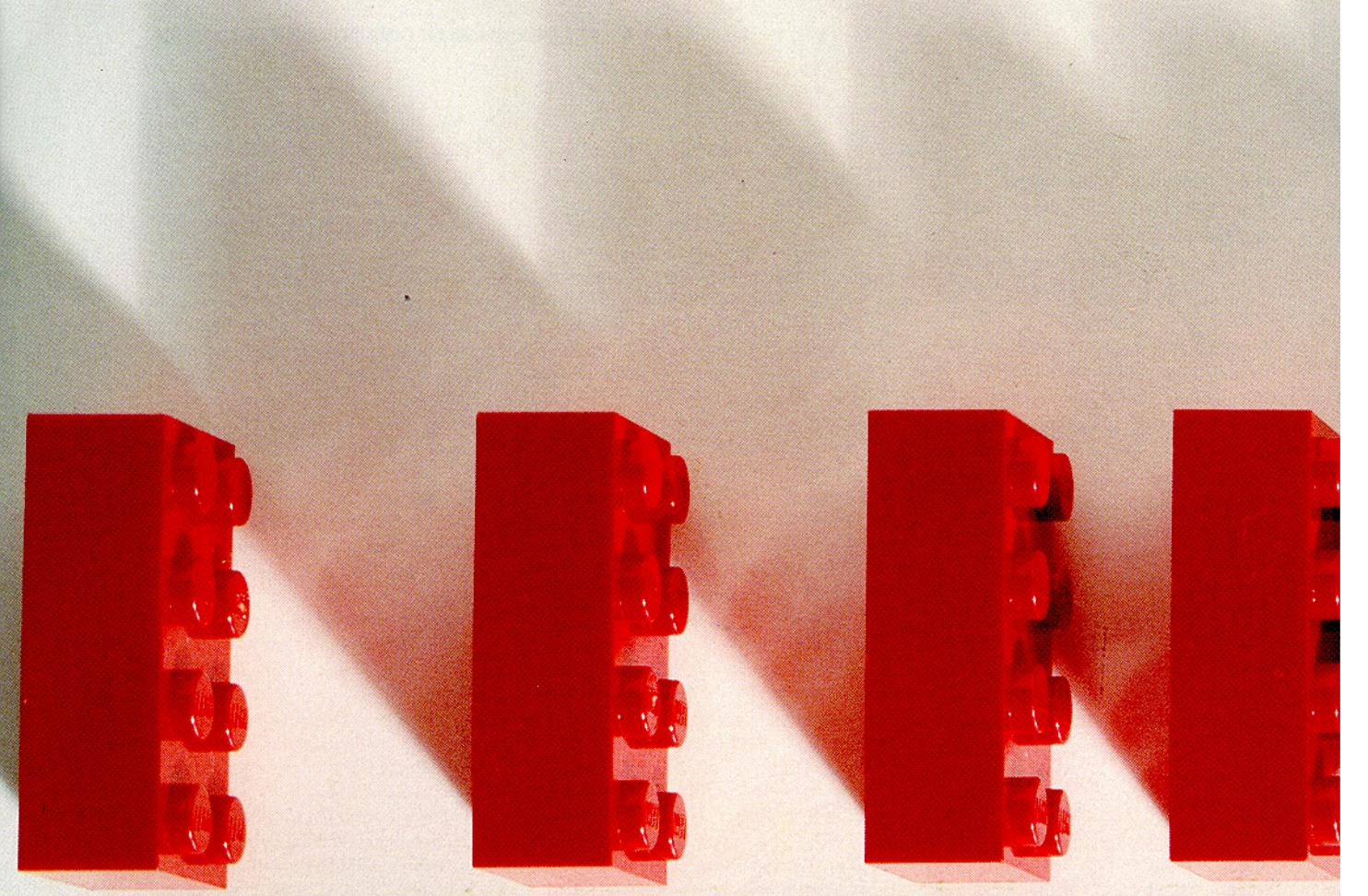
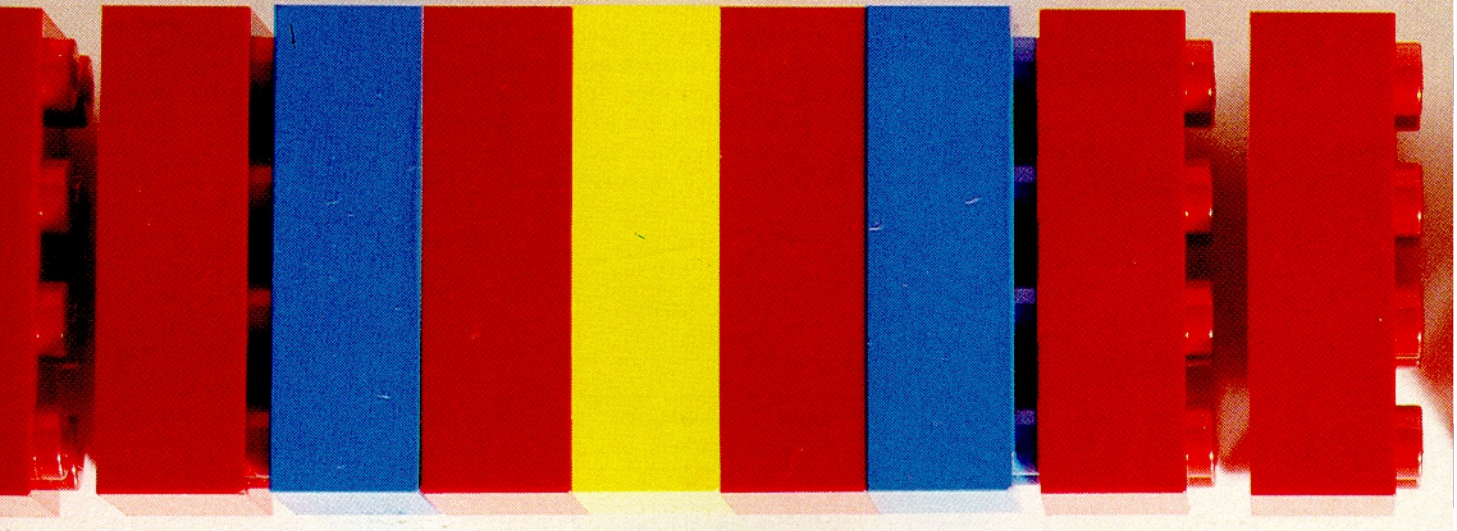


THE OBJECT-



OBJECT-ORIENTED SYSTEMS LIFE CYCLE

Brian Henderson-Sellers and Julian M. Edwards

In software engineering, the traditional descripton of the software life cycle is based on an underlying model, commonly referred to as the "waterfall" model (e.g., [4]). This model initially attempts to discretize the identifiable activities within the software development process as a linear series of actions, each of which must be completed before the next is commenced. Further refinements to this model appreciate that such completion is seldom absolute and that iteration back to a previous stage is likely. Various authors' descriptions of this model relate to the detailed level at which the software building process is viewed. At the most general

level, three phases to the life cycle are generally agreed upon: 1) analysis, 2) design and 3) construction/implementation (e.g., [36], p. 262; [42]) (Figure 1(a)). The *analysis* phase covers from the initiation of the project, through to users-needs analysis and feasibility study (cf. [15]); the *design* phase covers the various concepts of system design, broad design, logical design, detailed design, program design and physical design. Following from the design stage(s), the computer program is written, the program tested, in terms of verification, validation and sensitivity testing, and when found acceptable, put into use and then maintained well into the future.

In the more detailed description of the life cycle a number of subdivisions are identified (Figure 1(b)). The number of these subdivisions varies between authors. In general, the problem is first defined and an analysis of the requirements of current and future users undertaken, usually by direct and indirect questioning and iterative discussion. Included in this stage should be a feasibility study. Following this a user requirements definition and a software requirements specification, (SRS) [15], are written. The users requirements definition is *in the language of the users* so that this can be agreed upon by both the software engineer and the software user. The software requirements specification is written in the language of the programmer and details the precise requirements of the system. These two stages comprise an answer to the question of WHAT? (viz. problem definition). The user-needs analysis stage and examination of the solution space are still within the overall phase of analysis but are beginning to move toward not only problem decomposition, but also highlighting concepts which are likely to be of use in the subsequent system design; thus beginning to answer the question HOW? On the other hand, Davis [15] notes that this division into "what" and "how" can be subject to individual perception, giving six different what/how interpretations of an example telephone system. At this requirements stage, however, the domain of interest is still very much that of the *problem space.* Not until we move from (real-world) systems analysis to (software) *systems design* do we move from the problem space to the solution space (Figure 2). It is important to observe the occurrence and location of this interface. As noted by Booch [6], this provides a useful framework in object-oriented analysis and design.

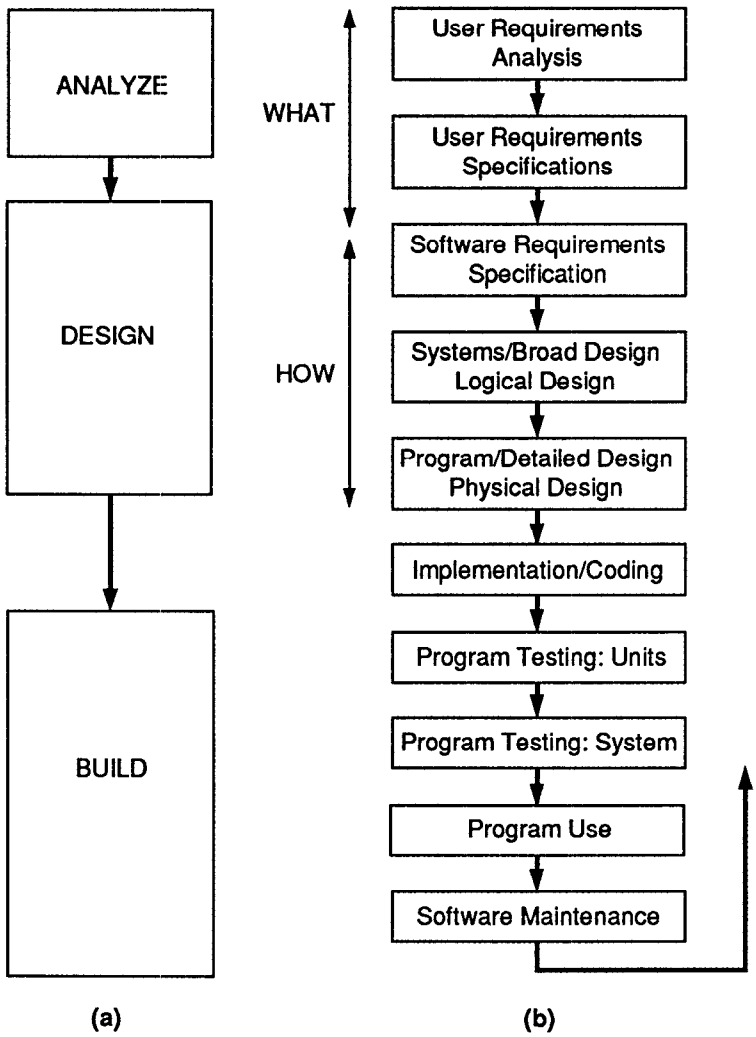


FIGURE 1. The traditional software life cycle: (a) at a gross scale three phases are identified; (b) at a more detailed scale a number of other steps become evident.

