# The impact of object-orientation on application development

by A. A. R. Cockburn

***Object-orientation introduces new deliverables, notations, techniques, activities, and tools. Application development consists not only of these items but also of work segmentation, scheduling, and managing the sharing and evolution of deliverables. This paper breaks application development into three major components: construction, coordination, and evolution, with the topic of reuse receiving extra attention. It highlights four aspects of object-orientation having impact: encapsulation, anthropomorphic design, reuse with extensibility, and incremental and iterative development.***

Object-oriented (OO) development is characterized by: (1) the encapsulation of process with data in both the application structure and the development methodology, (2) anthropomorphic design, in which objects in the application are assigned "responsibilities" to carry out, (3) modeling the problem domain throughout development, (4) emphasis on design and code reuse with extensibility, and (5) incremental and iterative development.

Encapsulation, anthropomorphic design, and the new reuse mechanisms give designers a new way of thinking about system decomposition and construction. Encapsulation and subclassing provide an incentive to pay more attention to reuse, affecting the structure of the development organization. Encapsulation and dynamic binding facilitate incremental and iterative development. For all the changes, incremental and iterative development are not newly arrived with object-orientation. They are well-established in modern, non-object-oriented methodologies. Many of the effects described can be felt or applied to non-object-oriented systems.

These areas have an impact on the individual developer, the application development methodology, and the organization developing the OO applications. Of the characteristics mentioned, only extensibility actually relies upon the OO mechanism of inheritance. That means that the effects described in this paper can be applied, or felt, by other systems, notably those that provide coencapsulation of procedure and data (object-based systems).

The effects are far-reaching and have mixed value. Each new mechanism provided by object-orientation requires training and judgment of engineering and business trade-offs. The emphasis on reuse brings difficult reuse issues to the fore, issues such as how to encourage sharing and at the same time how to protect the integrity of software modules.

The first section of this paper introduces the area affected by object-orientation. It is divided into four parts: three for the three major development components of construction, coordination, and evolution, and a fourth for reuse alone. The second section introduces and highlights encapsula-

tion, anthropomorphic design, and the OO mechanisms for reuse with extension. The last four sections examine the impact of those characteristics of object-orientation on the four initial application development topics: construction, including the new methodologies, coordination, evolution, and reuse.

**Object-orientation is apt to affect quite a few aspects of application development.**

## The impact zone

Object-orientation is apt to affect quite a few aspects of application development: (1) the deliverables produced, (2) the activities, techniques, and tools used, (3) the staging and scheduling strategy, (4) segmenting the work, and (5) sharing and controlling the evolution of the deliverables. I place these issues along three dimensions: construction (deliverables, activities, and tools), coordination (scheduling, staging, and work segmentation), and evolution (sharing, control, reuse, and modification). In this paper, reuse receives special attention, since it weaves itself across all three dimensions. Figure 1 captures the three areas in summary form.

**Construction.** A project delivers not only its final product but also models of the product at various levels of detail (requirements, design descriptions, source text, test cases) and various project management deliverables. Each deliverable is described using a notation. Tools are employed to work with each notation, perhaps only paper and pencil, but sometimes specialized CASE (computer-aided software engineering) tools. Discussion of deliverables, then, becomes mixed with the discussion of notation and tools, notwithstanding a desire to keep the three separate. With object-orientation producing different deliverables, the notations and tools will also be different.

The development activities and techniques are tied to the notations and deliverables, and to the tools available. The tools affect not only the way in which developers work, but also the structure of what they produce. Code browsers in the average OO toolkit, for example, facilitate the management of thousands of tiny subroutines (typically 3 to 10 lines long), a task that is prohibitive without them. Coding habits for such small subroutines can afford to be different than those for subroutines averaging 20 to 50 lines.

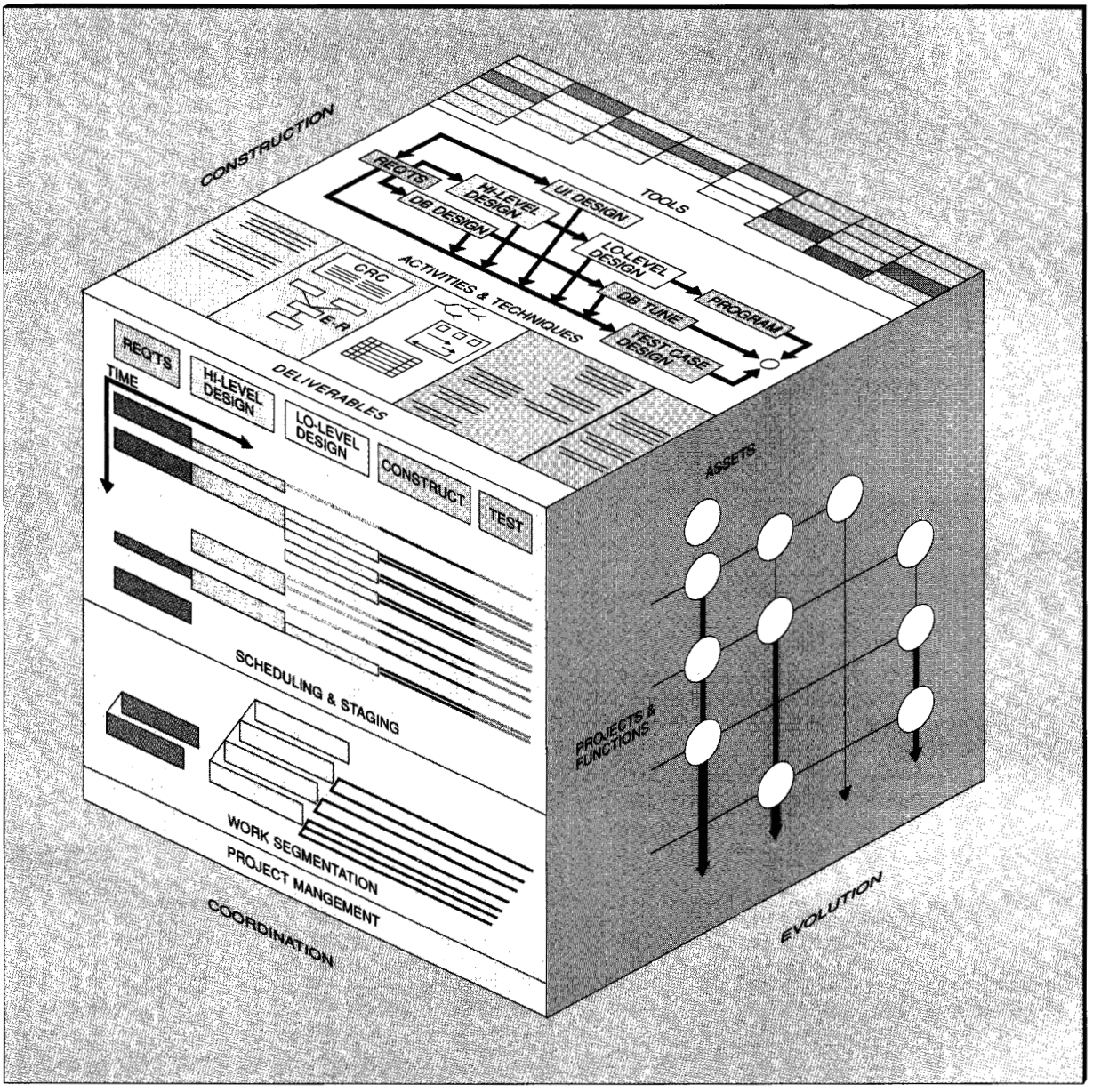
In short, once the deliverables and their notations change, the entire construction portion of a methodology can be expected to change too.

**Coordination.** Coordination includes scheduling, staging, and work segmentation.

*Work segmentation.* For a project with more than one person, the deliverables and work must be segmented and parceled out to separate teams and individuals. If the methodology does not make a statement about how the work will be segmented across the teams, someone in the organization must. A natural way to segment is by deliverables. If the methodology calls for requirements, high-level design, low-level design, and test deliverables, it will be quite natural to expect a requirements team to evolve, as well as a high-level design team, etc. Over time, they will become specialists, reinforcing the work segmentation policy. Note that this is not a necessary work segmentation policy and is possibly not the best one. A known alternative is to use the same team to develop several kinds of deliverables and partition the teams according to time-scheduled units.

*Staging and scheduling deliverables.* Some methodologies address the scheduling of activities and passing on of deliverables. What has been called "Waterfall Development" carries the restriction that all requirements be defined, reviewed and approved before high-level design or analysis can begin; in turn, high-level design must be completed before component design can begin, and so on. Alternatives to this staging and scheduling strategy are the incremental and iterative strategies recommended by the OO community. As is discussed next, the incremental staging and scheduling policies of OO methodologies are basically the same as those of non-OO methodologies. They are, however, more vocally expressed

**Figure 1 Three dimensions of application development**




and perhaps more uniformly practiced in the OO community.

In order to examine the assertion that incremental and iterative development are much the same with object-orientation as before, we must be careful about the terms incremental and iterative. They are used almost interchangeably by some people, possibly for historic reasons. At one time, just allowing a project to pass through requirements and high-level design twice was a major topic for discussion, so iterative was an apt term. Today, numerous strategies involve repetition of phases and are all iterative in some sense. The

meanings of iterative and incremental have evolved to refer to distinct, in fact, independent, development strategies. The definitions that follow do not match the way in which everyone in software development uses the terms incremental and iterative (none could, given the existing conflicts in terminology), and a succinct definition of the terms does not appear in the literature. With this apology, I provide definitions for the terms as they are used in this paper.

*Incremental development* is a scheduling and staging strategy allowing portions of the system to be developed at different times or rates, and integrated as they are completed. *Iterative development* is a scheduling and staging strategy supporting predicted rework of portions of the system. These definitions identify incremental and iterative as independent concepts to be used separately or together. The intent of an incremental strategy is to develop a system piece by piece and to permit additions to the requirements, improvements to the development process, or changes to the scheduling. The intent of an iterative strategy is to allow correction of mistakes and product improvements based upon user feedback, performance tuning, or maintenance criteria, and to allow it in a controlled manner.

Speakers and authors of an incremental methodology sometimes apologize when showing the chain of activities governing the evolution of deliverables. They (accurately) fear in advance that the audience will interpret the diagram as endorsement of "Waterfall Development." In the context of this discussion, waterfall development (not capitalized or in quotes) is defined as a one-pass scheduling and staging strategy requiring a given set of deliverables to pass checkpoints together, with the exact set of deliverables left unspecified. *Any* set of deliverables that must pass checkpoints together follows a waterfall development. This is only natural and needs no apology, since every line of code should have been preceded by design and that by some requirement. It should be unremarkable that a portion of a product undergoes waterfall development within any single iteration. From a project manager's view, waterfall development is simply the progression to completion of each component of the system. It is useful for tracking purposes.

