# 37 Things that Don't Work in Object-Oriented Modelling with UML

*Anthony J H Simons, <a.simons@dcs.shef.ac.uk>*
*Department of Computer Science, University of Sheffield,*
*Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK*

*Ian Graham, <grahami@compuserve.com>*
*Chase Manhattan Bank,*
*125 London Wall, London, EC2Y 5AJ, UK*

## Abstract

The authors offer a catalogue of problems experienced by developers, using various object modelling techniques brought into prominence by the current widespread adoption of UML standard notations. The problems encountered have different causes, including: ambiguous semantics in the modelling notations, cognitive misdirection during the development process, inadequate capture of salient system properties, lack of appropriate supporting tools and developer inexperience. Some of the problems can be addressed by increased guidance on the consistent interpretation of diagrams and the most helpful sequencing of techniques. Others require a revision of UML and its supporting tools. Most problems can be traced to the awkward transition between analysis and design, where UML's eclectic philosophy (the same notation for everything) comes unstuck. Modelling techniques that were appropriate for informal elicitation are being used to document hard design decisions; the same UML models are subject to different interpretations in analysis and design; developers are encouraged to follow analytical procedures which do not translate straightforwardly into clean designs. UML is itself neutral with respect to good or bad designs; but the consequences of allowing UML to *drive* the development process are: inadequate object conceptualisation, poor control structures and poorly-coupled subsystems.

## 1. Introduction

The following catalogue of problems arose out of discussions that the authors had at the UK Object Technology '98 conference in April. They are drawn from the experience of real developers working on real projects, in academia and in industry. Initially, we had expected to compare different design notations for their clarity and discuss the semantics of UML. Later, we found that it was not just that UML contained semantic ambiguities and inconsistencies, but rather that the increased prominence given to particular modelling notations had in turn placed a premium on carrying out certain kinds of analysis and design activity. Our analysts were enthusiastically adopting new approaches to conceptualising their system, eventually becoming trapped in unproductive arguments over the objects populating the system and the proper representation of the control structure of the system. Our designers were then refusing to implement the models produced by the analysts, since it was often impossible to map from use case models and sequence diagrams onto anything that a conventional software engineer would recognise. We decided then that the problem of *cognitive misdirection* was at least as important an issue as *semantic inconsistency*.

### 1.1 The context of our critique

An earlier draft of this article was made available to interested colleagues in academia and industry; and we also received feedback from the ECOOP referees. Many replied in support of our observations, especially the industrial contacts, who found our practical experiences in *using* UML more valuable than abstract debates on model semantics. From the comments received, it was also clear that some readers were viewing our remarks through an imagined filter, supposing that we had a prior position on UML, which we do not have. In particular, we would like at the outset to address the following questions:

- "Aren't the authors assuming that UML is only useful for design (analysis) when it is also intended to be used for analysis (design)?" No. Being aware of the eclectic ambition of UML to model both analysis and design, many of our criticisms impact at the point where analytical techniques are being pressed into service in documenting designs, hence the impression that we concentrate on the design perspective.

- "Aren't the authors unfairly assuming that UML is a process, when in fact it only claims to be a notation, plus a semantics?" No. At the time of writing, the *Rational Objectory Process* has yet to be published. However, the existence of UML sets the agenda: it puts a premium on developing certain models, which in turn promotes the use of certain conceptual modelling techniques over others. *Some* linking process will inevitably form in the minds of developers; and we focus on places where this misdirects.

We want to make clear that we are not against UML *per se*: indeed, we appreciate the benefits that a standard modelling notation may bring, allowing developers to communicate in an unambiguous way. However, the UML standard *as currently defined* [Rational, 1997] is open to some rather subtle differences in interpretation, leading to serious problems downstream, as we have discovered with our developers in practice. To this extent, UML just happens to be the modelling approach that is under the spotlight.

### 1.2 The critical framework

The body of this article is basically an enumeration of the problems experienced by our developers as they embraced the UML notations and engaged in what they considered was the most appropriate sequence of activities for building UML models. For convenience, we have discussed each problem under the heading of the most relevant UML model; the order of these major headings is not intended to be significant, but simply follows the flow of argument. We assume that the reader is already familiar with UML notations and, for reasons of space, we do not develop example diagrams where the problem description is adequate in itself. Mostly, each problem speaks for itself and needs no further comment. Each problem is classified by type; this is shown by a three-letter code, such as *ADQ* or *MIS* in the subheading. The classifications were decided retrospectively, after collection of as many problem-cases as time allowed. A key to these is given below:

> INC - inconsistency, meaning that parts of UML models are in contradiction with other parts, or with commonly accepted definitions of terms in object technology;

AMB - ambiguity, meaning that some UML models are under-specified, allowing developers to interpret them in more than one way;

ADQ - adequacy, meaning that some important analysis and design concepts could not be captured using UML notations and CASE tools;

MIS - misleading, meaning that the natural development path promoted by a desire to build UML models actually misdirects the developer.

Each problem cited below was placed into one of these categories, representing the major perceived underlying cause of the fault. The categories are not intended to be mutually exclusive, nor necessarily exhaustive, but merely indicative. In our conclusions, we discuss the significance of the numbers of problems in each category.

## 2. Use-Cases and Use-Case Models

*Use cases* were originally intended, in Jacobson's *Objectory* and *OOSE* methods [Jacobson, et al., 1992] as a means of eliciting user requirements. Each case was an *instance* of some user-interaction, described as a whole. The development of use cases was seen as an investigative, rather than documentary approach, whereby the developer collected together instances of user interactions until the set of cases seemed complete enough for proper design structuring to be imposed. The two developments to the basis case were probably intended merely as a means to avoid excessive duplication in the analyst's notes. In their adoption by UML [Rational, 1997], use cases have acquired a new *type*-perspective, which has also established the «uses» and «extends» relationships more formally as a kind of novel control structure. Our developers have experienced multiple conflicts of interpretation with use cases, as a result of this change.

### 2.1 Use-cases have two conflicting interpretations (AMB)

There is a conflict between the original "for instance" and the more recent "prototypical" interpretations given to use-cases. In the former view (more often useful during analysis), the developer is eliciting sample user-interactions. In the latter view (more often useful during design), the developer is attempting to specify the typical course of all interactions of this type. The analysis-centred view belongs with single-threaded non-branching sequence diagrams (cf Jacobson's original *Object Interaction Diagrams* [Jacobson, et al., 1992]), whereas the design-centred view has produced a need to incorporate multi-threaded, looping and branching syntax in sequence diagrams. The former interpretation happens to sit more easily with some of the proposed solutions to further problems of interpretation, which we discuss below. In any case, one or other interpretation should be taken, consistently.

### 2.2 Developers don't understand what «extends» means (INC)

The «extends» relationship between use-cases is misunderstood because developers compare this with the *extends* relationship in Java, where it has the meaning of *inherits*. In the latter sense, developers expect the extended use-case to be all of the original case, plus the extra information. Instead, an extension case is typically an alternative or exceptional track that

replaces part of the original. This is not so much a problem of the *inadequate* specification of the semantics of «extends», as the unhelpful *overloading* of the term with conflicting semantics of (genuine) *extension* in Java and *selection* in UML. We can illustrate the conflict by considering the semantics implicit in the choice of use case names: extension cases are named after the alternative history, rather than after the combination of both histories, which would be more consistent with inheritance. Compare: *class OrderedSet extends Set*, with: *use case AbortTransaction extends CreditPurchase*. *OrderedSet* includes the notion of a *Set*, whereas *AbortTransaction* does not include the notion of a *CreditPurchase*, instead it is part of an aborted credit purchase. To be consistent with inheritance, an extension use-case should stand for the whole adapted case, *viz*: *AbortedCreditPurchase extends CreditPurchase*, which then only works if the "for-instance" view of use-cases is adopted (see 2.1 above). Alternatively, if the "prototypical" view of use-cases is adopted, inheritance is the wrong mechanism for modelling alternative tracks (see 2.5, 2.7 below).

