Object-Oriented and Conventional Analysis and Design Methodologies

Comparison and Critique

Robert G. Fichman and Chris F. Kemerer
Massachusetts Institute of Technology

The question of whether emerging object-oriented analysis and design methodologies require incremental or radical changes on the part of prospective adopters is being vigorously debated.

Although the concepts underlying object-orientation as a programming discipline go back two decades, it's only in the last few years that object-oriented analysis (OOA) and object-oriented design (OOD) methodologies have begun to emerge. Object orientation certainly encompasses many novel concepts, and some have called it a new paradigm for software development. Yet, the question of whether object-oriented methodologies represents a radical change over such conventional methodologies as structured analysis remains a subject of much debate.

Yourdon has divided various object-oriented methodologists into two camps, *revolutionaries* and *synthesists*.[1] Revolutionaries believe that object orientation is a radical change that renders conventional methodologies and ways of thinking about design obsolete. Synthesists, by contrast, see object orientation as simply an accumulation of sound software engineering principles that adopters can graft onto their existing methodologies with relative ease.

On the side of the revolutionaries, Booch[2] states

> Let there be no doubt that object-oriented design is fundamentally different from traditional structured design approaches: it requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture.

Coad and Yourdon[3] add

> We have no doubt that one could arrive at the same results [as Coad and Yourdon's OOA methodology produces] using different methods; but it has also been our experience that the thinking process, the discovery process, and the communication between user and analyst are fundamentally different with OOA than with structured analysis.

On the side of the synthesists, Wasserman, Pircher, and Muller[4] take the position that their object-oriented structured design (OOSD) methodology is essentially an elaboration of structured design. They state that the "foundation of OOSD is structured design" and that structured design "includes most of the necessary concepts and notations" for OOSD. Page-Jones and Weiss[5] take a similar position in stating that

The problem is that object orientation has been widely touted as a revolutionary approach, a complete break with the past. This would be fascinating if it were true, but it isn't. Like most engineering developments, the object-oriented approach is a refinement of some of the best software engineering ideas of the past.

Factors to consider. One of the most important assessments a company must make in considering the adoption of a technical innovation is where the innovation falls on the incremental-radical continuum in relation to its own current practice. Incremental innovations introduce relatively minor changes to an existing process or product and reinforce the established competencies of adopting firms. Radical innovations are based on a different set of engineering and scientific principles, and draw on new technical and problem-solving skills.

If object-oriented analysis and design comes to be regarded as a radical change by most organizations, then a strong, negative impact on the ultimate rate of adoption of the technology can be expected. Compared with incremental change, implementation of radical change involves greater expense and risk, and requires different management strategies. Many development groups have already invested considerable resources in conventional methodologies like structured analysis/structured design or information engineering. These investments can take many forms, including training in the specifics of the methodology, acquisition of automated tools to support the methodology, and repositories of analysis and design models accumulated over the course of employing the methodology.

On an industry-wide level, vendors have been actively developing more powerful tools to support conventional methodologies, and a growing pool of expertise now exists in the use of these tools. To the extent that object orientation is a radical change, investments in conventional methodologies will be lost: Staff will have to be retrained, new tools will have to be purchased, and a likely expensive conversion process will be necessary.

Implementation of radically new technologies also involves a much greater element of risk because the full range of impacts is typically unknown. Moreover, the implementation of a radically new methodology requires different strategies to manage this risk and to overcome other implementation barriers (such as resistance to change).

The radical-versus-incremental debate is crucial to assessing the future of object orientation and formulating a transition strategy, but unfortunately no comprehensive analyses have been performed comparing leading object-oriented methodologies with conventional methodologies. Two surveys of object-oriented methodologies have been compiled, but these only cover either analysis[6] or design,[7] and neither draws specific comparisons with conventional methodologies. Loy[8] provides an insightful commentary on the issue of conventional versus object-oriented methodologies, although no specific methodologies are compared.

The current research fills the gap left by other surveys by analyzing several leading conventional and object-oriented analysis and design methodologies, including a detailed point-by-point comparison of the kinds of modeling tools provided by each. A review (described below in greater detail) was performed that resulted in the selection of six analysis methodologies and five design methodologies. The analysis methodologies were

- DeMarco structured analysis,
- Yourdon modern structured analysis,
- Martin information engineering analysis,
- Bailin object-oriented requirements specification,
- Coad and Yourdon object-oriented analysis, and
- Shlaer and Mellor object-oriented analysis.

The design methodologies were

- Yourdon and Constantine structured design,
- Martin information engineering design,
- Wasserman et al. object-oriented structured design,
- Booch object-oriented design, and
- Wirfs-Brock et al. responsibility-driven design.

Incremental or radical? We conclude that the object-oriented analysis methodologies reviewed here represent a radical change over process-oriented methodologies such as DeMarco structured analysis but only an incremental change over data-oriented methodologies such as Martin information engineering. Process-oriented methodologies focus attention away from the inherent properties of objects during the modeling process and lead to a model of the problem domain that is orthogonal to the three essential principles of object orientation: encapsulation, classification of objects, and inheritance.

By contrast, data-oriented methodologies rely heavily on the same basic technique — information modeling — as each of the three OOA methodologies. The main differences between OOA and data-oriented conventional methodologies arise from the principle of encapsulation of data and behavior: OOA methodologies require that all operations be encapsulated within objects, while conventional methodologies permit operations to exist as subcomponents of disembodied processes. At the level of detail required during analysis, however, we conclude that expert information modelers will be able to learn and apply the principle of encapsulation without great difficulty.

Regarding design methodologies, we conclude that object-oriented design is a radical change from *both* process-oriented and data-oriented methodologies. The OOD methodologies we review here collectively model several important dimensions of a target system not addressed by conventional methodologies. These dimensions relate to the *detailed* definition of classes and inheritance, class and object relationships, encapsulated operations, and message connections. The need for adopters to acquire new competencies related to these dimensions, combined with Booch's uncontested observation that OOD uses a completely different structuring principle (based on object-oriented rather than function-oriented decomposition of system components), renders OOD as a radical change.

Conventional methodologies

A systems development methodology combines tools and techniques to guide the process of developing large-scale information systems. The evolution of modern methodologies began in the late 1960s with the development of the concept of a systems development life cycle (SDLC). Dramatic increases in hardware performance and the adoption of high-level languages had enabled much larger and more complicated systems to be built. The SDLC attempted to bring order to the development process, which had outgrown the ad hoc project control methods of the day, by decomposing the process into discrete project phases with "frozen" deliverables—formal documents—that served as the input to the next phase.

Structured methodologies. The systems development life cycle concept gave developers a measure of control, but provided little help in improving the productivity and quality of analysis and design per se. Beginning in the 1970s, structured methodologies were developed to promote more effective analysis and more stable and maintainable designs. Early structured methodologies were largely *process-oriented*, with only a minor emphasis on modeling of entities and data. This emphasis on processes seemed natural, given the procedural programming languages and batch, file-based applications commonplace at the time. Although many authors contributed to the so-called structured revolution, our review concentrates on the critical contributions of Yourdon and Constantine,[9] DeMarco,[10] and Ward and Mellor.[11]

Yourdon and Constantine structured design provided a method for developing a system architecture that conformed to the software engineering principles of modularity, loosely coupled modules, and module cohesion. The structure chart (see the sidebar, "Tools for structured methodologies") was the primary tool for modeling a system design. (Although the emphasis of structured design was on creating a module architecture, the methodology also suggested dataflow diagrams for modeling processes and hierarchy diagrams for defining data structure.)

DeMarco's seminal work enlarged the structured approach to encompass analysis. DeMarco prescribed a series of steps for performing structured analysis, flowing from modeling of existing systems (using dataflow diagrams) to modeling of the system to be developed (using dataflow diagrams, mini-specifications, and a data dictionary). Although modeling of data was not ignored, the emphasis was on modeling processes. The ultimate goal of structured analysis and design was to create a top-down decomposition of the functions to be performed by the target system.

Continuing in the structured tradition, Ward and Mellor recommended significant extensions to structured analysis to better support modeling of real-time systems. Their methodology added entity-relationship diagrams and state-transition diagrams to the structured analysis toolset. Entity-relationship diagrams illustrate the structure of entities and their interrelationships, while state-transition diagrams focus on system and subsystem states and the events that caused transitions between states.

In recognition of the evolution of systems, languages, and tools over the past two decades, Yourdon[12] updated structured analysis under the name *modern structured analysis.* Modern structured analysis differs from DeMarco's original work in several respects: It no longer recommends modeling of current implemented systems; it adds a preliminary phase to develop an "essential model" of the system; it substitutes a technique known as "event partitioning" for top-down functional decomposition as the preferred technique for constructing dataflow diagrams; it places more emphasis on information modeling (via entity-relationship diagrams) and behavior modeling (via state-transition diagrams); and it encourages prototyping.

These updates have served to blur somewhat the one-time clear distinctions between structured methods and the data-oriented methods that we describe next.

Information engineering. In the late 1970s and early 1980s, planning and

Tools for structured methodologies

Dataflow diagram (DFD) — Depicts processes (shown as bubbles) and the flow of data between them (shown as directed arcs). DFDs are usually organized into a hierarchy of nested diagrams, where a bubble on one diagram maps to an entire diagram at the next lower level of detail. Does not depict conditional logic or flow of control between modules.

