Use Cases Considered Harmful

Anthony J H Simons
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.
A.Simons@dcs.shef.ac.uk

Abstract

This article traces the unstable semantics of use cases from Jacobson to UML 1.3. The UML 1.1 metamodel formally defined the «uses» and «extends» use case relationships as stereotypes of generalisation, *yet both received interpretations that varied between inheritance and composition, reflecting a large degree of confusion among developers. The recently revised UML 1.3 has quietly dropped these in favour of new «include» and «extend» relationships, which are styled instead as kinds of* dependency. *Despite this change, the deployment of use case diagrams encourages analysts to conceptualise and develop models which conceal arbitrary jumps in the flow of control, corresponding to* goto *and* comefrom *statements, and in which unpleasant non-local dependencies exist across modules. A discussion of examples reveals how a conscientious designer must disassemble use case models completely to produce properly-structured code. A radical solution is proposed.*

1. Jacobson's original use cases

The recording of *use cases* was hailed as a significant advance in object-oriented analysis, since it allowed the process of requirements capture to begin further upstream than with naïve object modelling [4]. Different scripting techniques in object-oriented analysis appeared at around the same time: as *use cases* in OOSE [17], as *scenarios* in OBA [23] and as *task scripts* in SOMA [13]; although formalised scripts first appeared in Schank's work on conceptual dependency [24].

1.1. The size and representation of a use case

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" [19, p66]. The emphasis on a single *observable result* is a deliberate constraint which seeks to ensure a minimum and maximum granularity for a use case, which cannot be an incomplete sequence (offering no benefit), nor multiple sequences (achieving several benefits).

A use case is drawn as a labelled ellipse in a *use case diagram,* connected by lines to stick-figure *actors*, denoting the typed rôles of the participants in a use case, accompanied by an informal paragraph describing either: the interaction of an idealised user with a computer system [17], or: the performance by an idealised business stakeholder of an everyday work task [18]. Various authors have attempted to improve on this informality, for example: [9, 14, 15, 5, 6].

1.2. The type-level semantics of a use case

Whereas the *scenarios* of OBA were collected as *instances* of user-interaction, each like a single *execution* of a process [12, 23], in Jacobson's approach [17, p127-8] a *use case* was considered a *type*-level concept, more like a procedural *definition*. This is evident in statements such as:

> "Each use case is a specific way of using the system and every execution of the use case may be viewed as an instance of the use case" (p127) ... "When a use case is performed, we therefore view this as: we instantiate the use case's class" (p127-8). "The use case instance exists as long as the use case is operating." (p127).

Furthermore, Jacobson considered a use case to be a kind of *class*, which defined state variables, whose instances could record how far a particular thread of execution had reached [17, p128]. Curiously, this account resembles exactly a class having a procedural body, a construct present in the languages Simula [7] and Beta [20]. Defining a use case as a class was in keeping with Jacobson's meta-theoretic goals to have a single framework for describing both systems and the Objectory method itself. He was keen to see Objectory as a "use-case driven approach", that is, a method that could reflexively be described in terms of use cases. Use cases therefore had to follow the class and object model used elsewhere to describe software systems.

1.3. Unequal *uses* and *extends* dependencies

Jacobson described all optional, alternative and exceptional branching behaviour as *extensions* to a putative *main* use case [17, p163-5]. The style in which extensions were developed was motivated chiefly by the observation that it was inconvenient to have to return to the textual description of the "main flow of events" and edit this, just to incorporate new branch points; instead the extension named the point in the main case at which its behaviour was to be inserted. Visually, this dependency was indicated by making the *extends* arrow run from the extension to the main case. Curiously, this directionality is the exact opposite of the *selection* decomposition in traditional Yourdon or JSP structure charts. The meaning of an extension was: a guarded block of optional functionality that could be *inserted into* a main use case (p165).

