

Focus on UML Testing Strategies – Part 2

In our last article we introduced the subject of UML test strategies; in part 2 we explore UML requirements models, problems associated with them and the need to review or inspect them before any test design work commences.

The UML functional requirements model consists primarily of Use Cases (mandatory) supported by scenarios, activity and sequence diagrams and statechart diagrams (optional). It is critical as it drives the entire design and development process; errors unchecked here will propagate through the lifecycle and cause significant damage. Testers use this model in one of two ways depending on how it is created:

1. If Use Cases are used to capture requirements then they are used for Validation; the Use Case model is the authority for user and acceptance testing.
2. If the Use Cases are derived from existing requirements definitions then they are used for Verification; the Use Case model is the authority for systems testing.

These forms of testing are different. Validation is running the system as a user would for end-to-end processing (probably covering many Use Cases); it is answering the question of have we built the correct system? Verification is running individual functions (use cases) to test they have been implemented properly; have we built the system correctly?

From the tester's perspective a good requirements model should clearly and completely describe the application functions of the system so that tests are relatively easy to design. The model should be independent of the system's design and implementation. Is this the case with UML?

First a Use Case is defined as

.... a kind of classifier representing a coherent unit of functionality provided by a system, a subsystem, or a class as manifested by sequences of messages exchanged among the system (subsystem, class) and one or more outside interactors (called *actors*) together with actions performed by the system (subsystem, class). (UML 1.4)

In plainer English it is a single, well-defined business or application function; e.g. the actions of a person depositing money in a bank account. Clearly functions vary in their size and complexity so Use Cases may vary significantly in these respects; there is no preferred size or number. Applications we have seen are, typically, decomposed into between 50 and 200 Use Cases. Use Cases may range from a few lines of action to some that represent large and complex functions. Levels of description vary enormously too, some specifying very detailed user actions and others giving abstract descriptions of major processing tasks.

UML allows Use Cases to have relationships, of which there are two types:

1. A piece of behaviour common to several Use Cases can be defined in a separate Use Case that the others "use".
2. Optional behaviour can be put in a Use Case that "includes" or "extends" the behaviour of other Use Cases.

The problem is that analysts lacking in skill or experience or with a traditional mindset can misuse Use Cases to try and re-create some sort of modular design. The Use Case model can take on characteristics of an implementation model thereby skipping the design stage. When a combination of decomposition and relationship requires design; that is the Use Case model breaks the rule that it should be independent of any formal design.

For an example, look at figure 1 below. Here we have use cases describing the relationship between a cat, a human and their food. We have added a curious feature that seems permissible; the Wash Cutlery Use Case is used by Feed Cat (non-optional behaviour) but extends the Cook Beans on Toast Use Case (washing up is optional).

