JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 9, November–December 2005

The Rationale for OO Associations in Use Case Modelling

Peter Merrick, University of East Anglia, UK
Pat Barrow, University of East Anglia, UK

Abstract

This paper introduces the topic of use cases for the capture and representation of requirements and then focuses on the associations between use cases. Specifically it makes clear the difference between the include and extend associations, and then goes on to argue why use case modellers should incorporate the more familiar aggregation and specialisation associations between use cases for the explicit representation of models at different levels of goal abstraction. The modelling experience reported here is drawn from a number of industrial case studies that specifically employ early lifecycle use case modelling for the purpose of improving project delivery through improvements to IT procurement.

1 INTRODUCTION

For many years, having recognised the limitations of natural language for requirements' expression, Ivor Jacobson laboured on his ideas that would eventually come to be known as *use cases*. While working in Sweden for Ericcson, modelling telephone switches, those ideas came to fruition. He showed use cases to have significant benefits to customer and developer alike. Jacobson was so encouraged he left Ericcson and formed his own consulting company, Objectory, to apply more widely what had become known as Object Oriented Software Engineering (OOSE) (Jacobson, Jonsson et al.'92). Objectory came to have around 20 customers who continued to refine the notion of his new 'use case driven approach'. Objectory's customers included Swedish Defence, Ericsson Mobile, Ericsson Radar Electronics, and ABB, all of whom were able to demonstrate the approach's merit.

The use case approach is iterative; introducing complexity gradually (Jacobson, Jonsson et al.'92). This iterative process is in accordance with the way the task is actually performed by practitioners where requirements become more clear over time (Kulak and Guiney'00). As the process continues, more detail is accumulated. Eventually the analyst has enough information to transform the use case model into sequence diagrams. This transformation from one model to another type of model is a key factor in the attractiveness of UML. The transformation of models indicates the project is moving

Cite this article as follows: Peter Merrick and Pat Barrow: "The Rationale for OO Associations in Use Case Modelling", in *Journal of Object Technology*, vol. 4, no. 9, November-December 2005, pp. 123-142, http://www.jot.fm/issues/issue_2005_11/article4

closer to implementation, where instructions as to what should be coded are sufficiently detailed. Sequence diagrams are a more concrete realisation of the structure necessary to deliver the functionality described by use cases. They are the link between the problem representation and the first artefact of a proposed solution.

Use cases can be thought of as an initial model of the problem under consideration that sits at the centre of the 'developmental wheel'. They allow requirements to be represented in a graphical form that can be transformed into solution specific models from which code can be written. They can act in a project management capacity to keep track of progress throughout the developmental lifecycle. They can be included in commercial contracts, and can inform user documentation. Use cases can also be employed as test case scripts (Pooley and Stevens'99). They can be the basis of measuring progress, the basis of parcelling up work for different teams working in parallel, and the basis of traceability. By any measure, use cases are a versatile and flexible mechanism that offers a unique contribution to software engineering that features a solid route from project inception to implementation. Use cases have proved to be a notation that is understood by business and by technical staff (Pooley and Stevens'99), thereby providing a joint language of communication that suffers less from ambiguity thereby improving communication in a project team.

2 WHAT IS A USE CASE?

Many definitions have been postulated for the definition of a use case, yet still the approach has been criticised for a lack of formality in definition (Firesmith'02). Originally Jacobson described a use case as "a specific way of using the system by using some part of the functionality. [A use case] constitutes a complete course of inter-action that takes place between an actor and the system" (Jacobson, Jonsson et al.'92). Later, in the UML, the subject of a suitable definition was expanded when it was suggested that "a use case is the specification of sequences of actions, including variant sequences and error sequences that a system, sub-system, or class can perform by interacting with outside actors (Rumbaugh, Jacobson et al.'99). Philip Kruchten provides a short elegant definition when he says that "a use case yields an observable result of value to a particular actor (Kruchten'00). Martin Fowler sees a use case as "a typical interaction between a user and a computer system [that] captures some user-visible function [and] achieves a discrete goal for the user" (Fowler and Scott'99). Constantine and Lockwood provide a definition that is more complete at the expense of being less easily digestible. They say that a use case is a "single discrete, meaningful, well-defined task of interest to an external user in some role in relation to the system, comprising the user's intentions and system responsibilities in the course of accomplishing that task, described in abstract, technology free implementation-independent terms using the language of the application domain and of users in role" (Constantine and Lockwood'01).

A use case is a representation of a user goal to be satisfied. A system can be considered a collection of use cases together represented in a use case model. The use

case model is a picture intended to be easily ‘surveyable’ and changeable by customers and developers alike (Jacobson, Jonsson et al.'92). A use case model (UCM) has actors, task ovals, associations and a system boundary. UCMs start simple and become more complex over time (Pooley and Stevens'99). Each use case has two parts; a graphical representation and a textural representation. The text part adds detail to the graphical representation. It is convenient to consider the graphical component of a use case as a kind of table of contents that directs the reader to the accompanying text.

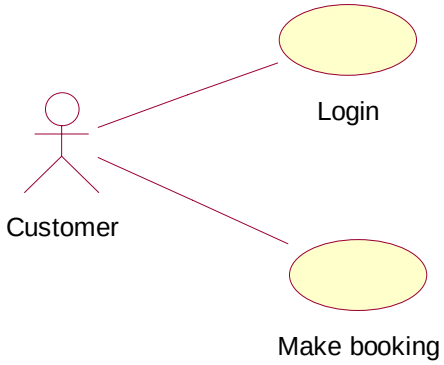


Figure 1: A simple use case model, featuring an actor, two task ovals, with associations.

In Figure 1, the most basic form of a use case is represented graphically by a named task oval that represents a user goal. This diagram illustrates a customer’s ability to *Login* and to *Make booking*. As more detail becomes available, it is added to the use case. Over time, the use case becomes primarily a textural construct that describes the system behaviour in semi-formalised natural language.