Data-dictionary — A repository of definitions for data elements, files, and processes. A precursor to the more comprehensive "encyclopedias."

Entity-relationship diagram (ERD) — Depicts real-world entities (people, places, things, concepts) and the relationships between them. Various notations are used, but usually entities are portrayed as boxes and relationships as arcs, with different terminating symbols on the arcs to depict cardinality and whether the relationship is mandatory or optional.

Hierarchy diagram — A simple diagram that shows a top-to-bottom hierarchical decomposition of data files and data items (enclosed within boxes) connected by undirected arcs.

Mini-spec — A structured-English specification of the detailed procedural logic within a process; performs the same function as the traditional flowchart. A mini-spec is developed for each process at the lowest level of nesting in a set of DFDs.

State-transition diagram — Depicts the different possible states of a system or system component, and the events or messages that cause transitions between the states.

Structure chart — Depicts the architecture of a system as a hierarchy of functions (boxes) arranged in a tree-like structure. Identifies interconnections between functions, and input and output parameters. Does not depict control structures like condition, sequence, iteration, or selection.

modeling of data began to take on a more central role in systems development, culminating in the development of data-oriented methodologies such as information engineering. The conceptual roots of data-oriented methodologies go back to the 1970s with the invention of the relational database model and entity-relationship modeling, although it took several years for mature data-oriented methodologies to emerge.

The data-oriented approach has two central assumptions:

(1) Organizational data provides a more stable foundation for a system design than organizational procedures.

(2) Data should be viewed as an organizational resource independent of the systems that (currently) process the data.

One outgrowth of the data-oriented approach was the creation of a new information systems subfunction, data administration, to help analyze, define, store, and control organizational data.

Martin[13] information engineering is a comprehensive methodology that extends the data-oriented approach across the entire development life cycle. While structured methods evolved backwards through the life cycle from programming, information engineering evolved forward through the life cycle from planning and analysis. Martin defines information engineering as consisting of four phases:

(1) Information strategy planning,
(2) Business area analysis,
(3) System design, and
(4) Construction.

Information engineering distinguishes activities that are performed on the level of a business unit (planning and analysis) from those that are project-specific (design and construction). Compared with structured methods, information engineering recommends a much broader range of analysis techniques and modeling tools, including enterprise modeling, critical-success-factors analysis, data modeling, process modeling, joint-requirements planning, joint-applications design, time-box methodology, and prototyping (see the sidebar, "Tools for Martin information engineering").

Information engineering describes planning as an organization-wide activity that develops an enterprise model and a high-level data architecture. Business area analysis attempts to capture a more detailed understanding of business activities and their interdependencies, using such tools as data-model diagrams, decomposition diagrams, process-dependency diagrams, and entity-process matrices. The design phase builds on the results of prior phases and produces a detailed model of a target

Tools for Martin information engineering

Action diagram — Used to depict detailed procedural logic at a given level of detail (for example, at a system level or within individual modules). Similar to structured English, except graphical constructs are used to highlight various control structures (condition, sequence, iteration, and selection).

Bubble chart — A low-level diagram used as an aide to normalization of relational tables. Shows attributes (depicted as bubbles) and the functional dependencies between them (depicted as directed arcs).

Dataflow diagram (DFD) — Conforms to the conventional notation and usage for dataflow diagrams (see the sidebar, "Tools for structured methodologies").

Data-model diagram — Depicts data entities (boxes) and their relational connections (lines). Shows cardinality and whether the connections are optional or mandatory. Similar to the entity-relationship diagram.

Data-structure diagram — Shows data structures in a format appropriate to the database management system to be used for implementation.

Encyclopedia — A more comprehensive version of the data dictionary that serves as an integrated repository for modeling information from all development phases, including the enterprise model; organizational goals, critical success factors, strategies, and rules; data models and data definitions; process models and process definitions; and other design-related information. Automated support is assumed.

Enterprise model — A model that defines, at a high level, the functional areas of an organization and the relationships between them. It consists of text descriptions of functions (usually an identifiable business unit such as a department) and processes (a repetitive, well-defined set of tasks that support a function).

Entity-process matrix — Cross-references entities to the processes that use them.

Process-decomposition diagram — A hierarchical chart that shows the breakdown of processes into progressively increasing detail. Similar to the conventional tree diagram, except a particularly compact notation is used to fit many levels on one page.

Process-dependency diagram — A diagram consisting of processes (depicted by bubbles) and labeled arcs. It shows how each process depends on the prior execution other processes. Similar to a dataflow diagram, except conditional logic and flow of control is also depicted.

State-transition diagram — Conforms to the conventional notation and usage for state-transition diagrams (see the sidebar, "Tools for structured methodologies").

system consisting of process-decomposition diagrams, process-dependency diagrams, dataflow diagrams, action diagrams, and data-structure diagrams. System construction, the last phase of information engineering, consists of translating the models from the design phase to an operational system — ideally using a code generator.

Object-oriented analysis methodologies

As with traditional analysis, the primary goal of object-oriented analysis is the development of an accurate and complete representation of the problem domain. We conducted a literature search to identify well-documented, broadly representative OOA methodologies first published in book form or as detailed articles in refereed journals from 1980 to 1990. This search resulted in the selection of three methodologies from Coad and Yourdon,[3] Bailin,[14] and Shlaer and Mellor.[15,16] Numerous OOA methodologies have emerged in recent years. Since no more than a few methodologies could be compared in depth, two criteria — maturity (first published prior to 1990) and form of publication (book or refereed journal) — were used to select among them. Several methodologies were identified that did not meet these criteria (see Fichman and Kemerer[17]) although this should not be taken to mean they are inferior to those that did. Object-oriented analysis is, of course, quite young; it is much too early to predict which (if any) of the current methodologies will come to be recognized as standard works in the field. The goal here is to provide a detailed comparison of representative methodologies at a single point in time, not a comprehensive review.

The three methodologies are presented in the order of their similarity to conventional methodologies. Bailin's methodology is viewed as most similar, followed by Coad and Yourdon's, and then Shlaer and Mellor's.

Bailin object-oriented requirements specification. Bailin developed object-oriented requirements specification (OOS) in response to a perceived incompatibility between conventional structured analysis and object-oriented design. Outwardly, the method resembles structured analysis in that a system decomposition is performed using a dataflow diagram-like notation. Yet, there is an important difference: Structured analysis specifies that functions should be grouped together only if they are "constituent steps in the execution of a higher level function," while OOS groups functions together only if they "operate on the same data abstraction."[14] In other words, functions cannot exist as part of disembodied processes, but must be subordinated to a single entity. (Bailin uses the term entity rather than object for stylistic reasons only; the terms are assumed to be interchangeable.) This restriction is used to promote encapsulation of functions and data.

Two distinctions are central to OOS. First, Bailin distinguishes between *entities*, which possess underlying states that can persist across repeated execution cycles, and *functions*, which exist solely to transform inputs to outputs and thus have no underlying states remembered between cycles. Entities can be further decomposed into subentities or functions, but functions can only be decomposed into subfunctions.

Second, Bailin distinguishes between two classes of entities, *active* and *passive*. Active entities perform operations (on themselves or other entities) important enough to be considered in detail during the analysis phase, while passive entities are of lesser importance and can therefore be treated as a "black box" until the design phase. These distinctions are important because, as we show below, active entities, passive entities, and functions are each modeled differently during the analysis process.

The OOS methodology consists of a seven-step procedure:

(1) *Identify key problem domain entities.* Draw dataflow diagrams and then designate objects that appear in process names as candidate entities.

(2) *Distinguish between active and passive entities.* Distinguish between entities whose operations are significant in terms of describing system requirements (active entities) versus those whose detailed operations can be deferred until design (passive). Construct an entity-relationship diagram (ERD).

(3) *Establish dataflows between active entities.* Construct the top-level (level 0) entity-dataflow diagram (EDFD). Designate each active entity as a process node and each passive entity as a dataflow or data store.

(4) *Decompose entities (or functions) into subentities and/or functions.* This step is performed iteratively together with steps 5 and 6. Consider each active entity in the top-level EDFD and determine whether it is composed of lower level entities. Also consider what each entity does and designate these operations as functions. For each of the subentities identified, create a new EDFD and continue the decomposition process.

(5) *Check for new entities.* At each stage of decomposition, consider whether any new entities are implied by the new functions that have been introduced and add them to the appropriate EDFD, reorganizing EDFDs as necessary.

(6) *Group functions under new entities.* Identify all the functions performed by or on new entities. Change passive to active entities if necessary and reorganize EDFDs as appropriate.

(7) *Assign entities to appropriate domains.* Assign each entity to some application domain, and create a set of ERDs, one for each domain.

The end result of OOS is an entity-relationship diagram, together with a hierarchy of entity-dataflow diagrams (see the sidebar "Tools for Bailin object-oriented requirements specification"). Bailin's methodology conforms to the essential principals of object orientation, although explicit object-oriented terminology is not used. (Loy[8] lists three principles that distinguish object orientation from other approaches: encapsulation of attributes, operations, and services within objects; classification of object abstractions; and inheritance of common attributes between classes.) The entity-relationship diagrams capture a classification of objects as well as opportunities for inheritance, and Bailin's functions map to the object-oriented concept of encapsulated services.

