Regular Paper

The Hybrid Process That Combines Traditional Requirements and Use Cases

Jesse Daniels[1, *] **and Terry Bahill**[1, 2]

[1]BAE SYSTEMS, 16250 *Technology Drive, Mail Zone* 6164-E, *San Diego*, CA 92127

[2]*Systems and Industrial Engineering, University of Arizona, Tucson*, AZ 85721-0020

Received 29 March 2004; Accepted 7 July 2004, after one or more revisions
Published online in Wiley InterScience (www.interscience.wiley.com).
DOI 10.1002/sys.20013

ABSTRACT

For many years systems engineers have produced traditional system requirements specifications containing shall-statement requirements. The rapid adoption of use case modeling for capturing functional requirements in the software community has caused systems engineers to examine the utility of use case models for capturing system-level functional requirements. A transition from traditional shall-statement requirements to use case modeling has raised some issues and questions. This paper advocates a hybrid requirements process in which use case modeling and traditional shall-statement requirements are applied together to effectively express both functional and nonfunctional requirements for complex, hierarchical systems. This paper also presents a practical method for extracting requirements from the use case text to produce a robust requirements specification. © 2004 Wiley Periodicals, Inc. Syst Eng 7: 303–319, 2004

Key words: unified modeling language; UML; requirements; object-oriented design; object-oriented systems engineering; engineering communication

Contact grant sponsor: AFOSR/MURI, grant number F49620-03-1-0377

*Author to whom all correspondence should be addressed (e-mail: jesse.daniels@baesystems.com; terry@sie.arizona.edu).

Systems Engineering, Vol. 7, No. 4, 2004
© 2004 Wiley Periodicals, Inc.

1. INTRODUCTION

For decades a traditional requirements specification, which provides textual statements of imperative, has served as the primary means by which systems engineers bounded and communicated a system's capabilities and constraints. In recent years, the popularity of capturing software requirements through use cases has increased dramatically, with use case-based analysis becoming the foundation of modern software analysis techniques.[1] Due to the success of the use case approach in the software community, systems engineers have begun evaluating use cases as a potential tool to document requirements. The hope is that using a common methodology for requirements capture will help align systems engineers and software engineers, allowing for ease of transition from system and subsystem requirements into software requirements. Adopting use cases for systems engineering has not been as simple as this. Pertinent questions such as "How do use cases relate to the requirements specification?", "Are use cases the requirements?", and "Where are the requirements actually documented?" have arisen. This paper argues that traditional shall-statement requirements are not mutually exclusive with use cases. Rather, combining shall-statement requirements and use case modeling techniques provides a complimentary and synergistic way to document and communicate relevant information to effectively design, develop, and manage complex systems. We believe that the resultant quality and clarity of knowledge gained by combing the two methodologies (use cases and shall-statement analysis) is worth the time spent.

This paper assumes at least a working knowledge of use cases. Many references are available to introduce unfamiliar readers with use case and requirements-gathering concepts [most notably, Cockburn, 2001; Kulak and Guiney, 2000; Armour and Miller, 2001; Leffingwell and Widrig, 2000].

2. WHY WRITE A REQUIREMENTS SPECIFICATION ANYWAY?

Systems engineers develop requirement specifications for a number of reasons (hopefully not just "because the customer said so!"). First and foremost, requirements are documented so that the characteristics of the to-be system can be discussed and analyzed among engineers and stakeholders to facilitate a shared clear vision of the completed system—before it is built. Higher level design decisions are reflected to lower level design as requirements; imperatives of textual, or model-based nature. This is a natural consequence of hierarchical design, and is essential when developing complex systems. Also, requirement specifications are necessary for subsystem procurement [Request for Proposal (RFP), proposal evaluation, award, production, and verification]. In many cases, requirements are a predominant contractual vehicle for payment of the builder and are used to measure system performance. A requirements specification is often called out as a deliverable in a Contract Data Requirements List (CDRL) as part of a Statement of Work.

Systems engineers and architects use the requirements set to ensure system and enterprise-wide coverage of capabilities and characteristics. Implementation engineers use the agreed-upon requirements set to derive additional requirements and develop subcomponents that contribute to a system that meets those requirements. Program managers track requirements volatility to measure and monitor risks associated with requirements creep and redefinition. Customers sign off on requirements during acceptance testing to ensure that the capabilities offered by the system meet their needs and intent. It is clear that gathering, discussing, and decomposing requirements is an essential step in the engineering process to ensure that the system, once built, actually satisfies the stakeholders' needs.

Table I (adapted from Young [2004]) provides some criteria for "good" requirements. Indeed, there are additional criteria that could be listed [Bahill and Dean, 1999]; however, we selected a set that was at least partially dependent on the *method* used to specify the requirement. Criteria such as "attainable" and "implementation free" are good criteria for a requirement, but whether these criteria are met is largely independent of the method used to capture the requirement and more dependent on the subject of the requirement. We will use Table I later to assess whether the techniques discussed in this paper generate good requirements specifications.

Another important point to mention is that, in general, there is not a "one size fits all" requirements development method. Some industries, such as defense, typically require a high level of formality (or ceremony) and detail in requirements specifications. A high level of formalism is usually associated with very complex systems with tight performance constraints that are hard to prototype. Systems that are divided into subcontracted components and projects that have little interaction with the customer also require very precise specifications, which usually results in formal requirements documents. Other industries do not need such formal specifications. Simpler projects with frequent customer interaction can sometimes be more informal in their approach. The concepts described in this paper can be tailored to develop formal or more informal requirements specifications.

[1]Although use cases are often included in object oriented analysis and design methodologies, there is nothing inherently object oriented about use cases.

Table I. Criteria of a Good Requirement [Young, 2004]

Criterion	Description
Necessary	Can the system meet prioritized, real needs without it? If yes, the requirement isn't necessary.
Verifiable	Can one ensure that the requirement is met in the system? If not, the requirement should be removed or revised. Note: The verification method and level at which the requirement can be verified should be determined explicitly as part of the development for each of the requirements. The verification level is the location in the system where the requirement is met (for example, the "system level," the "segment level," and the "subsystem level").
Unambiguous	Can the requirement be interpreted in more than one way? If yes, the requirement should be clarified or removed. Ambiguous or poorly worded writing can lead to serious misunderstandings and needless rework.
Complete	Are all conditions under which the requirement applies stated? Also, does the specification document all known requirements?
Consistent	Can the requirement be met without conflicting with all other requirements? If not, the requirement should be revised or removed.
Traceable	Is the origin (source) of the requirement known, and can the requirement be referenced (located) throughout the system?
Concise	Is the requirement stated simply and clearly?
Standard constructs	Requirements are stated as imperative needs using "shall." Statements indicating "goals" or using the word "will" are not imperatives.

3. PROBLEMS WITH TRADITIONAL REQUIREMENTS STATEMENTS ALONE

A traditional requirements specification attempts to capture the imperative functionality and constraints of a system by enumerating each requirement using "shall" notation through which individual capabilities and constraints are expressed (e.g., The system *shall* ...). The requirements set is typically structured according to functional areas, and each requirement is given attributes, possibly traced to and traced from other requirements. Shall statements are used to describe different types of requirements from functional, performance, and security to reliability, availability and usability.

The difficulty is that if the requirements specification consists solely of shall-statement requirements, then for even moderately complex systems the requirements set can become unwieldy, containing hundreds or thousands of requirements. This by itself is not a problem, as modern systems are very complex and often need this level of detail to avoid ambiguity. The problem is that documenting the requirements as a set of discrete and disembodied shall statements without context makes it very difficult for the human mind to comprehend the set and fully interpret the intent and dependencies of each and every requirement. This makes it hard to detect redundancies and inconsistencies. In short, large shall-statement requirement specifications make it difficult for engineers and customers to really understand what the system does! Because there is no straightforward way to assimilate the requirements set, requirements are often misinterpreted, redundant, and incomplete, which can result in inferior, overly expensive systems and ultimately dissatisfied stakeholders.

