

# Use Case Derived Test Cases

Dave Wood, Harris Corporation  
Jim Reis, Harris Corporation

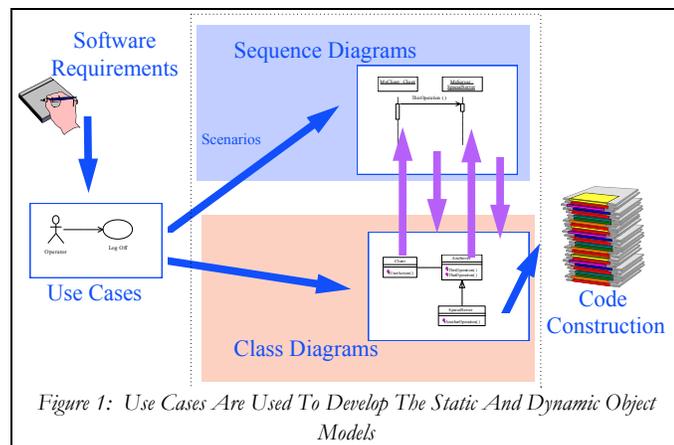
**Abstract:** Use Cases are used to specify the required functionality of an Object-Oriented system. Test cases that are derived from use cases take advantage of the existing specification to ensure good functional test coverage of the system.

A Use Case is a top level category of system functionality, (i.e.: Log on, Shut down, etc.). A Use Case has graphical representation, Figure 3, and a text description. The diagram identifies all the actors (outside of the system) involved in the function, as well as an indication of how the Use Case is initiated. The collection of Use Case diagrams provides a ‘context’ diagram of system interfaces. Each Use Case constitutes a complete list of events initiated by an Actor<sup>1</sup> and it specifies the interaction that takes place between an Actor and the System.[1]. In a Use Case the system is viewed as opaque, where only the inputs, outputs, and functionality matter.

The purpose of a Use Case May include:[2]

- Promoting Communication
- Understanding Requirements
- Helping to identify “capsules” to encapsulate data
- Focusing on the “What” rather than the How
- Providing Prototype Test Cases

We use the Use Cases to discover the objects (in the class diagram) that will construct a system to satisfy all functional requirements, and to construct the scenarios that ensure the functionality can be supported. Conceptually, we can view the functionality as a set of processes that run horizontally through the system, and the objects as sub-system components that stand vertically. Not every functionality uses every object, but each object may be used by many functional requirements. This transition from a functional to an object point-of-view is accomplished with Use Cases and Scenarios.



Parallel to the software development effort, the software test team can take advantage of the Use Case format by deriving Test Cases and Test Scenarios from them. Figure 2 depicts this process. The Use Cases and Test Cases are so closely coupled, that on some programs, the Test Team takes ownership of the Use Cases. In this way, both the software development and the test development remain synchronized if changes to the requirements specification occurs. The remainder of this paper describes the elements of a Use Case, and provides an example of Test Case development based on Use Cases.