The design stage is perhaps the most loosely defined since it is a phase of progressive decomposition toward more and more detail (e.g., [41]) and is essentially a creative, not a mechanistic, process [42]. Consequently, systems design may also be referred to as "broad design" and program design as "detailed design" [20]. Brookes *et al.* [9] refer to these phases as "logical design" and "physical design." In the traditional life cycle these two design stages can become both blurred and iterative; but in the object-oriented life cycle the boundary becomes even more indistinct.

The software life cycle, as described above, is frequently implemented based on a view of the world

interpreted in terms of a *functional decomposition*; that is, the primary question addressed by the systems analysis and design is WHAT does the system do viz. what is its *function*? Functional design, and the functional decomposition techniques used to achieve this, is based on the interpretation of the problem space and its translation to solution space as an interdependent set of functions or procedures. The final system is seen as a set of procedures which, apparently secondarily, operate on data.

Functional decomposition is also a top-down analysis and design methodology. Although the two are not synonymous, most of the recently published systems analysis and design methods exhibit both characteristics (e.g., [14, 17]) and some also add a real-time component (e.g., [44]). Top-down design does impose some discipline on the systems analyst and program designer; yet it can be criticized as being too restrictive to support contemporary software engineering designs. Meyer [29] summarizes the flaws in top-down system design as follows:

1. top-down design takes no account of evolutionary changes;
2. in top-down design, the system is characterized by a single function—a questionable concept;
3. top-down design is based on a functional mindset, and consequently the data structure aspect is often completely neglected;
4. top-down design does not encourage reusability. (See also discussion in [41], p. 352 et seq.)

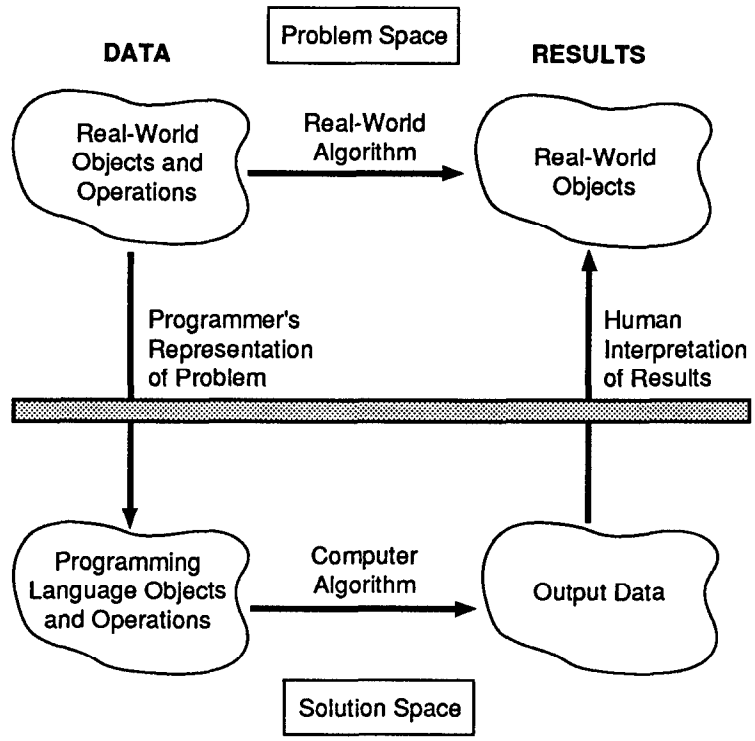


FIGURE 2. Problem space versus solution space for a typical software system (From *The Programming Language Landscape* by Henry Ledgard and Michael Marcotty [26], ©1981 Science Research Associates, Inc. Reproduced by permission of the publisher).

Alternative Decomposition Methodologies

Figure 3 considers three basic options for systems analysis and design (collectively called systems development). These three options are essentially on a grey scale from top-down functional decomposition which is process-driven, through Jackson Structured Development (JSD) to object-oriented decomposition. A more detailed discussion on

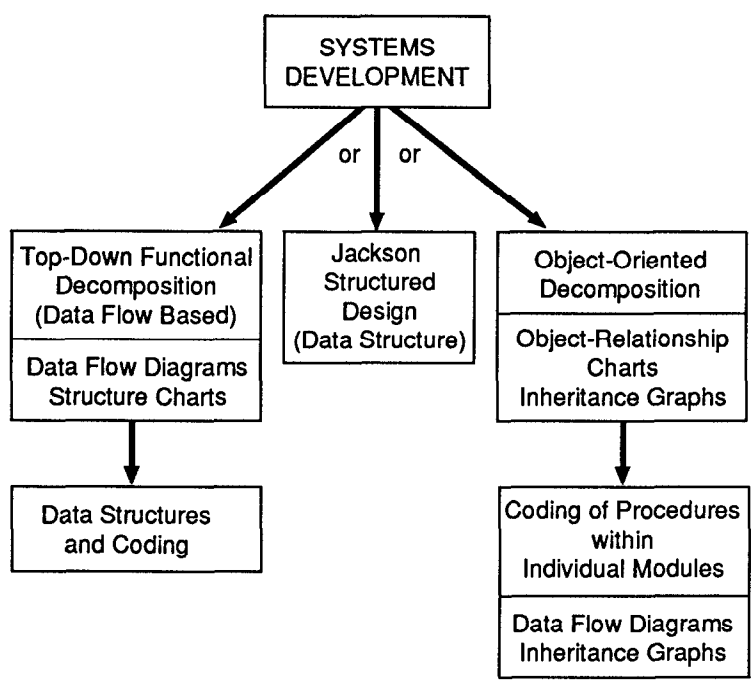


FIGURE 3. Three options for systems design methodologies: from top-down functional decomposition through JSD to object-oriented decomposition.

variants on these themes can be found in [15], for example.

Functional Decomposition

In functional decomposition, procedures and algorithms are uppermost in the designer's mind. Initially the system is viewed at a high level in terms of what it is intended to *do* and then at a detailed design stage *how* it will accomplish these process-oriented goals. Design tools used to support this methodology which is strongly based on *data flows* include data flow diagrams (DFDs), data dictionaries (DDs) and structure charts.

Functional decomposition is well supported by the older procedural languages and is therefore a natural mode of design expression in that context. The incorporation of subroutines into these languages led to the ability to undertake some degree of autonomy and information hiding; whilst at the same time shared data tended to be placed in globally accessible data storage areas (e.g., COMMON blocks in Fortran, externals in C). This leads to the "stack of dominoes" effect familiar to anyone working in program maintenance whereby changes to one part of a software system often cause a problem in an apparently dissociated program area. Furthermore, a system envisaged as providing a single service (single function) is unable to evolve to take into account new data structures or new functions with any degree of robustness.

Jackson Structured Development

Jackson Structured Programming and Jackson Structured Development (JSD) [21, 22] are techniques which initially model the real world, impose some chronology and use some object-based concepts (data structure methods). However, they are essentially functional decomposition methodologies ([41], p. 185) in which the data structures are used to assist in the functional decomposition; as compared to a functional decomposition in which the data structures are essentially determined by the procedure structures; although in some sense, JSD can be considered to be at least partially object-based and "middle-out" [10]. Certainly, it attempts to shift the emphasis away from procedures (or functions) by commencing analysis with a system-modeling exercise [22].

JSD attempts to model systems by taking some account of objects, but, as noted above, imposes a rigid chronology on the system components. For systems for which this is a reasonable assumption, the JSD methodology would appear to provide useful insights, especially since it provides the same representation for program structure as for systems and data structure analysis viz. elementary, sequence, iteration and selection ([9], p. 160). This provides a graphical interpretation and analysis tool which adopts a different approach to that embodied in the data flow diagram in which such iteration and selection structures are forbidden.

Object-Oriented Development

The object-oriented (OO) paradigm, at its simplest, takes the same components of a software system: data and procedures, but de-emphasizes the procedures, stressing instead the encapsulation of data and procedural features together, exemplified by the clear and concise specification of the module interface. In a systems decomposition based on an object-oriented approach, the system is viewed as a collection of objects, sometimes referred to as entities [3, 24] or object classes [29]. High-level analysis and design is accomplished not only in terms of these objects, but also the services they provide using a client-server model of object relationships in which objects interact with each other via "messages" which pass information, invoke the objects to implement a procedure etc. The use of the client-server model leads to the system being described as "responsibility-driven" [47].

Detailed design, including procedure implementation and specification of data structures, is deferred until much later in the development process and are private to the object, thus adhering strictly to the concepts of information hiding as promulgated by Parnas [34]. Consequently, algorithmic procedures and data structures are no longer "frozen" at a high level of systems design. Therefore a system based upon object representation can remain more flexible since changes at the implementation level are more easily accomplished without requiring changes to the systems design itself. It is important that data structures should not be specified too early in the design process. Data entities may, however, provide the basis of object identification around which an interface is then developed. Thus, object development focuses on data abstraction rather than freezing specific data structures into the object specification.

Since much of object-oriented program development is bottom-up, the differentiation between program design and coding is much less distinct than in a procedurally-based systems life cycle (e.g., [30]). However at this later stage, it would seem reasonable that *within individual code modules,* called *classes* in a object-oriented system, the tools developed for high-level functional decomposition and top-down system design, such as DFDs, can still be found to be useful. Other graphical tools which are indispensable at different stages within the OO systems life cycle are object-relationship graphs, client-server diagrams, inheritance charts or collaboration graphs [46].

In contrast to the common structured systems analysis based largely on top-down functional decomposition, object-oriented (OO) design and analysis has many attributes of both top-down and, perhaps predominantly, bottom-up design. Since one of the aims of an OO implementation is the development of generic classes for storage in libraries, an approach which considers both top-down analysis and bottom-up design simultaneously is likely to lead to the most robust software systems. Indeed several authors [13, 22, 27, 42] suggest that in reality, practitioners purporting to be following a strictly

top-down approach actually utilize a mixed mode of operation between top-down, bottom-up and middle-out.

