The OPEN Software Engineering Process Architecture: From Activities to Techniques.

ARTICLE *in* AUSTRALIAN COMPUTER JOURNAL · JANUARY 2000

Source: DBLP

2 AUTHORS, INCLUDING:

Brian Henderson-Sellers
University of Technology Sydney
395 PUBLICATIONS **4,729** CITATIONS

The OPEN Software Engineering Process Architecture: From Activities to Techniques

B. Henderson-Sellers

School of Computing Sciences, University of Technology, Sydney
Broadway, NSW, Australia

A.J.H. Simons

Department of Computer Science, University of Sheffield, Sheffield, Yorks, UK

The 1997 OPEN process metamodel was the first fully documented software engineering process architecture for object-oriented projects, predating the Catalysis method, Select Perspective and the still emerging Rational Unified Process by a number of years. The OPEN process metamodel is based on a three-tier architecture, in which process Activities are broken down into a number of distinct Tasks; and each Task may be achieved through the application of a number of approved Techniques. This paper describes the relationships between the three layers of the OPEN process metamodel and shows how OPEN's Techniques contribute to a particular tailored process. As an exemplar, we describe Techniques relevant to late design and coding.

Keywords: process, software lifecycles, object-oriented methods, object-oriented techniques

1. INTRODUCTION

Software engineering requires the underpinning of a flexible and reliable, process-focussed methodology. A process tells you how to do things and how to manage and monitor the software development. In most of the second generation object-oriented (OO) software development methods (i.e. those published around 1994), there was, with a couple of exceptions OOSE (Jacobson *et al*, 1992 and MOSES (Henderson-Sellers and Edwards, 1994), little substantive support for process to the degree required by professional software engineers. Methods such as Coad and Yourdon (1991) and OMT (Rumbaugh *et al*, 1991) concentrated mainly on developing sets of diagrams rather than identifying the underpinning for their development in terms of project management issues such as timeboxing and version control. Booch (1991) focussed on notation that was applied by intuition, using a "round trip gestalt design" approach and, later (Booch, 1994), with macro- and micro-lifecycles. Some sequencing is implicit in OMT, although the general flavour is still that of a waterfall design approach. More recently, it has been shown (Simons and Graham, 1999) that an over-emphasis on design diagrams can give rise to cognitive misdirection.

From these two early, process-focussed methods (MOSES and OOSE) have grown two third generation OO methods: OPEN published in 1997 (Graham *et al*, 1997) and, more recently,

Manuscript received: August 1999
Associate Editor: Graham Low

Jacobson *et al,* (1999) Unified Process which underpins the commercial RUP (Rational Unified Process) product. In this paper, we focus on the older, more established OPEN methodological framework and explain how this framework is used for software development environments tailored to specific industry domains, individual organizations and indeed distinct projects.

OPEN stands for Object-oriented Process, Environment and Notation. OPEN is the first full lifecycle object-oriented methodology to address all the process issues. Its main focus is in providing an architectural *framework,* which is evident in its process metamodel (Figure 1), which can then be tailored by the user, thus creating a useful and usable software engineering process (or SEP). Although each individually created/tailored OPEN SEP is different in its detail, each conforms to the overall OPEN metalevel architecture (Henderson-Sellers, 1999a). OPEN thus provides a standard method architecture[1] – a common process language across the OPEN community of worldwide users.

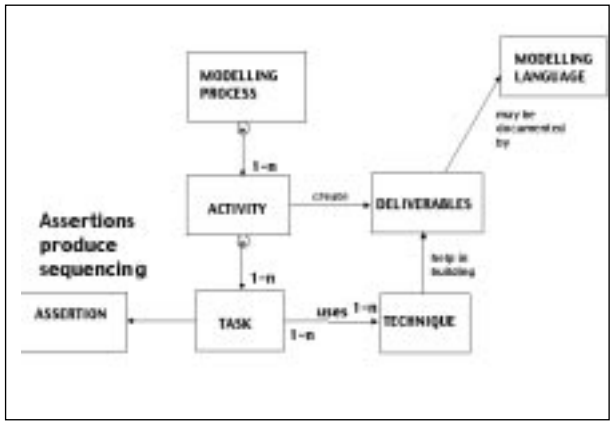

Figure 1: The OPEN process consists of Activities which in turn consist of Tasks. Tasks use Techniques for their realization (Graham *et al,* 1997).

OPEN has an underpinning lifecycle model based on the contract-driven model of Graham (1995b) which has been objectified (Section 2). In this model, process lifecycle objects, called Activities, monitor the progress of development which is accomplished through the carrying out of Tasks associated with the Activity. A Task is the smallest objectified unit of work which is managed directly in the OPEN process. The tools which a developer deploys to accomplish a Task are known as Techniques. There are many Techniques, both traditional and novel, available in the OPEN Toolbox (Henderson-Sellers *et al,* 1998). Previously, we have elaborated on the relationships between Activities and Tasks (Henderson-Sellers *et al,* 1997; Graham *et al,* 1997). In this paper, we reiterate the reasoning behind the whole three-level process architecture, but focus more on the relationships between Tasks and Techniques. To inform the reader about the spread and coverage of process-oriented Techniques now available in the OPEN Toolbox, we give an overview in Section 3. We go on to show in Section 4 how particular sets of compatible Techniques can be selected and grouped to fulfil the Tasks associated with model building and implementation (including "Code", "Construct the object model", "Design user interface", "Optimize the design"). In so doing, we

[1] Interestingly, it is at this metalevel that the OMG Software Process Engineering group are likely to issue their first RFP in late 1999.

illustrate the integrative capability of the OPEN metalevel process architecture for both old and new Techniques.

1.1 The need for process

Some software development seems to occur in a very ad hoc fashion. The majority of industries would appear to work at CMM level 1 (see review in Henderson-Sellers, 1996, Chapter 8). In these situations, when successes occur, the underlying reason is not obvious and there is no means to identify how to repeat the success. Conversely, when failures occur (as they inevitably will), in an ad hoc, CMM level 1, development shop, there is no way of identifying how to fix the process and learn from the failure and avoid a repeat failure in the future. A process of any sort lays down some guidelines to help developers set their own (personal and team) standards that they can follow. It is then possible for other personnel to temporarily or even permanently take over a role and for managers to control, monitor and evaluate how well the development is progressing towards completion. A process thus identifies activities that need to be done, recommends means by which to achieve these goals and, most importantly, creates a sequence which allows temporal planning. Indeed, the adoption of processes are commensurate with mature or maturing software development groups of CMM level 2 or probably 3 (or above).

1.2 An object-oriented process

Many of the traditional techniques for process and project management may be applied in an object-oriented project. However, there is a universal recognition by OO developers, consultants and mentors that three properties distinguish the best OO methods apart from traditional linear and single-pass lifecycles. Object-oriented development naturally lends itself to, and is most successful when coupled with, a process lifecycle that is:

(i) iterative – allowing revisions to rework existing deliverables;
(ii) incremental – producing a steady stream of deliverables in stages; and
(iii) parallel – working towards multiple modular deliverables simultaneously.

An iterative lifecycle is one in which its various development stages may be revisited (fully or partially) in cyclic order. Iterative development should not be used as an excuse for undisciplined design modification and code hacking; rather it should be properly tracked and the impact of changes managed appropriately.

Incremental delivery is linked with the iterative approach to some degree in that an OO development should deliver products to the users incrementally, usually at the end of each iteration, possibly every few weeks. Incremental delivery keeps the customer in the loop, ensuring that they always have in their possession a delivered and running version which contains progressively more of the required functionality. Distinction may be made between an evolving prototype and a production delivery – although this distinction is becoming increasingly blurred. Nevertheless, the availability of incrementally delivered software every few weeks or few months facilitates rapid feedback from users who, in a traditional waterfall development, might not have been able to do so for several years.

Finally, OO supports a parallel lifecycle in that the full software system awaiting development can be easily broken down into packages or subsystems. Because of the high degree of modularity supportable in an OO development, it is relatively easy to ensure that these several packages can be developed essentially independently of each other.

Using OPEN on a project requires access not only to the process specification (Graham *et al*, 1997) but also to a modelling language and a set of "how to" techniques (Henderson-Sellers *et al*,

1998). In this paper, we first summarize the basic architectural elements of OPEN and its contract-driven lifecycle (the process element) before exploring in more depth the Techniques which are required as an integral part of the tailorable methodological framework of OPEN.