To alleviate some of these problems, engineers may take a divide and conquer approach by partitioning the requirements set into manageable groups that are easier to comprehend. This technique adds clarity to the set, but it is still difficult to relate the system's capabilities to what stakeholders have in mind in terms of an operational system. A narrative describing a particular functional area may also be developed to provide some context for a given requirements section. Sometimes, a set of "user requirements" or "source requirements" is provided to describe the system's intended operations from the user's perspective. While user requirements better communicate the voice of the user, there is still a translation step that must be performed by engineers who must derive system requirements from the user requirements based on interpretation. With this approach, there is a fair likelihood that information will be misconstrued in the translation.

The intent of this section was to point out some commonly observed shortcomings of the shall-statement requirements specification, most notably communication, comprehension, and simplicity of the requirements set. In the next section we introduce use cases, which show promise in helping to overcome these difficulties.

4. ENTER USE CASES

Use cases have been proposed and almost universally accepted by the software community as a requirements gathering and documentation tool that captures system requirements through generalized, structured scenarios that convey how the system operates to provide value to at least one of the system's actors. The primary reason why use cases have become a popular method is the simple and intuitive way in which the system's behavior is described. Use case models are designed to serve as a bridge between stakeholders and the technical community. Through a use case model, stakeholders should be able to readily comprehend how the system helps them fulfill their goals. Simultaneously, engineers should be able to use the same information as a basis for designing and implementing a system that upholds the use cases.

A set of use cases (collectively referred to as the use case model) should capture the fundamental value-added services that user and stakeholders need from the system [Adolph and Bramble, 2003]. Use cases can be thought of as a structured, scenario-based method to develop and represent the behavioral requirements for a system. The use case approach subscribes to the notion that each system is built to support its environment or actors—be it human users or other systems. Use cases, by definition, describe a series of events that when completed, yield an observable result of value to a particular actor [Jacobson, Ericsson, and Jacobson, 1995]. The fundamental concept is that systems designed and developed from use cases will better support their users.

To further enhance the contents of a use case description, one or more visual models such as activity diagrams, statecharts, sequence diagrams (aka interaction diagrams in UML 2.0), and collaboration diagrams (aka communication diagrams in UML 2.0) may be associated with a use case. An activity diagram visually depicts the logical sequencing of events (activities) that take place when a use case is instantiated, complete with decisions to represent alternate sequences, swim lanes to delineate the system from its environment, and (optionally, but recommended) object flows to represent data exchange. Interaction and communication diagrams, from a use case modeling perspective, show the exchange of stimuli between the system under design and its actors. During object oriented analysis and design, interaction and communication diagrams show how the system's internal elements collaborate to realize a use cases's behavior. A statechart describes the lifecycle of a use case in terms of input stimuli, states (e.g., where are you in the sequence of events) and outputs. Statecharts are also an excellent means to more formally define the protocol associated with a use case through Protocol State Machines [OMG, 2003: 464]. Such a statechart rigorously specifies the allowed patterns of stimuli exchange between the system and an actor for a given use case. There is a lot more to be said about the object oriented analysis models that complement a use case model. Jacobson and coworkers [Jacobson, Ericsson, and Jacobson, 1995; Jacobson, Griss, and Jonsson, 1997; Jacobson, 2000a, 2000b] provide good discussions on how object analysis models complement use case models in terms of cooperating analysis objects (boundary, control, and entity objects, to be more specific). These object analysis models provide a very convenient segue between requirements and design and help solidify the understanding of requirements.

Use cases are also incorporated into the emerging Object Oriented Systems Engineering Method (OOSEM) proposed by Friedenthal, Lykins, and Meilich [2000], where use cases are employed to conceptualize the capabilities of a system. Using the OOSE approach, the analysis of each individual use case's sequence of events provides the basis for further elaborating a UML class that represents the system under design, as well as input/output (I/O) entities in the Elaborated Context Diagram (ECD).

Ultimately, use cases have been shown to be a very simple and effective tool for specifying the behavior of complex systems [Jacobson, Ericsson, and Jacobson, 1995]. Therefore, it is no surprise that systems engineers have been curious about this method of capturing and communicating requirements.

4.1. Use Cases and Concept of Operations

The notion of writing "stories" or narratives to describe how a system operates is not a new concept. In fact, the use case approach is similar to, but not identical to developing a Concept of Operations (CONOPS). CONOPS development is a well-known communication tool that has been used for many years in systems engineering (typically for military applications) to describe the functionality of a system, usually in the context of the overall enterprise. A CONOPS is generally written by the end-users or customers and is intended to provide the vision and intent for how the system should work within an operational environment in an easy to read format. CONOPS documents can be very detailed and include realistic stories in which the system improves some aspect of a workflow during a plausible scenario. Other CONOPS are more technical and describe user interfaces and data flow through the system.

Because of the variety of ways in which a CONOPS can be written, a CONOPS document does not typically contain the right information to be used directly for extracting system requirements in an unambiguous way. Indeed, CONOPS documents are rarely consistent in content, detail, and format. However, a CONOPS document can provide excellent input in generating a use case model. A CONOPS usually provides the business level context for the system, which is the ideal setting to extract use cases. If a CONOPS document is available, we recommend using it as the starting point for developing use cases. In addition to providing system context, which will lead to higher quality use cases, relating a use case model back to a customer-developed CONOPS will add credibility and customer buy-in to the use cases.

5. WHERE THE DIFFICULTY COMES IN

Although use cases solve some of the problems with specifying requirements when contrasted with the shall-statement method, there are issues with the use case method relative to the application of use cases in systems engineering. There are a few possible reasons for these issues:

1. Use cases don't *look* like a traditional requirements spec.
2. Use cases "feel" somewhat vague.
3. Use cases do not contain *all* of the requirements.
4. The completeness of a set of use cases is difficult to assess, especially for unprecedented complex systems.

Other contributing factors result from the fact that use cases are relatively new on the scene and systems engineers and customers do not have much experience with them. Systems engineers are accustomed to developing traditional requirement specifications. Use cases are a departure from the past. It is our belief that eventually use cases will be commonplace. Until then, it is important to educate customers and systems engineers on the use case concepts and the fact that employing use cases into the overall systems engineering process will help reduce risk, and increase the probability of delivering the system expected by the customer.

Additionally, the UML specifications [OMG, 2003] do not provide much guidance on applying use cases. Many authors, including Cockburn [2001] and Armour and Miller [2001], have improved on the available literature and give good practical direction on applying use cases. The Rational Unified Process and the Rational Unified Process for Systems Engineering provide detailed guidance on practical methods to employ use cases in both systems and software engineering projects [IBM RUP 2003, IBM RUP SE 2003].

6. HOW SHOULD WE STATE THE REQUIREMENTS?

Use case modeling has shown strengths in many areas of requirements capture. The dialog between the system and the actor that is expressed in a use case's sequence of events explains clearly how a system reacts to stimuli received by the system's actors. What engineers typically do not realize is that the actual functional requirements are embedded in the sequence of events. In other words, a use case *describes* the requirements. Whenever a use cases' sequence of events indicates that the system performs some function, a functional requirement has been imposed on the system. Each use case is chock full of functional requirements (possibly multiple requirements per sentence!) when viewed this way. By design, the requirements are an integral part of a use case's natural language story.

