Report of Use case and Class Diagram

Use case

Use cases can be valuable tools for understanding a specific system's ability to meet the needs of end users. When designing software or a system, enhance development efforts by thinking through practical scenarios about product usefulness. Use cases can also be effective for product marketing purposes. Here are some steps to guide through the writing process.

Defining the Purpose and Scope

1 **Write a goal statement.** Write a sentence or two that briefly describes the primary goal of implementing the technology or business process. Define specifically the goals of the primary user of the system. A use case can be written to describe the functionality of any business process or piece of software or technology a business uses.

- For example, write use cases about logging into a system, managing an account or creating a new order.

2 **Identify the stakeholders.** These are the people in the organization who care about the outcome of the process. They may not be users in the process described by the use case. But the system acts to satisfy their interests. List all of the stakeholders, including their names and their interest with respect to the operation of the system. Also, note any guarantees they expect from the system.

- For example, if writing a use case about how an ATM machine functions, the stakeholders would include the bankers and the ATM owners. They are not present when the user uses the ATM machine to withdraw cash. However, they must be satisfied that systems are in place to verify the amount of money in the user's account before dispensing cash and to create a log of transactions in the event of a dispute.

3 **Define what is in and out of scope.** Specifically identify the system that is being evaluated, and leave out elements that are not part of this system. It can be useful in defining the scope of a project to create a spreadsheet containing an in/out list. Create three columns. The left column lists any topic at all that might relate to the system. The next two columns are titled "In" and "Out." Go through the list and determine which topics are in and which are out.

- For example, if writing a use case implementing software to create purchase orders, topics that would be "In" would include producing reports about requests, merging requests to a purchase order, monitoring deliveries, and new and existing system software. Topics that would be "Out" would include creating invoices and non-software parts of the system.

Writing the Steps of a Use Case

1 **Define the elements of the use case.** All of these elements are required in every use case. Use cases accumulate scenarios. They define how a user uses a system, what happens when the system succeeds, and what happens when it fails. Each scenario describes a procedure and what happens as each step progresses.

- Users are all of the people who will engage in the activities described in the use case. For example, if writing a use case for logging into a software system, the users would be anyone who must log in.
- Preconditions are those elements that must be in place prior to the start of the use case. For example, users with permission to use the system have been identified and entered into the system ahead of time, so the system will recognize their usernames and passwords when entered.
- The basic flow is the procedure the users use to achieve the primary goal of the system and how the system responds to their actions. For example, the user inputs a username and a password, and the system allows the user in.
- Alternate flows explain less common actions. For example, the user is on a different computer and must answer a security question.
- Exception flows detail what happens when the user cannot achieve the goal. For example, the user inputs an invalid user name or password.
- Post conditions are those elements that must be present when the use case is completed. For example, the user can proceed to use the software.

2 **Define how the user will use the technology or process.** Each thing the user does becomes a separate use case. The scope of a use case is narrow. For example, if a company is implementing new software to create purchase orders, write several use cases about this. One use case might be about how users log in to the system. Another might be about how to run requisition reports. List all of the functions of the new technology or business process analyzing, and write a use case for each one.

3 **Describe the normal course of events for each use case.** Outline everything the user does and how the technology or process responds to those actions. In a use case about how users log into a software system, the normal course of events would state that the user enters a username and password. The software responds by verifying the user and either granting or denying access to the system.

- Alternate flows and exception flows are written to describe the actions when there are obstacles to the goal.
- If the user is denied access because the system didn’t recognize her computer, she may be prompted to verify her identity by answering a security question.
- If the user inputs an invalid username or password, she may be prompted to answer a security question and enter an e-mail address to receive new log in information.

4 **Repeat the steps for all other functions and users.** Write use cases for all of the other functions of the software or business process. Identify the users for each function, and write the steps for the normal course of events. Explain contingencies for when the goal cannot be achieved. For each step, explain how the system responds to the actions of the user.

Writing Valuable Use Cases

1 **Capture what the technology or business process does.** The use case explains the goal of the technology or process, not how the technology functions. In other words, a use case about logging in to software does not include how the code must be written or how the technological components are connected. It simply focuses on what the user needs to do and how the software responds.

- Get the level of detail right. For example, if writing a use case about implementing technology, don't exclude details about how the software responds to users.
- Alternatively, adding too much detail about how the software functions reads more like system design implementation than a use case.

2 **Keep the use case primarily textual.** Use cases do not need to include complex flow charts or visual diagrams that explain the process. Simple flow charts can often be used to clarify information. However, the use case should be largely word-based. The style of writing should be very simple so that others can read and comprehend it without specific training.

3 **Learn the most relevant details.** Writing a good use case helps you learn exactly how a piece of software or business process works. It educates and the reader about the correct use of applicable vocabulary. This way is not using technological terms incorrectly or gratuitously. Learn to discuss technology and business processes in a way that is useful and valuable to others in the business community.

Class Diagram

UML 2 considers structure diagrams as a classification; there is no diagram itself called a "Structure Diagram." However, the class diagram offers a prime example of the structure diagram type, and provides us with an initial set of notation elements that all other structure diagrams use. And because the class diagram is so foundational, the remainder of this article will focus on the class diagram's notation set. By the end of this article you should have an understanding of how to draw a UML 2 class diagram and have a solid footing for understanding other structure diagrams when we cover them in later articles.

The Basics

The purpose of the class diagram is to show the types being modeled within the system. In most UML models these types include:

- a class
- an interface
- a data type
- a component.

UML uses a special name for these types: "classifiers." Generally, you can think of a classified as a class, but technically a classifier is a more general term that refers to the other three types above as well.

