Listening to the Customer's Voice[1]

Karl E. Wiegers

Process Impact
www.processimpact.com

Perhaps the greatest challenge facing the software developer is sharing the vision of the final product with the customer. All stakeholders in a project—developers, end users, software managers, customer managers—must achieve a common understanding of what the product will be and do, or someone will be surprised when it is delivered. Surprises in software are almost never good news. Therefore, we need ways to accurately capture, interpret, and represent the voice of the customer when specifying the requirements for a software product.

Often the customer will present as "needs" some combination of: the problems she has in her work that she expects the system to solve; the solutions she has in mind for an expressed or implied problem; the desired attributes of whatever solution ultimately is provided; and the true fundamental needs, that is, the functions the system must let her perform. The problem becomes more complex if the systems analyst is dealing with a surrogate customer, such as a marketing representative, who purports to speak for the actual end users of the application. The challenge to the analyst is to distinguish among these four types of input and identify the *real* functional requirements that will satisfy the *real* user's *real* business needs.

Many techniques are used for eliciting user requirements, all of which attempt to include the voice of the customer in the product design process. A typical project might employ a combination of meetings with user representatives and developers, facilitated workshops (for example, joint application design sessions) with analysts and users, individual customer interviews, and user surveys. The use case approach is an especially effective technique for deriving software requirements, analysis models, and test cases.

The Use Case Method

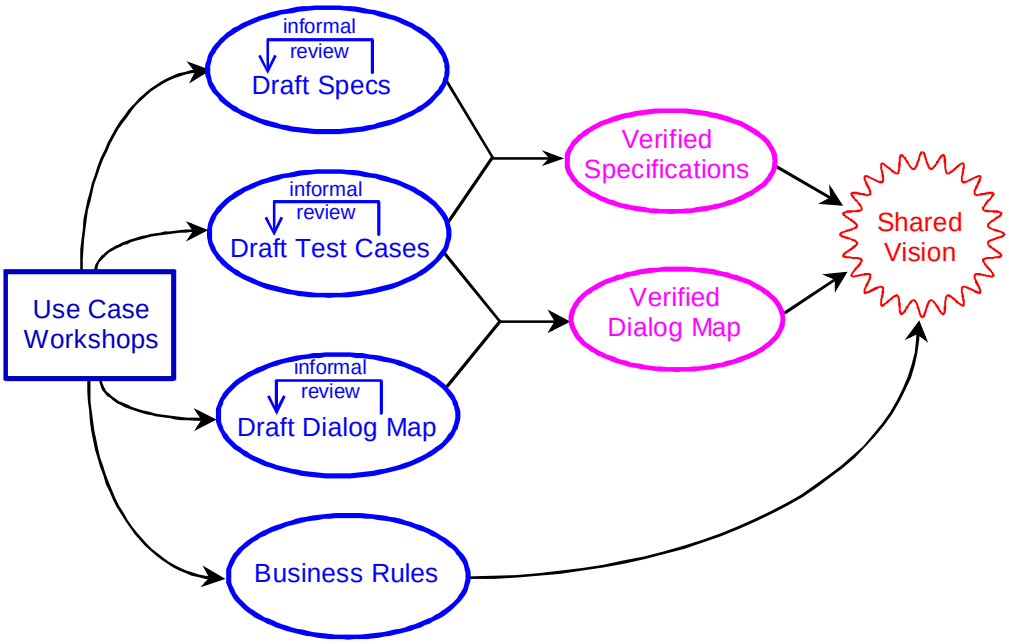
Use (pronounced "youce," not "youze") cases were introduced as part of an object-oriented development methodology by Ivar Jacobson in *Object-Oriented Software Engineering: A Use Case Driven Approach* (Addison-Wesley, 1992). More recently, Larry Constantine and others have extended the concept into a general technique for requirements analysis and user interface design. Each use case describes a scenario in which a user interacts with the system being defined to achieve a specific goal or accomplish a particular task. Use cases are described in terms of the user's work terminology, not computerese. By focusing on essential use cases, stripped of implementation constraints or alternatives, the analyst can derive software requirements that will enable the user to achieve her objectives in each specific usage scenario.

