Chapter 17

30 THINGS THAT GO WRONG IN OBJECT MODELLING WITH UML 1.3

Anthony J H Simons
University of Sheffield, UK
a.simons@dcs.shef.ac.uk

Ian Graham
Ian Graham Associates, UK
grahami@compuserve.com

Abstract The authors offer a catalogue of problems experienced by developers, using various object modelling techniques brought into prominence by the widespread adoption of UML standard notations. The catalogue is revised to reflect changes made between UML versions 1.1 and 1.3, in which a number of semantic inconsistencies in the notation were fixed. Notwithstanding this, developers still seem to create inordinate problems for themselves by pursuing unproductive development strategies that are apparently fostered by UML. This article shows how the biggest problem by far is *cognitive misdirection*, or the apparent ease with which the rush to build UML models may distract the developer from important perspectives on a system. This problem is more serious than the outstanding *inconsistencies* and *ambiguities* which still exist in UML 1.3. A number of *inadequacies* are also highlighted, where UML somehow still fails to express what we believe are important semantic issues. While UML itself is mostly neutral with respect to good or bad designs, the consequences of allowing UML to *drive* the development process include: inadequate object conceptualisation, poor control structures and poorly-coupled subsystems.

1. INTRODUCTION

The following catalogue of problems is a revision of an earlier survey of developer experiences using the OMG endorsed standard for object modelling, UML. In our original findings [SG98a,b] some 37 different difficulties associated with using UML 1.1 were reported. With the recent introduction of UML 1.3 [BRJ99] a number of semantic inconsistencies in the notation were fixed. However, a significant number of problems, both cognitive and semantic, still remain.

1.1 The context of this critique

The difficulties described below are drawn from the reports of developers managed by, or known to the authors, as they used UML on real projects, in academia and in industry. In analysing these experiences, the authors 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. 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. 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* or *ambiguity*.

The authors appreciate the benefits that a standard modelling notation such as UML is supposed to bring, by allowing developers to communicate in a common language. However, the UML standard *as currently defined* [BRJ99; R97] is open to some rather subtle differences in interpretation, leading to serious problems downstream, as we have discovered with our developers in practice.

1.2 The critical framework

The body of this article is an enumeration of the problems experienced by developers as they embraced the UML notations and engaged in what they considered to be the most appropriate sequence of activities for building UML models. Each problem is indexed, for ease of reference, and classified using a three-letter code, in the style: *(#2: MIS)*. The classification codes have the following interpretation:

- INC - inconsistency, meaning that parts of UML models are in contradiction with other parts, or with commonly accepted definitions of terms;
- 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;
- MIS - cognitive misdirection, meaning that the natural development path promoted by a desire to build UML models actually misleads 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 the following section, we discuss what we believe are some of the causes underlying the failures reported by developers in our survey. The subsequent sections form the body of our catalogue. In our conclusions, we discuss the significance of the numbers of problems in each category.

2. CRACKS IN THE HEART OF UML

UML is intended to be a general-purpose modelling notation, based on a few consistently applied principles. As a result of pressures brought to bear during standardisation, UML has accomodated heterogeneous elements from its precursor methods and notations. It is a difficult task to tread the tightrope between minimalism and expressiveness successfully; and UML does this moderately well by managing to apply the same conventions across a number of its models. However, these strengths of UML also eventually account for its weaknesses.

2.1 Universal notation has multiple interpretations

One of the claimed strengths of UML is that it is *universal*, that is, the same notation is used for analysis, design and documenting the implementation. This minimalism is typically cited as a benefit. However, the down-side is that one developer will interpret another developer's diagram under a different set of assumptions. What was intended as an analysis diagram in one context may be interpreted as a concrete design in another context. Our developers repeatedly ran into problems because they could not decide how concrete UML models were supposed to be. Should an association just represent a vague imagined connection in the problem domain, or should it mean a physical connection between classes in the solution domain? Diagrams representing one thing were often found to have been interpreted as the other. UML claims that its capacity to model both perspectives is an advantage; the fact that it offers no control over how diagrams are interpreted must then be considered a serious disadvantage.

2.2 Universal notation fosters naïve seamless development

Nowhere is this more obvious than when a class diagram, which is drawn as an initial analysis of relationships between concepts in the problem domain, is pressed into service as a concrete design. Associations are converted into pointers; many-to-many relationships are translated directly into set-valued attributes and strong, mutually-coupled *Observer* patterns appear in the design [GHJV95]. It would have been better if our developers had noticed that many-to-many relationships could be eliminated using *linker entities* (in Entity-Relationship Modelling) or *Mediator* patterns (in Responsibility-Driven Design) to reduce coupling between classes in the design [SSH98]. Yet, they repeatedly failed to perceive systems in terms of such co-ordinating structures (see also problem #28 below).

The practice of analysing the semantic relationships in the problem domain using classes and associations actively blinds developers to alternative practical structures. Gestalt theory predicts that the initial concepts formed during the perception of some phenomenon radically affect how subsequent constructs are formed. Following this, we have observed how early analysis modelling using classes tends to fix these concepts and suppress others. Development then becomes a steady elaboration of detail; the analysis classes and structures do not change, but merely add to their attributes and methods, in the style of some early naïve development methods [CY91a,b; WN95], which emphasised the *seamless transition* from analysis into design. This approach leads to systems which are overly coupled and exhibit poor modular structure [SSH98]. Well-structured systems typically do not exibit a 1:1 mapping from analysis to design objects.

2.3 Eclectic models fail to resolve competing design forces

Another of the claimed strengths of UML is that it is *eclectic*, that is, it keeps the best parts from a number of precursor methods and notations, in particular [B94; RBPEL91; JCJO92]. Eclecticism is regarded as a good thing, because this means that the notation contains something for everyone; no-one's point of view is left out. However, it is extremely difficult to ensure that all the parts work together all of the time. Like the old adage that a camel is an animal that was designed by committee, eclectic systems are often ugly because they borrow elements out of the original context in which they evolved. In UML, great care has been taken to ensure that the elements of the notation fit together, at least on the syntactic level.