The above definition may be contrasted with "Waterfall Development" (here capitalized and in quotes) referring to when *all* of the requirements must be approved before high-level design begins, etc. A single changed requirement would force all work to grind to a halt while all the deliverables get back into sync. Such a strict interpretation may be so difficult in practice that even groups claiming to use it may actually perform a form of incremental development, just to keep development of the product progressing.[1]

In incremental development, the repetitions of activities address new parts of the system, adding either end function, robustness, error checking, or security facilities (see Figure 2). The essential characteristic of incremental development is that the system is developed in portions. As the portions are completed, they are added to the growing system. There may be a period of evaluation after the integration of one increment and before work on the next increment is begun to gather feedback and new requirements. Alternatively, the increments may be staged in parallel. In a noniterative but incremental project, the increments are developed to full production standards from the start. Further discussion of incremental development is given by Hough[2] and Pittman.[3]

Iterative development no longer just means that an activity is performed multiple times in one project, it indicates that portions of the system undergo rework in a predicted manner. McMenamin[4] wrote:

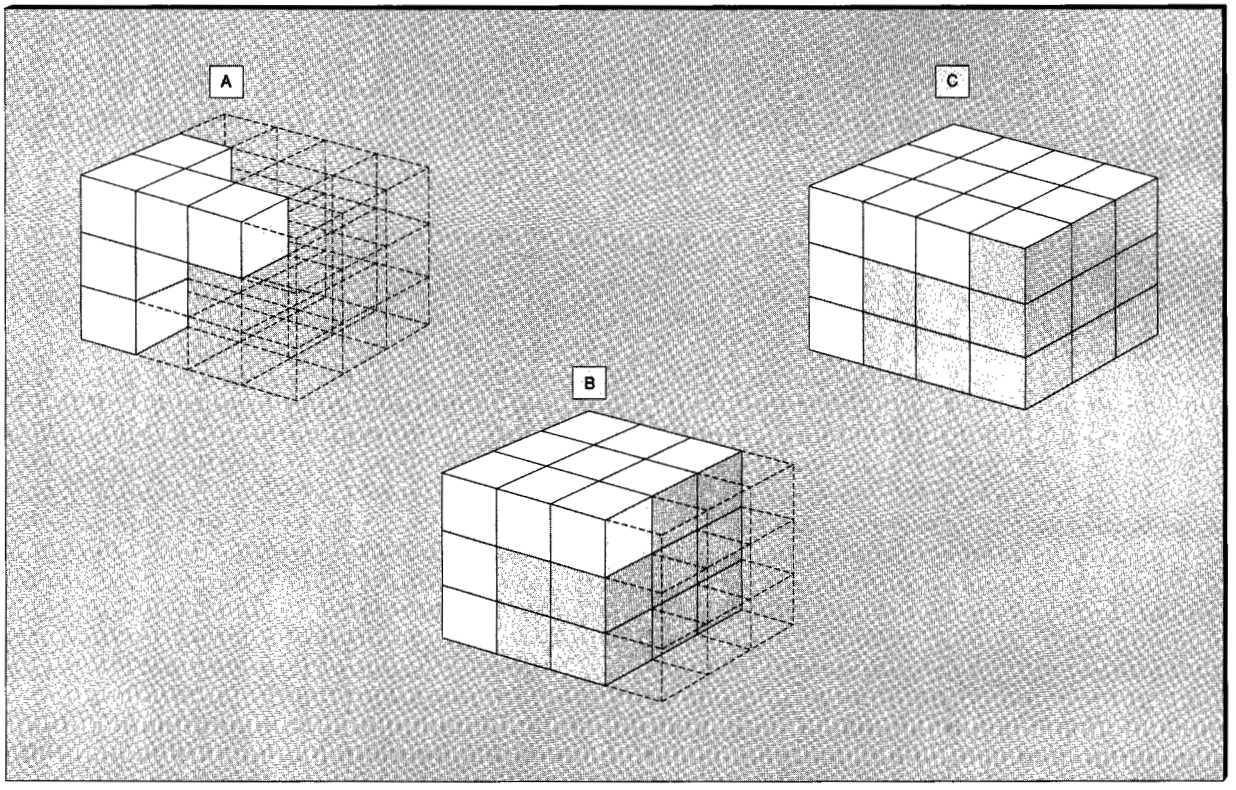
> Iterative refinement accommodates two widespread human traits:
>
> **Misconception** We get things wrong before we get them right.
> **Improvement** We make things badly before we make them well.

and added, verbally,

> When each iteration is considered "rework," that is your clue that iteration is not considered affordable. In truth, iteration *is* rework.

McMenamin's description omits a key element of iterative development: the positive act of discovery. Iterative development allows one to discover new information and improve the design, both positive undertakings. This positive act may require creating a disposable prototype for requirements gathering, altering portions of the sys-

**Figure 2 Incremental development model**




Courtesy Don Hough, IBM Consulting Group

tem following usability tests, or restructuring the system in preparation for evolution.

Iterative development may explicitly call for planning, execution, and evaluation phases within each iteration in order to control the scheduling of rework[5] (Figure 3). In Boehm's "spiral" model,[6] risk management techniques are used to decide what parts of the system need schedule time for preliminary work and rework.

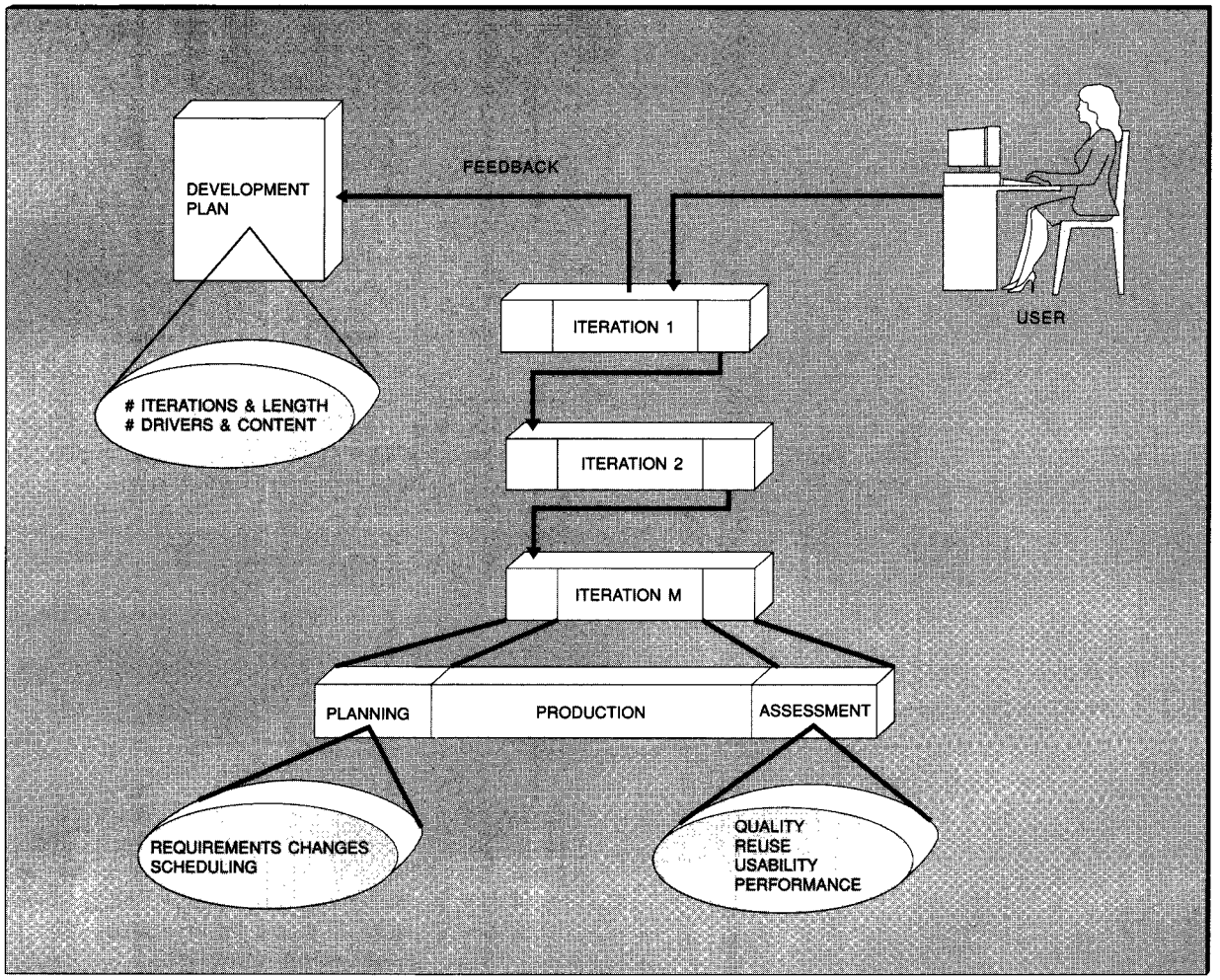
Most phrases describing iterative development give it a somewhat stately and ordered appearance (spiral, fountain, gestalt round trip, etc.). A more evocative and perhaps accurate term was coined by a leader recently reporting on an iterated OO project: "[The] 'Tornado' model has resulted in the highest quality for the least amount of work."[7]

Incremental development may be used with or without iterative development. A project team may plan on an increment reaching shipping quality in a single pass, or it may, in pursuit of quality or because of known risk, plan on an increment being reworked one or more times (developed iteratively). It is important to bear in mind that on a real project, the precise distinction between incremental and iterative is not critical, as long as the project team understands what they are to produce and when.

*Prototyping.* Prototyping is linked with incremental and iterative development. Unfortunately, two very different meanings are associated with the word prototype. A *disposable prototype* is one not intended for production and thus possibly not meeting production quality standards. An *evolutionary prototype* is one incomplete in scope but intended for eventual production, hence meeting production quality standards.