Coad and Yourdon object-oriented analysis. Coad and Yourdon[3] view their OOA methodology as building "upon the best concepts from information modeling, object-oriented programming languages, and knowledge-based systems." OOA results in a five-layer model of the problem domain, where each layer builds on the previous layers. The layered model is constructed using a five-step procedure:

(1) *Define objects and classes.* Look for structures, other systems, devices, events, roles, operational procedures, sites, and organizational units.

(2) *Define structures.* Look for relationships between classes and represent them as either general-to-specific structures (for example, employee-to-sales manager) or whole-to-part structures (for example, car-to-engine).

(3) *Define subject areas.* Examine top-level objects within whole-to-part hierarchies and mark these as candidate subject areas. Refine subject areas to minimize interdependencies between subjects.

(4) *Define attributes.* Identify the atomic characteristics of objects as attributes of the object. Also look for associative relationships between objects and determine the cardinality of those relationships.

(5) *Define services.* For each class and object, identify all the services it performs, either on its own behalf or for the benefit of other classes and objects.

The primary tools for Coad and Yourdon OOA are class and object diagrams and service charts (see the sidebar, "Tools for Coad and Yourdon object-oriented analysis"). The class and object diagram has five levels, which are built incrementally during each of the five analysis steps outlined above. Service charts, which are "much like a [traditional] flow chart," are used during the service definition phase to represent the internal logic of services." In addition, service charts portray state-dependent behavior such as preconditions and triggers (operations that are activated by the occurrence of a predefined event).

Coad and Yourdon explicitly support each of the essential principles of object orientation. The class and objects diagram (levels 1, 2, and 4) provides an object classification and identifies potential inheritance relationships. In addition, encapsulation of objects is modeled through the concept of exclusive services. Coad and Yourdon OOA is similar to modern structured analysis (MSA) and information engineering in its emphasis on information modeling, but differs in providing constructs for modeling exclusive services and message connections.

Shlaer and Mellor object-oriented analysis. Shlaer and Mellor developed their object-oriented analysis methodology over the course of several years of consulting practice in information modeling. Although information modeling forms the foundation of the method, two other views of the target system are prescribed as well: a state model and a process model. This three-way view of the system, contained in interrelated information, state, and process models, is proposed as a complete description of the problem domain. Shlaer and Mellor advocate a six-step procedure:

(1) *Develop an information model.* This model consists of objects, attributes, relationships, and multiple-object constructions (based on is-a, is-part-of, and associative relationships). (The term object, as used by Shlaer and Mellor, is equivalent to the conventional notion

Tools for Bailin object-oriented requirements specification

Entity-relationship diagram — Conforms to the conventional notation and usage for entity-relationship diagrams (see the sidebar, "Tools for structured methodologies").

Entity-dataflow diagram (EDFD) — A variant on the conventional dataflow diagram wherein each process node contains either an active entity or some function related to an active entity, rather than disembodied processes. Active entities and functions are enclosed within bubbles. Bubbles are connected to each other and to data stores by labeled arcs containing dataflows. Dataflows and data stores are passive entities.

Entity dictionary — A repository of entity names and descriptions, analogous to the data dictionary of DeMarco structured analysis.

Tools for Coad and Yourdon object-oriented analysis

Class and object diagram — A complex diagram consisting of five layers, each adding a level of detail. The layers are (1) class and object layer, which shows classes and objects enclosed within boxes with rounded corners, (2) structures layer, which connects classes and objects with arcs to show generalization-specialization and whole-part inheritance relationships, (3) subjects layer, which adds a border around closely related classes, (4) attributes layer, which adds a list of attributes inside the class and object boxes and identifies associative relationships between objects, and (5) service layer, which adds a list of services inside the class and object boxes and provides arcs showing message connections between boxes.

Object-state diagram — A simple diagram that shows all the possible states of an object and the allowed transitions between states. States are enclosed within boxes and transitions are represented as directed, unlabeled arcs between states.

Service chart — A flowchart-like diagram that depicts the detailed logic within an individual service, including object-state changes that trigger or result from the service.

Tools for Shlaer and Mellor object-oriented analysis

Action-dataflow diagram (ADFD) — Similar to DFDs, except ADFDs are used to model elementary "action" processes rather than to create a top-down functional decomposition of the entire system. Standard DeMarco notation is used, except additional notations are provided to show control flows and to show conditionality in the execution of dataflows and control flows.

Domain chart — A simple diagram that illustrates all domains relevant to the implementation of an OOA model. Domains are enclosed within bubbles and are connected by directed arcs. These arcs represent bridges between domains. Four types of domains are identified: application, service, architectural, and implementation.

Information structure diagram — A variant on the entity-relationship diagram that shows objects (boxes) connected by relationships (labeled arcs). Attributes are listed within object boxes. Relationship conditionality and multiplicity are also shown.

Object and attribute description — A text description of an object, including object name, object description, object identifier, a list of attributes, and descriptions of each attribute.

Object-access model — Shows the synchronous interactions between state models at the global system level. Synchronous interactions occur when one state model accesses the instance data of another object via an accessor process. State models (enclosed in ovals) are connected to each other by directed arcs labeled with the accessor process.

Object-communication model — Shows the asynchronous interactions between state models and external entities at the global system level. State models (enclosed in ovals) are connected to each other and to external entities (enclosed in boxes) by directed arcs labeled with communicating events.

Process description — A narrative description of a process. A process description is needed for every process appearing on an action-dataflow diagram.

Relationship specification — A text description of each relationship, including the name of the relationship (from the point of view of each object), conditionality (required or optional), multiplicity (one-to-one, one-to-many, many-to-many), a general description of the relationship, and identification of the attributes (foreign keys) through which the relationship is formalized.

State model — State models conform to the conventional notation for state-transition diagrams (see the sidebar, "Tools for structured methodologies"), except they are used to model the states of problem domain entities. (Traditional STDs, by contrast, model the states of a system, system component, or process.)

Subsystem access model — Shows synchronous interactions between object-access models (one OAM exists for each subsystem). Directed, labeled arcs represent synchronous processes flowing between OAMs (enclosed in boxes).

Subsystem communication model — Shows asynchronous interactions between object-communication models (one OCM exists for each subsystem). Directed, labeled arcs represent asynchronous events flowing between OAMs (enclosed in boxes).

Subsystem relationship model — Shows relationships between information models(where each subsystem has exactly one information model). Information models (enclosed in boxes) are connected by undirected arcs (labeled with relationships).

of an entity, that is, a person, place, thing, or event that exists in the real world.)

(2) *Define object life cycles.* The focus here is on analyzing the life cycle of each object (from creation through destruction) and formalizing the life cycle into a collection of states (some predefined condition of an object), events (signals that cause transitions from state to state), transition rules (which specify the allowable transitions between states), and actions (activities or operations that must be done by an object upon arrival in a state). This step also defines *timers*, mechanisms used by actions to generate a future event. The primary tool during this step is the state model. (See the sidebar, "Tools for Shlaer and Mellor object-oriented analysis.")

(3) *Define the dynamics of relationships.* This step develops a state model for those relationships between objects that evolve over time (dynamic relationships). For each dynamic relationship, an associative object is defined in the information model. Special assigner state models are defined for relationships in which there may be contention between object instances for resources of another object instance.

(4) *Define system dynamics.* This step produces a model of time and control at the system level. An object-communication model (OCM) is developed to show asynchronous control (akin to simple message passing). An object-access model is developed to show synchronous control (instances where one object accesses the instance data of another through an accessor process). Shlaer and Mellor also describe a procedure for tracing threads of control at a high level (by following events on the OCM) and at a more detailed level (by creating a thread-of-control chart for individual actions).

(5) *Develop process models.* For each action, an action-dataflow diagram is created that shows all of the processes for that action, and the data flows among the processes and data stores. (Standard DeMarco notation for DFDs is used, except additional notations are provided to show control flows and to show conditionality in the execution of dataflows and control flows.) OOA defines four types of processes (accessors, event generators, transformations, and tests) and provides guidelines for de-

composing actions into these constituent processes.

(6) *Define domains and subsystems.* For large problems, it can be useful to decompose the subject matter into conceptually distinct domains. Four types of domains are identified: application, service, architectural, and implementation. In addition, it is sometimes useful to decompose the application domain into multiple subsystems.

Shlaer and Mellor provide implicit, rather than explicit, support for the three essential principles of object orientation — classification, inheritance, and encapsulation. The objects and relationships contained in the information structure diagram, while not identical to object-oriented concepts of classification and inheritance, can easily be mapped to these concepts during design. (Regular entities and parent entities engaged in is-a style relationships correspond to classes and superclasses, respectively, and identify candidate inheritance relationships. The is-part-of style relationships correspond to whole-to-part class relationships.) The requirement that each action process (and associated dataflow diagram) be associated with exactly one object preserves encapsulation of those operations.