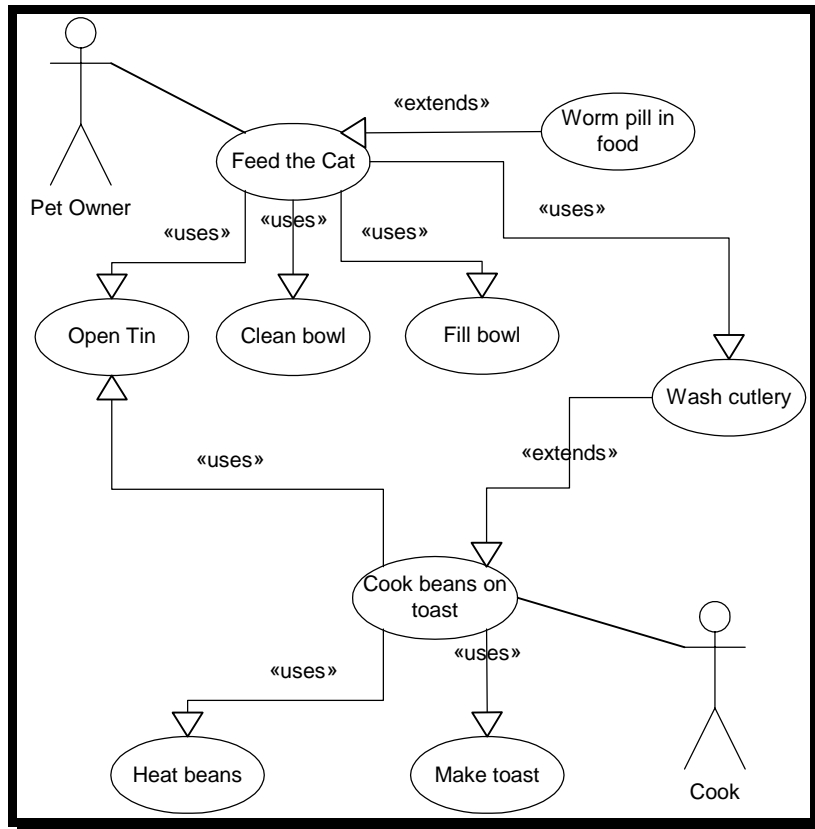


Figure 1 – the Feeding Use Case model

The GOTO problem

With structured programming we were able to banish “gotos”. Structured code provides explicit control flow structures we can test; gotos create implicit structures when control can disappear into oblivion. However UML has brought them back in another form.

Note: UML used badly can encourage use of poor design. Used well it supports good design.

DO THIS!
 Implement standards for use cases that encourage good practice in design and testing.
 Review use cases for “goto”- like behaviour

When a “uses” Use Case is invoked we have the equivalent of a goto and the point where control flow returns is a comefrom, but there is no guarantee of return.

With an include, once the condition for execution has been satisfied, the extending Use Case takes control and should return control to the extended Use Case at the same point. However it is possible to design control flow paths that are difficult to identify, let alone test. For

example, a Use Case might be “extended” by an exception handling routine, but that routine may not, depending on other conditions, return control back to its point of invocation. This jumping in and out of Use Case processing flows can hide significant control flow complexity that testers can only identify by completely flattening the model. This is a major task; why should we be faced with it?

Use of optional constructs and informal language

Note: UML has optional as well as mandatory constructs. Some of these allow informal, ambiguous modelling.

DO THIS!

Implement standards choice of and use of the optional constructs that support good design and testing.

Review use cases for ambiguous language

Review all constructs proper definition of dynamic behaviours

Another concern is that processing flow descriptions within a Use Case are narrative language, which is informal and error-prone. UML provides scenarios to flesh out important processes in more detail:

- activity diagrams to model the flows (akin to flowcharting)
- sequence diagrams to showing temporal (timing) relationships
- statechart diagrams are available if there is significant state behaviour.

The extent to which analysts use these optional models appears to vary widely. However there is always the temptation to perform the minimum amount of modelling. The problem is that important dynamic behaviour may go un-modelled in the requirements, and be left to designers to work out. This is too late for this work and can lead to incorrect assumptions by the designers. The problem for testers is that the requirements model lacks enough description of the functions and they need to delve into the design model to find it.

Pre and post conditions consistency

Note: UML does not enforce consistency – we need to enforce consistency for ourselves.

DO THIS!

Review pre and post conditions for consistency across all the use cases

Focus on interfaces for review and testing

During review, relate control flow from use case extensions to post conditions

Another problem relates to pre and post conditions, and has several flavours. The pre-conditions in a Use Case apply to all processing flows within it. If flows “use” other Use Cases they too should have the same pre-conditions. But how do we ensure consistency when many Use Cases are “using” many others? This appears a form of interface problem, and we know interfaces are a good source of error. One would expect the same to apply to extending Use Cases. With post-conditions they, too, appear to share this inheritance characteristic. However if an extending Use Case does not return control are the post-conditions of the Use Case that was being extended left in a valid state?

Size of a use case

Note: UML allows us to define use cases of any size – they can be too big or too small.

DO THIS!

*Review use cases for unrelated functionality
Review use cases for non-functional cohesion, e.g. temporal or logical
Review use cases for arbitrary decomposition or excessive relationships*

Consideration of pre and post conditions brings us back to the issue of how you create Use Cases that genuinely describe proper application functions. If a Use Case contains too much functionality that is not genuinely related then it will be difficult to identify meaningful pre and post conditions. An analogy from traditional design is the program that contains functions connected by non-domain characteristics. For example the Initialisation program that opens databases, broadcasts start up messages, checks system data and so on. This is called temporal cohesion, where time is the common factor, and where each function has its own pre and post conditions. Similarly, for a Use Case that describes only part of an application function it will be difficult to frame meaningful pre and post conditions because this type of decomposition is not domain-driven; it is largely arbitrary.