3 ACTORS

Use cases are triggered by actors. Jacobson defines an actor as a construct that represents a role a user may play (Jacobson, Jonsson et al.'92). An actor who triggers a use case is a human user who plays a well-defined role. Human actors are also known as primary actors. There are, in addition, secondary actors that represent other software systems with which communication must be established. Lastly, there are secondary actors that represent encapsulated behaviour; consider the concept of a *clock* actor that automatically triggers use cases, or an *email engine* that offers common behaviour to many individual use cases to aid communication. To Jacobson, actors are simple to discover, coming from an analysis of customers, partners, suppliers, authorities, and subsidiaries (Jacobson'95). Actors are considered outside the system boundary, although primary actors may come to have an internal representation even if it is on the trivial level of permissions, such as in the case of a notional *customer* who would need to have permission over their own account. It is useful to distinguish the ultimate primary actor; the actor who ‘really cares’ about the successful completion of a use case, rather than a proxy actor who is acting in

their stead (Cockburn'01). Thus, a clerk can be regarded as a proxy actor on behalf of a customer. This leads to the development of primary actor hierarchies which are related via specialisation.

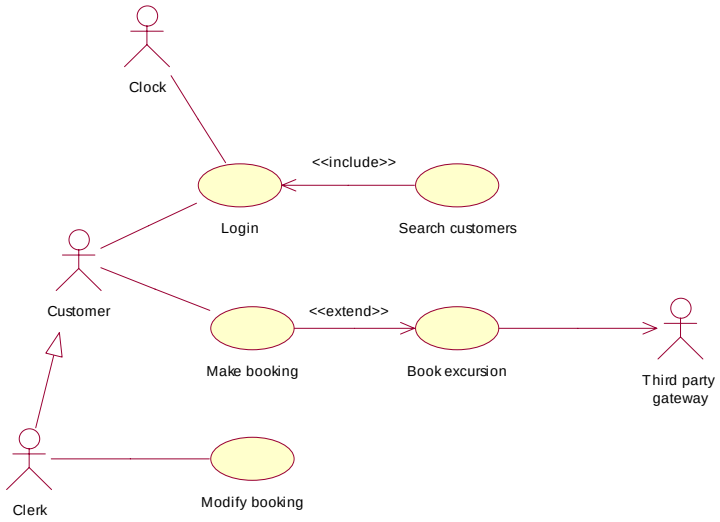


Figure 2: Actor types as part of a hierarchy. The *Clerk* and *Customer* are both primary (human) actors. The *Third party gateway* is a legacy system that accepts excursion bookings. The *Clock* is an automated convenience that logs users off the system when they have been inactive for a period of time.

Actors may be realised by devices, other systems, or by human beings; all three of which are illustrated in Figure 2. Actors are connected to use cases only via association, meaning they communicate by both sending and receiving information. Primary actors have a goal they wish to see fulfilled. Secondary actors assist in realising the goals of primary actors. A use case task oval is associated with a text description. An actor triggers a use case instance, termed a scenario, which performs a transaction on the actor's behalf. Cockburn considers that the basic course of action can be thought of as the 'archetype scenario' (Cockburn'01). Curiously, although the UML accepts that actors may be associated via specialisation, it does not define use cases as being capable of association via specialisation. This is an unfortunate limitation.

4 DEFINED ASSOCIATIONS BETWEEN USE CASES

In addition to the elements already described in a basic graphical use case (Figure 1) use cases themselves may also be associated through the <<include>> and <<extend>> stereotypes (Figure 3). This association between use cases is intended to be useful as a mechanism for reuse. Jacobson has recently postulated that such 'internal' use cases (not directly triggered by an actor) should not be considered as use cases in their own right,

but rather as use case *fragments* (Jacobson'03). In Figure 3, the <<include>> and <<extend>> association are illustrated.

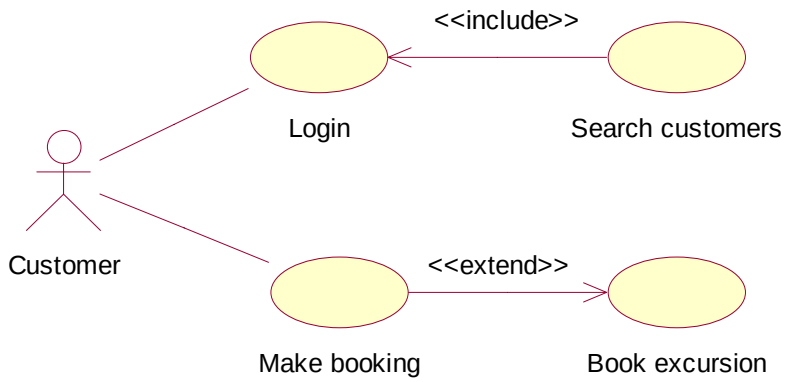


Figure 3: A simple use case model with an <<extend>> and an <<include>> stereotype.

Originally**,** use cases could be associated in two ways, either through the <<use>> or <<extend>> stereotypes (Jacobson, Jonsson et al.'92). In UML 1.3 the <<use>> tag was replaced with the <<include>> (Fowler and Scott'99) tag (probably to limit the word 'use' which had already acquired many connotations). Although Jacobson saw these stereotyped associations as being straightforward, they have since been the topic of considerable confusion and ambiguity. Jacobson believed the system modeller should associate two use cases with an <<extend>> stereotype when the extending use case represents additional behaviour. He suggests the extending use case should make sense on its own, that it represents new functionality being added. He believed this was a mechanism that would allow a system to grow (be extended) over time and where what was added would not require changes to what already existed (Jacobson, Jonsson et al.'92; Jacobson'03).

In considering the <<include>> association, Jacobson saw an included use case as making no sense on its own; that it must be *called* by the containing use case in order that it should perform something meaningful. It is sensible to extract *included* behaviour because it represents something common that can be reused (Jacobson, Jonsson et al.'92; Jacobson'03). Cockburn agrees, seeing <<include>> as employed when there exists a 'chunk' of behaviour that is similar across more than one use case and rather than defining behaviour over and over again it can be associated to a containing use case with an <<include>> (Cockburn'01).

