Putting Use Cases to Work

by **Benjamin A. Lieberman, Ph.D.**
Senior Software Architect
Trip Network, Inc.

Use-case-based analysis of requirements is a well-established technique for the solicitation, capture, and presentation of software requirements.[1] The success of this technique can be attributed to the use case's ability to present a cohesive view of the system requirements that can be readily understood and applied by all members of the development organization. In particular, there are four primary project stakeholders who will directly benefit from a well-conceived and well-executed set of use cases:

- *System architects and designers*
- *Quality assurance engineers*
- *Customers/clients*
- *Project managers*

Each of these stakeholders has dramatically different information needs, depending on each group's particular roles and responsibilities in the development organization. System architects and designers are interested in the system components and apply use-case descriptions to determine system boundaries, interfaces, controllers, and elements. QA engineers are focused on verification of adherence to requirements and overall robustness of the system; they read use cases to create and maintain a complete set of test cases and procedures. The customer/client is the primary source of requirements and the final arbiter of system quality; this stakeholder will be looking to the use-case descriptions for satisfaction of requested functionality. Finally, the program manager will be interested in applying the use cases for project estimation, scheduling, and risk and change management.

This article will explore each of these stakeholder needs and present examples of how a set of use cases can meet them. To aid the reader in applying use

cases effectively, a complete use-case example is included as a Sidebar, which demonstrates "use-case utilization" by each of the primary stakeholders noted above. The creation of use cases has been thoroughly discussed elsewhere,[2] but I will highlight some of the characteristics of well-written use cases. Use-case features, such as data element description, exceptional flow error messages, use-case dependencies, and pre- and post-processing conditions, are all critical to realizing the benefits of use-case-based modeling.

System Analysis and Design

Planning the detailed construction of the software system is the primary application of a use case. During use-case review, a system designer looks for four distinct groups of interacting system elements (see Figure 1)[3]:

Boundaries	System boundaries that separate internal system responsibilities from external systems or actors.[4]
Interface	Represents all external interactions with system actors. These interactions can be represented by a variety of interaction mechanisms (Figure 2). Interfaces are also defined between non-human actors and the system (Figure 3).
Controller	Represents the management of processing flow through the system. Multiple controllers may be needed for complex use cases.
Entity	Represents data or other system elements and relationships.

Although this approach to computer system analysis is well suited to object-oriented design and development, it should be noted that it is based in formal systems analysis[5] and can be used to describe any complex system.

Figure 1: Analysis Icons (Depicting Interacting System Elements)

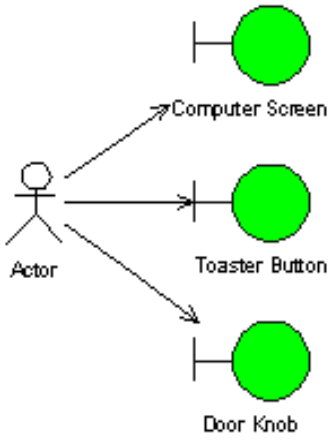


Figure 2: Depicting Actor to Interface Relationships

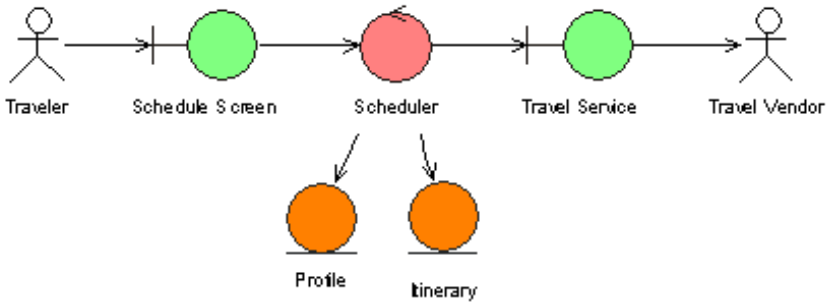


Figure 3: "Check Schedule" Sample Use-Case Analysis

Figure 3 shows a relatively simple system. This model was created directly from a sample use case (see Sidebar) by first establishing the system boundaries. This was done by inspecting for system actors and showing the system "services" that are being requested in the use case. The next step in the analysis is to study the use case *Basic*, *Alternate*, and *Exceptional* processing flows to determine:

- Where the system interface(s) will be located (usually based on the name of the use case).
- Data elements and relationships (as defined within the use-case flow or supporting documentation).
- Actions the system is expected to perform (based on use-case activities and illustrated by UML activity diagrams).[6]

When creating a use case, it is important to understand 1) the nature of the information (data entities) that will be manipulated by the system, and 2) the actions for which that data is required. The data entity description illustrates one of the key features of a well-written use case, namely the capture of detailed data requirements. However, this description should focus only on the data elements and relationships themselves, rather than on any particular implementation (e.g., avoid detailing data entity Name as `varchar(50)`, or `java.lang.String`). For one example of use-case data representation, please refer to the sample use case (see Sidebar). The example use-case description uses a modified technique based on Data Dictionary notation,[7] which shows elements composed of attributes or references to other elements.

Once you have a general understanding of the entities, actions, and interfaces required by the system, you can then create a more detailed *Design*, since the analysis has progressed to the point where code can reasonably be generated. The most commonly used UML models for design are the Class Diagram and the Sequence Diagram (Figures 4 and 5).

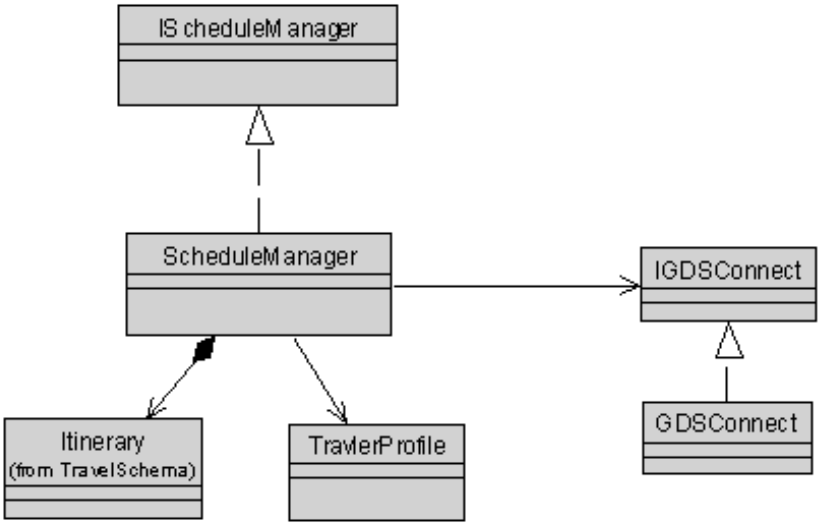
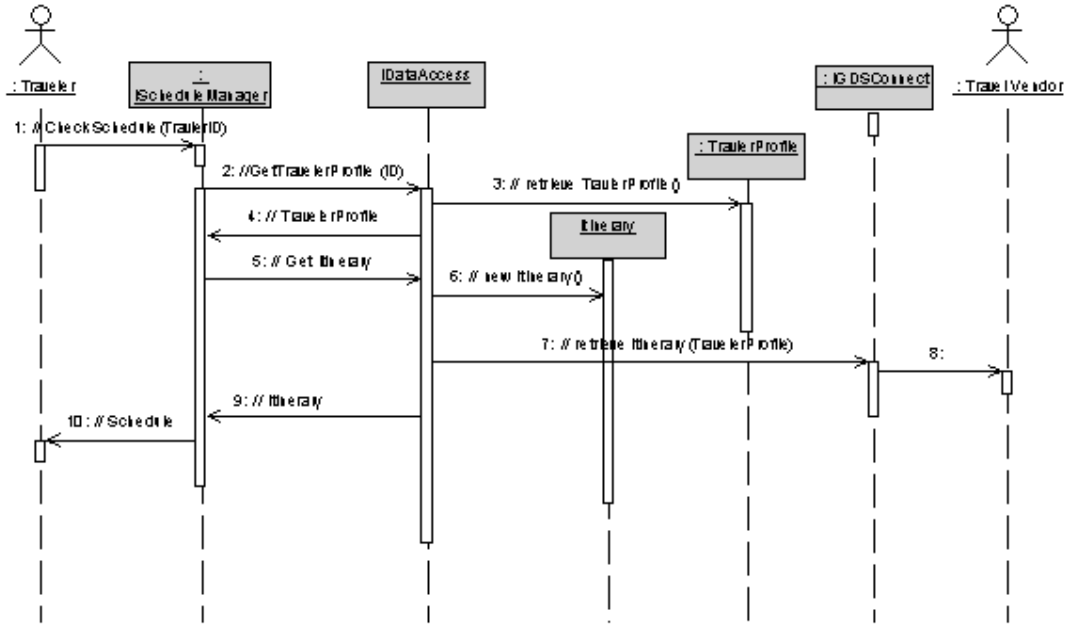


