

[▶ subscribe](#)[▶ contact us](#)[▶ submit an article](#)[▶ rational.com](#)[▶ issue contents](#)[▶ archives](#)[▶ mission statement](#)[▶ editorial staff](#)

## Requirements Archaeology

by [Benjamin A. Lieberman, Ph.D.](#)

Senior Software Architect  
Trip Network, Inc.

### "You call this Archaeology?"

- Dr. Henry Jones, Sr.  
from *Indiana Jones and the Last Crusade*,  
Lucas Films, Ltd. (1989)

*System requirements are at the heart of every meaningful software system. They not only explain the behavior of the system, but also provide context into the minds of the software users who will ultimately benefit. Often times, however, these requirements are either a) partially captured in an informal or locally developed mechanism, or b) captured in the heads of the developers and the mystery that is code. Unfortunately, for most software development organizations, both approaches lead to mistakes and added costs.*



## Costly Legacies

Many, if not most, companies have a significant investment in one or more legacy software systems that were developed over a number of years and are critical to the continued survival of the company. However, as most of these systems have missing, incomplete, or inaccurate requirement sets, there is a very high likelihood that they are fragile and expensive to maintain. Moreover, because the behavior of the software is not well defined, it is almost impossible to replace the system with a modern equivalent. This leads to systems that are "dated" and too inflexible to meet modern business demands.

Adding new functionality or behavior to the existing system typically complicates the issue further, as the effect on the existing system's stability is unpredictable. This is very much like adding a second story to a house with an unknown foundation: There is no documentation about how the system was built, and therefore no way to know if the structure will hold until it is too late to do anything about it. If you want to replace, re-

develop or continue to maintain a legacy system, you need a more complete understanding of the original driving forces of the system: the behavior, states, data elements, and business rules. That requires discovery of the system's business requirements.

The more obvious sources for this information -- those who developed the system -- probably have left the company by now. But the information also lies buried within the single reliable artifact remaining to the investigator: The Code. In other words, there is a way to rediscover "lost" requirement "artifacts." The discovery process is akin to an archaeological expedition: You must determine areas of investigation, interview and study information sources, carefully catalog the available artifacts, interpret cryptic comments, and, finally, present the software's "story" to system stakeholders.

All of this, of course, comes at a cost. Any "expedition" incurs significant risk and expense. Depending on the size of the system to be captured and the current state of the requirements base, this effort may require as many as three to four people working full time for several months. The expense might seem justifiable only when you consider the cost of not having a stable and complete requirements base. Without well-understood requirements, businesses can lose opportunities and experience increases in maintenance expenses, difficulties in integrating systems to realize economies of scale, and decreased employee job satisfaction (an expert who understands part of the system is unlikely to be allowed to work on anything else), and significant decreases in their strategic planning capability. Only by reestablishing a solid understanding of "what is" can an organization prepare to realize "what can be." In other words, although a company can survive without a solid requirements base, it will have a very limited ability to grow and thrive.

In this article I will discuss the process by which a legacy system's requirements can be captured and used to rebuild a complete requirement base. I will present some techniques for "excavating" requirements from existing artifacts and other information sources. Finally, I will discuss how to present the newly discovered artifacts so as to improve the probability of long-term maintenance.

If you have a stout heart and keen mind, read on. For mystery and adventure await the intrepid requirements explorer.

## **Preparation**

The prudent Requirements Archaeologist recognizes the need to acquire a few skills and supplies before embarking on an expedition into the Software Wilderness, including:

1. A firm grounding in the theory and practice of both Structured Requirements<sup>1</sup> and use-case-based requirements analysis.
2. Experience with researching and cataloging information.
3. Experience leading small teams of engineer-investigators.

4. Practice with presenting to, and facilitating for, small groups of business subject matter experts.
5. Strong familiarity with UML modeling (at a minimum, use-case and activity diagrams).
6. Experience with the system programming language (optional, but it's always useful to speak the language of the "locals").

It is also very helpful to have the right equipment at your disposal when facing the unknown. So it is a good idea to acquire the following tools before setting out:

1. A requirements management tool (e.g., Rational®RequisitePro®<sup>2</sup>), to hold and organize your findings.
2. A Unified Modeling Language (UML) diagramming tool (e.g., Rational Rose®<sup>3</sup>) for use cases, activity diagrams, and state charts to express the system information graphically.
3. A good code editor/profiler for the particular system language (e.g., TextPad<sup>4</sup>) to review the code artifacts.

Once you have collected your equipment and honed your skills, you will be ready to head out on your adventure.

## **Site Investigation**

Surprisingly enough, requirements that require discovery are seldom left just lying about waiting to be picked up and used. Typically, they are deeply buried in guarded legacy systems, mysterious legacy documentation, and even "legacy personnel." I have found that there are four primary sources of information about an existing software system:

- People
- Documentation
- Software Code
- Software Issue Tracking Reports

Each of these sources requires different techniques to sift through and recover meaningful descriptions of the software system. If a company has a defined requirements management team, then this is the best place to start investigations. Typically, these people will be referred to as "Business Analysts" or "Requirements Engineers," but they may also be members of product or project management, marketing, or even sales. In fact, anyone in the organization who has a hand in determining the functionality and performance of the system will serve as a good source of information to discover and document the driving forces that created the original need for the software system. (Within the Rational Unified Process® (RUP®), the needs of a well-architected system are first identified in "Stakeholder Requests"<sup>5</sup>.)