On the face of it, these associations do not appear to represent concepts that are difficult to apply, but there is evidence to suggest that they are not well understand in practice (Cox'00). Cockburn says that extension is an option when one use case is similar to another but it does a bit more. This sounds suspiciously similar to the test employed in specialisation where the modeller has the choice to either break the new specialised use case out of the containing use case or to deal with the specialised behaviour as an alternative within the same use case. He gives no guidance as to when to use one

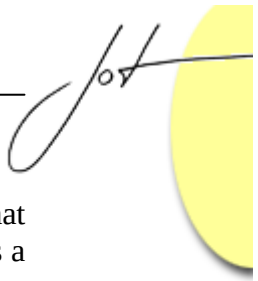
approach and when the other. By this definition a specialised use case may be said to describe *alternate* behaviour rendering the decision as to how the model should be constructed, unclear. The trouble comes when a modeller attempts to put the guidelines to the test; questions are uncovered to which no adequate answers are readily available. There is a problem in deciding how *alternative* behaviour should be modelled. Given that use cases are associated through <<include>> and <<extend>> associations they become use case *fragments*, rather than *first order* use cases in their own right. However, examples may be found where a use case may be both directly triggered by an actor in one situation, and also *called* through an *extend* or *include*. The rationale for employing an <<extend>> is easier to define when reference is made to implementation. According to one set of business rules it may be defined that an excursion can only be booked when it refers to a particular holiday. For example, booking an excursion is in addition to the normal case of booking a holiday; it is clearly additional behaviour. In execution the normal case executes through *Book Holiday*, and then the normal case may run in *Book Excursion* (Figure 7).

Light can be shed on this ambiguity by examining the differences between the <<include>> and <<extend>> stereotypes. They are not inverse associations. Where <<extend>> is taken to represent optional behaviour, the assumption that <<include>> represents mandatory behaviour is attractive, but incomplete. To reconcile the positions it is necessary to return to first principles. A use case has only one normal case where the user goal is *fully* satisfied. Alternatives are not equal to the normal case, as they represent another outcome which is *an* outcome (certainly not an error), but is not as successful as that of the normal case (the 'happy path' (Evans'99)). Alternate cases are therefore execution paths where the user goal is only partially satisfied.

Behaviour associated with an 'external' use case through <<extend>> takes the normal case behaviour further, thereby combining two normal cases. In this sense it is optional behaviour, but it is not optional to the execution of the base normal case; it is in addition.

Behaviour associated with an 'external' use case through <<include>> is mandatory to the execution of the normal case, and perhaps also to the alternate cases. Included behaviour is likely to become apparent over time, when the text cases are written, and is unlikely to be identified during *façade* modelling (a *façade* use case is one that features only the graphical component) (Kulak and Guiney'00). The modeller/author will notice that they are continually writing the same text; this becomes wearisome and so the decision is made to break the repetitive behaviour out into a fragment that can be associated through an <<include>> stereotype, if for no other reason then because it is more efficient in representation.

The subject of associations between use cases is full of ambiguity. This leads to confusion in model-making. Two modellers confronted with the same problem must inevitably create two different models. This leads to the problem of evaluating which model is superior and for what reason. The subject of associations between use cases has been largely accepted as a non-contentious subject by many but there is evidence the subject is not clear (Rosenberg and Scott'99). One author goes so far as to council that the



subject of stereotyped associations receives too much attention, that it is a distraction that can wait until an advanced state in the modelling (Pooley and Stevens'99), and that it is a fundamentally boring argument!

Glinz poses some of the most cogent criticisms of use case modelling (Glinz'00). Many of the rules that govern it are too restrictive and routinely broken. The rules (staff'99) were intended to encourage practitioners to model with use cases in a defined manner taken from an agreed specification written in wide consultation. Still, the restrictions are too onerous; compromising the richness of expression needed. This may be interpreted as a specific criticism of the omission of *inheritance* and *aggregation* from the use case association lexicon.

Use Case Discovery and Abstraction

The dictionary definitions of abstraction are intended for a broader audience than that of object modellers and systems analysts, yet it is helpful to return briefly to first principles before exploring the wider issues of abstraction in, specifically, use case expression. Abstraction is the 'act of leaving out of consideration one or more properties of a complex object so as to attend to others', according to the definition offered by Webster's Revised Unabridged Dictionary . From WordNet ® 1.6 abstraction is 'a concept or idea not associated with any specific instance' and 'the process of formulating general concepts by abstracting common properties of instances.' To the Free On-line Dictionary of Computing , abstraction is defined as 'generalisation; ignoring or hiding details to capture some kind of commonality between different instances.'

Representing use cases at the same level of functional abstraction is difficult (Pooley and Stevens'99). Use case modelling is full of different concepts that are abstractions to one extent or another. Within the world of object modelling, abstraction is generally understood as inheritance (generalisation/specialisation) and aggregation (whole-part structures).

A use case is primarily a mechanism for representing a goal abstracted to a level that makes the most sense to the user (or other stakeholder). One challenge is to define use cases that are comprehensible at the user level and useful at the developmental level. The question arises, how is it possible for the same construct to satisfy both? One way is to imagine use cases arranged in a hierarchical structure where they can be de-composed into sub-use cases (aggregation). Use cases can also be thought of from the perspective of inheritance. Although use cases are not defined as being able to apply either inheritance or aggregation, there are advantages of doing so.

Concerning inheritance, a hierarchy may be defined according to the strictures of an actor satisfying a goal expressed in the language of the domain. This allows representations such as Make booking, and also Make *Ferry* Booking, which are clearly related whereby one is more abstract than the other, but both are describing the same generalised action. Aggregation, in the form of a single user goal use case being sub-divided into several sub-goal use cases is equally a helpful concept to be able to apply

when modelling. A sub-goal use case may be of no interest to a user; being of interest only to the development team who are charged with implementation.

Abstraction is the omission of detail. In this sense, use cases are abstractions, as they begin with little detail and gain more as time passes. Kulak and Guiney describe this process of moving through a continuum of increasing detail in their four 'Fs' model of *façade, focused, filled* and *finished* stages of a use case (Kulak and Guiney'00). In each stage, a use case gains detail. In the beginning, the *façade* stage use case is solely a graphical construct (all use cases presented in this paper are *façades*). In the *finished* stage, a use case contains everything, including the completed text template.