A use case's strength is also its weakness. To keep use cases simple, readable, and manageable, they can only tell a fraction of the complete story without becoming unwieldy and difficult to understand [Cockburn, 2001; IBM RUP, 2003]. The fact is that use cases alone were not meant to capture *all* of the requirements. A use case is very good at capturing the functional requirements for a system in an understandable and unthreatening way. However, shall statements, tables, equations, graphs, pictures, pseudo code, state machines, statistics, or other methods must still be used to capture additional requirements and add richness to provide a sufficient level of detail to adequately characterize a system. All of these artifacts may be related to a use case, but they are typically not expressed directly into a use case's natural language story. In some instances, use cases are not the best method to express a system's functional requirements. For example, systems that are algorithmically intense and do not have much interaction with their environment may be better described using some other method—although use cases can be used in these cases too [Cockburn, 2001; Jacobson, 2000b; Cantor, 2003a, 2003b].

To summarize, use cases excel on being understandable, at the expense of being potentially ambiguous. Shall-statement requirements are typically very well defined and often stand alone, where individual capabilities are expressed in a rather abstract way, out of context with the rest of the system's characteristics. The use of shall-statement requirements alone makes it more difficult for engineers and stakeholders to assess one requirement in a virtual sea of disjointed capabilities. Shall statements are very good at precise expression, while use cases are good at communicating requirements in context. This tension promotes the need for a combined use case-shall-statement approach, where the system's behavior is documented precisely and understandably in the same way that it is derived—by analyzing all of the discrete, end-to-end interactions between actor and system that ultimately provide some value to an actor.

7. HYBRID REQUIREMENTS CAPTURE PROCESS

Towards this end, Leffingwell and Widrig [2000] and Rational University propose a convenient way to document requirements imposed in use case text by including a Specific Requirements section *within* each use case report (Leffingwell and Widrig and the RUP call this section the "Special Requirements" section). The Specific Requirements section in a use case report was originally intended to capture the nonfunctional requirements, which are typically performance requirements on a particular use case step, using shall-statement notation. In this paper, we propose using the Specific Requirements section to document not only nonfunctional, but also functional requirements specified in the use case using shall-statement notation. When the shall-statement requirements are captured in this way, they retain their context since they can be readily traced to the use case sequence of events they were derived from. Using this approach, the understandability of the use case is balanced by the razor-sharp precision of shall statements.

Engineers will also discover requirements that do not fit nicely within the context of one use case, but rather they characterize the system in general. The RUP recognizes this, and proposes including these "homeless" requirements in a separate Supplementary Requirements Specification using shall-statement notation. The Supplementary Requirements Specification is independent of the use case model, and is intended to contain all of those requirements that do not apply cleanly to any single use case. Requirements that describe physical constraints, security (physical security, antitampering, information assurance), quality, reliability, safety, usability, supportability requirements, or adherence to standards, for example, are good candidates for inclusion in the Supplementary Requirements Specification. These types of requirements are typically not local to a specific use case. The FURPS+ requirements model, used in the RUP and presented in Grady [1992], provides a good overview of functional and nonfunctional requirements categorization schemes.

The use case model (which contains the individual use case reports) together with the Supplementary Requirements Specification constitutes a way to completely document a system's requirements in an exact, yet understandable manner. In many cases, this alone will be satisfactory for specifying the requirements. However, some customers may still dictate that a traditional shall statement specification be developed. We discuss this in the next section.

Table II revisits the requirements criteria introduced in Table I. This time the criteria are used to compare and contrast the three methods presented so far: shall statements, use cases, and the hybrid process using both use cases and shall statements. Each technique is given a rating from 1 to 3 for each criteria (3 being the best) to assess how well the technique promotes requirements best practices. We believe that the hybrid process encompasses the strengths of each technique, and excels in each criterion.

Necessary. The use case method excels here. Since use cases are driven by actor needs, each use case represents a set of capabilities that are of value to the system's stakeholders. It is hard to determine whether a shall-statement requirement is necessary without considering the entire requirements set and additional documentation. The hybrid process, due to its use case-driven foundation, rates highly.

Verifiable. Since shall-statement requirements are concise and discrete, they are, in principle, easier to verify than a use case that can dish out multiple requirements in a single sentence. The hybrid process retains the discrete nature of the shall-statement requirements, which makes them easier to verify.

Unambiguous. Shall-statement requirements are more likely to be unambiguous because they can be written much more tersely and precisely. Use cases, due to their narrative format, are not as well suited to the

Table II. Rating the Requirements Methods

Criterion	Shall Statement Method	Use Cases	Hybrid Process
Necessary	1	3	3
Verifiable	3	1	3
Unambiguous	3	2	3
Complete	2	3	3
Consistent	1	2	3
Traceable	2	2	3
Concise	3	1	3
Standard constructs	3	1	3

same accuracy in expression. The context provided by a use case helps remove some of the ambiguity. The hybrid process takes the best of both worlds—context with use cases, and freedom to use more precise wording with shall statements.

Complete. A use case is structured in terms of basic and alternative paths, giving a clear understanding of when a requirement applies, and when it does not. Since use cases are generated by considering the needs of all of the system's actors, it is much less likely that requirements will be missed. Use cases are an excellent way to help ensure that the requirements set is complete. It is difficult to determine whether a shall-statement requirements set is complete without referencing the entire documentation. But many times, due to the structure of the requirements set, you can look for incompleteness [Davis and Buchanan, 1984]. The hybrid process retains the use case method's excellence in complete requirements description.

Consistent. Use cases typically deal with one goal at a time, and therefore it is easier to separate requirements and reduce the risk of conflicting with other requirements. However, neither method alone excels at ensuring a consistent requirements set. The hybrid process, through the use of the supplementary specifications, provides a mechanism to extract requirements that apply across use cases and specify them in one place. This helps maintain the consistency across the set and reduces duplicity. The authors do admit that requirements extracted from use case narratives may need to be refined and aggregated into a comprehensive, nonredundant set of system functional requirements for formal, high-ceremony projects.

Traceable. Use cases, through their actor-driven derivation are easily traced to higher-level actor goals. However, it is not as straightforward to allocate specific use case requirements to the system components. Shall statements can be traced through a numbering scheme as parent–child requirements. The hybrid process retains the traceability to actor goals through use cases, and allows the shall statements to be allocated to system components. The use of object models such as sequence diagrams in the hybrid spec help allocate responsibilities and requirements to the components of the system.

Concise. This is where shall statements really shine. A shall-statement requirement can be worded to be very concise and exact. Because use cases focus on understandability, they are typically not as concise as they have to be readable. The hybrid process incorporates the shall statement's conciseness.

Standard Constructs. The shall-statement method clearly uses the word *shall* for requirements. The use case method could use this standard terminology, but they seldom do. The hybrid process retains the use of standard terminology.

7.1. Hybrid Process

There are many methods for gathering requirements. For complex systems, we believe that a combined use case and shall-statement approach be employed to capture a system's requirements. Cockburn [2001] provides practical guidance for determining when use cases should and should not be used to describe a system's requirements. Shall-statement requirements add the precision necessary to completely and unambiguously specify the system. With any approach, early and frequent interview and review iterations should be conducted with the stakeholder community to ensure that their concerns and desires are addressed. Leffingwell and Widrig [2000] provide an excellent discussion on holding requirements workshops, brainstorming, and storyboarding to help facilitate this process.

The hybrid process for gathering requirements proceeds as follows:

1. Develop a business model to understand how the system under design fits into the overall enterprise. The business model provides context for the system and can profoundly increase the quality of later analysis. Business use case modeling is discussed in more detail later.
2. Discover the system actors and their goals with respect to the system under design. The business model is a valuable input in accomplishing this task. The IBM RUP [2003] provides a nearly mechanical process for deriving system use cases from a business model.
3. Use the actor goals to sketch out the use case report, concentrating on use case names and brief descriptions.
4. Iterate on the important or architecturally significant use cases, filling out more and more detail in the reports and capturing alternative sequences of events, preconditions, and postconditions.
5. In each use case report, document the nonfunctional requirements, typically performance, in the Specific Requirements section as they apply to each use case.
6. Iterate on the use case set to ensure consistency and completeness as the program progresses.
7. In parallel with steps 1–5, develop a Supplementary Requirements Specification to capture system-wide requirements that do not fit cleanly into individual use case reports.

