**Extended Unified Modeling Language (XUML)**

**for Use Case Validation**

Al Cline, PMP

1. **Overview**

Use case validation is required by the IIBA for all professional business analysts. This paper presents two simple extensions to the standard sequence diagram notation of the Unified Modeling Language (UML) for the purpose of use case validation and clarity. To support use case validation, so data is added to the standard UML sequence diagram flows. To support clarity, the return data from an object’s method call is shown on the called method’s flow lines to save space.

This paper describes the notational changes needed when comparing the sequence diagrams with the use case text and UML class diagrams to provide more powerful support for use case validation. A simple example is given to illustrate how hidden classes or misplaced data can be found through this kind of valiadation.

These options are available to the analyst and it is the opinion of the author that proper use case correctness cannot easily be accomplished otherwise. XUML is also helpful for design and design validation using similar principles.

1. **Notational Extensions**

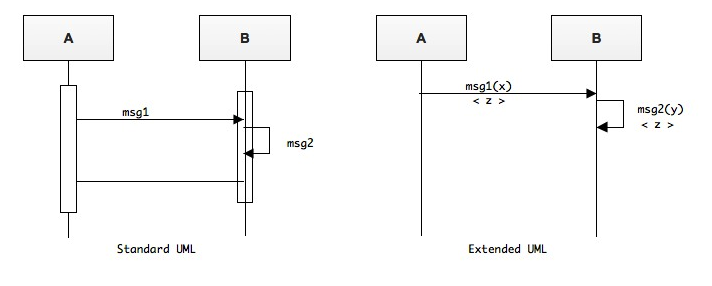
**Object Lifeline Boxes.** The first and most obvious difference is that the square boxes showing the lifetimes of objects are omitted from XUML. The background line behind the box is used instead. The boxes are not useful except for cases of object creation and destruction (memory allocation and deallocation, an implementation action), which is irrelevant for analysis. This is especially true for languages such as Java that have automatic garbage collection, in which the lifeline is not known. It is sufficient to merely create the object with the keyword new(…), which always returns the object to which it was sent.[[1]](#footnote-1)

**Message Passing (Called Methods).** Standard UML shows control flow (message passing) between two objects’ public methods as an arrow from the caller to the called object, with the name of the method on it. If the object calls itself (private method), then the line is a square loop back to itself, with the method on it.

Figure 1a shows object A calling the public method msg1() of object B, and object B calling its own private method msg2(). Data needed by the methods (input parameters) and data from the methods (return values) are not shown.

Figure 1b shows the XUML version of this transaction, augmented with input parameters and return values. B.msg1() requires parameter x, and returns value z; B.msg2() takes parameter y and returns value z.

The return value is shown under the method call in angle brackets instead of showing only an unadorned line from the called object, as with UML. Both UML and XUML show that control flow is returned to the caller, but only XUML shows what data is returned. If no data is returned, then the return is shown with empty angle brackets (< >). More on how XUML uses data in the flows are discussed in the validation section below.

****

**Figure 1a. Standard UML Figure 1b. Extended UML**

Objects are shown capitalized, but data is shown in lowerCamelCase. This is also illustrated in Figure 3a below.

**Method Encapsulation.** It is easy to see in Figure 1b that B.msg1(x) returns z because it was output from msg2(). However, object A does not need to know how object B implements its methods. For an entire use case, messages are chained from object to object by a sequence of method calls. During analysis, when only the problem domain objects are used (or known), calls can be made to implementation *objects* without caring about the implementation *methods*; each object’s public interface is defined in terms of method signatures and return values. The implementation methods can be defined later during design, but the required problem domain interface (API) is defined during analysis, and assists with the design tasks.

**3. Traversing the Sequence Flow.**

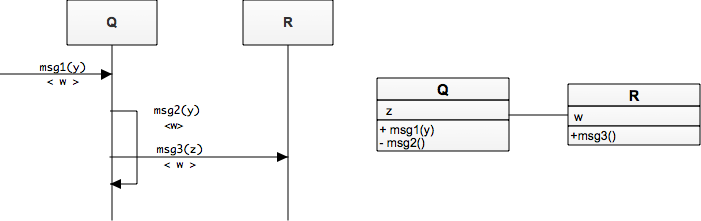
**UML sequence diagrams** reflect time flow from left to right, and top to bottom: messages to the left and above others are invoked sooner. Putting the return value directly under the method call bothers some people because it is perceived as violating that flow rule. However, if one looks at the method call as a call to an object with encapsulated method calls, such as implementation methods, then the rule is upheld.

