REQUIREMENTS ANALYSIS AND UML

USE CASES AND CLASS DIAGRAMS

UML PROVIDES A POWERFUL FRAMEWORK AND NOTATION FOR MODELLING BUSINESS PROCESSES AND OBJECTS. THIS TWO-PART ARTICLE FOCUSES ON USING UML TO UNDERSTAND BUSINESS REQUIREMENTS – 'WHAT' IS REQUIRED, RATHER THAN 'HOW' IT WILL BE ACHIEVED.

by Richard Vidgen

The unified modelling language (UML) has gained widespread acceptance as a notation for the analysis and design of software systems. UML will also support a broader notion of conceptual modelling; for example, in his book Holt (IEE, 2001) shows how UML can be used to analyse and model quality standards, such as ISO9001, and to support the definition of new business processes.

In response to the question 'why do we model?', Booch *et al.* (Addison Wesley, 1999) propose a fundamental reason: so that we can better understand the system we are developing. In his book Brian Wilson (Wiley, 1990) expands on the need to gain understanding by outlining four roles of conceptual modelling: to clarify our thinking about an area of concern; as an illustration of a concept; as an aid to defining structure and logic; and as a prerequisite to design. Although we might want to build models to better understand current business processes, we can also build models to represent different conceptualisations of the future, as might be the case when undertaking a radical change programme involving business process redesign. From a system development perspective, Booch argues that models not only help us visualise a system as it is or as we want it to be — they also allow us to specify the structure or behaviour of a system, provide a template to guide us when constructing a software system, and document the decisions we have made.

Before launching into a review of UML as a way of modelling business requirements, a note of caution should be raised. The question of why we model has been raised, but the status of the models themselves has not been considered. Booch argues that 'A model is a simplification of reality' and that 'the best models are connected to reality'. These definitions beg the question of what constitutes 'reality' and what form a 'connection' might take. More broadly, Brian Wilson defines a model as 'the explicit *interpretation* of one's understanding of a situation, or merely one's ideas about a situation... It may be prescriptive or illustrative, but above all it must be useful' (current author's emphasis). A model may indeed be a 'simplification of reality', but what to model and how to represent the situation are not neutral decisions. Models do not merely reflect reality (the 'as is') — they are also implicated in the construction of new realities ('to be'). However, we need also to be aware of the limitations of models and modelling notations. In modelling current situations and potential future realities we need to expose our assumptions, draw boundaries, and accept that there are personal, political and cultural aspects of the work situation that won't be expressed by the box and line diagrams of the systems analyst.

In this two-part article we will illustrate the use of UML techniques in the production of a requirements specification for an Internet ticket booking system at the fictional Barchester Playhouse. Although the aim is to produce a logical model of the Playhouse's requirements, the second part will also provide some pointers toward software system design.

THE UNIFIED MODELLING LANGUAGE

As object-oriented (OO) programming rose in popularity during the 1980s the ideas were taken from software development upstream into systems design and systems analysis. In the early 1990s a number of OO analysis and design methods were proposed. All had strengths and weaknesses: the Booch method (Benjamin Cummings, 1994) was strong on design and real-time applications, the object modelling technique (OMT) of Rumbaugh *et al.* (Prentice-Hall, 1991) on analysis and data-intensive applications, the use-case approach of Jacobson (Addison-Wesley, 1994) on business process modelling.

In 1994 Booch, Rumbaugh and Jacobson got together and pooled their ideas to create the unified modelling language, taking the best ideas from each and bringing some standardisation to the wide range of methods and notations for OO analysis and design emerging in the market. UML is now fast becoming an industry standard, has OMG (Object Management Group) acceptance, and a rich set of resources and software development tools available. Although many of the principles that underpin an OO approach will be illustrated as we work through the case study (for example, encapsulation and communication by messages), a thorough exposition is outside the scope of this article (see Vidgen *et al.*, Butterworth-Heinemann, 2003, for further details).

The core UML modelling techniques is illustrated using the development of a theatre ticket booking system at the fictional Barchester Playhouse. The Playhouse wants to start up an Internet ticketing facility in conjunction with a software house, Nimbus Information Systems. The plan is to make the theatre booking system an Internet application available to the two other theatres in Barchester, both of which rely currently on in-person and telephone ticket bookings. Once the system is operational it would be a relatively small step to make the service available to theatres in any part of the country.

