Analysis by Contract
or
UML with Attitude

Richard Mitchell

University of Brighton
Brighton BN2 4GJ, UK
<Richard.Mitchell@brighton.ac.uk>

Abstract

This paper summarises a tutorial entitled Analysis by Contract. Using fragments from a simple case study concerning a video store, the paper shows how a type model can provide the vocabulary needed to specify the behaviour of a video store system. Behaviour is modelled as events, such as the renting of a video by a member of the video store. It also shows how state models can be used to enrich the vocabulary of the type model.

The specifications of behaviour are in the form of contracts, with preconditions and postconditions. The "with attitude" part of the sub-title comes from the use of the Object Constraint Language to express the contracts, and to cross-reference the vocabulary of the state models to the vocabulary of the type models.

This paper is © 1999 IEEE. It will be published in the proceedings of TOOLS USA 1999. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

1. Introduction

The Unified Modeling Language—UML—provides a wide range of diagrammatic and textual notations (Booch, Jacobson and Rumbaugh 1998). It also provides a general mechanism for attaching constraints to individual elements of models. From its early days, it had placeholders for writing the kinds of constraints associated with design by contract: preconditions, postconditions and invariants (Meyer 1992). Now it has a formal notation for writing such assertions—OCL, the Object Constraint Language (Warmer and Kleppe 1998). This makes it possible to combine navigation expressions over a class diagram and constraints in OCL, in order to express properties of a model with great precision. Think of the addition of OCL to the original UML as allowing you to write 'UML with attitude.'

There are several challenges for those wishing to build precise models. One is to find and maintain a suitable level of abstraction—it is easy to be pulled downwards to the familiar world of programming languages in order to achieve precision. Another is to make sure that different kinds of modelling (static structure, behaviour specifications, state models, and so on) are cross-referenced, to ensure that they contribute to a single underlying model.

Using extracts from a small case study, this paper explores how these challenges can be met. It does not propose a method with an underlying process. Rather, it explores the technical background against which a development process could be defined. It shows how the ideas of design by contract can be applied during analysis.

The case study concerns a system to support the operation of a video store. Section 2 presents the requirements for the system. Section 3 shows a first attack on the problem of finding a level of abstraction, by showing how the problem can be divided into domains, along subject boundaries and along technical boundaries. Section 4 briefly describes the chosen approach to modelling behaviour, in which the static structure can be thought of as the vocabulary to be used to describe the behaviour. In Section 5, the approach is applied to the case study, leading to fragments of models using UML with OCL. The OCL notation is explained as it is needed. Section 6 explores how models can be connected. State models can provide extra vocabulary, and this extra vocabulary can be anchored to the static model. Then the combined vocabulary can be used to model behaviour. Section 7 briefly shows one way to remain abstract when we need to model communication between domains. Section 8 summarises the key points.

2. The case study

Here is a brief description of the operation of a video store. It is intended to provide the starting point for analysis-level modelling.

> "A video store rents and sells videos. Only members can rent videos; anyone can buy videos. The store makes a rental video available for sale when it is no longer earning sufficient rental income. To become a member, you pay a fee and receive a membership card with your membership number on it in the form of a barcode.
>
> Videos can be rented for 1 or 3 days. The daily rate for a 3-day rental is lower than for a 1-day rental. Videos returned late incur a penalty fee for each day late. This fee is set higher than the 1-day rate. Members who owe penalty fees may be forbidden to rent further videos until the fees have been paid.
>
> Members can reserve a video that is currently not available to rent. When a video becomes available, a notice is sent to the member holding the reservation, asking him or

her to collect the video within 3 days. A video not collected within 3 days is returned to the rental pool."

Before we begin modelling, we need to be reasonably clear about what we are modelling. Are we modelling the way the video store business operates now, or as we would like it to operate after we have put a computer-based system in place? Or are we modelling the computer-based system itself? To keep the example simple, we shall choose to build parts of a high-level model of a computer-based system. So, for instance, our static model will be an abstract view of the model the system is to maintain about the state of the outside world.

3. Domains

One of the principal benefits of abstraction is that it allows us to reduce the amount we need to think about at any one time. For example, we know that videos can only be rented to members. That means we must think about how people become members, how the store acquires videos to rent, and what happens when an existing member rents one of the store's videos. But we do not have to think about the details of all of these three aspects of the video store at once. We can divide the problem into a number of domains, each of which can be studied in detail separately from the other domains.