However, if you examine the intent behind sets of model elements in a single UML diagram, you will often find that these are in conflict, because they were taken from contexts which originally supported mutually exclusive design approaches. For example, a class diagram blends associative relationships between data structures with client-server relationships equivalent to inter-module procedure calls. If one were to resolve the data model alone in terms of minimising data dependency, this would lead to one structure, equivalent to 3NF. If one were to resolve the client-server class coupling model (the sense in which RDD originally used the term *collaboration graph* [WWW90]) then this would lead to a completely different structure, minimising inter-module dependency [SSH98]. The down-side of an eclectic notation is that different design forces compete within a single UML model - rather than offering a single perspective, it offers multiple perspectives simultaneously. As a consequence, developers do not profit from the structure offered by either perspective; they are unable to proceed because the UML model fails to resolve the competing design forces.

This duality is not just a feature of the class diagram. The sequence diagram switches between a dataflow and a method invocation perspective. The state and activity diagrams switch between a finite state machine and a flowchart perspective. Models which were highly constrained in their original context lose their useful constraint when extra enhancements are added in UML: an example of this is the attaching of extra *guard conditions* to events on transitions in the state model, which undermines the purpose of the state machine perspective. The richness of

UML is its own undoing, since it prevents its models from offering any clear perspective on the direction the design should take.

2.4 Universal definitions of notation elements transfer poorly

The elements of a universal notation must have an interpretation for analysis, design and implementation. Developers often take the most concrete examples of notational elements in use (because this is what they can understand) and retrofit these interpretations higher up in the analysis process. This impedes analysis, by imposing implementation concerns too early. The problem is more subtle than simply asserting that UML definitions are too concrete; it is rather that universalism fosters the transfer of elements and their definitions out of context.

Nowhere is this clearer than with the fundamental definitions of *association* and *dependency* in UML 1.3 [BRJ99]. An *association* is a concept from ERM that is used initially to represent some imagined relationship between entities. It has no immediate concrete interpretation. A *dependency* is likewise defined quite abstractly in UML as a directed relationship in which the behaviour of the source is functionally dependent on the behaviour of the target. However, when proceeding to the design level, UML asserts quite categorically that associations represent structural relationships that would require pointers (or embedding) in the implementation, whereas dependencies correspond to non-structural relationships, such as when one class uses another as an argument in its method signatures. This mutually exclusive definition is retrofitted on the analysis perspective and prevents *associations* from being seen as kinds of *dependency*; instead, developers are encouraged to make early distinctions between associations and dependencies in analysis [BRJ99, p74], decisions which really concern the implementation structure of systems. The notion of *dependency* is too important to be relegated to non-structural dependency only; several outstanding problems regarding the expressiveness of UML follow from this.

3. USE CASES AND USE CASE MODELS

Use cases were hailed in the field of object technology [JCJO92] as the first serious attempt to elicit requirements upstream of object modelling (though business tools such as *RequisitePro* may prove a better starting point). In OBA [G90; RG92], *scenarios* were originally collected during interview as *instances* of user-interaction, each describing a single execution path, like a procedure invocation. The view adopted by UML 1.1 [R97] gave a *use case* a *type*-level interpretation, rather like a procedure description. UML 1.1 also established the «uses» and «extends» generalisation relationships between use cases as a kind of novel control structure in the analysis domain. UML 1.3 now defines a *scenario* as an *instance* of a *use case* [BRJ99] and has replaced the «uses» and «extends» relationships with new «include» and «extend» dependency relationships.

3.1 Fixes made to use cases in UML 1.3

The UML 1.1 use case model was perhaps the most contentious and most widely misapplied part of UML. Originally, the «uses» and «extends» relationships were defined as specialised subtypes of the inheritance relationship between use cases (*stereotypes of generalisation,* in the jargon [R97]), using the standard generalisation arrow. In our previous survey [SG98a,b], we reported how developers simply disbelieved the official UML 1.1 semantics, preferring to infer different meanings for «uses» and «extends», with a number of consequential problems in communication and model consistency. Nowhere was this more painfully evident than during the OOPSLA 1998 use cases panel, in which Jacobson and Cockburn gave conflicting definitions on the meaning of «uses» without realising it, and some 80% of those present indicated that they failed to understand the «uses» and «extends» relationships [FCJAG98].

Jacobson's «uses» relationship, adopted in UML 1.1, was analogous to subclassing, in which a concrete use case "inherited" elements of an abstract use case and inserted them into its own sequence. In contradiction of this definition, developers commonly imagined that «uses» had the semantics of a simple subroutine call, in which the using case invokes the used case as a whole; they thus ignored the *generalisation* arrow semantics and the interleaving of elements.

The «extends» relationship was inconsistently defined from the start. On the one hand, the extension case was modelled as "inheriting from" the base case, but simultaneously was regarded as "inserting behaviour into" the base case. Both views cannot be held consistently: the extension must either represent the extra behaviour (insertion semantics), or the combination of the base and extra behaviour (specialisation semantics). In practice, the former semantics were applied, in violation of the metamodel description. UML 1.3 addressed these problems by abandoning the old «uses» and «extends» relationships. Their replacements are known as «include» and «extend». The main change is that these are defined as *stereotypes of dependency* (kinds of functional dependency) [BRJ99]. The effect of this change has been to remove the earlier conflicts with the generalisation/ specialisation semantics.

A number of problems still remain. Although use cases are supposed to be independent of any formal design, such that they "cannot be forward or reverse engineered" [BRJ99, p239], the conceptual structures fostered by use cases mislead developers about design structures. Logical faults are introduced, which prevent the use case model from scaling up to large systems.

3.2 Inadequacies of the «include» and «extend» relationships

The semantics of «include» is under-specified (#1: AMB)

By abandoning the old view that «uses» specialises an abstract base case and interleaves procedural elements, it would appear that UML is bowing to popular pressure for a straightforward *compositional* semantics for «include», as recommended by OPEN [FHG97] and as practised already by [C97a,b] in disregard of the UML 1.1 semantics. All of the examples given in [BRJ99, p227, 230, 337] are *consistent* with subroutines, rather than interleaved routines, but the authors fail to make absolutely clear whether this semantics is intended, since at the same time the *included* case is called "an aggregation of responsibilities" (p227) that is determined by "factoring out" (p226) common behaviour, which is reminiscent of generalisation [JCJO92, p170-173]. What we would like to see is a plain assertion that *included* cases are atomic subroutines that are executed as a whole.