Comparison of analysis methodologies

The conventional and OOA methodologies reviewed here can be compared along 11 modeling dimensions; these dimensions represent the superset of dimensions supported by the individual methodologies (see Table 1). Since the various methodologists tend to use widely divergent terminology and notations for similar concepts, Table 1 presents the dimensions at a level that captures essential similarities and differences between the methodologies. We examined the concepts and notations advocated by each methodology in detail to determine those that were variants on the same basic idea. (For example, Coad and Yourdon's concept of a generalization-specialization relationship between objects is viewed as essentially the same as the is-a style or subtype/supertype entity relationships described in the other analysis methodologies. When used as part of an OOA methodology, generalization-specialization and is-a relationships are both intended to identify candidate opportunities for inheritance.)

Object-oriented versus conventional analysis. As Table 1 shows, object-oriented analysis covers many of the same dimensions as Yourdon MSA and Martin information engineering, although there is a marked contrast between OOA and DeMarco structured analysis. MSA, information engineering, and all of the object-oriented methodologies provide a variety of tools for modeling entities. These include tools for defining entity relationships and attributes (see Table 1, rows 1 through 4) and partitioning large models by grouping naturally related entities (row 5). MSA, Coad and Yourdon OOA, and Shlaer and Mellor OOA support modeling of states (row 6), although within MSA states are modeled at the level of a system or system component, while in the OOA methodologies states are modeled at the level of problem domain entities (objects). DeMarco structured analysis, MSA, Coad and Yourdon OOA, and Shlaer and Mellor OOA provide tools for defining the detailed logic within functions or services (row 7).

The most important differences between object-oriented and conventional analysis methodologies ultimately stem from the object-oriented requirement of encapsulated operations. Conventional methodologies provide tools to create a functional decomposition of operations (row 8) and to model end-to-end processing sequences (row 9). A functional decomposition of systems violates encapsulation because operations can directly access a multitude of different entities and are not subordinated to any one entity; so it is appropriate that no object-oriented methodology provides support here. It is less clear why none of the OOA methodologies as reviewed here provide an explicit model of end-to-end processing sequences, since there is no inherent incompatibility between this view of a system and object orientation. This issue is discussed further in the concluding section.

All the OOA methodologists recognize a need to develop some sort of model of system operations, albeit in a way that preserves encapsulation. As a result, each methodology provides new tools, or variants on conventional tools, for modeling operations as exclusive services of objects (row 10). Row 11 illustrates a further distinction between object-oriented and conventional analysis that arises from the need in object orientation for active communication between entities. (Entities communicate explicitly in an object-oriented system, whereas in a conventional system, entities are passive data stores manipulated by active, independent procedures.)

OOA methodology similarities. The three OOA methodologies illustrated in Table 1 overlap significantly, although different notations and terminology are used for essentially the same concepts. These stylistic differences obscure the fact that, in each of the three methodologies, entities (objects) and relationships establish a foundation for later stages of analysis. Bailin uses a standard ERD notation, which includes the idea of subtype/supertype relationships, as well as any number of user-defined relationships. Shlaer and Mellor's information structure diagrams are similar in terms of content to ERDs. While neither of these methodologies specifically mentions such object-oriented notions as inheritance and object classification, ERDs do, in fact, capture candidate instances of these sorts of relationships using subtype/supertype constructs.

Dynamic entity connections and using-style relationships are also captured in ERDs through such relationship types as creates, destroys, uses, and modifies. Unlike the other two methodologies, Coad and Yourdon refer explicitly to object-oriented concepts such as inheritance and object decomposition. Nonetheless, layers 1, 2, and 4 of the class and objects diagram can easily be mapped to an ERD notation, and these three layers serve essentially the same purpose as an ERD. (The objects and classes identified in level 1 map to the ERD concept of an entity. The generalization-specialization relationships defined in level 2 correspond to subtype/supertype relationships in an ERD. The whole-part structures defined in level 2 and the associative relationships identified in layer 4 correspond to general relationships in an ERD.)

OOA methodology differences. The clearest differences between the methodologies occur in three areas:

(1) depiction of entity states,

Table 1. Comparison of analysis methodologies.

Component	DeMarco Structured Analysis	Yourdon Modern Structured Analysis	Martin Information Engineering	Bailin Object-Oriented Requirements Specification	Coad and Yourdon Object-Oriented Analysis	Shlaer and Mellor Object-Oriented Analysis
1. Identification/ classification of entities*	***Not supported***	**Entity-relationship diagram**	**Data-model diagram**	**Entity-relationship diagram**	**Class and objects diagram layer 1**	**Information-structure diagram**
2. General-to-specific and whole-to-part entity-relationships	*Not supported*	Entity-relationship diagram	Data-model diagram	Entity-relationship diagram	Class and objects diagram layer 2	Information-structure diagram
3. Other entity-relationship (creates, uses, etc.)	***Not supported***	**Entity-relationship diagram**	**Data-model diagram**	**Entity-relationship diagram**	**Class and objects diagram layer 4**	**Information-structure diagram**
4. Attributes of entities	Data dictionary	Data dictionary	Bubble chart	*Not supported*	Class and objects diagram layer 4	Information-structure diagram
5. Large-scale model partitioning	**Dataflow diagram**	**Event-partitioned dataflow diagram**	**Subject databases**	**Domain-partitioned entity-relationship diagrams**	**Class and objects diagram layer 3**	**Domain chart; subsystem communication, access, and relationship models**
6. States and transitions**	*Not supported*	State-transition diagram	*Not supported*	*Not supported*	Object-state diagram; service chart	State model
7. Detailed logic for functions/ services	**Mini-specification**	**Mini-specification**	***Not supported***	***Not supported***	**Service chart**	**Action data-flow diagram; process descriptions**
8. Top-down decomposition of functions***	Dataflow diagram	Dataflow diagram	Process-decomposition diagram	*Not Supported*	*Not Supported*	*Not Supported*
9. End-to-end processing sequences	**Dataflow diagram**	**Dataflow diagram**	**Process-dependency diagram**	***Not supported***	***Not supported***	***Not supported***
10. Identification of exclusive services	*Not supported*	*Not supported*	*Not supported*	Entity-dataflow diagram	Class and objects diagram layer 5	State model, action-data-flow diagram
11. Entity communication (via messages or events)	***Not supported***	***Not supported***	***Not supported***	**Entity-dataflow diagram**	**Class and objects diagram layer 5**	**Object communication model; object-access model**

* For stylistic reasons, the term entity, when it appears in this column, is intended to encompass the terms entity (as used in conventional methodologies and by Bailin), object (as used by Shlaer and Mellor), and class (as used by Coad and Yourdon).

** Conventional STDs as used in Yourdon's MSA describe the states of a system or system component, whereas Shlaer and Mellor's state model and Coad and Yourdon's object-state diagram describe the states of problem domain entities. STDs are not an integral part of information engineering because they are thought to be too detailed for the analysis phase, although Martin allows that they may be used occasionally.

*** Bailin does provide some support for decomposition of functions via entity-dataflow diagrams, but functions are decomposed only at the lowest levels of the diagram rather than at all levels.

(2) definition of exclusive services, and
(3) attention to attribute modeling.

Shlaer and Mellor place the most emphasis on modeling entity states and devote an entire phase of their methodology to defining entity life cycles and depicting them in state models. Coad and Yourdon also model entity states, although this does not appear to be a significant component of the methodology. (Coad and Yourdon's service chart contains much of the same information as Shlaer and Mellor's state model, although it also contains procedural logic unrelated to entity states and transitions. Coad and Yourdon recommend the use of an object-state diagram where helpful, but this diagram does not explicitly name the events that trigger transitions. The object-state model is referred to only sparingly, and does not appear to be a significant component of the final system model.) Bailin has no formal means of depicting entity states and transitions, although he notes that state-transition diagrams are being considered as one possible extension of the method.

Coad and Yourdon and Shlaer and Mellor provide the most detailed representations of exclusive services. In Coad and Yourdon, exclusive services are assigned to objects in layer 5 of the class and objects diagram, and the procedural logic contained in each service is defined in detail in an associated service chart. Shlaer and Mellor also identify exclusive services, which they term *actions*. Actions are identified on state models (object specific) and are defined in detail in the action-dataflow diagram (ADFD) and corresponding process descriptions. The primary tool for modeling Bailin's functions — the entity-dataflow diagram — contains much less detail than Coad and Yourdon's service chart or Shlaer and Mellor's ADFD with process descriptions.

The methodologies differ substantially in their level of attention to attribute modeling. Bailin places a very low emphasis on defining attributes of entities; in fact, he makes no mention of attribute modeling at all. Coad and Yourdon devote a phase to identifying attributes, although not to the extent of ensuring that attributes are normalized within entities. Shlaer and Mellor provide the most emphasis on attribute modeling of the three methodologies, including extensive guidance for describing and normalizing attributes.

Finally, Shlaer and Mellor support some concepts not addressed by Coad and Yourdon or Bailin. These include

(1) a distinction between asynchronous and synchronous control,
(2) the use of timers to generate future events, and
(3) the concept of a dynamic relationship and its role in handling contention between concurrent processes.