Figure 1 shows just five of the possible domains, modelled as UML packages. There are dependencies between the domains (only some of which are shown). Someone modelling the rentals aspect of the video store will know that you can only rent videos that are part of your inventory, and then only to people who are members and who make appropriate payments. But a model concerned with rentals need not be concerned with details of how people become members.

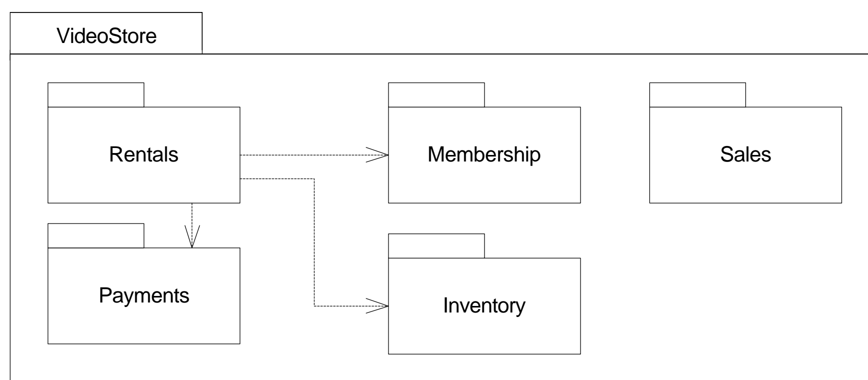

Figure 1. Five possible subject domains.

From now on, we shall call domains such as these *subject domains*, since they are concerned with different aspects of the subject matter of what we must model. There is another dimension along which we might provide different domains of modelling, which we shall call *technical domains.* Figure 2 shows a number of possible technical domains. At the heart of the system is a core domain, which might also be called the business domain. Here, we abstract away from technical issues such as user interfaces, peripheral devices and storage systems. Someone must model the components of a graphical user interface (dialog boxes, buttons, and so on). That someone might be in another company, from which we buy a GUI

library. Then, in the interaction domain, we consider how users interact with the core, using GUI components (by pressing buttons, filling text fields, reading scroll lists, and so on).

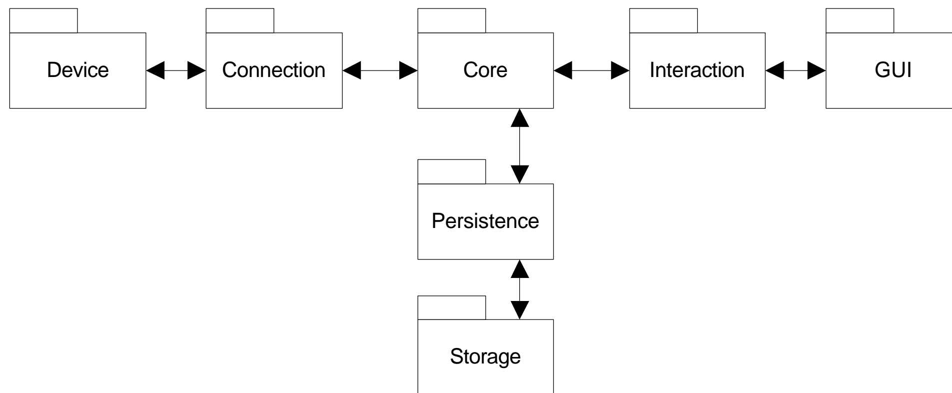

Figure 2. *Seven possible technical domains.*

Similarly, we might buy devices such as a barcode reader, complete with device drivers. Then, in the connection domain, we address the problem of mapping barcodes into core domain concepts such as members and videos. And we might choose to store objects in a relational database, which we already own. Then, in the persistence domain, we address the problem of mapping core objects to and from storage in relational tables.

The domains are not independent. But they can be treated as separate engineering problems. For example, the problem of identifying a member from a scanned barcode is a problem we can address separately from the problem of enumerating the business rules to be applied when a member wants to rent a video.

From now on, we shall concentrate on the core technical domain within the rentals subject domain. That means we shall not be concerned with barcodes, membership rules, database choices, and so on.

