Use Case Modeling Guidelines

Donald G. Firesmith
FiresmithD@AOL.com

1. Abstract

Use case modeling has become the foundation of the most popular de facto standard technique for performing software requirements analysis and specification. However, use case modeling has its well-known problems, and different requirements engineers typically perform use case modeling differently. This paper provides a hierarchically organized set of detailed guidelines for use case modeling.

2. Introduction

Over the last decade, use cases have become the foundation of the most popular de facto standard technique for software requirements analysis and specification within the object community. However, use case modeling has numerous well-known problems [1] [2] [3] [4]. It is inherently more functional than object-oriented, leading to a significant chasm and paradigm shift between requirements engineering and OO modeling. Putting the use case approach into practice also often illuminates problems that are not addressed in books and most articles on use cases, and different requirements engineers typically perform use case modeling differently.

This paper provides a hierarchically organized set of detailed guidelines for use case modeling. I have collected these guidelines from numerous real projects during the last six years and recently refined them using lessons learned while creating the software requirements specification for a large, embedded, distributed real-time system (the control software for an automated digital tape library).

3. Definitions

The following terms clarify use case modeling and the following guidelines:

- An **external [object or class]** (a.k.a., actor) is any relevant object or class that is external to the *application* (whether system or software) and interfaces (either directly or indirectly) with it. Externals can be stereotyped as [human] actors, data repositories, hardware externals, and software externals. For example, a digital thermostat *system* may have only a single external: its user. However, the *software* controlling the digital thermostat may have the following externals, many of which are internal to the system: actual temperature display, actual temperature sensor, desired temperature display, decrement actual temperature button, decrement desired temperature button, increment actual temperature button, increment desired temperature button, on/off button, status display, and user.
- A **use case** is a general way that an external uses a business or application to achieve some goal. A use case is primarily defined in terms of interactions between the business or application and its associated externals. Note that a use case is neither an operation of an external nor a business or application operation (function). Rather, a use case is typically that part of an external's operation that involves the business or application. For example, a simple digital thermostat may have the following use cases: "Change the state of the digital thermostat", "Change the desired temperature of the room", and "Control the actual temperature of the room".

  - An **essential use case** is a pure requirements-level use case that does not contain any unnecessary design constraints (e.g., GUI design details).

- A **use case path** (a.k.a., course) is a contiguous set of referential relationships traversed by the interactions of the use case. Because a single use case will typically have an indefinitely large number of paths that traverse it due to branching and looping, only a minimal basis set of paths is typically identified and analyzed. Each path is stereotyped as either a **normal path** (i.e., it achieves the underlying goal of the use case) or an **exceptional path** (i.e., it does not). For example, the normal paths through the "Change the desired temperature of the room" use case may be "Decrement the desired temperature" and "Increment the desired temperature". The exceptional paths of the "Control the actual temperature of the room" use case may include "Handle temperature sensor failure".
- A **usage scenario** is a single set of contiguous interactions that traverses a use case path. Whereas a use case is very general and a use case path is more specific, a usage scenario is totally specific. The large collection of usage scenarios that traverse the same path form an equivalence class with regard to software testing. For example, the following uniquely identifies a usage scenario: Increment the desired temperature when the digital thermostat is in the on state, the current desired temperature is 68°F, and the actual temperature is 69°F.
- A **context diagram** is a kind of static diagram that documents the application, externals, and the important relationships between them.
- A **use case diagram** is a kind of dynamic diagram that documents blackbox use cases, externals, and the important relationships between them.
- A **sequence diagram** (a.k.a., timing diagram) is a kind of interaction diagram that documents the dynamic behavior of objects in terms of a sequence of interactions between them.
  - A **blackbox sequence diagram** is one that treats the application as a blackbox interacting with externals. Interactions on blackbox sequence diagrams are typically specified using narrative English rather than as programming language messages between internal software objects.