2. THE OPEN PROCESS-FOCUSSED DEVELOPMENT METHOD

The OPEN methodological framework is described in terms of interacting objects, each of which represents an Activity (Graham *et al*, 1997). These Activity objects have responsibilities and associated contracts such that they can be configured in a variety of ways contingent upon the basic rule that the pre-condition of each Activity object must be met before this new Activity can commence. Secondly, before you leave the Activity to transfer to yet another one, you must meet its post-condition, which includes testing criteria. With its emphasis on contracts, this underpinning lifecycle model is thus known as the contract-driven lifecycle (Graham, 1995b).

Activities consist of Tasks (Figure 1) which are like Activities in the sense that they describe things that have to be done (but not how to do them). Activities are heterogeneous collections of goals whereas Tasks represent the smallest unit of work (that can be project-managed) that results in a Deliverable and which can be readily evaluated for completeness.

The Activities and Tasks, discussed in detail in Graham *et al*, (1997), describe what is to be done. They do not suggest any ways of accomplishing these goals. That is the role of the OPEN Techniques (Figure 1). The OPEN methodological framework provides this underpinning skeleton into which can be slotted, synergistically, selected and compatible Tasks and Techniques from those available. Many of these OPEN Techniques are in fact very well-known, others less so. While OPEN supports these well-established techniques, it also documents others ignored or poorly represented in the OO literature – particularly those associated with project management, business decision making, requirements engineering, metrics and usability. We discuss some of these briefly in Section 3 of this paper. Finally, although early OOAD methods tended to eschew coding concerns, OPEN provides some interesting support, as discussed here in Section 4 – as our main illustrative example of how OPEN provides a "scaffolding" for the successful integration and tailoring of both existing and new OO techniques.

OPEN specifies a number of deliverables which can be documented (Figure 1) using any one of a number of modelling languages (a modelling language is defined to be a metamodel plus a notation). One that is well-known is the Unified Modeling Language (UML) which was endorsed by the Object Management Group in late 1997. A second is the OPEN Modelling Language (OML) (Firesmith *et al*, 1997) which has many elements of the OMG's UML but provides additional benefits (Henderson-Sellers, 1998; Henderson-Sellers *et al*, 1999b). OML is more compatible with the OPEN process since both have a strong responsibility-driven focus emphasizing the identification of both objects and process activities according to their behaviours; and specify both of these using a similar contracting metaphor.

2.1 Tailoring OPEN

There are currently 30 Tasks (Table 1) identified within the OPEN framework, of which ten have subtasks. Each Task represents a single managed unit of work which is relevant to the completion of one or more process Activities. Most frequently, the goal of an Activity is achieved through the completion of several Tasks. For example, the Build Activity is accomplished by selecting Task units such as "Construct the object model", "Design user interface", "Map roles on to classes" and "Code". In general, though, the relationship between OPEN Tasks and Activities is many-to-many.

Analyze user requirements	Maintain trace between requirements and design
Code	Manage library of reusable components
Construct the object model	Map logical database schema
Create and/or identify reusable components ("for reuse")	Map roles on to classes
Deliver product to customer	Model and re-engineer business process(es)
Design and implement physical database	Obtain business approval
Design user interface	Optimise reuse ("with reuse")
Develop and implement resource allocation plan	Optimise the design
Develop business object model (BOM)	Test
Develop software development context plans and strategies	Undertake architectural design
Evaluate quality	Undertake feasibility study
Identify CIRTs	Undertake in-process review
Identify context	Undertake post-implementation review
Identify source(s) of requirements	Undertake usability design
Identify user requirements	Write manual(s) and prepare other documentation

Table 1: OPEN Tasks in alphabetical order.

For example, the Task: "Write manual(s) and prepare other documentation" is applicable to just about every Activity. Naturally, every Task will be highly relevant to some Activities and irrelevant to others. This leads to the OPEN idea of Activity-Task linkages.

While some Activity-Task pairs can be identified as of zero value and thus labelled as forbidden, other pairs may be identified as being mandatory. For example, in undertaking Project planning, one must use the Task: "Develop software development context plans and strategies". However, there are many other pairings that are more fuzzy. For instance, sometimes you might want to face the Task: "Evaluating quality" during the Build Activity and in other cases you might prefer to defer it to User Review or Evaluation. The link has a different possibility value than either a certainty or a zero chance.

Based on this observation of a "fuzzy" link between (at least some) pairs of Activity-Task, the notion of a possibility (or deontic) matrix can be introduced to formalize these two-dimensional pairings. Figure 2 exemplifies this approach. The values in this matrix, for all possible pairs, are categorized into one of five categories:

- Mandatory (M)
- Recommended (R)
- Optional (O)
- Discouraged (D)
- Forbidden (F)

The actual values in this matrix will vary depending upon a number of things - such as project size, organizational culture, domain of the application to be developed, skills and likes/dislikes of the development team. Indeed, at the smaller granularity of application, such as a single project, most of these values will be either M or F. This tailoring is in fact the aim of one of the subtasks of Task: "Develop software development context plans and strategies", a subtask called "Tailor the lifecycle process".

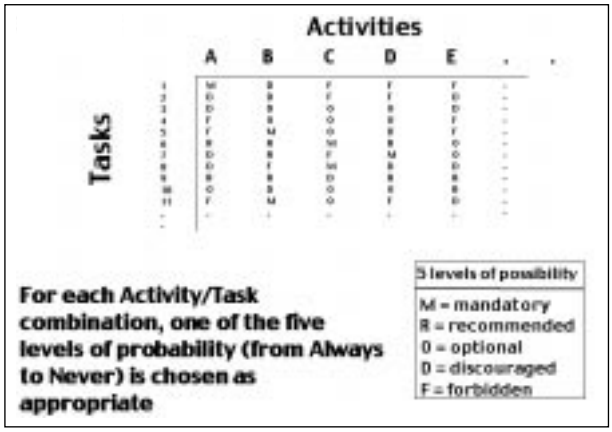

Figure 2: The lifecycle process consists of several Activities.

We noted above that OPEN establishes a linkage between Activities and Tasks; and that this is a many-to-many mapping. There is a second linkage between OPEN Tasks and the Techniques which are deployed to accomplish them. Whereas a Task is a unit of managed work, a Technique is a concrete method or approach that is selected to accomplish the Task. OPEN brings together all the tried-and-tested (and some new) OO techniques that have been used worldwide for the past decades. Each Technique is a tool to be applied by the users of the OPEN process, selected according to its fitness of purpose in completing a Task.

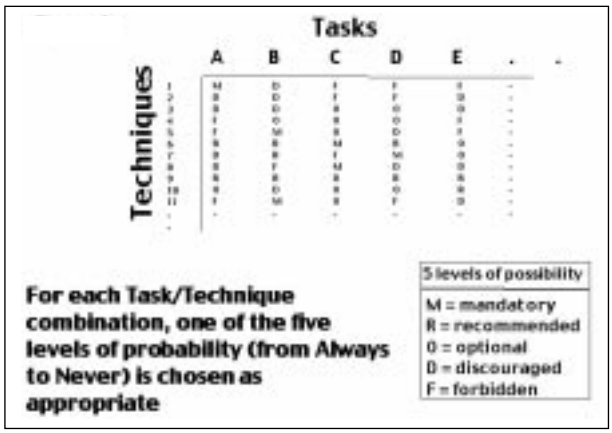

Figure 3: A core element of OPEN is a two-dimensional relationship between tasks and techniques. Each task may require one or several techniques in order to accomplish the stated goal of the task; and techniques may be applicable to several tasks. For each combination of task and technique, an assessment can be made of the likelihood of the occurrence of that combination. Some combinations can be identified as mandatory (M), others as recommended (R), some as being optional (O), some are discouraged (D) but may be used with care and other combinations that are strictly verboten (F = forbidden). (Adapted from Graham *et al*, 1997)

Task	Activity 1	2	3	4	5	6
Code		x				
Construct the object model		x				x
Develop and implement resource allocation plan						
develop iteration plan	x					
develop timebox plan	x					
set up metrics collection programme	x					
specify quality goals	x					
Evaluate quality			x	x		x
Identify CRTs		x				
Map roles on to classes		x (OOP)				
Test			x	x	x	
Write manuals and other documentation			x	x	x	x

Key:
1. Project planning
2. Modelling and implementation: OOA/D/P
3. V&V
4. User review
5. Consolidation
6. Evaluation

Figure 4: Binary values in the Activity-Task matrix appropriate for a small project using a responsibility-driven approach within OPEN and for which the requirements are prespecified.