4. Modelling with objects and events

We can use domains to limit how much we try to model at any one moment. Now we need to decide how to model. Here is one approach. We can think of the world as made up of two kinds of things: 'things that are,' and 'things that happen.' We model 'things that are' using objects. We model 'things that happen' using events. The objects have properties (local attributes and links to other objects). The events change the properties of the objects.

Another way to think about the approach is to see the objects as giving us the vocabulary we need in order to describe what really interests us, which is behaviour. And we model behaviour as events.

Yet another way to think about the approach is in terms of an underlying model of time, such as the one described in (Jackson 1995), which is illustrated in Figure 3. This shows a timeline, with time increasing from left to right. Each vertical line represents a moment in time at which the state of the world changes. In between the vertical lines, the world remains in some state for a period of time. We model the state of the world using objects, and the changes using events. (Of course, this model of time, and hence the use of events for modelling, are not appropriate for all aspects of all systems, such as aspects of process control systems.)

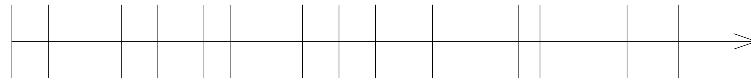

Figure 3. *A model of time.*

When we are modelling the core of a system, the events that interest us are those events happening in the world that are to affect the system. For example, suppose that we are building the system for an individual video store called LowPrice. Then we must model the core so that the event 'Kim rents copy 12 of Shakespeare In Love from LowPrice at 8.30 p.m. on August 4th, 1999, paying $4.50 in cash' is an event that changes the state of the system. But the event 'Kim borrows $5 from Heidi in order to go to LowPrice to rent Shakespeare In Love' is one that we can ignore.

We shall, of course, classify the objects and events, into types. So we shall model with object types such as *Member* and *Copy*, and event types such as *rent* and *return*.

Examples of how we model with event types and object types are coming up next, in Section 5. Section 8 discusses why we use event and object types, rather than operations and classes. But it will do no harm if you think of the object types as classes, and the event types as operations on a top-level system object.

5. A first taste of analysis by contract

In practice, we might spend some time just gathering information about our client's business and recording it informally, before attempting any modelling. For example, we could gather a list of terms used by those describing the business, and loosely classify them into 'things that are' and 'things that happen.' However, it will make the connection between object types and event types clearer if, in this paper, we build our models by a process that would not work in practice, because it records the models formally from the beginning. Strictly speaking, formal models can only be formalisations of things we understand informally.

We'll illustrate the approach by beginning to model the event type *rent*. (We'll talk about other types of event, such as *return*, *reserve* and *cancel*, but we shall not model them.) We'll begin modelling *rent* by identifying which objects we must know about for us to be able to discuss the effects of an event of this type. Whenever there is a *rent* event, we need to know in which video store it took place, who the member was, which video copy the member rented, what the time and date were, and how much money was paid. So the signature of an event of type *rent* is this:

rent(vs : VideoStore, m : Member, c : Copy, t : Timepoint, payment : Money)
    -- *Member m rents copy c from video store vs at time t, paying 'payment'*

We have identified that we need to talk about five types of objects (*VideoStore*, *Member*, *Copy*, *Timepoint* and *Money*).

Let us start work on a precondition for *rent*. Informally, we cannot rent a copy that is not part of our inventory, and we cannot rent it to someone who is not one of our members. We want to say:

pre: -- *m is one of the members of vs*
vs.members -> includes(m)
and
-- *c is one of the copies in the inventory of vs*
vs.inventory -> includes(c)

This fragment of a precondition uses OCL syntax. Lines beginning '--' are comments. The expression "vs.members -> includes(m)" says that *m* is in the set *vs.members*. The fragment introduces new terms (*members* and *inventory*). This is because it talks about relationships between types of objects. Figure 4 shows a UML class diagram that captures the object types and their relationships that we have talked about so far. In other words, Figure 4 provides the vocabulary for the signature and precondition of the event type *rent*.