For example, Figure 1a shows B.msg1() being called, and a second line showing it returning later. When adding the private implementation method B.msg2(), it would be inserted between the incoming and outgoing control lines, as if it were added later.

**Single-Stroke Rule**. Usually understanding the flow between called and caller objects is clear enough from context, but in the rare case that it isn’t, a simple rule can be used to show what is called (or returned) when. While traversing the method calls from top left to right, whenever an arrowhead is encountered vertically on the object lifeline, the control flow comes back; whenever a tail is encountered, the control follows that arrow. Of course, if there are no more method calls on the object lifeline, the control automatically returns. The ATM sequence diagram example In Figure 3 below includes numbers to show how the control flows.

**Internal Methods.** Sometimes a private method may make a public method call to another object. In Figure 2a below, a call is made to Q’s public method msg1(y), which calls its private method Q.msg2(y), which needs to call the public method R.msg3(z). The public call-to-R arrow is drawn from within the private call arrow to the other object, and the loop return is extended to embrace the public call arrow.

Figure 2a shows the private method msg2(y) returning w, but R.msg3(z) must be called to get w. Figure 2b shows the corresponding class diagram for Figure 2a. The data z is assumed here to be an attribute of Q, but it may also have been passed to Q previously.



**Figure 2a. Public method call Figure 2b. Class Diagram for Figure 2a.**

**from a private method call**

**Optional Methods.** In general, optional flows are not shown in sequence diagrams—if it is needed once, it must still be validated, and shown. In cases where the analyst feels he or she must show flow bifurcation, then a small circle is placed on the tail of the call arrow, with the option condition to the left of the circle. The ATM sequence diagram example (Figure 3) below includes several method calls with different kinds of return value options, including a method call with a Boolean result <T/F> from which the flow may go along the TRUE path, or along the FALSE path. Error conditions are frequently shown as <OK/ERR> return values; exception conditions and events are shown with dotted lines as in standard UML.

**4. Class Diagrams.**

XUML adds no extensions to UML for modeling classes. At the analysis levels, data types and method return types are not shown in the class diagram.

Each piece of passed data must be in the object using it, either because it is an attribute, it is passed in, or it is calculated. During analysis, calculated and passed data are not shown in the class, so the class diagram shows only the attributes used. Class diagrams also do not show CRUD services—Create, Read, Update, Delete, which means constructors without parameter are not shown, and setters and getters are not shown.

**5. Use Case Validation**

Use cases express functional decomposition, and class diagrams express scope-of-control decomposition (responsibility, behavior, and state). Both kinds of decomposition are orthogonal, which means that they have a mathematical basis for crosschecking each other, similar to the way a spreadsheet sums rows and columns to identify errors. The sequence diagram allows the analyst to execute the use case through the objects modeled by the class diagram, a kind of abstract desktop simulation of the use case through the system’s problem domain. The objects of the sequence diagram must be consistent with the classes of the class diagram, and consistent with the transactional flow of the use case. Any discrepancies in methods or data indicate an error in one of those artifacts. In my experience, the use case is in error most of the time, and about 30% of a use case will undergo changes as a result of XUML validation.

**Comparing Sequence and Class Diagrams.** An easy way to compare data and methods of the class diagram:

1. Select a class for consideration, and its object in the sequence diagram.
2. For each flow line tail on the object’s lifeline: Any data that is passed or returned must be shown in the class diagram as an attribute, unless it is calculated or passed in. If it is missing and it definitely needed, either add it, find the missing calculation method, or find what object owns that data from which it must be passed.
3. For each flow line arrowhead on the object’s lifeline: Any method that is called must be shown on the class diagram. Messages between objects are public methods; looped lines indicate private methods. Of course, CRUD services are not shown in the class diagram.