The *façade* iteration includes the creation of a model, taken from the original problem statement. The problem statement is constructed from existing documentation and intellectual capital. It will have been used to canvas the sponsors' viewpoint and for the identification of users, stakeholders and customers. The problem statement is used to find actors and suggest domain entities. The tasks the identified actors want to fulfil are identified by understanding their job function (Kulak and Guiney'00). The purpose of the *façade* iteration is to create graphical *placeholders* or use case names and short descriptions characterised as including minimal detail.

The abstraction of use cases based on detail is not the end of the story. There is also the abstraction based on the project stakeholder or project goal. Consider the question, "how many use cases does the typical specification contain?" Jacobson believes a system will have between 10 and 20 use cases (Jacobson, Jonsson et al.'92; Jacobson'03). John Smith at Rational Software defines the ideal number to be between 10 and 50 (Smith'99). Smith states that a large number (100+), indicates a 'lapse into functional decomposition'[1]; a state that is to be avoided (Cockburn'01). However, Cantor in (Cantor'03) makes clear that different processes are referred to by the term 'functional decomposition' and that no universal definition is accepted. The numerical discrepancy in the identified range of the ideal number of use cases in a specification differs because use cases can be expressed at different levels of goal abstraction. Where a user goal use case is modelled with inheritance or aggregation, the effect will be to increase the overall number of use cases that appear on the model.

Knowing the components of a use case does not address the process by which use cases are discovered. To enter into the discovery of use cases it is helpful to consider them as being underpinned by a semi-formalised language based on grammatical constructs.

In considering the naming of a graphical use case, the task oval component must be uniquely identified. The form of the name should include an active verb, in the present tense employing the active voice that names a recognisable user goal (Rosenberg and Scott'99). Jacobson favours the employment of singular nouns and verbs in the infinitive

1 Functional decomposition: There is no official definition of functional decomposition. The term is used to describe several activities including adding more detail to a general requirement, organizing requirements into packages, and determining the organization of subsystems. There is likely to be little harm in the first two, which are ways to better manage requirements. However using functional decomposition to derive a system architecture is a bad strategy that may jeopardize the project.

(Jacobson, Jonsson et al.'92). There is a balance to be achieved in the selection of concrete vs. generalised verbs. Alongside the chosen verb goes a suitable noun or noun phrase (e.g. *Book excursion*). A use case name should be an instance of identifiable and acceptably atomic behaviour of the system. In practice, use case names are short active verb phrases naming some behaviour found in the vocabulary of the system being modelled. To name a use case, one should employ a simple verb/noun construction (Jacobson, Jonsson et al.'92). The objective is to avoid ambiguity in the naming of task ovals, successfully achieved when the name is self-evident to the majority of users (Constantine and Lockwood'01).

The user goal may be described at different levels of abstraction that may be characterised as specialisation/generalisation relationships. Thus, some use cases are abstract, incapable of being instantiated themselves (Jacobson, Jonsson et al.'92), but providing a hierarchical structure for efficient use case management.

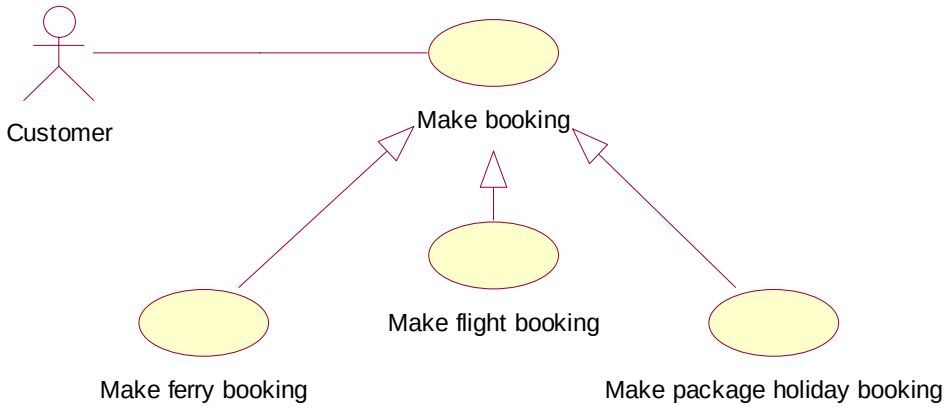


Figure 4: A use case structure where the specialisation association has been employed to add more information about the use case relationships.

In Figure 4, the *Make booking* use case is abstract (cannot be instantiated) but it acts as a convenient placeholder to illustrate that there is some commonality between the different types of possible bookings. The different booking types are represented by three *specialised* use cases with significant differences subject to different business rules in the detail of *Make ferry booking*, *Make flight booking* and *Make package holiday booking* over its parent *Make booking*. The customer is understood to trigger the specialisations rather than the parent use case. This is a succinct approach to diagram construction that would otherwise require the actor to be connected to each specialised use case, inevitably cluttering the diagram.

CRUD functionality

Most systems store and retrieve information (derived from stored data), a large class of which are transaction processing applications (Jackson'01; Sutcliffe'02). Different actors want to interact with the same data; some of whom have the access rights to make

changes, others who may only view data but not make changes. All data must be subject to the standard considerations that apply in relational database design, namely functionality to create, retrieve, update or delete (referred to as CRUD functionality) although there is no agreement on the degree of importance of use cases that perform this kind of management over data (Evans'99; Kulak and Guiney'00; Cockburn'01).

Cockburn provides a neat encapsulation of the argument around CRUD use cases when he considers the management of an imaginary entity called a 'frizzle' (Cockburn'01). He says, "So far there is no consensus on how to organise all those little use cases of the sort *Create a Frizzle*, *Retrieve a Frizzle*, *Update a Frizzle*, *Delete a Frizzle*. The question is, are they all part of one bigger use case, *Manage Frizzles*, or are they separate?" Cockburn specifically discusses this issue of CRUD representations, coming down on the side of 'manage' to represent the functions of create, retrieve, update and delete over a persistent entity. Other authors believe it is best to keep them separate to better aid identification of which actors have what permissions. Accepting that create, retrieve, update, delete should be kept separate has the disadvantage of multiplying the number of use cases that need to be tracked, and does nothing to improve the clarity of the resulting diagram. Given that *manage* use cases exist, the problem arises of where and how they fit into a use case model (Evans'99; Kulak and Guiney'00; Lilly'00; Biddle, Noble et al.'01; Cockburn'01). Although making available the functionality to *manage* data entities is necessary, this was not Jacobson's vision of how use case modelling should be performed. Therefore there is a tension in the articulation of use cases, between the representation of user goals at the *appropriate* level and at a *pragmatic* level. (CRUD use cases can be understood as pragmatic.)