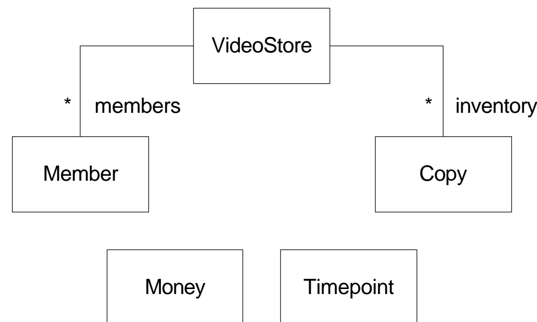

Figure 4. *An early model of object types*

The precondition is not finished. We need to say that you cannot rent a video copy that is already out on rent. The copy must be in the store. In fact, the constraint is more complicated than that. A copy could be in the store but not available to member Kim to rent because member Heidi has made a reservation. The informal requirements say that "Members can reserve a video that is currently not available to rent," but analysis has revealed that it is video *titles* that members reserve, and video *copies* that they rent. When a member holds a reservation for a title, we shall say that a copy with that title can be put *onHold* for the member. A copy that is in the store but not on hold will be *onShelf*.

These new terms we have introduced, *onHold* and *onShelf*, model different states that a copy can be in. Figure 5 shows an unfinished state model for objects of type *Copy*. The model uses some existing vocabulary: the state transitions are triggered by events of the types we listed earlier. The model also provides us with some new vocabulary: it introduces the terms *onHold*, *onShelf* and *rented*, which we can use when specifying behaviour. We now have the vocabulary to say that a member can rent a copy if the copy is on the shelf or if the copy is on hold against a reservation made by the member. In practice, the activity of constructing state models often reveals new states the objects can be in (for example, a copy can be lost).

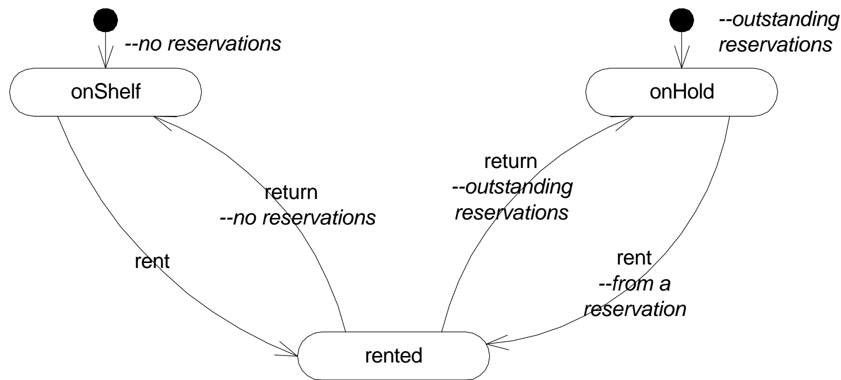

Figure 5. *An early state model for Copy.*

Standing back from the details, we see that we are modelling behaviour (by specifying event types) using a vocabulary given by types of objects and the states they can be in. To illustrate the relationships between event types, object types and object states more clearly, we are going to show some richer models resulting from more analysis, and use them to discuss cross-referencing of models.

6. Relationships amongst models

In this section, we shall see how the vocabulary of state models can be defined in terms of the type model. Then we shall see how all this vocabulary can be used to specify an event.

Figure 6 shows a richer type model, providing many more words for us to use in specifying events. (We shall not show all the state models that enrich this type model.)

In particular, the model shows us that a copy might or might not be linked to a current rental, and can be a held copy for zero or more reservations. (Reservations become fulfillable when there is a copy on hold for the member to collect. If, for example, three members hold reservations for the same title, there can be up to three copies on hold against these reservations. The model does not tie individual copies on hold to the individual reservations.) As copies pass through the states *onShelf*, *onHold* and *rented*, the values of these links must change. We can formally specify the connection between the states and the type model, as follows:

<u>c : Copy</u>

-- *A copy is on the shelf, available for rent if*
c.onShelf =
-- *it does not have a current rental*
c.currentRental -> isEmpty
-- *and it is not on hold for a reservation*
and c.~heldCopies -> isEmpty

and

-- a copy is on hold for a reservation if
c. onHold =
-- it does not have a current rental
c.currentRental -> isEmpty
-- but it is on hold for a reservation
and c.~heldCopies -> notEmpty

and

-- a copy is out on rent if
c.rented =
-- it has a current rental
c.currentRental -> notEmpty