### 2.3 Developers don't understand what «uses» means (INC)

The «uses» relationship between use-cases is misunderstood because developers compare this with the *uses* relationship between classes, *eg* in the Booch method [Booch, 1994], where it has the meaning of a simple client-server dependency. In the latter sense, developers expect a «uses» relationship to work in the same way as a subroutine call: they think that the using case invokes the used case as a whole, in an encapsulated way. However, «uses» is defined as a stereotype of a generalisation relationship [Rational, 1997]; in other words, it is a kind of inheritance between use-cases, not a kind of composition. Jacobson's original definition of «uses» [Jacobson, et al. 1992] supports this inheritance-like view: the "used" cases are known as "abstract" use-cases; and these are obtained by factoring out procedural elements that are common to several "concrete" use-cases, in the same way that you would create an abstract base class. Furthermore, the "using case" may use procedural elements of the "used case" out of order (see Figure 1); where two or more cases are so used, elements of each may even be interleaved by the using case. It is clear that the factored-out "abstract" use-cases are not components to be used as a whole, but inherited base cases, whose (therefore unencapsulated) procedural elements are actually ordered by the using cases. To avoid this confusion between use-case inheritance and composition in the minds of developers, the «uses» relationship should perhaps be redefined in the sense of straightforward procedural decomposition, for which no object-oriented modelling notation currently exists. There is merely the suggestion, in the UML semantics document [Rational, 1997] that "component" use cases may exist.

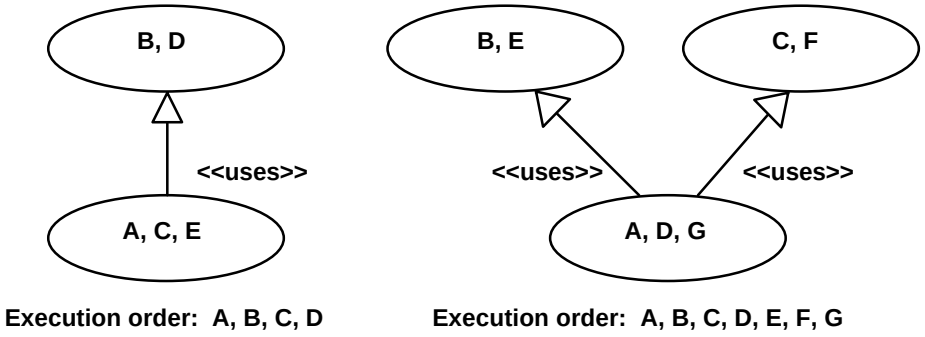


**Figure 1: Execution order in «uses» relationships**

### 2.4 Return of control from an «extends» case is under-determined (AMB)

An «extends» use case has three possible interpretations as a formal control structure: either, it is a single-branch conditional; or it is an exception; or it is an alternative track. CASE-tools typically have to take one or other interpretation, leading to inconsistencies in the way they generate code. In the first interpretation, the «extends» case is treated as the "then"-branch of a single-branch "if" statement - if the condition or guard is satisfied, the extension is invoked and afterwards control returns to the point of call. The main use-case continues uninterrupted thereafter. In the second interpretation, the «extends» case is treated as a set of exception-handling procedures, and the extension is deemed to terminate abnormally. Control is not therefore transferred back to the main use-case. In the third interpretation, the «extends» case is treated as an alternative track to the main use-case. The extension may continue forever as an alternative history, corresponding to an alternative timeline on a sequence diagram; or, it may at some point re-enter the main use-case. This point has to be specified explicitly and corresponds to an arbitrary *goto* intstruction. Extension cases are probably not one single concept, but several concepts conflated into one.

The most straightforward reading of the UML semantics document [Rational, 1997] gives the first interpretation above (the extension is "inserted into" the normal case). However, we have seen many examples where this interpretation cannot be both intended and correct. Consider the use case (originally due to Cockburn) for purchasing stock goods, shown in Figure 2. Instead of signing for the order, the customer may choose to pay direct by credit card. *PayDirect* is modelled as an extension to the *SignForOrder* case. Interpreted as a single-branch if, this means that an invoice will still be sent and have to be paid; which is clearly not intended. Some of the subsequent parts are intended (the delivery of goods) but not others (sending and payment of the invoice), requiring a peculiar re-entrant flow of control.

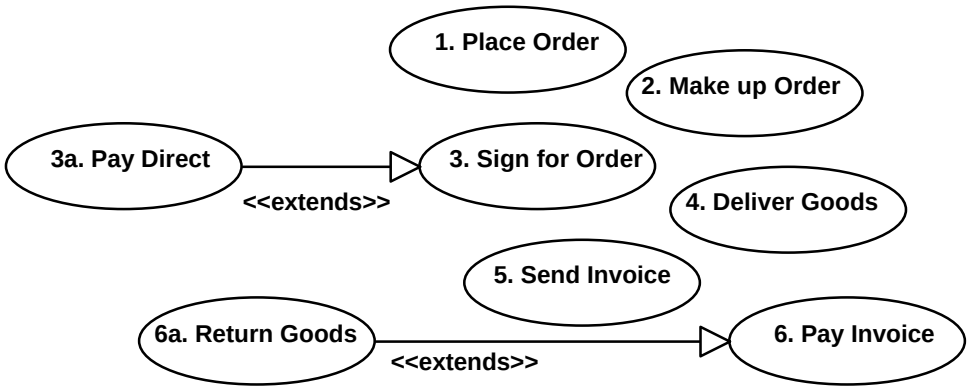


**Figure 2: Control flow problems in «extends» relationships**

### 2.5 The «uses» and «extends» conceptual model leads to poor control structure (MIS)

The conceptual modelling fostered by use-cases seems initially attractive, but forces developers down paths which lead inevitably to badly-structured code. Instead of proper block-structured designs with sequences, selections, and iterations, the use-case modelling approach leads to code which is strongly interdependent and interleaved. In the case of an «extends»

relationship, the point of dispatch from the main case must be specified in the extension case; and there are three possible ways in which the return of control may be handled (see 2.4 above), which must all be specified in the extension-cases. Likewise, with the «uses» relationship it is the "using" case which must specify to what part of the "used" case to dispatch, how to order the procedural elements of the "used" case and whether to interleave them with any of its own elements, or elements of other "used" cases (see 2.3 above). There are no longer single points of dispatch and return; in fact, many developers have commented on how similar this is to writing arbitrary *goto* instructions. The apparent simplicity of use-case modelling, based on the notion of adaptation through generalisation relationships, therefore hides a huge complexity of control-flow. Our developers concluded that «uses» and «extends» were *faux amis* during analysis, leading them down unhelpful paths which later proved difficult, if not impossible, to implement. Returning to Figure 2, consider that a customer may later *ReturnGoods* as an extension to the *PayInvoice* case. This works locally as an alternative track, but fails to make sense if the customer has already paid direct by credit card (he would then have to take additional steps to recover his money; this was not documented in the example). Even simple examples like this exhibit unpleasant interactions between different extensions to the normal use case.

### 2.6 The «uses» and «extends» generalisations are mutually inconsistent (INC)

If both «uses» and «extends» are to be understood as stereotypes of generalisation, which is how they are currently defined in the UML metamodel [Rational, 1997], then they are mutually inconsistent kinds of generalisation. The "using" case at the initiating end of a «uses» relationship stands for a complete, whole use-case which incorporates the "used" case, in the usual manner of a child class that is an extension of its parent class. This is consistent with the idea that the "using" case inherits the elements of the "used" case (see 2.3 above). Against this, the "extending" case at the initiating end of an «extends» relationship stands only for the alternative track, not the combination of the normal case plus the alternative track (see 2.2 above). Consider: *CreditPurchase uses GiveReceipt*, compared with: *AbortTransaction extends CreditPurchase*. A consistent view can only be taken if the recommendations made above for harmonising the «extends» relationship with normal notions of generalisation are accepted.

### 2.7 The «uses» and «extends» generalisations are inappropriate (ADQ)

Alternatively, the branching logic currently encoded with «extends» should not be defined as a stereotype of generalisation, but as some kind of selection, or alternative composition relationship. Similarly, «uses» may perhaps be better defined as some kind of sequential composition relationship (see 2.3 above). According to our developers, generalisation is best applied to "prototypical" cases which genuinely extend (add monotonically to) the behaviour of existing cases. Such cases exist, but they are far outweighed by the number of cases where simple composition of behaviours is required, with selection, sequence or iteration. Generalisation is the wrong kind of relationship for modelling alternative tracks; and also the wrong kind of relationship for expressing subroutine calls. Control-flow abstractions are much better if they are encapsulated with single points of call and return, like the old fashioned block-structured approach to control with sequences, selections and iterations.

