Understanding object-oriented: a unifying paradigm

Communications of the ACM 33.n9 (Sept 1990): pp40(21). (13262 words)

Author(s): Timothy Korson and John D. McGregor

Document Type: Magazine/Journal

Abstract:

Traditional software development does not support iteration, reuse or a unifying model to integrate phases of the program life cycle, but the object- oriented paradigm addresses all these issues. Boundaries between phases are blurred in the object-oriented software life cycle because objects are the items of interest in all phases. Both the analysis and design phases identify objects and the relationships between objects, providing a continuity that creates a seamless interface between phases. The object-oriented development process is iterative, replacing the 'waterfall' model of software development with a 'fountain' model. Individual software components are designed to represent concepts that will eventually be executable forms; the Abstract Data Type (ADT) is the way the object-based paradigm captures the conceptual information.

Full Text:

The need to develop and maintain large complex software system in a competitive and dynamic environment has driven interest in new approaches to software design and development. The problems with the classical waterfall model have been cataloged in almost every software engineering text [19,23]. In response, alternative models such as the spiral [2], and fountain [9] have been proposed.

Problems with traditional development using the classical life cycle include no iteration, no emphasis on reuse, and no unifying model to integrate the phases. The difference in point of view between following data flows in structured analysis and building hierarchies of tasks in structured design has alway been a major problem [4]. Each system is built from scratch and maintenance costs account for a notoriously large share of total system costs.

The object-oriented paradigm addresses each of these issues.

A look at the object-oriented software life cycle, as described by Meyer [5], Coad and Yourdon [4], and Henderson-Sellers and Edwards [9], identifies the three traditional activities of analysis, design, and implementation. However, each of the referenced descriptions eliminates the distinct boundaries between the phases.

The primary reason for this blurring of boundaries is that the items of interest in each phase are the same: objects. Objects and the relationships between objects are identified in both the

analysis and design phases. Objects and relationships identified and documented in the analysis phase serve not only as input to the design phase, but as an initial layer in the design. This continuity provides for a much more seamless interface between the phases. Analysts, designers and programmers are working with a common set of items upon which to build.

A second reason for the blurring of these boundaries is that the object-oriented development process is iterative. Henderson-Sellers and Edwards further refine this idea by replacing the waterfall model of software development with a fountain model. Department reaches a high level only to fall back to a previous level to begin the climb once again.

As an example of the blurring of the traditional boundaries of the life cycle phases, Coad and Yourdon recommend that classification relationships between objects be captured and documented during the object-oriented analysis (OOA) phase. This classification will be directly reflected in the class inheritance structure developed in the design and in the code. This classifications in no way required in order to document the system requirements. In other words, Coad and Yourdon are recommending a traditional design activity in the analysis phase.

The blurring of the traditional design and implementation phases has been fueled by the development of encapsulation and abstraction mechanisms in object-oriented and object-based languages. For example, Meyer claims [14] that Eiffel is both a design and an implementation language. He goes on to say that software design is sometimes mistakenly viewed as an activity totally secluded from actual implementation. From his point of view, much is to be gained from an approach that integrates both activities within the same conceptual framework.

The object-oriented design paradigm is the next logical step in a progression that has led from a purely procedural approach to an object-based approach and now to the object-oriented approach. The progression has resulted from a gradual shift in point of view in the development process. The procedural design paradigm utilizes functional decomposition to specify the tasks to be completed in order to solve a problem. The object-based approach, typified by the techniques of Yourdon, Jackson and Booch, gives more attention to data specifications than the procedural approach but still utilizes functional decomposition to develop the architecture of a system. The object-oriented approach goes beyond the object-based technique in the emphasis given to data by utilizing the relationships between objects as a fundamental part of the system architecture.

The goal in designing individual software components is to represent a concept in what will eventually be an executable form. The Abstract Data Type (ADT) is the object-based paradigm's technique for capturing this conceptual information. The class is the object-oriented paradigm's conceptual modeling tool. The design pieces resulting from the object-oriented design

technique represent a tighter coupling of data and functionality than traditional ADTs. These artifacts of the design process used in conjunction with a modeling-based decomposition approach yield a paradigm, a technique, which is very natural and flexible. It is natural in the sense that the design pieces are closely identified with the real-world concepts which they model. It is flexible in the sense of quickly adapting to changes in the problem specifications.

Object-oriented remains a term which is interpreted differently by different people. Before presenting an overview of a set of techniques for the design process, we will give our perspective so the reader may judge the techniques in terms of those definitions. Briefly, we adapt Wegner's [27] definition for object-oriented languages to object-oriented design. The pieces of the design are objects which are grouped into classes for specification purposes. In addition to traditional dependencies between data elements, an inheritance relation between classes is used to express specializations and generalizations of the concepts represented by the classes.