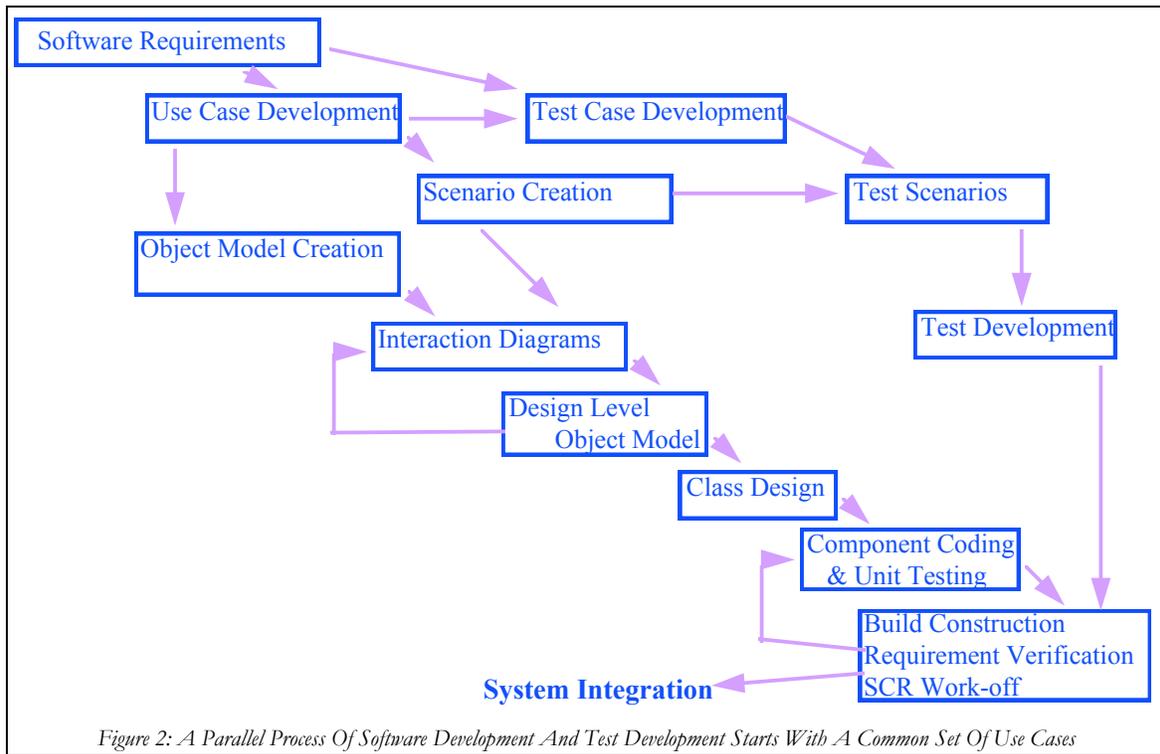
Testing, particularly software testing, means different things to different people. To help clarify our position it will be helpful to define the following:

- Verification Testing: proving that we have developed the required functionality
- Validation Testing: proving that the software is robust and error free

<sup>1</sup> Actor is a role that a user plays with respect to the system, and need not be human. A single Actor may perform in many Use Cases, A Use Case may have many Actors

In our organization, we perform Validation Testing during the Unit Testing process. The Use Case derived Test Cases are developed for Verification Testing, and (just like the Use Cases themselves) treat the system under test as a Black Box.

- Test Case: A portion of the overall test that is conducted to verify a required functionality.
- Test Scenario: a portion of a functional test conducted with one of the operational variations specified.

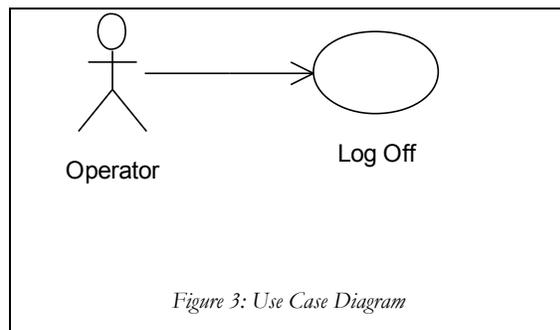


The first step is to develop the Use Case topics from the functional requirements of the Software Requirement Specification. The Use Case topics are depicted as an oval with the Use Case name. See Figure 3. The Diagram also identifies the Actors outside the system, and which participant initiates the action.

The Use Case diagram just provides a quick overview of the relationship of actors to Use Cases. The *meat* of the Use Case is the text description. This text will contain the following:

- Name
- Brief Description
- SRS Requirements Supported
- Pre & Post Conditions
- Event Flow

In the first iteration of Use Case definition, the topic, a brief description and actors for each case are identified and consolidated. In the second iteration the Event Flow of each Use Case can be flushed out. The Event Flow may be the *personification* and role playing of the requirements



specification. The requirements in the Software Requirement Specification are each uniquely numbered so that they may be accounted for in the verification testing. These requirements should be mapped to the Use Case that satisfies them for accountability.

The Pre-Condition specifies the required state of the system prior the start of the Use Case. This can be used for a similar purpose in the Test Case. The Post-Condition is the state of the system after the actor interaction. This may be used for test pass/fail criteria.

The event flow is a description (usually a list) of the steps of the actor's interaction with the system and the system's required response. Recall that system is viewed as a black box. The event flow contains exceptions, which may cause alternate paths through the event flow. The following is an example of a Use Case for a telephone systems.