If a traditional requirements specification (shall-statement requirements only) is to be developed:

a. Extract all of the functional requirements from each use case's flow of events and document them in the Specific Requirements section of the use case report, or the Supplementary Specification. Ensure that traceability is maintained.
b. Combine the requirements documented in each use case report's Specific Requirements section, along with the Supplementary Requirements Specification to generate the traditional requirements specification. This should be largely a simple copy and paste operation.

Step a above is really what this paper is about, and therefore deserves more attention. As mentioned before, a use case describes the functional requirements. Wherever in a use case's sequence of events, a system capability or function is called out, e.g., "The system finds...," or "The system sends...," or "The system checks...," or "The system displays...," functional requirements are being imposed on the system. Step a above is really talking about scanning through the use case's sequence of events and extracting out all such statements of behavior, as well as derived requirements that are not explicitly stated. This is where the precision comes in and is where shall statements excel. These extracted and derived requirements can then be easily translated into the shall-statement requirements, e.g., "The system shall find...," "The system shall send...", "The system shall check...", "The system shall protect...." Using natural language, as advocated with use cases to make them accessible by everyone, it is often difficult to be very precise and easy to understand at the same time. Extracting functional requirements and incorporating derived requirements in this way allows for very precise statements of capability without disrupting the narrative unfolding within a use case. The fact that these shall-statement requirements were extracted from and traced to use cases (very likely to a specific use case step) provides higher confidence that they actually satisfy a real need from the actor's perspective—this is where use cases excel. What we are left with is a shall-statement representation of the requirements contained in a use case, which can be made more precise than the use case narrative. Since these requirements are traceable back to the use case narrative, we can always go back and get the context from which it was derived if needed. We cannot stress enough that it is critical that the stakeholders be involved as much as possible in the use case generation and validation activities to ensure that the resulting specification is driven by their needs. Use cases make this easier.

Individual requirements, use cases, and the traceability and attributes of each should be managed in a requirements management tool. We recommend selecting tools that allow free-form text-based specification for documenting use cases and shall statements in addition to an underlying repository for storing and manipulating the requirements as database records. This allows use of analysis tools such as trend analysis, trace matrices, and reports, while keeping the use cases and requirements in user-friendly document format. As requirements are changed in the document, the database is automatically kept in synch, and vice-versa, if the database is updated, the document is synchronized accordingly.

The UML class diagram given in Figure 1 shows how use case, requirement, and specification concepts are related. For a given system, a Use Case Requirements Specification contains one Use Case Model and one Supplementary Requirements Specification. A use case model contains one or more use case reports (also referred to by other authors as use case descriptions). Each use case report contains one or more sequences of events (one main sequence of events, and zero or more alternative sequences), as well as one Specific Requirements section. The sequence of events within the use case report contains the informal functional requirements for that use case, while the Specific Requirements section contains the nonfunctional requirements and the formal functional requirements extracted from the sequence of events.

The Supplementary Requirements Specification contains system-wide requirements that do not fit nicely within the sequence of events or Specific Requirements section of one of the use cases. A traditional requirements specification can be generated by combining the functional requirements and nonfunctional requirements from each use case report along with the system-wide requirements from the Supplementary Requirements Specification.

7.2. Hybrid Process Example

This section presents a simple use case report and is intended to demonstrate how functional and nonfunctional requirements can be derived from the use case description. The primary sections of the use case report are the sequence of events (and alternate sequences), and the Specific Requirements Section. This example was adapted from Bahill and Daniels [2003] and simplistically describes the behavior of a home climate control system thermostat. In this simple example, we chose not to include more advanced concepts such as user-programmable thermostat control or "dead zones" in heater/air conditioner control.

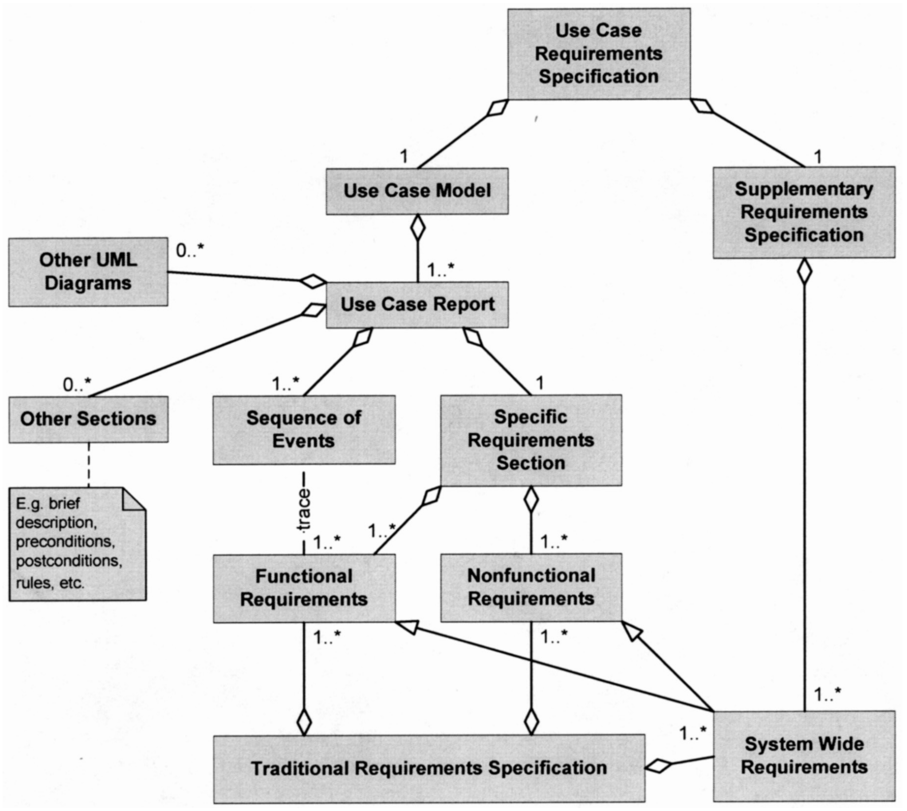

Use Case Requirements Specification
Use Case Model
Supplementary Requirements Specification
Other UML Diagrams
Use Case Report
Other Sections
Sequence of Events
Specific Requirements Section
E.g. brief description, preconditions, postconditions, rules, etc.
«trace»
Functional Requirements
Nonfunctional Requirements
Traditional Requirements Specification
System Wide Requirements
1
1
0..*
1..*
0..*
1..*
1
1..*
1..*
1..*
1..*
1..*
1..*
1..*

Figure 1. An abstract UML class diagram illustrating the hybrid process for requirements development. Lines with arrowheads represent generalization relationships and lines with diamond heads represent aggregation relationships.

7.2.1. Use Case Report: Regulate Room Temperature

Use Case Diagram: Figure 2

Use Case Name: Regulate Room Temperature

Brief description: This use case describes the steady state behavior of the thermostat controller for a home climate control system. The goal of this use case is to maintain the room temperature (roomTemp) between lowerThreshold (see Rule1) and upperThreshold (see Rule2) degrees Fahrenheit using a Heater and an Air Conditioner, which are also part of the home climate control system.

Added value: Home Owner controls the temperature of the house.

Scope: A typical Tucson family house

Primary actor: Home Owner

Supporting actors: Heater, Air Conditioner, Ambient Room Air

Preconditions:

- An upperThreshold and lowerThreshold are defined in the system.
- The roomTemp is between the lowerThreshold and the upperThreshold.
- The lowerThreshold is less than the upperThreshold.

Main Success Scenario: Hot Day

* [continuous] The system displays the current room temperature, in degrees Fahrenheit, to the user.