The principle of minimising the impact of changes was abandoned when analysts were encouraged later to restructure their use cases. A common underlying set of interactions in several main cases could be pulled out as an *abstract use case* (so-called because it was not to be instantiated independently, since it did not refer to a complete sequence), which was then *used* by several *concrete use cases* [17, p170-3). Visually, this dependency was indicated by making the *uses* arrow run from the concrete to the abstract case. The meaning of a concrete *using* case was that it was equivalent to the original use case prior to refactoring, the only change being the internal way in which the information was described. In this respect, the *uses* relationship was considered to be a "kind of inheritance" (p170). The rationale for developing *uses* relationships (restructuring) therefore seems to contradict that for developing *extends* relationships (convenience). Why was it not equally appropriate to restructure use cases to include branch points? This asymmetric treatment lies at the cause of a major inconsistency in the way in which the notion of *dependency* is interpreted in UML 1.3 (see section 3.4).

1.4. The procedural meaning of *uses* and *extends*

The procedural interpretation of *extends* is over-simplistic (and challenged in section 3.2). Jacobson's main idea, which persists into UML 1.3, seems to have been that you can *insert optional behaviour into* existing procedural descriptions, and that this model should then be sufficient to cover: optional subroutines, alternative courses and exceptional cases [17, p165]. Jacobson's examples of extensions (p164-5) can all be interpreted as *guarded blocks*, that is, single-branch *if* statements that execute if the appropriate condition, or guard, is satisfied.

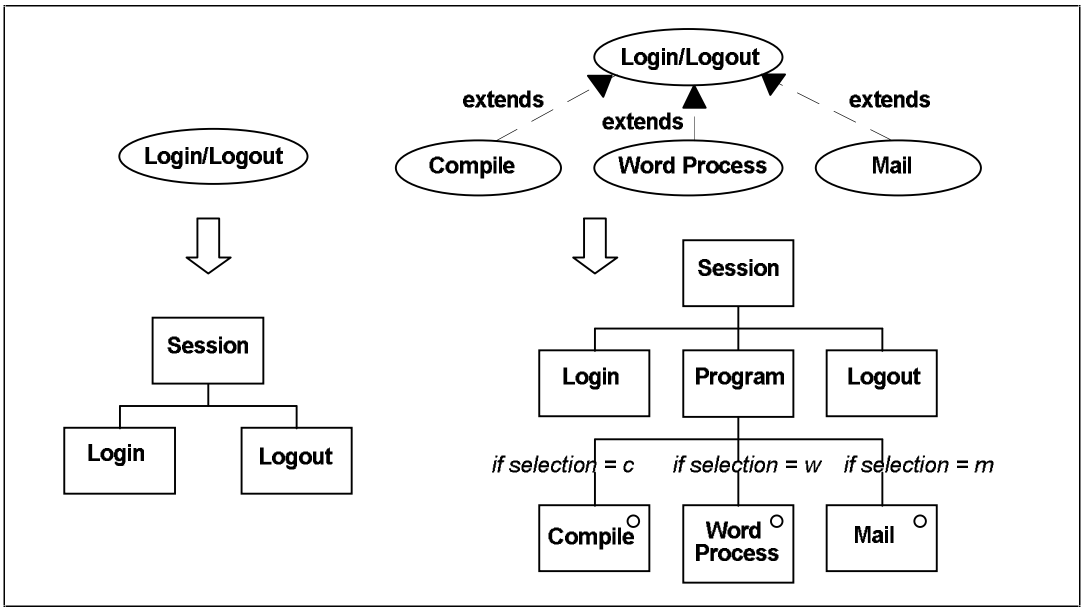

Figure 1: Jacobson's *extends* relationship translated into a structure chart

Translating this into a regular structure chart gives the comparison shown in figure 1. This shows how, from the point of view of traditional block structure, the behaviour of an extension can only be explained through an implicit selection node (here, the node labelled *Program*) inserted into the sequence of actions in the main case.