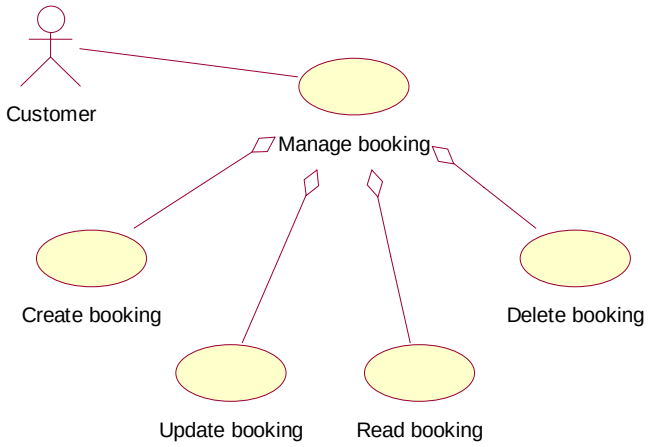


Figure 5: A set of CRUD use cases over the entity *booking*. The children use cases are shown related to the parent use case via the *aggregation* relationship.

Employing the *Manage* tag to represent CRUD functions can make the diagram less cluttered as in Figure 5. Aggregation has been employed to associate the sub-use cases to the container *Manage booking*. Sometimes it is necessary to show all the constituent use

cases contained in *Manage* to deal with actor permissions. This diagram implies the Customer has total control over a booking, a situation unlikely to exist in practice (no company would want a customer to be able to make unrestricted changes to an existing booking).

CRUD use cases cause controversy. Kulak and Guiney (Kulak and Guiney'00) state it is a mistake to begin use case modelling with *manage* use cases believing it to be a sign of over-engineering. However, accepting *manage* use cases exist as a first principle is fundamental to Biddle (Biddle, Noble et al.'01) simply for the sake of completeness. It is not essential to reuse the standard language of relational databases to describe maintenance functionality over an entity. If create, retrieve, update and delete do not capture the functionality required, the specifier should change the names to better reflect the domain. For instance there may be more than one way to create an entity, either by making a new one, or by copying an existing one. Providing utility functionality over entities should not be done slavishly, without first considering the nature of the discovered entity, however, it is likely to be an area of functional specification that will lend itself to a degree of automation (Biddle, Noble et al.'02). The full range of *manage* use cases are not required over all the identified domain objects, such as the example where a 'time period' is modelled as a first order entity (parent table) rather than as an attribute of some containing table. Thus, the essential task of ensuring the domain model is a minimal representation of the entities needed to manage the business rather than a fully normalised database that includes parent – child relations which are inappropriate at this early stage of modelling.

It is important in the beginning to understand how to begin collecting use cases, how much detail to collect, the level of abstraction to employ, in order to determine when it is safe to move on.

When counting use cases in a model, Smith employs the 'external use case' test. This is defined as a use case that has a direct association with an actor. Any use case that does not have a direct actor association is an internal use case (or fragment) that should not be counted as a *first order* use case. Unfortunately this convenient convention is exploded when one actor's external use case is another actor's internal use case. This is illustrated in Figure 6.

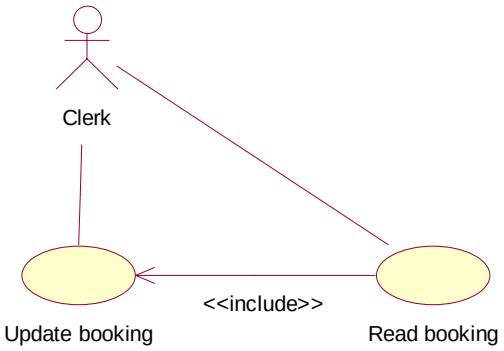


Figure 6: A sub-set of basic functionality triggered by the Clerk.

In Figure 6, the Clerk may either *Update booking*, which includes 'reading' over the set of all bookings (searching for a booking to find the correct one), or may have, as an ultimate goal in its own right, to find a booking so that some detail may be confirmed to a waiting customer. Therefore, Read booking is both an external and an internal use case.

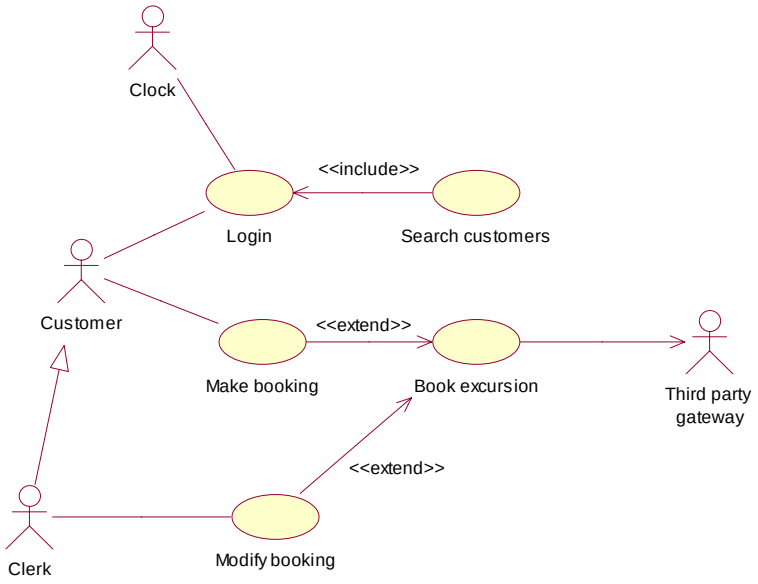


Figure 7: An extract of the total functionality delivered by the booking system.

Figure 7 provides a second example of how the external/internal use case categorisation can break down. *Book excursion* is a use case fragment from the perspective of the Customer, who can instantiate the use case as part of *Make booking*. To the Clerk, *Book excursion* is a specialisation of *Modify booking* and therefore a first order use case in its own right.