1. The Home Owner turns on the thermostat.
2. The system samples the Ambient Room Air temperature every 5 seconds.

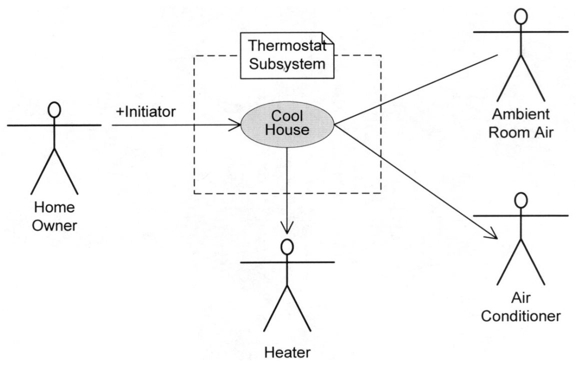


Figure 2. Simple use case diagram for the HVAC example.

3a. The system determines that the roomTemp exceeds upperThreshold.
4. The system turns on the Air Conditioner.
5. The system determines that the roomTemp drops below upperThreshold.
6. The system turns off the Air Conditioner [repeat at 2].

Alternate Flow 1: Cold Day

3b. The roomTemp drops below lowerThreshold.
3b1. The system turns on the Heater.
3b2. The roomTemp exceeds lowerThreshold.
3b3. The system turns off the Heater [repeat at 2].

Alternate Flow 2: Change upper and lower temperature thresholds

At any time during the use case sequence of events, the Home Owner may change the upper and lower temperature thresholds, subject to Rule3 and Rule5.

Rules:

Rule1: lowerThreshold default value is 70°F.
Rule2: upperThreshold default value is 73°F.
Rule3: The lowerThreshold must be less than the upperThreshold.
Rule4: The air temperature sampling frequency is fixed at 5 seconds.
Rule5: upperThreshold and lowerThreshold are adjustable to 0.5°F increments.

Author: Terry Bahill

Date: January 1, 2002

Specific Requirements

Functional Requirements:

- Req. F1: The system shall display the current room temperature, in degrees Fahrenheit. [*From the continuous step at the beginning of the sequence of events]*
- Req. F2: The system shall sample the ambient room temperature (roomTemp) in degrees Fahrenheit. [*From step 2*]
- Req. F3: The system shall be able to compare the sampled room temperature to a stored threshold temperature given in degrees Fahrenheit. [*From steps 3a and 3b*]
- Req. F4: The system shall turn on the Air Conditioner when the roomTemp exceeds the upperThreshold temperature. [*From step 4*]
- Req. F5: The system shall turn on the Heater when the roomTemp drops below the lowerThreshold. [*From step 3b1*]
- Req. F6: The system shall store user-defined values for the upperThreshold and lowerThreshold temperatures in degrees Fahrenheit. [*Derived From Alternate Flow 2*]
- Req. F7: The system shall provide a means for the user to select the upper and lower temperature thresholds [*From Alternate Flow 2*]
- Req: F8: The system shall incorporate an operational delay capability in which the Air Conditioner or Heater must remain on for at least 5 minutes after the upperThreshold or lowerThreshold is crossed. [*Derived requirement based on experience with similar systems. If this requirement is not imposed, the system will wear itself out*]

Nonfunctional Requirements:

- Req. N1: The system shall sample the room temperature every 5 seconds. [*From step 2 and Rule4*]
- Req. N2: It shall take less than 0.1 seconds for the system to sample the room temperature. [*Obtained from stakeholder interviews*]
- Req. N3: The system shall take no longer than one second to turn on the Heater or Air Conditioner when required. [*Obtained from stakeholder interviews*]
- Req. N4: The system shall be capable of sensing a temperature range from negative 20 to 140°F. [*Obtained from stakeholder interviews*]
- Req. N5: The room temperature value displayed by the system shall be visible to a user with 20/20 vision standing 5 feet from the thermostat unit in a room with an luminance level between 10 and 100 foot-candles. [*From the continuous requirement to display the room temperature, and from stakeholder interviews*]
- Req. N6: The user-defined values for lowerThreshold and upperThreshold shall be specifiable in the range of –20 to 140°F. [*Obtained from stakeholder interviews*]
- Req N7: The default lowerThreshold shall be 70°F. [*From Rule1*]
- Req N8: The default upperThreshold shall be 73°F. [*From Rule2*]
- Req N9: The lowerThreshold shall always be less than the upperThreshold. [*From Rule3*]
- Req N10: The upperThreshold and lowerThreshold shall be adjustable to 0.5°F increments. [*From Rule5*]

Note that some of the requirements, most commonly those in the nonfunctional requirements section, are not derived directly from the use case text. Some nonfunctional requirements do not describe behavior, and consequently these requirements will not normally show up in the use case description, as use cases are tools to describe the system behavior. Many times nonfunctional requirements are derived from legacy systems (if available) and interviews with designers and users.

Requirements that are directly derived from the use case text should be traced to the step or section from which they were derived. This traceability is nicely managed with a good requirements management tool. The example here is simple when compared to a real use case report for a real system. The functional and nonfunctional requirements given in the Specific Requirements Section may be further categorized if it makes them easier to understand. For example, we might want to group all performance requirements under a "Performance Requirements" section, etc. We have also included a "Rules" section in this use case report to hold business rule information. A business rule is a kind of nonfunctional requirement that places specific constraints on how a system or process should perform [IBM RUP, 2003].

Requirements that do not pertain directly to this use case should be documented in the Supplementary Requirements Specification. In this example, requirements pertaining to characteristics such as the thermostat's dimensions, weight, color, user interface layout, wall mounting concept, shock resistance (if dropped, for instance), reliability/availability, and power consumption should be included in the Supplementary Requirements Specification because they are not restricted to the Regulate Temperature use case.

8. WHAT IF WE ARE REQUIRED TO DEVELOP A TRADITIONAL REQUIREMENTS SPECIFICATION?

Sometimes, especially in the defense or nuclear industry, the customer dictates that a traditional requirements specification be developed. In these communities, the use case approach is often seen as too loose for measuring system success. No worries—the information captured with the hybrid process can be readily used to populate a traditional requirements specification. As mentioned earlier, functional requirements can be extracted from a use case's sequence of events to form shall-statement requirements. Once the functional requirements have been extracted, they may be documented in the Special Requirements section of the use case from which they were derived, or they may be documented in the Supplementary Specification if they apply to more than one use case. A traditional requirements specification can be mechanically generated by taking the union of the Specific Requirements in all of the use case reports (to get the nonfunctional and functional requirements documented there), plus the Supplementary Requirements Specification (to pick up the remaining requirements).

The resulting requirements specification has a good chance of accurately reflecting the correct requirements, since they were largely derived from actor-driven use cases. It is important to note that when documented separately from the use case from which they were derived, each requirement should be traceable back to the originating use case. This provides a way to return to the source of the requirement to better understand its context. It is possible to write the use case sequence of events narrative using shall-statement verbiage and skip the extraction step, but experience shows that this is awkward and detracts from the readability of the use case narrative. Therefore, we recommend sticking with natural language for the sequence of events, and leave the shall-statement requirements to the Specific Requirements section of a use case report and the Supplementary Requirements Specification document.