Figure 4: Sample Travel Schedule Class Diagram



Click to enlarge.

Figure 5: Sample Check Schedule Sequence Diagram

Both the analysis and designs are added by the creation and maintenance of activity diagrams. Activity diagrams are vital to communicating information regarding process flows, as shown in the sample use case (see Sidebar). These diagrams permit the analyst to rapidly view all available paths through the use case. This is extremely helpful for discovering system controllers and process flows, as it indicates the points where system processing is expected to occur, and the nature of that processing.

The system designer can now apply the use-case flows, the system analysis, sequence diagrams, and class diagrams to further describe the detail of the system operation; data elements are mapped to specific types, architectural mechanisms (concurrency management, transactions, communication protocols, persistence mechanisms, etc.) are described, design patterns are

discovered and applied, and the primary code base is created.

Quality Assurance and Testing

Test cases are created in order to verify that the desired system functionality has been implemented by the development effort. The creation of test cases is directly tied to the functional flows detailed in the use case. Test cases are created based on the concept of a use-case *scenario*, which is defined as a single execution path through a use case. This is best illustrated by examining a use-case activity diagram (see Sidebar, for example). The scenario begins at the start of the diagram and selects one particular path through the execution. In this manner, all possible use-case execution paths can be exercised.

While writing a use case, you should consider the following critical points regarding the creation of test cases:

- The potential error conditions (e.g., exceptional flows) should be detailed, particularly those that will have a user presentation or prevent completion of the user task.
- Data essential for flow should be highlighted, as well as the requirements regarding data format and content (i.e., validation).
- Entry and exit conditions for a use case should be clearly defined.

The following shows how the sequence of test cases can be derived from a simple use case:

Test Case 1: Basic Flow (positive test)

- Select the **View Schedule** button from the main screen.
- Enter the user id `TESTER` and the

An Example Use Case

Preconditions

- A valid itinerary exists.
- A valid user profile exists.

Restrictions

- Only one itinerary is permitted for a user.
- Only one itinerary may be viewed at a time.

Basic Flow

1. The user requests to view his or her schedule (view schedule button on the user interface).
2. The system validates the user.
3. The system retrieves the user's Traveler Profile from persistent store (e.g., database).

   a. A Traveler Profile consists of the following elements:

   1..1:Traveler = TravelerID + First + Middle + Last

   2..2:Address = Street Address + City + State/Region + MailCode + Country

   0..5:Credit Card = Number + ExpirationDate + Provider(Visa | AMEX | MasterCard | Discover)

   1..3:Phone Number = Number + AreaCode + CountryCode + NumberType(Mobile | Home | Work | Other)

   NOTE: TravelerID is externally created by the Travel Vendor actor

   NOTE: There is (1) Mailing and (1) Billing Address
4. The system uses the TravelerID to retrieve the Itinerary from the Travel Vendor actor.

   a. An Itinerary consists of the following elements:

   0..N:Trip = Flight + Car Rental Reservation + Hotel Reservation

password `GOFORIT` in the user login screen.