A disposable prototype is useful for gathering requirements and is not intended to evolve to a

**Figure 3 An iterative development model**



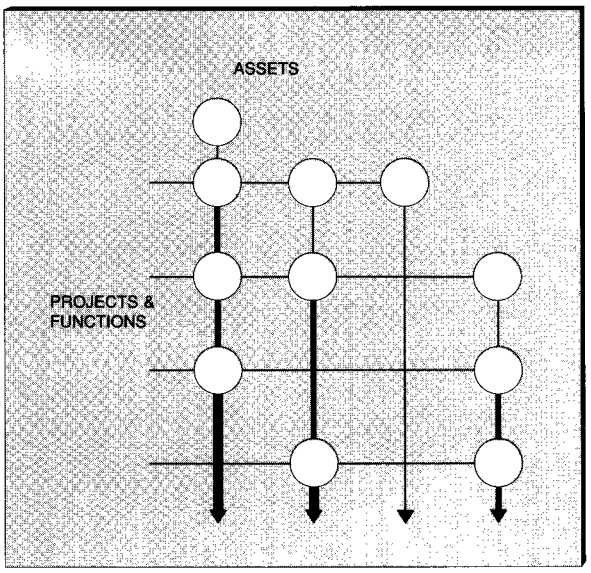

Courtesy Mark Lorenz, IBM OO Technology Council

product. The hazard with a disposable prototype, of course, is that in the pressure to meet deadlines, it may be pressed into service as the final product (at which point it is neither disposable nor a prototype). An evolutionary prototype is one created according to full production standards so that it may evolve to be added to the final product. It may be disposed of in the course of evolution, or it may become one increment of the final system. Evolutionary prototypes are becoming increasingly preferred as organizations suffer maintenance and evolution costs following instances in which disposable prototypes were turned into products.

Most of the issues surrounding prototypes remain the same with object-orientation as without. The distinction becomes significant in the use of subclassing (class inheritance), as is discussed later.

**Evolution.** Some deliverables, such as database definitions, can be identified as organization-critical assets, whose sharing across projects results in cost savings (alternatively, the absence of whose sharing is a major cost contributor). Non-OO methodologies already call for controlled sharing and evolution of data definitions. They rarely, if ever, include a similar emphasis on pro-

**Figure 4 Sharing and evolution of information technology assets**




grams. Most OO methodologies include recommendations for sharing class definitions and objects.

To avoid the cost incurred by inconsistent definitions and bad data in the database, database administrators and support groups are used to support sharing, to ensure integrity, and to control growth of database definitions. In some organizations, database administration even includes control over the data-accessing routines in an effort to see that bad data do not get into the database.[8] It is not odd, therefore, that database administrators have a place in organizations, or that they have started to include data-accessing programs in their domain. It is odd, rather, that the same has not taken place for programs and program designs.

It is quite a challenge to create an organizational structure to promote sharing and to control the evolution of program parts. Creation of this structure may not be so hard for an organization that already is comfortable with shared database definitions but may be for other organizations. That challenge, associated with object-orientation, is really a reuse issue.

The organization as a whole and an individual project have conflicting interests with regard to such assets. It is in the best interests of a project team to have the assets tuned to the specific needs of the project, but it is in the best long-term interests of the organization to maximize the utility of the asset across projects, even those that may be costly to an individual project. Balancing the needs of the organization against the needs of a project motivates creation of an "asset evolution" department or job, in which a person or team is assigned to monitor each asset, promote its use (as opposed to reinvention), and prevent its overspecialization by a single project (see Figure 4).

Two facts should be evident already. An organization using a current non-OO methodology has a challenge in moving to object-orientation. An organization that has not updated its (non-OO) methodology has several additional challenges. For some organizations, object-orientation may be viewed as an incentive to introduce incremental development and asset management.

**Reuse and extensibility.** Extensibility involves creating a new solution from an existing one by programming the differences. It allows economic growth of systems, particularly when the new one varies in only minor ways from an existing one. Extensibility relies upon reuse.

Object-orientation provides three new mechanisms for reuse and extension: classes, inheritance, and polymorphism. In addition to the organizational issues already mentioned, these three mechanisms affect the impact analysis, editing, testing, and installation of changes.

Stevens adds as a reuse factor the degree of coupling in the reuse.[1] If the using component references but does not interact with the reused components, reference and interaction are decoupled. Managing parts whose reuse is decoupled may be less complex since the reuse is guaranteed to be "clean" (see below). Data flow and event flow systems, such as IBM's data flow system DFDM* (non-object-oriented), and Digitalk's PARTS** product (object-oriented), exemplify decoupled reuse. Further discussion of coupled vs decoupled reuse falls outside the domain of this paper since it involves techniques that are independent of object-orientation.

*Traceability and impact analysis.* When a part is to be changed, all occurrences of the part should

be examined for validity and retested. This is easier said than done. The using part often refers back to the used part, but a part almost never refers to the parts using it.

When the reuse is being tracked within one system, tools can be applied to examine the entire system to locate all references to a part in question and construct both forward and backward traces. Some such tools are provided with object-oriented development systems, the class hierarchy browser being an example. As soon as the tracing goes across a wider area, the problem takes on different dimensions. Solving this problem is important but is beyond the scope of this discussion.

Given the difficulty of tracing the effects of a change to a part, a desirable practice is to reference only the minimum necessary part of a reused component, producing a narrow area of impact. Grouping components into subroutine libraries, classes, or include files makes reuse more convenient at a cost: more references than necessary have to be examined for impact and retested (wide area of impact). A conscious exchange of reuse convenience against difficulty of impact analysis is rarely made—both are so difficult that a manager is usually content if either can be improved.

*Single point of evolution. Single point of evolution* is present when a change to a component is automatically reflected in all occurrences of its use. *Multiple points of evolution* is present when a change to all occurrences of a component can only be accomplished by making changes in multiple places.

Subclassing, procedure calls, and data inclusion are examples of single point of evolution. Subclassing is specifically designed to provide access to groups of data and procedure definitions with a single reference. The new part references a base part, and every change to the base part is automatically obtained by the parts using it. An alternative form of reuse is to copy and modify the base part. With copy and modify, when the base part is found to need an update, every copy of what was once the base part must be found and changed individually (hence, multiple points of evolution).

A single point of evolution can be good or bad, depending on how the reused abstraction fits with its new use. It is beneficial if the using part is committed to whatever abstraction is provided by the base part (the base represents "essential commonality" across all of its uses). It is hazardous if the base part represents only coincidental commonality across its uses, because a change to the reused part may break the coincidence and ruin a previously working component. Determining which case applies is part of impact analysis.

**A desirable practice is to reference only the minimum necessary part of a reused component.**

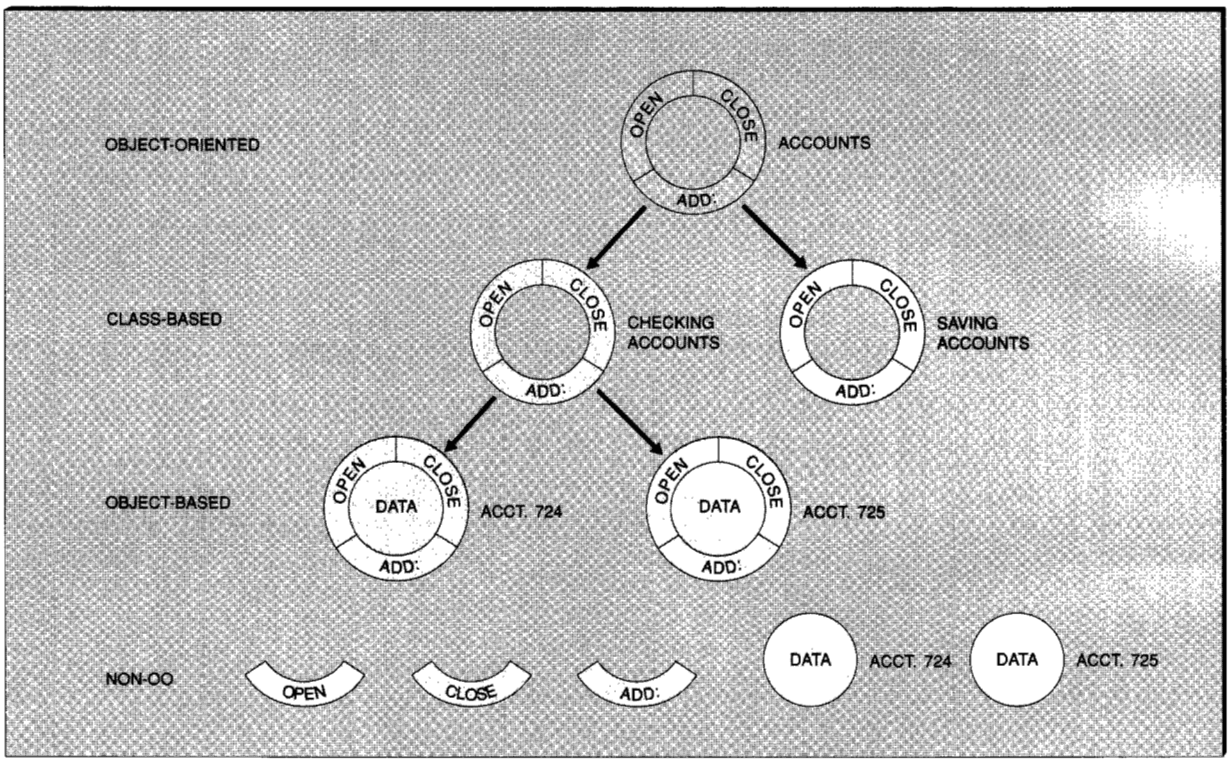
*Clean vs messy reuse. Clean reuse* is reuse in which the reused item needs no retesting (although the new combination of parts needs testing). *Messy reuse* is reuse in which retesting of the reused item is required.

A part used without any modification to its execution or definition need not have its functioning retested (the reuse is clean). As soon as any modification is made to its definition or execution, it becomes a new part and must be freshly tested and maintained (hence, messy reuse). Messy reuse benefits the developer only, and not the test or evolution teams, whereas clean reuse can benefit them all. The usual form of messy reuse is copying and modifying an existing component. It is often faster to copy and modify a component, keeping even just the form, than to write the new one from scratch.

Object-oriented programming gives the ability, through subclassing and overriding, to modify the execution of a component without altering its source text. Subclassing with the possibility of overriding therefore constitutes messy reuse.

Outside of object-orientation, clean reuse correlates with no modification of the base part and therefore the single point of evolution, and messy reuse correlates with modification of the base part

**Figure 5 The three levels of object-orientation**




and multiple points of evolution. Object-orientation combines messy reuse with a single point of evolution. It also allows clean reuse needing multiple points of evolution (using composition of objects).

*Distributing and installing updates.* Once a part is changed, the decision must still be made as to how to install the newly updated component. Ideally, each newly updated base part is considered a new part, and sophisticated tools allow the new base part to be independently accepted or not for each part using the older version.