OOA: Incremental versus radical change. With regard to the incremental versus radical debate, object-oriented analysis does represent a radical departure from older process-oriented methodologies such as DeMarco structured analysis, but is only an incremental change from data-oriented methodologies like Martin information engineering. Table 1 shows that OOA methodologies typically model six dimensions of the problem domain not contained in a structured analysis model (see rows 1-3, 6, 10-11) and do not model two process-oriented dimensions (rows 8-9) that form the foundation of a De Marco structured analysis model. OOA decomposes the problem domain based on a classification of entities (objects) and their relationships, while structured analysis provides a decomposition based on processes. Developers schooled in DeMarco structured analysis will find the competencies they developed in the construction of hierarchies of DFDs to be, for the most part, irrelevant. Meanwhile, a whole new set of competencies relating to the classification and modeling of entities will have to be developed.

The revolutionaries quoted in the introduction rightly observe that object orientation is fundamentally at odds with the process-oriented view of systems favored by structured methodologies during the 1970s. However, they ignore important changes in these same methodologies over the course of the 1980s towards a more balanced view of data and processes. OOA methodologies only model two dimensions of the problem domain not modeled by Yourdon MSA or Martin information engineering (see Table 1, rows 10-11).

All the OOA methodologies reviewed here contain a heavy information modeling component, and potential adopters with a strong information modeling background should require only limited exposure to absorb the notational differences between conventional information modeling diagrams and the variants developed by OOA methodologists. The idea of shifting from disembodied processes (modeled in dataflow diagrams) to encapsulated services will be more challenging. However, at the level of detail required for analysis, this conceptual shift can probably be absorbed without great difficulty. Shlaer and Mellor OOA, with its emphasis on modeling object life cycles, appears to represent the most significant change of the three OOA methodologies.

Object-oriented design methodologies

Design is the process of mapping system requirements defined during analysis to an abstract representation of a specific system-based implementation, meeting cost and performance constraints. As was done with OOA methodologies, we conducted a literature search to identify broadly representative OOD methodologies first published in book form or as detailed articles in refereed journals from 1980 to 1990. This resulted in the selection of three methodologies from Booch,[2] Wasserman et al.,[4] and Wirfs-Brock et al.[18] Implementation-specific methodologies, such as those targeted at real-time systems using the Ada language, were excluded from consideration.

We present the methodologies in an order based on their similarities to conventional methodologies. Wasserman et al. draws most heavily on structured design and is presented first, followed by Booch, and Wirfs-Brock et al.

Wasserman et al. object-oriented structured design. Object-oriented structured design (OOSD) was developed by Wasserman, Pircher, and Muller. The methodology provides a detailed notation for describing an *architectural design*, which they define as a high-level design that identifies individual modules but not their detailed internal representation. Wasserman et al. state that the overall goal of OOSD is to provide a standard design notation that can support every software design, including both object-oriented and conventional approaches.

OOSD offers a hybrid notation that

Tools for Wasserman et al. object-oriented structured design

Object-oriented structure chart — An updated version of the classical structure chart that adds notations for objects and classes ("information clusters"), methods, visibility, instantiation, exception handling, hidden operations, generic definitions (abstract classes), inheritance, and concurrency. The charts can also be used to show multiple inheritance, message passing, polymorphism, dynamic binding, and asynchronous processes.

incorporates concepts from previous work from several areas, including structure charts from structured design; Booch's notation for Ada packages and tasks; hierarchy and inheritance from object orientation; and the concept of monitors from concurrent programming. However, as Wasserman et al. observe, OOSD does not provide a detailed procedure for developing the design itself.

The primary tool for OOSD is the object-oriented structure chart (see the sidebar, "Tools for Wasserman et al. object-oriented structured design"). This chart takes the symbols and notations from conventional structure charts, including modules, data parameters, and control parameters, and adds notations for such object-oriented constructs as objects and classes (called "information clusters" by the authors), methods, instantiation, exception handling, generic definitions (similar to abstract classes), inheritance, and concurrency. Object-oriented structure charts can be used to show multiple inheritance, message passing, polymorphism, and dynamic binding. OOSD also supports the concept of a monitor, which is useful in depicting the asynchronous processes typically found in real-time systems.

Although OOSD is intended primarily for architectural design, the authors state that OOSD provides a foundation for representing design decisions associated with the physical design. The authors recommend that annotations be used to reflect the idiosyncrasies of individual implementation languages, while preserving the generic character of basic symbols. For example, OOSD includes optional Ada language-specific annotations to provide for packages, sequencing, and selective activation.

Booch object-oriented design. Booch pioneered the field of object-oriented design. As originally defined in the early 1980s, Booch's methodology was Ada language specific, but it has been significantly expanded and generalized since then. Booch views his methodology as an alternative to, rather than an extension of, structured design.

Although Booch describes a host of techniques and tools to assist design, ranging from informal lists to formal diagrams and templates, he is reluctant to prescribe a fixed ordering of phases for object-oriented design. Rather, he recommends that analysts work iteratively and incrementally, augmenting formal diagrams with informal techniques as appropriate to the problem at hand. Nevertheless, Booch does delineate four major steps that must be performed during the course of OOD:

(1) *Identify classes and objects.* Identify key abstractions in the problem space and label them as candidate classes and objects.

(2) *Identify the semantics of classes and objects.* Establish the meaning of the classes and objects identified in the previous step using a variety of techniques, including creating "scripts" that define the life cycles of each object from creation to destruction.

(3) *Identify relationships between classes and objects.* Establish class and object interactions, such as patterns of inheritance among classes and patterns of cooperation among objects. This step also captures visibility decisions among classes and objects.

(4) *Implement classes and objects.* Construct detailed internal views of classes and objects, including definitions of their various behaviors (services). Also, allocate objects and classes to modules (as defined in the target language environment) and allocate programs to processors (where the target environment supports multiple processors).

The primary tools used during OOD are

- class diagrams and class templates (which emphasize class definitions and inheritance relationships);
- object diagrams and timing diagrams (which stress message definitions, visibility, and threads of control);
- state-transition diagrams (to model object states and transitions);
- operation templates (to capture definitions of services);
- module diagrams and templates (to capture physical design decisions about the assignment of objects and classes to modules); and
- process diagrams and templates (to assign modules to processors in situations where a multiprocessor configuration will be used).

(See the sidebar, "Tools for Booch object-oriented design.")

Booch OOD provides the widest variety of modeling tools of the OOD methodologies reviewed here. Although he does not prescribe a fixed sequence of design steps, Booch does provide a wealth of guidance on the design process by describing in detail the types of activities that must be performed and by working through the design of five hypothetical systems from different problem domains.

Wirfs-Brock et al. responsibility-driven design. Wirfs-Brock, Wilkerson, and Wiener developed their responsibility-driven design (RDD) methodology during several years of internal software development experience in various corporate settings. RDD is based on a client-server model of computing in which systems are seen as being composed of collections of *servers* that hold private responsibilities and also render services to *clients* based on contracts that define the nature and scope of valid client-server interactions.

To map these terms to more conventional object-oriented terminology, clients and servers are different kinds of objects, while services and responsibilities correspond to methods. Contracts and collaborations are metaphors for the idea that, to preserve encapsulation, some objects must be willing to perform certain tasks (such as modifying the values of their own internal variables) for the benefit of other objects, and that some kinds of services require several objects to work together to achieve the desired result.

Their methodology is responsibility driven because the focus of attention during design is on contracts between clients and server objects. These contracts spell out what actions each object is responsible for performing and what information each object is responsible for sharing. Wirfs-Brock et al. contrast their approach with what they term data-driven object-oriented design methodologies (they cite no specific authors), which are said to emphasize the design of data structures internal to objects and inheritance relationships based on common attributes. In contrast, the responsibility-driven approach is intended to maximize the level of encapsulation in the resulting design. Data-driven design is said to focus more on classes and inheritance, while responsibility-driven design focuses more on object interactions and encapsulation.

Like Booch, Wirfs-Brock et al. recommend an incremental/iterative approach to design, as opposed to rigid phases with fixed deliverables. RDD provides for a six-step procedure spread across two phases. An exploration phase finds candidate classes, responsibilities, and collaborations. A second analysis phase builds hierarchies, defines subsystems, and defines protocols. The steps are

(1) *Find classes.* Extract noun phrases from the requirements specification and build a list of candidate classes by looking for nouns that refer to physical objects, conceptual entities, categories of objects, and external interfaces. Attributes of objects and candidate superclasses are also identified.

(2) *Find responsibilities and assign to classes.* Consider the purpose of each class and examine the specification for action phrases to find candidate responsibilities. Assign responsibilities to classes such that system intelligence is evenly distributed, behaviors reside with related information, and responsibilities are shared among related classes.

(3) *Find collaborations.* Examine responsibilities associated with each class and consider which other classes are needed for collaboration to fulfill each responsibility.

(4) *Define hierarchies.* Construct class hierarchies for kind-of inheritance relationships such that common responsibilities are factored as high as possible and abstract classes do not inherit from concrete classes. Construct contracts by grouping together responsibilities used by the same clients.

(5) *Define subsystems.* Draw a collaborations graph for the complete system. Look for frequent and complex collaborations and identify these as candidate subsystems. Classes within a subsystem should support a small and strongly cohesive set of responsibilities and should be strongly interdependent.

(6) *Define protocols.* Develop design detail by writing design specifications for classes, subsystems, and contracts. Construct the protocols for each class (the signatures for the messages to which each class responds).