4. Use Case Modeling Guidelines

The following guidelines for performing use case driven requirements analysis and specification, use case driven scheduling, and use case driven testing have been proven by experience to be cost-effective. They fall into the following categories:

- General Guidelines
- Modeling Languages and Tools
- Modeling Externals
- Modeling Use Cases
- Modeling Use Case Paths

4.1 General Guidelines

- **Training**. Provide initial classroom training for all relevant personnel (requirements engineers, marketing, domain experts, customers, developers, and testers) who will either develop, inspect, or read the use case section of the requirements specification. Provide ongoing on-the-job training for all members of the requirements team including analysts, domain experts, and testers. *Rationale*: Use case modeling is still

relatively new to most people. Requirements analysts who are new to use case modeling tend to produce many defects in the use case model and resulting requirements specification. Developers, who are used to traditional requirements specifications, also often have difficulty interpreting and understanding use case-oriented specifications, especially items intended for others (e.g., the use case categorization that is used by independent testers to prioritize testing).

- **Operational requirements only**. Use a use case model to analyze and specify the operational (a.k.a., functional) requirements of an application. Do not use case modeling for specifying quality requirements (e.g., extensibility, operational availability, portability, reliability, and reusability). *Rationale*: Use cases provide a powerful, industry-standard technique for analyzing and specifying operational requirements in a user-centered manner. However, quality requirements cannot be reasonably stated in terms of interactions.
- **Avoid use case driven design**. Do not drive the architecture or design from the structure of the use cases. Instead, use domain experts and object modeling to identify the key business abstractions. Use externals and the information passed with the interactions of the use cases to identify additional classes of objects. *Rationale*: Use cases are functional abstractions that are often functionally decomposed. Thus, use case driven design often results in a functional decomposition design based on god-like controller objects violating the encapsulation of dumb data objects.
- **Design verification**. Verify the object-oriented design against the use case model by tracing paths through the design. *Rationale*: The design must implement the requirements captured in the use case model.
- **Requirements validation**. Use the technique of use case driven testing to validate the implemented application against the use case model. *Rationale*: Because use cases capture the operational requirements, functional test cases can be chosen to exercise use case paths.
- **Use an iterative, incremental, parallel development cycle**. Iterate the use case model during the course of development. Develop the use case model in a series of increments associated with the builds. Develop the use case model in parallel with design, development, and testing. *Rationale*: The requirements are never known completely and correctly at the start of requirements elicitation, analysis, and specification. Domain experts and users are much better at identifying what is wrong or incomplete with a partial model than specifying it perfectly up front. The use case model of any nontrivial application is too large to develop efficiently in a single phase. Incremental development allows the associated application to be incrementally developed and tested. It also allows for course corrections that insure that the delivered application is not obsolete as soon as it is released. This guideline results in higher productivity and user (e.g., designer, tester) satisfaction. Design, coding, and testing against the use case model will identify defects and holes, resulting in a higher quality.
- **Schedule later builds by use cases and paths**. Although the initial builds should provide a foundation of core classes that capture an object-oriented architectural framework, later builds and releases should be scheduled in terms of use cases and paths in the use case model. *Rationale*: Schedules based on use cases and use case paths provide incremental value to the users (actors) and provide testable increments to the independent test team. However, scheduling all builds on use cases without an overriding architectural vision tends to produce a functional design that requires significantly more iteration than is necessary.

4.2 Modeling Languages and Tools

- **Standard modeling language**. Use a single industry standard modeling language to document the use case model. Start with UML, but extend it with the latest approved version of the OPEN Modeling Language (OML). *Rationale*: A standard modeling language promotes communication among developers. It also promotes increased productivity and model quality. This extension of the Unified Modeling Language (UML) is easier to learn, more expressive and intuitive than vanilla UML. It also better supports different kinds of externals, relationships between use cases, and logic.
- **CASE tools**. Use an upperCASE tool that supports all necessary aspects of the chosen modeling language to electronically capture the results of use case modeling. But do not let the choice of tool drive the choice of modeling language. *Rationale*: This guideline ensures that the actually tool does what is intended.