# 3. Sequence Diagrams

Sequence diagrams are UML's adoption of Jacobson's *object interaction diagrams* [Jacobson, et al., 1992]. Originally, these were introduced during the design ("construction") phase of OOSE, after object models had been built. A single sequence diagram was drawn for each use-case, showing the objects affected by each clause of the case. Testers have remarked on how useful this is for ensuring that every system requirement is handled by some object [McGregor, 1997]. As a result, sequence modelling has typically been introduced as an earlier development activity, after use case modelling. Our developers have suffered multiple problems of interpretation here, both as a result of the type/instance conflicts transferred forward from use case modelling, and also as a result of the inadequate conceptualisation of objects at this early stage. This is one of the occasions where an implicit process forms in the mind of the developers - they follow the order of presentation of UML models in the literature [Fowler, 1997], despite Fowler's explicit injunction not to do this. In fact, it would be much more appropriate to concentrate on declarative specifications of each use case, than to proceed to concrete sequence diagrams.

## 3.1 Sequence diagrams have two conflicting interpretations (AMB)

There is a conflict between the original "for instance" and the more recent "procedural" interpretations given to sequence diagrams. In the former view (more often useful during analysis), the developer is describing the sequence of interactions for a single thread and for a single branch through the many possible alternative paths in the system. In the latter view (more often useful during design), the developer is attempting to specify the typical course of all interactions of this type. The analysis-centred view sits well with single-threaded non-branching sequence diagrams (cf Jacobson's original *object interaction diagrams*), whereas the design-centred view requires multi-threaded, looping and branching syntax in sequence diagrams, as introduced in UML 1.1 [Rational, 1997], in order to account for the various kinds of branching logic necessary in procedural descriptions. Our developers have suffered occasions of misunderstanding because of the implicit switch between these two viewpoints when moving from analysis to design. They find reasons to criticise both applications of sequence diagrams, as we explain below, but feel that they suit better the "for instance" analysis viewpoint.

## 3.2 Sequence diagrams are considered too early by developers (MIS)

Our developers have suffered repeatedly from an inability to translate directly from use-case descriptions into properly-constituted sequence diagrams, that is, where each stimulus arrow is genuinely a request from one object to another, invoking a method. We think this is because the kinds of object-concept available at this early stage of development relate almost exclusively to human actors and to passive datastore concepts, such as letters, forms, price records and stock records. There is a temptation to model the nouns in the description of the use-case as the objects in the sequence diagram, in a naïve one-to-one way. Sentences like: *the warehouse manager looks up the price and the stock-level* result in messages between the external actor (the *WarehouseManager*) and object concepts representing a *PriceRecord* and a *StockLevelRecord*, because this is how the client perceives the system (or, because this is how

the legacy system operates). Instead, price and stock-level should most likely be access methods of a *GoodsItem* object. Likewise, we have seen sentences like: *the account executive sends the rejection letter to the customer* result in arrows between an *AccountExecutive* actor and a *RejectionLetter* object, which in turn has an arrow to a *Customer* object, intended to illustrate how the letter somehow forwards itself there. Here, the developers are clearly being fooled into drawing something more like a workflow, or dataflow diagram, instead of a properly constituted object-stimulus diagram. We have found that most existing descriptions of business processes (such as ISO 9000 business procedure forms) are phrased in this workflow-like manner and lead to all kinds of confusion with developers. To combat this, we have forbidden our developers to draw sequence diagrams until *after* all the object concepts have been established using other techniques, such as CRC-card modelling [Beck and Cunningham, 1989; Wirfs-Brock et al., 1990], which is much more productive and has the right behavioural focus for generating object abstractions.

### 3.3 The "main track" plus "exceptional tracks" model is a fiction (ADQ)

Sequence diagrams work from the premise that you can draw a "main track" for the normal course and then supplement this with "alternative tracks" for alternative courses of events. This is a reasonable model for the handling of exceptions, but not particularly good for handling ordinary decision logic. Our developers have great problems in identifying what should constitute the "main track". This is because many business procedures are in fact multibranching and each branch has equal importance or weight. One of our example systems under analysis is a credit reinsurance application, developed for a subsidiary of BTR [Hung, et al., 1998]. In this application, there are four ways a *Credit Limit Application* can succeed and four ways in which it can fail. The business logic is structured such that early rejection or acceptance cases are spun off the "continue to investigate creditworthiness" cases, which in turn may lead to acceptance or rejection. It is impossible to decide which of these should be considered the "normal course", since acceptance and rejection occur with equal likelihood. Given the constraints of use-case semantics and the modelling notation (see 3.4 below), the only feasible approach is to choose the "continue to investigate" case as the "normal course" and spin off early accept and reject cases; however, this strikes both our developers and the client as highly artificial. From the analysis point of view, it emphasises one large use-case (the "continue to investigate" case), which in all probability is a fairly *unlikely* pathway through the system; the client tends to reject this as an inappropriate model of the way he views his business. From the design point of view, it is poor because it produces one large "continue to investigate" use-case and lots of calls out to the small "accept" and "reject" cases; the designers consider this to be poor system modularisation. There are alternative ways of breaking down the business logic into sequence diagrams, but these all tend to violate one or other principle of the modelling notation. We discuss some of these below.

### 3.4 Use-case granularity conflicts with logical units needed in sequence diagrams (INC)

A use-case is defined as some sequence of interactions that produces an observable benefit to the participating actor [Jacobson, et al., 1992]. The "observable benefit" constraint is intended to ensure that use-cases have a sufficiently large granularity to be worth representing. However, we find our developers wanting to truncate use-cases at points where conditions and

branching are introduced, because of the weak support given to branching and iteration in sequence diagrams (UML 1.0 had no support; UML 1.1 has some support in the form of "alternative timelines" for objects - see also 3.5, 3.8 below). Using the example of the *Credit Limit Application* introduced in 3.3 above, a different way of representing the cases is to write a single case up to the first early rejection point; then to spin off an alternative case if the application is not rejected, up to the point where it succeeds early. If the application does not succeed at this point, a new case can be spun off to represent its further investigation until a second rejection point is reached; and so on. From the designers' point of view, this is a habitable alternative, because each sequence diagram supports a single branch of the logic and is of a comparable modular size. However, the granularity principle for use-cases is now broken, since there are many cases spun-off which do not correspond to a single, complete interaction of a user with the system. The cases also acquire unlikely-sounding names in this approach - consider: *Early Credit Limit Acceptance that was Previously Not Rejected Early*.

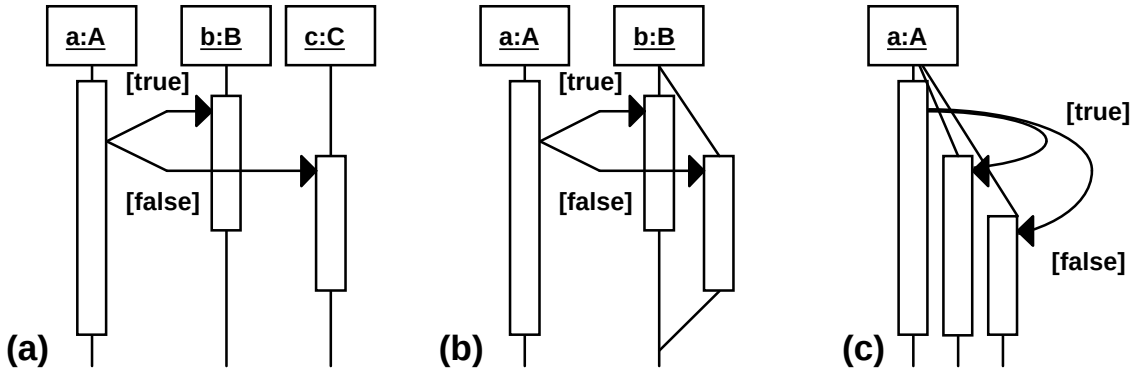


**Figure 3: Simple and pathological sequence diagram branching**

### 3.5 Branching timelines in sequence diagrams can only handle simple cases (ADQ)

UML 1.1 introduced a branching logic whereby the timeline of an object can split at a decision-point. This is shown by a forking of the timeline into two parallel timelines, which may or may not be subsequently merged again later at a symmetrical join point. This increases the power of the "procedural" interpretation of sequence diagrams and removes the need to develop many small alternative sequence diagrams for each case. Examples on the Rational website [Rational, 1997] illustrate a simple case, shown in Figure 3(a), in which alternative messages are sent to different server objects; and a moderate case, shown in Figure 3(b), where alternative messages are sent to the same server, which enters a different timeline for each. These cases are both simple, because all timelines are merged from the viewpoint of the client, after the different branches have terminated. Our developers have found a much harder and fairly common case, shown in Figure 3(c), in which the client object enters an alternative timeline as a result of a test. Typically, each such timeline represents a mutually-exclusive path through the business decision logic; and therefore, all subsequent processing after this branch-point is conditional on being in that particular branch. It is unlikely that timelines merge after this point, or if they do, then it is in a manner much more easily illustrated using a flowchart than using a sequence diagram. This is because the business logic tends to produce many parallel timelines representing alternative ways of reaching the same outcome, but which are contingent on having passed through different histories. To join these timelines usually

requires crossing over the paths of other variant timelines, making the sequence diagram very cluttered.