9. WHAT IF THE STAKEHOLDER GIVES YOU A REQUIREMENTS SPECIFICATION?

In some cases, the systems engineer is handed a traditional requirements specification by the customer. The question is: Should use cases still be generated for this system? Our general recommendation is yes, but not necessarily for requirements gathering purposes, depending on the detail and quality of the provided requirement specification. A use case model can still provide valuable information to better understand the system under design:

1. A use case model can serve as a Concept of Operations for the system. This CONOPS helps engineers and stakeholders understand and agree on the operation of the system without diving into the detailed requirements specification.
2. A use case model can be used to validate the requirements given by the customer to ensure that the requirements really support the actors. This helps ensure that the customer is asking for what they really need. In our experience, customers appreciate this analysis and feedback.
3. A use case model can be used to discover shortcomings in the stakeholder-generated requirements. Maybe the stakeholder forgot about an important actor, or system administration or maintenance requirements? This situation has been observed in practice, and the results were used as discriminators during proposal work.
4. A use case model can be used to abstract to the right level in cases where the customer's requirements specification contains a mix of system and

lower level or design requirements (which often occurs). Applying use cases allows systems engineers to focus on requirements at the right level, saving the lower-level requirements to be addressed at the appropriate level of abstraction.

The actual customer requirements along with the way in which the requirements are organized in the customer's documentation can be used to determine which use cases are needed. Once developed, the customer requirements should be mapped to the Specific Requirements section of the use cases, or placed in the Supplementary Requirements Specification as appropriate. This mapping exercise will ensure that the developed use cases capture the behavior of the system as intended by the customer. It will also help identify shortfalls in a clear way. If there is behavior that the systems engineer believes should be part of a use case and there are no customer requirements to support it, then there is a potential shortcoming in the customer's requirements, and this should be discussed during a requirements review with the stakeholders.

It is hard to say whether it is worth it to generate a use case model in the case where the customer's requirements specification is robust, and is accompanied by a Concept of Operations. We almost always generate a use case model anyway. The problem is that the Concept of Operations is many times not based on system behaviors and is typically written at a higher level, such as mission areas. A use case model can be used as a bridge between the customer's Concept of Operations and their requirements. This will ensure consistency and put all of this information into a form that is familiar to the development team.

Depending on the level of detail given in the customer's specifications, a use case model may still be developed by the systems engineers or implementation engineers to describe the behavior of subsystems and further refine the requirements [Booch et al., 1999].

10. HIERARCHICAL USE CASE MODELS

This section has particular importance to engineers involved in specifying very large, complex systems such as those encountered in military applications. This section is a bit of a departure from the overall theme of this paper, but we feel that it is important when it is necessary to specify requirements for complex, multi-subsystem projects. The hierarchical use case modeling concepts described here are presented by Jacobson and coworkers [Jacobson, Ericsson, and Jacobson, 1995; Jacobson, Griss, and Jonsson, 1997; Jacobson, 2000a, 2000b] and are briefly described in this paper to show how the use case modeling technique (and thus requirements gathering) scales for large systems. It would easily be possible to devote an entire paper to this topic. The authors are still elaborating on and validating the techniques in this section through experience on actual projects.

For large-scale systems it is common for the use case model to be expressed at multiple levels, where a use case model is developed for more than one perspective or business area in an effort to manage complexity. In this section, two different, but related concepts involving use case hierarchies will be explored. The first deals with modeling the business or enterprise in which the system operates (i.e., actually modeling the actors of the system, and even touching on the actor's actors!), and the second deals with use case models at multiple levels where a given use case model is broken down into more manageable parts. The first concept is a shift in perspective from system modeling to business (enterprise) modeling, while the second concept describes how a given use case model (system or business) scales to manage complexity.

10.1. Business Modeling

Many sources, including the authors, strongly advocate the development of a business use case model in order to better understand the context in which the system is to be deployed [Jacobson, Ericsson, and Jacobson, 1995; Jacobson, Booch, and Rumbaugh, 1999; IBM RUP, 2003, Cockburn, 2001]. A business use case model takes a step outside of the system under design and employs the use case modeling techniques to model the context (business) in which the system will operate. Therefore, along with specifying the system, the use case technique can be applied as a powerful tool for specifying the business that uses the system. In fact, as advocated by Ivar Jacobson in his many publications and books, the very same use case modeling and object oriented analysis techniques used for system specification can be readily applied towards specifying business processes through a shift of focus where the "system" being described is actually the business in which the system will be deployed. The resulting business use case model is then analyzed in a straightforward and elegant way to determine use cases (requirements) for information systems[2] built to help automate those processes [Jacobson, Ericsson, and Jacobson, 1995; Jacobson, Booch, and Rumbaugh, 1999, IBM RUP 2003]. The shift in perspective from describing an information

[2]The system we are interested in building is only one of possibly many systems used at this level of analysis.

system to describing the context of the system represents the first type of use case modeling hierarchy.

Within a complete enterprise architecture development process [e.g., IBM RUP, 2003; Ronin International Enterprise Unified Process, 2003] a business use case model is developed and exploited by analyzing the interplay of the system under design with other systems and humans to determine the business behaviors, and subsequently find *the correct* use cases for the system to be built. Then, a use case model of the system is developed to further describe how it specifically provides the value alluded to in the business use case model. A system use case model, in turn, is analyzed to model the interplay between subsystems to accomplish the system behaviors and subsequently determine a set of requirements on each subsystem. It all fits together in a very nice framework, using the same set of analysis tools. Gomaa [2000], Friedenthal, Lykins, and Meilich [2000], and Jacobson [2000b] provide practical guidance for analyzing use cases to determine the subsystems for complex, distributed systems. A business model provides context for the system requirements by linking them directly to implementation-independent business processes as captured in the business use cases. A business model also helps drive out a robust set of system use case model actors.

10.2. Superordinate and Subordinate Use Case Models

The second type of hierarchy involves developing use case models at different levels for the same subject, specifically through a *superordinate use case model* and a *subordinate use case model*. The development of these types of use case hierarchies are useful when a “flat” use case model is not enough to fully specify all of the functional requirements without becoming too large to be easily understood. To apply this type of hierarchy, a superordinate use case model is built for the top-level system (or system-of-systems), which specifies just enough of the requirements to derive an appropriate subsystem structure and interfaces between the subsystems, which are more manageable. Then a subordinate use case model is developed for each of the resulting subsystems. It is called a subordinate use case model because it is derived directly from the superordinate model. When taken together, the subordinate use case models fill out the details left out of the superordinate model to completely specify the requirements for the overall system-of-systems.

In this way the use case modeling techniques can be considered “fractal,” or recursive in that they can be applied at varying levels of abstraction to specify use cases at a system-of-systems (superordinate) level down to each individual (subordinate) system level. This type of scalability is essential in complex system development. Booch et al. [1999], Jacobson [2000a, 2000b], and Jacobson, Griss, and Jonsson [1997] give pioneering discussions on applying use cases with respect to large-scale systems and reflect on the recursive application of use cases in terms of superordinate and subordinate models. The Systems of Interconnected Systems pattern [Jacobson, 2000a; Jacobson, Griss, and Jonsson 1997] more completely discusses the concepts mentioned here. The IBM RUP SE [2003] presents a use case flow down method introduced by Cantor [2003a, 2003b] that compliments the superordinate/subordinate use case model concepts elegantly. With the use case flow down method, requirements from use cases developed at a higher (superordinate) level of abstraction are flowed down and allocated to the subordinate systems.

The business modeling and subordinate/superordinate hierarchical use case modeling techniques introduced in this section help align business analysts, systems engineers, and software engineers by applying a common modeling methodology to specify systems at all levels of abstraction, including the system context itself. Using a common modeling paradigm at all levels of abstraction will also facilitate smoother traceability between the models so that information in one model does not have to be translated into another modeling style in order to be understood. Also note that developing hierarchical use case models is not analogous to performing functional decomposition on the system. With hierarchical use case models, the intent is still to extract requirements for the system. Functional decomposition is often taken too far and leads to premature design. This is a delicate distinction.

The requirement derivation and documentation techniques described in this paper apply to use case models at any level of a use case model hierarchy. For hierarchical models, however, a strong emphasis should be placed on maintaining the traceability of requirements and use cases expressed in the models. Finally, tradeoffs should be exercised to ensure that developing hierarchical models are worth the cost to generate them. In fact, Jacobson [2000b] suggests that even for large, complex systems-of-systems only one division (two hierarchical levels) of the use case models are usually necessary. That is, one superordinate use case model, and a set of subordinate use case models. In any case, one level of use case modeling is many times not enough to efficiently capture the intricate requirements of modern distributed information systems.