USE CASE NOTATION

The starting point for modelling business requirements with the UML is the use case. Use case diagrams are a formalised notation for modelling the system from the perspective of the user. The focus is on what the system does — its behaviour — rather than how it achieves it. A use case typically represents some functionality of the proposed system as perceived by a user. Use cases will add business value, such as 'process ticket returns', and have a business outcome, such as 'returned tickets reallocated'. In the early stages of a development project it is important that use cases focus on business goals rather than system goals.

The use case notation comprises actors, use cases and associations (Fig. 1). An actor reflects a role that is played by a human (or non-human) with respect to a system. A role can be played by many people, e.g. doctor, and one person can play many roles, e.g., the theatre manager could play the role of box office clerk during busy periods and a box office clerk might also attend a performance as a member of the public. It →

FIG. 1 USE CASE DIAGRAMMING NOTATION

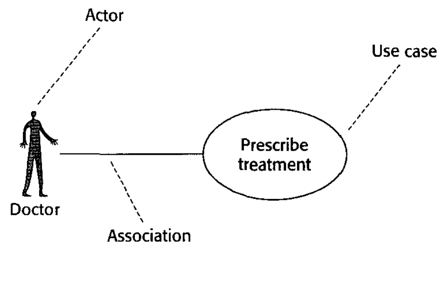
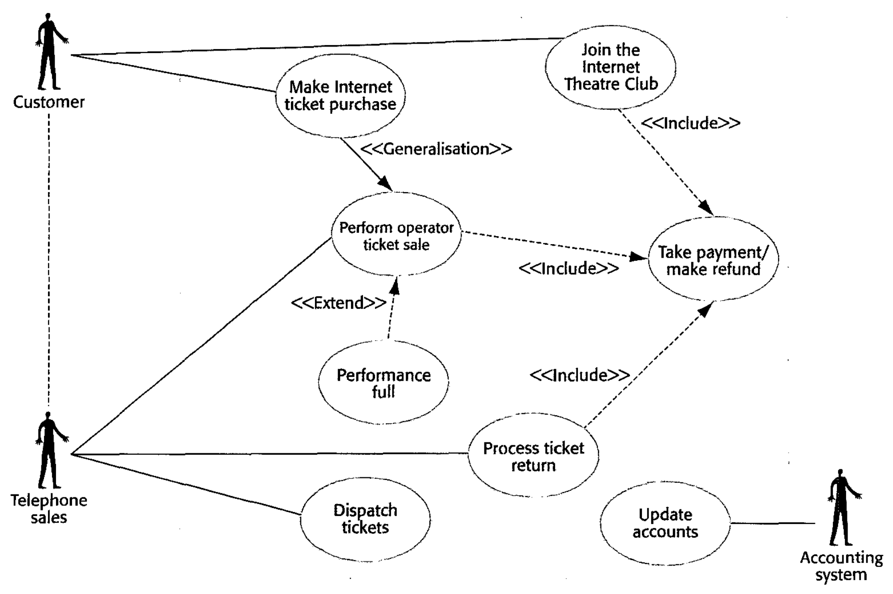


FIG. 2 USE CASE DIAGRAM FOR THEATRE OPERATIONS



is therefore important to think in terms of actors and roles rather than individuals and job titles. Actors execute use cases, such as a doctor prescribing treatment. The link between actors and use cases is shown by an association (Fig. 1), which indicates that there is communication between the actor and the use case, such as the sending and receiving of messages.

Fig. 2 shows a use case diagram for the box office of a theatre, such as the Barchester Playhouse. Human actors (we are not talking here about the actors in a play!) include customers and telephone sales operators. The accounting system is a non-human actor. It is external to the box office domain but it has requirements of the box office and is therefore shown as an actor.

Customers are associated with two use cases: 'make Internet ticket purchase' and 'join the Internet Theatre Club'. These will be activities that the customers can perform for themselves online via the Internet. It might be argued that the inclusion of a technology, the Internet, is misplaced in a conceptual model of the box office. However, although we might remove the word 'Internet' or replace it with 'online', from a business perspective it may be appropriate to make it absolutely clear that these will be Internet services available to customers. When developing a use case diagram it is important that the use cases are described from the reference point of the actor—for example, operators sell tickets, but customers purchase tickets on the Internet (from the Playhouse's perspective they are both ticket sale mechanisms).

EXTENSION, INCLUSION AND GENERALISTAION

Generally, it is best to first capture the use cases in simple terms, identifying the core set of use cases and actors. The use case diagram can then be refined to show three types of association between use cases: extension, inclusion and generalisation. Note that the arrows in Fig. 2 do not represent flows or process dependencies—these are more properly modelled using process flow diagrams such as the UML activity diagram.