In the *Credit Limit Application* example (see 3.3, 3.4), one modelling approach resulted in *eight* parallel timelines running for an *AccountExecutive* object. In a flowchart, the parallel histories are implicit and can be induced by following different paths through the flowchart; it is also much easier to show the merging of outcomes after different decision paths have been traversed. Our developers concluded that sequence diagrams are not nearly as good a vehicle for representing business logic.

### 3.6 Exemplar use-cases produce too many sequence diagrams (ADQ)

Taking the opposite "for instance" interpretation of use-cases, most usually adopted during analysis, removes the need to show branching, but forces the development of too many exemplar sequence diagrams, each one corresponding to a single path through the business logic. Our developers were initially more comfortable with this interpretation of sequence diagrams, since it did not require use of the inadequate branching logic, described in 3.5 above. For anything other than trivial business processes, they found themselves again coming up against the use-case granularity principle (see 3.4 above) in which a use-case was not complete unless it delivered an observable benefit to the participating actor. In order to ensure the completeness of each case in the face of multibranching business logic, the developers found themselves writing many sequence diagrams which had a large common part, representing the shared decision paths taken up to the final branch-point in the business logic. The amount of effort required in generating the duplicated information was judged very unsatisfactory, despite the fact that some developers were cutting and pasting from previous examples using CASE tools. Many developers wanted to split use-cases at the decision points, preferring modular units that break at sequence, selection and iteration points, but this also breaks the granularity principle for use-cases. Our designers recommended that we discard the granularity principle for use cases.

### 3.7 Developers put non-logical threads of control into sequence diagrams (MIS)

The kinds of problem described so far above were elicited by our more experienced developers, who understood the "exemplar" (analysis) versus "prototypical" (design) interpretation of sequence diagrams and then understood the logical reasons for initiating and terminating alternative timelines in objects. It was far more often observed, among the less experienced developers, that sequence diagrams were used in ways that emphasized object interactions at the expense of proper decision logic. For example, one of the models developed for the *Credit Limit Application* example (see 3.3, 3.4, 3.5 above) showed a single timeline in which a rejection point was reached, which should logically have resulted in the termination of the case; however, because of the CASE tool's inability to support branching and alternative timelines, the developer had continued using the very same timeline for the "continue to investigate" case, which was logically inconsistent. We have since emphasised to our developers that a timeline must conclude where a logical path terminates. CASE tool problems aside (see 3.8 below), we consider that sequence diagrams are problematic where they suppress the developer's perceptions of proper decision logic.

### 3.8 The CASE tools do not support alternative timelines (ADQ)

Some of our developers considered that they were forced into emphasizing object interactions at the expense of proper decision logic, because of the inability of the CASE tools to support alternative timelines. We experimented with *Rational Rose '98*, which still only supports single object timelines, and found two possible solutions to the branching problem. In one approach, it is possible to illustrate two alternative branches in a simple sequence-diagram by placing mutually exclusive guards (conditional expressions) over the arrows which initiate the alternative processing streams. In this approach, shown in Figure 4(a), the alternative timelines are described sequentially, one after the other, but it is logically impossible to resume or continue with a merged timeline past the endpoint of each branch. The timeline must terminate at the end of a decision path; the only thing which may follow at the end of a completed decision path is the initiation of a different (alternative) decision path.

Those of our developers who favoured the "for instance" (analysis) viewpoint raised objections to this solution. They did not like the fact that alternative tracks (which logically would execute during the same time-period) were placed sequentially and wanted to have them run concurrently. The second solution we developed was to allow multiple copies of the same essential object to appear in a single sequence diagram, shown in Figure 4(b). Each copy of the object represented an alternative timeline for that object. If ever an alternative timeline merges with a master timeline, this is shown using a (dotted) return arrow, which has the sense of a synchronisation or rendezvous (see 3.11 below). Some of our developers did not like this duplication of objects; however one conceded advantage was that the different messages sent to the many copies of an object were nonetheless automatically collected together as methods in a single class, in the corresponding class diagram.

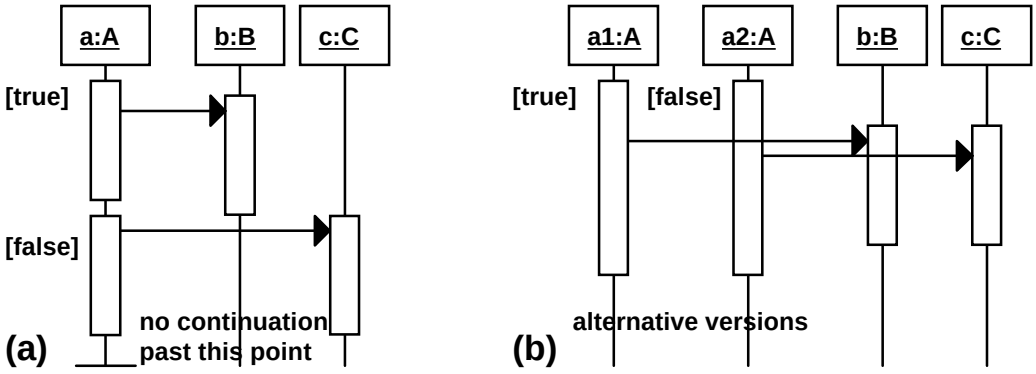


**Figure 4: Simulating timelines in CASE tools without them**

### 3.9 Sequence diagram alternative timelines are easily misread (MIS)

One of the associated problems with branching was that timelines could be misread. In the first of the two approaches suggested above, in which two alternative branches are enumerated sequentially, developers found it easy enough to visualise the alternative branches in the timeline which contained the mutually exclusive guards (see 3.8 above), but they were less clear when following the history of other objects in the same diagram. In sequence diagrams which describe a "for instance" (analysis) viewpoint, it is normally possible to follow the thread

of processing in any single object by reading down its timeline; but this is not possible in the "prototypical" (design) viewpoint. Developers transferring from analysis to design tend, accidentally, to merge the alternative processing streams carried out in object timelines that are at several removes from the object timeline in which the original branching took place. We emphasise that this is a problem of interpretation relating to the different semantics of the same UML model when it is used for analysis and design. It arises from taking a technique originally developed for elicitation purposes and then pressing it into service in a different design context. We advised appropriate training to overcome this misperception.

### 3.10 Return arrows are indistinguishable from call-back arrows (AMB)

UML 1.0 did not distinguish between the "return arrow" and the standard "invocation arrow", making it possible to confuse a call-back invocation with the termination of a method. This problem was discussed by Fowler in his book [Fowler, 1997] and was subsequently addressed in UML 1.1. Here, a "return value" is indicated using a dotted line for the shaft of the arrow, to contrast with the solid line used for a "call-back" invocation. Nonetheless, most published UML textbooks and all the available CASE tools do not yet support this refinement, making it impossible to distinguish the two in practice. With proper use of focus bars (see below), it would be possible to dispense with "return arrows" altogether in synchronous computation, since an answer is always implicit in the original message-send. However, we have found it extremely hard to dissuade our developers from using the return arrow in this sense, particularly if they have a dataflow-based mindset. This is a conceptual pitfall acting against the interests of object modelling and allowing developers to produce dataflow diagrams.

### 3.11 Return arrows are indistinguishable from rendezvous arrows (AMB)

A different problem, which has not been reported before, is the fact that it is still impossible in UML 1.1 to distinguish a *rendezvous* from a simple *synchronous* return. A rendezvous occurs where a process has previously forked, using the "half arrowhead" asynchronous message notation to indicate that the remote focus bar is deemed to execute in parallel with the local focus bar. At some future point, the spun-off concurrent process returns a value to its caller and the caller must synchronize with it, in order to receive the return value. We consider that this rendezvous case is important to illustrate, since otherwise the diagram-reader could infer that the spun-off process continues indefinitely without returning to its caller. We use the dotted-shaft arrow to indicate a rendezvous, showing the point at which the spun-off process must synchronize with its caller. This, however, creates confusion with the usage of this arrow in the sense of a simple return value. Eventually, we had to recommend to our developers *NOT* to use the dotted arrow in the sense of a simple return value, since this may be indicated implicitly in the sending of messages (see 3.10 above) and the nesting of focus bars (see 3.12 below), whereas a rendezvous must always be indicated explicitly.