Class name

The UML representation of a class is a rectangle containing three compartments stacked vertically, as shown in Figure 1. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations. When drawing a class element on a class diagram, you must use the top compartment, and the bottom two compartments are optional. (The bottom two would be unnecessary on a diagram depicting a higher level of detail in which the purpose is to show only the relationship between the classifiers.) Figure 1 shows an airline flight modeled as a UML class. As we can see, the name is *Flight*, and in the middle compartment we see that the Flight class has three attributes: flight Number, departure Time, and flight Duration. In the bottom compartment we see that the Flight class has two operations: delay Flight and getArrivalTime.

Class attribute list

The attribute section of a class (the middle compartment) lists each of the class's attributes on a separate line. The attribute section is optional, but when used it contains each attribute of the class displayed in a list format. The line uses the following format:

1 name : attribute type

1 flightNumber : Integer

Continuing with our Flight class example, we can describe the class's attributes with the attribute type information, as shown in Table 1.

Table 1. Table 1: The Flight class's attribute names with their associated types

Attribute Name	Attribute Type
flightNumber	Integer
departureTime	Date
flightDuration	Minutes

In business class diagrams, the attribute types usually correspond to units that make sense to the likely readers of the diagram (i.e., minutes, dollars, etc.). However, a class diagram that will be used to generate code needs classes whose attribute types are limited to the types provided by the programming language, or types included in the model that will also be implemented in the system.

Sometimes it is useful to show on a class diagram that a particular attribute has a default value. (For example, in a banking account application a new bank account would start off with a zero balance.) The UML specification allows for the identification of default values in the attribute list section by using the following notation:

1 name : attribute type = default value

For example:

1 balance : Dollars = 0

Showing a default value for attributes is optional; Figure 2 shows a Bank Account class with an attribute called *balance*, which has a default value of 0.

Class operations list

The class's operations are documented in the third (lowest) compartment of the class diagram's rectangle, which again is optional. Like the attributes, the operations of a class are displayed in a list format, with each operation on its own line. Operations are documented using the following notation:

1 name(parameter list) : type of value returned

the delayFlight operation has one input parameter — numberOfMinutes — of the type Minutes. However, the delayFlight operation does not have a return value. [Note: The delayFlight does not have a return value because I made a design decision not to have one. One could argue that the

delay operation should return the new arrival time, and if this were the case, the operation signature would appear as delayFlight(numberOfMinutes : Minutes) : Date.] When an operation has parameters, they are put inside the operation's parentheses; each parameter uses the format "parameter name : parameter type".
When documenting an operation's parameters, you may use an optional indicator to show whether or not the parameter is input to, or output from, the operation. This optional indicator appears as an "in" or "out" as shown in the operations compartment in Figure 3. Typically, these indicators are unnecessary unless an older programming language such as Fortran will be used, in which case this information can be helpful. However, in C++ and Java, all parameters are "in" parameters and since "in" is the parameter's default type according to the UML specification, most people will leave out the input/output indicators.

Inheritance

A very important concept in object-oriented design, *inheritance*, refers to the ability of one class (child class) to *inherit* the identical functionality of another class (super class), and then add new functionality of its own. (In a very non-technical sense, imagine that I inherited my mother's general musical abilities, but in my family I'm the only one who plays electric guitar.) To model inheritance on a class diagram, a solid line is drawn from the child class (the class inheriting the behavior) with a closed, unfilled arrowhead (or triangle) pointing to the super class. Consider types of bank accounts

Abstract classes and operations

The observant reader will notice that the diagrams in Figures 4 and 5 use italicized text for the BankAccount class name and withdrawal operation. This indicates that the BankAccount class is an abstract class and the withdrawal method is an abstract operation. In other words, the BankAccount class provides the abstract operation signature of withdrawal and the two child classes of CheckingAccount and SavingsAccount each implement their own version of that operation.
However, super classes (parent classes) do not have to be abstract classes. It is normal for a standard class to be a super class.

Associations

When you model a system, certain objects will be related to each other, and these relationships themselves need to be modeled for clarity. There are five types of associations. I will discuss two of them — bi-directional and uni-directional associations — in this section, and I will discuss the remaining three association types in the *Beyond the basics* section. Please note that a detailed discussion of when to use each type of association is beyond the scope of this article. Instead, I will focus on the purpose of each association type and show how the association is drawn on a class diagram.

Packages

Inevitably, if you are modeling a large system or a large area of a business, there will be many different classifiers in your model. Managing all the classes can be a daunting task; therefore, UML provides an organizing element called a *package*. Packages enable modelers to organize the model's classifiers into namespaces, which is sort of like folders in a filing system. Dividing a system into multiple packages makes the system easier to understand, especially if each package represents a specific part of the system. [Note: Packages are great for organizing your model's classes, but it's important to remember that your class diagrams are supposed to easily communicate information about the system being modeled. In cases where your packages have lots of classes, it is better to use multiple topic-specific class diagrams instead of just producing one large class diagram.]

Importance of understanding basics

It is more important than ever in UML 2 to understand the basics of the class diagram. This is because the class diagram provides the basic building blocks for all other structure diagrams, such as the component or object diagrams (just to name a few).

Conclusion

There are at least two important reasons for understanding the class diagram. The first is that it shows the static structure of classifiers in a system; the second reason is that the diagram provides the basic notation for other structure diagrams prescribed by UML. Developers will think the class diagram was created specially for them; but other team members will find them useful, too. Business analysts can use class diagrams to model systems from the business perspective. As we will see in other articles in this series on UML basics, other diagrams — including the activity, sequence, and statechart diagrams — refer to the classes modeled and documented on the class diagram.