As natural and flexible as the object-oriented technique is, it is still possible to produce a bad design when using it. We will consider a number of general design criteria and will discuss how the object-oriented approach assists the designer in meeting these criteria. We will refer to a number of design guidelines developed specifically for the object-oriented design paradigm and will discuss how these properties reinforce the concepts of good design.

The paradigm sprang from language, has matured into design, and has recently moved into analysis. The blurring of boundaries between these phases has led us to include topics in this article that are outside the realm of design, but which we consider important to understanding the design process. Since the paradigm sprang from language, we define the concepts basic to object-oriented programming in the following section.

Basic Concepts of Object-Oriented Programming

Five general concepts will be discussed in ths section: objects, clases, inheritance, polymorphism, and dynamic binding. The first three concepts appear in high-level design and analysis. The last two are added during low-level design and implementation.

Although these concepts are basic to object-oriented programming, the various object-oriented communities often associate different specifics with each concept. In this section we focus on the basics of the concepts. Later sections elaborate on the possible variations.

Of the five concepts, only one (inheritance) is a unique contribution of the paradigm.

It is the blending of inheritance with the other four concepts in specific ways that characterizes object-oriented programming. The paradigm is new enough, however, that there is not universal agreement on how to characterize it. Other definitions can be found in [5, 28].

In the following discussion we introduce and illustrate each of the five concepts using Eiffel syntax and semantics.

Figure 1 shows a group of related classes referred to as a subsystem or a class cluster [14]. This cluster implements all of the types necessary for implementing a simple hierarchical menu system.

objects

Objects are the basic run-time entities in an object-oriented system. Objects take up space in memory and have an associated address like a record in Pascal or a structure in C.

The arrangement of bits in an object's allocated memory space determines that object's state at any given moment. Associated with every object is a set of procedures and functions that define the meaningful operations on that object. Thus, an object encapsulates both state and behavior.

From a design perspective, objects model the entities in the application domain.

Classes

A class defines a set of possible objects. From the point of view of a strongly typed language, a class is a construct for implementing a user-defined type. For example, in Figure 2 the variables main_menu and sub_menu are both type MENU. Execution of the statement Main_menu.create dynamically allocates space for the object "Main_menu." Subsequent execution of the statement sub_menu.create dynamically creates a second instance of the MENU class.

Ideally, a class is an implementation of an ADT. This means that the implementation details of the class are private to the class. The public interface of such a class is composed of two kinds of class methods. The first kind consists of accessory functions that return meaningful abstractions about an instance's state. The other kinds of methods are transformation procedures used to move an instance from one valid state to another.

Information-hiding guidelines dictate that all data within a class be private. This is to guarantee that the interface of the class is in fact an abstraction.

Other languages provide a similar construct for creating an ADT. Ada, for example, has the package. A package differs from a class in that a package encapsulates the type but is not the

type itself. This is an important difference. It results in a weaker connection between state and behavior as well as the syntactic burden of an additional parameter to most of the package's procedures.

Some systems, such as Smalltalk, have run-time class-objects (meta-class objects). By this we mean that for each class in the system there is a corresponding object that stores information about the class as a whole and implements the class-level operations. Such objects are useful for storing dynamic information about the type, such as the number of current instances of the type, and provide an elegant way to implement operations such as class constructors and destructors, or maintain global information specific to the class.

Inheritance

Inheritance is a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes. Instance-based inheritance has also been studied and implemented [21], but we will limit this discussion to traditional class-based inheritance.

Inheritance is the most promising concept we have to help us realize the goal of constructing software systems from reusable parts, rather than hand coding every system from scratch. Procedural abstraction has worked well for some select domains, such as mathematical libraries, but the unit of abstraction is too small, the procedural focus not general enough, and the parameter mechanism too rigid.

Another construct that has been touted as holding a key to reusability is the concept of parameterized types or generics as applied to procedures, and Ada packages. The ability to create libraries of generic procedures and packages is a very useful one that has not been exploited nearly enough and will prove to be a central concept in the future of software development. It, however, is only applicable to strongly typed systems and is not as general a concept as inheritance.

Inheritance not only supports reuse across systems, but it directly facilitates extensibility within a given system. This theme will be elaborated upon throughout the article. We will show that inheritance minimizes the amount of new code needed when adding additional features, and that inheritance coupled with polymorphinism and dynamic binding minimizes the amount of existing code that must be changed when extending a system.

To understand exactly what inheritance is and how it provides the claimed benefits we introduce some basic terminology, elaborate on the inheritance relation, and then illustrate it with a simple example.

When class Y inherits from class X (see Figure 3), we will refer to class Y as a derived class and class X as a base class. In this case, class Y has two parts, a derived part and an incremental part. The derived part is the part inherited from X. The incremental part is the new code, written specifically for Y.