### 3.12 Focus bars have several conflicting interpretations (AMB)

Our developers have complained repeatedly about the meaning of the *focus bars* in sequence diagrams. A focus bar is a thin rectangle that is placed onto an object timeline during the period in which that object is active. However, the meaning of "being active" is undefined and

ranges from: "executing a concurrent thread" to "activating a stack frame". In the available examples from Rational [Rational, 1997], there are cases where the focus bar is used to indicate, stylised here in Figure 5(a) a concurrently-executing thread in each of several distributed objects; 5(b) the collapsing of several stack-frames corresponding (probably) to the time an object is activated, that is, receives its first message up until it yields its last return value; and 5(c) a stack-frame which is created anew for each method invocation. The meaning of (a) could be implicitly distinguished from the other two, provided that all message-sends were indicated using the asynchronous arrowhead and all rendezvous indicated using dotted-shaft arrows. We think that the usage in Figure 5(b) arises from developers not being careful enough about the stack-frame semantics of Figure 5(c), which would require overlapping focus bars *both* for self-delegation *and* call-back invocation, as illustrated (see 3.13 below). The usage (b) is "sloppy" and often occurs when sequence diagrams are used during analysis, to indicate that an object is somehow "busy" without wanting to draw attention to the number of levels of invocation it has suffered. When such models are transferred into design, it becomes impossible to put any sensible interpretation on the use of the focus bar.

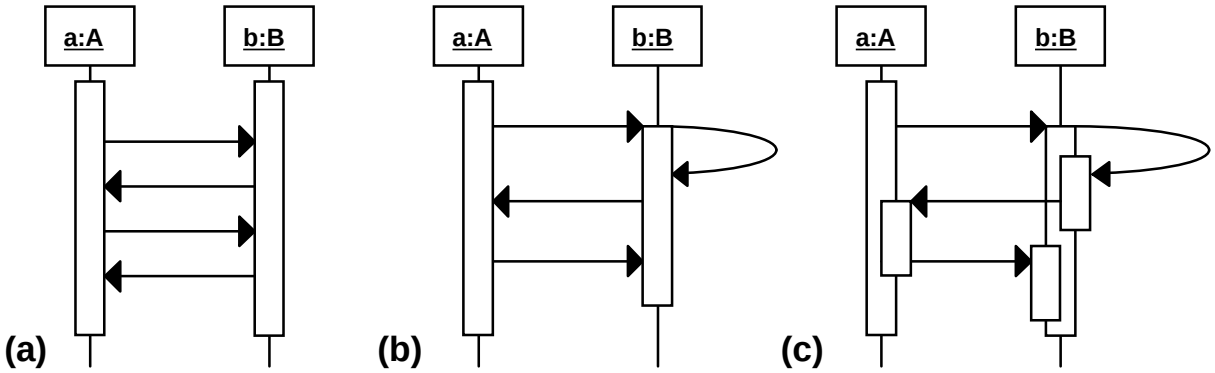


**Figure 5: Thread, activation and stack-frame semantics of focus bars**

### 3.13 Focus bars are used inconsistently in sequence diagrams (INC)

In Jacobson's original intention [Jacobson et al., 1992], focus bars were used to demonstrate graphically whether a centralised (comb-like) or decentralised (stair-like) control structure had been designed into the model. In this context, the most likely interpretation of a focus bar is a *stack frame*. In UML 1.0, focus bars are optional and may be switched on or off by the CASE tools. In this case, they are considered redundant, when in fact they do bear important information about method termination. Without focus bars, it becomes imperative to use return-arrows to indicate the termination of processing in one object and the passing of control back to the caller (but see 3.10, 3.11 above for problems with return arrows). With focus bars, the simple return-value arrow is no longer necessary, since the stacking of invoked methods is made clear visually from the lengths of the focus bars. In UML 1.1, this idea was taken further, using overlapping focus bars to denote *self-delegation*, or the fact that an object may send a message to itself to carry out some sub-process as part of the current process it is executing. Unfortunately, this principle of overlaid focus bars was not extended (as it should have been) to call-back messaging. Where an object sends a message to an already-activated object, then this should be shown by an overlapping focus bar, as shown in Figure 5(c) above.

Against this, UML 1.1 does not mandate such overlapping bars for nested invocations for any case other than self-delegation, which is an inconsistency.

# 4. Collaboration Diagrams

In UML, both *sequence diagrams* and *collaboration diagrams* are considered to be kinds of *interaction diagram*, showing alternative views of the same sequence of messages sent between a group of objects. This reinterprets the notion of a collaboration in a way that is inconsistent with the original literature on collaborators [Beck and Cunningham, 1989; Wirfs-Brock and Wilkerson, 1989; Wirfs-Brock, et al., 1990]. In particular, UML has no model that exactly corresponds to the *collaboration graph* from Responsibility-Driven Design (RDD), a design-level concept with a strong client-server flavour. As a result, UML does not, and cannot currently support the important subsystem identification and design-level optimisation stages of methods influenced by RDD, such as OPEN [Firesmith, et al., 1997; Henderson-Sellers, et al., 1998]. We consider that this weakness is due to UML's heritage from OMT [Rumbaugh, et al., 1991]. It is strong on modelling abstract relationships between concepts in the analysis domain, where these relationships may have many different semantics. It is weak on isolating the important client-server dependencies, motivated by the real communication requirements of the system, which bear strongly on the modular structure of system-level design. Our criticisms in this section focus on the inadequacy of UML for system-level design.

## 4.1 The term "collaboration" has lost its proper distinctiveness (AMB)

In the original CRC approach [Beck and Cunningham, 1989] and the *Responsibility-Driven Design* method [Wirfs-Brock, et al., 1990], the term "collaborator" had a particular meaning as the server-class on which some client-class depended, for the fulfilment of its responsibilities. A "responsibility" was a terse statement of purpose, corresponding potentially to several methods. A "collaboration" was a static client-server relationship, representing a functional dependency, and supporting potentially many message interactions with the collaborator. Unfortunately, in UML the term "collaboration" is not differentiated sufficiently from "interaction", meaning a single messaging event between a client and a server instance. A distinct notion of "collaboration" is a prerequisite for reasoning about functional dependency and system coupling. A "collaboration" is (should be) a type-level concept, whereas an "interaction" is an instance-level concept.

## 4.2 A collaboration diagram is not an interaction diagram (INC)

A *collaboration graph* in RDD [Wirfs-Brock, et al., 1990] was originally a static directed graph of client-server class couplings, where an arrow represents a functional dependency of one class upon another. The collaboration graph was a faithful measure of system coupling, being derived from the inter-class references needed to support actual messaging between their instances. The collaboration graph was an essential tool for investigating system coupling: mutually-linked classes or classes linked in rings could be easily identified. The system could then be transformed to "minimize contracts" (i.e. minimize the number of distinct client-server interfaces, in [Wirfs-Brock, et al., 1990]) and thereby reduce "real" system coupling. Against this, the so-called "collaboration diagram" of UML is merely another name for a kind of object

*interaction diagram*, illustrating actual message sequences between instances, rather than static relationships between classes. In its canonical form, the UML "collaboration diagram" is a portion of an object diagram (instantiating part of a class diagram) with undirected links (instantiating static associations) on which are superimposed arrows representing the sending of messages. A single UML "collaboration diagram" typically follows the consequences of one external message-send, related to a single use-case, and so concerns only some of the objects in the system.

### 4.3 An interaction diagram cannot be coerced into a collaboration diagram (ADQ)

The UML semantics document [Rational, 1997] claims to allow an RDD-style collaboration diagram as an additional option. However, this is supposed to be obtained by removing message arrows and replacing instances by classes in a canonical UML "collaboration diagram". This guideline is inadequate. Such a transformation would typically result only in a partial model of the system and not give a true measure of coupling; more importantly, without the message arrows, it would be impossible to determine the direction of client-server dependency between the classes, since UML associations are undirected; furthermore they correspond to semantic relations *other than* functional dependency (see 4.4 below). We consider it a failure of UML to have misunderstood the nature and purpose of a collaboration diagram. To rectify this, we should have to say that an RDD-style class collaboration diagram is constructed by considering *all* object interaction diagrams for a system, discarding links which do not support messaging, but converting all others into a single directed arc for unidirectional messaging and two reciprocal arcs for bidirectional messaging.