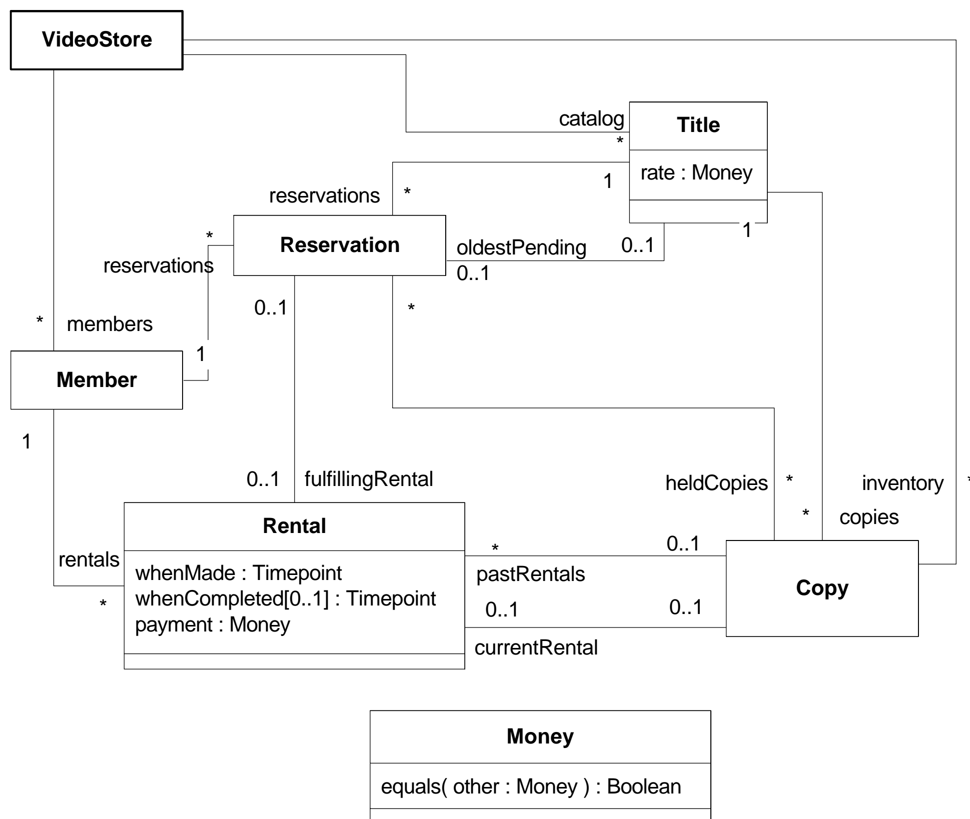

Figure 6. *A richer type model*

The left-hand sides of the three equations use terms from the state model for copies, whereas the right-hand sides use terms from the object type model. The result is that the three states of copies are now fully defined in terms of the object type model. In practice, the activity of defining states like this usually reveals that the object type model needs enriching, often with quite subtle detail, in order to be able to distinguish all states.

Now we can use the vocabulary of the object type model, augmented by the vocabulary of the state model, in order to specify an event type. Here is the specification of the *rent* event type. It uses some additional OCL operators: *select* selects a subset of elements with a

given property, *exists* is the 'there exists' quantifier, and the @ operator gives us access to earlier states (especially the state in which the *pre*condition was evaluated). The specification also uses some proposed changes to OCL (the proposals come from OCL's designers): *isNew* can be used in postconditions to test whether an object is a new object in the state after an event; *let...in* is a construct for making local definitions.

```
rent( vs : VideoStore, m : Member, c : Copy, t : Timepoint, payment : Money )
        -- Member m rents copy c from video store vs at time t, paying 'payment'
pre:
        -- The member and the copy are known to the video store
        vs.members -> includes( m ) and vs.inventory -> includes( c )
        -- and the copy is on the shelf, available for anyone to rent or
        and ( c.onShelf or
                -- is on hold ...
                (c.onHold
                -- ... for member m (i.e., m has a fulfillable reservation for c)
                and c.~heldCopies -> select(fulfillable) -> exists( v | v.member = m ) ) )
        -- and the payment is correct for this title
        and payment.equals( c.title.rate )
post:
        -- There is a new rental object
        Rental.allInstances -> includes( r : Rental | r.isNew

                -- which is a rental by member m of copy c at time t
                and r.member = m and r.~currentRental = c and r.whenMade = t
                -- for a payment of 'payment'
                and r.payment = payment
                -- and which has no 'whenCompleted' date and which is not a past rental
                and r.whenCompleted -> isEmpty and r.~pastRentals -> isEmpty

                -- and if the copy had been on hold for m against a reservation then
                and c.onHold@pre implies
                        let
                                -- m's (relevant) reservation is fulfilled
                                fulfilledReservation =
                                        c.~heldCopies -> select( v | v.member = m )
                        in
                                -- and it is r that fulfils this reservation
                                fulfilledReservation.fulfillingRental = r
                -- otherwise r has no fulfilling reservation
                and not c.onHold@pre implies r.~fulfillingReservation -> isEmpty )
```

Once again, in practice, the activity of defining behaviour this precisely usually reveals the need for additional vocabulary, uncovering subtleties that are very helpful to an analyst.

