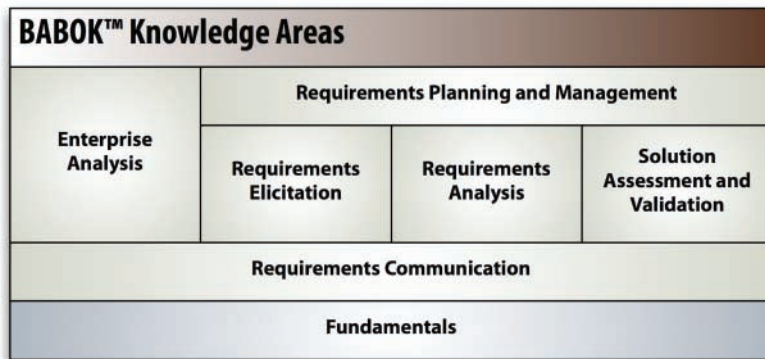
business but fail to replace that knowledge with the skills that would allow them to be effective in their new role.

That's why the IIBA™ was founded in 2004. We hope to professionalize the business analysis community by developing two things: A body of knowledge that describes a standard set of competencies for business analysts (BAs), and a certification program that enables BAs to demonstrate their understanding of that body of knowledge. I've been working on the first of those, the Guide to the Business Analysis Body of Knowledge™ (BABOK™) since 2004, and leading that effort since the spring of 2006.

What's a Business Analyst?

The very first question we had to answer was "what is a business analyst?" Every company defines the job a little differently. Some business analysts write queries and ad-hoc reports. Others are responsible for testing and QA, and still others lead project teams or act as junior project managers. Some companies define several jobs for BAs, including system analyst, business systems analyst, business process analyst, data analyst, and the list goes on and on.

In order to develop an appropriate concept of what a BA does and should do, we looked at the work currently done by BAs and tried to determine which needs were not being met. There's already a great deal of professional support for software developers, testers, and project managers. We had to focus on the part of the BA role that was unique: the development of requirements in the broader context of solutions for businesses. So, finally, we settled on the following definition for the business analyst profession.



© 2007, International Institute of Business Analysis

Figure 1

A business analyst works as a liaison among stakeholders in order to elicit, analyze, communicate, and validate requirements for changes to business processes, policies, and information systems. The business analyst understands business problems and opportunities in the context of the requirements and recommends solutions that enable the organization to achieve its goals.

It's a complex description, but it was intended to highlight certain key aspects of the business analysis role:

- Business analysis is about developing requirements for the solution of business problems. Not all requirements analysis is business analysis—we don't cover the creation of products for commercial sale, for instance, or for the creation of hardware. A lot of what we have to say about requirements development applies in other fields, to be sure, but we're focused on what it means in the context of developing solutions for organizational needs.
- Requirements should be viewed in a broader strategic context. Requirements and the solutions they define are not an end in themselves; they have to support the overall goals of the business. The ability to tie requirements to strategic goals is one of the most important skills BAs can contribute to their employers.
- Business analysis is not necessarily an information technology role. While most BAs work with IT in some form, they can also be found in business process management, business intelligence, and other disciplines.
- Business analysts may fill different roles in different organizations. Some view the BA as a consultant, who plays an active role in helping the business reach strategic objectives. Others view the BA as a facilitator who helps the stakeholders in an organization properly articulate their needs.

The Business Analyst Role

The BABOK is not intended to define a new way of performing projects or to raise the bar on how we do requirements analysis. Rather, it's to set a baseline and raise the average level of practice. When it comes to requirements, we seem to be trying to learn the same lessons over and over. By identifying and describing a set of common standards through the BABOK, the IIBA hopes to eliminate any need for that.

The structure of the BABOK parallels many software development lifecycles, and this is simply because we felt that this approach would make it easiest to understand. While real projects rarely progress in a straightforward fashion, this presents an ideal structure that people can easily understand and relate to.

The only downside is that readers frequently view this as the recommended and expected approach to projects. Most books on requirements begin with the (perhaps implicit) premise that the analyst is involved in the creation of an application unlike anything the organization has used before. In reality, those kinds of projects are rare. We've been working hard to make it clear that the content of the BABOK applies to other lifecycles where the business analyst role is found, including maintenance efforts, agile development, business process or data management, business intelligence, and other organizational change initiatives. Whether or not there's a person with the formal job title of "business analyst" on those projects, all of them require the effective use of business analysis to identify and execute a solution that will have a positive return on investment.