The semantics of «extend» cannot handle exceptions (#2: ADQ)

By abandoning the inconsistent view that the old «extends» is both an insertion and a specialisation, this allows the new «extend» relationship to be considered purely as an insertion. The inserted optional behaviour is executed if a trigger condition is satisfied [BRJ99, p228]. This definition is equivalent to a guarded block, or a single-branch *if*-statement, in which control returns to the point of call afterwards.

As well as *optional* branches, developers commonly use «extend» to indicate *exceptions*; and this intention is also clear in [JCJO92, p165]. Unfortunately, the insertion semantics of «extend» does not support exceptions. When an exception is raised, control *never* returns to this point, but may return to the *end* of the failed transaction after the exception has been processed, or not at all.

The semantics of «extend» cannot handle alternative history (#3: ADQ)

Likewise, developers commonly use «extend» to indicate alternative history. Figure 1 overleaf shows a typical example (adapted from [C96]) in which *PayDirect* is intended as an alternative to *SignForOrder*; likewise *ReturnGoods* is intended as an alternative to *PayInvoice*. Under the insertion semantics of «extend», this diagram is nonsensical, because the base cases would still execute, once the inserted optional behaviour had terminated (see also problem #5 below).

Jacobson originally expected the «extend» relationship to be able to characterise insertions, exceptions and alternatives [JCJO92; p165]. It is clear from the published semantics of «extend» that it can only handle the first of these. Use case diagrams are therefore dangerously ambiguous; developers have to rely on intuitions about the labelling of the cases to establish the intended logic, in disregard of the official semantics. The «extend» relationship is not one, but several different relationships that have been conflated.

3.3 Misdirection fostered by use case development

Use case modelling misses long-range logical dependency (#4: MIS)

Use case modelling promotes a highly localised perspective which often obscures the true business logic of a system. As a result, developers fail to capture important long-range dependencies. In figure 1, the extension cases aim to capture local alternatives (problem #3 notwithstanding). However, UML does not capture explicitly the exclusive alternation of *PayDirect* with the more distant main cases *SendInvoice* and *PayInvoice*; likewise, the *PayDirect* and *ReturnGoods* extensions are secretly inter-dependent. A customer who paid direct should not only *not receive* an invoice, but must *obtain a refund* in addition to returning faulty goods. Even simple examples like this exhibit unpleasant mutual interactions between extensions and between these and other base cases. Most alarming is the fact that the redress for the cash-paying customer is not captured at all - this logical loophole is completely obscured in the use case model.

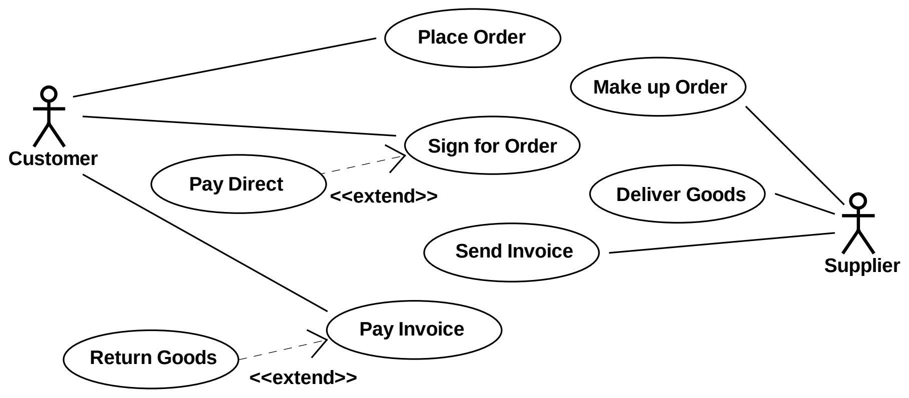

Figure 1: Pitfalls in a use case diagram

The granularity of use cases and logical task units are different (#5: ADQ)

Some of these problems could be fixed if UML admitted functional task units that were both larger and smaller than the nominal grain-size of use cases. A *use case* is defined as: "a sequence of transactions performed by a system, which yields an observable result of value for a particular actor" [JGJ97, p66]. The emphasis on *an observable result* is a deliberate constraint which seeks to ensure a minimum and maximum granularity: use cases may not be vacuous (they must deliver some useful result) and may not be multiple (they deliver a single result).

Unfortunately, to model alternative history requires *empty* base cases that may then be extended by each of the substantive alternatives. Unlike the doomed attempt to treat one alternative branch as the extension of the second branch (see problem #3), which fails under the semantics of insertions, both of the alternative branches could be legitimately modelled as insertions into some vacuous base case. Jacobson may even be accused of doing this [JCJO92; p164-165] where he considers *LogOn/LogOff* to be a base case into which the substantive alternative activities of a terminal session (such as *Compile, Mail* or *WordProcess*) are inserted. This base case is really of no observable benefit to the user!

Likewise, to model long-range logical dependency (see problem #4) requires *large scale* hierarchical units to co-ordinate the use cases that logically belong together. In figure 1, the true logical alternation is between the missing concepts *CreditPurchase* and *DirectPurchase*, not represented in UML, but which would appear as abstract intermediate nodes in a conventional structure chart. These are necessary to encode the long-range dependencies that exist between the use cases arranged under them. Neither of these would qualify a use case in UML 1.3, since they involve multiple transactions for many actors.

Use case dependency is non-logical and inconsistent (#6: INC)

The direction of dependency is misleading for «extend», flowing from the insertion to the base case that it modifies. This is an artefact of the order followed in the analysis procedure, rather than indicating any real logical dependency. Logically, it is the superordinate *selection* node, not represented in UML, but understood as the bundling of the base and all extension cases, which depends on all of its parts. The current UML 1.3 position is like saying that all branches of a multibranch-statement depend logically on one distinguished branch, clearly nonsense.

Whereas «include» can perhaps be construed as a genuine logical dependency, that of a *sequence* node on its subroutines, «extend» cannot; it merely records how the analysis paperwork was indexed. These are not stereotypes of the same dependency, they are two *completely different* kinds of relationship.

Use case modelling is unsound and must be deconstructed (#7: MIS)

This non-logical formulation of dependency also explains why «extend» cases break encapsulation, unfairly gaining access to the internal structure of the main cases they extend [G97]. In a logical formulation, all alternatives would be encapsulated inside the dispatching selection node. It also explains why «extend» seems to behave like a *come from* instruction, seizing control from the main flow.