4.3 Modeling Externals

- **Clearly differentiate the boundary of the use case model**. Decide whether one is modeling a business, a system application, or a software application. Be consistent. Label the boundary on relevant diagrams. *Rationale*: Unless you clearly distinguish what you are modeling, it will be difficult to determine what the externals are. Because a system may contain software, hardware, paperware (documentation), and wetware (personnel), an external to a software application may be an internal part of a larger system.
- **Externals should be cohesive**. Externals should have a cohesive set of goals and responsibilities. Every external should not involve every use case. Use inheritance to factor out common external characteristics. *Rationale*: This guideline makes externals and their resulting use cases easier to understand.
- **Properly name the externals**. Provide a unique, meaningful name to each external. The name should be consistent with, and implied by, the associated definition and responsibilities. Watch out for unintended synonyms due to parallel development. *Rationale*: Remember that the name of a human actor need not be a job title, because the same person may play multiple roles when interacting with the application.
  Bad Example: Button pusher. (What the user of a digital thermostat does, not who the user is.)
  Better Example: User
- **Define the externals**. Provide a clear, concise, glossary definition of each external that is consistent with the name of the external and its responsibilities. *Rationale*: Because externals need not be job titles, they may be new concepts, even to domain experts. Because domain experts and requirements analysts often have different undocumented definitions for the same terms, formally defining the externals helps avoid confusion.
  Bad Example: The user is the person who presses the buttons on the digital thermostat.
  Good Example: The user is the actor who uses the digital thermostat to control the temperature of a room.
- **Specify external responsibilities**. Specify the responsibilities of each external. Where appropriate, include responsibilities for doing, knowing, and enforcing business rules. Specify responsibilities rather than specific tasks or operations that implement the responsibilities. *Rationale*: An external is largely defined in terms of its responsibilities. An external should have a cohesive set of responsibilities.
  Bad Example: The following "responsibilities" of the user of a digital thermostat are at too low of a level of detail:

- Turn on the digital thermostat.
- Turn off the digital thermostat
- Decrement the desired temperature of the room.
- Increment the desired temperature of the room.

Good Example: The following responsibilities are above the level of abstraction of an operation:

- Change the state of the digital thermostat.
- Change the desired temperature of the room.
- Observe the display.

- **Document external state behavior**. Document the state machines of any externals. *Rationale*: This guideline clarifies use case path preconditions and postconditions.
- **Specify actors rather than people or job descriptions**. *Rationale*: Remember that human actors are roles, not persons or job titles. A single person may play multiple roles when interacting with an application.
- **Do not specify requirements on externals.** *Rationale*: The use case model specifies requirements on the blackbox business or application rather than the externals with which it interacts.
  Bad Example: The user shall control the temperature of the digital thermostat.
  Good Example: The digital thermostat shall permit the user to set the desired temperature of the digital thermostat to any temperature between the minimum and maximum desired temperatures.
- **Document required actor expertise**. Specify the required expertise and training for all human actors. Different users of an application require different levels of training and expertise. *Rationale*: By specifying the required levels, training plans can be developed.
  Example: An average six year old should be able to successfully use the digital thermostat with minimal (i.e., 5 minutes of) training.
- **Organize by externals**. Organize the use case model into packages, first by external stereotype (e.g., actor, hardware, software) and then by specific external (e.g., customer, user). Within a package, use a standard sort order, either chronologically or alphabetically. Those subordinate use cases that are not primarily associated with any one external should be grouped into a package associated with the application. *Rationale*: This guideline provides a natural organization to the use case model that is both easy to use and user-centric.
- **Specify interfaces to externals**. Specify any required interfaces to externals, especially to existing hardware devices and software applications. *Rationale*: This guideline ensures that the designers have adequate requirements to implement the interface across which the interactions flow.
- **Document all relevant externals**. Document both externals that directly interface with the application as well as externals that indirectly interface via other externals *Rationale*: The object model often includes classes corresponding to both indirect externals and direct externals. Direct externals (e.g., keyboard, console) are sometimes less important than the indirect human actors that use them.
- **Summarize the context**. Use one or more context diagrams to summarize the static environment of the application in terms of its externals and the relationships between it and its externals. Also, document all significant relationships between externals on the context diagram. Provide narrative English description of the context diagram that

clarifies anything that is not totally obvious. Also, summarize the workflow between externals. *Rationale*: Context diagrams help one understand the context of an application. Relationships between externals often make the resulting context more understandable, especially inheritance relationships or associations when the order of use cases depends on interactions between externals. This guideline makes it easier for domain experts and others less familiar with this diagram to understand it. The overall goals of the business or application are implemented by workflows that often involve multiple externals and use cases.