Matching Systems Analysis and Design Methodologies

Perhaps the most important question regarding the introduction of object-oriented methodologies into *commercial* environments is whether it is necessary to use the techniques throughout the whole life cycle or whether it is possible to "mix and match" with functional decomposition techniques. Although it would appear that the former would be more self-consistent, the arguments for proposing the latter approach consider the reality of the large current investment in top-down functional decomposition, both in terms of expertise and front-end CASE tools, which many industries currently possess (e.g., [43]). Secondly, even if the OO design methods are viewed as "more natural," the investment in code in COBOL, for example, leads to the question: can an object-oriented design, viewed simply as a better design, be implemented in a non-OO language?

Figure 4 summarizes the possible routes. On the left-hand side of the figure is the traditional top-down functional design approach. The analysis of the problem is undertaken in problem space, which is full of objects, and the transition to the solution space model (Figure 2) requires a mapping between inhomogeneous concepts i.e., from problem space real-time objects to solution space functions (indicated by the wavy arrow). If this design is then described using the same language it will be implemented in, then there is a one-to-one mapping to the implementation. However, if the design language (DL) is not the same as the implementation language, there is a potential mismatch at the DL to implementation stage.

On the right-hand side of Figure 4 is illustrated a thoroughbred object-oriented methodology: an analysis of objects in problem space being directly mapped on to objects in solution space and, if the same language environment is used for implementation, a second direct mapping to the code. Indeed, the use of the same language environment for both systems design and systems implementation is seen by many as one of the great strengths of an object-oriented approach ([3, 7]). Furthermore, many of the methodologies currently being developed for the object-oriented programming environment stress the implementation phase since an OO design, in some sense designed top-down at a high level, can be implemented bottom-up by identifying existing library classes, extending these by using inheritance and constructing new classes as appropriate (e.g., [19, 29, 40]). In such a process, both top-down design and bottom-up construction occur concurrently.

Two major likely linkages are illustrated in Figure 4: from a functional description at the analysis stage which is then transferred to an object view at the design stage [2, 6] and presumably then an OO implementation (referred to hereafter as the FOO, for **F**unctional analysis, **O**bject-oriented design, **O**bject-oriented implementation methodology) or the implementation in a standard procedural language of an object-oriented design (OOF). Either of these paths is indeed feasible and is currently under investigation. Here we analyze the OOO (**O**bject-oriented analysis, **O**bject-oriented design, **O**bject-oriented implementation) approach, commenting only briefly upon the utility of the two hybrid approaches, FOO and OOF.

It should be noted that in much of the work in the Ada environment, the object-oriented development approach is seen by many as only a partial life-cycle methodology in that it is considered to be a design and implementation methodology only ([6], p. 47). As such it is required to be

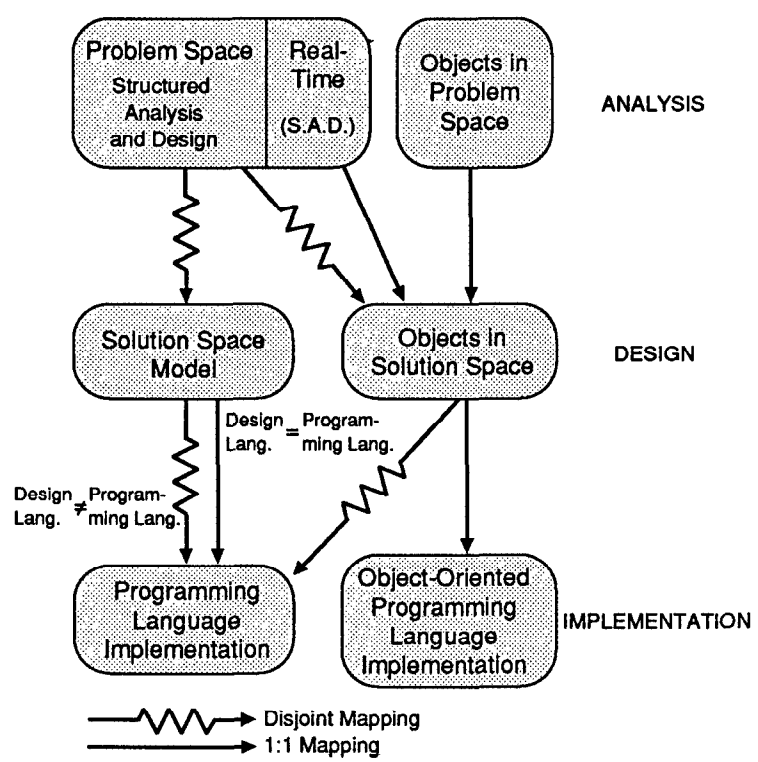


FIGURE 4. Functional and object-oriented decomposition and their possible interlinking. One-to-one mappings are indicated by arrows and disjoint mappings by wavy arrows.

preceded with a methodology for requirements analysis and systems analysis and consequently viewed as being of a FOO type. Booch [6] recommends JSD as providing the requirements analysis tool in this approach. On the other hand, Seidewitz [37] considers that the DFD to OO design step of e.g., [2, 38, 43] is more appropriately replaced by *beginning* the cycle with an object identification and object-oriented requirements analysis in the problem domain [3]. Ladden's [25] perspective is that of a thoroughbred object-oriented life cycle but the entity-data flow diagrams (EDFDs), as defined in [3], to be written in the analysis stage are seen as being "carved" from a functionally-derived DFD in the traditional sense. Although this builds on existing, understood methodologies, it would appear to add unnecessary complication in the form of an additional intermediate step in deriving an entity data flow diagram, a step which is often difficult and/or time consuming. Ladden [25] notes this problem and also suggests a more direct approach. Overall, he sees functional decomposition and object-oriented design as complementary.

Most of the object-oriented development methodologies presented in the literature to date are not generic, having been structured specifically with Ada in mind. Since Ada is not an object-oriented language, but only an object-based language [45], some of the major features of the object-oriented paradigm, such as inheritance, are not addressed directly in the analysis and design notations, such as those of Booch [5, 6], although some of the analysis and design ideas can be more generally applied (see the following section). (Overviews of Ada-specific design methodologies are contained in e.g., [8, 25, 37].) In recent years non-Ada oriented methodologies have begun to appear, for example [11, 12, 29]. These are mainly partial life-cycle models concerning program design or the requirements phase only. Meyer [29], for example, concentrates on class development using formal specification of abstract data types (ADTs) to define behavior and provides examples on how to transform a functional design to an object-oriented design (OOD). Coad and Yourdon [12] present an object-oriented analysis (OOA) method, using data modeling techniques, in which they define the system architecture in terms of assembly and classification structures. Services are specified for objects by techniques such as entity-life-history and state-event-response charts. Wirfs-Brock and Wilkerson [47] use a responsibility-driven approach to OOD. This method concentrates on the responsibilities of an object and the information it shares with other objects. The aim here is to increase encapsulation of information within the object.

These approaches were designed to try to utilize all the features of an object-oriented design, unlike many of the Ada methodologies.

A Coherent Object-Oriented Development Methodology

As illustrated in Figure 4, there is a distinct difference between object-oriented programming and object-oriented systems analysis and design. Many techniques discussed by e.g., [6, 29] are primarily aimed at OO program implementation; although such authors also make some propositions regarding the analysis and design stages. Booch ([6], p. 48) identifies five major stages:

- Identify objects and attributes
- Identify operations affecting objects
- Establish visibility
- Establish interface
- Implement each object

It can be seen that the first items are, indeed, at a system level, but the last few are more strictly detailed design or implementation details. As a set of criteria, therefore, they are better suited to "object design" rather than "system design," confirming that bottom-up or middle-out design is also appropriate in an object-oriented development process. In other words, as can be seen simply from this list, the distinction between system design and code design can become rapidly blurred. This results from the twin, concurrent needs to analyze the overall system at a high level of abstraction in terms of an object-oriented systems design, while acknowledging that implementation in an object-oriented programming language is accomplished best by using a bottom-up development of new objects which can utilize the extensive set of library classes already in existence. These library classes are the coded versions of the design objects, or object classes. In general, the term *class* refers to the compile-time description of the abstract data type used to describe the solution space objects and design objects.

Both top-down analysis and bottom-up class design, seen as the hardest part of the entire object-oriented software life cycle [28], must therefore be either concurrent or, at least, iterative. In order for successful code reuse, existing, *well-validated* classes must be seen to be part of the design stage, not just the implementation. It should also be noted that these five steps do not provide guidelines on establishing object-object relationships in terms either of client-server or in terms of the powerful inheritance mechanism. This may be, in part, due to the Ada-orientation of Booch's methodology which precludes a straightforward use of inheritance concepts.

In a study of an object-oriented *requirements specification* method, Bailin [3] also identifies the large degree of overlap between not only design and coding, but also between requirements specification and systems design. He identifies seven steps for his specification method, which could obviously transcend the requirements stage well into detailed design:

1. identification of key problem space objects (Bailin uses the term "entities" throughout);
2. distinguish between active and passive objects;
3. establish data flows between active objects;

4. decomposition of objects into "sub-objects";
5. check for new objects;
6. group functions under new objects;
7. assign new objects to appropriate domains.

Bailin [3] sees the steps 1–3 as once-only steps, with steps 4–7 being performed iteratively. He applies these steps to the analysis phase of the life cycle. However, there are immediate links to class design and implementation. For example, decomposition of objects can be identified at the implementation stage with the procedures of classification, aggregation and generalization. In this method, a data-flow diagramming technique is retained to describe the system in which the nodes become objects (i.e., classes at implementation). Although Bailin [3] suggests that EDFDs can also possess nodes to represent functions, both he and other authors (e.g., [29]) suggest that functions should be part of the implemented object (i.e., the class) and, therefore by implication, part of the analysis phase object.

Based on these two possible criteria for object-oriented systems development, a consensus can be drawn on both the stages pertinent to an object-oriented life cycle and the common features associated with each phase. Although several stages will be identified (Table I), each overlaps and there are many iterative possibilities that must be borne in mind.