• *Extends:* The extends relationship is used where there are similar use cases but one use case does more than the other. For example, a ticket sale where the performance is full (Fig. 2) requires that a variation on the ticket sale use case be carried out. For an extends relationship:

1 First, capture the simple, normal case, e.g., operator ticket sale.
2 Then, for each step in the use case ask what could go wrong, e.g.,
    Check seat availability (performance full)
    Take payment (credit card not authorised)

These variations can then be modelled as extensions of the base use case. An extends relationship models the part of a use case that the user may see as optional behaviour, such as the situation in Fig. 2 where the requested performance is full.

• *Include:* If multiple use cases require the same chunk of functionality then it can be made into a separate use case and referred to with an 'includes' relationship. For example, ticket sales and ticket returns both need to use the take payment use case, as does join Internet theatre club. This functionality can be separated out into a single use case rather than being duplicated in multiple use cases.

With 'extends' the actor will deal with the base case and the variations — the same operator will deal with ticket sales that can be fulfilled and ticket sales that cannot because no seats are available. With an 'includes' relationship a different actor might be responsible for making a refund for returned tickets (e.g. box office supervisor) from the actor involved in an operator ticket sale (e.g. box office operator).

• *Generalisation:* The Internet ticket purchase, where the customer interacts directly with the ticket booking system rather than using a telephone operator or box office clerk as an intermediary, can be modelled as a specialisation of the more general case of ticket sale. This would allow the Internet ticket purchase to inherit some of the general characteristics of the operator ticket sale and to add refinements as needed. For example, the operator would be able to see the theatre layout with available seats in one colour and booked seats in another colour. The Internet customer might not be allowed to see exactly which seats are booked (an empty theatre might put them off booking) and be allocated seats automatically by the system.

CLASS DIAGRAMS

Booch defines a class as 'a description of a set of objects that share the same attributes, operations, relationships and semantics'. A class diagram is a model of the things that are of interest in the problem domain being studied. In different domains there will be different things of interest; in modelling a university the developer might find classes such as 'Student', 'LectureCourse', and 'LectureRoom'. In the context of a theatre, classes for consideration could be 'Actor', 'Star', 'Theatre', 'Seat', 'Production', and so on. These are the conceptual categories that help us make sense when structuring our perceptions about the theatre situation. In UML classes are shown as rectangles (Fig. 3).

The class name should be a noun or noun phrase and begin with a capital letter. The class model represents things that last over time — the functionality of the system may change often, but a good class model will either cope with new functionality as is or be capable of being extended without a major redesign. It is unlikely that a 'correct' class diagram will be produced at the first attempt; the class diagram will evolve as the developer better understands the domain and the requirements. Classes can represent tangible things, such as the seats in a theatre, and intangible things, such as an account balance in an accounting system. Classes can also be used to model roles, such as the box office manager in a theatre.

CLASSES AND ASSOCIATIONS

Classes represent things; relationships represent the connections between things. UML caters for three types of relationship: association, generalisation and aggregation. An association is a structural relationship between things showing that one can navigate from the instances of one class to the instances of another (and possibly vice versa).

Associations are shown as solid lines that connect the same or different classes. In Fig. 3 the association could reflect the real-world business rule that 'a production *must* be staged by a *single* theatre'. The association can be read in two directions — the inverse in Fig. 3 would be that 'each theatre *may* stage *many* productions'. Each of the directions represents a *role of* the association; connections between two classes, known as binary associations, have two roles. Just as classes have instances, so do associations. An instance of the theatre/production association could be 'the theatre Barchester Playhouse stages a production of Hamlet'.

• *Multiplicity:* In the case of Fig. 3 the '1' indicates that a production must have one — and only one — →