- The system will display the current schedule.

Test Case 2: Alternate Flow (positive test)

- Follow Test Case 1.
- Click on the **Print Schedule** button.
- The printer will produce the current schedule.

Test Case 3: Exception 1 (negative test)

- Select the **View Schedule** button from the main screen.
- Enter the user id `WHOOPS` and the password `NOPE` in the user login screen.
- The system should display the following error message:

```
You have provided an incorrect user ID and/or password; please reenter.
```

- The system should return to the user login screen.

Test Case 4: Exception 2 (negative test)

Etc…

The Sidebar shows how the structure of the use case mirrors the structure of the test case. But bear in mind that, because system test cases are directly derived from the use cases, any gaps or errors will be propagated into the test plan. It is therefore very important that the test engineer participate in the creation and review of the system use cases. It is also important that all use-case-specified functionality be verifiable, further underscoring the importance of QA team participation in the use-case

0..N:Flight = VendorCode + Flight Information

0..N:Car Rental Reservation = VendorCode + Car Information

0..N:Hotel Reservation = VendorCode + Hotel Information

NOTE: VendorCode is defined in a file provided by the Travel Vendor actor.

NOTE: See Travel Vendor Interface document (XML Schema) for detailed data field structure and content (e.g., Flight, Car, Hotel Information).

5. The system presents the schedule to the user (please refer to GUI: Schedule Display -- Figure 6).

Alternate Flow

1. At step 5 in the basic flow, the user may opt to send a summary of the itinerary to a printer.
2. The system will submit the schedule to the printer queue.
3. The Basic Flow continues at Step 5.

Exceptional Flow

1. At Step 2 of the Basic Flow, if the user is not authenticated, then the following message is displayed:

   ```
   You have provided an incorrect user ID and/or password; please reenter.
   ```

   The flow resumes at Step 2.
2. At Step 3 of the Basic Flow, if the database is unavailable, then the following message is displayed:

   ```
   The traveler profile is unavailable; you cannot view your itinerary at this time.
   ```

   The use case ends.
3. At Step 4 of the Basic Flow, if the Travel Service is unavailable, then the following message is displayed:

   ```
   The traveler itinerary is not available; you cannot view your schedule at this time.
   ```

definition process.

The steps involved in the creation of a validation test plan identify the end-to-end flow for the use case, then identify all interaction dependencies (i.e., other use cases that are `<<included>>` or `<<excluded>>` by the current use case), essential data, optional data, data validation conditions, and finally primary, alternate, and exceptional flows. The test cases should focus heavily on less likely flows, since these are the paths most likely to contain flaws. This is often a result of management pressure to show working functionality, which reduces attention to exceptional conditions and error handling. As such, these paths receive insufficient detail during the development process. The quality of a system is often more apparent in its ability to gracefully handle error conditions than in the performance of a particular task.

Although use cases may specify special performance requirements (e.g., "The system must respond with an itinerary or error to the user request within five seconds"), performance testing is usually not considered a part of use-case testing. Performance and volume testing are usually driven by a more global system approach and thus will not be considered here.

Customer Expectation Management

Use cases are a very effective tool for managing customer expectations for the system. However, although clients are often experts in their own fields, they are usually not well versed in software development methodologies. That is why you should observe the following guidelines when writing use cases:

- Ensure that all technical terms are defined in a Glossary.
- Avoid implementation details.

The use case ends.

4. At Step 2 of the Alternate Flow, if the printer is unavailable, then the following message is displayed:

   ```
   The printer is unavailable.
   ```

   The flow resumes at Step 3 of the Alternate Flow.