### 4.4 A class diagram cannot be coerced into a collaboration diagram (ADQ)

Some of our developers asked why they could not use a UML *class diagram* to represent the same inter-class dependency captured in an RDD-style collaboration graph. There are significant differences between the two kinds of diagram: in particular, the connections from class to collaborator in RDD are derived from a full knowledge of real communication between their instances, and correspond to a *functional dependency* of the client class upon the server, rather than some imagined association between entities in the analysis domain, as in UML. Furthermore, all client-server connections are *directed* in RDD; mutual coupling must be indicated using two arrows. This is quite different from UML associations, which can be interpreted either in an *undirected*, or *bidirectional* way, depending on whether role-names are attached at the association ends. Associations represent abstract connections in the analysis domain, rather than real class references in the design domain (see 6.2, 6.3, 6.5 below). The RDD collaboration graph corresponds most closely to a class diagram containing *only* uni-directional associations, each having *only* the semantics of a client-server functional dependency. To achieve this, other kinds of association and dependency (*data dependency*, *existence dependency*) would have to be detected and removed from a UML class diagram. Furthermore, there is no guarantee that a class diagram brought forward from the analysis phase would contain all the eventual client-server collaborations needed to support real messaging. Basically, a (genuine) collaboration diagram can only be reverse-engineered from full messaging information and cannot be forward-engineered from a class diagram.

### 4.5 A duplicate interaction model is not required (ADQ)

Our developers have occasionally questioned the purpose of having two kinds of interaction diagram in UML (the "sequence diagram" and the "collaboration diagram"). These two models are nearly equivalent, as demonstrated by the Rose '98 tool, which can transform from one to the other. The "collaboration diagram" may contain additional information about inter-object links, but otherwise it is a cosmetic transformation of the "sequence diagram". Our developers were familiar with the idea of building complementary system models (the so-called "orthogonal views" of a system) and found that UML was counter-intuitive by providing two versions of exactly the same model.

### 4.6 A genuine collaboration diagram is still required (ADQ)

Instead, it would be more useful to have a genuine collaboration diagram. We have explored the use of the RDD collaboration graph in [Simons and Snoeck, 1998], finding it a necessary tool to identify situations where system-level design transformations may be applied to reduce cross-coupling and interdependency. Two main kinds of design transformation are possible. Faced with a closed ring of collaborators, a superordinate entity is introduced, corresponding to a *Mediator* pattern [Gamma, et al., 1995], which aggregates over the members of the ring. These therefore no longer need to maintain collaborations with each other. Instead, the collaborations supported individually by the members of the ring migrate to the aggregate entity, which mediates any residual communication between the members. The second transformation involves identifying groups of clients which collaborate with the same server. If the server interface used by each client overlaps to any degree, then a generalized entity is introduced as the parent of the clients. The individual collaborations between each client and the server are merged as a single collaboration between the parent and the server. *Template Method* and *Command* patterns are typically generated by this process. This kind of system-level design modelling, present in RDD and *Discovery* [Simons, 1998] is absent from most object-oriented methods, and cannot yet be supported clearly by UML. This could be fixed if UML were to adopt the kind of RDD collaboration diagram outlined above, in 4.2, 4.3, 4.4. This could be syntactically similar to a class diagram, but with directed arrow connections representing client-server relationships.

## 5. State and Activity Diagrams

UML's *state diagram* derives from the *dynamic model* of OMT [Rumbaugh, et al., 1991] and ultimately from Harel's *statecharts*. It is an extension of a finite state machine in which each state at one level of description may be expanded to a substate machine at a finer level of granularity. Entering a superstate corresponds to entering a substate machine; arcs leading from the superstate indicate exit transitions common to all substates. The standard event-driven model of computation is also augmented by extra program conditions restricting arc-traversal.

Statecharts are a very good formalism for representing a system which has control states. Such a system has various functions which are enabled and disabled during its lifetime. This kind of system contrasts with a standard hierarchical menu-driven system, where any system

operation is equally likely to be selected at any time. UML's *activity diagram* is a variant of a *state diagram* in which the states are "action states", meaning some mode of system operation in which a particular computation is enabled and executed. The term "action state" contrasts with "data state", such as "stack full", and may even be compared with the "action" box of a traditional flowchart. An activity diagram is otherwise like a finite state machine with program conditions on its arcs, rather than events. Concurrency may be represented using Petri-net style transition-bars to indicate concurrent dispatch and synchronisation points.

### 5.1 Statecharts were originally for system-level control modelling (MIS)

Most object-oriented methods seem to assume that state diagrams are only useful for modelling the life history of single objects. This smaller-scale modelling is useful for specifying class designs and later testing the behaviour of individual objects. However, the over-emphasis on object states has tended to obscure the original intention behind statecharts, which was to model whole systems in terms of their states, or modes of operation. Most systems have either a hierarchical control structure (representable using *eg* JSP structure charts), or a mode-based control structure (representable using a Harel statechart).

### 5.2 Activity diagrams are considered too late by developers (MIS)

The lack of proper control-flow semantics in use-case and sequence diagrams (see above) is quite adequately addressed in the UML activity diagram [Rational, 1997]. However, this diagram has not received the attention it deserves, and is often relegated to a late stage in the presentation of object modelling techniques. It is typically ignored by developers, despite Fowler's recommendation to consider it earlier [Fowler, 1997]. We have found it useful to analyse system tasks in the early stages using the logical equivalent of activity diagrams (see the *precedes* and *invokes* relationships between *tasks* in the OML *task model*, [Firesmith et al., 1997]), to order the tasks and show how some are consequent on others. However, we have found it hard to persuade our developers to go direct from use case models to activity diagrams in UML. Instead, they have been misled by the prominence given to other modelling approaches in the published literature to proceed directly to sequence diagrams, which we showed above were introduced too early. Activity diagrams are typically considered too late, on the other hand.

### 5.3 Activity diagrams are not supported by CASE tools (ADQ)

Unfortunately, Rose '98 and other CASE tools do not yet support activity diagrams. This means that there is no mechanism to compensate for the poor control flow mechanisms of use-cases and sequence diagrams in most tools, which support UML 1.0 only.

### 5.4 Use-cases compete with state diagrams for control modelling (MIS)

We have experienced, among our developers, occurrences of the "use case too far" syndrome, in which the developer tries to program too much of the system logic into his use-cases, where it would be more appropriate to leave this refinement to the later design stage. An *In-Flight Shopping* system was developed with many exceptional use-cases for each time the "abort"

button was pressed. This was a considerable expenditure of effort, which could have been saved if the developer had not attempted to capture so much detail initially in the use cases, but instead had produced a state machine for each shopping transaction. In each such transaction, there were only two kinds of "abort" state, one in which no further action had to be taken and one in which the transaction had to be "undone". We have since advised our developers not to proceed to the finest level of detail in use case descriptions. Instead, they are advised to follow the broad lines of each business transaction, ignoring the ways in which these may be interrupted or aborted until later on in the development cycle, when these concerns are addressed in state models.

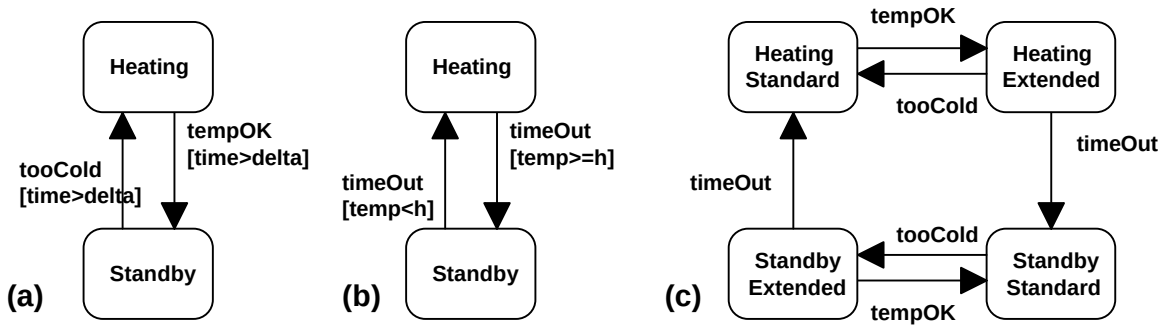


**Figure 6: Exposing events and states concealed by conditional transitions**

### 5.5 Conditional transitions conceal a duplication of control states (MIS)

State diagrams have been extended with *conditional transitions* which, if used uncritically, completely undermine the purpose of the state machine formalism. The assumption behind any state machine model is that the states and arcs somehow capture all the salient aspects of the flow of control in a system. What happens next is determined uniquely by the current *state* and the next *event* (input, message, etc.) that arrives from outside. However, once you introduce an additional *condition* onto an arc, this spoils the simple assumption above. The states no longer capture all the salient aspects of control. To see why, consider the three equivalent heating systems in Figure 6. State changes in 6(a) are governed by temperature events, with extra timing conditions. In 6(b), state changes are governed by timing events, with extra temperature conditions (showing how you can fool yourself about the real events in a system). In 6(c) the conditions are revealed for what they are: extra hidden control states. Because of this possible self-deception using extra conditions, we have trained our developers to explore the full state machines obtained by expanding out conditional transitions, so that they are at least aware of the number of real states in the system.