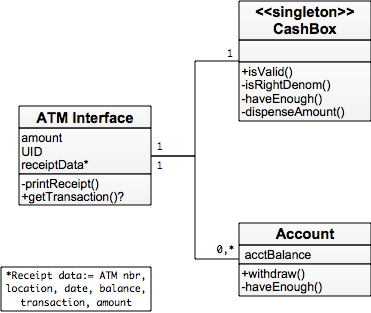
Use case correctness is achieved when the class diagram’s methods and attributes match the sequence diagram’s methods and data for each path of the use case

A note of warning: Some people will develop the class model by building a sequence diagram reflecting the use case. This seems to work at first, but there is no validation between the class model and the functional use case because one is merely a reflection of the other. This class model will tend to contain a high number of function bags instead of proper classes, and more rework will be required to remove use case errors later.

**6. Example: ATM Withdraw Transaction**

An ATM problem example is so pervasive in object literature that it is almost obligatory. The following “Withdraw money from an ATM” example illustrates the ideas above, plus a few others. See Figure 3. The numbers shown are not part of XUML, but are entirely for clarification to show in what sequence the methods and their values are returned.

**ATM Withdraw Use Case**: Withdraw a requested amount of money from the customer account, assuming that the cashbox and account has sufficient funds, and that the amount is in the correct denomination. (These days $20 bills often are the smallest denomination in the ATM.)



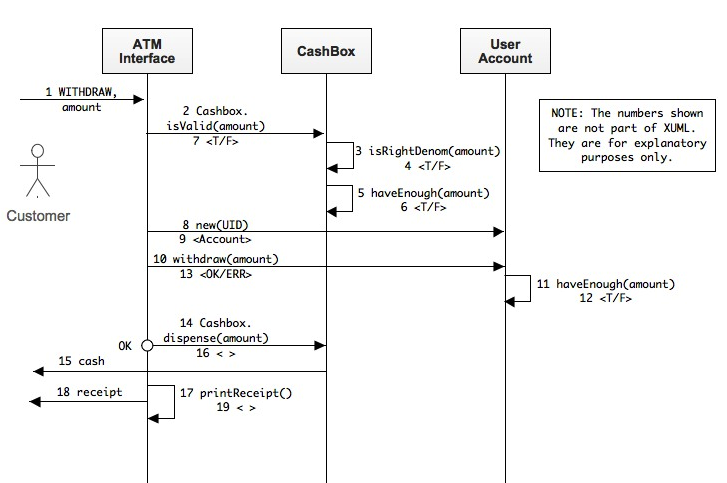
**Figure 3. First Pass: class diagram supporting ATM Withdraw use case**

**ATM Class Diagram**. Compare this use case’s sequence diagram with its class diagram, shown in Figure 3b. To develop the class diagram, we may think along these lines.

1. There is only one cashbox in an ATM, so a cashbox abstraction object will be a singleton across multiple sessions. We don’t know what kinds of data it needs yet, but it should have methods to dispense money and validate withdrawal transactions.
2. Each customer may only take from their own account, so the customer’s user ID (UID) and account balance must be maintained within this account. It must have methods to validate the request and deduct the money (withdraw). *NOTE: In this diagram, the UID is in the wrong place as reflected by our sequence diagram. The discrepancy says that we must adjust who owns that data.*
3. The ATM must coordinate the transaction and interact with the user. It is responsible for collecting the correct command and withdraw amount, and ensuring valid withdraw requests that result in providing the customer money and a receipt from the proper account. It will “own” the withdraw amount, the UID to get the right account, and the data for the receipt. Note that, as with UML, small notes can be placed outside the classes to represent passive data as is found on the receipt.
4. The ATM Interface uses the CashBox and the Account objects to fulfill its responsibility, so those relationships are shown. The ATM may interface to all accounts for all its customers, thus the 0: \* relationship.

**ATM Withdraw Sequence Diagram.** Figure 3 shows the sequence diagram for this use case. This example is for illustrative purposes only; there is no need for the reader to evaluate whether the design is a good one or not.