Projects have different stakeholders, who have different needs from the use case models they consume. From this perspective there is no right or wrong level of use case goal representation, but rather a requirement to see them being part of a hierarchical goal structure. This tension in goal expression has been addressed by Cockburn who has proposed a workable hierarchy that ranges through *Summary*, *User goal*, *Sub-goal* and *Too low*. Unfortunately, Cockburn does not provide extensive examples of his hierarchy at work, but does provide the foundation for navigating such a hierarchy based on goals where one form of expression is neither right nor wrong depending on the value offered to the model's audience (Cockburn'97; Cockburn'01). This hierarchy of goal decomposition will determine the number of use cases in a use case model, and accounts for differences of opinion. Evans warns against letting developers write high level use cases because they normally cannot help themselves but to delve into implementation detail that is inappropriate during the early stages (Evans'99). The need to be aware of

both abstraction of detail and abstraction of goal expression is necessary to successfully produce use case models for all stakeholders when and where appropriate.

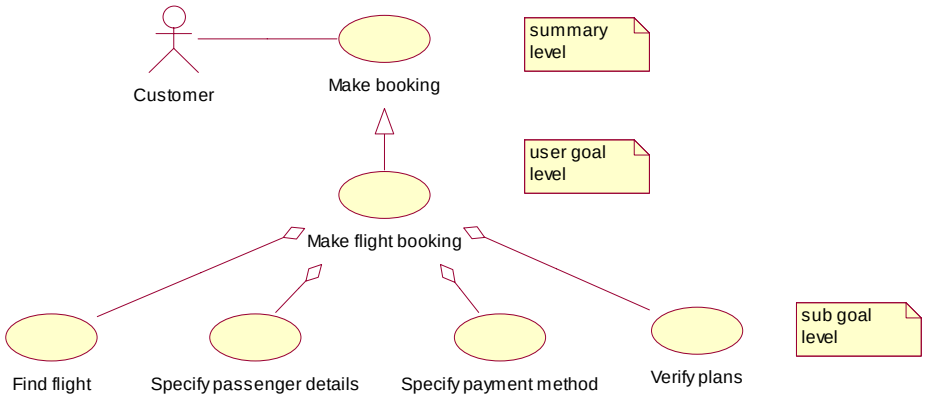


Figure 8: An example of Cockburn's hierarchy of abstraction of user goals in use cases. Strictly speaking, this type of association between use cases is not defined in the UML specification (staff'99). *Make flight booking* is shown as a specialisation of *Make booking*. *Make flight booking* is shown to be comprised of a series of use cases that combine to represent a whole-part relationship.

To summarise, there is no agreement in the literature on the nature of abstraction in use case modelling, although there are sound approaches that can be employed to better understand the issues a requirements modeller must satisfy. Use cases are represented in a goal hierarchy in Figure 8. One of the major problems in use case modelling is deciding on the correct goal 'level' at which to represent requirements. The evidence suggests they are all correct, depending on the need of the target stakeholder. The other debate centres around the inclusion of either too much or too little detail, and the necessity to represent the use case set similarly in order to preserve some 'homogeneity of expression'. Use case modelling can be thought of as having different, inter-related, axes of abstraction. On one axis, the abstraction of increasing detail as articulated by Kulak and Guiney, and on the other axis, goal expression, as articulated by Cockburn. Together they provide guidance to build comparable and consistent models (Merrick and Barrow'03).

5 OO ASSOCIATIONS IN USE CASE MODELLING

If the use case hierarchy of abstraction is accepted then guidelines have to be formalised for better use case representation. This is necessary because a use case at the summary level will be decomposed into the user goal level (and potentially further), requiring it to be unambiguous to the modeller when to employ specialisation, aggregation, *include* and *extend* notation. The UML standard itself defines only <<include>> and <<extend>>. The guidelines presented here are not part of the UML standard, but have been successfully employed in modelling to improve clarity of expression on a variety of assignments (Merrick'04; Merrick and Barrow'04c; Merrick and Barrow'04b; Merrick and Barrow'04a).

Aggregation

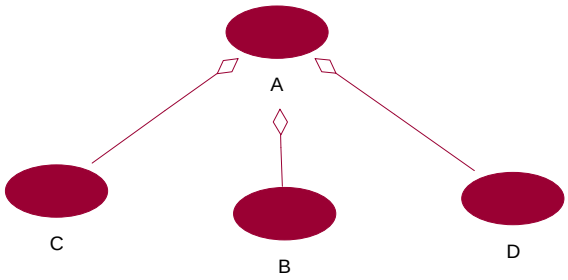


Figure 9: Aggregation is shown with a line adorned by a diamond. The use case most close to the diamond is the containing use case.

In Figure 9, use case ‘A’ is comprised of use cases ‘C’, ‘B’, and ‘D’. In set notation, set A is made up of the sets C, B and D or A:={C,B,D}. There is no time or order associated with this construction. By this convention, a *Manage Entity* (A) summary use case could be decomposed into a *New Entity* (B), *Modify Entity* (C), and *Search Entity* (D) use case set.

Inheritance

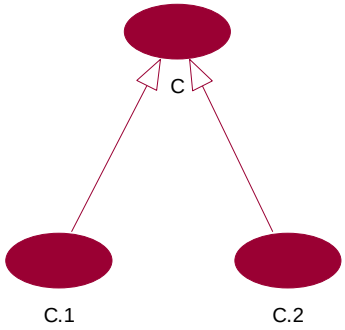


Figure 10: Specialisation (or inheritance, or generalisation) is represented in this diagram by a line annotated with a hollow headed arrow. The use case that is pointed to is the parent (C). It is specialised by its children use cases (C.1, C.2).

In Figure 10, use cases C.1 and C.2 are *types of* use case C (also known as an *is a* relationship). If use case C was a recipe for baking bread, use case C.1 might be a recipe for rye bread, and C.2 a recipe for brown. Use case C could describe the use of a bowl, water, and yeast. C.1 would describe the type of flour (rye), the proportion of water and the cooking time. So, C.1 and C.2 have things in common (contained in C) but they also have things that are unique to themselves. In a requirements model, C might represent the creation of a *New booking*, whereas C.1 would represent the creation of a *New ferry booking*. Use case C is abstract in this example.