A software team at Eastman Kodak Company recently applied the use case method to specify the requirements for a chemical tracking system. Figure 1 illustrates the overall process we found to be effective with the use case technique and the key deliverables. We used a series of highly interactive sessions with customer representatives to identify and describe use cases. Functional requirements, behavioral characteristics, and business rules fell right out of the use case discussions.

---

[1] This paper was originally published in *Software Development*, March 1997. It is reprinted with permission from *Software Development* magazine.

Figure 1. Process and deliverables from the use case method.



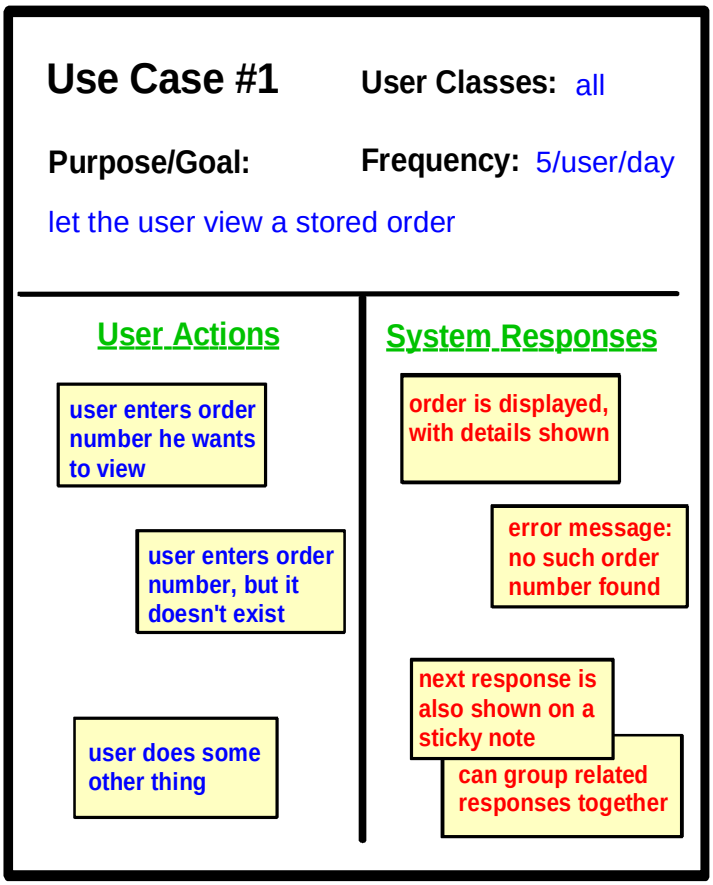
I worked as an analyst with a team of chemists to collect their needs for this application; other analysts worked with other teams representing different user classes. Members of different user classes might have the need to use different subsets of the system's functionality, different vocabularies, different security requirements, and different frequencies of use of various features. Each user class generated its own set of use cases, since each needed to use the system to perform different tasks.

The chemist team consisted of one designated key customer representative (project champion) and five other chemists whose needs differed slightly. Customers having multiple viewpoints and needs are valuable, to avoid designing a system that is excessively tailored to meet a narrow subset of the needs of the broader user base.

I began by asking the chemists to think of reasons they would need to use this planned chemical tracking system, or tasks they needed to accomplish with it. Each of these "reasons to use" became a use case, which we then explored in depth in a series of two-hour sessions that I facilitated. We used forms such as that shown in Fig. 2, drawn on a large flipchart. I prepared one flipchart for each use case prior to the group sessions.