While the UML authors may undermine attempts to impose logical consistency by asserting that use cases “cannot be forward or reverse engineered” [BRJ99, p239], the consequence of promoting non-logical relationships is that analysts will develop illogical use case models that have to be *completely deconstructed* later during design. Developers are misguided from the start.

4. SEQUENCE AND COLLABORATION DIAGRAMS

Sequence diagrams and collaboration diagrams are two equivalent ways of illustrating object-to-object message interactions in UML [BRJ99]. A sequence diagram makes the time ordering of messages explicit, but hides the structural relationships between objects. A collaboration diagram displays the structural connections between objects and superimposes on this a sequence of messages. The sequence diagram may show the focus of control (stack frame invocation level), but is limited in the degree of branching it can accommodate; whereas the collaboration diagram may illustrate more sophisticated branching and iteration.

4.1 Fixes to sequence and collaboration diagrams in UML 1.3

A number of missing emphases have been added since UML 1.1, which address some of the criticisms raised in [SG98a,b]. A sequence diagram may be used to represent *either* a single execution of a single decision path, *or* a procedural view of the decision paths available for execution. In the former case, the sequence diagram models a *scenario*, whereas in the latter case, it models a *use case* (see 3 above). This still puts the onus on the developer to realise when the diagram is being used in either sense. The failure of sequence diagrams to handle all but the simplest kind of branching is acknowledged. The UML 1.1 policy of splitting and merging an object's timeline when it enters an alternative history seems to have been abandoned. Focus bars now have a clear stack-frame semantics and are correctly mandated for every *call-back* and every *self-delegation* [BRJ99, p247]. Unfortunately, the UML authors don't observe their own rule on p252, so we list focus bar interpretation as a continuing problem for developers.

Collaboration diagrams are largely unchanged. The interpretation of the complicated syntax for branching and iteration is explained briefly on p249, but the conventions for referring to iteration variables are not explained. An alarming new concept is the admission of *non-procedural message flow.* This serves to confuse further the object/messaging and dataflow/flowchart perspectives. In our previous critique [SG98a,b], we reported how UML had subverted the original meaning of the term *collaboration* from RDD [WWW90]. UML still lacks the concept of a *class-level* client-server functional dependency (see problem #29).

4.2 The method invocation and dataflow/workflow duality

Sequence diagrams developed from use cases produce dataflow (#8: MIS)

In a properly-constituted sequence diagram, the meaning of the arrow stimulus is a *message* sent to *activate a method* in the target object, the receiver. The arrow has the semantics of an invocation. Against this, we find time and again that developers do not have this perspective when drawing sequence diagrams. Instead, they tend to create *dataflow diagrams*, in which the meaning of the arrow is the transfer of information to destinations corresponding to imagined processes and datastores. This persistent failure to adopt a proper object-oriented mind-set is disturbing.

Eventually, we put this behaviour down to inadequate prior object modelling. If developers proceed directly to sequence diagrams from use cases, the kinds of object-concept available at this early stage relate almost exclusively to human actors and passive datastore concepts, such as letters, forms, price and stock records. There is a tendency to model the nouns in the description of the use-case literally in the sequence diagram. Sentences like: *the warehouse manager looks up the price and the stock-level* result in messages between the external *WarehouseManager* actor and object concepts representing a *PriceRecord* and a *StockLevelRecord*. Instead, price and stock-level should most likely be access methods of a *GoodsItem* object.

To counter this, developers must be prevented from drawing sequence diagrams as a primary model; instead, they should concentrate on eliciting responsible object abstractions [BC89, WW89, WWW90]. Sequence diagrams were deployed *much later* in OOSE to *confirm* that the object model covered the behaviour expected in use cases [JCJO92]; this has also been found useful in testing [M97].

Flat, or non-procedural message flow is workflow (#9: MIS)

In UML 1.3, the authors recognise both procedural and non-procedural message flow between objects in collaboration diagrams [BRJ99, p213]. The former is drawn with the filled arrowhead and has the semantics of method invocation, in which messages may nest. The latter is drawn with the stick arrowhead and has the flat semantics of "nonprocedural progression of control from step to step", in other words, the workflow in a flowchart. Workflow has no place in an object-messaging model and only serves to distract developers.

The return arrow is dataflow in an invocation model (#10: MIS)

UML 1.1 introduced the dotted-shaft return value arrow to distinguish the passing of return values from call-back invocations. A new problem is that this is now the only *dataflow* in an *invocation* model. It confuses developers about the intended meaning of arrows in sequence diagrams. If focus bars are used consistently (see problem #12), then return arrows are unnecessary - they should be dropped, except to express a request from a concurrent thread to resynchronise (a rendezvous).

Focus bars are misunderstood and used inconsistently (#11: INC)

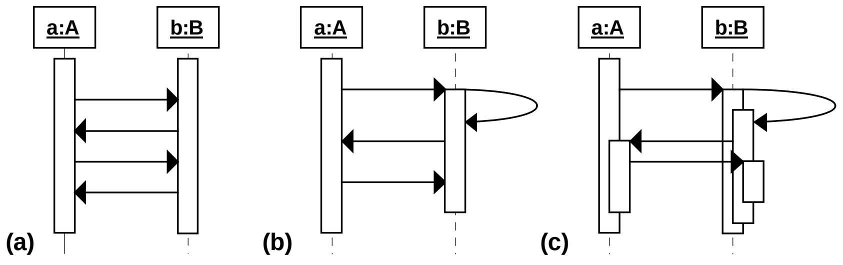

Figure 2: Thread, activation and stack-frame semantics of focus bars

Focus bars should have a *stack-frame* semantics, as shown in figure 2c. A nested focus bar must be shown for every *call-back* and *self-delegation* [BRJ99, p247]. It should be clear from the nesting and length of focus bars which messages are the master routines and which are the subroutines. In UML 1.1, a second *process thread* semantics was allowed [R97], shown in figure 2a. In this case, the length of the focus bar indicates the liveness of the thread. Most developers, including [BRJ99, p252] misuse focus bars in the style of figure 2b. The length of the bar conveys no useful logical information, corresponding only to the interval between when an object was first and last touched.

Absent, or misused focus bars promote dataflow over invocation (#12: MIS)

The style of figure 2b also visually promotes a conflicting *dataflow* semantics for the stimulus arrow, since it cannot now mean *invocation* (as no nested focus bar is raised). A similar visual conflict arises if no focus bars are displayed. The correct style of figure 2c promotes a proper *invocation* semantics. From this, it is always possible to determine when a subroutine terminates and to which caller the return value should pass. The value returned is implicit in the request, so need not be annotated, thus freeing the diagram from another dataflow (see problem #10).