FIG. 3 CLASSES AND ASSOCIATIONS

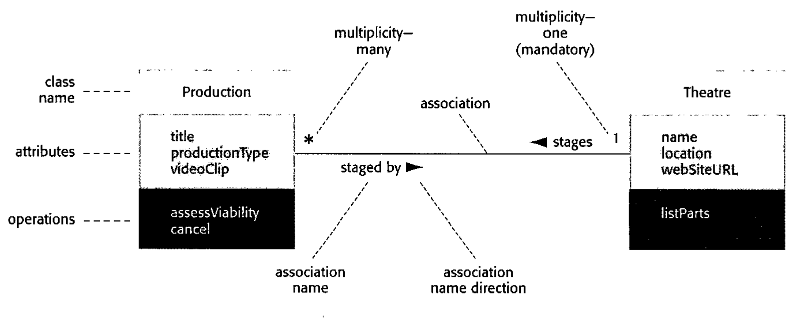
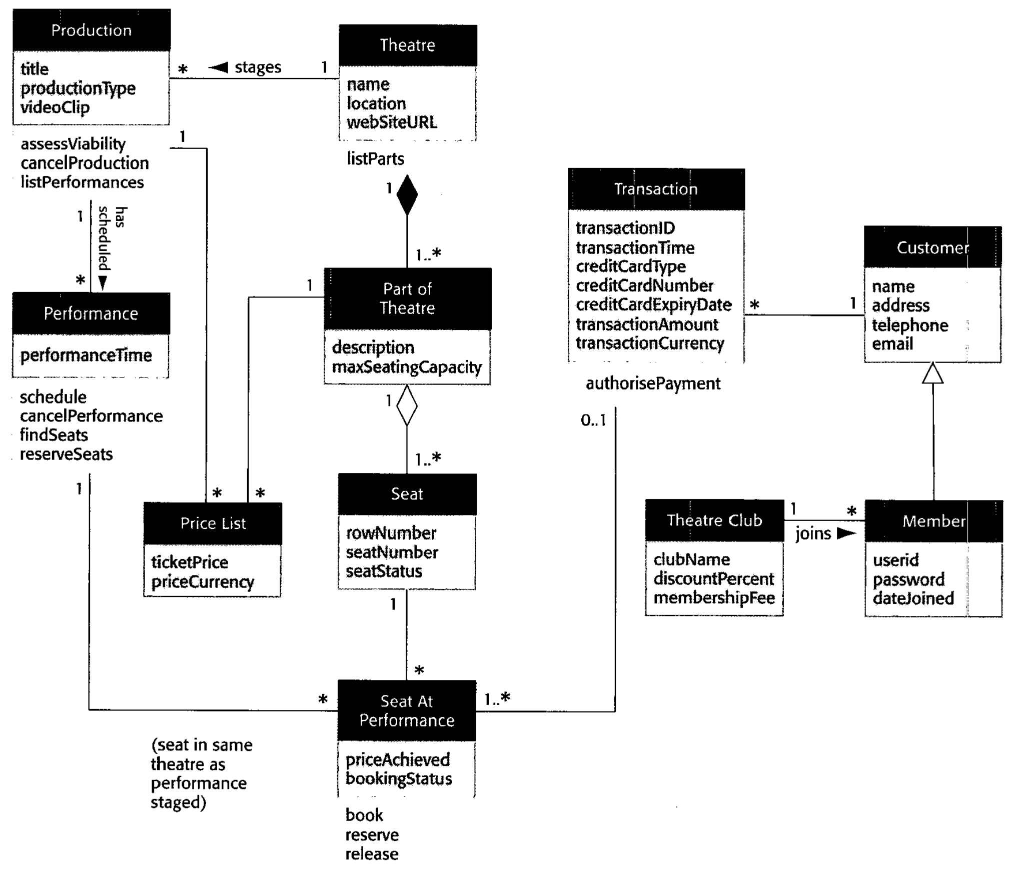


FIG. 4 CONCEPTUAL CLASS MODEL FOR THEATRE BOOKING SYSTEM



theatre. This does not stop another theatre staging a production of, for example, Hamlet, but this would be a different production and that would be involved in a separate instance of the 'stages' association. If it was optional for an production to be assigned to a theatre (e.g. in the early life of a production, before a theatre has been identified), then the association would be labelled '0..1', indicating that the minimum number of theatres a production can be allocated to is zero and the maximum number one.

ATTRIBUTES AND OPERATIONS

● *Attributes:* Booch defines an attribute as 'a named property of a class that describes a range of values that instances of the property may hold'. More intuitively, an attribute describes the instances of a class, e.g. every production would be expected to have a title and every theatre a name. Attributes are shown below the class name and each compound word should begin with a capital with the exception of the first word, e.g. productionType (Fig. 3). Some attributes will be mandatory, such as title, while others are optional, e.g. videoClip. Optionality is not usually shown on the class diagram, although it can be, e.g. videoClip(o).