Tools used throughout the design process include

- Class cards (steps 1, 2 and 3);
- Hierarchy diagrams (step 4);
- Venn diagrams (step 4);
- Collaborations graphs (steps 4 and 5);
- Subsystem cards (step 5);
- Class specifications (step 6); and
- Subsystem specifications (step 6).

(See the sidebar, "Tools for Wirfs-Brock et al. responsibility-driven design.")

In advocating an approach that emphasizes the dynamic behavior and responsibilities of objects rather than their static class relationships, RDD provides a significant contrast to Booch OOD and to the OOA methodologies reviewed earlier. Unlike these other meth-

Tools for Booch object-oriented design

Class diagram/template — Shows the existence of classes (enclosed in dotted-line "clouds") and their relationships (depicted by various kinds of directed and undirected arcs) in the logical design of a system. Relationships supported include uses, instantiates, inherits, metaclass, and undefined.

Module diagram/template — Documents the allocation of objects and classes to modules in the physical design of a system. Only needed for languages (such as Ada) that support the idea of a module as distinct from objects and classes.

Object diagram/template — Used to model some of dynamics of objects. Each object (enclosed in solid line "clouds") represents an arbitrary instance of a class. Objects are connected by directed arcs that define object visibility and message connections. Does not show flow of control or ordering of events.

Operation template — Structured text that provides detailed design documentation for operations.

Process diagram/template — Used to show the allocation of processes to processors in the physical design of a system. Only for implementations in multiprocessor environments.

State-transition diagram — Shows the states (depicted by circles) of a class, the events (directed arcs) that cause transitions from one state to another, and the actions that result from a state change.

Timing diagram — A companion diagram to the object diagram, shows the flow of control and ordering of events among a group of collaborating objects.

Tools for Wirfs-Brock et al. responsibility-driven design

Class cards — A physical card used to record text describing classes, including name, superclasses, subclasses, responsibilities, and collaborations.

Class specification — An expanded version of the class card. Identifies superclasses, subclasses, hierarchy graphs, collaborations graphs. Also includes a general description of the class, and documents all of its contracts and methods.

Collaborations graph — A diagram showing the classes, subsystems, and contracts within a system and the paths of collaboration between them. Classes are drawn as boxes. Subsystems are drawn as rounded-corner boxes enclosing multiple classes. Collaborations are directed arcs from one class to the contract of another class.

Hierarchy diagram — A simple diagram that shows inheritance relationships in a lattice-like structure. Classes (enclosed within boxes) are connected by undirected arcs that represent an inheritance relationship. Superclasses appear above subclasses.

Subsystem card — A physical card used to record text describing subsystems, including name and a list of contracts.

Subsystem specification — Contains the same information as a class specification, only at the level of a subsystem.

Venn diagram — Used to show the overlap of responsibilities between classes to help identify opportunities to create abstract superclasses. Classes are depicted as intersecting ellipses.

odologies, the initial steps of RDD do not focus on establishing a hierarchy of classes, but rather attempt to construct a close simulation of object behaviors and interactions.

Comparison of design methodologies

Object-oriented design versus conventional design. The distinctions between conventional and object-oriented development, some of which were identified in the discussion of analysis methodologies, are amplified during design due to the growing importance of implementation-specific issues (see Table 2). None of the conventional methodologies support the definition of classes, inheritance, methods, or message protocols, and while it may not be necessary to consider these constructs explicitly during object-oriented analysis, they form the foundation of an object-oriented design (Table 2, rows 6 through 10). In addition, while conventional and object-oriented methodologies both provide tools that define a hierarchy of modules (row 1), a completely different method of decomposition is employed, and the very definition of the term module is different.

In conventional systems, modules — such as programs, subroutines, and functions — only contain procedural code. In object-oriented systems, the object — a bundling of procedures and data — is the primary unit of modularity. Structured design and information engineering both use function-oriented decomposition rules, resulting in a set of procedure-oriented program modules. OOD methodologies, by contrast, employ an object-oriented decomposition resulting in collections of methods encapsulated within objects.

The greatest overlap between conventional and object-oriented design methodologies is between Booch OOD and information engineering. Both methodologies provide a tool for defining end-to-end processing sequences (row 4), although Booch's timing diagram contains much less detail than information engineering's data-dependency diagram. Both methodologies provide for a detailed definition of procedural logic.

Booch recommends the use of a generic program definition language (PDL) or structured English, while information engineering recommends the use of a graphical action diagram for this purpose. Finally, for information-intensive applications, Booch recommends that a normalization procedure be used for designing data. This normalization procedure is very similar to the one employed by information engineering.

OOD methodology differences. The most notable differences among the three OOD methodologies have to do with

(1) data design,
(2) level of detail in describing the process of OOD, and
(3) level of detail provided by diagram notations.

Booch, as mentioned above, employs a detailed procedure (where appropriate) for designing the data encapsulated within objects. In fact, Booch[2] sees many parallels between database design and OOD:

> In a process not unlike object-oriented design, database designers bounce between logical and physical design throughout the development of the database . . . The ways in which we describe the elements of a database are very similar to the ways in which we describe the key abstractions in an application using object-oriented design.

Wasserman et al. and Wirfs-Brock et al., by contrast, say little on the issue of data design or normalization.

Wirfs-Brock et al. provide a very thorough description of the design process, which they break into 26 identifiable design activities spread across six steps. Booch offers less in the way of explicit, step-wise design procedures, although he does provide a wealth of implicit guidance, using a detailed description of five hypothetical design projects.

Table 2. Comparison of design methodologies.

Component	Yourdon and Constantine Structured Design	Martin Information Engineering	Wasserman et al. Object-Oriented Structured Design	Booch Object-Oriented Design	Wirfs-Brock et al. Responsibility-Driven Design
1. **Hierarchy of modules (physical design)**	**Structure chart**	**Process-decomposition diagram**	**Object-oriented structure chart**	**Module diagram**	***Not supported***
2. Data definitions	Hierarchy diagram	Data-model diagram; data-structure diagram	Object-oriented structure chart	Class diagram	Class specification
3. **Procedural logic**	***Not supported***	**Action diagram**	***Not supported***	**Operation template**	**Class specification**
4. End-to-end processing sequences	Dataflow diagram	Dataflow diagram; process-dependency diagram	*Not supported*	Timing diagrams	*Not supported*
5. **Object states and transitions**	***Not supported***	***Not supported***	***Not supported***	**State-transition diagram**	***Not supported***
6. Definition of classes and inheritance	*Not supported*	*Not supported*	Object-oriented structure chart	Class diagram	Hierarchy diagram
7. **Other class relationships (instantiates, uses, etc.)**	***Not supported***	***Not supported***	**Object-oriented structure chart**	**Class diagram**	**Class specification**
8. Assignment of operations/ services to classes	*Not supported*	*Not supported*	Object-oriented structure chart	Class diagram	Collaborations graph; class specification
9. **Detailed definition of operations/ services**	***Not supported***	***Not supported***	***Not supported***	**Operations template**	**Class specification**
10. Message connections	*Not supported*	*Not supported*	Object-oriented structure chart	Object diagram and template	Collaborations graph

Wasserman et al., by contrast, assume that the particulars of an implementation environment will dictate what kinds of procedures and quality metrics are best; they do not offer a procedural description of OOSD.

Wasserman et al. and Booch both provide a comprehensive and rigorous set of notations for representing an object-oriented design. Wirfs-Brock et al. provide a less detailed notation in their RDD methodology, and do not address such concepts as persistence, object instantiation, and concurrent execution. The authors claim that RDD is appropriate for object-oriented and conventional development projects alike; this may explain the lack of attention to implementation issues that are more closely associated with object orientation.

OOD: Incremental versus radical change. Regarding the incremental-versus-radical debate, object-oriented design is clearly a radical change from both process-oriented methodologies and data-oriented methodologies (Yourdon and Constantine structured design and Martin information engineering, respectively). Table 2 shows that the number of modeling dimensions on which conventional and object-oriented methodologies overlap ranges from a maximum of four out of 10 (information engineering and Booch OOD) to as few as one out of 10 (structured design and Wirfs-Brock OOD). Although conventional methodologies such as information engineering support a data-oriented view in modeling the problem domain during analysis, they use a function-oriented view in establishing the architecture of program modules during design. As a result, not only is the primary structuring principle for program code different — functions versus objects — but at least half of the specific dimensions of the target system model are different.

Object-oriented design requires a new set of competencies associated with constructing detailed definitions of classes and inheritance, class and object relationships, and object operations and message connections. The design tradeoffs between maximizing encapsulation (by emphasizing object responsibilities) versus maximizing inheritance (by emphasizing commonalties among classes) are subtle ones. Designing classes that are independent of the context in which they are used is required to maximize reuse, and here again, very subtle design decisions must be made.[19] As mentioned in the introduction, the important point is not whether object-oriented concepts are radically new in some absolute sense, but rather whether they are radically new to the population of potential adopters. The idea of building systems devoid of global-calling programs, where everything literally is defined as an object, will certainly be a radical concept to designers schooled in conventional design methodologies.