Written documentation is a useful, but often misleading, source of information. This is because most documentation is not maintained and so tends to become obsolete rather quickly. In addition, this resource is typically the most difficult to locate, because a company that doesn't organize its requirements is unlikely to organize its documentation; usually, critical system documents exist only on local hard drives or difficult-to-access shared drives. Once collected, however, these historical records can provide a good starting point for discussions and further investigation.

The most accurate source of information on the system's behavior is, of course, the software code itself. But it takes special skills to make use of this resource. In particular, as I mentioned earlier, the investigator should be familiar with the specific programming language(s) used by the development team. This can range from assembly language used for mainframes, to C++, Java, and VisualBasic™ used for PC applications, to more specialized languages like Perl and Fortran. Although proficiency in the specific programming language is very helpful for discovering embedded requirements<sup>6</sup>, running the application and observing its behavior will provide nearly as much information about the system's functional characteristics.

Another excellent resource is the company's issue tracking system (automated or not), which often becomes the de facto repository of system requirements as enhancements are logged as system "fixes." If the investigative team is fortunate, they can run reports on issues logged as ENH or Enhance or some other identifier that indicates the "fix" was actually a new piece of functionality. If they are unlucky, then an extended excavation may be in order (see tips on [excavating legacy documentation](#) below).

In addition to identifying sources of information, you should also consider the conditions of the "dig-site" for your system requirements investigation. Based on what you know about the organization's culture, pressures, needs, and so forth, is the company environment favorable or neutral? The environment is favorable if the organization recognizes the need to reform the requirements capture process and welcomes the investigator and his/her team. A neutral environment is one in which the local requirements team is "too busy" to be bothered with what the investigator is doing, but offers to help "when they get the time." A neutral environment will require a heavy reliance on independent study, as defined below in the section on [Excavation and Cataloging](#).

## **The Curse**

What good archaeology story would be complete without a curse? Here, the curse is the likelihood that existing requirement artifacts will be misleading or false; people with the knowledge will have left the company; the code is obscure and filled with dead-ends and traps for the unwary. Woe to those who would seek to disturb the resting-place of this most sacred software design!

Needless to say, caution must be exercised when unearthing system

requirements. It is best to validate the discovered functionality with the requirement stakeholders, system users, and other knowledgeable individuals. In particular, the successful requirements Archaeologist will befriend the testing team; they can be the best allies of those seeking to discover the "true" system function.

Sometimes the gaps in a requirements base are more telling than the requirements themselves. For example, if a piece of functionality was "rushed" into production, often there are few or no supporting requirements. Moreover, if something was important enough to the business to be rushed into production, then it is likely a critical part of the software functionality.<sup>7</sup> In this case the investigator should look to the software issue management system and personal interviews for clues as to what's missing.

It's also important to be careful with sections of the code that are not executed, otherwise known as "dead code." As it is very difficult to determine which code blocks are "live" and which are "dead," a good debugging or profiling tool will be useful in determining areas of code that are no longer functional.

## **Excavating and Cataloging**

What about the actual mechanics of retrieving and organizing the requirement "artifacts"? Let's consider the techniques of interviewing, diagramming, investigating, and capturing requirements from existing sources.

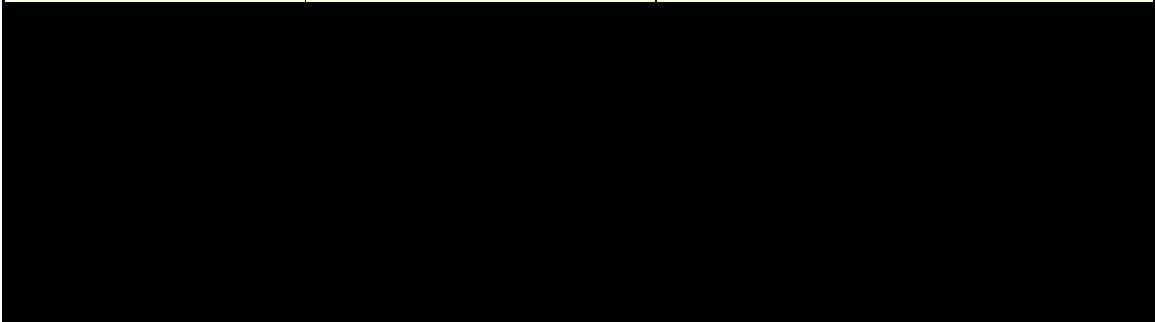
Often times the principal investigator will work alone, sifting through the available materials to locate information of interest. Although this style of investigation may be the only one available in neutral or hostile site environments, it carries a high probability of making an incorrect hypothesis -- that is, you might easily capture a requirement that was never actually implemented or was later modified. It is therefore best that the investigation team work as collaboratively as possible with the "natives." Most of the activities detailed in Table 1 assume system experts are available, but a great deal of groundbreaking work can be performed prior to meeting with system stakeholders and experts. The resourceful investigator can gain a good understanding of the existing system through all of these activities. Even a poorly documented legacy system will begin to divulge its secrets when you use systematic investigative techniques. Although you might not need to undertake these activities in a particular order, the order presented is the one I've found most effective.