The link between Technique and Task follows the same argument as for the links between Task and Activity. OPEN represents these links by a many-to-many fuzzy relationship represented again as a two dimensional matrix (Figure 3). Again, this matrix must be tailored to your requirements. Indeed, this matrix is much more likely to reflect the individualistic needs of your developers and your project than the first matrix of Figure 2. This is because there are, in fact, many "duplicates" in OPEN's toolbox. For example, there are several techniques for finding objects. Some OO software developers start by a textual analysis, some use simulation, some use CRC cards and yet others prefer a use-case driven beginning to a software project. For example, if we had a small project with pre-defined requirements, a small team and short project timelines, we might identify the Activities and Tasks (and their connections) in Figure 4 as relevant. These would correspond to lifecycle Activity objects as shown in Figure 5.

Figure 4 has identified, for this small example, six activities and eight main tasks. Which Techniques might be useful to accomplish these Tasks? Figure 6 lists those that may be identified, based on a team's predisposition to the use of a responsibility-driven OPEN approach[2].

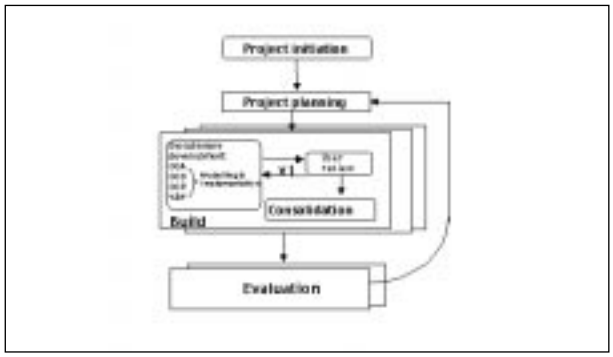

Figure 5 : Activity objects appropriate to small project and to matrix values of Figure 4.

[2] As opposed to a use-case-driven OPEN approach which is an alternative, feasible option.

Figure 6: Technique-Task linkages suggested for small exemplar project (Figures 4 and 5).

Thus a successful tailoring of OPEN results in the selection of appropriate Tasks and Techniques. This matrix tailoring is one of the strengths of OPEN which makes it suitable for a wide range of project types[3]. This is why OPEN can be called a methodological *framework* rather than a methodology.

3. THE SCOPE AND COVERAGE OF OPEN TECHNIQUES

In discussing the three layer OPEN process architecture, we wish to highlight in particular the little explored relationships between Tasks and Techniques. A fairly complete encyclopaedia of OO development techniques has recently been compiled by the OPEN Consortium (Henderson-Sellers *et al*, 1998). This work lists Techniques alphabetically in five catalogues (the appendices A-E). To give the reader some idea of the breadth of scope and coverage of that work, we provide a brief survey of some of the Techniques relating to some quite disparate Tasks, ranging from project management to interface design. For more detailed definitions and context of applicability, the reader should then refer to Henderson-Sellers *et al*, (1998). In Section 4, we then discuss in more detail, as an exemplar, the linkage between modelling and coding Tasks ("Identify CIRTs", "Construct the object model", "Map roles on to classes" and "Code") and the Techniques deployed to accomplish them.

3.1 Project Management and Quality Assurance Techniques

OPEN has a significant focus (and therefore a large number of techniques) on project management and quality (testing and metrics). Project management (PM) is, in many ways, an overlay to the

[3] Specific tailorings are explored in a recent paper by Henderson-Sellers et al, (1999a).

whole process of building software, addressing also many business (as opposed to technical) concerns. In the business context, before any software is even contemplated, there are a number of business decisions that have to be made. An assessment of the business problem and its likely feasible solutions (OPEN Task: "Undertake feasibility study") will use Techniques such as cost-benefit analysis, simulations, critical success factors, impact analysis, business process modelling and so on.

Once approval has been given for the construction of a software solution to the business domain problem, PM techniques begin to focus on planning techniques such as package identification and coordination together with traditional planning tools such as CPM and Gantt charts, perhaps as implemented in one of the shrink-wrapped project management software tools.

At the personal level, there are a number of Techniques that have been found to enhance personal productivity in certain cases. For many software developers, an awareness of their own work strategies for success has been engendered by the application of the PSP Technique, perhaps linked to SMART goals (McGibbon, 1995). This may also be enhanced if the organisation supports the goals of an organisational quality scheme such as TQM.

PM also extends to the deployment of the software at the customer site. Here techniques such as customer (on-site) training and standard cut-over strategies need to be planned and actioned.

Under the PM umbrella are also placed testing and metrics. In OPEN, testing is seen as an integral part of the Activity objects, being part of the post-condition. This means that not only software artefacts but also process elements are evaluated en route rather than *post facto*. Testing can, however, still be implemented at several levels: including unit (class) testing, sub-system/package testing, integration testing. It can be done internally (alpha tests) or externally (beta tests) or on-site (acceptance tests). Increasingly, it is important to include specific testing strategies to address usability (OPEN Technique: "Usability testing").

Object-oriented metrics (Henderson-Sellers, 1996) are fully supported in OPEN although we fully realise that the industry understanding of metrics and their utility is often limited. For example, one approach seeks to measure readily available surface properties of source code without there being any sound understanding of whether high or low scores in these metrics correspond to underlying quality (Chidamber and Kemerer, 1994). By contrast, the Goal-Question-Metric (GQM) approach identifies qualitative software properties that are desirable and then seeks to discover ways of measuring factors that contribute to these qualities (Offen and Jeffery, 1997; Henderson-Sellers, 1999b). Other quality-focussed techniques in OPEN include class naming standards, exception handling techniques, the use of formal methods and standards compliance.

3.2 Techniques Relating to User Requirements

The general area of requirements engineering is a nexus between social science and (computer) technology. It bridges between the user(s) and their requirements and the ability of the software engineer to construct a design that is both implementable in a language suitable for computation and also understandable (and therefore endorsable) by the end-user.

The first steps in requirements engineering are identification of the source or sources of the requirements (e.g. people, paper-based files, electronic databases) and then the elicitation of the particular requirements (for a particular project) from those sources. This is perhaps most difficult when the source is human since there is much sociological understanding necessary for (and often foreign to) the requirements engineer, who is often science/technology-focussed in their mindset. Many of the OPEN Techniques involve traditional people interaction/problem solving techniques

such as brainstorming sessions, active listening, interviewing and questionnaires. Indeed, there is a myriad of techniques that can be used to elicit requirements and it is certainly not the intention that you should use all of them on each particular project. Choose the one that suits the skills of the requirements engineer and the culture of the end-user. For many organizations, a well-structured approach will be appreciated; perhaps a highly structured questionnaire. For more creative environments or ones in which the requirements are perhaps not well understood by the potential users, roleplay, storyboarding and rich pictures may prove helpful.

When more than a single user is involved (as is usual), more extensive techniques are required. Here, RAD and JAD, and their associated workshops, have been found most beneficial. Whilst it is important to ensure that the adoption of RAD/JAD is NOT seen as a permissible reason/excuse to prototype rapidly ("hack"), such RAD workshops can bring together in a highly productive environment system analysts and users. Good sources of reference here are Graham (1995a; 1996b).

Following elicitation is a process of understanding, in conjunction with users, exactly what are the implications (in terms of software) for the satisfaction of those requirements. Here problem solving skills are to the fore. Techniques such as simulation to undertake what-if scenario evaluation and CRC cards (Beck and Cunningham, 1989) can be most useful.

Many of the techniques included in OPEN's requirements engineering group of Techniques are relatively new to OT although well-known in other domains of computing and information systems. What is needed is an interviewer/requirements engineer who has an OO mindset and can tailor questions and lines of enquiries that will permit the discovery of CIRTs[4] with the greatest facility. Discussions should focus on responsibilities, some scenarios and concepts in the business domain. It may be dangerous to focus on data particularly if the interviewees have been educated/trained in entity relationship modelling and since this can lead to the development of a data rather than an object model. Similarly, over-dependence on use cases (Jacobson *et al*, 1992) may lead to top-down functional decomposition (Firesmith, 1995; Korson, 1998).