The procedural interpretation of *uses* is much more complex [17, p171-2]. If the abstract case were invoked *as a whole* inside the concrete case, this might be thought equivalent to procedural decomposition, or a subroutine call. Jacobson referred to such abstract cases as *atomic sequences* (p171). In general, though, abstract sequences were not atomic and the *uses* relationship did not follow the semantics of a subroutine call:

> The course need not be an atomic sequence, although this is often the case. However, we may have a situation where the abstract use cases can be used through interleaving them into the concrete use case" (p171).

To illustrate this, Jacobson provided an example of double interleaving, in which a concrete case interleaved elements of two abstract use cases into its own sequence (p172). We can compare this to the structure chart in figure 2, in which abstract elements A, B, C, D are interleaved with local elements X, Y, Z in the concrete case. From this it is clear that *uses* cannot mean a subroutine call, since it is impossible to construct any traditional block-structured relationship between the concrete and abstract cases as a whole.

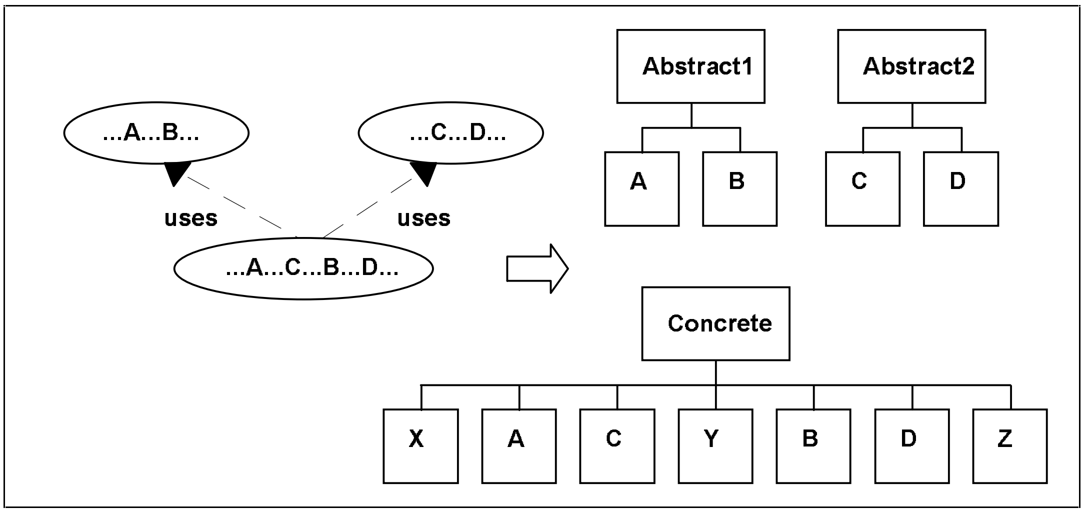

Figure 2: Jacobson's *uses* relationship translated into a structure chart

In fact, the terms *abstract* and *concrete* deliberately evoke the refinement of classes in a specialisation hierarchy. *Uses* corresponds much more to method combination, in which elements of general routines are inherited from ancestors and assembled in descendant classes. This is consistent with Jacobson's view of a use case as a class.

2. Evolution of use cases from UML 1.0 to UML 1.3

Use cases were adopted with type-level semantics in UML 1.0 and are now referred to as *classifiers*, which is UML 1.3's term for something which has instances and may define attributes and behavioural properties [3, p121]. The «uses» and «extends» relationships were formally defined in UML 1.1 as a *kind of inheritance* between use cases [22]. Because of the conceptual problems that this caused, and the fact that developers completely disregarded this semantics in the way in which they deployed use cases, UML 1.2 and 1.3 later abandoned this in favour of a more plausible definition based on a notion of *dependency* [3].

2.1. The semantics of «uses» and «extends» in UML 1.0 and UML 1.1

In the UML 1.1 jargon [21, 22], the «uses» and «extends» relationships were defined as *stereotypes of generalisation* and were drawn using the standard *generalisation* arrow. The term *generalisation* is used in UML to describe the inheritance relationship, which we think of normally as *specialisation*, but is renamed in UML to reflect the fact that the arrow points towards the more general entity. A *stereotype* is a label which is used to partition a class internally according to some semantic selection criterion. Thus, within the class of *generalisations*, there are two identifiable subclasses of *using* and *extending* relationships. Asserting that «uses» and «extends» are *stereotypes of generalisation* is equivalent to saying that they are subtypes of inheritance.