Include

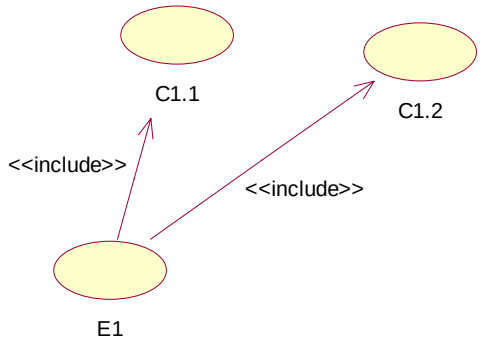


Figure 11: An <<include>> is a standard UML association represented by an open arrow that points away from the use case that is being included. Inclusion is a good way of allowing reuse.

In Figure 11 the use case E1 is included in use case C1.1 and C1.2. This way E1 can be written once and used twice, thereby encouraging reuse. For example, if use case C1.1 is *Modify Provisional Booking* and C1.2 is *Modify Confirmed Assignment*, as both of these use cases require the assignment to be found, use case E1 might be *Search booking*. This notation is employed with *concrete* use cases.

Extend

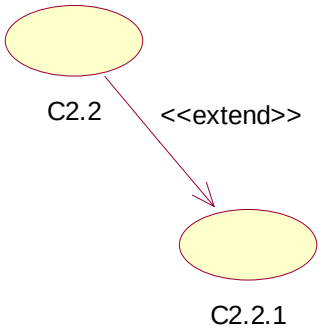


Figure 12: An <<extend>> is a standard UML association represented by an open arrow that points to the use case that is being employed as an extension. An extension increases the behaviour of the base use case

As both <<extend>> (Figure 12) and <<include>> (Figure 11) employ the same notation, there is plenty of opportunity for confusion. The difference between these two annotations is the direction the arrow points and the text adornment that seeks to make explicit which association is intended.

<<Extend>> implies optional behaviour; given the activation of use case C2.2, under certain conditions the behaviour in C2.2.1 will be called. This technique also helps with reuse (as does <<include>>). It is employed with concrete use cases.

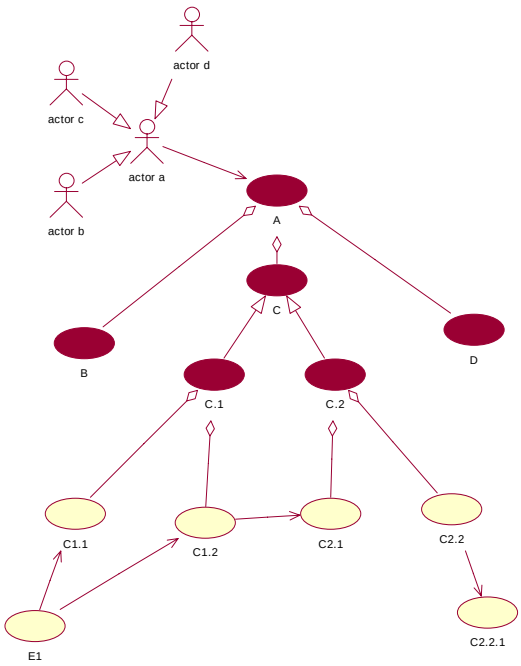


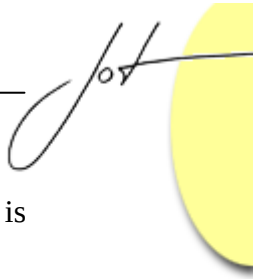
Figure 13: A complex use case model showing four kinds of use case association.

There are two different kinds of use case shown in Figure 13 which are distinguished by their colour and all four proposed use case associations are employed. In Figure 13 all of the associations, aggregation, specialisation, *include*, and *extend* are used in the same diagram. A dark coloured use case is abstract and is used in problem decomposition. An abstract use case has no use case text associated with it. Light coloured use cases are concrete and have use case text associated with them.

6 USE CASE MODELLING SUMMARY

Since the day Brookes wrote identifying the non-existence of a silver bullet to put all the things right that are wrong with software engineering (Brooks'86), practitioners have reacted with suspicion to any innovation that promised a holy grail. This holds true for use case modelling.

Glinz complains that by limiting inter-actor associations to exclude inheritance relationships the richness of the resulting models is excessively restricted. At times the specification simply seems incomplete as opposed to being deliberately restrictive. For instance, Glinz is critical of the actor model, with its emphasis on the definition of human



roles representing the only mechanism that can trigger a use case execution. Indeed this is too restrictive where the modeller wishes to define automated behaviour.

To some, the fact that use cases are not fundamentally an object-oriented notation is a particular issue (Firesmith'02), although because use cases are a notation for the representation of problems rather than solutions, this criticism may be unfounded. Given that use cases could be associated via *inheritance* and *aggregation*, (and there are good reasons for doing so) use case modelling would become more compliant with an over-arching OO philosophy. However, drawing any parallels between use case modelling and the eventual objects that are defined to satisfy the behaviour they represent must be resisted. Certainly use case modelling is founded on the principles of object modelling (Jacobson'03), which is sufficient, provided there is a recognised process for moving from the problem representation to an OO solution representation. To accomplish a mapping from problem to solution the interested reader is referred to the concepts embodied in the Model-View-Controller pattern (Krasner and Pope'88) and the work that has been done subsequently (Spielman'03). The UML specification does not expressly forbid the incorporation of useful modelling techniques such as *inheritance* and *aggregation* relationships between use cases. Certainly all of these concepts have been employed implicitly by Cockburn (Cockburn'01) and in other work done by the author with respect to early lifecycle use case modelling for the purpose of IT procurement (Merrick'04; Merrick and Barrow'04c; Merrick and Barrow'04b; Merrick and Barrow'04a). *Aggregation* and *inheritance* are perhaps best suited to modelling the abstraction between use case layers in the goal hierarchy, whereas <<include>> and <<extend>> are sufficient for modelling at the layer where the use case has a text component associated with it. At this point a use case takes on detail and becomes more concrete.