1) *Undertake system requirements specification.* This stage is a high-level analysis of the system in terms of objects and their services, as opposed to the system functions. However, if a great deal of OOA is undertaken, then the result may be an object-oriented requirements specification (OORS) including timing details, hardware usage, cost estimates and other documentation. Coad and Yourdon [12] and Shlaer and Mellor [39] both present OOA methods that may be used at this stage.
2) *Identify the objects.* At both the analysis and the high-level design

TABLE I. Summary of proposed seven-point methodological framework for object-oriented systems development

Step	Summary description (see text for details and explanation of terminology)
1.	Undertake object-oriented system requirements specification
2.	Identify the objects (entities) and the services each can provide (interface)
3.	Establish interactions between objects in terms of services required and services rendered.
4.	Analysis stage merges into design stage: use of lower-level EDFDs/IFDs
5.	Bottom-up concerns → use of library classes
6.	Introduce hierarchical inheritance relationships as required
7.	Aggregation and/or generalization of classes

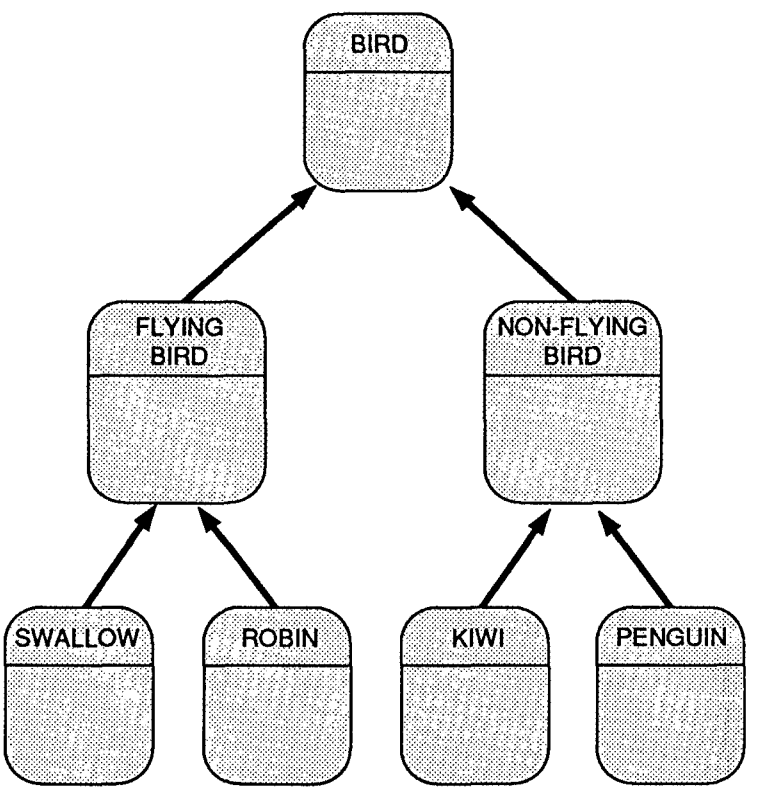


FIGURE 5. Inheritance diagram. The classes of flying and non-flying bird inherit the majority of their attributes (e.g., feathers, beak, lay eggs) from the parent class bird but the flying attribute is defined in these classes. Individual bird types then inherit the appropriate set of characteristics, including that for the capability of flight, and add attributes for that species e.g. kiwi, penguin. Individual birds are then viewed as instantiations of these lowermost classes.

stage, it is necessary to identify the objects or entities, their attributes and the services they provide. This is where the functional features will be defined, although no indication of implementation is required; this being one of the basic tenets of the object-oriented paradigm. Objects can often be identified in terms of the real-world objects, although as Meyer [28] notes, often, abstract nouns also provide excellent objects. However, at this high level of abstraction it seems unnecessary to decompose the object and look for more primitive object representations, since these are more reasonably part of the detailed design stage. Here, an object (entity) dictionary should be used as the object-oriented analogue of the data dictionary.

Identification of appropriate objects must therefore not be undertaken solely on the objects identified in the requirements specification and the analysis and design stage, since it is anticipated that well-designed object classes will be used again and again. Such identification of genericity (see also Steps 6 and 7), although encouraged in standard procedural language courses, is seldom taken seriously. Identification of objects and ultimately of classes will *de facto* define the operations affecting the objects, and the services they offer, hence defining the visible interface.

3) *Establish interactions between objects* in terms of services required and services rendered (analogous to the data flow of Bailin's EDFD). For this stage, a diagrammatic notation akin to the EDFD and/or the entity-relationship (ER) diagram would be most useful. Perhaps a better name for the object-oriented equivalent of the DFD, rather than EDFD, might be IFD (information flow diagram) where "information" relates to messages and/or message-arguments, since in general, data are contained *within* an object and *do not* flow in the same sense as that described by a DFD in functional decomposition techniques. Another type of software tool, useful in specific programming environments is illustrated by the Eiffel*-specific editor, GOOD [30]. GOOD is an interactive object browsing facility that permits the user to trace the object hierarchies and object relations. It also permits examination of attributes and routines of individual objects. (Browsing tools are discussed in more detail in [18]).
4) As the analysis stage merges into the design stage, *lower level EDFDs/IFDs can be drawn* to illustrate more internal details of the objects. From this stage onward, bottom-up concerns should be taken into consideration. The identification of reusable design components, or classes, from previous designs is an important part of the OO strategy. In some languages (e.g., Ada), classes can be embedded inside other classes (e.g., [23]). In a pure object-oriented life cycle (right-hand side of Figure 4), especially one which is using the same language for both systems analysis, design and implementation, the decision of whether to represent embedded classes or not will therefore reflect the language being used. In Eiffel, no embedding is possible. Instead, classes refer to other classes so that including an attribute of a given (abstract data) type inside a class is simply a reference to an object (which at the coding level now refers to a single instantiation) of the class which defines the implementation of that abstract data type. Diagrammatically design-level objects would remain as individuals. However, at the top level an object of higher-level abstraction would be represented by a single piece of notation (which we note we have not yet defined!—see the following section, "Graphical Notation for Object-Oriented Software Development").
5) Concurrently, bottom-up concerns should be being realized insofar as entity-relationship diagrams, such as EDFDs or IFDs, can be used to describe the more detailed internal structure of the objects of the analysis diagrams. Objects are themselves constructed from libraries of more primitive objects using concepts of client-server and contracting [31, 47]; the libraries themselves contain object classes created as one of the successful outcomes of a previous application of this (or other) proposed development methodology. Initially, implementation (coding plus testing) of low-level classes may begin at this stage, following the cluster model described below.
6) As more objects are identified within the detailed design, reevaluation of the total set of classes will require an iterative analysis of whether new superclasses (parents) or new subclasses (children) will be useful, creating a need for inheritance diagrams (Figure 5). For example, a class of "bird" with an attribute that "birds can fly" is successful until we consider the Southern Hemisphere and "penguins," "ostriches," "kiwis" etc. In this case, one solution is to introduce an additional level in the inheritance hierarchy by introducing two children classes of class bird as "flying bird" and "non-flying bird" and redefining the parent class to remove the attributes relating to flight [7, 30]. This process tries to develop a logical hierarchy of objects so there are no "missing" objects. This step is needed in order to provide a well-defined hierarchy so that future projects can reuse the structure without having to redesign the inheritance chart for themselves. This process will probably take place during the generalization phase of the cluster model of class development.
7) Aggregation and/or generalization of classes, as undertaken in the previous step, may require iteration back to reconsider the

*Eiffel is a trademark of Interactive Software Engineering, Inc.

EDFDs/IFDs describing the system. It may be that prototyping will already have commenced by this stage, providing constructive feedback to the potential users so the requirements documents may require modification and clarification (e.g., [42]) leading to further development of the class specified. Although this is contrary to expectations of the traditional life-cycle model, such feedback, essentially from one end of the traditional life cycle to the other, is made possible by the object-oriented techniques and is seen as providing a more reliable, robust and useful software system.

The system classes that are identified and developed during this process may then undergo another stage of development, known as generalization, following the cluster model. At this stage the components continue to be worked on until they are general, generic and robust enough to be placed in a library of components. In the short term, this adds a development overhead to the current project which is more than compensated for by the long-term saving when future projects can directly utilize library objects rather than have to design, implement and test them from scratch.

As can be seen from this suggested OOO methodological framework, class design and system design go hand-in-hand. Meyer [29] stresses the bottom-up nature of class design in order to provide generically useful classes. However, these cannot be decided *in vacuo.* System design must influence class design and implementation; whilst class design and class availability must influence system analysis and structural design.

The cluster model (Figure 6) has been proposed by Meyer [30] as a life cycle for a tightly related group of *classes,* or *cluster,* in which three phases are identified. First a specification is written by the systems designer (SPEC), then this is designed and implemented (DESIMPL) (one process in a language like Eiffel) and finally it is validated and generalized (VALGEN). This life cycle occurs for different clusters of classes at different times. For example, a graphics cluster and a windows cluster of classes could be specified, designed and implemented and then validated and generalized at different times. It should be noted that this model applies to software classes and not to software systems. The specification of a class is refined from a specification of the system and describes in as much detail as possible the services and semantics of the class. This would best be expressed theoretically as a formal specification of an Abstract Data Type [29].

High-level object identification at relatively early information systems design stages allows programmers to begin work while analysis of the system continues. In this way, the requirement may be changed as the customer develops an understanding of his or her needs without adversely affecting the software design stages. The initial object identification will probably be of persistent objects which can be identified early and thus are unlikely to change a great deal as new requirements are added. For example, in an aircraft radar system objects such as radar, aircraft and windows will be required. These may be presented to programmers as groups of object classes to program at a very early stage with a high degree of confidence that even if the systems specifications were to change (or the project be cancelled), these classes would still be useful.