UML 1.1 continued to support Jacobson's view of «extends» as a guarded block that could be inserted into another procedure, but also introduced the notion of *extension points*, labels in the main use case that could be referred to by (and hence anticipated) extensions. Mostly, this was in preparation for CASE tools, which had to be able to relate the backward-pointing extension cases to equivalent forward-pointing selection branches in the design. The argument for making extensions "depend on" the main case (see section 1.3) is now flawed, since the main case is no longer independent.

UML 1.1 also supported Jacobson's view of «uses» as a generalisation relationship that was subject to interleaving. The wording of the official UML 1.1 definition [22] referred to a *used* (*cf* abstract) case and a *using* (*cf* concrete) case whose elements could be interleaved, in the manner of interrupted co-routines.

2.2. Confusion and inconsistency in use of terms

During the popular panel discussion on use cases at OOPSLA '98 [11], 80% of those delegates present indicated that they did not understand the difference between the «uses» and «extends» relationships. The panel attempted to explain the meaning of these relationships. While Ivar Jacobson upheld the view that «uses» meant a *kind of generalisation*, Alistair Cockburn asserted (incorrectly, according to section 2.1) that it meant *functional decomposition*. Bruce Anderson stated that he typically did not employ «extends» relationships, preferring instead to embed all branching inside the main use case (*cf*. the question posed in section 1.3). Statements like these show to what extent even the experts disagree about use cases.

The «extends» relationship was inconsistently defined from the start. On the one hand, the extension case was modelled as "inheriting from" the base case (*cf* the current usage of *extends* in Java), but simultaneously was regarded as "inserting behaviour into" the base case (*cf* the usage in Jacobson). 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. While «uses» could be construed as inheritance, the majority of developers disbelieved and disregarded this semantics, preferring to follow a subroutine interpretation, like the *using* client/supplier subcontracting relationship in the earlier Booch method [2].

2.3. Recent revisions in UML 1.2 and UML 1.3

After adoption by the OMG [21], UML subsequently went through two revisions. In UML 1.3, the old «uses» and «extends» relationships have quietly been replaced by new relationships called «include» and «extend». The most immediately noticeable change is that these new relationships are defined as *stereotypes of dependency* [3, p227-8]. This is also indicated in the style of arrow used, illustrated in figure 3. Where UML 1.1 adopted the *generalisation* arrow for use-case relationships, UML 1.3 now employs the *dependency* arrow. A *dependency* is defined abstractly as: any relationship between a dependent entity and a master entity, such that changes to the master would have consequential effects on the dependant [3, p63]. Thus, extensions depend on the main case they extend (which may stand alone), and main cases depend on the cases they include (wholes depend on their parts). This formalises Jacobson's faulty notion of dependency (see section 1.3), which is challenged below in section 3.4.

UML 1.3 reserves *generalisation* to mean a separate relationship between use cases. Figure 3 illustrates two specialisations of the *Validate User* case, *Check Password* and *Retinal Scan*. Here, the source of the generalisation arrow now has exactly the sense of a subclass [3, p226], reifying the abstract behaviour of a super use case, standing properly for the combination of base and extra behaviour as we insisted in section 2.2. As a result of this change, it is possible to view «extend» simply as an *insertion of behaviour* (p228), avoiding the earlier logical inconsistency.