● *Operations:* 'An operation is the implementation of a service that can be requested from any object of the class to affect behaviour' (Booch *et al.*). Operations are listed in the bottom compartment of the class box (Fig. 3). To invoke behaviour in an object, another object sends that object a message. For example, instances of the class Production support the operation 'assessViability'. A production object receiving this message will assess its viability and, let us assume, return a binary value: 'viable' or 'non-viable'. How the Production class implements this method is not the concern of the object sending the message. The production class might carry out the equivalent of flipping a coin, or it might make a forecast of bookings and compare this with production costs such as stage set design. The complexity of the implementation is hidden; the sender of the message invoking the operation need only know how the public interface is defined for the class Production.

THEATRE BOOKING SYSTEM CLASS MODEL

The conceptual class diagram for the theatre booking system is shown in Fig. 4. Although the model is basic it does support the core requirements of a simple ticketing system. At the heart of the model is the class SeatAtPerformance. Instances of this class tie together a performance and a part of a theatre, e.g. Hamlet at 8.00pm on 5 March 2003 in seat B15 of the circle of the Barchester Playhouse. To allow seats to be reserved prior to payment, the association between SeatAtPerformance and Transaction is marked as optional. The theatre booking system demonstrates all three types of relationships: associations, generalisation, and aggregation.

- *Generalisation:* Generalisation is a relationship between a general thing and a more specific thing. The more general thing is of a supertype class and the more specific thing is of a subtype class. In the theatre booking class model the class Member is shown as a subtype of class Customer. Some customers will be members and get benefits such as discounted ticket prices, but all members are customers. The benefit to the theatre booking system owner is that customers who register as members can subsequently be uniquely identified by their user id and targeted for promotions and relationship building. Instances of the class Member will inherit the characteristics (attributes, behaviours, and associations) of the class Customer while adding their own specialisations (attributes such as userid and an association with the class TheatreClub).
- *Aggregation:* The association and the generalisation are two types of UML relationship — the third type of association is the aggregation. The aggregation represents a 'whole/part' relationship. In practice it is often difficult to distinguish between associations and aggregations; in many cases the aggregation is just a strong form of association between two classes and is shown by an open diamond (e.g. the association between PartOfTheatre and Seat in Fig. 4). A composition, shown by a filled diamond, is a stronger form of aggregation. With a composition the parts live and die with the whole and cannot be transferred. For example, it does not make sense to move part of one theatre to another theatre. If a theatre is deleted then the parts of that theatre must go as well. But, one could move seats from one part of a theatre to another part of that theatre, or indeed to another theatre altogether.
- *Constraints:* A transaction is simply a device for grouping together seats at a performance for the purposes of payment. A transaction must have a customer, but some of the customers will be members of the theatre club and qualify for a discount on the ticket price. Because it is possible to get to the class theatre via two routes — performance and part of theatre — a constraint is needed to ensure that the theatre is the same via both routes for a given 'seat at performance'. This situation arises because in the early life of a production no seats have been sold and therefore an association with Theatre is needed via Performance and Production.

Note that the class model includes a Theatre class. This could be instantiated with a single theatre, e.g. the Barchester Playhouse, but it could also be instantiated with multiple theatres and therefore forms the basis for the theatre booking system — a generic multi-theatre booking system suitable for a theatre industry portal site. However, the model in Fig. 4 provides a basic facility for making Internet ticket sales, but does little more. A more sophisticated system might allow customers to go onto a wait list for performances or productions that are full. Furthermore, recursive classes are usually an essential feature of any reasonably complex model, e.g. to model organisation structures, but are outside scope of this article (see the book by Vidgen *et al.*, 2003 for further details).

THE BENEFIT TO THE OWNER IS THAT CUSTOMERS CAN BE IDENTIFIED AND TARGETED

BUSINESS REQUIREMENTS

The UML notation can be used to model business and organisational requirements from a conceptual perspective with aims that include gaining understanding of the current situation, envisioning new business processes, and the development of a structural and behavioural model that will support the development of a software system. In this article, use cases were used to model functions from a user perspective and class diagrams to lay out the structure of the 'things' in the situation. In part 2 (next issue) the behavioural aspects will be modelled using interaction diagrams and state transition diagrams.

ACKNOWLEDGMENTS

This article is reproduced with the permission of the publisher and is an abridged extract from: Vidgen, R. T., Avison, D. E., Wood, J. R. G. and Wood-Harper, A. T.: 'Developing Web Information Systems' (Butterworth-Heinemann, 2003) Further details of WISDM are available at www.wisdm.net together with a demonstration theatre ticket booking system developed using ColdFusion MX. ■

Richard Vidgen is with the School of Management, University of Bath, Bath BA2 7AY, UK, e-mail: mnsrtv@management.bath.ac.uk