For each use case, we stated the goal that the user needed to accomplish—one reason someone would use this application. We also noted which of the various user classes we had identified for this application (chemist, technician, stockroom staff, purchasing department) would be interested in each use case. Estimating the anticipated frequency of execution for each use case gave us a preliminary indication of concurrent usage loads, the importance of ease of learning versus ease of use, and capacities of data storage or transaction throughput. We could also

Figure 2. Use case flipchart created in a workshop with user representatives.



identify the relative priority of implementing each use case at this stage. The sequence in which users identify candidate use cases suggests an approximate implementation priority.

We spent the bulk of each workshop exploring the actions the user would expect to take with the computer for a specific use case, and the possible responses the system might then generate. As the chemists suggested user actions (or inputs) and system responses, I wrote them on sticky notes and placed them under the appropriate heading on the flipchart form. Often, a system response would trigger further dialog with the computer, necessitating additional user input, followed by yet another response. The movable sticky notes made it easy to revise our initial thoughts and to group together pieces of related information as we went along.

By walking through individual use cases in the workshops, we drilled down to the fundamental customer needs the system had to satisfy. We also explored many "what if" scenarios to reveal exception and decision situations the system must handle. New use cases sometimes came to mind as we worked through those that the chemists had already suggested.

Some of the pieces of information elicited in the workshops were not really functional requirements. Instead, they were business rules about the system and the work processes it supported, stating such policies as: "Safety training records must be up to date before a user can order a hazardous chemical" and "A purchasing agent can modify a user's order only with written permission from the user." We documented these business rules separately from the functional requirements, and then communicated them to the project champions of the other user classes, who had their own, sometimes conflicting, business rules. Discrepancies were reconciled in a meeting between the analyst team and all of the project champions.

We found that the use case strategy of asking "What do you need to accomplish with this system?" kept us focused on visualizing how the application ought to perform some function. More open-ended discussions that begin with "What do you want this system to do?" can lead participants to suggest features that are not necessarily linked to specific user tasks. The distinction between these two questions is subtle, but important. The use case angle emphasizes user objectives, not system features. Use cases provide a way to decide what features are necessary, as opposed to building in specific features and hoping they let the users get their work done.

After each workshop, I took the flipcharts festooned with sticky notes to my office and began to extract requirements from each use case. Most of these requirements represented those software functions that would be needed to allow a user to execute each use case. I began documenting the requirements and business rules in a structured software requirements specification, or SRS, which grew from week to week.

Within two days after each workshop, I delivered the preliminary SRS to the chemist team members. They reviewed the SRS changes informally on their own, in preparation for the next weekly workshop, at which we discussed problems they found during their review. The next iteration of the SRS incorporated any necessary corrections, as well as new requirements extracted from the latest use case discussion. By making multiple passes through the growing SRS in this way, we created a much higher quality product than if we had waited until the complete SRS was done to begin the review process.

For certain complex use cases, we also drew dialog maps, or models representing a possible user interface at a high level of abstraction. A dialog map is a form of state-transition diagram, in which each dialog element—screen, window, or prompt—is shown as a state (a labeled rectangle), and each allowable navigation pathway from one dialog element to another is shown as a transition (a labeled line). We used dialog maps to represent the possible sequences of human-computer interaction in several scenarios related to a specific use case, without worrying about details of screen layout or interaction techniques.

After all the information was translated into structured textual requirements specifications, business rules, and structured analysis models, we were able to take another important step: deriving test cases.

Deriving Test Cases from Use Cases

Writing functional test cases is a great way to crystallize the fuzzy image in your mind about how a program should behave. Deriving test cases from our use cases proved to be easy, since we had already been thinking about how the user would interact with the program to accomplish various tasks. Each test case represented one specific scenario of interaction with the system, within the context of one of the essential use cases. The test case might result in a successful transaction, or it could generate an anticipated error response of some kind.