Graphical Notation for Object-Oriented Software Development

Object-Oriented Life Cycle Notation

Any graphical representation of the object-oriented version of the overall software development life cycle must take into account the high degree of overlap and iteration implicit ([27, 30, 35, 42]). Rather than using a revised waterfall model, the "fountain model" of Figure 7 seems to be appropriate. First, it provides a diagrammatic version of the stages present in an object-oriented software life cycle and a clearer representation of the iteration and overlap made possible by object-oriented technology. Second, since the foundation of a successful software project is its requirements analysis and specification, this stage has been placed at the base of the diagram. The life cycle thus grows upward to a pinnacle of software use, falling only in terms of necessary maintenance. This effectively reverts the stage of the cycle to

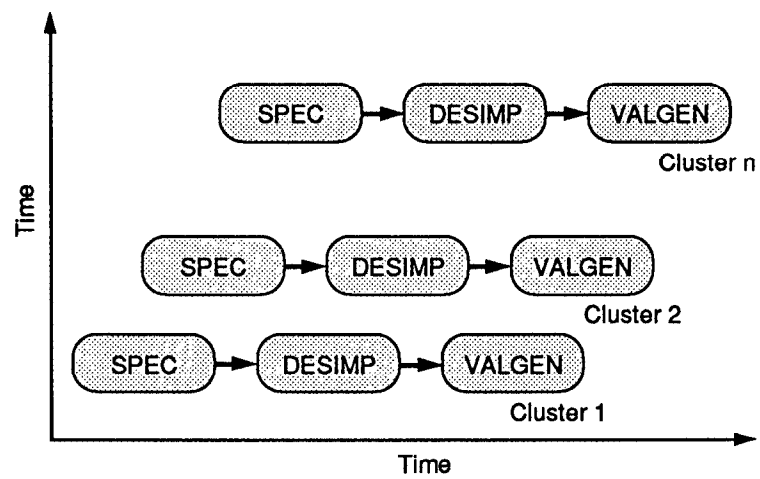
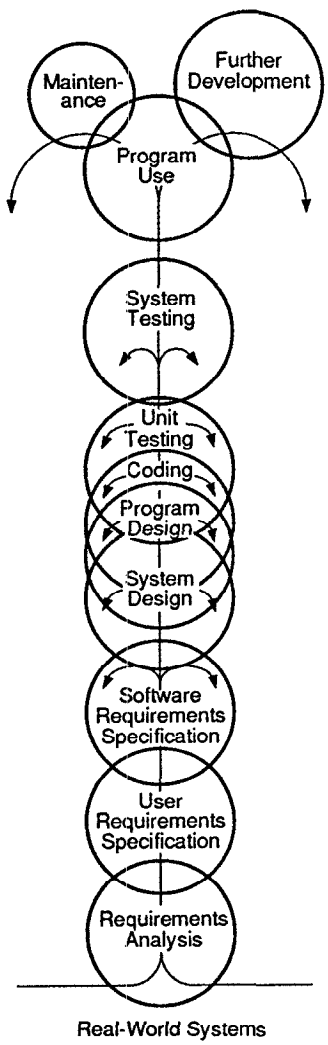


FIGURE 6. Software lifecycle for module/class design: the cluster model (after [30]). SPEC = specification; DESIMPL = design + implementation; VALGEN = validation and generalization.

a lower level. Application of this model to specific environments may lead to software life cycles such as prototyping: a development environment for which object-oriented programming languages (OOPLs) are well suited. (This model is also applicable to the traditional functional-decomposition-based description of the life cycle, as described in Figure 1. In this case, however, there is a significant decrease in the overlap of the circles, and in some cases disjoint processes may be represented by tangential circles.)

The system can be viewed not only in terms of the *systems* life cycle, as presented in Figure 7, but also in terms of a synergistic amalgamation of a (usually large) number of autonomous classes. Modifications can more easily be made interactively between class development and systems specification so there is no longer a need to freeze the overall systems requirement specification at an early stage of the system life cycle—an attribute of the object-oriented methodology viewed very positively by many authors. Consequently, since an object-oriented program will be developed essentially as an interacting system of classes (usually developed independently), the stages of the life cycle model can be applied more accurately to the development cycle of each individual class (Figure 8) rather than the system as a whole [19]. More generally, the stages of Figure 8 would apply concurrently to a small number of classes grouped as a cluster (Figure 6). In this sense the model represented by Figure 8 could be utilized directly for clusters as well as individual classes. Through the introduction of explicit stages of aggregation and generalization, case-specific classes are revised so that they can be sufficiently generic to be of use in a substantially wider range of applications than the single one for which they were originally developed. This requires a greater effort than one-off design/implementation in the short term, but in the long term, when a sufficiently broad library has been constructed, will lead to significant reduction in overall systems development time and effort. This is a consequence of the emphasis on code reusability within the object-oriented design environment which can be accomplished best by both bottom-up development of classes often within the framework of an overall top-down object-oriented systems analysis and high-level design.

The significance of the cluster models (described previously—Figures 6 and 8) in the software life cycle is as a branch of the systems specification and a result of its modular refinement. Figure 9 synthesizes the systems concepts of Figure 7 with the cluster life cycle ideas of Figures 6 and 8 and is a diagrammatic representation of how requirements design and implementation stages grow and iterate over time while individual classes or clusters of classes undergo their own cluster and fountain life cycle. The advantages of such a representation are that characteristics of a system, which evolves dynamically as users' and analysts' knowledge grow, can be incorporated in the overall life-cycle model. It has been



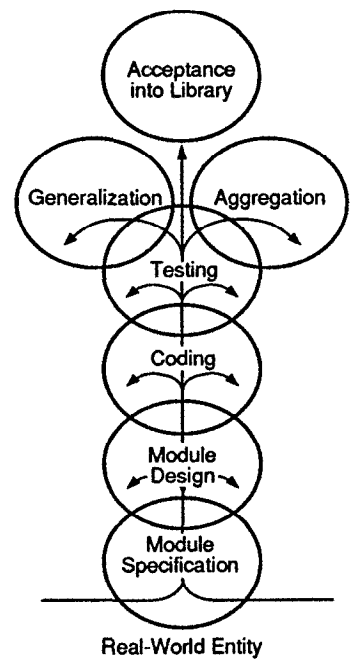


FIGURE 8. Life-cycle model for module development, where the module may relate to an individual class or to a cluster of classes. The stages in the model can be related to the proposed seven-point methodological framework as follows: module specification (Step 1); design (Steps 2, 3, 4); coding and testing (Step 5); generalization and aggregation (Step 6); possible iteration (Step 7). Classes (or clusters of classes) are only accepted into the library following successful completion of all seven steps.

FIGURE 7. Fountain model for the object-oriented software life cycle. Merging and overlap of activities are represented by the degree of overlap of the circle symbols. The decreased need for maintenance in an object-oriented system is symbolized by the smaller activity circle.

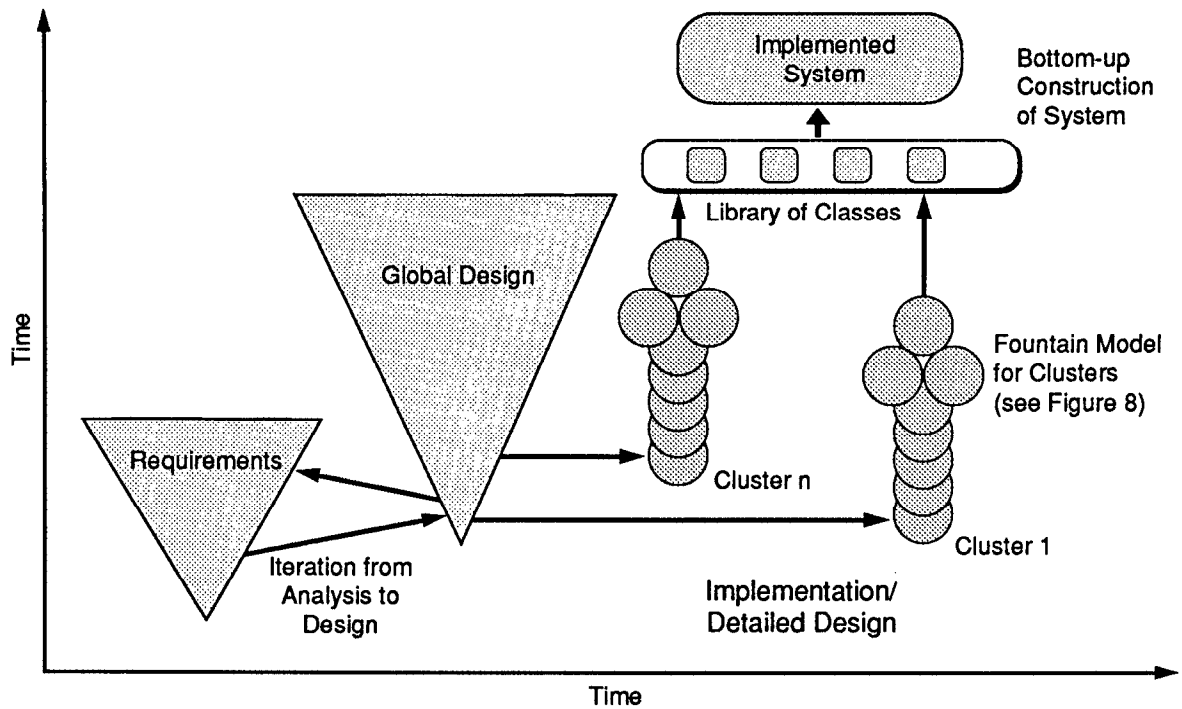


FIGURE 9. Interactions between requirements evolution and design and implementation development using object-oriented methodologies.

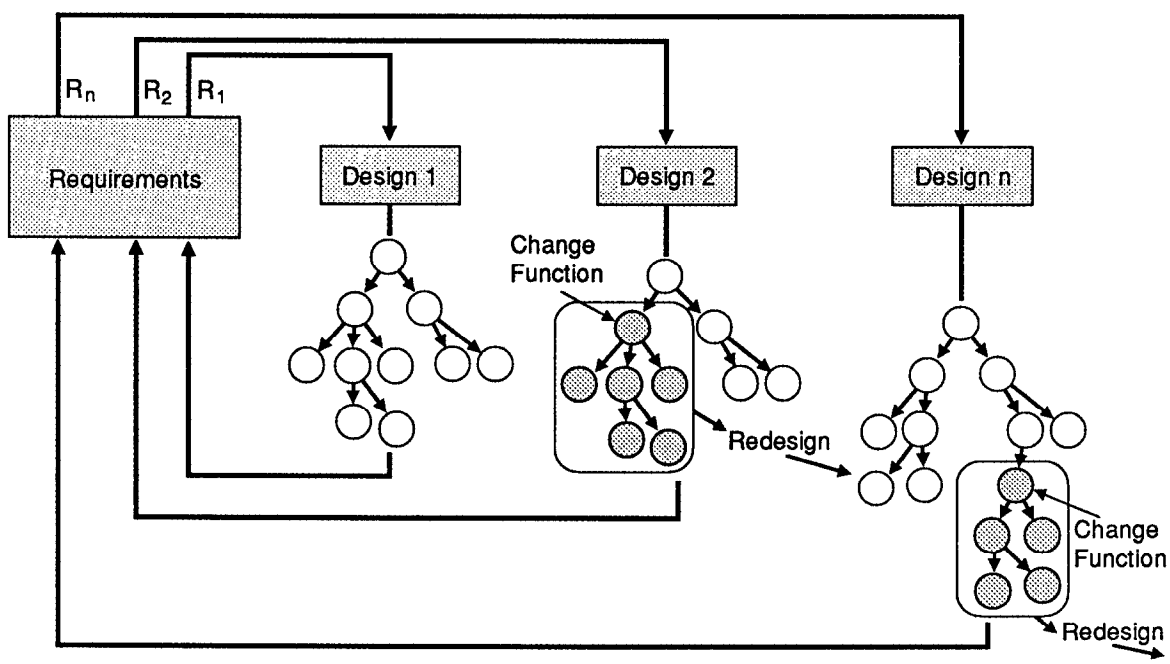


FIGURE 10. The effect on a top-down functional design of changes imposed at the requirements or broad design level.

recognized for some time that users' needs continually change and specifications are rarely fixed (e.g., [16]), but only now are methodologies being developed and applied which are able to incorporate such characteristics within the overall systems development. The decentralized architecture of object-oriented methods is the key to this approach.