### 5.6 History icons conceal a multiplication of state machines (MIS)

The normal interpretation of a substate machine is that traversal always begins from an identified default starting state. Harel allows the placement of a "history icon" to signify that a superstate remembers the last occupied substate, when the superstate is exited. Upon reentry to the superstate, the substate machine resumes where it last left off. This enrichment was adopted in OMT [Rumbaugh, et al., 1991] and Booch [Booch, 1994], but not yet explicitly in UML. We hope that UML declines this enrichment, since it corresponds to a repeated

duplication of the entire super state machine, once for each alternative remembered substate. (Unfortunately, we have just heard that the draft UML 1.2 aims to incorporte history markers). We have shown our developers how state machines with history icons correspond to multiplied state machines, as a means of discouraging them from using the history icon.

# 6. Class Diagrams

Class diagrams are perhaps the best-known and best-understood part of UML, since the "class boxes" in the diagrams translate in a straightforward way into program classes. There is little doubt as to what the notation means. For this reason, developers feel they are on firm ground when confronted with a class diagram. Paradoxically, this can lead to overconfidence after the analysis stage and inadequate object modelling in the design stage. The real strength of UML's class diagram is its ability to capture a wide variety of semantic associations, which are the perceived relationships between entities in the analysis domain. UML is more like OMT [Rumbaugh, et al., 1991] in its treatment of associations as abstract connections, and less like Booch [Booch, 1994], which is clearer about the structural *has* and functional *uses* relationships, where the latter are based on real client-server dependency. The associations in UML may correspond to anticipated, but as yet uninterpreted connections, to data dependency and multiplicity relationships, as used in traditional entity-relationship modelling, to existence dependency relationships (the *derived* association of OMT, which indicated an existence-dependent property or entity), and also eventually to client-server dependencies. This is a very rich representation which frankly confuses developers. It is not so much that the semantics of the associations are poorly specified (they are among the most clearly specified parts of UML), but that any one of these perspectives taken individually would drive the design stage in a different direction, based on whether the number of data files, inter-module communication, or object creation order were being optimised.

## 6.1 Class diagrams tend to fix object abstractions too early (MIS)

Class diagrams are typically developed to model the analysis domain, before alternative object concepts have been explored. We have found that our developers tend to propose the most obvious domain nouns as the object abstractions in the application, whereas a more considered approach might experiment with different abstractions, testing them for their functional salience in the context of the application. The objects proposed by our developers correspond typically to passive datastore concepts; this is not surprising, considering the influence of entity-relationship modelling on OMT [Rumbaugh, et al., 1991] and so on UML. It is not that concepts such as *Invoice* and *Payment* are wrong, but rather that once these have been identified, then our developers tend to want to ascribe all remaining methods to these objects. The gestalt of the analysis class diagram inhibits more productive thinking about more active, "agent"-like abstractions. We have seen example class diagrams for a *Lending Library* application in which *borrow* is attached variously as a method of *Borrower* (accepting a *Book* as its argument), or as a method of *Book* (accepting a *Borrower*), when in fact it should be the constructor for a *Loan* owned and initiated by a *Lender* abstraction, which has control over (is responsible for) the resources lent out. A responsibility-driven perspective [Wirfs-Brock, et al., 1990] would capture this idea at the outset. Again, in the *Credit Limit Application* example (see 3.3, 3.4, 3.5), we have seen a majority of methods assigned to the

*AccountExecutive* object, which has god-like status, compared to the passive *RiskAssessment* and *CountryAssessment* forms. This is inadequate object modelling: instead, the behaviour of the *AccountExecutive* needs to be distributed over a society of manager-like abstractions, each of which handles part of the real executive's job. The *AccountExecutive* itself could become an instance of the *Command* pattern [Gamma, et al., 1995]. Eventually, the form-like abstractions should be replaced by searching-agents which take on some of the executive's tasks and record the results of searches in their state.

### 6.2 Associations in analysis obscure real connections in design (AMB)

Associations between classes are undirected, typically labelled with the rôle-name played by each entity at the ends of the association. The rôles are usually symmetrical and correspond to perceived relationships in the analysis domain, and do not always translate into necessary connections in the design domain. Associations correspond to many different kinds of semantic dependency, such as *data dependency*, with its multiplicity and optionality constraints; *existence dependency*, with its connascent creation and deletion semantics; and *functional dependency*, with its client/server invocation semantics. As a result, our developers become confused over which associations to implement as inter-class references in the design. In particular, we have seen examples of class diagrams in which perceived associations in the analysis domain did not eventually correspond to any useful client-server connection in the design domain, and the opposite case, functional dependency relationships implied by the needs of message-sending which were not captured anywhere as static associations in class designs. Our developers report that they find too many possible associations in their analysis; they would prefer some more constrained approach to modelling. This problem concerns identifying the real client-server connections from the total set of associations, which could be addressed by a proper semantic characterisation of collaborations, as discussed in section 4. A different problem concerns the divergent pressures on design (see 6.4 below).

### 6.3 Automatic translation of associations can lead to poor design coupling (MIS)

Recently, various articles have appeared discussing the possibility of automatic translation from UML analysis diagrams into designs which implement these models. While this is a clever idea, we wish to point out that this abrogates the responsibility of the designer to ensure good coupling characteristics in his system. We have seen examples of undirected associations which our developers have implemented as bidirectional class references, where the connection is only ever traversed in one direction. Associations with 1..* multiplicity are used universally to justify set-valued attributes and mutual interdependency relationships between classes, corresponding to something like the *Observer* pattern, where a more judicious analysis might suggest the appropriate application of the *Mediator* pattern [Gamma, et al., 1995]. In a *Real Estate Agent* example, the *Property* being sold was mutually connected both to the *Purchaser* and to the *Vendor* as a result of direct translation from analysis. The developer was unhappy about the fact that every house in his database had to have attributes for a buyer and seller. He was advised to be less literal in his trust of analysis diagrams. Instead, a *SaleContract* mediator aggregates over *Purchaser*, *Vendor* and *Property*; and these do not have any connections to each other.

### 6.4 A class diagram and a data model are different things (AMB)

Because UML captures many different kinds of semantic relationship in the same analysis class diagram, competing divergent pressures start to apply in the design stage. If we assume that the function of a *class* is eventually to define some kind of collaborating agent within a system, then the function of a *class diagram* should be more like the kind of RDD-style collaboration graph described in section 4 above. We encourage our developers to take this *client-server* perspective only, if they wish to produce class diagrams in which directed associations translate into real references used in the design. This is quite different from the *data dependency* perspective, which is appropriate when optimising the independence of data files for a database. Then, we are not concerned about client/server relationships, but only M:N data dependency. In the system design stage of any method, one strives to reduce inter-module coupling, essentially minimising the number of connections between parts of the system. Entity-relationship modelling is the appropriate design technique for minimising associations. This technique is intended to remove the direct connections between entities which can be reached indirectly through others (generating the so-called *access path*). The result (up to 3NF) is a set of files which are optimised for independent access and modification, such that changes are picked up throughout the system, no matter what the initial point of access. This is quite different from the design pressures operating when minimising the number of distinct client-server interfaces between collaborating objects (the "minimising contracts" phase of RDD [Wirfs-Brock, et al., 1990]). To handle the needs of inter-object communication, connections are necessary which violate 3NF from the data modelling perspective. Instead, *Mediator*-pattern aggregations are created in different locations than the linker-entities mandated by ERM. These implications of the difference between functional and data dependency are not really recognised in UML; but are recognised in OML [Firesmith, et al., 1997], which asserts that data modelling and class modelling are different activities, therefore class diagrams should be less influenced by data modelling techniques.