4.3 Adequacy and expressiveness of interaction diagrams

The normal course plus extensions model is a fiction (#13: MIS)

Sequence diagrams are supposed to be drawn for each use case. A use case starts from the premise that you can construct a *normal course* of events and supplement this with various *extensions*. This works only for simple examples. In general, business logic is much more complicated, with multibranching decision paths. In a credit reinsurance system developed for BTR [HSR98], there were four ways a credit limit application could succeed and four ways in which it could fail.

The natural business logic is structured such that *early rejection* or *acceptance* cases are spun off the *continue to investigate* creditworthiness case, 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 eventually occur with equal likelihood. The analysts were artificially constrained into accepting the *continue to investigate* case as the normal course, despite the fact that this is a fairly unlikely path through the system.

The granularity of use cases and logical sequence units is different (#14: ADQ)

In the same credit reinsurance system [HSR98], the designers rejected the analysis model from (problem #13), because the different cases were of unequal size and complexity. The modularisation of the system was judged to be poor. Instead, the designers wanted to truncate use-cases at points where conditions and branching were introduced, because of the weak support given to branching and iteration in sequence diagrams.

The designers represented in a single diagram the initial portion of the *continue to investigate* case up to the first *early rejection* point; then spun off a second sequence diagram to cover the continuation of the case up to the first *early acceptance* point; and so on. This was judged habitable, because each sequence diagram supported a single branch of the logic and was of a comparable modular size. However, the granularity principle for use cases was broken (see problem #5), since many cases were spun off which did not correspond to a single, complete interaction of a user with the system. Cases also acquired unlikely sounding names, such as: *Early Credit Limit Acceptance that was Previously Not Rejected Early*.

Decision logic in sequence and collaboration diagrams is limited (#15: ADQ)

In UML 1.3, a branching condition is indicated in [ ] brackets and placed as a guard on message stimuli. The guards for all branches at a fork must be mutually exclusive; only one branch is selected at one time. This allows client objects to dispatch requests to alternative servers, or to dispatch alternative requests to the same server, but cannot represent the client object entering a different timeline [SG98a,b].

UML 1.1 introduced the splitting of object timelines in sequence diagrams [R97], but this seems to have been dropped in UML 1.3 [BRJ99]. Instead, objects may occur multiple times in a collaboration diagram [BRJ99, p254] if they enter a substantively different state. In our earlier critique [SG98a,b] we noted how business logic tends to produce many overlapping and parallel timelines representing alternative ways of reaching the same outcomes, but which are contingent on having passed through different histories. To split and join multiple timelines is impossible in a single sequence diagram and clutters a collaboration diagram. The need to enter an alternative timeline forces the creation of another diagram (see problem #14) which also forces the premature truncation of use cases (see problems #5 and #14).

4.4 Conflicts between object messaging and decision logic

The sequence perspective generates incorrect control logic (#16: MIS)

Sequence diagrams are often used in ways that emphasize object interactions at the expense of proper decision logic. One of the models developed for the credit reinsurance system [HSR98] (see problems #13, #14) produced an example in which an *early rejection* point should logically have terminated the timeline; however, because of the CASE tool's inability to support branching and alternative timelines, the developer had continued using the same object timeline for the *continue to investigate* case, which was logically inconsistent.

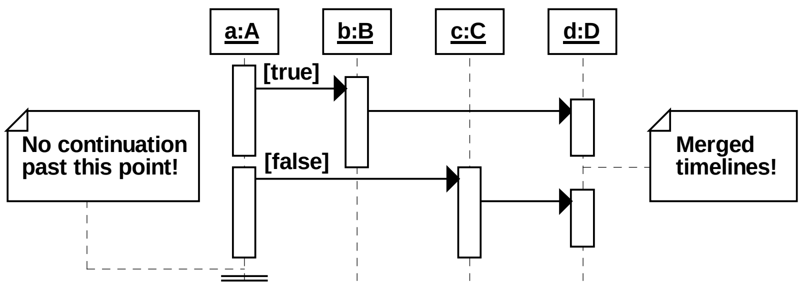

Figure 3: Pitfalls in a sequence diagram

This logic fault is equivalent to continuing past the termination point indicated in figure 3. Sequence diagrams are problematic where they suppress the developer's perceptions of proper decision logic. A timeline must either represent a single sequence of actions (with no branching), or a mutually exclusive set of guarded alternative histories, as in figure 3.

Alternative timelines are combined and misread (#17: MIS)

Sequence diagrams also merge object timelines which are logically distinct, especially when they are at several removes from the point of branching. These timelines are easily misread by designers and result in merged processing streams in the implementation. Figure 3 illustrates a case where the two alternative processing branches initiated by the object *a:A* are eventually merged as a single timeline in the object *d:D*. It is possible to view this as a single history for *d:D* when in fact this represents two alternative histories.

5. STATE AND ACTIVITY DIAGRAMS

UML's *state diagram* derives from the *dynamic model* of OMT [RBPEL91] and ultimately from Harel's *statecharts*. It is a fusion of *Moore* and *Mealy* finite state machines in which computational activity may be attached to transitions (*Mealy*) and to states (*Moore*) [BRJ99, p336]. Each state at one level may be expanded to a substate machine, to model this computational activity. Arcs entering a superstate correspond to commencing the substate machine in its initial substate; arcs leaving the superstate boundary indicate exit transitions from *all* substates.

UML's *activity diagram* is a synthesis of a state machine, a flowchart and a Petri net. Described in state machine terms, the so-called *action* and *activity states* are really processing stages in a flowchart and the transitions represent program conditions rather than events. Diamond decision nodes indicate branching; and Petri-style synchronisation bars indicate concurrent fork and join points.

5.1 Status of state and activity diagrams in UML 1.3

Since UML 1.1, the scope of the state diagram has been widened to model whole system behaviour, rather than merely local object behaviour; and the importance of the activity diagram has been elevated, in line with our earlier recommendations [SG98a,b]. The activity diagram is much better suited to modelling real business processes than either of the interaction models (see problems #13-#17 above), since it models workflow, rather than object messaging. Alternate and parallel paths through the business logic may easily merge at points where the outcome is thereafter the same.