Before leaving this discussion of cross-referencing models, it is worth mentioning that OCL can be used to write precise invariants over object type models, which can be thought of as cross-referencing one part of a model against another. For example, invariants can:

- reduce the range of object snapshots that are allowed (for instance, an invariant can constrain every rental object to being a current rental or a past rental but not both)
- define redundant properties in terms of existing ones (for instance, the model in Figure 6 includes an *oldestPending* association between titles and reservations. This is useful in defining which reservation becomes fulfillable when a copy is returned. It can be defined in terms of other model properties such as timestamps on the reservations (not shown in Figure 6). If our client talks in terms of the 'oldest pending reservation' then so should we, in our models, and an invariant can define the term.)

7. On remaining abstract

This short section tries to show that the architectural model of domains is more than a starting point for analysis. It can support modular specification, and, if desired, a corresponding implementation. If we use domains carefully, they can help us remain appropriately abstract.

The original outline requirements said, "When a video becomes available, a notice is sent to the member holding the reservation, asking him or her to collect the video within 3 days." How should we model the sending of a notice to a member, when we are considering only the core of the system? There is a simple answer, but it illustrates an important modelling principle. The answer is that we need an object type called, say, Notice. In the postcondition of event types such as *return* we model "a notice is sent to a member" by asserting that a new object of type Notice is created, and linked to a reservation. And we say no more.

The modelling principle is that each technical domain is responsible for a certain kind of detail. The core domain is not concerned with details of printing and posting notices (or emailing them, or generating telephone call lists, or ...). Somewhere else, in a CustomerCommunication domain, perhaps, we can specify that every time a Notice object is created it is turned into a printed letter. (When we come to design the system, the CustomerCommunication domain could be built as a separate sub-system, and connected to the core using the kind of event notification that underpins the Observer pattern (Gamma, Helm, Johnson and Vlissides 1995).)

Separating different aspects of how notices are dealt with into different domains can make it difficult to show that a single requirement in a requirements document is actually being properly expressed in the models. Collins-Cope (1999) proposes separating requirements (usually expressed textually) from user interface issues, and goes on to show how both can be related to the core services of the system (which we are modelling as event types). The relationships between requirements, services and interface elements are many-to-many.

In Collins-Cope's terms, the words in quotes near the beginning of this section about sending a notice are a requirement. This requirement is related to several services. One such service is modelled by the *return* event type in the core domain, whose postcondition asserts that a new Notice is created. Another is the event that turns Notice objects into documents for printing. Somewhere else, we can show the design of a user interface that gives the user access to these services in an appropriate way.

8. Discussion

The preceding sections have shown how a model expressed using UML diagrams can be expressed more formally, and so more precisely, with the aid of OCL. The following

characteristics of the approach are the important ones (the words in brackets are the way we achieved the characteristics—other choices could be equally good):

- We developed an architecture for the modelling, so that we could work on different parts of the modelling separately (we divided the problem up using subject domains and technical domains, and hinted at how inter-domain connections might be captured).
- We knew when we were defining new terms and when we were using existing terms (we defined a vocabulary in the object type model, and used it to specify behaviour by writing event specifications).
- We cross-referenced models built from different points of view (object type models present a 'panoramic view' of many object types; state models 'zoom in' on a single object type. We defined the states very precisely in terms of the object type model).

The same ideas can be used at lower levels of abstraction, nearer to the eventual code that implements the system. Indeed, it is not necessary to climb to the high levels of abstraction we have been using in order to find places to add precision to models. Eiffel programmers are very used to adding assertions to their classes and methods (Meyer 1997).