Perhaps surprisingly, multiple points of evolution actually improves control over installation of the update in the absence of the ideal case. Every separate place in which the update might need to be made provides an opportunity to accept or reject the change. With the single point of evolution, the only recourse is to avoid recompilation, relinking, or reloading (or there may not be any recourse). It is for this reason that a multiple-points-of-evolution solution should not be rejected out of hand.

## The constructs having impact

In the context of this paper, an object is a software packet containing the data and procedures needed to carry out its responsibilities.[9–12] The use of the word "responsibilities" in the definition indicates that the object serves a purpose in the system. It sets the stage for one of the major impacts of object-orientation, the promotion of software modules from merely inanimate to "socially responsible" modules.

The availability of objects, classes, and inheritance is used to distinguish different levels of "object-oriented-ness" (Figure 5). A *class* is a "template for defining the methods and variables for a particular type of object,"[9] a definition carefully couched in implementation terms. A business an-

alyst using the word class will use it in the sense of classification. The classes in the implementation may not correspond to the classes identified by the analyst because of implementation considerations. *Subclassing* (class inheritance) is a "mechanism whereby classes can make use of the methods and variables defined in all classes above them on their branch of the class hierarchy."[9]

The three degrees of object-orientation are object-based, class-based, and object-oriented.[13] Object-based systems allow the coencapsulation of data and procedures, but the objects are not generated from a class. They are individuals. Class-based indicates the use of classes to describe and manufacture objects. The state or data of the object may be individualized, but the methods (procedures) and data definitions are common. Object-oriented uses class inheritance to obtain a single point of evolution on class definitions.

Many of the effects described in the paper stem just from encapsulation. These effects can therefore be found or applied to any object-based or class-based system. There are other effects that are a consequence of inheritance and are not found in object- and class-based systems.

**Encapsulation.** *Encapsulation* is a "modeling and implementation technique that separates the external aspects of an object from the internal, implementation details."[10] Encapsulation is provided by object-based systems.

Encapsulation is not a new concept that has originated with object-orientation. Its principles were described by Parnas in 1972[14] and are present in structured design and numerous non-OO languages, notably Ada and Modula2. Parnas described the ideal in 1976: A program is developed as a family tree in which a design decision creates a branch of the tree, and a particular program can be found by traversing the decisions made. Each design decision is encapsulated and restricted to affecting a small amount of code:

> . . . the design decisions which *cannot* be common properties of the family are identified and a module is designed to hide each design decision . . . Objective criticism of a program's structure would be based upon the fact that a decision or assumption which was likely to change has influenced too much of the code either because it was made too early in the development or because it was not confined to an information hiding module.[15]

Parnas's design evaluation technique has not received a name in the literature, much less been widely practiced, but it might be called *variation analysis*. Variation analysis involves reviewing the effects on the design of predicted changes in the system requirements, either user or implementation requirements. A better (more robust) design will have relatively smaller areas of impact from the changes, indicating reduced risk and simpler evolution. Variation analysis can and is beginning to be practiced meaningfully with object-based and object-oriented systems, since the coencapsulation of procedure and data allows meaningful encapsulation of design decisions.

Objects and classes can encapsulate not only design decisions but also business rules and control processes. Business rules are those parts of the operating practices of an organization that place constraints on data throughout the system. A business rule is a candidate for encapsulation because it captures a design decision on behalf of the business. Ideally, a change to that decision causes changes only to the directly affected objects. It is a candidate for encapsulation for implementation reasons, too. The rule may touch multiple objects and so not properly belong to any one of them. Business control processes, such as how an organization should react to a particular business event, are candidates for encapsulation for similar reasons.

The objects in the software system, therefore, are not just those found as entities in the application data model, but rather are those obtained by considering which aspects of the business and application are worth encapsulating. No doubt other examples of objects that can be created by focusing on encapsulation will occur to the reader. All of the above are available within object-based and class-based systems, as witnessed by Booch's class-based development methodology for Ada.[16]

*Inheritance.* Subclassing is an optimization mechanism that provides a single point of evolution. A different form of inheritance, interface inheritance,[11,17,18] commits a (lower) object or class to providing (at least) the same services as another (higher), without necessarily giving access to the data or code of the higher one to the lower one.

Interface inheritance is for organizing and for specification activities.

Class and interface inheritance differ philosophically and practically. Interface inheritance can be used to obtain a strong form of encapsulation: Stating that one class will provide the same interface as another emphatically does not mean that they share the same representation or implementation. In fact, no presumption about relative implementations can be derived from a relationship between interfaces.[19] Different organizations result from the two—interface inheritance produces an organization of capabilities and stresses substitutability. Subclassing produces an optimization of code; subclasses are not required to be substitutable.

Which inheritance, interface or class, captures the data analyst's and programmer's notion of type? Type is generally understood to mean both subset and substitutable.[20] However, the two are not strictly compatible. Subsetting can decrease the number of imperatives that are valid while increasing the number of queries,[21] whereas substitutability requires an increase or nondecrease in both. For example, a set of squares is certainly a subset of rectangles. Whenever a rectangle is needed, it would seem that a square could be used. However, in programming, a rectangle is given the ability to accept a different aspect ratio, which a square is not, so the square cannot be substituted for the rectangle.

The above discussion carries over into data modeling. *Subtype* in data modeling means "having the same data attributes as . . . ," which is the data version of subclassing. However, it does not translate to either class or interface inheritance and introduces dangers of its own. Suppose employee and customer are modeled as different data subtypes of person. Then an employee who becomes a customer will end up with two name and address records in the database, presenting an exposure to the database on updates.[22] Techniques to address this sort of issue, e.g., dependency modeling,[23] are applicable to both OO and non-OO systems.

*Polymorphism and dynamic binding.* Polymorphism is the ability to serve a common operational purpose in more than one way, using a common interface having more than one implementation. It allows the same name to be given to different methods (procedures) in several different kinds of objects, or even in different parts of the inheritance hierarchy for the same object. *Dynamic binding* is a "form of method resolution that associates a method with an operation at run time, depending on the class of one or more target objects."[10] Polymorphism and dynamic binding can both be made available in object-based systems.

**Polymorphism aids programming by allowing each method to become simpler and more specific in nature.**

We use polymorphism without much notice in our daily life, giving the same name to similar services attached to different kinds of objects, even when the details are different. Answering the door involves quite a different action than answering a phone, letter, or question. So does computing the balance of our checking account as compared to our savings account. We "print" a letter onto a printer or a fax machine, "exit" from an unaltered file or a file that has been changed, etc.

Polymorphism aids programming by allowing each method to become simpler and more specific in nature, hence easier to understand and less likely to contain errors. At the same time, it distributes the total definition of the name over a wider area, making the full meaning of the name more difficult to learn. Dynamic binding extends polymorphic programming, allowing very simple and general algorithms to be written whose exact outcome cannot be determined at the time of program compilation. On the one hand it provides the programmer with flexibility; on the other hand it makes the workings of programs harder to understand.

**Anthropomorphic design.** Taking the step from encapsulation to responsibilities requires a bit of indulgence in anthropomorphism ("attribution of human motivation, characteristics, or behavior to inanimate objects, animals, or natural phenome-

na"[24]). The procedures of the object are called its *services*, and the guarantee that those services will be carried out correctly becomes a *contract*. A contract involves an agreement between objects whereby a service provider promises to deliver the expected results if the service requestor makes requests in prearranged ways.[12,25]

The *responsibility* of an object is "the purpose of an object and its place in the system . . . all the services it provides for all of the contracts it supports."[12] The responsibility is a description of why the object was created (and encapsulated) in the first place. The data and methods of an object help it in carrying out its responsibilities.[26]

**Responsibilities and contracts are part of an anthropomorphic design style.**

The difference between encapsulation and responsibility is one of intent and degree. A data structure may be encapsulated without encapsulating the purpose of the data structure, in which case the encapsulating methods provide little more than debugging assistance or integrity checks on the data (the integrity checks actually may be the purpose of the encapsulation, in which case they are the true responsibility of the object). In paying attention to the responsibility of an object, the designer's thinking is shifted away from just data and processes. It is this shift in thinking that I single out as a major consequence of object-orientation.

Responsibilities and contracts are part of an anthropomorphic design style. Objects may be called actors, having roles. They are given responsibilities and assignments and are required to meet contracts with collaborators. An effective and widely taught OO design exercise for partitioning a system into finer-grained subsystems or objects involves constructing cards detailing the responsibilities and collaborators of each object.[27] According to field reports, people do well with anthropomorphic techniques and produce effective designs using them. Possibly it is because it allows a developer to bring intuitive and social knowledge to bear on the design problem as a supplement to technical skills.

*Messages.* A *message* is a "signal from one object to another that requests the receiving object to carry out one of its methods."[9] Messaging is almost as widely associated with object-orientation as classes and inheritance. Once objects are anthropomorphized to the point of being responsible for services, it becomes natural to talk of sending them messages requesting services (or talk of their sending messages to each other).

One of the benefits of the messaging metaphor is that the definition of message says nothing about whether the signal is traveling within the same machine or across a network, or whether it is blocking or nonblocking. These very real and serious implementation issues can be concealed during initial or general discussions and addressed at the appropriate moment in design.

**Reuse with extension.** Object-orientation provides three kinds of reuse: class libraries, subclassing, and polymorphism.

*Reusing class libraries.* Class libraries are the direct descendant of the venerable run-time libraries. A class is, in effect, an individual run-time library. It provides the benefit of breaking a monolithic collection of thousands or tens of thousands of available methods (procedures) into related clusters indexed by keyword and organized hierarchically. Just by itself, this is a benefit to the developer searching for a particular capability.

Reuse of a class is done by reference, composition, or subclassing. Composition and subclassing are alternative ways to gain the capabilities of one or more base classes for a new class. In composition, the desired portion of the interface definition of the base class is copied and modified as needed. The new object passes along a service request to the base object for service. Composition uses copy and modify reuse in the service interface definition and referencing (method call) in the service implementation. That means a single point of evolution is available for each service, but the combination of services requires multiple points of evolution, since a change to the inter-

face of the base class requires a separate change to the composing class.

*Subclassing.* Subclassing introduces a tension between code savings and evolution issues. Any alteration made to a class is inherited by all of its subclasses. The safest use of subclassing, respecting this single point of evolution, is to ensure that all the methods of the superclass are essential to the abstraction of each subclass. Then, a change to the superclass will logically entail corresponding changes to the subclasses, and the single point of evolution is an advantage. Should a superclass be just a convenient collection of methods momentarily in common, the single point of evolution provided by inheritance is actually a menace: with each change to the superclass, every subclass must be investigated for (un)currency and (in)validity.