**Event Flow: Local Telephone Call**

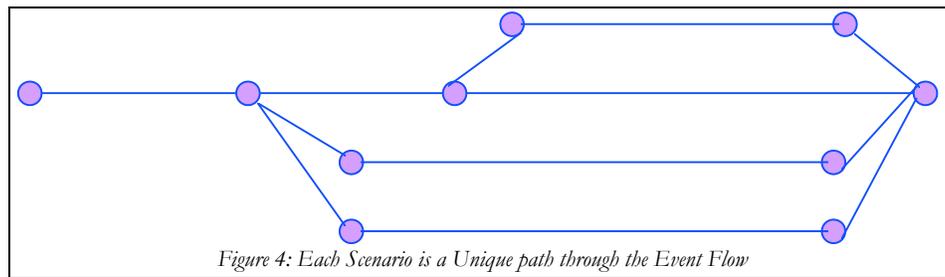
Use case starts when user picks up hand set  
Phone supplies audible dial tone  
User selects 7 digit number [E1, E2]  
User is supplied with ring tone [E3]

Exceptions:

E1: User enters invalid number,  
Voice Prompt 42 is played,  
User goes on-hook,  
Use case ends

E2: etc...

A *scenario* is a single path through the Use Case event flow. Generally there is one path that may be traversed that does not encounter exceptions (this is referred to as the Happy Day scenario). There are as many exceptions path scenarios as there are exceptions. Figure 4 graphically depicts the alternate scenarios through a single Use Case.



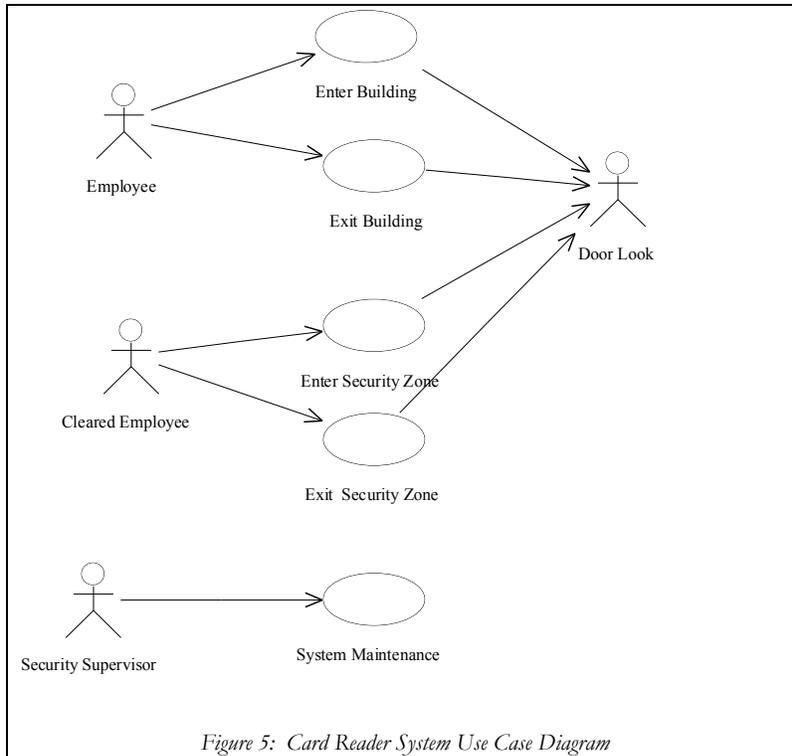
Now that we have had an introduction to Use Cases, let's look at an example of a Use Case and the Test Case derived from it. The Following problem statement (from [3]) will serve as our Software Requirement Specification.

**Example Problem Statement**

*Monitor and control the normal entry and exit of building occupants through the use of personal security cards.*

*This includes entry and exit of the building proper, and entry and exit from particular security zones within the building.*

*The system controls the locks on the doors, and will not unlock a door unless the appropriate security card is swiped through the card reader by the door*



Given this problem statement, we may arrive at a Use Case diagram shown in Figure 5.