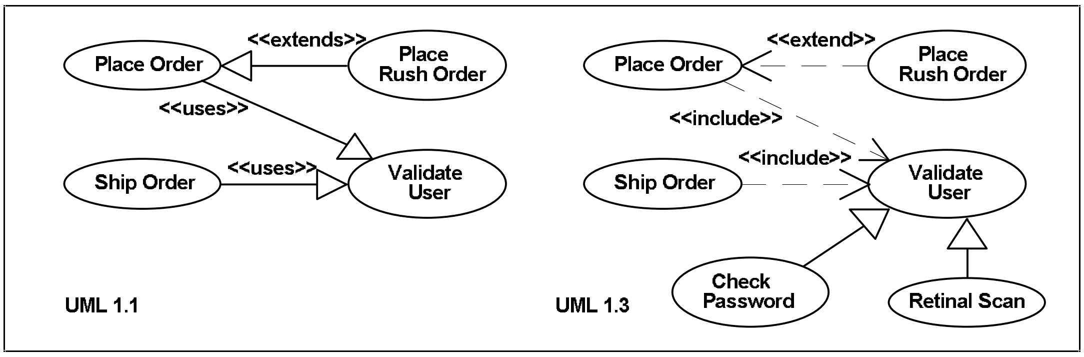

Figure 3: Use case relationships in UML 1.1 contrasted with UML 1.3

By abandoning the old view that «uses» specialises an abstract base case, it would appear that UML is bowing to popular pressure for a straightforward *compositional* semantics for «include», as recommended by OPEN [10] and as practised already by [5, 6] in disregard of the UML 1.1 semantics. All of the examples given in [3, 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 [17, p170-3]. What we would like to see is a plain assertion that *included* cases are atomic subroutines that are executed as a whole.

3. Serious outstanding problems with use cases

Although use cases are supposed to be independent of any formal design, such that they “cannot be forward or reverse engineered” [3, p239], the conceptual structures fostered by use case development mislead developers about design [26]. There are problems with the model logic (inadequacy) and problems in the way in which use cases encourage the wrong kinds of conceptualisation (cognitive misdirection), leading to missed logical dependencies in systems analysis.

3.1. Arbitrary *goto* and *comefrom* jumps in the flow of control

The underlying flow of control in use cases may be illustrated using a flowgraph style [1], to show where the focus of control is transferred internally within use cases. Figure 4 shows two examples from UML 1.1 and 1.3. The interleaving behaviour of «uses» in UML 1.1 is controlled from the using case, so this corresponds to repeated *goto* and *comefrom* instructions where the using case seizes back control. So far, we cannot guarantee that «include» is any different in UML 1.3; although there is an opportunity to adopt a more straightforward subroutine call semantics.

In both UML 1.1 and 1.3, an extension seizes control at the point where its guard is tested and ultimately returns control to this point, whether or not the extension eventually executes. This is shown again as *comefrom* and *goto* instructions in figure 4.

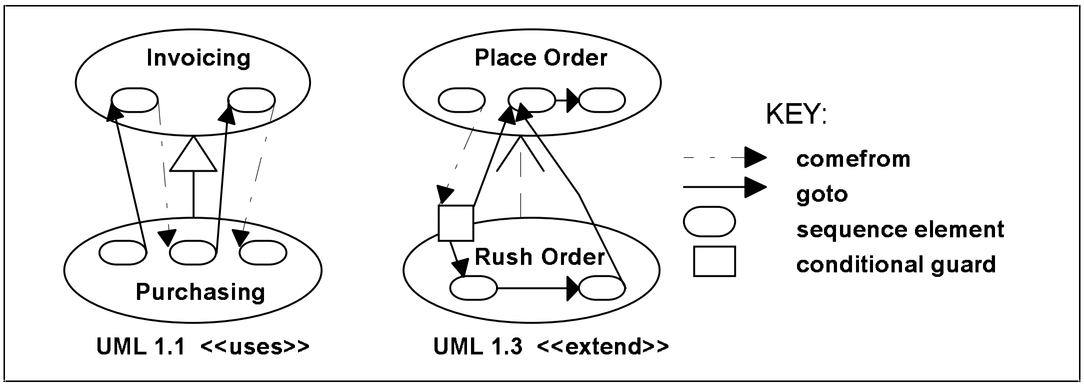

Figure 4: Control flow semantics in UML 1.1 and 1.3

These extraordinary flows of control speak for themselves. You may compare the kind of jumping out of blocks and breaking into blocks required by the UML use case model with the kinds of programming style that were judged "harmful" at the beginning of the block-structured programming era [8]. Use cases conceal an extraordinary complexity in the flow of control, which programmers must completely deconstruct, if they are to avoid disastrous logical program structures.