Transition from analysis to design

Analysis is usually defined as a process of extracting and codifying user requirements and establishing an accurate model of the problem domain. Design, by contrast, is the process of mapping requirements to a system implementation that conforms to desired cost, performance, and quality parameters. While these two activities are conceptually distinct, in practice the line between analysis and design is frequently blurred. Many of the components of an analysis model have direct counterparts in a design model. In addition, the process of design usually leads to a better understanding of requirements, and can uncover areas where a change in requirements must be negotiated to support desired performance and cost constraints. In recognition of these realities, most current methodologies recommend that analysis and design be performed iteratively, if not concurrently.

One of the frequently cited advantages of object orientation is that it provides a smoother translation between analysis and design models than do structured methodologies. It is true that no direct and obvious mapping exists between structured analysis and structured design:

> Anyone involved with [structured design] knows that the transition from the analysis model to the design model can be tricky. For example, in moving from a dataflow diagram view of the system to creating design-structure charts the modeler is forced to make a significant shift in perspective. There are strategies to assist in the matter (transform analysis, transaction analysis, etc.), but it remains a difficult task because the mapping is not truly isomorphic.[6]

With object orientation, the mapping from analysis to design does appear to be potentially more isomorphic, as a comparison of Tables 1 and 2 reveals. Every analysis model component supported by at least one OOA methodology can be mapped to a similar (albeit usually more detailed) component supported by at least one design methodology. Rows 1-3, 4, 5, 6, 7, 10, and 11 in Table 1 correspond to rows 6-7, 2, 1, 5, 9, 8 and 10 in Table 2, respectively.

Only two object-oriented methodologists provided detailed procedures encompassing both analysis and design (Coad and Yourdon[20] and Rumbaugh et al.[21]). Shlaer and Mellor also briefly describe a procedure for translating OOA into OOD. Development groups that do not elect to adopt a single methodology spanning analysis and design will face the problem of matching up incompatible terminology and notations from different methodologists. The blurring between analysis and design is a particularly acute issue because the somewhat arbitrary line between analysis and design is drawn in different places by different methodologists. Of the OOA methodologies, Coad and Yourdon's and Shlaer and Mellor's seem to encroach the most on design. Coad and Yourdon explicitly identify inheritance relationships (usually considered a design activity) and provide for a formal and detailed specification of the logic within services. Shlaer and Mellor provide for complete normalization of attributes and advocate detailed modeling of entity life cycles. Of the design methodologies reviewed here, Wirfs-Brock et al. RDD appears to encroach the most on analysis in that it assumes that only an English-language specification (rather than a full-analysis model) is the input to the methodology.

Overall critique

Object-oriented methodologies are less mature than conventional methodologies, and may be expected to undergo a period of expansion and refinement as project experience uncovers gaps in modeling capabilities or misplaced assumptions. Three areas currently stand out as candidates for further development work. To begin with, a rigorous mechanism is needed for decomposing very large systems into components, such that each component can be developed separately and subsequently integrated. Second, tools for modeling end-to-end processing sequences that involve multiple objects are either cumbersome or wholly lacking. Third, in the area of reuse, much is made of designing in reuse ("sowing" reuse), but no more than passing mention is made of techniques or procedures for finding and exploiting existing models, domain knowledge, or components ("harvesting" reuse). The first two areas are ones where object-oriented methodologies lack functionality provided by conventional methodologies, while the third area lacks support in both object-oriented and conventional methodologies.

System partitioning/object clustering. Traditional methodologies, such as structured analysis and information engineering, provide mechanisms for creating a natural, coarse-grained decomposition of systems (nested processes in the case of structured analysis, and subject databases in the case of information engineering). This decomposition is essential because many projects are too large to be developed by a single team within the desired time frame and, hence, must be divided into components and assigned to multiple teams working in parallel. To be most beneficial, the decomposition must be performed early in the development process, which also suggests it must be created in top-down fashion rather than bottom-up. In addition, the decomposition must create natural divisions between components and allow for a rigorously defined process of subsequent reintegration of the components.

The most coarse-grained, formally defined entities in object orientation are objects and classes. While objects and classes certainly provide a powerful mechanism for aggregating system functionality, they are usually defined in a bottom-up fashion as common characteristics get factored to ever higher levels in an inheritance structure. In addition, very large systems, even after this factoring process has been completed, may still consist of hundreds of top-level classes. De Champeaux[22] notes

> While the analysis of a toy example like the popular car cruise control system yields only a "flat" set of objects [classes], the analysis of . . . an airline system or a bank will yield "objects" [classes] at different abstraction levels.

The objects and classes, even at the highest level, are too fine-grained and defined too late in the development process to provide a basis for partitioning large development projects. This limitation has apparently been recognized by several methodologists; they have responded by inventing high-level constructs for clustering related object classes. These constructs include subject areas,[3] domains,[16] systems,[16,18] and ensembles.[22]

Two of these constructs — Coad and Yourdon's subject areas and the Wirfs-Brock et al. subsystems — appear to be very similar conceptually and provide a starting point for partitioning object-oriented models. Yet they are quite informally defined, and they provide little indication of how individually developed system components might interact. Shlaer and Mellor's concepts of domains and subsystems are better developed, and four of the methodology's diagrams are devoted to modeling the interactions between domains and between application domain subsystems (domain chart, subsystem relationship model, subsystem communication model, and subsystem access model).

De Champeaux's *ensembles* and *ensemble classes*[22] are the most rigorously defined of the clustering mechanisms. Ensembles are analogous to conventional objects, while ensemble classes are analogous to conventional classes. An ensemble is a flat grouping of objects (or other ensembles) that naturally go together — usually because they participate in whole-to-part relationships. An automobile, for example, is an ensemble consisting of an engine, doors, wheels, etc. Ensembles have many of the same characteristics as conventional objects, including attributes, states and transitions, and the capability of interacting with other objects and ensembles.

De Champeaux distinguishes ensembles from objects on this basis: Ensembles can have *internal parallelism* while objects cannot. That is, ensembles may consist of subordinate objects or ensembles that each exhibit behaviors in parallel during system execution. Objects, by contrast, are assumed to exhibit only sequential behaviors (for example, as modeled in a finite-state machine.)

De Champeaux distinguishes ensembles from other clustering mechanisms (such as, the Wirfs-Brock et al. subsystems) in that they are more than just conceptual entities; they exist during system execution and may have persistent attributes. It is less clear how ensembles differ from the conventional notion of a compound or composite object,[2] except that ensembles seem to be a more general concept than composite objects. That is, an ensemble might refer to a cluster of related entities, such as a fleet of ships that would not ordinarily be viewed as a composite object in the real world.

Yet, in terms of how they behave, ensembles and composite objects appear to be quite similar. De Champeaux notes that the constituents of an ensemble only interact directly with each other or with the encompassing ensemble. An ensemble hides the details of constituents that are irrelevant outside of the ensemble, and acts as a gateway that forwards messages or triggers to external objects and ensembles. Likewise, Booch recommends that when using composite objects, the encapsulating object should hide the details of the constituent objects and mediate between constituent objects and external objects.[2]

Although De Champeaux's use of ensembles seems promising on a conceptual level, actual project experiences will tell whether or not ensembles provide a practical basis for partitioning large projects. An interesting question for language designers is whether ensembles, or some similar construct, should be explicitly supported, for example, through mechanisms that limit the allowable patterns of interaction between ensembles, their constituents, and external objects to just those envisioned by de Champeaux.

End-to-end process modeling. Many problem domains contain global processes that impact many objects and involve the serial or parallel execution of numerous intermediary steps between initiation and conclusion. Examples of such processes include the ordering process for a manufacturer, daily account reconciliation in a bank, and monthly invoice processing by a long-distance telecommunications carrier. Conventional methodologies provide well-established tools such as dataflow diagrams (see the "Tools for structured methodology" sidebar) and process-dependency diagrams (see the sidebar, "Tools for Martin information engineering") for modeling these sorts of processes.

None of the object-oriented methodologies reviewed here provide a specific model for describing global processes end to end, although individual parts of the process are modeled piecemeal using such concepts as operations,[2,4] services,[3] actions and processes,[16] and responsibilities.[18] (Shlaer and Mellor describe a procedure for following threads of control, but this procedure spans several different diagrams and seems rather cumbersome. In any case, no distinct view of end-to-end processing — devoid of extraneous information — is provided.)

Bailin supports the idea of using dataflow diagrams (and presumably, global process modeling as well) during analysis, but only to help achieve a better understanding of objects. The resulting diagrams serve only as an intermediate representation and are not part of the object-oriented specification.[14]

Booch's timing diagram (see the sidebar, "Tools for Booch object-oriented design") is the closest that any of the methodologies come to supporting a distinct view of end-to-end process modeling. Yet this diagram contains very little expressive power compared with, for example, information engineering's process-dependency diagram. A timing diagram only shows flow of control information, whereas a process-dependency diagram shows flow of control, flow of data, and conditional execution. Bailin also recognizes the need for end-to-end process modeling and has listed composition graphs (similar to timing diagrams in terms of expressive power) as a possible extension of his methodology. (Note that the most recent Bailin methodology refers to compositions graphs as "stimulus-response diagrams.")

This lack of support for global processes is not surprising since the concept of a global process, not subordinated to any individual object, seems to be at odds with the spirit of object orientation. In fact, Booch[2] and de Champeaux[22] both warn against the use of even throwaway dataflow models, for fear that it will irrevocably bias subsequent object modeling towards a "function" orientation.