The fact that clusters of classes can undergo detailed design and be implemented in isolation from the system means that system requirements and design spawn clusters which are then passed on to a team of programmers for detailed design and implementation, only later to be incorporated in the final system design by bottom-up methods. This makes iteration between systems requirements analysis and system design a much easier process, as designs are not based upon the first (and often least informed) decisions made, as is the case in top-down functional design. For example, Figure 10 shows how a change in the requirements in a system designed and implemented "top-down" necessitates a change in the "top" of the functional decomposition design, resulting in a great deal of necessary reworking since the more detailed design, having been decomposed from the uppermost function, is highly dependent on this main systems function. On the other hand, as seen in Figure 9, a change in requirements may cause the abandonment of a single cluster but this will not cause any major redesign

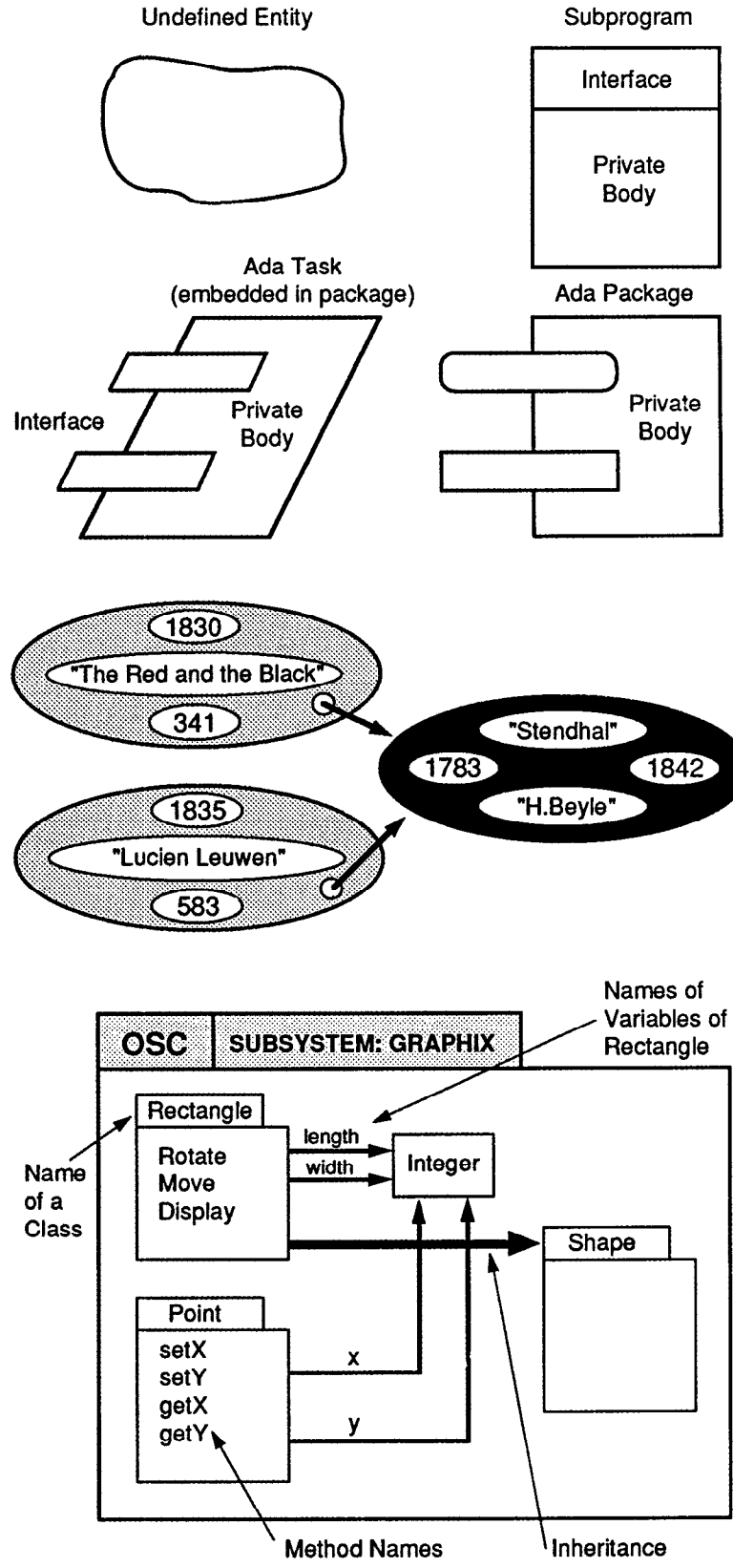


FIGURE 11. Booch's graphical notation for object-oriented implementation (after [6]). Generic symbols are these but dotted. Communication is by arrowed line.

FIGURE 12. An object is represented here as an oval shape containing several fields, some which contain attribute data relevant only to that particular object, others which require client-server access to a further object (on right). Here both objects on the left have an attribute of type "Author" which utilize common biographical data from the object on the right (after [29]).

FIGURE 13. Object Structure Chart (OSC) (after [2]). [By permission of Association for Computing Machinery].

since the overall system synthesis occurs much later in the systems life cycle.

Object-Oriented Design Notation

Diagrammatic notation at the level of class or cluster design and refinement requires representation for individual object classes, their subclasses and their interrelationships, especially in terms of inheritance and client-server mechanisms. In this context, Booch's graphical notation (Figure 11), despite being conceived as Ada-specific, is worthy of further study. These symbols relate most successfully to the detailed design stage or to class design. Indeed, Seidewitz [37] is critical of such a methodology applied at the systems-design level since he suggests that for a system of any real size such a graphical method becomes unwieldy (see also [23]). In more general terms, the graphical representation of an Ada package can be utilized as a general representation of an object, since it successfully typifies the discrimination between the private, hidden part of the object and the public portion, as represented by the protrusions. However, the Ada "task" has no general implementation; although the object-oriented paradigm is now beginning to embrace the need for concurrency (see e.g., [1]). An alternative graphical notation for objects and their interrelationships is given by Meyer [29] (Figure 12). Here, the object is shown to contain several attributes, some of which are defined within the object itself and some of which are references to other objects. In this example, these references, which access a shared object, represent services supplied to and obtained from other objects. However, no differentiation is made here between private and public parts, as in the Ada package notation. Alabiso [2] outlines an object structure chart (OSC) (Figure 13) notation which represents inheritance by a dashed line and object use by a solid line. Although differentiation is made between attributes and services provided by the class, it seems likely that with any significant number of interrelating classes, such a notation would become unwieldy. However at the detailed design stage such a notation could be useful and is similar to that described below.

A general, non-language-specific notation is therefore required. Object notation pertains to the design-level objects (or entities) and also to the classes of run-time objects prior to implementation. Since interrelationships between design objects and also between classes can be represented similarly, but at different stages of the life cycle, we will refer to such generic applicability by "O/C" (objects *or* classes). These diagrammatic representations must possess the following properties: O/C name, and public interface in terms both of services offered and informational attributes available. A slight modification of the Booch diagram seems to satisfy these requirements (Figure 14(a)). In addition, object-object (or class-class) relationships must be represented. For client-server, in which an attribute of an O/C is of an abstract data type and hence requires reference to another O/C for its realization, the arrow from a circle internal to the O/C directed to the name of the server O/C is useful (Figure 14(b)). Inheritance can be represented simply by a thick, directed arrow from the outer shell of the child O/C to the outer shell of the parent O/C. This is analogous to the **is a** arrowed rela-

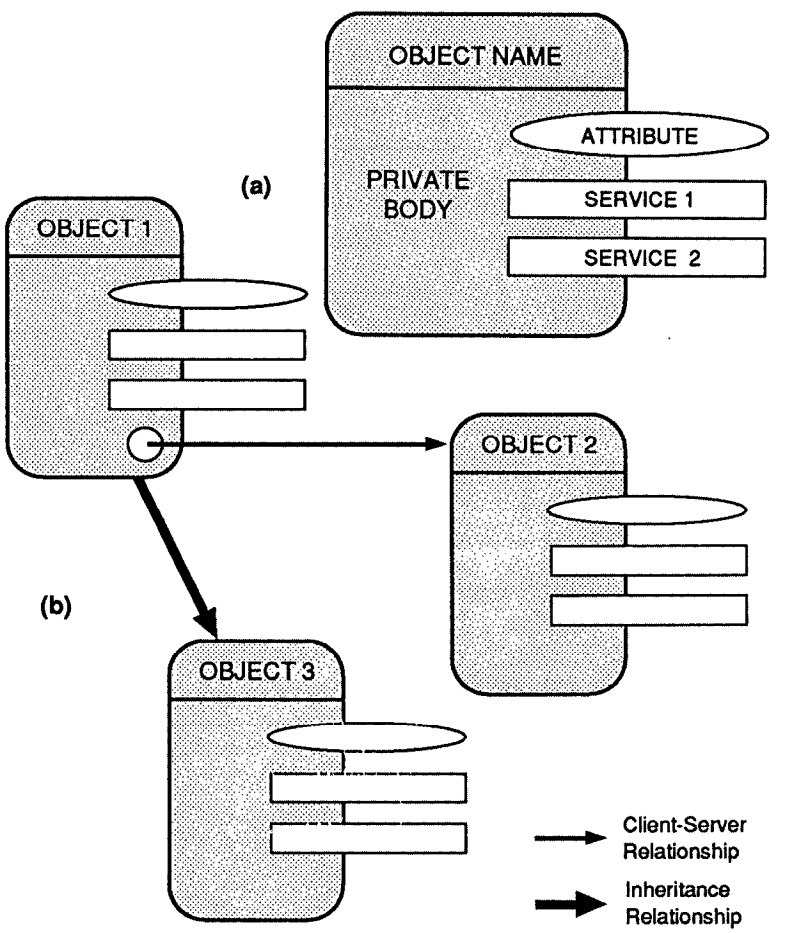


FIGURE 14. (a) Suggested notation for a single object/class and (b) notation for the interrelationships between objects/classes, either as client-server (Object/Class 1 uses the services of Object/Class 2) or by inheritance (Object/Class 3 is the parent of Object/Class 1).

tionships of e.g., [13].