Nonetheless, we still find that developers make too little use of this model and proceed to interaction modelling at a far too early stage. Likewise, state models are under-used and we occasionally still see instances of the "use case too far" syndrome in which developers try to encode all the business logic in use cases [SG98a,b]. We have also identified a number of new inconsistencies since our last survey.

5.2 Interpretation problems with initial and final states

The initial state is both an indicator and a state (#18: AMB)

Developers do not know whether the *initial state* represents a true state, or merely points to the first substantive state. The Harel-semantics conflict with Mealy-semantics in which a free transition arrow proceeds to the first substantive state.

Harel-semantics encourages the view that the *initial state* is a real state, in which the system is dead, not yet switched on. However, when used to indicate the starting point of a substate machine, the *initial state* icon cannot represent a dead system state; it is used exactly like the Mealy initial free transition arrow. Similarly, since a majority of UML state diagrams place no event on the initial transition, the *initial state* may as well not be a state.

The final state is both an accept state and a halt state (#19: AMB)

Developers do not know whether the final state represents a true *accept state* in the system, or merely a *halt state* after the system has terminated and is dectivated. The Harel-semantics conflict with the Mealy-semantics in which the ringed state is an *accept state*, that is, the last substantive state in the machine.

The *accept state* view causes developers to place *final state* icons in substate machines [BRJ99, p303], to indicate that these eventually terminate. Under Harel-semantics, this indicates the premature halting of the whole system, rather than the successful completion of a substate machine. If the *final state* icon is reserved for a *halt state* (*after* the last substantive state), there is no way to indicate a true *accept state*. This mostly makes it harder to see where a substate machine terminates, without breaking the encapsulation of nested state machines (see problem #21).

5.3 Consistency problems when decomposing state machines

Transitions across a superstate boundary violate encapsulation (#20: ADQ)

The advantage of Harel statecharts is supposed to be that they allow the developer to view the control structure of a system at different levels of abstraction, hiding different amounts of detail inside substate machines. However, if major alterations to the state transition pattern are needed when a superstate is exposed as a substate machine, this negates the benefit of encapsulation and abstraction.

UML state diagrams routinely allow transitions to state boundaries at one level of description to be redrawn to connect with substates when these are exposed [BRJ99, p299, 301, 333, 437]. This breaks the encapsulation of state machines. The labelling of such transitions at the superstate and substate levels must often be different (see problem #21). Further problems of encapsulation are caused by admitting *shallow* and *deep history* connectors (see problem #24).

Boundary crossing makes a nested accept state redundant (#21: ADQ)

The most irritating example of boundary-crossing (see problem #20) is the practice of identifying an *accept state* in a substate machine (see problem #19) by a free transition, which exits this substate across the superstate boundary. This transition would have to be properly labelled with an *event* at the higher level (triggered by completion of the substate machine), thus illustrating the inconsistency of labelling at each level. Furthermore, a free transition exiting a nested *accept state* effectively makes this state redundant - it might as well not be there; instead, the penultimate states could exit directly when their events were triggered.

These points illustrate how boundary crossing breaks encapsulation by upsetting the state machine logic at both the higher and lower levels. Instead, substate machines should not connect directly across the superstate boundary. Reaching the *accept state* of a substate machine should be deemed equivalent to signalling a separate event at the higher level.

States with free exit transitions are not decomposable (#22: INC)

Sometimes, states representing processing stages, in the sense of a *Moore*-machine, have free exit transitions, meaning that the state may be quit automatically once its associated processing has terminated. However, if such a state is ever expanded, there is an immediate problem in interpreting the free exit transition consistently, since under Harel-semantics, a transition leaving the superstate boundary is equivalent to an exit transition from every substate. This would mean that every substate exits immediately! The substate machine cannot execute.

5.4 Misdirection in capturing the underlying control logic

Conditional guards conceal a duplication of control states (#23: MIS)

The admission of extra *conditional guards* on events in UML 1.3 undermines the purpose of the state machine formalism and conceals a duplication in control states. Figure 4a illustrates a simple heating system whose basic events are temperature triggers, which have been augmented by timing guards to introduce a delay in switching.

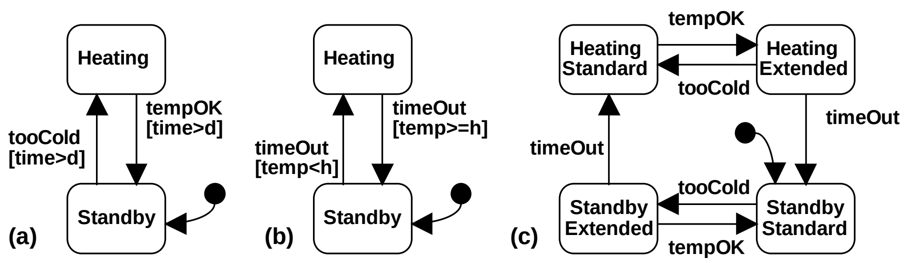

Figure 4: Pitfalls in state machine logic

This model is exactly equivalent to figure 4b, in which the basic events are timeouts, augmented by temperature threshold guards, illustrating how the augmented notation makes the choice of events essentially arbitrary! In figure 4c the guards are revealed for what they are: extra hidden control states. Because conditional guards lead to self-deception about the real control structure of systems, we advise our developers to expand them away, so that they are at least aware of the real number of states and events in the system.

History connectors conceal a multiplication of state machines (#24: MIS)

UML 1.3 admits *shallow* and *deep history* connectors [BRJ99, p301]. Upon reentry to a superstate containing a *shallow history* connector, the substate machine resumes where it last left off, rather than starting in its initial state. This enrichment not only breaks encapsulation (problem #21), since it requires reasoning across state boundaries, but also corresponds to a *repeated duplication* of the *entire super state machine*, once for each alternative remembered substate! A *deep history* connector applies this expansion recursively, to all nested substate machines.

We appreciate that this approach is typically thought necessary to model interrupts and co-routines; unfortunately it blinds most developers to the real complexity of what they have created. We discourage its routine use.

Entry/exit actions and internal transitions promote flowcharts (#25: MIS)

The simple logic of state machines is upset yet again by the concern to execute attached procedural elements in the correct order [BRJ99, 295-298]. For example, a distinction is drawn between standard *self-transitions*, which trigger entry/exit actions, and *internal transitions*, which do not. This imposes a procedural flowchart perspective, rather than an event-driven one, when designing state diagrams.