Still, there is no reason to believe that complicated business processes and the system components that automate them will no longer exist simply because one adopts object orientation. Nor is elimination of end-to-end processes listed by any methodologist as a precondition for adopting object orientation. Thus, it would seem that a separate tool is needed to arrange the mosaic of encapsulated services into a model that illustrates sequencing, conditional execution, and related ideas for certain key global processes.

Harvesting reuse. One of the most persistently claimed advantages of object orientation is that it enables pervasive levels of software reuse. If properly applied, object-oriented mechanisms such as encapsulation, inheritance, polymorphism, and dynamic binding certainly obviate many technical barriers to reuse of program code. In addition, it has been claimed that object orientation opens the way to reuse of design models, or frameworks,[7] and even analysis models from relevant problem domains.[6] At the level of analysis and design, reuse can take two basic forms: reuse of components from previously developed analysis and design models, and reuse of abstractions of previously implemented program components.

Even within an object-oriented implementation environment, achieving high levels of reuse is by no means automatic; virtually all object-oriented methodologists emphasize that reuse must be designed into an application from the start. This emphasis on sowing reuse is not surprising; however, it is curious how little attention object-oriented methodologists pay to harvesting reuse during analysis and design. Analysis and design consume more resources than programming, and perhaps more importantly, development budgets and management decisions — both of which should be strongly influenced by anticipated levels of reuse — are set early in the development process.

Of the methodologies described here, only two address the issue of harvesting reuse from beyond the confines of the project at hand. Coad and Yourdon refer to the need to examine previous analysis models for reusable components and also provide a procedure for merging existing design or program components with new applications.[20] Like Coad and Yourdon, Booch emphasizes the importance of seeking reusable software components from existing class libraries during design. Yet, neither author provides specific guidance on how to find or evaluate existing components.

De Champeaux and Faure[6] and Caldiera and Basili[19] discuss the issue of harvesting reuse at the level of analysis and design. De Champeaux and Faure recommend a repository-based approach to managing reuse. They suggest that the software development process can be seen as a process of creating and modifying three cross-referenced repositories with analysis, design, and implementation components. In this view, the analysis components serve as annotations to the design and implementation components and may point to alternative realizations of the same requirements (for example, with different performance parameters). They further suggest that these annotations could be the basis for a smart library transversal mechanism. This mechanism could assist in identifying candidate reusable components.

Caldiera and Basili provide a much more thorough examination of the issue of harvesting reuse, especially in the areas of identifying and qualifying software components. They suggest a model for project organization where application developers are segregated from "reuse specialists." Reuse specialists work in a "component factory" and are responsible for the development and maintenance of a repository of reusable components. The component factory is responsible for identifying, qualifying, and tailoring reusable components for subsequent integration — by application developers — into ongoing applications development projects.

Object-oriented analysis and design methodologies are rapidly evolving, but the field is by no means fully mature. None of the methodologies reviewed here (with the possible exception of Booch OOD) has — as of this writing — achieved the status of a widely recognized standard on the order of the conventional methodologies of Yourdon and Constantine or DeMarco. Object-oriented methodologies will continue to evolve, as did conventional methodologies before them, as subtler issues emerge from their use in a wide array of problem domains and project environments. As discussed above, three areas — system partitioning, end-to-end process modeling, and harvesting reuse — appear to be especially strong candidates for further development work. In the meantime, adopters of current object-oriented methodologies may need to develop their own extensions to contend with these issues, or alternatively, limit application of the methodologies to problem domains where these issues are of lesser importance.

Compared with object-oriented methodologies, conventional methodologies fall at different places along the incremental-radical continuum. Developers schooled only in structured analysis circa 1978 can be expected to have great difficulty making the transition to OOA, while those with an information modeling background will find much of OOA to be based on familiar concepts.

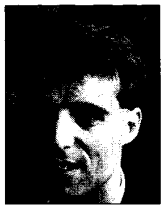
During design, all conventional methodologies revert to a process-oriented view in establishing the architecture of program modules, and as a result, object orientation will likely be viewed as radical change by developers schooled in any of the conventional design methods reviewed here. Since organizations will have to adopt object-oriented design methodologies to end up with object-oriented implementations, a move to an object-oriented environment in general may be seen predominantly as a radical change.

Object orientation is founded on a collection of powerful ideas — modularity, abstraction, encapsulation, reuse — that have firm theoretical foundations. In addition, trends in computing towards complex data types and complex new forms of integrated systems seem to favor the object model over conventional approaches.

Although little empirical evidence exists to support many of the specific claims made in favor of object orientation, the weight of informed opinion among many leading-edge practitioners and academics favors object orientation as a "better idea" for software development than conventional approaches. Organizations that are able to absorb this radical change may well find themselves in a significantly stronger competitive position vis-a-vis those incapable of making the transition. ■

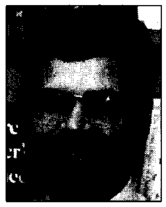
References

1. E. Yourdon, "Object-Oriented Observations," *Am. Programmer*, Vol. 2, No. 7-8, Summer 1989, pp. 3-7.
2. G. Booch, "What Is and What Isn't Object-Oriented Design?" *Am. Programmer*, Vol. 2, No. 7-8, Summer 1989, pp. 14-21.
3. P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd edition, Prentice Hall, Englewood Cliffs, N.J., 1991.
4. A.I. Wasserman, P.A. Pircher, and R.J. Muller, "An Object-Oriented Structured Design Method for Code Generation," *Software Eng. Notes*, Vol. 14, No. 1, Jan. 1989, pp. 32-55.
5. M. Page-Jones and S. Weiss, "Synthesis: An Object-Oriented Analysis and Design Method," *Am. Programmer*, Vol. 2, No. 7-8, Summer 1989, pp. 64-67.
6. D. De Champeaux and P. Faure, "A Comparative Study of Object-Oriented Analysis Methods," *J. Oriented-Oriented Programming*, Vol. 5, No. 1, 1992, pp. 21-33.
7. R.J. Wirfs-Brock and R.E. Johnson, "Surveying Current Research in Object-Oriented Design," *Comm. ACM*, Vol. 33, No. 9, Sept. 1990, pp. 104-124.
8. P.H. Loy, "A Comparison of Object-Oriented and Structured Development Methodologies," *ACM SIGSoft Software Eng. Notes*, Vol. 15, No. 1, Jan. 1990, pp. 44-48.
9. E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Programming and Design*, 2nd edition, Prentice Hall, New York, 1979.
10. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Inc., New York, 1978.
11. P.T. Ward and S.J. Mellor, *Structured Development of Real-Time Systems*, Yourdon Press, Englewood Cliffs, N.J., 1985.
12. E. Yourdon, *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, N.J, 1989.
13. J. Martin, *Information Eng., Books I, II, and III*, Prentice Hall, Englewood Cliffs, N.J., 1990.
14. S.C. Bailin, "An Object-Oriented Requirements Specification Method," *Comm. ACM*, Vol. 32, No. 5, May 1989, pp. 608-623.
15. S. Shlaer and S.J. Mellor, *Object-Oriented Analysis: Modeling the World in Data*, Yourdon Press, Englewood Cliffs, N.J., 1988.
16. S. Shlaer and S.J. Mellor, *Object Life Cycles: Modeling the World in States*, Yourdon Press, Englewood Cliffs, N.J., 1992.
17. R.G. Fichman and C.F. Kemerer, "Object-Oriented Analysis and Design Methodologies: Comparison and Critique," MIT Sloan School of Management, Center for Information Systems Research Working Paper No. 230, Nov. 1991.
18. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, N.J., 1990.
19. G. Caldiera and V. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, Vol. 24, No. 2, Feb. 1991, pp. 61-70.
20. P.Coad and E. Yourdon, *Object-Oriented Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
21. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
22. D. De Champeaux, "Object-Oriented Analysis and Top-Down Software Development," *Proc. European Conf. Object-Oriented Programming, Lecture Notes in Computer Science*, P. America, ed., Springer-Verlag, Geneva, 1991, pp. 360-376.



Robert G. Fichman is a PhD student in the Information Technologies group at the MIT Sloan School of Management. Previously, he was an applications development supervisor at Williams Telecommunications. His research interests include software engineering management, software development methodologies and tools, and technology diffusion.

Fichman received his BS and MS degrees in industrial and operations engineering at the University of Michigan-Ann Arbor in 1982 and 1983, respectively.



Chris F. Kemerer is the Douglas Drane Career Development Associate Professor of Information Technology and Management at the MIT Sloan School of Management. His research interests are in the measurement and modeling of software development for improved performance, and he has published articles in leading academic journals on these topics. He serves on several editorial boards, including *Communications of the ACM*.

Kemerer received his BS degree in economics and decision sciences from the Wharton School of the University of Pennsylvania and his PhD from the Graduate School of Industrial Administration at Carnegie Mellon University. He is a member of the IEEE Computer Society, the ACM, and the Institute for Management Sciences.

Readers can contact the authors at the Massachusetts Institute of Technology, E53-315, Cambridge, MA 02139, fax (617) 258-7579.