**Table 1: Techniques for Investigating Requirements**

Method	Activity	Description
Independent Study	Documentation Review	<p>Most organizations will have <i>some</i> documentation, even if it is not very accurate or complete. Read and review as much of this information as possible, identifying "candidate" requirements and recording them in a requirements management tool. Make particular note of exception handling (i.e., errors). Developers usually start with the desired functionality first and worry about error conditions later, if at all. Consequently, system exceptional conditions and handling is usually the last area for development and often the most poorly documented.</p>
Independent Study	System Functional Modeling	<p>Next, excavate the application itself. Use activity diagrams<sup>8</sup> ("flow charts on steroids") to quickly capture most of the existing system functionality. Create them by using the application as if you were an end user (e.g., entering data via the user interface, creating accounts with a command line, performing searches, etc.). Do this alone or in conjunction with a system expert (e.g., a member of the testing team). The expert can use the application while you record and/or diagram the user activity steps using a separate laptop computer. It is likely that much of the functionality will not have a corresponding requirement in the available documentation.</p>

Independent Study	Defect Tracking Review	<p>Review issue tracking reports, which might be organized by functional area via a tracking tool such as Rational ClearQuest.<sup>9</sup> Run a report on bugs matching a particular functional area. System enhancements often masquerade as "fixes" and are therefore not captured within the legacy requirement documentation. Correct identification of these requirements requires cross-referencing against documented feature requirements, so review documents first. If the development staff uses a "trial-and-error" approach, there will likely be a chain of issue reports that chronicle changes to a piece of functionality.</p>
Independent Study	Code Tracing	<p>Even if you are not familiar with the coding language, often programmer comments will be embedded in the code base with references to issues (e.g., "Fixed BUG1403 - 01/03/2001"). Asking a programmer to review that section of code and report on the expected behavior may be the <i>only</i> way to find exceptional conditions that are difficult to simulate and reproduce during the functional activity-modeling task described above.</p>
Interviewing	Individual Interviews	<p>Conduct interviews one-on-one, reviewing system documentation beforehand to provide a familiar starting point for system experts to begin discussions.</p> <ul style="list-style-type: none"> <li>• Ask subject matter experts (SMEs) to validate the documentation and</li> </ul>

		<p>facilitate closure of gaps or update incorrect information. Use activity diagrams created for system functional flows, which highlight areas of the system that are not well defined -- particularly exception handling aspects of the system.</p> <ul style="list-style-type: none"> <li>• Alternatively, ask SMEs to walk through the application and comment on how and when each section of the system was established. Multiple interviews with SMEs are usually required to cover a reasonably complex software system.<sup>10</sup></li> </ul>
<p>Interviewing</p>	<p>Group Sessions</p>	<p>An extension of the individual interviews, these are best used to review updated documentation and findings. Investigators present their work and facilitate constructive review of the information, perhaps using a projector attached to a laptop computer to display and capture comments in the documentation. Save this approach until a sizeable portion of the system requirements have been rediscovered and documented.</p>



<p>Multi-Day Workshops (Facilitated Re-Discovery)</p>	<p>Small Group Sessions</p>	<p>Rediscovery workshops serve a dual purpose:</p> <ul style="list-style-type: none"> <li>• To capture and document requirements into use cases directly from people closest to the system behavior.</li> <li>• To begin teaching the team a good technique for capture, presentation, and maintenance of system requirements as a set of well-formed use cases and supplemental specifications.</li> </ul>
---	-----------------------------	---

## Deciphering and Translating

After the capture of requirements come the tasks of deciphering and translating. Many requirements are captured using some form of functional decomposition technique. Although this common technique is useful for breaking a problem domain into manageable sections, it is often misused to indicate implementation details, and is very difficult to interpret because the standalone statements don't form a cohesive "story" about the system functionality. Even more common is capturing requirements as highly abstract business statements (e.g., "the system shall be intuitive and user-friendly"). As a result, a common viewpoint must be found that can be used to translate these into a more understandable "story-telling" format (e.g., use cases). Needless to say, performing this task will require diligent attention to detail and some form of Rosetta Stone (such as a business vision document) that will provide guidance in piecing together the elements of system history.

**Capturing Functional Requirements.** Functional requirements represent the bulk of requirements for a user-intensive system. When translating these requirements, it is usually advisable to take a top-down approach. As the traditional structured decomposition approach is still very popular, I will use this form in the example (see [Appendix](#) for a complete translation example, from structured declarations to use-case flows and supplemental specifications). When translating structured declarative statements, look for phrases such as "the system shall present" or "the system shall display" to indicate requirements for user interaction. When a requirement begins with "The system shall perform," this is a good indication that the requirement is for a system-processing step. Any requirements that contain restrictions, such as "the system must respond within 60 seconds," are good candidates for supplemental specifications. A

decomposition, by its very nature, is hierarchical, so this structure can be used to indicate an initial separation of functionality. For example, if a requirement hierarchy is called "Stock Purchase and Reconciliation," this may represent a set of use cases around "Purchase Stock" and "Reconcile Accounts." Thus, you can rapidly construct an initial use case survey from a primary grouping of similar functionality, or from the structure of the statement hierarchy itself. This will provide a framework for tracing statements into functional flows.