Experienced programmers recognize this situation from its counterpart in subroutines: A subroutine provides a single point of evolution for a collection of program instructions. Programmers recognize the difference between essential and coincidental commonality for sequences of instructions and can identify proper vs improper uses of subroutines. In the case of classes, a class collects methods (subroutines) instead of single instructions, but the issue is the same.

During the early phases of development, subclassing can be used to create an initial version of a class with little effort, which may be very useful for evaluating a design decision or pasting together a system for user feedback. The result may be considered a disposable prototype, since the inherited abstraction may not be essential. The correct (and production) system should ideally only use essential abstractions. However, the cost of creating, or even deciding upon, a correct solution may be unacceptably high. This factor introduces the need to make an engineering and economic decision. Commenting on making a mistake using subclassing instead of composing, Dave Thomas of Object Technology International offered the analogy, "Sometimes I welded when I should have used screws."[28] Good programming habits and tools can minimize the inconvenience of moving the class to its proper place.

Three approaches to defending against improper or inaccurate subclassing suggest themselves, depending on the capabilities and personal outlook of the development team: (1) program conservatively, writing to minimize the effect of changes in the class hierarchy and then subclassing with caution; (2) avoid subclassing entirely and build or rely upon programming tools to obtain a similar effect; (3) change the rules of inheritance. This last alternative is not as farfetched as it may seem. The original rules of inheritance exposed the representation of the superclass to its subclasses. In IBM's System Object Model*, the representation of the superclass is concealed from the subclass, although the data and implementations are still inherited.[17] In the more recent CORBA standard,[18] only the interface declarations must be inherited; the implementations may be shared or not.

*Polymorphism and dynamic binding.* Polymorphism allows reuse in two directions, not just the single direction of most reuse mechanisms. Consider an algorithm (e.g., sorting). There are many sort algorithms with fairly complex internal logic. In order for them to work, they need to know only two things about the items being sorted: how to tell which should come first in the sort order, and how to make two items change places. Those two pieces of knowledge can, in principle, be passed as parameters to the sort algorithm. Each algorithm can work with many kinds of data structures, and each kind of data structure can be sorted with many algorithms. In a non-OO programming language, the comparison and exchanging procedures are sometimes passed as parameters to the sort algorithm (not all languages support this). In an object-based or OO language with polymorphism, the comparison and exchanging procedures are associated with the objects being sorted and need not be passed as parameters. The algorithm just instructs the two objects to compare themselves to identify which comes first, and instructs their container object to exchange their places.

Once a polymorphic algorithm works with one class, the developer can use it with other classes that provide the required operations. Any other class that requires sorting need only have the compare operation defined. The effect of polymorphism is that an algorithm can be extended to work with many kinds of objects with little effort, and alternative algorithms will immediately work with all of the objects with which the first algorithm worked.

An illustration of the effect of polymorphism on a product line is the addition of facsimile to the NeXT system.[29] All objects on the system already knew how to respond to the print message for the local printer. The facsimile program was written to take advantage of this knowledge, so that it immediately worked with all objects in the system. Sending an object to the printer makes it print onto paper; sending the object to the facsimile machine makes it "print" onto the phone line.

Polymorphism creates a unique danger for reuse and evolution: inadvertent polymorphism. Inadvertent polymorphism combines with subclassing to produce "superoverriding" (my term): A new superclass method is inadvertently overridden by an existing subclass method, due to an accidental name conflict. Imagine that a subclass defines a method, *m*, not in its superclass. The superclass is changed one day to add *m*, inadvertently using the same name but not the same intent. The code in the superclass references *m* in the mistaken belief that either its own definition or a compatible one will be used. In fact, the method in the subclass overrides the superclass's version of the method inadvertently and subverts the intent of the program. Superoverriding is more likely to occur in type-free OO languages such as Smalltalk and CLOS, less likely to occur in strongly typed languages such as C++** and Eiffel, and is addressed by specific mechanisms in IBM's System Object Model[17] and the CORBA standard.[18]

Inadvertent polymorphism is not a great danger among a small group of developers who can be made aware of one another's changes and adopt naming standards. However, as commerce in class libraries grows, so does the likelihood of inadvertent polymorphism.

*Frameworks.* A framework is a multiclass component.[30] It is a template for a group of objects that manage a responsibility jointly, using a predefined protocol among themselves. The objects exchange carefully structured sequences of messages, called their *protocol*. The framework itself consists of the statement of how the responsibility is divided and the definition of the protocol.

Frameworks offer a level of design reuse above that of classes. A good framework defines a combination of classes and interactions that solves an interesting design problem and can be applied in multiple circumstances.[31] The solution is applied to different circumstances by substituting new classes for member classes. The only requirement of any substituted class is that it must satisfy the contract or protocol. Polymorphism with dynamic binding allows classes to be substituted at run time. Subclassing allows member classes to be tailored, with the subclass using some inherited methods and overriding others. In some cases, frameworks can be used without subclassing.[30]

Frameworks carry forward the goal of Parnas described earlier, the creation of classes of programs with common design decisions. Frameworks are receiving a surge of interest in the design and research communities because of the design savings they offer. Discussion of frameworks is complicated by the fact that no really good documentation techniques have been found for them, something that also makes it difficult to build commerce in frameworks and to learn a new framework.

## The impact: construction

Of the characteristics of object-orientation discussed in the preceding section, some represent normal shifts in technology and vocabulary: objects, classes, subclassing, dynamic binding. The learning curve involved in getting people to master the new vocabulary is not to be minimized. In this section, however, I highlight selected areas of impact: development methodologies, the effects of encapsulation, modeling continuously through implementation, the use of responsibilities, and the effect of the object and messaging models on the design of the user interface.

**The OO methodologies.** To begin the discussion on OO methodologies, this quotation by Ken Orr indicates to some extent the current environment:

> The way to identify an emerging technology is that there is more written about it than known about it, there are more people selling it than using it, and the vendors are making more money from education than from selling the tools.

An evaluation published in late 1992 included 23 OO development methodologies or methodology fragments for comparison.[32] At least four noticeably different methodologies have appeared since that article went to press,[33–36] and more are com-

ing. The methodologies fall roughly into three camps: those that work from commonly practiced non-OO techniques where possible (e.g., data flow diagram decomposition), those that are based on formal models with existing notations (e.g., Petri nets, finite state machines), and those that work from new and uniquely OO techniques (e.g., contracts, responsibility-driven design). Monarchi and Puhr[32] call the three approaches *adaptive*, *combinative*, and *pure-OO*, terms I adopt here.

The single most common characteristic among the methodologies is the call for incremental and iterative development. To some extent, it just represents current thinking applied to a current topic. Some people have expressed the view, however, that OO is sufficiently new, different, and difficult that it is just not practical to plan on getting the system built properly in one pass. Beyond that recommendation, and the need for objects, the methodologies diverge.

Classes and objects are, of course, present, but the notation used for them (and everything else) varies according to the methodology. There is not even agreement on the need for or sufficiency of entity-relationship diagrams or their equivalent as indicated by Rubin and Goldberg:

> . . . we tried to define a small set of relationships between objects and their attributes . . . We concluded that no such small set exists that adds real value to capturing the deep semantic relationships and at the same time can be used by the designers to specify the deliverable system . . . when desired, we can augment these <extensible tables> with diagrams that, in fact, can be generated from the glossaries.[37]

Some authors espouse data flow diagrams for behavioral descriptions,[35] others tolerate them,[10] and others eschew them.[34] Every other piece of notation suffers a similar fate.

With disagreement on the deliverables, there can be no agreement on the activities, techniques, tools, or work segmentation. The lack of consensus does not at all mean the various techniques do not work. It is quite possible that several, quite different, methodologies will prove very successful, although evolving along different paths.

*Pure-OO approaches.* These OO approaches "use new techniques to model object structure, functionality, and dynamic behavior."[32] Included here are Wirfs-Brock et al. (responsibility-driven design),[12] Beck and Cunningham (CRC cards),[27] Booch,[38] Jacobson (use-case-driven design),[34] ParcPlace Systems (object-behavior analysis),[37] and Reenskaug (role modeling).[36] The techniques proposed have no particular relation to the structured techniques most widely practiced in commercial application development houses today.

**The single most common characteristic among the methodologies is the call for incremental and iterative development.**

The authors typically have long experience with building systems in object-oriented languages, going back to Smalltalk and Simula67. They are concerned with strong information hiding ("attributes are logical and not necessarily physical properties"[37]), explanations of why classes exist (e.g., use cases and responsibilities), contracts between classes with respect to goals, roles of objects with respect to each other, etc. This group of methodologies is nicely identified by the acronym, B-O-R-D (behavior, objects, relations, dynamics),[37] showing that behavior comes first and that attributes are a secondary consideration.

Users of the pure-OO approach seem to have little need for many of the current diagramming techniques and corresponding CASE tools (see combinative approaches below). Until quite recently, they worked largely with paper and pencil and a class hierarchy browser, demonstrating, more than anything, the power of those two tools. A notable exception on the topic is Booch, whose method has been criticized as having "too much graphical notation." In keeping with the pure-OO methodologies, though, the notation is all new.

Current tool efforts are focusing on hyper-linked data dictionaries and groups of list boxes with text windows.[34,37] The feeling is, as quoted above, "diagrams can be generated" to "provide a graphical

perspective," but are "not sufficient for capturing the deep semantics." The good news is that tables are a form of documentation with which people work well;[39] the bad news is that the techniques are quite different from those currently in use in many organizations.

*Combinative approaches.* These approaches "use object-oriented, function-oriented, and/or dynamic-oriented techniques to separately model structure, functionality and/or dynamic behavior and provide a method for integrating the different models."[32] The methodologies of Rumbaugh et al. (Object Modeling Technique),[10] Edwards, Martin and Odell (Ptech**),[35] Embley et al. (Object Structure Analysis),[33] Coad and Yourdon,[40] and Shlaer and Mellor (Recursive Design)[41] represent the combinative approach. The emphasis of this group is on producing formal or quasi-formal models of the classes, attributes, relationships, private behavior, and interacting behavior. This group of methodologies is nicely identified by the acronym C-R-A-B (classes, relationships, attributes, behavior). They are distinguished, among other things, by attributes being a first-class interest. Of course, the authors differ on their recommendations for modeling formalisms.