3.3 Techniques for User Interface Design

Designing for usability (including user interface design) together with usability testing is critical to the development of high quality software. Before an OO method can be regarded as complete, it must include advice on these HCI issues, drawing on the wealth of established information from the HCI community. In OPEN, we choose to draw the developer's attention to sources of HCI advice rather than "reinvent the wheel" (Preece *et al*, 1994). It is clear that these issues are at the same time (a) important and (b) often ignored for expediency. OT is a technology that can bring benefits of quality to the software industry. A major component of quality, at least as far as the user is concerned, is the GUI. For a successful software industry, it is thus crucial to focus on the quality of the user interface. The goals of dialogue design were highlighted by Downes *et al*, (1991) and elaborated upon by Cox and Walker (1993) into a series of dialogue design steps involving visualization of the interface, abstraction into objects and their interactions, adding detail to the objects and their representation, prototyping the UI design and detailed program construction. A more detailed set of heuristics for good UI design, especially concerning the content of messages, is discussed by Graham (1995a). The constraints on human short-term memory (Miller, 1956) have for long been appreciated, leading to restrictions on the number of menu items and depth of menu trees. Issues concerning hierarchical versus random navigation must be addressed. The state-based modality of older user interfaces was one of the original targets of the Smalltalk project (Goldberg, 1985).

[4] CIRT = Class, Instance, Rôle or Type.

More recently, 4GL-style screen painting techniques have given way to interface building toolkits (Valaer and Babb, 1997), such as NeXT's Interface Builder, Visual Basic and the recent Java Beans technology, all of which allow the programmer to assemble visual prototypes of the UI and wire the visual controls to application objects. These tools alleviate much of the programming effort involved and help to impose a common look-and-feel; but they do not obviate the need for careful design. Usability testing is therefore an extremely important technique: some fourteen checks are specified in Graham (1995a).

3.4 Reuse Strategies and Supporting Techniques

Reuse is an integral part of OPEN. There are issues of technical concern and issues of management concern. It has been argued that many of the technical issues are solved and it is really the NIH[5] syndrome that prevents reuse.

CIRTs designed for a single project tend not to represent the totality of the concept they are modelling. The characteristics included will be those pertinent to the current project rather than to the broader domain. For reuse, classes (and other artefacts) need to be "complete", highly tested and robust (among other things). Techniques such as "Completion of abstractions" and "Refinement of inheritance hierarchies" are relevant here. These focus on, firstly, creating a full and complete specification of the concept and secondly on the likely modifications to the inheritance hierarchy to permit a more flexible representation of concepts in the problem domain. Addition of genericity (OPEN Technique: "Genericity specification") can also be useful here at the detailed design stage.

Reuse is not merely at the class level; there is currently much enthusiasm in engendering reuse via patterns, mechanisms, components and frameworks (Szyperski, 1998, Chapter 9). Indeed, OPEN describes a process for the initial identification, development and maturation of frameworks (Henderson-Sellers *et al*, 1998), evolving from the whitebox framework stage to the blackbox framework stage as described by Pree (1997).

Creation of reusable artefacts is one thing (OPEN Task: "Create and/or identify reusable components ("for reuse")"; finding existing components requires a totally different set of Techniques (OPEN Task: "Optimize reuse ("with reuse")" and Task: "Manage library of reusable components"). If developers cannot identify an appropriate artefact *rapidly,* then they will not reuse it but will redevelop it from scratch. Thus it is important that artefacts are not only well catalogued (OPEN Technique: "CIRT indexing") but that tools exist to locate these stored artefacts (Freeman and Henderson-Sellers, 1991).

On the PM side of reuse, we need to consider how the quality of reusable artefacts destined for storage in the company library can be assessed. Quality metrics (coupling, cohesion, complexity etc.) are available and new reuse metrics have been proposed (Henderson-Sellers *et al*, 1998). Another important element is a person responsible for the quality of the library. This important role necessitates both a reactive and a proactive response: accepting and checking artefacts proposed for inclusion in the library together with the active encouragement of the provision of such classes and encouragement to (re)use them in later projects. This approach is supported, in part, by OPEN Technique: Application Scavenging. It should be practised in the context of well-defined roles of the development team (Application Developer, Reuse Manager, Librarian, Application Scavenger) in order to avoid conflicts of interest.

The hardest part seems to be how to implement a "rewards" strategy that will encourage a reuse mindset. Many obvious such strategies are open to abuse. For instance, "royalty" payments to developers may be made on the basis of how often a class they have contributed is used by others

[5] NIH = Not Invented Here

or on the extent to which the classes in the current project have been extracted from the library (friends can easily be cajoled/entreated to use these "as a favour"). Some newer ideas are currently being investigated as a subproject of the OPEN project.

4. MODELLING TECHNIQUES FOR SEAMLESS ANALYSIS AND DESIGN

Modelling techniques form the largest grouping of OPEN Techniques. They also provide the most object-oriented flavour – many were newly invented within the object-oriented paradigm in comparison with PM or UI techniques, for instance, which have an obvious heredity outside of OT. In this section, we "weave together" a number of these modelling-focussed tools from the OPEN "toolbox" of techniques.

To illustrate how a wise choice may be made of a set of OPEN Tasks and Techniques, we will use the small (toy) example of Figure 4 and concentrate on issues regarding the OPEN Activity: "Modelling and Implementation: OOAD/D/P" (labelled 2 in Figure 4). For this Activity, the appropriate Tasks are seen to be

- Identify CIRTs
- Construct the object model
- Map roles on to classes
- Code

In the following subsections, we explore the implications for appropriate Techniques for these four Tasks. Some of the relevant Techniques (a subset extracted from Figure 6) to be discussed here are

Task: "Identify CIRTs"

- Textual analysis
- CRC card modelling
- Abstraction utilisation
- Responsibility identification
- Contract specification

Task: "Construct the object model"

- Service identification
- Relationship modelling
- Collaborations analysis
- Interaction modelling
- Generalisation and inheritance identification
- State modelling
- Scenario development

Task: "Map roles on to classes"

- Role modelling

Task: "Code"

- Class internal design
- Implementation of services
- Implementation of structure

4.1 CIRT identification

CIRT identification may rely on any one (or in fact more than one) of several techniques including textual analysis, task analysis and use case evaluation. CIRTs may be identified from state transition

diagrams or from CRC roleplay exercises. It is rare for anyone to be able to identify relevant concepts just by considering a static view of the domain. Our view of the world is interactive and dynamic; our design of the software system is also implicitly dynamic. CIRTs are only of value if they interact. Thus in identifying concepts it is almost inevitable that we concurrently think about how they interact (see Section 4.2.2).

One good way of identifying concepts is to select key nouns in the requirements specification (candidate CIRTs) and then to look for the responsibilities (Appendix A) held by each such CIRT. Responsibilities can be identified from requirements, elicited by techniques such as CRC or roleplays and can be categorized as either (i) responsibilities for knowing; (ii) responsibilities for doing; and (iii) responsibilities for enforcing. These are then implemented by one or more operations in the CIRT interface. Operations in the interface then link smoothly across to one or more (often only one) methods in the code.

The responsibility is a high-level statement of the service(s) to be provided by each CIRT. This must be supplemented by a set of rules which govern that service provision. This is known as "software contracting" (Meyer, 1988; 1992b). The objective is to clearly document the meaning of services and/or responsibilities. Contracts and responsibilities are thus not synonyms – although they are often confused (Rational, 1997). Contracting is so important that not only is it used in Modelling but it is also an integral part of the lifecycle process model.

Good OO design is also highly modular. CIRTs have restricted interfaces detailed by their responsibilities (see above) – data (attributes) and methods are detailed design or coding decisions. Each CIRT thus identifies, encapsulates and fully represents a single concept in the domain *at the current abstraction level* (Appendix B). As abstraction levels change during the OPEN process as more and more detail is added, more CIRTs will be identified, some CIRTs will be found to be aggregates and thus decomposable into their parts.