We think that a self-transition should always be equivalent to dwelling in a state, rather than exiting and re-entering. Self-transitions typically decompose into lower-level transitions in the substate machine. This leads us to conclude that there should only be *one* activity attached to a state, modelled by its substate machine.

6. CLASS DIAGRAMS

Class diagrams are perhaps the most stable and widely used part of UML, since they translate in a straightforward way into program classes. Paradoxically, this can lead to developer overconfidence after the analysis stage and inadequate object modelling in the design stage. The strength, and also the weakness, of UML's class diagram is its ability to capture a wide variety of semantic relationships, which are *either* the anticipated, but as yet uninterpreted associations between entities in the analysis domain, cf. OMT [RBPEL91]; or the *actual* structural and functional connections between classes in the design domain, cf. *has* and *uses* in Booch '94 [B94].

Eventually, it is the richness of the representation which confuses developers. They are wrestling simultaneously with analysis and design perspectives, with data modelling and client-server functional dependency perspectives, all in the same diagram. A number of these problems were already described in section 2.

6.1 The premature curtailment of object modelling

Class diagrams tend to fix object abstractions too early (#26: MIS)

As we noted in sections 2.1 and 2.2, class diagrams drawn during the analysis phase exercise an undue influence on the eventual design. Initial class diagrams contain obvious domain concepts, related by uninterpreted associations. These are typically pressed into design in a naïve way. Developers do not reconsider the true nature of functional coupling between classes, so they simply add behaviour to the initial domain concepts, eventually overburdening them. They are hindered from thinking more imaginatively about classes as smaller, less concrete, agent or manager concepts exercising limited sets of responsibilities [WWW90].

Associations fix object connections and translate poorly into design (#27: MIS)

Again, in section 2.2 we noted how the recording of uninterpreted associations during analysis produces poorly-coupled designs, because these associations do not necessarily correspond to the optimal functional couplings in a design. As with the object abstractions, it is hard to undo initial perceptions. Associations are often pressed prematurely into physical class connections (viz. pointers). The universalism of UML fosters this (see 2.1).

We noted in [SG98a,b] how automatic translation schemes also resulted in overly-coupled designs, with set-valued attributes and *Observer*-style mutual connectivity between unimaginative domain classes (see also problem #26). In any case, the association perspective is not the correct one for optimal class coupling (see problems #29, #30).

Detailed analysis labelling impedes structural transformation (#28: MIS)

In [SG98a,b], we noted how developers worry inordinately over detailed labelling concerns, such as disjoint, versus overlapping, subclasses; or the difference between an association, a shared aggregation and a composite aggregation; or how far to go with adornments such as OCL constraints and stereotypes. While it is important to ensure that the semantic characterisation of the analysis domain is preserved in the design and implementation stages, many subtle annotations on analysis models are often irrelevant later in design, especially if model structure is transformed.

Consider the *Mediator* pattern [GHJV95], which can be applied systematically to reduce inter-object coupling [SSH98]. It matters little whether the *Mediator* itself is a true composition, a shared aggregation, or simply some kind of coordinating abstraction, so long as all cross-connections are removed between the mediated objects. Applying the *Mediator* pattern radically alters the structure of a system, deleting relationships and their adornments. This would seem to indicate that developers should be wary of investing too much effort in analysis labelling. Unfortunately, we find a reverse effect, which is that developers are motivated against applying radical structural transformations to the design, because of the earlier effort invested in labelling the structure of the analysis model.

6.2 The functional coupling and data dependency duality

The term collaboration *is misconstrued and the concept is missing (#29: ADQ)*

UML 1.3 still lacks an unequivocal concept for an abstract, class-level, client-server functional dependency. This is arguably the most important relationship in object-oriented modelling, the dual of an association in data modelling. In RDD, this concept is known as a *collaboration* [BC89; WW89; WWW90].

Unfortunately, the term *collaboration* has been subverted in UML to refer instead to a cluster of objects and their interactions; hence the use of *collaboration diagram* to refer to an object *interaction* diagram. This is an instance-level model, in which the arrows denote individual messages to objects. In RDD, a *collaboration* is an abstract, directed relationship between two classes, representing the functional dependency of one class on the other. This missing perspective is critical for the proper modular analysis and transformation of systems [SSH98, S98].

The available UML concepts do not cut the cloth in the same way: a *directed association* represents a concrete structural connection indicating navigability in one direction. A *dependency* indicates a non-structural dependency on method arguments (see 2.4). Both make premature assumptions about implementation and fail to capture the abstract notion of client-server dependency.

The class diagram mixes data and functional dependency (#30: MIS)

UML class diagrams freely mix *associations* with *dependencies*, thus confusing the *data modelling* and *functional dependency* perspectives. It is not clear whether *dependency* stretches to include *existence dependency* also (like the *derived* relationships in OMT [RBPEL91]). In section 2.3 we noted how such eclecticism leads to diagrams that offer no clear perspective on the direction in which a design should proceed. The class diagram fails to resolve the competing forces.

In its proper context, an *association* hails from data modelling (ERM) and is used to establish minimal coupling between data files. The RDD notion of a *collaboration* hails from responsibility analysis (CRC) and is used to establish minimal functional coupling between subsystem modules [BC89; WWW90; SSH98; S98]. Applying either technique will optimise a class diagram in a different way. UML class diagrams do not in any case offer an abstract *collaboration graph* [WWW90] perspective (see problem #29), which would allow subsystem optimisation to proceed from an analysis of inter-module functional dependency.

7. CONCLUSIONS

Our survey has emphasised the way developers embrace and use UML 1.3. 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 30):

INC	(inconsistency)	3 counts
AMB	(ambiguity)	3 counts
ADQ	(adequacy)	8 counts
MIS	(misdirection)	16 counts

The high level of misdirections (MIS) encountered in our survey is worrying. The ratio of MIS to the other three (INC, AMB, ADQ) has grown since our previous survey [SG98a,b], partly as a result of fixes made in UML 1.3 to inconsistencies and ambiguities in UML 1.1. However, the absolute increase in MIS scores (previously 12 out of 37) shows how developers are finding more ways to confuse themselves with UML. This is alarming, not just because it represents a waste of effort, but because it is not a problem which can be fixed simply by trying to clarify the semantics of UML as it stands; instead, large chunks of UML need to be reconstructed to take into account the ways in which developers' minds operate.