Step 1. The user enters the WITHDRAW command and the amount of money requested into the ATM Interface. No parentheses are shown because the Customer (person) cannot make method calls into objects. (There is probably a method called getTransaction() that collects the customer requests.)



**Figure 4. ATM Withdraw sequence Diagram in XUML**

Steps 2-6. The CashBox is a singleton, so the ATM makes a static method call (at least in Java) to check that the withdraw request is valid. The CashBox calls its own private methods isRightDenom(amount) to verify correct denominations are available, and haveEnough(amount) to verify if there is enough money in the CashBox to dispense the requested amount.

Step7-9. The CashBox says the request is valid, so the ATM interface creates a new instance of the user’s Account, passing the user ID (UID) with the constructor. The new keyword always calls an object’s constructor and returns it.

Step 10-13. Now that the ATM Interface has access to the right account, it attempts to withdraw the amount. The Account first checks for sufficient funds with its private method haveEnough(amount).

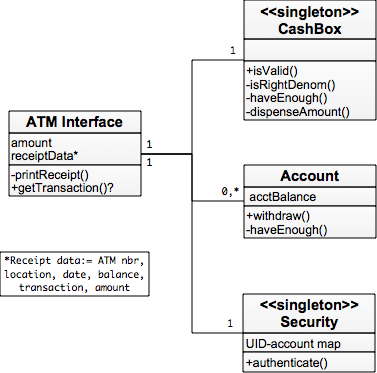
Step 14. Up until now, error conditions were ignored, and the flow walked the use case’s “happy path.” The optional circle is shown on the next method call to emphasize that money will not be dispensed unless all verifications are true. On this path, the static call CashBox.dispense(amount) is called. It is static because ATM Interface does not have a reference to CashBox (that we know of); it never called CashBox’s constructor. It is possible that an earlier use case (e.g. Logon) got the reference to CashBox, but that would show in the pre-condition or invariant of the use case.

Step 15-16. CashBox dispenses cash to the customer, as shown by the arrow back to the customer. The single-stroke rule says that the flow continues from the tail of the arrow before returning back along the CashBox.dispense() method call.

Step 17-19. After the CashBox method returns, the ATM Interface calls its private printReceipt() method. All data needed for the receipt is in the ATM Interface object, since nothing was passed. The receipt is printed, the method returns, and the use case ends.

**But what about that misplaced UID?** The sequence diagram shows that the UID was needed to instantiate the correct account. At first though, many novices reasonably place the UID in the account, but the ATM Interface wouldn’t know which account to create—it cannot get to a piece of data in an uninstantiated object. How does the ATM Interface get the UID? In all likelihood, the UID is best captured from a Security class that keeps the UID and account mappings. See Figure 5.

During the user’s session login use case, the security agent authenticates the user login data and returns the proper UID. Since analysis rules dictate that only one class can own the same piece of data, the Security class owns the UID. To create an instance of the user account, the ATM Interface would save the UID during the login use case and use it later. The case of the misplaced UID shows how these two approaches force the analyst to rethink his or her analysis to get a consistent solution between the use case, sequence diagram, and class diagram.

****

**Figure 5. Second Pass: class diagram supporting ATM Withdraw use case**

**7. Design Extensions**

This kind of use case-to-object model validation can be performed at the design level too. Design level extends the analysis level by adding implementation classes, methods, and non-problem domain classes (e.g. persistence objects and GUI widgets). It also adds data types for parameters and return values. Due to the increased features in design, the design diagrams are 3.5 times more complex (in my experience), although some have reported up to ten times more complex. Design level validation is worth the time and effort if quality is a high priority.

**8. Conclusion**

XUML is another option for the analyst or designer, another tool to put into their belts. Standard UML is a fine tool, and all other rules of UML apply to what was said here, but validating use cases for correctness requires XUML at the analysis level because of the data flow checking. Over half of a software product’s defects are in the requirements, so the quality of the product can be more than doubly improved simply by using this approach to build a consistent and unified object model.

1. The ellipsis in the method call indicates that the parameters are not known, or perhaps no parameters are passed. [↑](#footnote-ref-1)