The brief descriptions for these Use Cases might read as:

Enter Building :	Employee enters the building using card reader passage
Exit Building :	Employee exits the building using card reader passage
Enter Security Zone:	Cleared Employee enters the vault using card reader passage
Exit Security Zone:	Cleared Employee exits the vault using card reader passage
System Maintenance:	Authorized user enters/edits employee card data

Note that the enter and exit use cases could be combined into one use case for the building and one for the security zone due to similarity between the enter and exit functions.

Looking now at one particular Use Case, Enter Building, we would expect an Event Flow such as:

**Enter Building Event Flow**

- Use case starts when user slides card through card-reader
- Card-reader scans employee ID from card [E1]
- System validates employee access [E2]
- System unlocks door for configured time period [E3]
- Employee opens door [E4]
- Employee enters and door shuts [E5]
- System locks door [E6], Use Case ends

The following events and the associated exceptions can then be generated for the Enter Building use case.

1. Use case starts when the user slides a card through the card-reader
2. Card-reader scans employee ID from card
  - Exception 1:** Card can't be read
    - Log event
    - Use case ends

3. System validates employee access
  - Exception 2:** Employee ID is invalid
    - Log event
    - Use case ends
4. System unlocks door for configured time period
  - Exception 3:** System unable to unlock door
    - Log event
    - Use case ends
5. Employee opens door
  - Exception 4:** Door is not opened
    - System waits for timeout
    - System locks door
    - Use case ends
6. Employee enters and door shuts
  - Exception 5:** Door is not shut
    - System waits for timeout
    - System attempts to lock door
    - Log event
    - Set alarm condition
    - Use case ends
7. System locks door, Use case ends
  - Exception 6:** Door fails to lock
    - System waits for timeout
    - System attempts to lock door
    - Log event
    - Set alarm condition
    - Use case ends

For this Use Case, based on the above events and exceptions, we have derived the following Test Case.

## Enter Building Test Case

**Test Condition 1:** Happy days scenario – valid employee card is used

- Swipe card
- Verify door is unlocked
- Enter building
- Verify door is locked

**Test Condition 2:** Card can't be read

- Swipe a card that is not valid
- Verify event is logged

**Test Condition 3:** Invalid employee ID

- Swipe card with invalid employee ID
- Verify door is not unlocked
- Verify event is logged

**Test Condition 4:** System unable to unlock door

- Swipe card
- “Injected” failure of unlocking mechanism
- Verify event is logged

**Test Condition 5:** Door is not opened

- Swipe card
- Verify door is unlocked
- Don't open the door and wait until timeout is exceeded
- Verify door is locked

**Test Condition 6:** Door is not shut after entry

- Swipe card
- Enter building
- Hold door open until timeout is exceeded
- Verify alarm is sounded
- Verify event is logged

**Test Condition 7:** Door fails to lock

- Swipe card
- Enter building
- “Injected” failure of locking mechanism
- Verify alarm is sounded
- Verify event is logged

Test cases 4 and 7 would normally be verified at the unit test or integration test phase due to having to, presumably, use an intrusive method to fail the lock mechanism.

There are, however, limitations of Using Use Cases For Test Case Generation. Use cases are not used to model capacity and performance related requirements, they are used to only model functional requirements. Consequently the non-functional requirements need to be verified outside of the use case generated tests cases.

Given the large number of use cases, and even larger number of scenarios, some prioritization of the generated test cases must be performed to obtain a reasonable set of tests to actually run. Besides prioritizing the tests that are to be performed at system test time, use cases can also be verified at unit test and integration and test time, thus spreading the test effort out over the more of the life cycle of the project.

**Conclusions**

Deriving test cases from use cases ensures that the required functionality of the system is both developed in the software and reflected in the test plan. By working from the same specification, the use cases, development and test are more likely to arrive at test time with the same expectations of the system. Coordination between development and test is further ensured by having the test organization take ownership of the use cases. In our organization, this method has been used successfully on multiple projects.

References

[1] Ivar Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1994

[2] Martin Fowler, UML Distilled. Addison-Wesley, 1997

[3] Robert Martin, Designing Object-Oriented C++ Applications using the Booch Method, Prentice-Hall, 1995