By focussing on the interface, as should all good application developers, we are also focussing on another key element of object technology: encapsulation/information hiding. Encapsulation is a boundary – it may be transparent, translucent or opaque. Information hiding requires an opaque encapsulation in which only those details of the coded class which are reflected in the externally available services (operations and (logical) attributes) or, at a higher level, external responsibilities can be seen from outside the class. These details, together with the class name, constitute the interface. They should be supplemented by responsibilities for enforcing linked to internal rulesets.

4.2 Constructing the Object Model

OPEN Task: "Identify CIRTs" focusses on autonomous concepts. To build an "object model" (a model describing classes, objects, use cases etc.), consideration needs to be given to how these CIRTs connect together, both statically and dynamically. In the early stages of building the object model, the emphasis will be on mappings/associations some of which may be usefully modelled as aggregations (composition structures) or containment. Later, as knowledge is gained regarding the static view of the object model and refined, generalisation/specialisation will be used increasingly to create inheritance structures. Also, usage structures are elaborated as inter-object communication patterns (the dynamic view) unfold. In this subsection, we explore a subset of the OPEN Techniques relevant to the Task: "Construct the object model".

4.2.1 Service identification

Once some of the candidate CIRTs have been identified, together with their high-level responsibilities, we need to transform these into the services offered by the CIRT. Services are the

properties (logical attributes) and operations seen in the interface of the class. Each responsibility may lead to the identification of one or more properties and operations, the latter being related to behaviour and the former to knowledge. Operations may also be found by examining other classes and seeing what operations they are likely to send out requests for – operations are services *offered*, not required.

4.2.2 Relationship modelling

In OOAD, relationships are usually established between classes or types although these relationships (except for generalisation/inheritance) in fact represent instance-instance links. OPEN makes the usual primary modelling distinction between simple-valued attributes and associations. However, OPEN prefers the majority of its associations to have the semantics of mappings, or one-way dependencies. Later during implementation, mappings may be encoded directly as pointer-valued attributes, or as access functions returning the desired object. Aggregations are mappings which have the semantics of whole-part relationships – although the definition is still not agreed (Henderson-Sellers and Barbier, 1999).

4.2.3 Collaboration and interactions

In OPEN, interaction analysis and collaborations analysis are two distinct techniques, although there is significant closeness in the two techniques. A collaboration is, in OPEN and OOram (Reenskaug *et al*, 1996), a sub-contracting relationship between two roles, eventually to be embodied in classes. It represents the client-supplier relationship between roles that supports one or more messages that the client delegates to the supplier in order to help the client fulfil its own responsibilities (see Section 4.2.1). Collaboration diagrams are used for documentation. Interactions, on the other hand, represent the enactment of collaborations (i.e. its dynamic counterpart). In UML, there are two types of interaction diagrams: collaboration diagrams and sequence diagrams which give different visual emphases on these interactions: the sequence diagram emphasizes the time-ordering of interactions and the collaboration diagram emphasises a network of static connections between instances that form a subsystem, along which messages may flow. In UML, a collaboration is therefore more like a network of static associations between instances, rather than a client-server relationship between types. Choice between using either UML diagram is often a matter of taste.

4.2.4 "Inheritance"

Conceptual generalisation is the most productive technique to apply during classification, since it leads to inheritance graphs that support both reuse of conceptual designs and polymorphism (dynamic type substitution). On the other hand, 'implementation inheritance' refers to the opportunistic reuse of code, considered irrespective of any considerations of subtyping or the relatedness of the two concepts. It should only be used in the context of optimisation of design/code since it mitigates strongly against reuse at the application developers' level.

Inheritance structures may also include multiple specialisations. Thisnetwork structure is harder to maintain and understand and multipleinheritance should only be used when necessary[6]. Indeed, we would go even further and suggest that inheritance itself (as single inheritance), whilst being a "trademark" of an OO system, should be used sparingly. There are many cases when it is the first, unthinking modelling option. In reality, reuse is better served in many of these cases by

[6] The analogy we like to use is that of a chainsaw – great for sawing down trees (the difficult job) but highly dangerous for more mundane tasks such as pencil sharpening.

delegation/subcontracting/collaborations. Some useful reminders of this necessary balance in using different techniques for different jobs are given in texts by Page-Jones (1995) and Riel (1996).

4.2.5 Dynamic Modelling

The dynamic side of OO modelling may be seen in user interactions with the system (use cases), in the state-transition descriptions of how a single class (or more strictly interfaces belonging to a single class) handles events or in the way that a small group of CIRTs may collaborate together to fulfil a single service request by delegation, subcontracting and inter-CIRT collaborations. Indeed, many such designs could be regarded as underpinned by patterns (Gamma *et al*, 1995), a topic of immense value to realise a component software industry.

While collaborations (Section 4.2.3) are usually documented in collaboration diagrams, the same kinds of diagram can also be used to document use cases, scenarios and task scripts (OPEN Technique: "Scenario development"). Although no unique definition of use cases exists (Cockburn, 1997), they can be understood as describing how a system functions from the (external) user's viewpoint.

Use cases (and their variants) have many uses. They may be useful in eliciting and describing user requirements; they may help in the identification of CIRTs or, conversely, be identified from the CIRTs and the semantic net and collaboration diagrams; they may be useful in directing the testing program.

Whilst many authors have embraced use cases as part of their methods, it must be remembered that

1) They show functionality. Decomposition could too easily lead, not to objects, but to detailed DFDs (Firesmith, 1995). Decomposition of task scripts is argued (e.g. Graham, 1995a; 1996a) to lead to an identifiable end-point in the *atomic task script*, thus providing a more reliable modelling technique.
2) There is no obvious way to "find the object" directly from a use case. Techniques for converting the information found in use cases to a set of interacting classes (as depicted, for instance, in a class diagram) are little different from those aimed at finding objects directly from the requirements documents. Whilst this is not surprising, care is required. Indeed, Korson (1998) recommends that you "do not derive your design directly from your use cases".
3) There are still many "dialects" (Cockburn, 1997) which foil inter-communication between development groups (Simons, 1999).

Some methodologists advocate that their methods should be totally use-case driven (Jacobson, 1996). We beg to differ. Use cases (and their alternatives) need to be an integral part of a method – their primary use in some situations may in fact be for driving the testing, rather than for eliciting requirements descriptions.

4.3 Roles

One frequent error is the use of inheritance to represent an apparent is-a-kind-of knowledge structure (specialisation inheritance) when a more appropriate modelling technique would be that of roles. The importance of roles as a complete modelling technique, distinct from the labels on associations in OMT which have the same name, has been recognised in the publication of the OOram method (Reenskaug *et al*, 1996) and in their incorporation into the UML and OML metamodels. Essentially, a role is a temporary object classification in the sense of an instance, say of type *PERSON*, temporarily adding an additional classification, say *EMPLOYEE* and *COMMUTER*. Consider an instance of type *PERSON: Margaret*. Between 8 and 9 a.m., she is a

commuter, between 9 and 5 an employee and between 5 and 5.45 a commuter again. Yet all the day she remains a person. Using subtypes to model this would be inappropriate since it would lead to an instance of subtype *COMMUTER* (the instance Margaret in fact) moving, at 9 o'clock, to become an instance of subtype *EMPLOYEE.* Such temporary migration between subclasses can be indicative of the need for role modelling.

Roles can thus be thought of as temporary reclassification. Perhaps more importantly, they are additional classifications rather than changes of classification - for instance between *EMPLOYEE* and *RETIREE.* Roles occur frequently in real life. Role modelling is a software technique of representing that reality in the software design.

If UML is used as the notation with OPEN, roles can be shown in collaboration diagrams directly or in class diagrams using the <<role>> stereotype (Appendix C). With OML as the notation, a separate and distinctive icon is used on both diagrams.

4.4 Coding Styles and Implementation Techniques

OOAD produces a system-level design in which classes, the attributes they manage, inter-class connections and the numbers and semantics of methods have been determined. This is large programming language independent; although it is recognized that the presence of particular programming language constructs may profitably influence the choice of design architecture. An example of this is shown in the C++ Standard Template Library (Stepanov and Lee, 1994), where considerable responsibility for copying, inserting and appending elements in containers is transferred to generic template functions acting on iterators, rather than supplied as methods of the containers themselves.