3.2. Insertion semantics inadequate to model exceptions and alternatives

As guarded blocks [3, p228], extensions characterise *optional* branches. However, developers commonly use «extend» to indicate *exceptions*. Unfortunately, the insertion semantics of «extend» does not support exceptions [26]. When an exception is raised, control *never* returns to the point of call, but may return to the *end* of the failed transaction after the exception has been processed, or not at all.

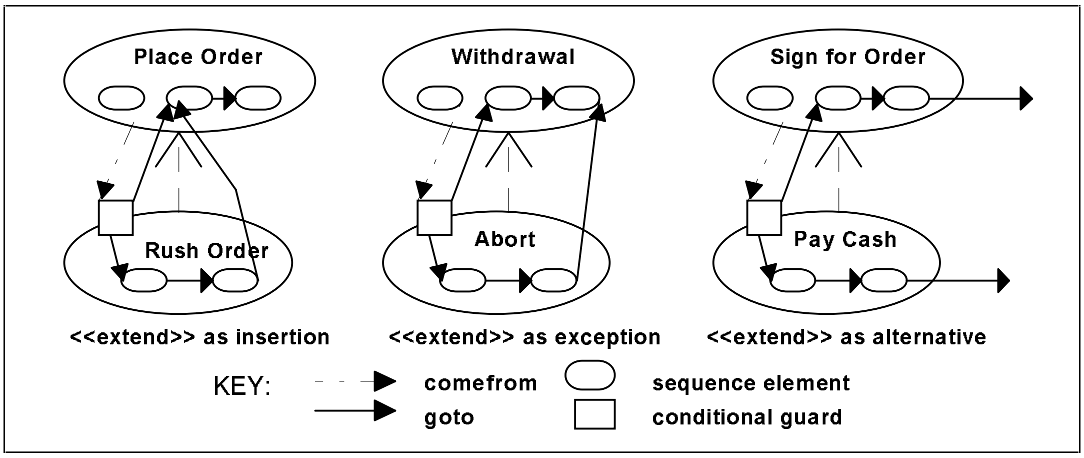

Figure 5: Control flow for insertions, exceptions and alternatives

Likewise, developers commonly use «extend» to indicate *alternative history*. Figure 6 below shows a typical example (adapted from Cockburn) 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 eventually execute to completion [26].

Jacobson originally expected the «extend» relationship to be able to characterise insertions, exceptions and alternatives [17, 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.

To model these control variants properly would require three *different* definitions of «extend» altogether, illustrated as flowgraphs in figure 5. Here, the flow of control is shown as: returning to the point of call (insertion); aborting to the end of the transaction (exception); and transferring out to a parallel timeline (alternative).

3.3. The missed long-range logical dependencies in use cases

Use case modelling promotes a highly localised perspective which obscures the true business logic of a system. As a result, developers miss important long-range dependencies. In figure 6, the extensions represent local alternatives (notwithstanding UML's failure here - see section 3.2). 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 the faulty goods. Even simple examples exhibit unpleasant mutual interactions between extensions and base cases. Most alarming is the fact that the correct form of redress for the cash-paying customer is not captured at all - this logical loophole is completely obscured in the use case model.