- **Identify external stereotypes**. Differentiate the different stereotypes of externals with intuitive icons. *Rationale*: This guideline makes context diagrams and use case diagrams easier to understand and avoids confusion (e.g., when the UML stick figure is used for hardware and software externals as well as human actors).

4.4 Modeling Use Cases

- **Use cases should be functionally cohesive**. Each use case should fulfil all related business or application responsibilities involved in the primary external's goal. Do not base use cases on GUI screens. *Rationale*: This guideline makes use cases easier to understand.
- **Properly name the use cases**. Name each use case with a unique meaningful active verb phrase. Name the use cases from the external's viewpoint rather than the system viewpoint. Optionally, include the name of the primary external that benefits from the use case. *Rationale:* Verb phrases are best because use cases are functional abstractions. This guideline increases understandability.
  Bad Examples:
  - Turn on the digital thermostat. (This is really a use case path)
  - Desired temperature is changed. (This is not an active verb phrase and is from the wrong viewpoint.)
  - Temperature Control. (This is not a verb phrase and is from the wrong viewpoint.)

  Good Examples:
  - Change the state of the digital thermostat.
  - The user changes the desired temperature of the room. (Optional external)
- **Specify the use case requirements**. Provide a uniquely identified, textual requirement for each use case that captures its functional abstraction and external's goal. *Rationale*: This guideline provides an overview of the use case. A textual requirement is also the information expected by domain experts who are used to textual requirements specifications.
  Bad Examples:
  - The user changes the desired temperature of the room. (This is not a requirement, but rather an observation.)
  - The user shall change the desired temperature of the room. (This constrains the user. It is not a requirement on the digital thermostat).

  Good Examples:
  - The digital software shall permit the user to change the desired temperature of the room. (Written as a requirement - "shall" - on the digital thermostat.)
- **Create use case diagrams**. Create a top-level use case diagram for each external that summarizes the behavioral context of the application in terms of its externals, its use cases, and the relationships between them. Optionally create a lower-level use case diagram for each use case that has numerous relationships to other use cases. Identify

the primary external associated with each use case on the use case diagram. Provide narrative English descriptions of the use case diagrams that clarify anything that is not totally obvious in the diagram. Also, document the boundaries of the blackbox application on each use case diagram, and label the scope of each diagram. *Rationale*: Applications often have too many externals and use cases to document on a single top-level use case diagram. By limiting the scope of the diagram, the diagram becomes simpler, more focused, and therefore easier to understand. The diagrams also become easier to maintain because the impact of changes is more localized.

- **Provide a business justification**. Document the business justification for each use case. Ensure that the business justification actually justifies the use case and is more than merely a restatement of the use case requirement. *Rationale*: This guideline ensures that the use case specifies a real requirement.
- **Use essential use cases**. Avoid unnecessarily specifying design decisions in use cases by using essential (i.e., analysis level) use cases. Use preconditions rather than screen navigation details. *Rationale*: By including design information in use case models, the distinction between requirements and design is blurred. Design-level use cases must be updated each time the design changes.
  Bad Example:
  – The user chooses an account type from a pull-down menu bar on the GUI.
  Good Example:
  – The user requests an account type using the GUI.