**Detailing Data Descriptions.** Data descriptions are often scattered about the documentation and code, and may contain more implementation detail than is appropriate for a requirement statement. This typically happens when data dictionaries created from database tables are placed in requirements documentation. The details of primary/foreign key implementation and data types do not belong in a requirement description. Instead, requirements should describe the relationships between data elements and the restrictions on values that these elements can contain. For example, taking a section on user activity audit trails from the Appendix use case:

**Audit Trail**

Date of Change

Change Event (Refund, Pre-Paid Minutes, Customer Data, Payment)

Original Information

Altered Information

User Identifier (Lastname + Firstname + UserID)

Supervisor Approval Indicator (Yes/No)

This approach describes the data elements at a level of abstraction that is more readily apparent to business subject matter experts. They can see data relationships without being confused by implementation details that are better suited to a design document.

**Deciphering Business Rules.** System business rules relate to usability, reliability, performance, and scalability (URPS). These types of requirements are typically scattered throughout the documentation and should be re-organized into a set of nonfunctional requirements, including:

- Data Validation Rules
- Processing and Performance Restrictions
- Exception Handling
- External Systems Interfaces
- Security

The most logical place for these requirements is in system supplemental specifications. These can be either directly associated with a particular use case, or created as global requirements for the full system.

## **Public Display**

Once the original requirements base has been discovered and recreated, you need some method for the preservation and presentation of that knowledge. The role of the software archaeologist doesn't stop once a system has been unearthed; it is also his or her responsibility to teach the teams involved how to care for the system artifacts. To be of continued value, requirements must be accurate, current, and presentable.

A requirements base is best maintained with the use of a suitable organizational tool, and there are a number of excellent candidates, some relatively inexpensive. At a minimum, the tool should have the ability to record requested features, requirements (including use-case scenarios), and supplementary documentation (e.g., a glossary<sup>11</sup>). There are a number of good papers and books on the topic of requirements management.<sup>12</sup> All of the discovered and created artifacts, including use-case models, activity diagrams, and data dictionary information, should be controlled and maintained through such a documentation management system.

To ensure the continued maintenance of the requirement artifacts, you should establish a process of continuous training and development for the requirements management team. This could include attending industry conferences, bringing in consultants/trainers to instruct new and existing personnel, and rewarding employees who seek out additional training and experience. As of this writing, there are no universally accepted certifications for requirement engineers, but many in the industry are attempting to build a body of knowledge that can be used for this purpose.<sup>13</sup>

One of the greatest dangers for the requirements archaeologist is the possibility of backsliding into the same problems that necessitated the reconstruction effort in the first place. Things to watch out for include inappropriate schedule pressures, changeovers from experienced to inexperienced personnel, funding cuts, rapid expansion of the business (and consequently the software functionality), and a drop in visibility (e.g., "Hey it's working; let's forget about it!"). Constant vigilance is the price you must pay to prevent these pitfalls from destroying all of your hard-won work.

## **The End?**

I consider the process for capturing requirements no different from the process for maintaining software code: Periodic refactoring and reorganization are required for a healthy system, and system requirements must be continuously revisited and revised to ensure that they meet the real needs of customers and end users. The requirements base represents a key company asset, one that is often overlooked. Another way to put it is: *Requirements are money*. There is a direct relationship between correct system requirements and final acceptance of the software product, which usually means *payment*. The relationships are rather straightforward:

Correct Requirements -> Correct System  
Correct System -> Customer Acceptance

## Customer Acceptance -> Money

Unfortunately, if you're working with a legacy system that's new to your team, most likely you will have to re-capture the system's business requirements before you can move forward with a system upgrade or integration. Although it's entertaining to think of system requirements rediscovery as a form of archaeology, the sad truth is that we must do this only because companies so often fail to understand the simple relationships described above. Although the rediscovery process is difficult and sometimes expensive, however, it always pays off in the end. Once a team understands the system's driving forces - its behavior, states, data elements, and business rules - they will understand how to make the system more successful as well as how to leverage it in more challenging and difficult projects.

## References

Alistair Cockburn, "Goals and Use Cases." *Journal of Object-Oriented Design*. September 1997, pp. 35-40.

E.F.J. Ecklund, L.M.L. Delcambre, and M.J. Freiling, "Change Cases: Use Cases that Identify Future Requirements." OOPSLA, 1996.

Ivar Jacobson, Grady Booch, and Jim Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, 1999.

Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wesley, 2000.

Ben Lieberman, "UML Activity Diagrams: Versatile Roadmaps for Understanding System Behavior." *The Rational Edge*, April 2001.

Tom Rowlett, "Building an Object Process Around Use Cases." *Journal of Object-Oriented Design*. March/April 1998, pp. 53-58.

## Appendix: Structured Requirements to Use-Case Translation

The following statements are excerpts from a larger requirements definition for a customer care and automated service-provisioning system requested by a fictitious telecommunications firm. These requirements can be used to identify Actors, create initial use cases, and extract the beginnings of a data dictionary for the system.