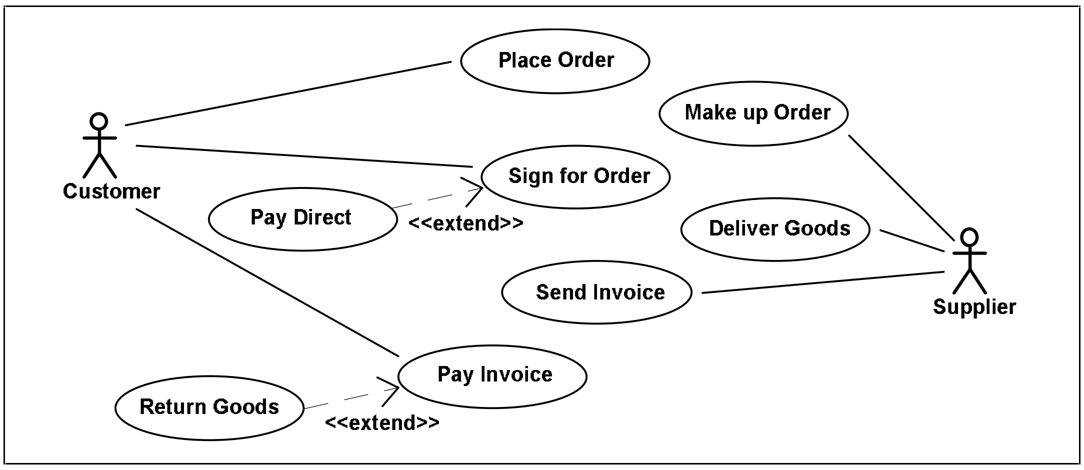

Figure 6: Use case model concealing undetected logical dependencies

In contrast to this, the corresponding structure chart of figure 7 introduces certain intermediate abstract nodes, *Credit Purchase* and *Direct Purchase*, to encode the long-range dependencies which exist beneath them in the chart. This pattern repeats for *Inspect Goods Unpaid* and *Inspect Goods Prepaid*, which encode the different options for redress. None of these nodes would qualify currently as use cases, since they have a larger than accepted granularity (see section 1.1) and may in general be quite abstract in nature.

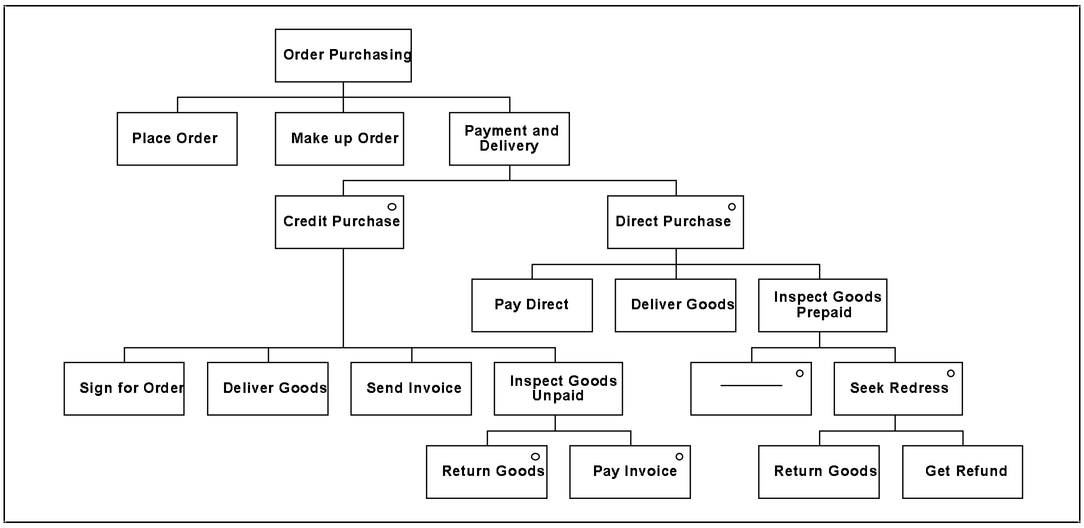

Figure 10: Making non-local dependencies explicit using a structure chart

So, there is a serious conceptual problem with use cases, which is that they foster a kind of tunnel vision in analysts. By contrast, the structure chart, with its simple rules of *sequence, selection* and *iteration*, captures the correct business logic. In this respect, a structure chart imposes a different cognitive bias from use cases.