There are a number of OPEN techniques which give advice on coding issues. When coding, the focus of the development team switches from consideration of the interface (the "type" aspect of the CIRT) to the internal viewpoint (the class implementation). Operations in the interface are now implemented or realised by a method (member function in C++, for example) in the code. Firstly, it may be necessary to design the way that a particular method is to be constructed. Since C++, Eiffel and Java (and less so, Smalltalk) are essentially object-oriented *procedural* programming languages, then the way in which the code is structured internally to a method is indeed procedural in nature. Consequently, it is generally believed that designing such methods simply requires the application of traditional structured design techniques *but at a much smaller scale* than occurs in a traditional, structured programming development environment. Thus, one might anticipate seeing structures such as loops, if/then/else, case/switch statements and linear series of assignment statements. *However,* whilst conventional wisdom suggests that in C++ (for instance) such a piece of procedural code to implement a single method may be 20-30 lines long, more recent evidence (Haynes and Henderson-Sellers, 1997) shows that in *all* OOPLs, the typical method size in well-written OO code is 2-3 lines long only. In other words, good C++ style should be not procedural style but "Smalltalk style". If this observation is repeated and upheld, then the OPEN Technique: "Class internal design" will fade into insignificance.

Implementation (coding) of services (or characteristics) and structure is in terms of methods, properties (exceptions, links, parts and attributes) and assertions (pre-conditions, post-conditions and invariants). Some of these are truly hidden or private; while others cross the boundary and are in fact public operations (Wirfs-Brock and Wilkerson, 1989). It is important to note that the public section of a class should include only those methods which provide its interface – and this should be minimal, according to the purpose of the abstraction. In C++ and Java, there is a third category of visibility: protected. Whether to mark attributes as private or protected is a difficult decision.

There are two common coding practices:

(i) in systems that rely more on composition than inheritance, make all data attributes private;
(ii) in systems that rely heavily on inheritance, make all data attributes protected, so that descendants may easily access them.

In the first style, data are strongly encapsulated, so that only the declaring class may access them directly. This means that any descendant, although it inherits the data attribute, cannot access it. If the descendant needs access, it is common to define protected access methods in the original class. If systems make heavy use of inheritance, this can lead to code-bloat, with many internal access methods. In this case, it is more sensible to decide to make all data protected (the second style).

The notion of visibility relates both to what features are seen as part of the class interface i.e. what are external responsibilities or external services; and also the extent to which other classes are permitted to see, and hence have access rights to, this particular class. Objects are visible to each other when they are in the same scope i.e. within the same Namespace. Fusion (Coleman *et al*, 1994) contains a richer family of visibility relationships (reference lifetime, server visibility, server binding and reference mutability) which may also be useful.

Finally, it is sometimes the case that several public methods rely on a common algorithm – for example, insertion, removal and membership testing in a *SET* class may involve a common searching operation. This may be made into a private (or protected) internal method, reducing the size of the code overall. However, a review should be conducted of classes that acquire too many internal methods of this kind. Often this indicates a class hiding a functional abstraction, which should really be conceived differently, with the behaviour distributed over several objects.

Related to the same encapsulation theme are the notions of "friends" in C++ (Stroustrup, 1991) and selective exporting in Eiffel (Meyer, 1992a). C++ originally provided the "friend" mechanism to allow a class to break its own encapsulation selectively, for the sake of efficiency. A class may declare another class, a method, or a global function to be a friend, meaning that this program component is trusted and has privileged access rights to the internals of the declaring class. In principle, friend declarations should only be used as a last resort. However, certain language limitations in C++ have led to "friend" declarations being commonly used in three identifiable situations:

(i) To circumvent bounds-checks in operations involving two different class abstractions, e.g. multiplication of a Vector and a Matrix;
(ii) To extend the interface of library classes not available to the developer, e.g. overloading operator >> and operator <<, conceptually members of the iostream classes, to read and write new user-defined classes;
(iii) To enable automatic type conversion, by replacing binary methods, which normally block type conversion, with global friend functions, e.g. global overloads of operator +, operator -, which access inside Vector and Matrix.

Eiffel provides a flexible export mechanism which is the dual of friends – instead of breaking encapsulation, it offers selective interfaces to different clients. By default, every feature is public; however, **feature {NONE}** makes the following declarations secret (equivalent to protected), and **feature {CLASSA, CLASSB}** makes the following declarations accessible only to the named classes.

Stylistic guidelines for different languages are emerging. These are often referred to as the "idioms" of that particular language. Some more general guidelines for "good coding" are to be found in the Law of Demeter which is aimed at the preservation of encapsulation. It was formulated by Lieberherr *et al*, (1988) and later revised by Lieberherr and Holland (1989) and is a form of

voluntary design restriction, which prohibits the sending of messages to certain categories of objects from within methods. The need for such a law arises mostly in languages which pass objects by reference, such as Smalltalk, Eiffel or Java. In these languages, it is possible, by some combination of message expressions, for an object to obtain access to part of another object (a sub-object) and cause this to be altered, without the permission of the owning object. The intent of the law is to force all messages to sub-objects to pass through the interface of the owning object first, which delegates the request to the sub-object (OPEN Technique: "Delegation").

Other design/coding decisions include whether to store information or to recalculate it on each method invocation. As an example, consider the characteristic/property of a class which gives an enquirer the current balance of a bank account. The value of the *Balance* object to be returned to the client object could be stored as an attribute of the *BankAccount* object. Alternatively, what is stored may be a transaction history, probably since the last bank statement was issued. The current balance is thus calculated anew each time the *balance* operation/service is requested as *previous balance* + Σ *recent transactions.*

The use of a distributed system also forces certain coding decisions. For instance, using an ORB focusses attention closely on the interface and possibly on IDL considerations – inheritance is no longer as important since what is important is the interface and services provided by the CIRT in question. Design decisions regarding virtual and actual nodes need to be implemented, as does code to implement asynchronous or synchronous message passing sequences. It is likely that concurrent threads will be created and coded.

5. SUMMARY

In developing the third generation OO methodology OPEN, thoroughbred OO ideas have been embodied in both the lifecycle process and the tasks and techniques which are used to effect successful OO software development. Linkages between Activities and Tasks and between Tasks and Techniques are the focus of the OPEN Task: "Tailor the lifecycle process", in which the project manager (typically) creates a specific OO process for the current software development and/or as the organisation's own standard OO process. Techniques then provide the practical advice on how to build the software system and have been discussed here in clusters, with more detail on those focussing on model building and coding. Thus OPEN provides a comprehensive methodological framework (as outlined in Figure 1) for the application and utilisation of object technology in a true software engineering context.

6. ACKNOWLEDGEMENTS

This is contribution number 98/7 of the Centre for Object Technology Applications and Research.

REFERENCES

BECK, K., and CUNNINGHAM, W. (1989): A laboratory for teaching object-oriented thinking, *SIGPLAN* Notices, 24 (10), 1-6.

BERNER, S., BLINZ, M. and JOOS, S. (1999): A classification of stereotypes for object-oriented modeling languages, in *Procs. <<UML>>'99,* Springer-Verlag (in press).

BOOCH, G. (1991): *Object Oriented Design with Applications,* Menlo Park, CA, USA: Benjamin/Cummings.

BOOCH, G. (1994): *Object-Oriented Analysis and Design with Applications,* (2nd edition), Menlo Park, CA, USA: Benjamin/Cummings.

BUDD, T. (1991): *An Introduction to Object-Oriented Programming,* Wokingham, UK: Addison-Wesley.

CHIDAMBER, S. and KEMERER, C. (1994): A metrics suite for object oriented design, *IEEE Trans. Software Eng.,* 20(6), 476-493.

COAD, P., and YOURDON, E. (1991): *Object-Oriented Analysis* (2nd edition), New York, USA: Yourdon Press/Prentice Hall.

COCKBURN, A. (1997): Goals and use cases, *JOOP,* 10(5): 35-40.

COLEMAN, D., ARNOLD, P., BODOFF, S., DOLLIN, C., GILCHRIST, H., HAYES, F. and JEREMAES, P. (1994): *Object-Oriented Development: the Fusion Method*, Englewood Cliffs, NJ: Yourdon Press/Prentice Hall.