## System Requirements

CUSTOMERPro™ Customer Care System

## **Service Provisioning**

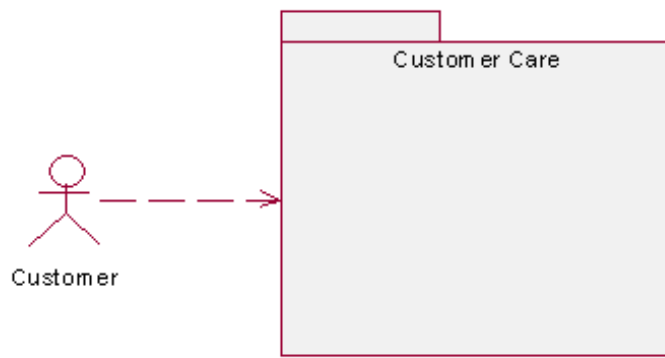
1. Customer shall be able to call a specific 888 number to automatically provision the telephony service after headset purchase via the PHONEMaster™ provisioning system.
2. Customer shall be able to call a specific 888 number or use a Web site to suspend telephony service for a period of up to six months. The Web interface will be easy to use and intuitive to the user.
3. Customer shall be able to have service disconnected by calling a specific 888 number or using a Web site.

## **Customer Service**

4. System shall support Internet capabilities for customers to access and check their account(s), service status, and remaining pre-paid minutes.
5. System will permit the Call Center to accept payments into the CUSTOMERPro™ system via check, cash, credit card, or electronic fund transfer. Payments that are overdue will be noted and charged a 2 percent late fee, which will be waived if the customer has premier service account status.
6. Call Center personnel will be limited to making refunds of \$50 or less without supervisor approval.
7. All Call Center or customer actions will be logged and tracked for audit purposes.
8. The Web interface will be easy to use and intuitive to the user.
9. All Call Center personnel shall be able to access the same information as well as issue refunds and add pre-paid minutes to customer accounts (within specified limits, unless approved by the supervisor).

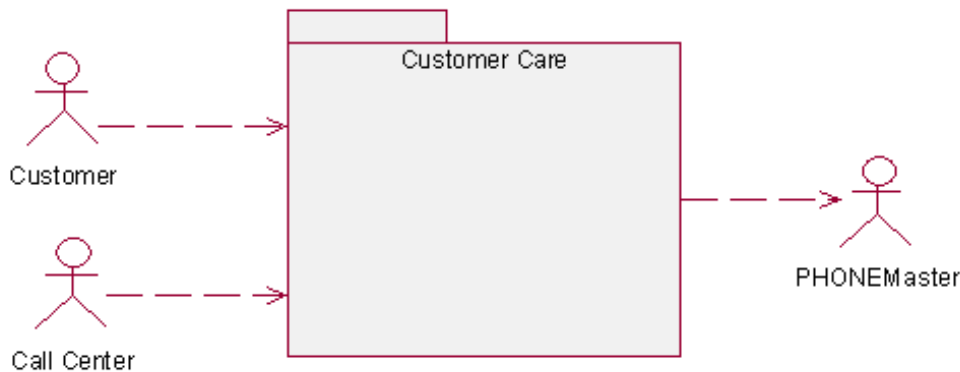
## **Translating Requirements into Use Cases**

The "requirements" presented above are a collection of statements intended to describe the desired system (CUSTOMERPro). However, there is no clear theme or story. At first glance, there seem to be two clearly defined systems: one for Service Provisioning and a second for Customer Care. In reading the Service Provisioning section, it is clear that the term *customer* is used frequently to describe the functionality; this suggests that the functionality can actually be considered part of Customer Care. So we begin by defining a system boundary, as shown in Figure A-1.



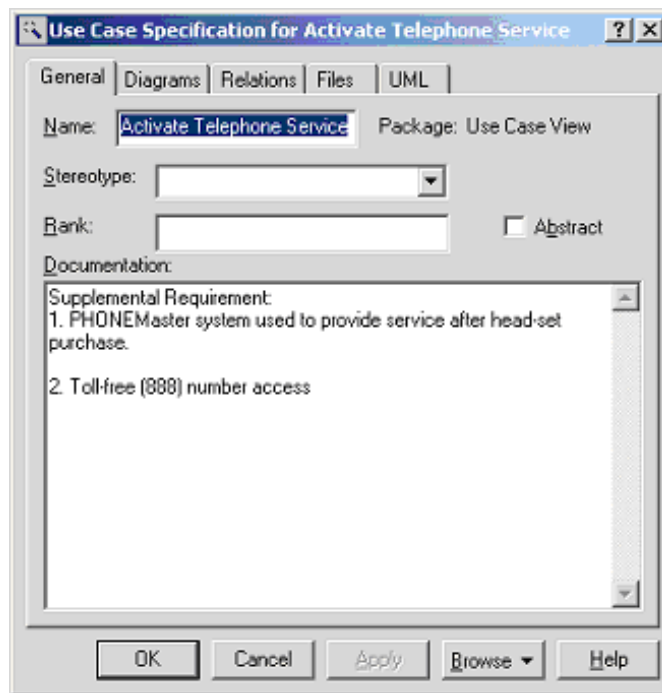
**Figure A-1: Defining a System Boundary**