It is clear from Gestalt psychology how important initial conceptualisations are; and how much they influence subsequent concept formation. If the initial UML analysis view is of a pre-normalised data model, and if communications between entities are conceived initially as dataflow (or workflow), then substantial mental effort is required to undo these faulty perceptions. It involves going against the tide, fighting against the conceptual clutter in the minds of developers.

The INC and AMB scores reflect on problems with the *consistency* of UML, whereas the ADQ scores reflect on problems with the *completeness* of UML. These counts suggest that further revisions to UML are necessary. In particular, the authors identified 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 [G95], *Discovery* [S98] and OPEN [FHG97; HSY98]. 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.

References

[BC89] 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.

[BRJ99] G Booch, J Rumbaugh and I Jacobson (1999), The Unified Modeling Language User Guide, Addison Wesley Longman.

[B94] G Booch (1994), Object-Oriented Analysis and Design with Applications, 2nd edn., Benjamin-Cummings.

[CY91a] P Coad and E Yourdon (1991), *Object Oriented Analysis*, Yourdon Press.

[CY91b] P Coad and E Yourdon (1991), *Object Oriented Design*, Yourdon Press.

[C96] A Cockburn (1996), *Basic Use Case Template, TR.96.03a,* rev. 1998, Humans and Technology; also pub. *http://members.aol.com/acockburn/papers/uctempla.htm*

[C97a] A Cockburn (1997a), "Goals and use cases", *J. Obj.-Oriented Prog., 10 (5),* 35-40.

[C97b] A Cockburn (1997b), "Using goal-based use cases", *J. Obj.-Oriented Prog., 10 (7),* 56-62.

[FHG97] D Firesmith, B Henderson-Sellers and I Graham (1997), *OPEN Modelling Language (OML) Reference Manual,* March, SIGS Books.

[FCJAG98] M Fowler, A Cockburn, I Jacobson, B Anderson and I Graham (1998), "Question time! About use cases", *Proc. 13th ACM Conf. Obj.-Oriented Prog. Sys., Lang. and Appl., pub. ACM Sigplan Notices, 33 (10),* 226-229.

[GHJV95] E Gamma, R Helm, R Johnson and J Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley.

[G90] E A Gibson (1990), "Objects born and bred", *BYTE magazine, 15(10),* 255-264.

[G95] I Graham (1995), *Migrating to Object Technology*, Addison-Wesley.

[G97] I Graham (1997), "Some problems with use cases... and how to avoid them", *Proc. 3rd Int. Conf. Object-Oriented Info. Sys.,* eds D Patel, Y Sun and S Patel, (London: Springer Verlag), 18-27,

[HSY98] B Henderson-Sellers, A J H Simons and H Younessi (1998), *The OPEN Toolbox of Techniques,* Addison-Wesley.

[HSR98] 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.

[JCJO92] I Jacobson, M Christerson, P Jonsson and G Övergaard (1992), *Object-Oriented Software Engineering: a Use-Case Driven Approach*, Addison-Wesley.

[JGJ97] I Jacobson, M Griss and P Jonsson (1997), *Software Reuse: Architecture, Process and Organisation for Business Success,* Addison-Wesley and ACM Press, Reading MA, USA, 497pp.

[M97] J D McGregor (1997), *Testing Object-Oriented Components, ECOOP '97 Tutorial 2* (Jyväskylä, AITO/ACM).

[R97] Rational Software (1997), *UML 1.1 Reference Manual*, September, *http://www.rational.com/uml/* .

[RG92] K Rubin and A Goldberg (1992), "Object-behaviour analysis", *Comm. ACM, 35(9).*

[RBPEL91] J Rumbaugh, M Blaha, W Premerlani, F Eddy and W Lorensen (1991), *Object-Oriented Modeling and Design,* Prentice-Hall.

[S98] A J H Simons (1998), *Object Discovery - a Process for Developing Medium-Sized Applications, ECOOP '98 Tutorial 14,* (Brussels, AITO/ACM), 90pp.

[SG98a] A J H Simons and I Graham (1998), "37 things that don't work in object-oriented modelling with UML", *Proc. 2nd ECOOP Workshop on Precise Behavioural Semantics,* eds. H Kilov and B Rumpe, *Technical Report TUM-I9813* (TU Munich, Institut für Informatik), 209-232.

[SG98b] A J H Simons and I Graham (1998), "37 things that don't work in object-oriented modelling with UML", *British Computer Society Object-Oriented Programming Systems Newsletter, 35,* eds. R Mitchell and S Kent (BCS: Autumn, 1998), *http://www.oopsnl.ukc.ac.uk/Issue35Autumn1998/contents.html.*

[SSH98] A J H Simons, M Snoeck and K S Y Hung (1998), "Design patterns as litmus paper to test the strength of object-oriented methods", *Proc. 5th. Int. Conf. Object-Oriented Info. Sys.*, eds. C Rolland and G Grosz (Paris: Springer Verlag), 129-147.

[WN95] K Waldén and J-M Nerson (1995), *Seamless Object-Oriented Architecture*, Prentice Hall.

[WW89] 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.

[WWW90] R Wirfs-Brock, B Wilkerson and L Wiener (1990), *Designing Object-Oriented Software*, Prentice Hall.

About the Authors

Anthony Simons is a lecturer in Computer Science at the University of Sheffield, with 14 years experience using, researching and teaching object technology. He joined the department as a research assistant building speech recognisers; later he moved into mainstream software engineering. He has research interests in type theory, language design, analysis and design methods, and verification and testing. He is the author of over 50 research publications and the creator of the *Discovery Method* for developing object-oriented systems. He holds a PhD in object-oriented type theory and language design and a Masters degree in modern languages and linguistics.

Ian Graham is chairman of IGA Ltd, a consultancy specialising in advanced information technology and change management. Ian has over 20 years experience as a practitioner in IT. Previously, he was a VP at Chase Manhattan Bank and senior manager at the Swiss Bank Corporation (now Warburg Dillon Reed). Ian created the system development methods for both Chase and SBC. He is the author of 9 books and over 60 papers and the creator of the *SOMA* object-oriented development method. Ian is a Fellow of the British Computer Society, has a Masters degree in mathematics and holds an Industrial Chair in Requirements Engineering at De Montfort University.