COX, K. and WALKER, D. (1993): *User Interface Design*, 2nd edition, Prentice Hall.

DOWNES, E., CLARE, P. and COE, I. (1991): *Structured Systems Analysis and Design Methodology: Application and Context*, (2nd edition), Prentice Hall.

FIRESMITH, D.G. (1995): Use cases: the pros and cons, Report on Object Analysis and Design, 2(2), 2-6.

FIRESMITH, D., HENDERSON-SELLERS, B. and GRAHAM, I. (1997): *OPEN Modeling Language (OML) Reference Manual*, New York, USA, 271, SIGS Books.

FREEMAN, C. and HENDERSON-SELLERS, B. (1991): OLMS: the Object Library Management System, POTTER, J., TOKORO M. and MEYER B. (ed.), in *TOOLS 6*, Prentice Hall, 175-180.

GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES J. (1995): *Design Patterns. Elements of Reusable Object Oriented Software*, Reading, MA, USA: 395, Addison-Wesley.

GOLDBERG, A. (1985): *Smalltalk 80: the Interactive Programming Environment*, Addison-Wesley.

GRAHAM, I.M. (1995a): *Migrating to Object Technology*, Wokingham: Addison-Wesley.

GRAHAM, I.M. (1995b): A non-procedural process model for object-oriented software development, *Report on Object Analysis and Design*, 1(5): 10-11.

GRAHAM, I. (1996a): Linking a system and its requirements, *Object Expert*, 1(3): 62-64.

GRAHAM, I. (1996b): The organization of workshops, *Object Expert*, 1(6): 52-54.

GRAHAM, I., HENDERSON-SELLERS, B. and YOUNESSI, H. (1997): *The OPEN Process Specification*, London, UK: Addison-Wesley, 314.

HAYNES, P. and HENDERSON-SELLERS, B. (1997): Bringing OO projects under quantitative control: an output, cash and time driven approach, *American Programmer*, 10(11): 23-31.

HENDERSON-SELLERS, B. (1996): *Object-Oriented Metrics. Measures of Complexity*, NJ, USA: Prentice Hall, 234.

HENDERSON-SELLERS, B. (1998): OML: proposals to enhance UML, Procs. <<UML>>'98. Beyond the Notation, Universite de Haute-Alsace, Mulhouse, France: 319-329.

HENDERSON-SELLERS, B. (1999a): A methodological metamodel of process, *JOOP/ROAD*, 11(9): 56-58, 63.

HENDERSON-SELLERS, B. (1999b): OO software process improvement with metrics, Keynote presentation at Procs. *METRICS99*, IEEE Computer Society Press: 2-8.

HENDERSON-SELLERS, B. and BARBIER, F. (1999): What is this thing called aggregation?, in *TOOLS29*, MITCHELL, R., WILLS, A.C., BOSCH J. and MEYER B. (eds.), IEEE Computer Society Press, 216-230.

HENDERSON-SELLERS, B. and EDWARDS, J.M. (1994): *BOOKTWO of Object-Oriented Knowledge: The Working Object*, Sydney: Prentice Hall, 616.

HENDERSON-SELLERS, B., GRAHAM, I.M., SWATMAN, P., WINDER, R. and REENSKAUG, T. (1997): Using object-oriented techniques to model the lifecycle for OO software development, Patel, D,. Sun, Y. and Patel, S. (eds.), *in Procs. OOIS '96*, London: Springer-Verlag, 211-220.

HENDERSON-SELLERS, B., SIMONS, A.J.H. and YOUNESSI, H. (1998): *The OPEN Toolbox of Techniques*, UK: Addison-Wesley, 426 + CD.

HENDERSON-SELLERS, B., FIRESMITH, D.G., GRAHAM, I. and SIMONS, A.J.H. (1999a): Instantiating the process metamodel, *JOOP (ROAD)*, 12(3): 51-57.

HENDERSON-SELLERS, B., ATKINSON, C. and FIRESMITH, D.G. (1999b): Viewing the OML as a variant of the UML, *Procs. <<UML>>'99*, Fort Collins, CO, October 1999, Springer LNCS 1723, 49-66.

JACOBSON, I. (1996): Public communication, ObjectExpo, Sydney.

JACOBSON, I., CHRISTERSON, M., JONSSON, P. and OVERGAARD, G. (1992): *Object-Oriented Software Engineering: A Use Case Driven Approach*, New York, NY, USA: 524, Addison-Wesley.

JACOBSON, I., BOOCH, G. and RUMBAUGH, J. (1999): *The Unified Software Development Process*, Reading, MA, USA: Addison-Wesley Longman Inc.

KORSON, T. (1998): The misuse of use cases (managing requirements), *Object Magazine*, 8(3): 18-20.

LIEBERHERR, K. J. and HOLLAND, I. (1989): Formulations and benefits of the Law of Demeter, *Sigplan Notices*, 24(3): 67-78.

LIEBERHERR, K.J., HOLLAND, I. and RIEL, A. (1988): Object-oriented programming: an objective sense of style, *Procs. OOPSLA '88*, ACM Press, 323-334.

MCGIBBON, B. (1995): *Managing Your Move to Object Technology. Guidelines and Strategies for a Smooth Transition*, New York: SIGS Books, 268.

McGREGOR, J.D. and KORSON, T. (1993): Supporting dimensions of classification in object-oriented design, *J. Obj.-Oriented Programming*, 5(9): 25-30.

MEYER, B. (1988): *Object-Oriented Software Construction*, Hemel Hempstead, United Kingdom: Prentice Hall, 534.

MEYER, B. (1992a): *Eiffel: The Language*, New York: Prentice Hall, 594.

MEYER, B. (1992b): Applying "design by contract", *IEEE Computer*, 25(10): 40-52.

MILLER, G. (1956): The magical number seven, plus or minus two: some limits on our capacity for processing information, *The Psychological Review*, 63(2): 81-97.

OFFEN, R.J. and JEFFERY, R. (1997): Establishing software measurement programs, *IEEE Software*, 14(2): 45-53.
OMG. (1997a): UML Semantics. Version 1.1, 15 September 1997, OMG document ad/97-08-04.
OMG (1997b): UML Notation. Version 1.1, 15 September 1997, OMG document ad/97-08-05.
PAGE-JONES, M. (1995): *What Every Programmer Should Know About Object-Oriented Design*, New York: Dorset House Publishing, 370.
PREE, W. (1997): Essential framework design patterns, *Object Magazine*, 7(1): 34-37.
PREECE, J., ROGERS, Y., SHARP, H., BENYON, D., HOLLAND, S. and CAREY, T. (1994): *Human Computer Interaction*, Addison-Wesley.
RATIONAL. (1997): UML Semantics, version 1.0, 13 January 1997 (unpubl), available from http://www.rational.com
REENSKAUG, T., WOLD, P. and LEHNE, O.A. (1996): *Working with Objects. The OOram Software Engineering Manual*, Greenwich, CT, USA: Manning, 366.
RIEL, A.J. (1996): *Object-Oriented Design Heuristics*, Reading, MA, USA: Addison-Wesley, 379.
RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., and LORENSEN, W. (1991): *Object-Oriented Modeling and Design*, New Jersey, USA: Prentice Hall.
SHARBLE, R.C. and COHEN, S.S. (1993): The object-oriented brewery: a comparison of two object-oriented development methods, *ACM SIGSOFT Software Engineering Notes*, 18(2): 60-73.
SIMONS, A.J.H. (1999): Use cases considered harmful, MITCHELL, R., WILLS, A., BOSCH J. and MEYER B. (eds.), in *Procs. TOOLS29*, IEEE Computer Society Press, 194-203.
SIMONS, A.J.H. and GRAHAM, I. (1999): 30 things that go wrong in object modelling with UML 1.3, Chapter 16, KILOV, H., RUMPE B. and SIMMONDS I. (eds.), in *Behavioural Specifications of Businesses and Systems*, Kluwer Academic Publishers, 221-242.
STEPANOV, A. and LEE, M. (1994): *The Standard Template Library*, Hewlett Packard Labs.
STROUSTRUP, B. (1991): *The C++ Programming Language*, (2nd edition), Reading, MA: Addison-Wesley, 328.
SZYPERSKI, C. (1998): *Component Software. Beyond Object-Oriented Programming*, Harlow, England: Addison-Wesley, 411.
VALAER, L. and BABB, R. (1997): Choosing a user interface development tool, *IEEE Software*, 14(4): 29-39.
WIRFS-BROCK, A., and WILKERSON, B. (1989): Variables limit reusability, *J. Obj.-Oriented Programming*, 2(1): 34-40.
WIRFS-BROCK, R. (1994): Adding to your conceptual toolkit: what's important about responsibility-driven design?, *Report on Object Analysis and Design*, 1(2): 39-41.