Next we look for Actors to the system. A "customer" is mentioned in statements 1, 2, 3, and 4. Call Center personnel are noted in statements 5, 6, 7, and 9. A system called "PHONEMaster" is noted in statement 1. So we amend our diagram as shown in Figure A-2:



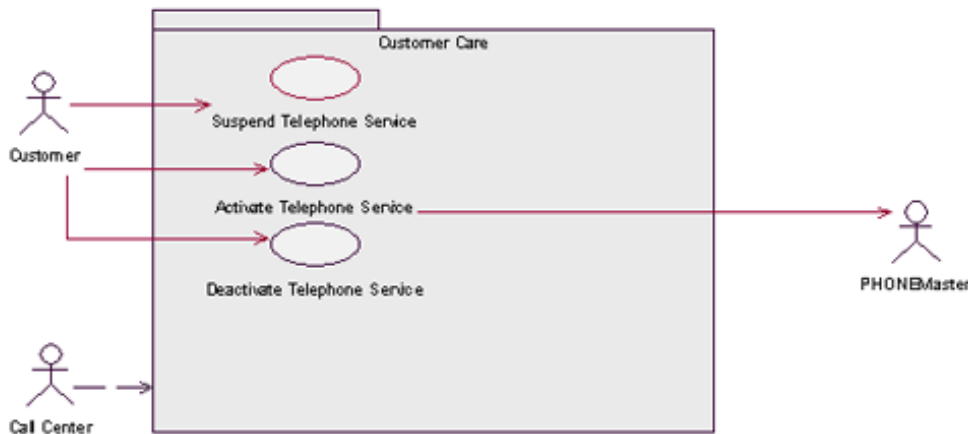
**Figure A-2: Adding Actors**

Now we can begin to look for use cases and assign an Actor for each. Requirement 1 suggests some form of ability for the customer to activate phone service, while statement 2 suggests the ability to suspend said service, and statement 3 provides for disconnection of service. In addition, we find that there is some interaction with a separate system ("PHONEMaster"), although it is unclear whether this interaction is only for "activation," or also for the "suspend" and "disconnect." The implied supplemental requirement is that we will be developing a system that can be accessed via a regular telephone (see statements 1, 2, and 3). I usually capture such information directly into the documentation for a use-case diagram element, as shown in Figure A-3.



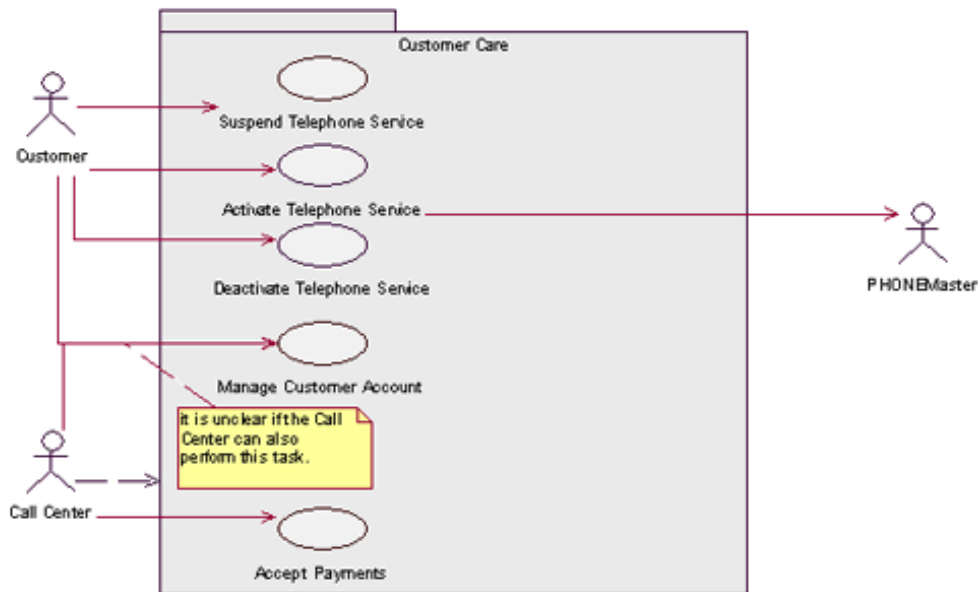
**Figure A-3: Capturing a Supplemental Requirement**

Then we can capture the use cases for this requirement, as shown in Figure A-4.