11. VERIFICATION TESTING AND USE CASES

Another question that arises when applying use cases to general systems engineering is: "If we generate use cases, what do we use for verification testing?" Many authors, including us, believe that use cases, written as described in this paper, serve as an excellent start for a system-level verification test development. As part of the systems engineering process, a test plan is commonly written to outline the tests that will be performed to demonstrate that the system complies with its requirements. The test plan typically references one or more test scripts that detail step-by-step scenarios that, when successfully performed as scripted, demonstrate how the system satisfies the requirements. The union of the test scripts should be sufficient to demonstrate all of the required behavior of the system.

The use case model partitions the behavior of the system into discrete use cases. The structure of the use case model can be re-used to define the structure of the test plan by including a test for each use case in the use case model. Use cases provide a good view to the testers early enough to adequately plan the test effort. Basing the test plan on the use case model ensures that the system tests will cover all of the functionality of the system. If the concepts described in this paper are used, there is a strong traceability between the use cases and the requirements. Therefore, by structuring the test plan in accordance with the use case model, each requirement should be covered. Each use case's Brief Description, Preconditions, and Postconditions should be incorporated into the test plan to explain each test and the expected results.

To get deeper into the test development, one or more verification test scripts are generated for each test described in the test plan. A test script is typically a very detailed, keystroke-by-keystroke sequence of steps that are performed to confirm that the system performs as specified in the requirements. A use case report can be thought of as an ideal test scenario and a starting point from which the more detailed test scripts can be developed. The use case report describes in a step-by-step fashion how the system should behave to fulfill the intent of the use case. The use case report will not (should not!) include the level of detail required for a test script. A test script is written after the system has been designed and implemented; whereas a use case is a requirements artifact, largely written before the system is even designed (it will likely be updated throughout the design and implementation phases [IBM RUP, 2003]).

For simple transaction-based systems, generating a test script from a use case report is a fairly mechanical process. In other cases much preparation must be done in order to get to the test script steps. However, even in these instances the use cases can be used to help focus this effort. To the extent possible, a procedure for generating the test script can proceed as follows: Start with the use case sequence of events and augment the steps with the individual user button presses, keystrokes, and other system details to fill out the required information for the test script. Since the requirements were derived from the use case, the requirements that are to be verified by the test script are already called out in the Specific Requirements section of the use case report, so it is clear which requirements are being verified. This mapping is used to develop a report indicating which requirements were satisfied by which test scripts. Once the test results have been generated a complete mapping from test result to the highest level requirements can be attained.

12. THE MISSING LINK

In a traditional systems engineering environment, *Test Procedures* are detailed and sequential. Sometimes they look like a use case sequence of events. The *Requirements Specification* may have hundreds of requirements and it paints a picture of the system being designed. It is usually written before the test procedures, but it might not bear a close resemblance. The *Design Model* captures the architecture and the interfaces. What do these three views of the system have in common? In our hybrid process, they have a common ancestor: the Use Case Requirements Specification, as shown in Figure 3.

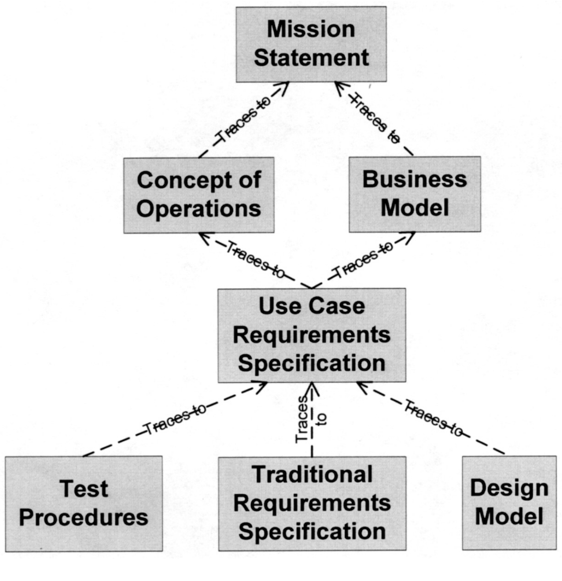


Figure 3. Ancestry model.

A human, a chimpanzee, and an orangutan have a common ancestor. Archeologists studying primate evolution have never seen this missing link, but they have a good idea of what it would look like. Tracing ancestors has been a big research effort over the last few centuries. It evidentially is a useful scientific method. Continuing with this analogy, the Use Case Requirements Specification is the missing link in the traditional systems engineering process.

13. MISSILE EXAMPLE

Use case requirements specifications and traditional requirements specifications offer orthogonal redundancy that can help engineers detect mistakes. Consider the fragment from a traditional requirements specification for a missile printed circuit board (PCB) shown in Table III. These requirements were read and transformed into the derived requirements shown in Table IV.

Type P means the trace delivers power. Power traces are 12 mm wide and signal traces are 4 mm wide. As you can see, someone failed to denote trace-11 as a power trace in the table. No one redid the requirements to table translation and Testing only tested one valve at a time, so the error was not detected until the flight test, when the trace burned out, and the missile failed.

We do not suggest that someone should have repeated the task of transforming the customer requirements into the derived requirements: that would not add value. But we think a use case would add value and it would help catch the mistake. Consider the following use case.

Name: Control Missile Attitude

Iteration: 1.0

Brief description: Valves 1–6 open and close, allowing pneumatic fluid to flow through their orifices, thereby controlling missile attitude.

Added value: Guidance, Navigation, and Control (GNC) can control the system state.

Level: Very low

Table III. Customer Requirements for a PCB

PCB trace number	Requirement
10	Trace-10 shall transmit signals between element 1 and 2. Blah, blah, blah, etc.
11	Trace-11 shall supply valves 1 through 6. Blah, blah, blah, etc. This trace shall have a capacity of X5 mA. Blah, blah, blah.
12	Trace-12 shall deliver power to components 1 through 4. Blah, blah, blah, etc.

Table IV. Derived Requirements for PCB

PCB trace number	Origin	Destination	Type	Length
10	Stuff	Stuff		Stuff
11	Pin-3 IC-2	Pin 4 of Valves 1 to 6		10
12	Stuff	Stuff	P	Stuff

Scope: Control of attitude control valves

Primary actor: GNC

Supporting actors: Power Supply, Pneumatic Fluid, Attitude Control Valves

Frequency: On the order of 1 Hz

Precondition: The missile is flying, and all valves are closed.

Trigger: GNC commands a course change.

Main Success Scenario:

1a. GNC commands Valve-1 to open for X1 ms.
2. Valve-1 opens, drawing X2 mA from the power supply through printed circuit board (PCB) trace-11, allowing pneumatic fluid to flow through its orifice.
3. After X1 ms Valve-1 closes, drawing X3 mA, thereby stopping the flow of fluid [exit use case].

Alternate Flows:

1b. GNC commands Valve-2 to open for X4 ms.
1b1. Valve-2 opens, drawing X2 mA from the power supply through PCB trace-11, allowing pneumatic fluid to flow through its orifice.
1b2. After X4 ms Valve-2 closes, drawing X3 mA, thereby stopping the flow of fluid [exit use case].
[Similar alternative flows exist for Valves 3–6.]

Postcondition: Missile has changed course and all valves are closed.