4.5 Modeling Use Case Paths

- **Properly name the use case paths**. Name each use case path with a unique, meaningful phrase that captures the essence of the path. The name should be consistent with, and implied by, the associated requirement, preconditions, interactions, and postconditions. *Rationale*: This guideline improves communication by improving understandability.
  Bad Example:
  – Control the actual temperature of the room fails. (May fail multiple ways.)
  Good Example:
  – Handle temperature sensor failure.
- **Specify the use case path requirements**. Provide a uniquely identified, textual requirement for each use case path that captures its functional abstraction, preconditions, and postconditions. *Rationale*: This guideline makes the use case path easier to understand and promotes the testability of the resulting implementation.
  Bad Examples:
  – The user successfully increments the desired temperature of the room. (Not written as a requirement.)
  – The user shall be able to increment the desired temperature. (Vague – temperature of what. Written as a requirement on the user, not the digital thermostat.)
  Good Example:
  – The digital thermostat shall allow the user to increment the desired temperature of the room.
- **Capture exceptional paths and interactions**. Specify the exceptional basis paths as well as the normal paths through a use case. Document exceptional interactions as well as normal interactions, but avoid analysis paralysis due to trying to cover too many, extremely unlikely exceptional paths. *Rationale*: Too often, requirements only specify

how an application must function under normal circumstances, but not how it should function during abnormal circumstances. Requirements are therefore incomplete if only normal paths are specified. Exception handling software can often make up to 80% of business-critical or safety-critical software needing high operational availability and reliability. Unless exceptional paths are specified, there is little likelihood that they will be implemented or implemented correctly, and exceptional interactions must be documented if exceptional paths are to be completely documented. This guideline is critical if the resulting application is to be robust or have high operational availability.
Example path:
- Handle temperature sensor failure.

Example interaction:
- The digital thermostat software shall display the string "ERR" on the actual temperature display when the temperature sensor fails.

- **Document relevant assertions**. For each use case path, document both the preconditions and postconditions. *Rationale*: Without preconditions, there is no way to determine what is required to successfully execute the path. Without postconditions, there is no guarantee that the execution was correct, even if the visible interactions were correct. Postconditions can be used to specify otherwise hidden internal constraints. This greatly helps with the specification of the associated test cases.
Examples:
  - Preconditions of the "Increment desired temperature" normal path of the "Change the desired temperature" use case include:
    - The digital thermostat is in the on state.
    - The desired temperature is below the maximum.
  - Postconditions of the "Increment desired temperature" normal path of the "Change the desired temperature" use case are requirements and include:
    - The digital thermostat shall be in the on state.
    - The new desired temperature shall be one degree larger than the old desired temperature.

- **Document the interactions**. Completely document each interaction between the application and externals including the client and server of the interaction, the interaction itself, and any information that flows with the interaction. Optionally document interactions between the externals if it helps one understand the purpose of the interactions. Ensure that each interaction is actually an interaction and not really a precondition, postcondition, or traditional textual functional requirement or hidden calculation. *Rationale*: It is important to know which is the client and which is the server in order to determine who provides the resulting service. In addition, only the interactions initiated by the application represent requirements. The specification is incomplete if it does not specify what information is passed with the interaction, and this information helps to identify classes in the corresponding object model. The interactions are confusing and hard to understand if preconditions, postconditions, and traditional textual functional requirements are incorrectly labeled as interactions.
Bad Examples:
  - The user presses the on/off button. (Not part of the increment desired temperature path. Use a precondition instead.)
  - The digital thermostat shall increment its desired temperature. (Actually, a postcondition on internal data that technically is not visible at the interface.)
  - The desired temperature display displays the new desired temperature. (Actually a postcondition.)

- The digital thermostat software shall display the desired temperature. (Not specified as an interaction. What is the external?)

Good Examples:

- The user presses the increment desired temperature button.
- The increment temperature button interrupts the digital thermostat software with the string "increment desired temperature".
- Within .1 second of the user pressing the increment desired temperature button, the digital thermostat software shall request the desired temperature display to display the desired temperature of the room as a 1 to 3 digit integer followed by either "°F" or "°C".