The discussion so far is based largely upon practical experience of working with Use Cases and associated models. Since 1996 UML has gone through several revisions, from version 1.1 to 1.4, each trying to address some of these issues. Where analysts and developers have realised the problems they have flagged them for others to investigate. However we are sure many will have fallen into the traps and designed Use Case models containing many errors of the types we describe.

How can testers respond?

So how do testers respond? We believe the points raised make it essential for testers to review or inspect UML requirements models at the earliest opportunity. We have to appreciate that many projects adopt incremental and iterative lifecycles. Therefore we need a review strategy that not only applies the right checks and tests, but also at the time when they will be most effective. Let us consider what we mean by the right checks.

We have identified several areas:

- 1. Is the model understandable with syntactic and semantic conformance to UML?*
- 2. Does the model represent all the behaviours and properties in the initial user requirements (if applicable)? Is there traceability between the two sets of information?*
- 3. Alternatively if the Use Case model is the highest-level requirements definition how do we confirm it has been fully understood and agreed by the users?*
- 4. Does the Use Case model contain any properties that make testing unnecessarily difficult or impossible? The model should adhere to design heuristics and exceptions should be documented. Non-functional objectives should not be overlooked.*
- 5. Is the Use Case model logically consistent with related artefacts, e.g. scenarios, statechart, activity and sequence diagrams? Are other models required?*
- 6. The Use Cases may be packaged for implementation and the interactions between packages should be small and obvious so that incremental testing can be achieved.*
- 7. Consider the 3Cs of Completeness, Correctness and Consistency throughout a review.*

Each area generates many specific tests and it impossible to describe them all here. As an example we will give some of the tests we have developed for area 4, as this is a big topic. It is essential that Use Case models follow the design principle of high functional cohesion; that is each Use Cases describes one clearly defined application function. Also they should follow another design principle of loose coupling; that is relationships between Use Cases, whether uses or includes, should be simple and not contain hidden internal dependencies. The following table lists some of the problems we look for in a Use Case model.

Design Problem	Comments
Monolithic Use Cases	<ul style="list-style-type: none"> ▪ There is too much behaviour in one place. ▪ A monolithic Use Case is akin to a god class in OO. ▪ A Use Case contains many business functions that do not necessarily have a business relationship. <p>These Use Cases create testability problems because they create artificial dependencies between unrelated functions. It is difficult to identify the specific attributes of a business function, e.g. pre and post conditions. Dynamic descriptions of state or sequence behaviour will also be compromised.</p>
Too many Use Cases	<ul style="list-style-type: none"> ▪ Use Cases describe individual bits of behaviour that do not represent a business function from the User's viewpoint. ▪ Many small Use Cases are needed to design a business-related test pattern. <p>The testability problem stems from the many interfaces generated by these small Use Cases – interfaces are a large source of error.</p>
Imbalance of abstraction	<ul style="list-style-type: none"> ▪ The Use Cases vary wildly in the level of detail they describe. ▪ Some are valid descriptions of business functions but others describe low-level detail that should be dealt with in the system's design. <p>The testability issue is that design-level detail cannot be specified at the requirements level and will usually be incorrect. Too much detail may cloud understanding of the model.</p>
Attribute-based Use Cases	<ul style="list-style-type: none"> ▪ The Use Cases do not describe How to do perform a business function from the User's viewpoint but how to Change some attribute of the system. ▪ Changing attributes is a part of Use Case behaviour but is not normally the justification for a Use Case. <p>The testability problem is that to perform a business-related test you need to identify all the attribute-based Use Cases that may apply, and their interfaces. Further problems arise if an attribute-based Use Case applies to a variety of business</p>