Rules:

1. Valves 1–6 will open and close at the same or at different times.

Specific Requirements

Nonfunctional requirements:

Req. N1: PCB Trace-11 is a power trace. It shall have a capacity of X5 mA, because each of the six valves may open or close at the same time. [*From steps 2 and 3 and Rule 1*]

Req. N2: Maximum pneumatic fluid flow rate shall be X6 kg/s. [*Obtained from Stakeholder interviews*]

Req. N3: Valve opening and closing times shall be less than 1 ms. [*Obtained from Stakeholder interviews*]

Functional Requirements:

Req. F1: GNC shall command valves 1–6 to open. [*From step 1*]

Req. F2: Valves 1–6 shall open and close on command. [*From steps 2 and 3*]
Req. F3: Valves 1–6 shall regulate the flow of pneumatic fluid through them. [*From step 2*]
Author/owner: Terry Bahill
Date: April 1, 2004

As previously stated, Use Case Requirements Specifications and traditional requirements specifications are orthogonal, not redundant. The traditional requirements specification concentrated on the printed circuit board traces, whereas the use case describes the functional behavior of the missile. We postulate that the existence of this use case would have prompted engineers to see the mistake in the derived requirements table.

Applying use cases might help systems engineers discover conflicts in the requirements. For example, consider these two requirements for an automobile. (1) The vehicle shall accelerate from 0 to 60 mph in less than 9.5 s. (2) The vehicle shall tow a 3000 lb trailer at highway speeds (65 mph). The vehicle does not have to satisfy these two requirements simultaneously: It would be possible, but expensive, to build a vehicle that did so. If these two requirements existed far apart in a traditional requirements specification, it would be difficult to discover this interaction. However, in a Use Case Requirements Specification, these two requirements would appear in two different use cases; therefore, it would be obvious that they need not be satisfied simultaneously. Requirements that must be satisfied for all use cases are put in the Supplementary Requirements Specification, not in the Specific Requirements section of individual use cases.

14. SUMMARY

This paper has shown that use case models and traditional shall-statement requirements are synergistic specification techniques that should be employed in a complimentary fashion to best communicate and document requirements. Are use cases requirements? Not exactly—they are a vehicle to discover requirements. The requirements are actually embedded within the use case's textual description, making use cases a container for the requirements. How do use cases relate to a traditional requirements specification? Use cases provide context for requirements that are documented using shall-statement notation in a traditional requirements spec. These shall-statement requirements can be extracted from a use case's narrative. Where are the requirements actually documented? We suggest that the requirements be documented using the requirements specification structure proposed in this paper, which conforms to the specification presented in Leffingwell and Widrig [2000] and the IBM RUP [2003]. This specification includes (1) a use case model for capturing requirements that are associated with individual use cases and (2) a Supplementary Requirements Specification for capturing system-wide requirements. The specific contribution of this paper is the introduction of a Functional Requirements Segment to the Specific Requirements Section. This Functional Requirements Segment contains the functional requirements written in formal shall statement language. This paper has shown two examples illustrating our hybrid process for combining use case models with traditional shall-statement requirements. Hierarchical use case models are common in complex, distributed system design and deployment. The hybrid process that combines use case modeling and shall-statement requirements also applies to hierarchical models. This paper also briefly described the utility of use cases in the testing and verification disciplines of system design.

ACKNOWLEDGMENTS

We thank Rick Botta of BAE Systems for supporting this work and Sandy Friedenthal of Lockheed-Martin, Alexander Kossiakoff of Applied Physic Laboratory, and Rick Steiner, Joan Koesterherm, T.J. Theodore, Joseph DelGaudio, and Bill Marksteiner of Raytheon for astute comments on the manuscript. This paper was supported by AFOSR/MURI - F49620-03-1-0377.

REFERENCES

S. Adolph and P. Bramble, Patterns for effective use cases, Addison-Wesley, Reading, MA, 2003.

F. Armour and G. Miller, Advanced use case modeling, Addison-Wesley, Reading, MA, 2001.

A.T. Bahill and J. Daniels, Using object-oriented and UML tools for hardware design: A case study, Syst Eng 6(1) (2003), 28–48.

A.T. Bahill and F.F. Dean, "Discovering system requirements," Handbook of systems engineering and management, A.P. Sage and W.B. Rouse (Editors), Wiley, New York, 1999, pp. 175–220.

G. Booch, I. Jacobson, and J. Rumbaugh, The Unified Modeling Language user guide, Addison-Wesley, Reading, MA, 1999.

M. Cantor, Rational Unified Process for Systems Engineering Part 1: Introducing RUP SE Version 2.0, http://www.therationaledge.com/content/aug_03/f_rupse_mc.jsp, August 2003.

M. Cantor, Rational Unified Process for Systems Engineering Part II: System architecture, http://www.therationaledge.com/content/sep_03/m_systemarch_mc.jsp, September 2003.

A. Cockburn, Writing effective use cases, Addison-Wesley, Reading, MA, 2001.

R. Davis and B.G. Buchanan, "Meta-level knowledge," Rule-based expert systems, the MYCIN experiments of the Stanford Heuristic Programming Project, B.G. Buchanan and E.H. Shortliffe (Editors), Addison-Wesley, Reading, MA, 1984, pp. 507–530.

S. Friedenthal, H. Lykins, and A. Meilich, Adapting UML for an Object Oriented Systems Engineering Method (OOSEM), http://www.omg.org/cgi-bin/doc?syseng/2001-09-05, 2000.

H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, Addison-Wesley, Reading, MA, 2000.

R. Grady, Practical software metrics for project management and process improvement, Prentice-Hall, Englewood Cliffs, NJ, 1992.

IBM RUP, IBM Rational Unified Process, http://www-306.ibm.com/software/awdtools/rup/, 2003.

IBM RUP SE, IBM Rational Unified Process for Systems Engineering, available on the Rational Developer Network, www.rational.net (account required), 2003.

I. Jacobson, Systems of interconnected systems, Road to the Unified Process, Cambridge University Press, Cambridge, 2000a.

I. Jacobson, Use cases in large-scale systems, Road to the Unified Process, Cambridge, University Press, Cambridge, 2000b.

I. Jacobson, M. Griss, and P. Jonsson, Software reuse: Architecture, process and organization for business success, Addison-Wesley, Reading, MA, 1997.

I. Jacobson, M. Ericsson, and A. Jacobson, The object advantage: business process reengineering with object technology, Addison-Wesley, Reading, MA, 1995.

I. Jacobson, G. Booch, and J. Rumbaugh, The unified software development process, Addison-Wesley, Reading, MA, 1999.

D. Kulak and E. Guiney, Use cases: Requirements in context, Addison-Wesley, Reading, MA, 2000.

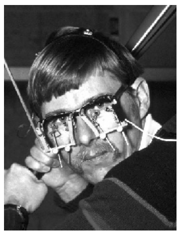
D. Leffingwell and D. Widrig, Managing software requirements, Addison-Wesley, Reading, MA, 2000.

Object Management Group: Unified modeling language: superstructure version 2.0 final adopted specification, http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf, 2003.

Ronin International, Enterprise Unified Process (EUP), http://www.enterpriseunifiedprocess.info/, 2003.

R. Young, Criteria of a good requirement, http://www.ralphyoung.net/artifacts/CriteriaGoodRequirement.pdf, 2004.

Jesse Daniels is a systems engineer with BAE Systems in San Diego. He earned a B.S. and an M.S. in Systems Engineering in 1999 and 2000 from the Department of Systems and Industrial Engineering at the University of Arizona.



Terry Bahill has been a Professor of Systems Engineering at the University of Arizona in Tucson since 1984. He received his Ph.D. in Electrical Engineering and Computer Science from the University of California, Berkeley, in 1975. He holds U.S. Pat. No. 5,118,102 for the Bat Chooser, a system that computes the Ideal Bat Weight for individual baseball and softball batters. He is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) and of INCOSE. He is the Founding Chair Emeritus of the INCOSE Fellows Selection Committee. This picture of him is in the Baseball Hall of Fame's exhibition *Baseball As America*.