APPENDIX A: RESPONSIBILITIES

Concepts and abstractions focus on the external view – how an object is seen, what it is responsible for knowing and how it behaves. Responsibilities, introduced by Wirfs-Brock and Wilkerson (1989), represent high level abstractions of integrated state and behaviour. As illustration, consider the CIRT to represent a horse. This can be done either using:

- The *data-driven* approach describes a horse in terms of its parts: head, tail, body, leg(4).
- The *procedural* (functional interface) approach describes a horse in terms of operations it can perform: walk, run, trot, bite, eat, neigh.
- The *responsibility-driven* approach describes a horse interms of its responsibilities: communicate, carry things, maintain its living systems.

A data-driven approach harks back to earlier structured methods, such as the entity-relationship modelling adopted in Shlaer/Mellor, OMT and Fusion, in which the emphasis was more on pure storage entities. Whilst OMT does not prevent the developer from identifying "behaviour" during analysis, all examples are data-focussed. The Coad and Yourdon methodology has a similar data-driven focus. Fusion, on the other hand, *does* mandate that only data are considered during analysis. The use of a data-driven ("OO") methodology such as these may, if used by an unskilled developer, lead to traditional entity-relationship models rather than true object models. In a true object model, behaviour (as exemplified by responsibilities) tends to be more equally distributed among CIRTs - this was amply demonstrated in a metrics study by Sharble and Cohen (1993) in which the data-driven design (DDD) was of a lower quality (as measured by their set of metrics) than the responsibility-driven design (RDD).

Whilst both a DDD and a RDD approach can work, particularly in the hands of a skilled developer, we have generally found that novices prefer a RDD. Indeed, it has been found that a

responsibility-driven approach is more than useful in teaching undergraduate students the OO paradigm. Whilst permitting DDD, we strongly recommend a RDD focus to your development strategy. Whilst a responsibility driven design focusses on an holistic approach at the model building level, a contract-driven approach takes that one level higher and applies those ideas to the life-cycle process itself. The objects representing Activities have both responsibilities and contracts: pre- and post-conditions controlling the overall development process.

APPENDIX B: ABSTRACTION AND CLASSIFICATION

Modelling is representing a system, here first a business system and then a software system, at a given level of abstraction. The use of abstraction is not only one of the most central tenets of OT, but also for many one of the most difficult. Abstraction requires the capturing of the essence of a problem and its elements – in terms of types, objects, relationships etc. – at a given level of detail or granularity.

An abstraction mindset leads to the successful introduction and use of classification. It is natural for people to group together common items in their everyday world in order to impose a cognitive or mental model of the external of "real world" in which we all exist. Indeed, some philosophers argue that since the only reality is our own cognitive perception of the world around us – and each of us constructs our cognitive map independently and differently – then the real world has no meaning or existence!

This skill of classification to a higher abstraction level leads to the notion of generalisation – identifying a superclass or supertype, which may often be an abstract class in OO jargon (i.e. one with no instances), but which captures the essence of a shared concept.

APPENDIX C: STEREOTYPES

Stereotypes are widely used in CIRT modelling. The word stereotype, in a linguistic sense, suggests something representative – sometimes pejoratively. In OT, its meaning has been changed to mean a temporary metalevel subclassification. In OOSE (Jacobson *et al*, 1992), CIRTs could be either controller objects, entity objects or interface objects – a similar grouping technique was used by Budd (1991). These stereotype labels indicate the superset to which the object belongs. This approach is merely a convenient way of dividing up objects following a "divide and conquer" style. There is nothing sacrosanct in these classification categories and no impact on the semantics or implementation styles implied. Following Wirfs-Brock's (1994) discussion on OO stereotypes, the concept has been embodied in the UML approach to modelling. Stereotypes are applied liberally as means of extending the concepts as portrayed in the UML metamodel. Whilst providing much-vaunted extensibility (an important aim of the metamodel), excessive use of stereotyping runs the risk of different developers producing overlapping or ill-defined classifications (Berner *et al*, 1999). For example, airline A may stereotype their frequent flyers as bronze, silver or gold based on thresholds of annual miles flown of 100,000 and 200,000 whereas airline B uses the same stereotypes based on annual expenditures. Thus the <<gold>> stereotype is ambiguous and third (and fourth) parties adopting it will find no agreed semantics. In fact, this leads to the identical problem observed by McGregor and Korson (1993) in their introduction of the discriminant – used later in MOSES (Henderson-Sellers and Edwards, 1994), OML (Firesmith *et al*, 1997) and UML (OMG, 1997a;b).

As well as classes, some relationships are suggested as ripe for stereotyping. OMG (1997a) encourages the use of the <<uses>> and <<extends>> stereotypes[7] on the dependency relationship

[7] Recently (OMG, 1999), these were replaced by <<include>> and <<extend>> stereotypes.

between use cases (the subset approach to stereotyping). Similarly, in the class diagram, there are a large number of predefined stereotypes on the Dependency relationship and its subtypes. However, it should be noted that, while a stereotype is defined to be, essentially, a user-defined partition at the model rather than the metamodel level, there are several stereotypes predefined in UML e.g. <<access>>, <<friend>> and <<import>> as stereotypes of the Permission relationship (Figure 7). Furthermore, in the UML standard, several *metatypes* (e.g. Include, Extend, Binding, Uses in Figure 7 – but there are many others) are depicted using a stereotype notation.

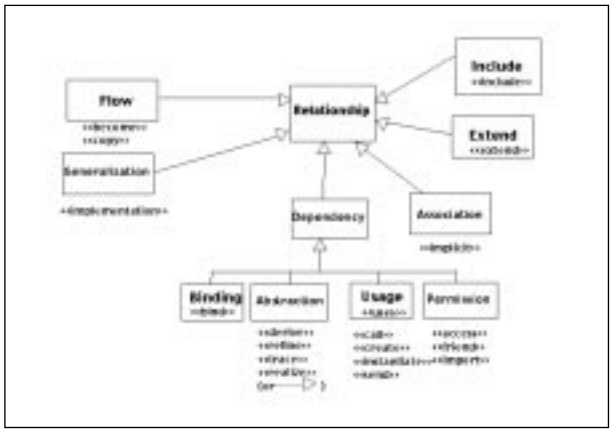

Figure 7: The Relationship metalevel hierarchy of UMLV1.3. Some of the metatypes in the metamodel are shown using stereotypes while others have stereotypes which represent partitions of the metatype.

BIOGRAPHICAL NOTES

Brian Henderson-Sellers is Director of the Centre for Object Technology Applications and Research and Professor of Information Systems at University of Technology, Sydney (UTS). He is author of eight books on object technology and is well-known for his work in OO methodologies (MOSES, COMMA and OPEN) and in OO metrics.

Brian has been Regional Editor of Object-Oriented Systems, a member of the editorial board of Object Magazine/Component Strategies and Object Expert for many years. He was the Founder of the Object-Oriented Special Interest Group of the Australian Computer Society (NSW Branch) and Chairman of the Computerworld Object Developers' Awards committee for ObjectWorld 94 and 95 (Sydney). He is a frequent, invited speaker at international OT conferences.

Tony Simons is a Lecturer in the Department of Computer Science at the University of Sheffield in the UK. His research interests span object-oriented models of speech events and low-level phonetic decoding, object-oriented type theory and language design as well as object-oriented analysis and design methods, verification and testing. Tony is a regular presenter at international object-oriented conferences such as ECOOP, OOPSLA and TOOLS. Tony is also a member of the OPEN Consortium and the pUML group.

He is the chief architect behind the Discovery Method for developing object-oriented systems and co-author (with Professor Henderson-Sellers) of The OPEN Toolbox of Techniques (Addison-Wesley, 1998).