### 6.5 Fine semantic labelling of associations may be a waste of time (MIS)

Many notations, including UML [Rational, 1997] and OML [Firesmith, et al., 1997] make a great deal out of the semantic labelling of different kinds of association. We are naturally in favour of improved semantic modelling in general; but have come to doubt whether all the effort initially invested in analysis diagrams is eventually worth it. Our developers agonise over whether a particular association is a *containment*, or an *aggregation*, or some other kind of *using* or *dependency* relationship. *Aggregations* in UML are then either *exclusive* (black diamond) or *shared* (white diamond); the notion of *dependency* may correspond either to *existence*- or *functional* dependency. Other kinds of semantic characterisation are possible, such as whether a generalisation relationship forms an *exclusive/overlapping* or *exhaustive/ extensible* set of partitions. How many of these characterisations eventually impact on the design? So far, we have only found that the 1:M *multiplicity/optionality* constraint in analysis (which maps onto decisions about set-valued objects) and the combination of *exclusive* and *connascent* ownership (which maps onto decisions about object embedding and object database clustering) have any significant impact in later design. There may be others.

We have found that our developers invest too much effort in the initial semantic characterisation of the analysis domain, thinking that this will help in later design. In fact,

many of these characterisations are irrelevant later. Consider the *Mediator*-transformation (see 4.6, 6.3), which is applied to reduce inter-object coupling. It matters little whether the *Mediator* is a true composite aggregation: a *Rectangle* with its *Point origin* and *Integer sides*; a shared aggregation: a *WordProcessor* and a *SpreadSheet* which both share a *SpellChecker* and a *GrammarChecker*; or simply some kind of coordinating abstraction, such as a *SaleContract* with its *Purchaser, Vendor,* and *Property*. What matters eventually is that the system design is layered properly and unnecessary cross-connections between objects are removed. The only relevant concern in design is whether there eventually has to be a reference from one class to another. Notice again how a single design transformation, which introduces new object abstractions, may completely invalidate or remove the sets of connections between objects on which the fine semantic labels were placed. On the other hand, there are cases where the kinds of semantic labelling advocated in analysis fail to capture common properties of designs. The combination of *exclusive* (having only one owner) and *connascent* (born and die at the same time) ownership means that the life of one object wholly depends on another, so the dependent object may be embedded (by value) in the master object in C++. This is irrespective of whether the association was originally marked as a *composite aggregation,* or simply a *dependent* association with *exclusive* semantics.

Although we recognise this is a potentially contentious point of view, we feel that a lot of the effort spent in labelling initial analysis diagrams is a waste of time, distracting developers from the more important, but too often neglected, creative and inventive object modelling aspects. Investing effort in early labelling acts as a disincentive to making radical transformations to the structure of systems, something which is usually badly needed. In the Discovery method [Simons, 1998], semantic labelling of client-server connections is not performed until *after* the system design has stabilised; and fewer binary properties are needed to characterise the space of possible implementations. In other respects, the focus of effort on labelling and distinguishing different kinds of *aggregation* and *containment* to identify these apart from normal associations is probably misplaced. As we noted above (see section 6, introduction), more important semantic characterisations of associations in the class diagram are whether these correspond to: *data, functional* or *existence* dependency relationships.

## Conclusions

We emphasise that some of the problems described are not unique to UML - for example, the class diagrams of many methods allow similar ambiguous shifts of interpretation between the data-dependency and client-server views; and Harel statecharts are adopted by several other methods. However, we used UML as the exemplar model notation of the moment; and a good many problems had their origins in inconsistency and ambiguity in UML. We are aware of several other surveys of UML in preparation that are due to report on the semantic inconsistencies and ambiguities present in the notation. It is now widely appreciated that the so-called "UML semantics document" is not in fact a formal semantics, rather it is a meta-syntax description. The meta-model is, if you will, the BNF for well-formed UML models, but does not interpret these. We expect that forthcoming surveys will address the formal semantics issues properly. Our survey here is different, in that we have considered the way developers embrace and use UML, irrespective of how well or badly defined it is. The cognitive issues surrounding the focus of developers' attention, how this is engaged and

directed, are at least as important as the static issues of model semantics. By problem category, we identified the following counts (totalling 37):

| | | |
|---|---|---|
| INC | (inconsistency) | 6 counts |
| AMB | (ambiguity) | 9 counts |
| ADQ | (adequacy) | 10 counts |
| MIS | (misdirection) | 12 counts |

When analysing these counts, we decided to let the AMB and MIS scores reflect equally on the inexperience of the developers and on the unhelpfulness of the notation. Considering first the problems of ambiguity (AMB), it would be possible to train developers to interpret models in a particular way, but then again, the models should be better defined and should not permit developers to get the wrong idea. The high level of misdirections (MIS) encountered in our survey was quite worrying. One of the ECOOP reviewers kept raising the issue that surely all of the problems with UML could be fixed by insisting on a better declarative semantics for the analysis models; and that good design is surely a matter of arbitrary choice in the way in which the semantics of the analysis models are implemented. We disagree on the basis of the MIS scores, which indicate the number of times developers were "led up the garden path" by particular analytical approaches. It should be obvious from Gestalt psychology how important initial conceptualisations are; and how much they influence subsequent concept formation. If the initial view of a system is a semi-normalised data model, and if communications between entities are conceived initially as workflows (dataflow by any other name), then substantial effort has to be invested to undo the effects of these first perceptions. This is not simply an issue of selecting one design approach from among many, it involves going against the tide, fighting against the conceptual clutter in the minds of developers.

The INC scores reflected on problems with the *consistency* of UML, whereas the ADQ scores reflected on problems with the *completeness* of UML. These counts suggested that further revisions to UML were necessary. In particular, the authors identified many problems with the control flow logic in use cases and sequence diagrams; and the absence of a proper model to illustrate client-server coupling for system design optimisation. The interested reader is referred to alternative treatments of these topics in the methods SOMA [Graham, 1995], *Discovery* [Simons, 1998] and OPEN [Firesmith, et al, 1997; Henderson-Sellers, et al., 1998]. Here, task analysis replaces use cases; and the relationship between tasks corresponds to clear sequence, selection and iteration compositions. Class diagrams correspond to proper client-server graphs, allowing the system design stage to proceed smoothly.

We have frequently highlighted the problems in using UML for design, not because we assume it is only useful for design, but because the conceptual activities and techniques associated with building UML models are not as well adapted for design as they were for analysis. Use cases, sequence diagrams and association-based class diagrams were originally intended as informal techniques to help elicit information in the analysis domain. These same models are now pressed into service as design artefacts, documenting control structures and data structures which, frankly, fail to make sense. Paradoxically, the successive revisions to UML have emphasised more and more the design-level usage of these same models, with increased support for direct code generation from UML case tools, such as Rose '98. This can only lead

to further problems, as developers discover how their initial analysis concepts do not transfer well to designs that generate quality code.

## References


K Beck and W Cunningham (1989), "A laboratory for teaching object-oriented thinking", *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. Sigplan Notices, 24(10)*, 1-6.

G Booch (1994), *Object-Oriented Analysis and Design with Applications, 2nd edn.*, Benjamin-Cummings.

D Firesmith, B Henderson-Sellers and I Graham (1997), *OPEN Modelling Language (OML) Reference Manual*, March, SIGS Books.

M Fowler (1997), *UML Distilled*, Addison Wesley.

E Gamma, R Helm, R Johnson and J Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

B Henderson-Sellers, A J H Simons and H Younessi (1998), *The OPEN Toolbox of Techniques*, Addison-Wesley.

I Graham (1995), *Migrating to Object Technology*, Addison-Wesley.

K S Y Hung, A J H Simons and A Rose (1998), "Can you have it all? Managing the time and budget against quality issue in a dynamic business object architecture development", *Proc. 6th Conf. Software Quality Management* (Amsterdam: MEP), 21-34.

I Jacobson, M Christerson, P Jonsson and G Övergaard (1992), *Object-Oriented Software Engineering: a Use-Case Driven Approach*, Addison-Wesley.

J D McGregor (1997), *Testing Object-Oriented Components, ECOOP '97 Tutorial 2* (Jyväskylä, AITO/ACM).

Rational Software (1997), *UML 1.1 Reference Manual*, September; also available through: http://www.rational.com/uml/ .

J Rumbaugh, M Blaha, W Premerlani, F Eddy and W Lorensen (1991), *Object-Oriented Modeling and Design*, Prentice-Hall.

A J H Simons (1998), *Object Discovery - a Process for Developing Medium-Sized Applications, ECOOP '98 Tutorial 14*, (Brussels, AITO/ACM), 90pp.

A J H Simons and M Snoeck (1998), "Rigorous object-oriented system design", *Proc. 2nd. Conf. Rigorous Object-Oriented Methods*, Bradford.

R Wirfs-Brock and L Wiener (1989), "Responsibility-driven design: a responsibility-driven approach", *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. Sigplan Notices, 24(10)*, 71-76.

R Wirfs-Brock, B Wilkerson and L Wiener (1990), *Designing Object-Oriented Software*, Prentice Hall.