Special Requirements

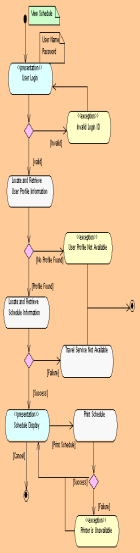
- The system must respond with an itinerary or error to the user request within five seconds.

Figure 7 shows an activity diagram for this use case.

Click to enlarge

Figure 6: Schedule Visual Display

- Use non-technical language whenever possible.
- Maintain a consistent level of abstraction (i.e., don't include detailed descriptions with general statements of functionality).
- Tell a clear, coherent story.
- Be as complete as possible in the descriptions of the flows.
- Review all use cases with the customer to ensure clarity and understanding.



Click to enlarge

Figure 7: Sample Check Schedule Use-Case Activity Diagram

Remember that the use case represents a contract between the development staff and the customer. Misunderstandings at the use-case development stage can have dramatic and often adverse consequences later in development.

The use case should clearly "speak" to the customer, which is particularly important during the review process. If the customer is confused about a particular flow, then you should rewrite the flow to better describe the conditions. One useful validation technique is to ask the customer to imagine their daily activities and to walk through these activities step-by-step. If the description includes steps not covered in the use case, then they should be added if they are within the scope of the system.

The reviewer should also pay particular attention to exceptional cases, since these error conditions often prevent the user from accomplishing the intended task. Make sure that the client understands and agrees to the system behavior in the event of an exception.

Project Management

In this final section, I will discuss how the use case represents a key tool for project management. Although no one technique will satisfy all of the needs for project management, in my experience use cases are a very effective tool for addressing the following three project needs:

- Management and mitigation of risk.
- Creation and maintenance of schedules.
- Control of cost and scope changes.

Use cases can be scored and ranked relative to one another based on five major criteria:

- *Risk* - The chances of an occurrence that will prevent or seriously hamper ability to deliver the requested functionality.

- *Effort* - The amount of personnel and development resources required to implement the functionality.
- *Cost* - The actual cost to the business, in terms of both financial and political capital (see below).
- *Dependency* - The natural or required order for use-case implementation (e.g., inventory must be searched for availability before assignment or purchase).
- *Customer Priority* - The relative importance of each use case as viewed by the customer.

First, I will provide a sample table that a project manager might create as a way to rank use cases by relative priority. I will then explain the value of this method, along with each of the above criteria used to provide an initial ranking.

Table 1: Use-Case Categorization and Prioritization[8]

Use Case	Effort	Risk	Cost	Dependency	Customer Priority
UC1	H	H	H	UC3	1
UC2	H	L	L		6
UC3	L	M	M		5
UC4	M	H	L	UC3	3
UC5	M	L	H		4
UC6	L	H	L	UC1, UC4	2

The main advantages of use cases are that they focus on the user experience, organize functionality into discrete boundaries, and indicate dependencies between areas of functionality. Unlike an absolute ordering based on some form of in-depth sizing metric (e.g. function points), the relative-ordering approach detailed in Table 1 permits the project manager to quickly determine the scope of the project, looking at the level of effort and cost to balance against risk, schedule demands, and customer needs. This approach also permits the mitigation of risk by focusing on "highest risk, greatest benefit" items early in the project. Finally, the project manager can apply this approach to the creation of schedules, since the relative ranking will permit a natural development order to be determined.

Consider how each of the five criteria in Table 1 can be used to rank use cases:

Risk takes on many forms, including technical, political, timing, market-driven, and staffing risks. Each of these risks can be determined for a use case and assigned a relative ranking of High, Medium, or Low.

Effort can be determined by taking a rapid estimate based on the number and type of activities detailed in the use-case activity diagram. For example, a use

case that contains three user presentation elements, seven activities, and five exceptions will be about fifty percent larger than a use case containing one user element, four activities, and three exceptions. Again, this relative ordering approach is primarily useful early in a project, and will give an order of magnitude estimate of the level of development effort that will need to be expended. Since this approach is comparative (i.e., non-absolute) between use cases, there is less concern with determining exact levels of effort. Instead, the comparison of use-case effort will provide an outline for the schedule that can be detailed, as greater information is known about the exact consequences of the use case.