Two attractions of the combinative approach are that the modeling techniques already exist, for the most part, and that the models are formal. There are several dimensions to the modeling. The structural model usually resembles an entity-relationship (E-R) diagram and defines relationships including inheritance, attributes, and method names. The object behavior model may resemble predicate logic, finite state machines, or data flow diagrams (in some cases without any data stores allowed, since accessing a data store is a responsibility of an object). The interaction model may be in data flow diagrams, finite state machines or state charts, event traces or interaction diagrams, or Petri nets.

The attractions of formal models are that they yield an unambiguous description of the system (for whatever portions of the system they capture), and one can build sensible CASE tools for them. One could hope that since existing modeling techniques are being used, existing CASE tools could, too. They can to a limited extent. Handling inheritance and polymorphism requires at least an upgrade. In many cases, the gap cannot be bridged with existing tools, which are then reduced to drawing aids.

The formal nature of the models is part of their drawback as well as part of their attraction. Not every part of the description of a system fits into one of the models—quality requirements and performance and timing constraints being notorious examples. Working with the formal model, even just reading it, takes special training. The different views may not integrate well to form a complete, object-oriented design. It is probably safe to say that formal models are appreciated more by people working on developing a CASE tool or working in an advisory capacity than those in the throes of constructing applications.

*Adaptive approaches.* These approaches "use existing techniques in a new (object-oriented) way, or extend existing techniques to include object-orientation."[32] Bailin, Gorman, Choobineh, Henderson-Sellers, and Constantine and Kappel are among those in this category.[32] They blend use of standard techniques and notations, e.g., essential systems analysis, data flow diagramming, E-R diagramming, with the need to create objects. The challenge faced by this approach is the mismatch between the design produced by the traditional techniques and the object model. Despite algorithms invented for extracting methods from data flow diagrams (DFDs),[42] the essential technique is basically, "read the non-OO design document very carefully; design some OO structures to do that." Notwithstanding the difficulty involved, there are times, even for an OO product, when an E-R/DFD design has already been generated, must be generated, or is conveniently generated due to existing skills. In these situations, the adaptive approach is to be considered.

As a final comment, it must be said that designers and methodologists over the last several decades have discovered good (non-OO) solutions to system development questions, ones that cannot afford to be ignored by the OO community. Until the solutions are transferred, OO developers will grapple with the same problems as their non-OO predecessors.

**Responsibilities.** Designers approaching a design problem who look for objects and analyze for responsibilities follow a different train of thought and probably will produce a different result than other designers. In the B-O-R-D OO methodologies,

an object is given a responsibility to produce a certain piece of information in a certain format on demand. The definition of just what data are attached to the object is considered a local design issue. In the other methodologies, identifying the attributes (object data) and possibly their format is a major design activity.

Responsibilities are a normal way for people to organize systems. For example, employees in the United States have the responsibility of producing their social security number on demand. It

**Responsibilities are a normal way for people to organize systems.**

does not matter whether the number is memorized, emblazoned on their person, or whether the employee has to retrieve a piece of paper or even ask someone else for the number. Actually, these methods are each ultimately only a cache. The *responsibility* of deciding the real number lies with the U.S. Social Security Administration (and just how *it* stores or obtains the information is its own private matter).

Designing with responsibilities in mind produces a change as subtle or dramatic as avoiding the use of data stores in a data flow diagram. According to the responsibility model, no process has the right of direct access to a piece of data; each has the right to request a copy of it in a particular format. One object, somewhere, has control over the actual data. Making data access a service offering means that the actual location and format of the data are hidden from view of the client processes (exemplifying encapsulation). The data access service may freely involve network or database access, data caching, or format conversion algorithms. Of course, these solutions can be designed with other approaches, but designing with responsibilities leads to these solutions in a satisfyingly direct way.

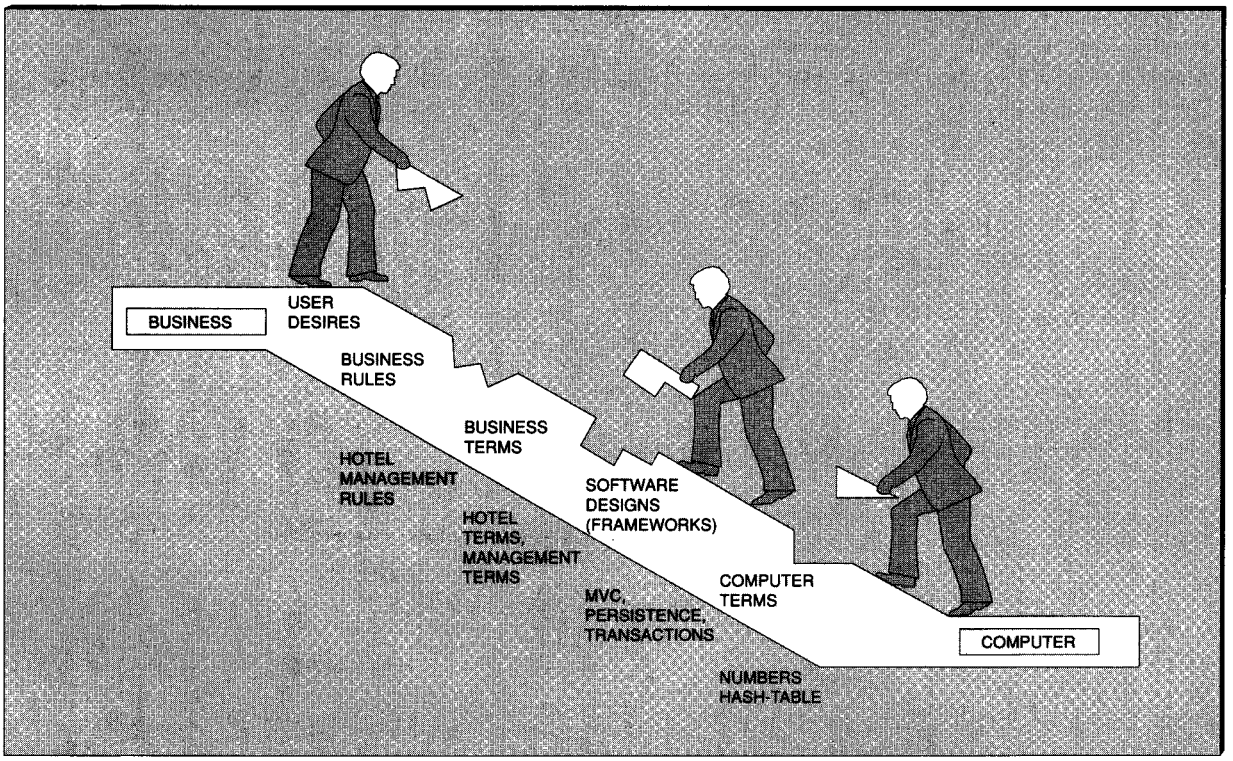
The difference in working with responsibilities goes all the way back to requirements gathering. Knowing that the system will be designed in terms of E-R diagrams and data flow diagrams gives the interviewer the incentive to tune the requirements-gathering questions to data and processes ("What process do you follow when . . . ," "What data appears in the report . . ."). The answers are then funneled along a data development path and a process development path. This method is quite reasonable, unless the system is going to be designed in terms of objects and responsibilities. The object-oriented developer would like to have the requirements gatherer elicit information revealing the objects and their characteristics, whether B-O-R-D or C-R-A-B. In an ideal world, the users' answers are insensitive to the way in which the questions are asked. In the absence of that ideal world, the work of the systems designers is dependent on the training and preferences of the requirements gatherers.

**Encapsulation.** Having a full range of encapsulation affects modeling and the split between programmers and data modelers.

*Modeling the problem domain.* Just as concern with responsibilities affects the design process and the final design, so does the decision to model the workings of the problem domain. Many techniques try to model the problem domain, most of them in business analysis and database design. A characteristic of object-oriented practices is that a serious attempt is made, down to the programming level, to capture the structure and workings of the world. The program designers, as well as the analysts and database designers, worry about the accuracy of the model, along with their other concerns. This attention to verisimilitude is partly a predictable outcome of the pervasive use of encapsulation, partly it is a consequence of the use of anthropomorphic design (responsibilities, contracts, etc.), and partly it is technology facilitating something that has been desired for a long time.[43]

The effects of modeling down to the implementation level are profound but not complicated. The first is that every person who comes in contact with the system, from the first stages of design down to the final programming, obtains a greater knowledge of the user domain. An executable model of the way in which the company does business is provided by I/S, which gives the business person greater visibility into the workings of I/S, and I/S a better understanding of the business needs. Application requirements are

**Figure 6 Classes interlock from problem to solution**




gathered with an increased motivation to get the business users' view of their work. The designers are concerned about the accuracy of the model in addition to its performance, robustness, etc. And when a program fault shows up, the programmers are likely to find themselves arguing about the functioning of the objects in the real world instead of just data fields and program structures.[44]

Objects, classes, and frameworks provide a *vocabulary* for describing the problem. The abstractions that are encapsulated, and the resulting vocabulary, can be taken from various domains: computer programming, design abstractions, business terms, and business rules. The result is a chain of abstractions and vocabulary that bridge the distance from the business problem and user's desire to the capability of the computer (Figure 6). The two ends are given: business needs and the capabilities of the computer. Objects and classes provide the vocabulary in the middle. As more industry-specific and generic business rule classes become available, the gap to be bridged becomes smaller, more people can afford to do their own programming, and the commercial potential of specialized class libraries and business models increases.

The last effect of modeling is on the user interface. The programmer has a direct way to present a set of abstractions and a vocabulary closely matched to the user's interests. Object-orientation is so well-suited to the task that modern user interfaces are almost exclusively object-oriented.[45]

*The data/program development split.* At first glance it would seem that objects, containing both data and program, remove the split between program developers and database developers. Certainly the programmers will be concerned with the relationships between data in new ways. Also, having an OO database implies that the program and data are simply put onto disk and saved. However, there is reason to suppose that this will not be the case in general or in the long run.

Many shops will continue to use their relational, hierarchical, or networked databases for quite some time. They will still need application data models, logical data models, and physical data models, with qualified specialists to refine and maintain them and see that they evolve. That situation reproduces the program-database design split as before.

With or without OO databases, responsibility-based design indicates that there will be special classes devoted to working the database, so that specialized skills in database design are likely to evolve. Both application logic and database specialists may work on classes, so the program-database split may not be visible in the methodology, but it will make little difference in the daily operations of the organization, which will contain the two sets of specialists.

Finally, there is a split between the services provided by the reuse catalog components and those needed by the application. One team is assigned to develop or enhance components of a cross-project nature, and another to develop the application function based on those components (as illustrated in Figure 4). Both groups will produce class definitions, but for different kinds of components, having different probability of reuse. The program-data split then takes on a new form as an *applications-vs-components* split.