Object identification, the first step of Booch's [6] methodology requires clarification. Although object-oriented terminology is not yet standardized and agreed upon, there are common threads, some of which are, nevertheless, language-dependent. Sommerville [41] identifies two types of objects: direct specification of the object (with all its attributes and operations), and defining an (object) class as a template. The latter is always to be used whenever more than a single instantiation of that class is required. Although supported in Ada, other languages do not require such differentiation. In Eiffel, all objects are instantiations of a class type (which is itself defined as an implementation of an abstract data type). In other words, the code contains only classes which are the templates for the creation of objects as instantiations of these class types at runtime.

A brief example will illustrate this approach. (This example is by no means intended to be complete, merely illustrative.) Consider the problem of constructing and manipulating a bibliography. A simple functional DFD to describe inputting and sorting the bibliography is shown in Figure 15(a). In contrast, consider the OOO methodology applied to this problem. In step 1 the objects are identified. These are bibliography itself, reference, author, title, date (for the simplest case). Also the notion of command, session and stored file will be useful as objects. One possible EDFD/IFD diagram, analogous to the functional DFD in Figure 15(a), is shown in Figure 15(b). In this diagram are indicated some of the top-level relationships (step 3) between objects. Further lower-level design details are added at step 4, probably for individual clusters of O/Cs (Figure 16). (In this example, all the classes used would be essentially part of the same cluster.) In step 5, details of the bottom-up concerns are realized in terms of the implementation of these new O/Cs in terms of existing library classes (Figure 17, together with Table II for the associated object dictionary). In this limited example, steps 6 and 7 are not evident.

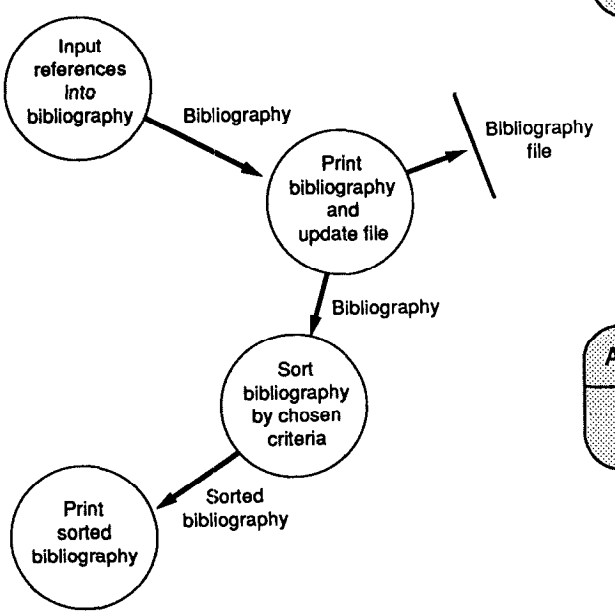


FIGURE 15a Functional data flow diagram for inputting and sorting a bibliography.

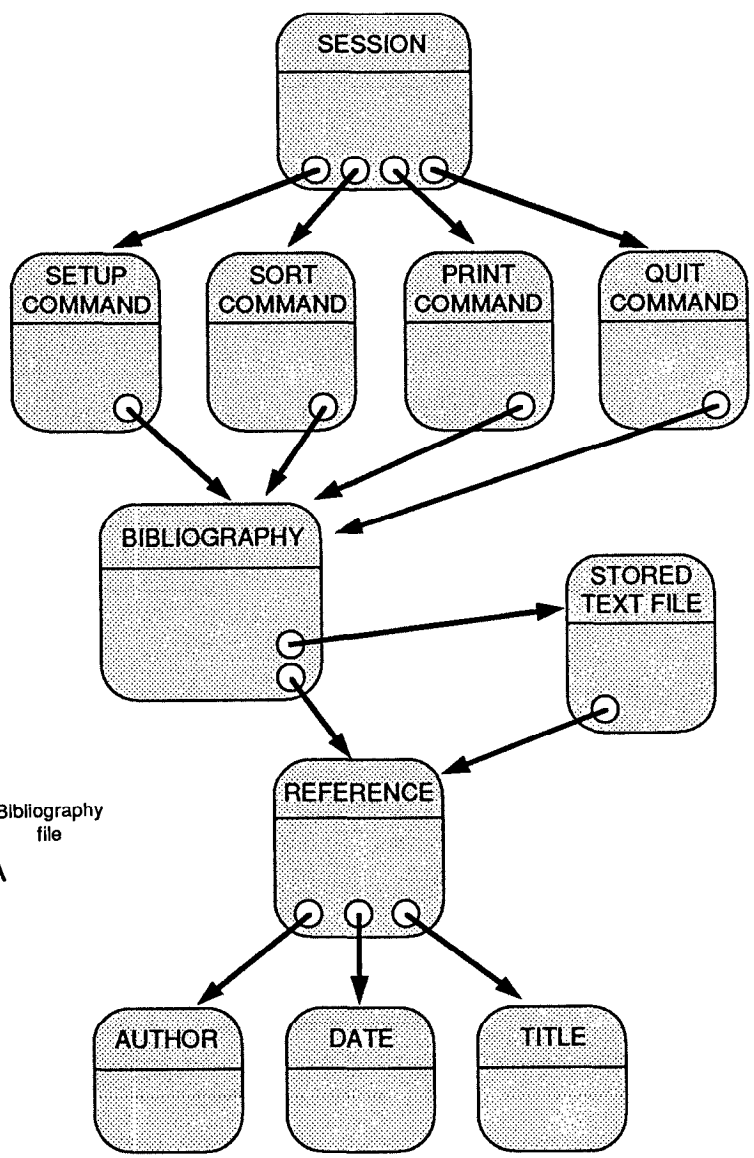


FIGURE 15b EDFD/IFD for high-level object-oriented design of a bibliography.

TABLE II. *Object dictionary*

Object/Class	Inherits from	Services Offered	Services Required
Session	—	Executes command	Setup command Print command Quit command Sort Command
Bibliography	FIXED__LIST	Sorts by author Sorts by date Sorts by title Prints bibliography	Reference
Reference	STD__FILES	Enters individual reference Prints individual reference	Author Title Date
Stored text file	FILE	Reads in text from file Provides next reference	String

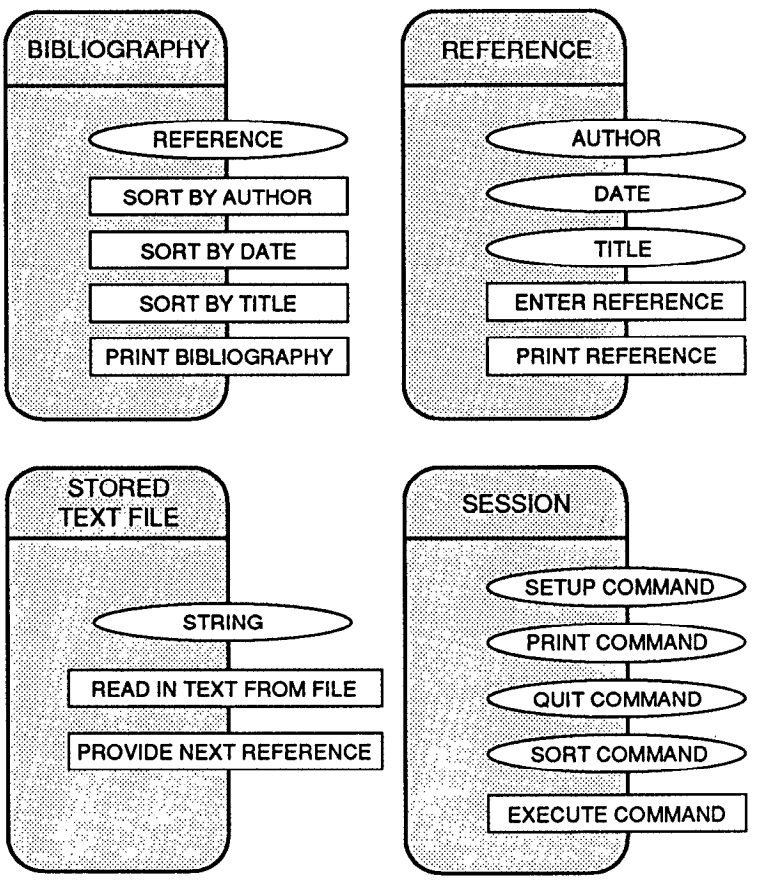


FIGURE 16. Further details of the object classes showing attributes and references to other object classes.

Productivity Concerns

Figure 18 shows schematically the predicted time and effort associated with the object-oriented model described above. It shows that different stages of the information system and software life cycle could occur concurrently. The result is that at any one time the same amount of effort could be going into the project as with the "linear" life cycle but that this effort would be distributed more evenly across the project stages. The result is a reduction in the time taken to complete the project, but not a reduction in the time spent in any one area. Indeed, it is likely that the class design and implementation could take considerably longer since genericity and code reuse is one of the aims of the object-oriented life cycle, in order that maintenance costs (often, unwisely, excluded from project development costs) can be dramatically reduced. Consequently, the traditional software metrics, either in terms of source lines of code or function points, will inevitably indicate a decrease of productivity since they do not address the subsequent phase of program maintenance. Thus it may be that traditional metrics are of little use to OOD. In terms of code and design elements there may now be two conceptual libraries to deal with: one for the project, storing revisions for the project under way and one for components that are fully generalized. This product library would be where the components produced following the cluster model of Figure 8 would eventually reside ready to be accessed for other developmental projects [32, 33]. In the object-oriented life cycle, more effort is required in the design and implementation phases and considerably less in program maintenance. Strategic planning is therefore required by an industry together with concomitant software metrics in order for the actual productivity gain foreseen for large systems developments using the object-oriented paradigm to be con-