The architecture for the specification might or might not be the architecture for the implemented system. For example, we could build a system in which the subject domains became subsystems within a federated architecture, each maintaining its own version of, say, the membership list. The different versions of the list would not be identical in content: different subsystems would need different properties of members. A change notification mechanism could allow the membership subsystem to bring other subsystems up-to-date with relevant changes. Alternatively, we could treat the subject domains as divisions only within the specification, and we could combine what we learn about members from several different domains before building a system around a single membership list.

There are arguments for and against modelling using separate domains. An advantage of modelling in separate domains is that it allows modellers to focus their efforts. A danger is that the structure of the models will be taken (perhaps unconsciously) to be the subsystem structure of the implementation, even though the domains might not have been chosen to be suitable parts of an implementation. By delaying the identification of parts of the overall domain, we avoid premature decisions about system structure, but make the modelling process more difficult.

Sometimes, it is possible to decide the subsystem structure of the implementation before modelling begins, so the architecture of the specification mirrors the chosen architecture of the system. Then, we can have a more seamless process, but we begin modelling at a lower level of abstraction.

We chose to model with types of objects and types of events. The choice of object types over classes is mostly a matter of terminology. When we think about what behaviour we want from a certain kind of object, we are thinking about its type properties (types are for specifying). We write code in classes to provide those properties (classes are for implementing). The choice of events over operations is also not critical, but behind it is an important modelling choice. If we think of the types of events such as *rent* as being operations on a video store system (rather than events a video store must react to) it makes almost no difference to our models (except, perhaps, to make it harder to build models before deciding what parts of the business to automate, and to lead us towards a particular design in which a video store object is a façade in front of the other objects). However, it would be different if we added operations to any of the other types. For us, there is a design step in which we decide which type of object will do what in order to provide the behaviour of the whole system. We do not want to make design decisions of this kind whilst performing analysis.

We have deliberately avoided discussion of the difficult topic of connecting the vocabulary in models to the outside world. No matter how much precision you add to UML models, there is an informal world out there that your models are related to. There is no way to formalise that world. You do not have the choice of making the people who visit a video store into formal entities. They will always be people. There is an excellent introduction to the topic of grounding models in the world in (Jackson 1995), under the heading of designations.

The ideas expressed in this tutorial paper come from many sources. They are most directly expressed in course material from InferData Corporation (1999), which, in turn, builds on earlier work, particularly Fusion and Syntropy. The Fusion method (Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes and Jeremaes 1994) was a significant step in the development of methods—it showed clearly that specification models could be freed from many design decisions. Syntropy (Cook and Daniels 1994) showed the power of using events to model behaviour, and showed how state models and type models could support each other. Catalysis (D'Souza and Wills 1998) makes the role of a type model as a vocabulary clear, and goes much further than previous methods in achieving precision in modelling at many levels of abstraction.

Acknowledgements

I thank Vladimir Bacvanski, Petter Graff, Benedict Heal and Ted Velkoff for their helpful comments on earlier drafts. I gratefully acknowledge support from InferData Corporation, and the Distributed Systems and Software Engineering Unit at Monash University.

References

Booch G, Jacobson I and Rumbaugh R (1998). *The Unified Modeling Language User Guide.* Addison Wesley.

Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F and Jeremaes P (1994). *Object-oriented development. The Fusion method*. Prentice Hall.

Collins-Cope M (1999). *The Requirements/Service/Interface (RSI) Approach to Use Case Analysis*. In Mitchell R, Wills A, Bosch J and Meyer B (editors). Proceedings of TOOLS 29. IEEE.

Cook S and Daniels J (1994). *Designing object systems. Object-oriented modelling with Syntropy*. Prentice Hall.

D'Souza D and Wills A (1998). *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley.

Gamma E, Helm R, Johnson R and Vlissides J (1994). *Design Patterns.* Addison Wesley.

InferData Corporation (1999). *Object-Oriented Analysis and Design Course Notes*. InferData Corporation, Austin, Texas.

Jackson M (1995). *Software Requirements & Specifications: a lexicon of practice, principles and prejudices.* Addison Wesley.

Meyer B (1992). *Applying Design by Contract.* IEEE Computer, 25(10), pp40-52.

Meyer B (1997). *Object-Oriented Software Construction*. Prentice Hall (2nd edition).

Warmer J and Kleppe A (1998). *The Object Constraint Language - Precise Modeling with UML.* Addison Wesley.