### The impact: coordination

Nothing in the definition of object-orientation indicates that any particular staging and scheduling strategy ought to be adopted. As mentioned earlier, both incremental and iterative development strategies are already standard recommendations in the non-OO methodologies. It should be no surprise, therefore, that the OO community also recommends those two strategies. An organization already comfortable with incremental or iterative development should feel little impact here.

Object-orientation actually facilitates incremental and iterative development. The encapsulation provided by objects and the dynamism provided by dynamic binding make the ongoing addition and modification of the software relatively less burdensome and less error prone.

Object-orientation does add two wrinkles to the smoothness of development. One is the need to restructure the class hierarchy to improve the placement of classes and functions. Experienced OO developers expect to restructure the inheritance hierarchy as the project progresses and as they gain experience. The new structure improves reuse potential and evolution. It is no news that a new design will need revision. It is, perhaps, news that the designers recognize at the beginning that even their best attempts will warrant revision. Again, any risk-based management technique allows for iteration once the risk is identified. It is still rare to find a group that admits that their design will need revision and plan for it as part of the project.

Knowing in advance that the class hierarchy will change puts pressure on two places: the schedule and the coding conventions. Time should be allocated at the beginning for restructuring the class hierarchy. Such allocation may be considered preparation for evolution and reuse. Similarly, coding conventions should be adopted at the start that will minimize the dependency on functions and classes being at particular points in the hierarchy (e.g., only one method ever touches any given state data; all other methods use that one access method[46]).

The other wrinkle is parallelism in development due to use of frameworks as prestructured designs. A good software designer, object-oriented or not, designs a system in such a way as to promote parallelism in the development process. Frameworks provide a vehicle to formalize and replicate some of those design decisions. The structure provided by a framework need not be new (in fact, quite the reverse: it is likely to be a distillation of previous, successful designs), but it can be put into place as a matter of routine. The framework neatly extracts risky or highly variable subsystems, which can then follow an iterative development path in relative isolation.

### The impact: evolution

**Controlled sharing and evolution.** Objects, classes, and frameworks are organizational assets as are database definitions and data, and the general technique of managing their evolution is the same: Some specific person or team should be placed to promote the sharing of classes and frameworks across projects and to ensure that no single project overspecializes the behaviors of the classes to its specific needs.

The idea that a project or function owner may not be able to get certain behaviors put into a class raises an issue: Where should those behaviors then be put? Different variants are possible, all of which boil down to the idea that the available reusable classes may not completely cover the needed functionality of the application. One possible approach is to simply leave the behavior unattached. Not all methodologies or languages require every behavior to be attached to a class (e.g., Ptech[35] and C++, respectively). Even if they do, the attachment of those behaviors to classes may be considered an implementation issue. If they are put into a class, it may be a special, application-dependent class. As the system evolves, some of that behavior may be found to be more general and moved to a shared class. In all cases, it must be deliberately managed.

**Subsystem strategy is to choose a subsystem of the application that can use the strengths of object-orientation.**

Not everyone agrees that evolution of assets can or should be centrally managed.[47] In a large company or across organizations, there is simply no chance to collect all of the developers affecting or affected by a class (or any other software asset). Easy as it is to state this objection, a workable solution is not at hand, although some initial architectural research is being done.[48]

**Strategies for migrating to object-orientation.** Object-orientation can be introduced by subsystem, by language feature, or by methodology feature. The strategies appropriate for different organizations are different.

The subsystem strategy is to choose a subsystem of the application that can use the strengths of object-orientation. That might be the user interface subsystem or some section that can take advantage of inheritance or polymorphism. Moving one subsystem at a time to object-orientation means that relatively few people need to be trained at one time, so they can be trained by an expert, or mentor. Once they become experts they can be dispersed to lead other designs or to carry on with the evolution of the subsystem. The new experts become mentors to other groups, and the process repeats itself on a growing scale. The subsystem can be designed in a fully OO way, using a "pure" OO methodology, language, etc., because the scale of the training is relatively small.

A second choice, one that delays the need for complete OO training, is to use a hybrid OO language that is fully compatible with its non-OO counterpart. With this strategy, the existing code base is maintained, new developers do not have to learn all about object-orientation at one time, new subsystems do not have to be completely object-oriented in design, and the existing methodology can be used to start with, i.e., nothing need change, except what the team decides to change. OO methodology, design techniques, and language features are introduced slowly on a person-by-person or team-by-team basis. Note that in selecting this option, an organization foregoes many of the advantages of object-orientation in exchange for a less traumatic and extended educational period.

The third choice is to create OO designs in stages, using staged OO facilities in the methodology. This choice is the architectural correspondent to the previous strategy and requires a hybrid methodology fully compatible with its non-OO counterpart, probably to be found in the adaptive and combinative groups of OO methodologies. The design can be implemented in a hybrid language, or there can be a jump from non-OO to fully OO at some stage in the evolution of the system. At some stage, the architectural description may pass through a point allowing either OO or non-OO implementation. Such a design must contain sufficient information for the inheritance hierarchies to be designed but described in such a way as not to require inheritance or removal of inheritance in the design. This third strategy is very attractive, but has not been demonstrated.

**Legacy systems.** No strategy for migrating would be complete without handling legacy, or pre-existing systems. In object-orientation, any system or subsystem at all can be considered a unique instance of its class. All functions at the interface

**Table 1 OO reuse mechanisms and reuse issues**

| Type of Reuse | Reusing | Clean or Messy | Single or Multiple Point of Evolution | Source Modified | Traceable | Area of Impact |
|---|---|---|---|---|---|---|
| Subclassing, no polymorphism | class | clean | single | no | yes | wide |
| | method | clean | single | no | yes | wide |
| Subclassing, with polymorphism | class | clean | single | no | yes | wide |
| | method | messy | single | no | yes | wide |
| Object composition | class | messy | multiple | yes | no | wide |
| | method | clean | single | no | yes | wide |
| Copy and modify | either | messy | multiple | yes | no | narrow |
| Method call | method | clean | single | no | yes | narrow |
| Polymorphism | method interface | messy | multiple | yes | no | narrow |
| | method content | clean | single | no | no | narrow |

of the system are considered as its services (methods). To interface an OO system with an existing non-OO system, one or more new classes are created with methods that correspond to the services of the non-OO system. Several classes may be created to jointly provide the full service set of the old system so as to provide continuity for when the old system is broken into smaller pieces and rewritten. OO messages are converted (within the method) to call the non-OO services in the legacy system.

The new class is referred to rather grandiosely within the OO community as a wrapper for the non-OO system. The term "peephole" would be more appropriate, since the new code only affects how the OO portion views the rest of the system. It is called a wrapper because, from the point of view of the OO system, it encapsulates the old system, hiding the details from sight. Once wrapped, the old system can be left in peace or replaced over time (presumably by objects).

The above technique is applicable at any scale. It has been used to reverse engineer and upgrade portions of large systems.[49] At the other end of the scale, the New World Infrastructure system[50] provides fine-grained wrappers at the class and individual object level. The internals of these fine-grained objects can be written in an OO or non-OO language, allowing non-OO programmers to participate in the production of an OO system.

## The impact: reuse

**The reuse issues revisited.** The reuse and extension mechanisms offered by object-orientation interact with the basic issues of reuse in rather surprising ways. Table 1 summarizes the effect of six different reuse mechanisms.

Subclassing obtains the entire interface and all implementation, so any change at all to the superclass forces all subclasses to be completely re-evaluated, even if none is actually affected. In the table, that action is considered to be a wide area of impact. Subclassing is usually traced in both directions by the standard OO programming environment.

Subclassing without polymorphism is a clean, traceable form of reuse with the single point of evolution. However, subclassing without polymorphism is generally considered uninteresting, for whatever reason, and no serious attempts have been made to remove polymorphism from inheritance to improve security for

evolution. Subclassing with polymorphism constitutes messy reuse, because the subclass can redefine the meaning of the methods of its superclass.

Object composition is sometimes presented as being cleaner than subclassing. It uses method call for each method, but copy and modify for the interface definition. It is clean reuse with the single point of evolution for the method implementations, but messy reuse with multiple points of evolution for the interface definition. The area of impact is wide because any change to the reused class forces a re-evaluation of the entire using class.

Copy and modify is the standard of messy, untraceable reuse with multiple points of evolution (which does not mean it is not useful).

A method call, like a subroutine call, provides clean, traceable reuse with the single point of evolution and narrow area of impact. It is useful for obtaining the implementation of a single method and does not address obtaining the class interface.

Polymorphism splits up the implementation of a method. The definition of the interface of the method is spread over multiple locations, having to be mutually consistent. The interface definition is copied, making for messy reuse of the interface definition. Changing the interface or intent of the polymorphic method forces updates in multiple places, making for multiple points of evolution. Each method content is obtained using a method call, which is clean and provides the single point of maintenance.

Polymorphism with dynamic binding does not produce a clear trace. It presents a significant challenge to impact analysis, since the addition of a new polymorphic method to a system results in a combinatorial explosion of possible interactions between classes and methods. Fortunately, the area of impact is narrow (only methods using a particular name can be affected), and tools can search the entire reachable system to locate all parts of a polymorphic method. The difficulty involved with polymorphism can be expected to grow with a commerce in class libraries. What is perhaps surprising is that catastrophic errors from this source have not been reported (yet).

**Project management.** With the availability of precut classes and design frameworks, significant effort goes into evaluating ways to use and extend the components, and which mechanisms to use. Multiple departments may become involved because of the evolution issues. Further, sharing components during their development is fairly complex. An object relies upon other objects for services, and typically, several mutually dependent classes undergo design concurrently. Each layer of objects provides services to objects above and relies upon services from objects below. All of this acts to complicate project management.

When serious attention is given to reuse, more time than previously is spent studying the components catalogs, seeing what parts are available and what their detailed capabilities and restrictions are. The positive aspect is that reusing previous solutions often takes considerably less time than construction from scratch.[44] The negative aspect is that while studying catalogs for reusable components, the designers are not actively designing. In terms of lines of code per day, the bad news is that fewer lines of code are produced per day; the good news is that fewer lines of code are needed.

This poses a challenge for schedule estimation. Fortunately, when incremental development is used, the development process itself provides a mechanism for developing a schedule estimation history and a chance to improve its accuracy.

A new project may receive a time estimate in any number of ways. Common sense and experience from previous, non-OO projects may be the best.[51] Another is to estimate the number of new classes that will have to be created and the number that will have to be learned. The initial estimate is not as important as what happens with the estimates during the project. From increment to increment, the estimation basis is improved by learning from the previous increments. This estimation history has the advantage of rating the same people using the same tools on the same problem.