Design Problem	Comments
	functions – this should be modelled at the design level.
Incorrect relationships	<ul style="list-style-type: none"> Uses and extend relationships are incorrect or used incorrectly. These relationships are used in two circumstances: Extends - where special behaviour should be in a separate Use Case. Uses – where common behaviour should be in a separate Use Case. <p>Misuse of these relationships can lead to overly complex Use Cases that are difficult to test. They may also lead to co-variant behaviour (see later).</p>
Actors that are passive	<ul style="list-style-type: none"> A passive Actor is one that does not trigger behaviour and is only the recipient of the outcomes from a Use Case function. <p>They should not be Actors, but passive objects modelled elsewhere. The primary characteristic of an Actor is initiation of behaviour, so they are key points of reference for testers.</p>
Controller Use Cases	<ul style="list-style-type: none"> A controller Use Case has no function other than to pass control to another Use Case that will actually perform some business function. This Use Case does not describe a business function but how the system should handle processing. This is detail that should be resolved at the design level. <p>The testability problems arise from the Use Case model not being a description of discrete functions but of internal control flow mechanisms.</p>
Unstable, non-extensible	<ul style="list-style-type: none"> An unstable or non-extensible Use Case describes a function with a highly likelihood of future change, but in a way that does not support change. An example is a function tied to regulations or law subject to change, but where the Use Case makes assumptions otherwise. <p>Use Cases that are difficult to change create significant testability problems; the underlying changes are difficult to design and program and lead to unexpected errors through a ripple effect.</p>

DO THIS!
Prepare review checklists based on testability issues

Please note our aim is not to second-guess the analysts. Our primary objective is to ensure that the Use Case model is *testable*, whether for Validation or Verification. If the model does not contain the information we need to design tests, or contains unnecessary complexity through poor design then it may not be testable in realistic timeframes.

Experience of applying these and other tests has been encouraging. On one project a review of the first 20 Use Cases out of an anticipated total of 190 uncovered a number of major errors that were being repeated. The projection was that, without any change to the analysis process, the final model would contain 2000 errors waiting to go downstream into design and implementation.

This situation can create the un-recoverable “Lumberjack project” a North American colleague once described;

“This project is like a log floating down the river with a thousand ants on it, and each of them thinks they are steering it”.

In this case the analysts team was strengthened with a senior tester joining them as well.

The second issue was when to review or inspect, and this is a judgement issue dependant on the approach analysts take. They may build an abstract Use Case framework and gradually add detailed concrete Use Cases. This top-down approach is very difficult for testers as it may not be possible to perform effective reviews until late in the analysis phase. Alternatively they may model different areas in detail, starting from core functions and building upwards. This bottom-up approach enables early reviews. The reality is likely to be one of the many shades of grey in between.

DO THIS!

*As part of the strategy for the project and testing discuss and agree an order for the development, review and testing of use cases and other constructs
Expect to negotiate a compromise that meets the best interests (or the “least worst arrangement...”) of all parties.*

Conclusion

In conclusion UML is only a language and so there is huge room for discussion on how it can and should be used. There are no prescriptive solutions and none are intended here. We must remember that UML does not solve the systems analysis problem, which is one of the most complex and demanding tasks in IT.

To lighten the descending gloom we have cast on UML, one of our best experiences was a project where the Use Case model enabled the test team to complete all their test work to schedule and for the system to meet its operational deadline. It was hard work but generated that rare frisson of a successful project.

UML can be made to deliver if we expend the effort to make it so.

The authors:

Richard Warden is an independent consultant and founder of Software Futures Ltd. In 27 years experience he has worked in development, support, quality management and test in both technical and managerial roles. Since starting his company in 1991 he has added extensive experience in areas of job design and work motivation, process analysis and

development, and goal setting and metrics design. His UML experience includes nearly two years as a testing consultant for the Swiss Stock Exchange working on UML-based trading systems. Subsequently he spent another 15 months as principal test consultant for Sema4 Europe, an object-technology company. In conjunction with Isabel Evans he is developing UML testing techniques and associated training courses. He will present a UML testing seminar at the Unicom March 2003 DevTest. More information can be found at www.softwarefutures.ltd.uk.

Isabel Evans is Principal Consultant at IE Testing Consultancy Ltd. She has nearly 20 years in software quality and testing, as a practitioner, consultant and trainer. She is an active member of the BCS SIGIST Standards Working Party, particularly working on the usability, installability and conversion techniques. She will be presenting seminars on User Acceptance Testing and Software Quality and Testing for Project Managers at the Unicom March 2003 DevTest. More information can be found at www.ietesting.co.uk.