**Figure A-4: Use-Cases for Supplemental Requirement**

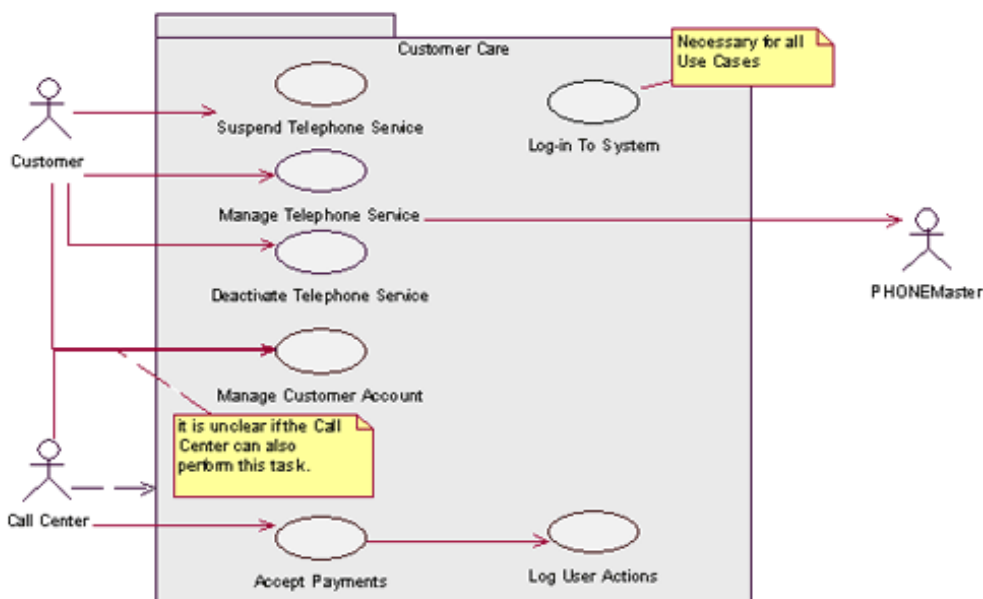
Statement 4 is a combined requirement statement, providing for both account management services and something called "pre-paid minutes." Note that this statement also contains a non-functional requirement for Internet-based access. Statement 5 is another compound requirement that combines information on acceptable payment types (check, cash, credit card, and EFT) with requirements about handling overdue payments. The astute investigator will note that statement 5 mentions something about "premier service account status." This suggests that the account management functionality noted in statement 4 will have some kind of information about the customer's role with the company (we can also capture this information into the use-case documentation section of the diagram). Our use case model is now shown in Figure A-5.



**Figure A-5: Use-Case Model**

Statement 6 suggests that the Accept Payments use case will also have a scenario for refunding customer accounts, with value limitations on the amount and approval requirements.

Statement 7 introduces a new functionality: logging. Also implied is some form of security (it would be hard to log a Call Center individual without unique identification). So we have an additional use case, as shown in Figure A-6.

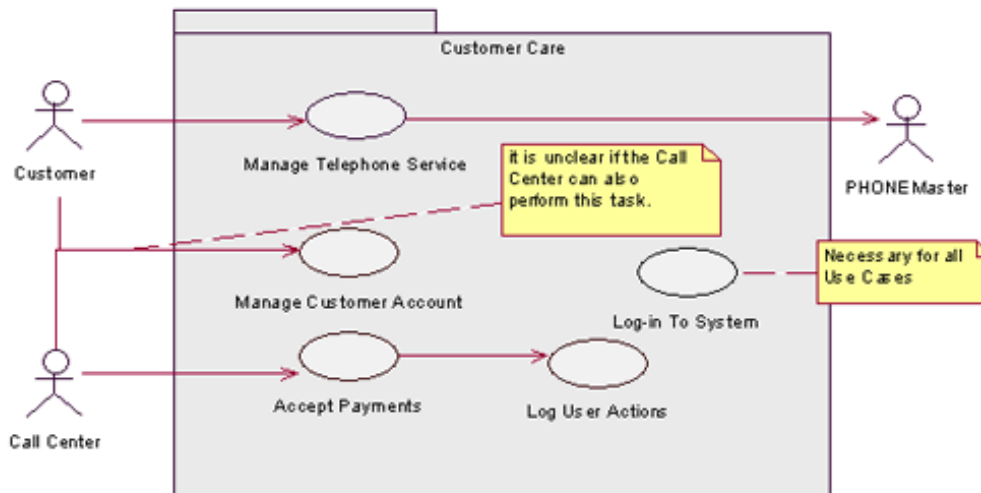


**Figure A-6: Model with Additional Use Case**

Note that statement 8 is not really a requirement, but rather a generic statement about usability that cannot be measured or tested.