Note that the above discussion focuses on schedule estimation and does not suggest measuring programmer productivity on the basis of a volumetric measure such as lines-of-code. To do so would work against reuse. Of the two, volume and reuse, achieving reuse is the more important.

## Summary

Object-orientation revitalizes a two-decade-old goal with a different mind-set backed by different programming mechanisms. The goal is the encapsulation of design decisions, the mind-set is anthropomorphic design, and the mechanisms are objects, classes, inheritance, polymorphism, and dynamic binding. Objects provide the developer with a full range of program modules, from data-only, through data-plus-function, to function-only. The result is the ability to model business domains, business rules, and software architectures, preserve the model within the computer, and present it to the user at the user interface. Anthropomorphic design allows developers to use their own intuitions and experience with social organizations as well as to use analytical techniques.

The good news associated with object-orientation is that it facilitates the use of incremental and iterative development strategies, which have been recommended for years. Organizations already using these strategies are in a comfortable position with regard to this aspect of object-orientation.

Object-orientation brings with it multiple new mechanisms for reuse, complicating decisions about which to use, but refreshing interest in attaining significant amounts of reuse. Classes and frameworks are development units that can be treated as organizational assets to be shared and protected. The organization with a good handle on shared data assets has an advantage in learning to work with the new ones.

A major cost of moving to object-orientation is training. The construction aspect of OO development is very different from the way in which it was done before. Several dozen new methodologies have appeared, with little appearance of reaching consensus, although trends are emerging. An additional complexity is project management, given the interdependence of multiple classes undergoing development at the same time, and the absence of a base of experience in project estimation.

Two strategies are available for migrating to object-orientation: moving a few people completely, or many people slowly. The former results in the full benefits of object-orientation in a few places, the latter reduces the trauma of retraining at the expense of some of the benefits of object-orientation. Legacy systems can coexist with or slowly be rebuilt to become object systems through the use of object peepholes, also known as wrappers.

Finally, many of the impacts associated with object-orientation are not dependent on inheritance, the characteristic that fully separates object-oriented systems from object-based and class-based systems. Therefore, the effects described can be found or felt in object-based and class-based systems.

## Acknowledgments

The members of the Methodology Department of IBM's Application Development Consulting Practice spent a good deal of energy bringing me up-to-date on non-OO application development. Wayne Stevens of IBM and Dick Antalek of American Management Systems deserve special mention for their discussions, insights, and careful readings of the paper. In the object-oriented community, Dave Thomas of Object Technology International and Tom Morgan of Brooklyn Union Gas helped both fire up and temper my views. The authors cited in the references and members of the organizations I visited contributed experiences and insights. My family provided patience and support, and Sean (age two) instructed me to be happy.



## Cited references and note

1. Observation of Wayne Stevens, IBM Consulting Group, Southbury, CT.
2. D. Hough, "Rapid Delivery: An Evolutionary Approach for Application Development," *IBM Systems Journal* **32**, No. 3, 397–419 (1993, this issue).
3. M. Pittman, "Lessons Learned in Managing Object-Oriented Development," *IEEE Software* **10**, No. 1, 43–54 (January 1993).
4. S. McMenamin, "The New Economics of Requirements Modeling and Prototyping," *The First National Conferences on Software Methods*, Orlando, FL, presented by Technology Transfer Institute, 741 Tenth Street, Santa Monica, CA 90402 (March 30, 1992), Section 9.362.
5. M. Lorenz, *Object-Oriented Software Development: A Practical Guide*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1992).

6. B. Boehm, "A Spiral Model of Software Development," *Computer* **21**, No. 5, 61–72 (May 1988).
7. McGehee, *Corporate Infusion of Object Technology: The "Big-Bang" Approach*, Texas Instruments Microelectronics Manufacturing Science and Technology, presented at Experience Session, *OOPSLA '92*.
8. Observation of Dick Antalek, then at AMS, currently at IBM Consulting Group, Bethesda, MD.
9. D. Taylor, *Object-Oriented Technology: A Manager's Guide,* Servio Corporation, Alameda, CA (1990).
10. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1991).
11. A. Snyder, "The Essence of Objects: Concepts and Terms," *IEEE Software* **10**, No. 1, 31–43 (January 1993).
12. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
13. P. Wegner, "Dimensions of Object-Based Language Design," *OOPSLA 1987, SIGPLAN Notices* **22**, No. 12, 168–182 (1987).
14. D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* **15**, No. 12, 1053–1058 (December 1972).
15. D. Parnas, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering* **2**, No. 3, 1–9 (March 1976); reprinted in *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, Springer, New York (1978), pp. 429–447.
16. G. Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering* **12**, No. 2, 211–221 (1986).
17. *System Object Model Guide and Reference*, No. 1066309, first edition, IBM Corporation (December 1991).
18. *The Common Object Request Broker: Architecture and Specification*, OMG Document 91.12.1, Rev. 1.1, Object Management Group, 47 Walnut, Suite 206, Boulder, CO 80301 (December 1991).
19. Observation of Bill Harrison, IBM Research, Yorktown Heights, NY.
20. B. Liskov, "Data Abstraction and Inheritance," *OOPSLA '87, Addendum, SIGPLAN Notices* **23**, No. 5, 17–34 (May 1988).
21. A. Cockburn, "Using Natural Language as a Metaphoric Base for Object-Oriented Modeling and Programming," *Addendum to OOPSLA 1992 Proceedings*, to appear. Full text in IBM Technical Report TR-36.0002, IBM Corporation (1992).
22. Observation of Ghica van E.nde Boas Lubsen, IBM European Systems Architecture and Technology, Uithoorn, Netherlands.
23. *Business System Development Method, Business Mapping Part 1: Entities*, No. SC19-5310-00, IBM Corporation (1992); available through IBM branch offices.
24. *American Heritage Dictionary*, second College Edition, Houghton Mifflin Co., Boston (1976).
25. R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *OOPSLA '90 Proceedings, SIGPLAN Notices* **25**, No. 10, 169–180 (October 1990).
26. R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," *OOPSLA 1989, SIGPLAN Notices* **24**, No. 10, 71–76 (October 1989).
27. K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *OOPSLA 1989, ACM SIGPLAN Notices* **24**, No. 10, 1–6 (October 1989).
28. Dave Thomas, Object Technology International, Ottawa, Ontario, personal communication.
29. "Software Made Simple," *Business Week* (September 30, 1991), pp. 92–97.
30. R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming* **1**, No. 2, 22–35 (June/July 1988).
31. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," *ECOOP'93* (July 1993), to appear.
32. D. Monarchi and G. Puhr, "A Research Typology for Object-Oriented Analysis and Design," *Communications of the ACM* **35**, No. 9, 35–47 (September 1992).
33. D. Embley, B. Kurtz, and S. Woodfield, *Object-Oriented Systems Analysis and Specification: A Model-Driven Approach*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1992).
34. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering*, Addison-Wesley Publishing Co., Wokingham, England (1992).
35. J. Martin and J. Odell, "Object-Oriented Analysis and Design," Prentice-Hall, Inc., Englewood Cliffs, NJ (1992).
36. T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Lnadmark, O. Lehne, E. Nordhagen, E. Näss-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet, "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems," *JOOP* **5**, No. 6, 27–43 (October 1992).
37. K. Rubin and A. Goldberg, "Object Behavior Analysis," *Communications of the ACM* **35**, No. 9, 48–62 (September 1992).
38. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing Co., Redwood City, CA (1991).
39. G. Cantin, *The Use of Function-Tables to Specify Services and Protocols*, Technical Report 91-236, Telecommunications Research Institute of Ontario, McMaster University, Hamilton, Ontario (1991).
40. P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press, Englewood Cliffs, NJ (1991).
41. S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis, Modeling the World in Data* (1988); *Object-Oriented Systems Analysis, Modeling the World in States* (1992), Yourdon Press, Englewood Cliffs, NJ.
42. M. Branson, E. Herness, and E. Jenney, "Moving Structured Projects to Object Orientation," *Proceedings of the Second Annual Repository AD/Cycle International Users Group Conference*, Chicago (October 1991), pp. 151–170.
43. M. Jackson, *System Development*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).
44. J. Davis and T. Morgan, "Object-Oriented Development at Brooklyn Union Gas," *IEEE Software* **10**, No. 1, 67–75 (January 1993).
45. R. Berry and C. Reeves, "The Evolution of the Common User Access Workplace Model," *IBM Systems Journal* **31**, No. 3, 414–428 (1992).
46. E. Klimas, D. Thomas, and S. Skublics, *Smalltalk with Style*, Addison-Wesley Publishing Co., Reading, MA, to appear.
47. W. Harrison and H. Ossher, *Extension-by-Addition: Building Extensible Software*, Research Report RC-16127, IBM Corporation, Yorktown Heights, NY (1990).

48. W. Harrison, M. Kavianpour, and H. Ossher, "Integrating Coarse-Grained and Fine-Grained Tool Integration," *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering* (CASE '92), (July 1992), pp. 23–25.
49. I. Jacobson and F. Lindström, "Reengineering of Old Systems to an Object-Oriented Architecture," *OOPSLA 1991, SIGPLAN Notices* **26**, No. 11, 340–350 (November 1991).
50. New World Infrastructure was jointly developed by IBM and Softwright Systems, Ltd.
51. F. Bergeron and J-Y. St.-Arnaud, "Estimation of System Development Costs Should Be Done Differently at Each Phase of Development," *Really Useful Research* **1**, No. 6, 7–8 (August 1992), RUR Publishing, 212 E. Ontario St., Chicago, IL 60611. Extracted from "Estimation of Information Systems Development Efforts," *Information & Management* **22**, No. 4, 239–254 (April 1992).

*Accepted for publication March 22, 1993.*

**Alistair A. R. Cockburn** *IBM Consulting Group, 150 Kettletown Road, Southbury, Connecticut 06488.* Mr. Cockburn is the focal point for object-orientation in the Methodology Department of IBM's Worldwide Application Development Consulting Practice. He continually works to reduce the impedance mismatch between people and computers. He has been a designer, programmer, and researcher in computer graphics, software techniques, and the human interface aspects of program specification. He published over a dozen technical papers in these areas and coined the phrase "computational rhetoric." Mr. Cockburn obtained an M.S. in computer science from Case Western Reserve University in 1975. He enjoys sitting under water, learning, and dancing and has forgotten as many languages as he still speaks.

Reprint Order No. G321-5519.