Using the requirements specification to provide details, we developed test cases without the chemists ever having sketched out screens or discussed the exact database contents. Our general concept of the sequence of events that characterized a use case provided enough detail to permit writing many test cases. If you can't readily think of test cases that demonstrate your understanding of a use case scenario, you haven't adequately clarified the vision.

Having test cases in hand this early in the development process provided several benefits:

- We could "test" parts of the specification long before it was complete by having the chemist group walk through the test cases to find any changes that we needed to make in our specifications so that the tests could be executed.
- We could use the test cases to "test" the dialog map. As we walked through the conceptual test cases derived from the SRS, I traced the corresponding paths on the dialog map with a highlighter pen. This revealed allowable execution paths we had missed in the test suite, as well as identifying test cases that could not be "executed" according to the current dialog map, indicating errors in either the dialog map or the test cases.
- We could establish traceability between each test case and the requirements it addressed. Any requirements that were not covered meant writing more test cases; any test cases that relied on functions not present in the requirements document showed that some requirements were missing.
- We could use the suite of test cases to evaluate commercial products that might meet the needs of the project, enabling us to decide whether to build or buy. We simply had to walk through the test cases with an evaluation copy of the commercial package to see which cases it handled and which it did not. The test cases that mapped to top priority use cases were the ones to focus on.

Developing test cases early in the project reinforced our fundamental philosophy that quality activities must not be left to the end of the development cycle, but should permeate the entire project from the outset. (For a related example, see "Reduce Development Costs with Use-Case Scenario Testing," by Scott Ambler, in *Software Development*, July 1995.) This use case approach with immediate test case generation reinforced another valuable lesson. Every time I do any kind of systematic testing, I find errors in the work products (in this case, in the requirements specification document and the dialog map), and I find errors in the test cases (such as requirements that were not covered by the test cases).

From Use Case to Construction

Once you have uncovered most of the use cases and assigned priorities to them, you can devise an implementation strategy. One approach is to use the concept of "threads" to design and implement use cases incrementally, building new classes or modules as needed to support the functionality associated with new use cases. A thread is a grouping of modules or classes that implement a specific set of related requirements, and use cases provide a natural mapping of requirements to threads. The concept of threads is nicely explained by Michael Deutsch and Ronald Willis in *Software Quality Engineering: A Total Technical and Management Approach* (Prentice-Hall, 1988).

A thorough exploration of use cases before doing any serious construction will help you avoid building a fragile, incoherent architecture, with new classes or functions being slapped together willy-nilly. Select a construction sequence of threads to implement critical use cases and supporting functionality first, but design a sufficiently robust and extensible architecture to permit incremental assembly of additional use cases over time. This approach will deliver the most essential functionality to your customers as early as possible.

Use Case Technique Evaluated

The use case approach is an efficient and effective technique for collecting essential requirements from a group of customers, helping to focus on their real needs, not just what they initially say they want. It will help all those involved—analysts and customers—arrive at a common, shared vision of what the product they are specifying will be and do. This is key to constructing quality software.

Two less obvious benefits are derived from the use case method. First, the act of developing use cases helps analysts gain an understanding of the user's application domain. Second, it helps avoid superfluous requirements. When approached with the question, "What do you want the system to do?" customers are tempted to add features that seem like good ideas and might be useful some day. From the use case angle, though, every requirement directly supports a specific task the user needs to accomplish with the help of the computer.

It may be a little confusing when people start thinking in terms of use cases. Suggested use cases may still really be lists of features, or desired behaviors, or attributes. However, you should be able to map each of these characteristics to one or more existing use cases. If you cannot, perhaps the features or behaviors described are unnecessary. A more likely possibility is that some use cases were missed initially, so these features, behaviors, and attributes can help illuminate additional use cases.

Use cases constitute a powerful, user-centric tool for the software requirements specification process. They are applicable whether the product will be constructed using object-oriented techniques or in a more traditional environment. The perspective provided by use cases reinforces the ultimate goal of software engineering: to create products that let customers do useful work.