Structure of the BABOK

As Figure 1 illustrates, the BABOK is divided into seven knowledge areas:

- **Enterprise Analysis**, which discusses how initiatives fit into the larger context of the organization. This covers the business and application architecture, development of business cases and feasibility studies, as well as strategies for identifying overall business requirements.
- **Requirements Planning and Management** covers the organization of a business analysis effort. We discuss how to identify stakeholders, estimate the work involved, prioritize requirements for one release or several, and how to manage changes to the requirements.
- **Requirements Elicitation** discusses how requirements are gathered from stakeholders. It covers techniques such as brainstorming, surveys, interviews, and workshops.

- **Requirements Analysis** discusses how business analysts produce finalized requirements from the information provided by stakeholders. It shows how requirements are structured, how they are specified, and how the analyst ensures that they are accurate.
- **Requirements Communication** describes how business analysts make sure that all stakeholders have a shared understanding of the requirements. It describes techniques including peer reviews, presentations, and formal sign-off and approval, as well as some common requirements documentation formats.
- **Solution Assessment and Validation** covers the methods used by a business analyst to make certain that a proposed solution meets the agreed-to requirements. It also discusses what to do when gaps are identified.
- **Fundamentals** is the final knowledge area. This covers the “soft” skills that, though they are not unique to business analysis, nonetheless play a major role in a BAs success. It covers consulting skills, communication, general management and business knowledge, and related topics.

Each knowledge area is divided into tasks (which represent the core work done in that knowledge area across all lifecycles and/or disciplines) and techniques (which describe the different ways of accomplishing a task, depending on the methodology); see Figure 2.

The content of each knowledge area represents the “generally accepted” tasks and techniques used throughout the industry, as best as we can determine. But this is not to say that this is how we think you should perform business analysis; rather, it’s a summary of how people are performing it. In order to determine what’s generally accepted, we looked at how many people are using a particular technique, the availability of instructional materials

related to it, and other resources that are available to support it. It doesn’t mean these techniques are universally accepted, but certainly a significant percentage of BAs should have actual experience with a technique before it’s considered for inclusion in the BABOK.

What’s Next?

Right now, Version 1.6 of the BABOK is available for download from www.theiiba.org, and this version is used as the basis for our certification exam. This summer, we’re going to release Version 2.0, the first complete revision of the draft content we’ve created. Version 2.0 has been restructured to be much more adaptable to different lifecycles and approaches. We’ve clarified how to perform business analysis in agile and iterative lifecycles, how to create requirements for the maintenance of existing systems, and how business analysis is performed when the goal is something other than the creation of a software application.

In Version 2.0, each knowledge area will have a smaller set of core tasks that are applicable to the vast majority of projects and initiatives, but much of the content will be placed into a menu of techniques that exhibit different methods for executing the tasks. Our goal is to make it easy to adapt the content of the BABOK to different methodologies and to new technologies as they emerge. For instance, we don’t have a lot of material on how to use wikis for requirements development because, so far, people don’t have a enough experience with this to form a general consensus. In a few years, though, it’ll probably be necessary to add a section on that to the BABOK.

After the release of Version 2.0, we will organize a review by members of the business analysis community as well as experts in the various disciplines in which business analysis plays a role. Once we’ve finished incorporating the results of those reviews into the BABOK, and we’re comfortable that the content is as good as we can make it, the result will be turned over to a group of volunteers responsible for editing, integration, and improving its presentation and readability. After all of this, the revised document will be released as BABOK Version 3.0. It certainly won’t be the “final” edition, as there could never be a truly final edition. But Version 3.0 is expected to be a “stable” release that will remain in circulation for at least two years before the IIBA considers an update.

For more information on the BABOK, please visit the IIBA’s website at www.theiiba.org to download both Version 1.6 and information on the changes planned for Version 2.0 of the BABOK. □

	Agile/Lean	BPM	Business Rules	Business Intelligence	CMMI/Waterfall	Data Mgt.
Business Analysis						
Development						
Quality Assurance						
Project Management						
Sales						
Training						
etc..						

© 2007, International Institute of Business Analysis

Figure 2