3.4. The solution: fix the faulty notion of dependency

Most of these outstanding faults with the UML use case model relate to a non-logical formulation of use case dependency (see section 1.3). The direction of dependency is misleading for «extend», being 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 main and all extension cases, which depends on all of its parts, since the outcome of a *selection* can be any one selected branch. 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» might be construed as a genuine logical dependency, corresponding to the dependency of a *sequence* node on its subroutine calls, «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.

The fundamentally muddled thinking here goes right back to Jacobson's informal intuitions about dependency (section 1.3), for which the rationale was the beguiling fiction (see section 2.1) that alternatives do not disturb the main flow. To handle true alternative histories, UML must admit genuine selection nodes as use cases, whose sole purpose is to dispatch to the alternatives, reversing the direction of dependency in line with logic (*cf* the structure chart in section 3.3) and suspending the use case granularity requirement (section 1.1). To do this, UML would have to concede that a selection node, even if vacuous by itself, accomplishes something of "observable benefit" to the user, meaning at least one of its parts.

Apart from this, the «include» relationship should be acknowledged explicitly as a straightforward compositional relationship. Task decomposition is practised already in SOMA [13] and in the *Discovery Method* [25]. Drawing on the policies adopted in SOMA and *Discovery*, all use case and task relationships have always been defined as *sequence* or *selection* compositions in the OPEN standard [10, 16].

4. References

[1] K. G. van den Berg, *Software Measurement and Functional Programming*, PhD Thesis, University of Twente, Netherlands, 1995.

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

[3] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, Reading MA, 1999.

[4] P. Coad and E. Yourdon, *Object-Oriented Analysis, 2nd edn.,* Prentice-Hall, 1991.

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

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

[7] O. J. Dahl, B. Myrhaug and K. Nygaard, *SIMULA 67 Common Base Language*, Norwegian Computing Centre, Oslo, 1968.

[8] E. W. Dijkstra, "Goto statement considered harmful", *Comm. ACM, 11 (3),* 1968, 147-8.

[9] M. Dodani, "Semantically rich object-oriented software engineering methodologies", *Report on Object Analysis and Design, 1(1),* 1994, 17-21.

[10] D. Firesmith, B. Henderson-Sellers and I. Graham, T*he OPEN Modelling Language (OML) Reference Manual,* SIGS Publications, 1997.

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

[12] E. Gibson, "Objects born and bred", *BYTE, 15(10),* 1990, 245-254.

[13] I. Graham, *Migrating to Object Technology,* Addison-Wesley, Wokingham UK, 1995.

[14] I. Graham, "Task scripts, use cases and scenarios in object-oriented analysis", *Object-Oriented Systems, 3(3),* 1996, 123-142.

[15] I. Graham, "Some problems with use cases ... and how to avoid them", *Proc. Object-Oriented Information Systems '96,* eds. D. Patel, Y. Sun and S. Patel, Springer-Verlag, London, 1997, 18-27.

[16] B. Henderson-Sellers, A. J. H. Simons and H. Younessi, *The OPEN Toolbox of Techniques,* Addison Wesley, Wokingham UK, 1998.

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

[18] I. Jacobson, M. Ericsson and A. Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology,* Addison-Wesley/ACM Press, Reading MA, 1995.

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

[20] O. L. Madsen., B. Møller-Pedersen and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language,* Addison-Wesley and ACM Press, Reading MA, 1993.

[21] Object Management Group, "UML Semantics, Version 1.1" and "UML Notation, Version 1.1", *OMG documents ad/97-08-4 and ad/97-08-5,* 15 September, 1997.

[22] Rational Software Corporation, "UML Version 1.1", *http://www.rational.com/uml/,* September, 1997.

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

[24] R. C. Schank and R. P. Abelson, *Scripts, Plans, Goals and Understanding,* Lawrence Erlbaum Associates, New York NY, 1977.

[25] A. J. H. Simons, *Object Discovery, Tutorial 14, 12th European Conf. Obj.-Oriented Prog.,* AITO/ACM, Brussels, 1998.

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