There are different audiences for use case models. Management may wish to engage with use cases at the *summary* level, users at the *user goal* level, architects at the *sub-goal* level and technicians at the *too low* level. Contrary to Jacobson's original vision, there is no harm in allowing this multiple view of functionality provided the views can be reconciled. Incorporating aggregation and specialisation into use case modelling does that, and thereby serves the needs of various project stakeholders more faithfully.

REFERENCES

Dictionary.com, www.dictionary.com

Free On-Line Dictionary of Computing (FOLDOC), http://foldoc.doc.ic.ac.uk

WordNet, www.cogsci.princeton.edu

Biddle, R., J. Noble and E. Tempero, "Patterns for Essential Use Cases." *Australasian Pattern Languages of Programming (KoalaPLoP)* (Melbourne, Australia, 2001), pp

Biddle, R., J. Noble and E. Tempero, "Supporting Reusable Use Cases." *International Conference on Software Reuse (ICSR'02)* (Austin, Texas, 2002), pp p. 210-226 Springer Verlag.

Brooks, F. P., "No Silver Bullet." *Information Processing, Elsevier Science*, 1986.

(generic ref) Cantor, M., Thoughts on Functional Decomposition, 2003,

Cockburn, A., Structuring Use Cases with Goals, 1997, http://members.aol.com/acockburn/papers/usecases.htm

Cockburn, A., *Writing Effective Use Cases*. Addison Wesley Longman, Upper Saddle River, N.J., 2001.

Constantine, L. and L. Lockwood (2001). Structure and Style in Use Cases for User Interface Design. *Object Modeling and User Interface Design*. M. Van Harmelen. Upper Saddle River, N.J., Addison Wesley**:** 245-279.

Cox, K., "Cognitive Dimensions of Use Cases - feedback from a student questionnaire." *12th Workshop of the Psychology of Programming Interest Group* (Corenza, Italy, 2000), pp

(generic ref) Evans, G., Nazzano,F., Why Are Use Cases So Painful, 1999, Columbia, SC.

(generic ref) Firesmith, D. G., Use Cases: the Pros and Cons, 2002,

Fowler, M. and K. Scott, *UML Distilled - A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Longman, Upper Saddle River, N.J., 1999.

Glinz, M., "Problems and Deficiencies of UML as a Requirements Specification Language." *Tenth International Workshop on Software Specification and Design (IWSSD'00)* 2000), pp IEEE Computer Society.

Jackson, M., *Problem frames*. Pearson Education Ltd., Addison Wesley, Harlow, 2001.

Jacobson, I., Use Cases: Yesterday, Today and Tomorrow, 2003, www-106.ibm.com/developerworks/rational/library/775.html

Jacobson, I., P. Jonsson, M. Christerson and G. Overgaard, *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison Wesley Longman, Upper Saddle River, N.J., 1992.

Jacobson, I. E., M.;Jacobson,A;, *Object Advantage - Process Re-engineering with Object Technology*. Addison Wesley, Upper Saddle River, N.J., 1995.

Krasner, G. E. and Pope S. T., "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object Oriented Programming*, **1** (3), 1988.

Kruchten, P., *The Rational Unified Process - An Introduction*. Addison Wesley Longman, Upper Saddle River, N.J., 2000.

Kulak, D. and E. Guiney, *Use Cases - Requirements in Context*. Addison Wesley Longman, Upper Saddle River, N.J., 2000.

(generic ref) Lilly, S., How to Avoid Use-Case Pitfalls, 2000,

Merrick, P., "A Requirements Pattern Based Approach to Improvements in Government I.T. Procurement", Ph.D., University of East Anglia, 2004.

Merrick, P. and P. Barrow, "Towards a Requirements Formalism in Procurement." *8th Annual Conference of United Kingdom Academy of Information Systems* (Warwick, England, 2003), pp

Merrick, P. and P. Barrow, "A Requirements Pattern Based Approach to Achieving Internal Stakeholder Agreement." *Requirements Engineering*, **submitted**, 2004a.

Merrick, P. and P. Barrow, "Testing a Requirements Pattern Language Through Reverse

Modelling." *INCOSE 2004* (Toulouse, France, 2004b), pp International Council of Software Engineers.

Merrick, P. and P. Barrow, "Testing the Predictive Ability of a Requirements Pattern Language." *Requirements Engineering*, pp (Online edition at time of writing), 2004c.

Pooley, R. and P. Stevens, *Using UML - Software Engineering with Objects and Components*. Addison Wesley Longman, Harlow, 1999.

Rosenberg, D. and K. Scott, *Use Case Driven Object Modeling with UML*. Addison Wesley Longman, 1999.

Rumbaugh, J., I. Jacobson and G. Booch, *The Unified Modeling Language - Reference Manual*. Addison Wesley, Reading, MA, 1999.

(generic ref) Smith, J., The Estimation of Effort Based on Use Cases, 1999, Cupertino, CA.

Spielman, S., *The Struts Framework - Practical Guide for Java Programmers*. Morgan Kaufmann Publishers, San Francisco, 2003.

staff, "OMG Unified Modeling Language Specification Version 1.3", 1999.

Sutcliffe, A., *The Domain Theory - Patterns of Knowledge and Software Reuse*. Lawrence Erlbaum Associates, London, 2002.

About the authors

Peter Merrick earned his PhD from the University of East Anglia in 2005 with his thesis entitled "A Requirements Pattern Based Approach to Improvements in Government I.T. Procurement". He specialises in the production of requirements specifications and has worked with various commercial organisations in the U.K. His research was sponsored by the Health and Safety Executive. Most recently he has been working with the University of Cambridge Examinations Syndicate. Peter's ambitions are to make concrete improvements in the field of unambiguous requirements representations for the purpose of 3rd party software procurement. He is the author of a number of other papers in the field of requirements engineering written with Dr. Pat Barrow, also at the University of East Anglia. Peter has his own consultancy, Active Requirements Ltd., which offers its services to industry and the public sector.

Dr. Patrick D.M. Barrow is a Lecturer in the School of Computing Sciences, University of East Anglia (UEA) Norwich, UK. He gained his B.Sc. in Business Information Systems from UEA in 1993 and his Ph.D. in 2000 (Investigating Stakeholder Evaluation in Rapid Application Development). He is currently researching stakeholder involvement in large-scale information systems development.