- **Factor out common interactions**. Use subordinate use case paths and the "invokes" relationship to factor out common cohesive sets of interactions. However, watch out for excessive functional decomposition of the use cases. *Rationale*: This guideline simplifies the interactions and improves maintainability by eliminating unnecessary redundancy.
- **Document the logic**. Where appropriate, document the logic of the use case (e.g., branching, looping, critical regions) using pseudocode in the interactions and logic boxes on sequence diagrams. However, branching can be ignored if interactions and sequence diagrams are documented one per path. *Rationale*: This clarifies the actual behavior of the interactions.
- **Categorize the paths**. Categorize each use case path with its stability (how likely is it to change), frequency (how often does it execute), criticality (how important to the user is it), probability of defects (how likely are the developers to implement it correctly), and resulting risk. Use values of high, medium, and low. Calculate risk as a function of stability, frequency, criticality, and probability of defects. *Rationale*: The implementation of unstable paths may be postponed until the paths solidify, or else implementation may be moved forward in the schedule so that early prototyping may stabilize them. Reliability may be improved if limited test resources are allocated to paths that are executed frequently. Testers can emphasize the paths, the correct execution of which is critical to the user. Testing resources should also be allocated to those paths most likely to contain defects due to complexity or uncertainty (e.g., exceptional paths). This guideline allows one to identify the risk associated with each path in order to prioritize the scheduling of the development and testing of each path.
- **Create sequence diagrams**. Use blackbox sequence diagrams to document complex paths containing numerous interactions or complex logic. *Rationale*: Complex interactions are easier for most readers to understand if they are documented graphically.
- **Some interactions are requirements**. Formally specify those interactions that are initiated by the application as requirements using the standard terminology of requirements (e.g., the use of the term "shall"). Uniquely identify each such requirement including each item in the list of information passed with the interaction. *Rationale:* Any interaction initiated by the application is visible behavior and therefore required.

  Bad Examples:

  - The user shall press the increment desired temperature button. (This improperly constrains an external.)
  - The digital thermostat displays the new desired temperature to the user. (Not written as a requirement.)

Good Example:

- The digital thermostat shall display the new desired temperature to the user.

- **Postconditions are requirements**. Document each postcondition as a requirement using the standard format for requirements (e.g., the use of the term "shall"). Uniquely identify each postcondition. *Rationale:* A postcondition of the application is a required outcome of a use case path.
  Bad Example: Postconditions of the "Decrement desired temperature" normal path of the "Change the desired temperature" use case include:
  - The desired temperature shall be decremented. (Merely repeats the use case name.)
  - The new desired temperature is one degree larger than the old desired temperature. (Not written as a requirement.)

  *Good Example*s:
  - The digital thermostat shall be in the on state.
  - The new desired temperature shall be 1° larger than the old desired temperature.

- **Specifying complex internal algorithms**. If a new value is the result of a complex calculation that occurs internally to the blackbox application, use a postcondition to explicitly specify the complex algorithm to be used to calculate the new value. *Rationale*: Because the interactions only specify visible interactions between the application and its externals, they are not appropriate for specifying the complex algorithms sometimes needed to calculate a post-execution value.
  Example:
  - The new account balance equals the function foo() of its original value.

- **Standardize interaction formats**. Use a standard format to document interactions. For example, interactions may be requests, responses, and event notifications. *Rationale*: Differences between use case path specifications should be significant. This guideline greatly increases understandability and maintainability.
  *Example format*s:
  - External 'A' sends a 'B' request to the application including the following information: X, Y, and Z.
  - The application sends an 'A' response to external 'B' including the following information: X, Y, and Z.
  - The application sends an 'A' event notification to external 'B' including the following information: X, Y, and Z.

- **Requirements, not existing interface design**. Base the interactions on how the application to be developed must interact with its externals. Do not slavishly base the interactions on existing user interfaces or manual procedures. *Rationale*: Existing user interfaces or procedures may be obsolete. Part of the reason to create a new application is to enable process improvement.

5. References

[1] Donald G. Firesmith, "Use Cases: The Pros and Cons," in *Wisdom of the Gurus: A Vision for Object Technology*, Charles F. Bowman, ed., SIGS Books Inc., New York, New York, 1996, pp. 171-180.

[2] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997, pp. 738-740.

[3] Alistair Cockburn, "Structuring Use Cases with Goals," Journal of Object-Oriented Programming, SIGS Publications, Sep-Oct 1997 and Nov-Dec 1997.

[4] Susan Lilly, "Use Case Pitfalls: Top 10 Problems from Real Projects, in the proceedings of TOOLS USA'99.