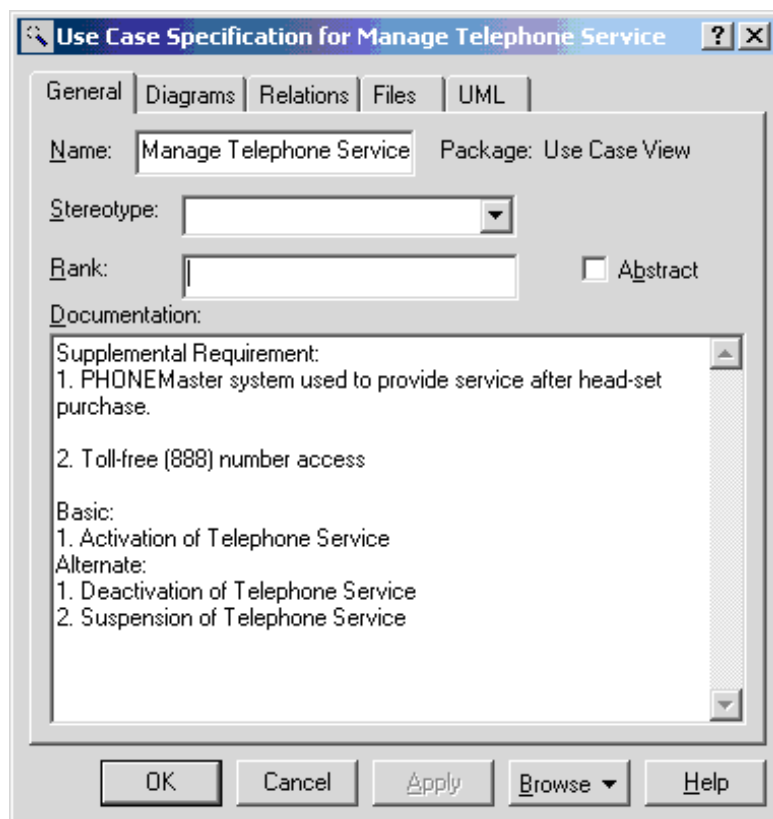
Statement 9 suggests that Call Center personnel can "access the same information," but is unclear about what information is referenced (Customer Account?). Also, it reiterates refund capability as well as the ability to add "pre-paid" minutes to a customer account (with undefined limitations and, again, supervisor approval).

We can now trace all of these statements to the use-case model, capturing non-functional requirements that are directly associated with specific use cases. We can also refactor the Telephone Service use cases to take advantage of the similarity between activation, suspension, and deactivation (Figure A-7).



**Figure A-7: Use-Case Model Capturing Non-Functional Requirements**

We can also capture the scenarios captured in the documentation, as shown in Figure A-8.



**Figure A-8: Capturing Use-Case Scenarios in Documentation**

Clearly, this analysis has led to more questions than answers. Some additional follow-up activities can include: definition of the "pre-paid" minutes functionality; clarification of the 888 number interface; a definition of "head-set" (statement 1); and inquiry on the logging needs and data elements:

### **Audit Trail**

- Date of Change
- Change Event (Refund, Pre-Paid Minutes, Customer Data, Payment)
- Original Information
- Altered Information
- User Identifier
- Approval Indicator

The next steps also include the creation of user flows to describe each of these system interactions (e.g., activity diagrams), and a further description of alternate and exceptional scenarios for each use case.

As shown in this example, even starting with a minimal description of the system, it is possible to create a reasonable model to capture and organize information as an investigation proceeds. I used Rational Rose to create and capture the information; this is my usual approach because Rose permits me to capture and organize my models in one step. Alternatively, you could use a less complex approach with a simple drawing tool and

skeleton use-case documents. The key is to take an analytical approach and maintain traceability to the original requirement statements.

---

## Notes

<sup>1</sup>The term *structured requirement* refers to the classic declarative form, "The system shall ý"

<sup>2</sup>See <http://www.rational.com/reqpro>

<sup>3</sup>See <http://www.rational.com/>

<sup>4</sup>See <http://www.textpad.com>

<sup>5</sup>See Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. (Addison-Wesley, 2000) and Ivar Jacobson, Grady Booch, and Jim Rumbaugh, *The Unified Software Development Process* (Addison-Wesley, 1999).

<sup>6</sup>I use the term *embedded* to refer to requirements that exist in no other location than the code base. This is often a result of direct communications between a specific developer and the client, and the functionality is folded directly into the code base without review.

<sup>7</sup>It has been pointed out to me that using "lack" of documentation as an indicator of potentially critical requirement(s) is similar to the classic method of detecting pneumonia with a stethoscope -- it's where you don't hear any breath that a serious problem is likely to exist.

<sup>8</sup>See Ben Lieberman, "UML Activity Diagrams: Versatile Roadmaps for Understanding System Behavior." *The Rational Edge*, April 2001.

<sup>9</sup><http://www.rational.com/products/clearquest/index.jsp>

<sup>10</sup>For additional information on soliciting requirements using interviews, see Dean Leffingwell and Don Widrig, *Managing Software Requirements, A Unified Approach*. Addison-Wesley, 2000, p. 491.

<sup>11</sup>Requested features are based on customer requests for functionality; requirements can be captured as use-case scenarios to represent the bulk of the functional requirements; supplementary documentation is useful for defining terms, nonfunctional requirements, examples, and implementation restrictions (e.g. third party interfaces).

<sup>12</sup>See the References listed below and the following Rational Edge articles:

"Ending Requirements Chaos"

[http://www.therationaledge.com/content/aug\\_02/m\\_endingChaos\\_sa.jsp](http://www.therationaledge.com/content/aug_02/m_endingChaos_sa.jsp)

"Features, Use Cases, Requirements, Oh my!"

[http://www.therationaledge.com/content/dec\\_00/t\\_usecase.html](http://www.therationaledge.com/content/dec_00/t_usecase.html)

"Iteration-Specific Requirements"

[http://www.therationaledge.com/content/mar\\_01/t\\_iteration\\_mt.html](http://www.therationaledge.com/content/mar_01/t_iteration_mt.html)

<sup>13</sup>Software Engineering Body of Knowledge (SWEBOK): <http://www.swebok.org/>



**For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided.**

***Thank you!***

**Copyright [Rational Software 2002](#) | [Privacy/Legal Information](#)**