Cost is determined from the true return on investment for the development organization. This includes all development expenses, staffing costs, and the potential cost of *not* providing the functionality (what I term the *political cost*). Each of these costs can be estimated to provide a relative ranking against all of the proposed use cases. (In Table 1, I use High, Medium, and Low to represent all of these factors combined.) This measure is usually the most difficult to determine, and so should be considered carefully when negotiating with the customer on project scope.

Dependency is determined from the natural ordering of the desired functionality. If a business model has been created, then the ordering for the use cases will be apparent from the business flows. If a business model is unavailable, then inspection of the use-case model will permit a determination of the dependencies between use cases. For example, if one use case is for purchase of an inventory item, then a dependent use case may be the credit card check, since purchase cannot occur without this use case being present. Noting the functional dependencies in this manner permits the project manager to schedule dependent use cases in the proper order.

Finally, ***Customer Priority*** is a determination created directly by the customer stakeholder. This allows the customer to feed into the ranking and even to override the level of risk and effort determined by the project manager. For example, a moderate-risk use case may be elevated above a high-risk use case due to a high customer priority level. One key element here is to ensure that the customer does not try to make every use case a "priority." This is the equivalent of not assigning any priorities. Instead, require that the customer identify between one and three top needs, followed by all the remaining use cases in order of preference or need.

By organizing the project along these measures, project managers will be well on their way toward effective scope management. The use cases that have high customer priority and high risk should be completed early in the project, whereas low-priority, high-risk elements can be eliminated or moved to non-critical delivery paths. For all other use cases, a trade-off against cost and effort can be considered when negotiating with the client.

Taken together, these factors can all be readily determined by examining the use case, even in incomplete form, which will provide important information for shaping projects goals and milestones.

Conclusion

A well-conceived and well-executed set of requirements is essential to the success of any significant software development project, and use cases have been shown to be one of the best methods for requirement solicitation, capture, and presentation. They can be used by a variety of stakeholders, including architects, designers, customers, testers, and managers. Each of these groups will have different needs that must be fulfilled by the use-case set, including discovery of system functionality, verification of desired system behavior, validation of stakeholder needs, and project organization and scope. By illustrating the multiple applications for use cases, we have shown that use cases represent an extremely flexible and powerful approach for requirements management.

Notes

1 I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

2 See the first five listings under References below.

3 Jacobson, *Object-Oriented Software Engineering*.

4 Actors represent any external user of the system and may include humans or other computer systems.

5 See Jacobson, *Object-Oriented Software Engineering*, as well as M.E. Model, *A Professional's Guide to Systems Analysis* (McGraw-Hill, 1996) and T.H. Athey, *Systematic Systems Approach* (Prentice-Hall, 1982).

6 T.H. Athey, *Systematic Systems Approach*. Prentice-Hall, 1982.

7 A.M. Langer, *The Art of Analysis.* Springer-Verlag, 1997.

8 For more information on prioritization and project scope, see Chapter 20 of D. Leffingwell and D. Widrig, *Managing Software Requirement*. Addison-Wesley, 2000.

References

T.H. Athey, *Systematic Systems Approach.* Prentice-Hall, 1982.

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

E.F.J. Ecklund, L.M.L. Delcambre, and M.J. Freiling, "Change Cases: Use Cases that Identify Future Requirements." OOPSLA-96, 1996.

M. Fowler, "Use and Abuse Cases." *Distributed Computing*, April, 1998.

I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

A.M. Langer, *The Art of Analysis*. Springer-Verlag,1997.

D. Leffingwell and D. Widrig, *Managing Software Requirements, A Unified Approach*. Addison-Wesley, 2000.

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

M.E. Model, *A Professional's Guide to Systems Analysis*. McGraw-Hill, 1996.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!