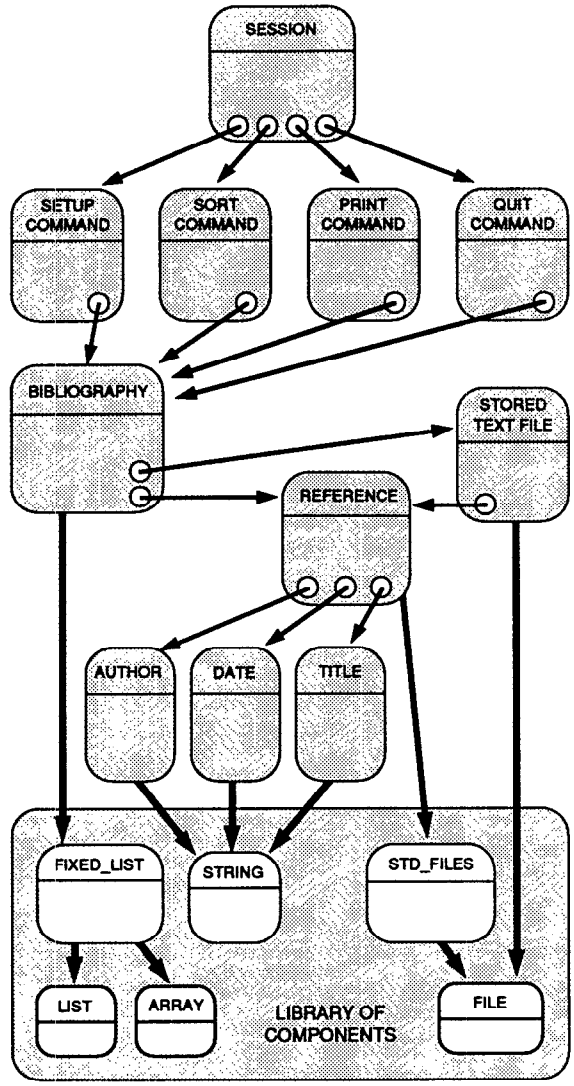


FIGURE 17. Inheritance and client-server relationships between object classes including base library object classes such as are supplied in some object-oriented programming environments e.g., Eiffel. (In general, depicting the relationship of objects to base, library classes such as string, would be an unnecessary detail).

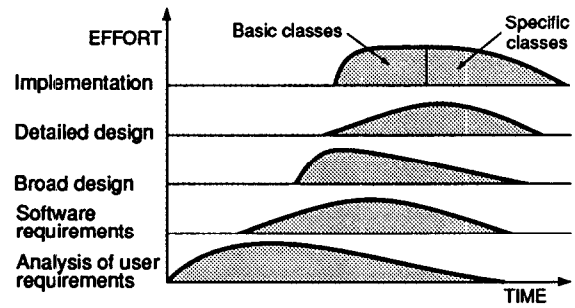


FIGURE 18. Effort as a function of time for the various stages in an object-oriented development.

sistently evaluated. It should also be noted that the object-oriented life cycle retains the advantages of the prototyping approach so the user can develop a better understanding of the system before signing off the specification documents.

Conclusions

There is no one object-oriented software model of the life cycle that has yet gained universal acceptance (e.g., [35]). However, by trying to maximize the advantages of the application of the object-oriented paradigm, the seven-step methodological framework seems appropriate. By using bottom-up design for classes at the same time as top-down object-oriented systems design, interactions can be encouraged so that the design is more robust and at the same time more flexible and the classes themselves are generically useful. Use of the same programming language environment from requirements analysis through to implementation also facilitates the coherent binding between the various processes in software development. The result should be a more maintainable system, a more extensible (flexible) system closer to the users' requirements and a system that requires overall less total time to produce and maintain.

Acknowledgments.

J.M. Edwards wishes to thank The Urban Water Research Association of Australia for financial support for the duration of this project. The authors are also grateful to the reviewers of this article.

References

1. Agha, G. Concurrent object-oriented programming. *Commun. ACM 33,* 9 (Sept. 1990).
2. Alabiso, B. Transformation of data flow analysis models to object-oriented design. In *Proceedings of OOPSLA '88* (1988) ACM, New York, pp. 335–353.
3. Bailin, S.C. 1989, An object-oriented requirements specification method. *Commun. ACM 32,* 5 (1989), 608–623.
4. Boehm, B.W. Software engineering. *IEEE Trans Comput.* C-25, (1976), 1226–1241.
5. Booch, G. Describing software design in

Ada. *Sigplan Not. in Ada, 6* 9 (1981), 42-47.
6. Booch, G. *Software Engineering with Ada.* Benjamin/Cummings, California, 1987, 580.
7. Borgida, A. Greenspan, S. and Mylopoulos, J. Knowledge representation as a basis for requirements specification, *IEEE Comput. 18,* 4 (1985), 82-101.
8. Boyd, S. Object-oriented design and PAMELA™: A comparison of two design methods for Ada. *Ada Lett. 7* (July-Aug 1987), 68-78.
9. Brookes, C.H.P., Grouse, P.J., Jeffery, D.R. and Lawrence, M.J. *Information Systems Design.* Prentice-Hall, Sydney, 1982, 477.
10. Cameron, J.R. An overview of JSD, *IEEE Trans. Softw. Eng. SE-12,* 2 (1986), 222-240.
11. Carver, D.L. and Cordes, D.W. An object-oriented framework to support architectural design development, In *Proceedings of Hawaii International Conference Systems Science,* v II, IEEE, (Los Alamitos, CA, 1990) pp. 349-357.
12. Coad, P. and Yourdon, E. *Object-Oriented Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1990, 240.
13. Constantine, L.L. *Object Oriented and Structured Design Seminar.* Digital Consulting Pacific, 1989.
14. Constantine, L.L. and Yourdon, E. *Structured Design.* Prentice-Hall, Englewood Cliffs, NJ, 1979.
15. Davis, A.M. A taxonomy for the early stages of the software development life cycle. *J. Syst. Softw., 8,* (1988), 297-311.
16. Davis, A.M., Bersoff, E.H., and Comer, E.R. A strategy for comparing alternative software development life cycle models. *IEEE Trans. Softw. Eng. 14,* (1988), 1453-1460.
17. DeMarco, T. *Structured Analysis and System Specification.* Yourdon Press, New York, (1978).
18. Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O. and Pintado, X. Class management for software communities, *Commun. ACM 33,* 9 (Sept. 1990).
19. Gindre, C. and Sada, F. A development in Eiffel: design and implementation of a network simulator. *J. Object Oriented Program. 2,* 1 (1989), 27-33.
20. Hawryszkiewycz, I.T. *Introduction to Systems Analysis and Design.* Prentice-Hall, New York, 1988, 373.
21. Jackson, M.A. *Principles of Program Design.* Academic Press, London, 1975.
22. Jackson, M.A. *System Development.* Prentice-Hall, London, 1983.
23. Jalote, P. Functional refinement and nested objects for object-oriented design. *IEEE Trans. Softw. Eng. 15,* 3 (1989) 264-270.
24. Korson, T. and McGregor, J.D. Understanding Object-Oriented: A Unifying paradigm. *Commun. ACM 33,* 9 (Sept. 1990).
25. Ladden, R.M. A survey of issues to be considered in the development of an object-oriented development methodology for Ada. *Ada Letters,* 2, (1989), 78-88.
26. Ledgard, H. and Marcotty, M. *The Programming Language Landscape,* Science Research Associates, Chicago, 1981.
27. Malhotra, A., Thomas, J.C., Carroll, J.M. and Miller, L. Cognitive processes in design. *J. Man-Mach. Studies 12,* (1980), 119-140.
28. Meyer, B. Bidding farewell to globals, *J. Object Oriented Program. 1,* 3 (1988), 73-76.
29. Meyer, B. *Object-Oriented Software Construction.* Prentice-Hall, Hemel Hempstead, 1988, 534.
30. Meyer, B. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming, 1,* (1989), 19-39.
31. Meyer, B. Programming as contracting, *Tech. Not. TR-EI-12/CO, Version 2,* Interactive Software Engineering, CA, 1989, 46.
32. Meyer, B. The new culture of software development: reflections on the practice of object-oriented design, In *Proceedings of TOOLS '89,* (Paris, November 13-15, 13-23, 1989).
33. Meyer, B. Tools for the New Culture: Lessons from the design of the Eiffel libraries, *Commun. ACM 33,* 9 (Sept. 1990).
34. Parnas, D. On the criteria to be used in decomposing systems into modules, *Commun. ACM, 15,* 12 (1972), 1053-1058.
35. Rajlich, V. Paradigms for design and implementation in Ada. *Commun. ACM 28,* 7 (1985), 718-727.
36. Reynolds, G.W. *Information Systems for Managers.* West Publishing Co., St. Paul, MN, 1988, 437.
37. Seidewitz, E. General object-oriented software development: background and experience, *J. Syst. and Softw.* 9, (1989), 95-108.
38. Seidewitz, E. and Stark, M. Towards a general object-oriented software development methodology. *Ada Letters,* 7 (July/August 1987), 54-67.
39. Shlaer, S. and Mellor, S.J. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press Computing Series, 1988, 144.
40. Snyder, A. Encapsulation and inheritance in object-oriented programming languages, In *Proceedings of OOPSLA '86,* ACM, New York (1986), pp. 38-45.
41. Sommerville, I. *Software Engineering,* 3rd ed., Addison-Wesley, Wokingham, 1989, 653.
42. Turner, J.A. Understanding the elements of system design. In *Critical Issues in Information Systems Research,* R.J. Boland, Jr. and R.A. Hirschheim, Ed. John Wiley, Chichester, 1987, 97-111.
43. Ward, P. How to integrate object orientation with structured analysis and design. *IEEE Softw.* (March, 1989), 74-82.
44. Ward, P. and Mellor, S. *Structured Development for Real-Time Systems,* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
45. Wegner, P. Learning the language, *Byte,* (March 1989), 245-253.
46. Wirfs-Brock, R.J. and Johnson, R.E. Surveying current research in object-oriented design, *Commun. ACM, 33,* 9 (Sept. 1990).
47. Wirfs-Brock, R.J. and Wilkerson, B. Object-oriented design: a responsibility-driven approach, In *Proceedings of OOPSLA '89,* (1989), 71-75.

CR Categories & Subject Descriptors: O.2.10 [**Software Engineering**]: Design—methodologies, representation

General Terms: Design

Additional Key Words and Phrases: Development life cycle; Object-oriented software systems life cycle

About the Authors:

BRIAN HENDERSON-SELLERS is Senior Lecturer in the School of Information Systems at the University of New South Wales. His current research interests include object-oriented systems development methodologies and notation; implementations of the object-oriented paradigm in the commercial environment; environmental decision support and simulation modelling. He is Convenor of the Object-Oriented Special Interest Group of the Australian Computer Society (NSW Branch).

JULIAN M. EDWARDS is a Ph.D. candidate at the University of New South Wales in the School of Information Systems and the Urban Water Policy Centre. He is investigating the application of the object-oriented approach to the economic evaluation of water treatment processes.

Authors' Present Address: School of Information Systems, University of New South Wales, P.O. Box 1, Kensington, NSW 2033, Australia.