The mapping from the members of X to the derived members of Y is defined by the rules of the language in conjunction with code written by the programmer in class Y. The mapping may be a simple identity in which the derived part of Y is exactly the same as the members of X. This inheritance mapping may however, be much richer than a simple identity.

In general, a feature of X may be renamed, re-implemented, duplicated, voided, have its visibility changed, or under to almost any other kind of transformation as it is mapped from X to Y. As with any language feature, inheritance can be misused. What constitutes the proper use of inheritance is a widely debated topic. We favor a very disciplined use of inheritance.

Each object-oriented language has its own set of allowable inheritance mappings. At the language level, keywords are provided to indicate the kind of mapping desired. Certain kinds of mappings require additional information. For example, if a given member of X is to be re-implemented as it is mapped to Y, the code for reimplementation must be given.

The inheritance relation is often called the "is,a" relation. This is because when a class Y inherits from class X, class Y now has, by inheritance, all the features of X. Thus Y is an X. Y is undoubted more than an X, but in addition to whatever else it may be it is also an X.

Because of this, the inheritance relation is often used to reflect abstraction and structure present in an application domain. A commonly used example domain is from graphics. A rectangle is a special kind of polygon (see Figure 4). This relationship is easily captured by the inheritance relation. When rectangle inherits from polygon, rectangle gets all the features of a polygon. In addition, a polygon is a closed figure and so rectangle also inherits all the features of a closed figure.

Yourdon advised capturing this classification structure explicitly during the analysis phase. If a software system is developed using object-oriented analysis, object-oriented design, and implemented in an object-oriented programming language, then objects and classifications identified during analysis are preserved and enriched during design, and are directly implemented in code. We believe this ability to capture and encapsulate abstraction directly in code represents a major breakthrough in software technology.

At the highest level, all figures in a graphics system have a pixel width and color and the property of being able to be scaled, rotated or drawn. These properties are defined in the root class "Figure." Closed figures have, in addition, a perimeter and an area. Polygon adds the

attribute "number_of_sides." Thus, a rectangle has, by inheritance, a pixel width, scaling procedure, calculate_area function, etc. Some of these features can be inherited as is, others should be modified as they are inherited. The set_pixel_width as defined in the figure class may be appropriate for rectangles "as is" but the calculate_area procedure should be redefined to give a more efficient implementation.

In an object-oriented graphics system, inheritance minimizes the incremental effort of adding a new graphics primitive. In the same manner, the object-oriented approach with inheritance facilitates rapid prototyping.

The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in a way that does not introduce unwanted side effects into the rest of the class. Furthermore, the object paradigm focuses on identifying and encapsulating commonality in higher level abstractions. If these higher level classes are accumulated in a software repository in would eventually be quite likely that for almost any desired class, a generalization of that class would already exist in the software repository.

Polymorphism

There are many kinds of polymorphism [3], but in general polymorphism means the ability to take more than one form. In an object-oriented language, a polymorphic reference is one that can, over time, refer to instances of more than one class. Because of this ability to refer to more than one class of object, a polymorphic reference has both a dynamic and a static type associated with it.

The dynamic type of a polymorphic reference may change from instant to instant during the program execution. In strongly typed object-oriented environments, the run-time system keeps all polymorphic references automatically tagged with their dynamic type.

The static type is determined from the declaration of the entity in the program text. It is known at compile time and determines the set of valid types that the object can accept at runtime. This determination is made from an analysis of the inheritance graphs in the system.

The "is a" nature of inheritance is tightly coupled with the idea of polymorphism in a strong typed object-oriented language. The idea is that if Y inherits from X, Y is an X, and therefore anywhere that an instance of X is expected, an instance of Y is allowed.

Figure 5 provides an outline for discussion. After execution of statement 1, the static type of x is X and the dynamic type of x is X. After statement 4, the static type of x is still X but its dynamic type is now Y.

A more realistic example will help to clarify the use and importance of this concept. In a graphics system we may want the ability to perform some action (such as scale or rotate) on all currently displayed figures. It would facilitate this operation if a list of all displayed figures were available.

Referring to the inheritance diagram of Figure 4, one can see that the variable "current_figures: ARRAY [Figure]" provides just what is needed. Since Figure is an ancestor of Circle, the Eiffel type system knows that Circle is a Figure and allows the assignment: current-figures.put(C1:1); {where C1 is an object of type Circle). After this assignment the static type of current figures .item [1] would still be figure, but its dynamic type would be "circle." Likewise, the elements 2...n of the array could be Squares, Lines, or any other graphics primitive.

Dynamic Binding

The binding referred to in this section is the binding of a procedure call to the code to be executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the moment of the call at runtime.

In the object-oriented world, dynamic binding is associated with polymorphism and inheritance in that a procedure call associated with a polymorphic reference may depend on the dynamic type of that reference.

Suppose x is a polymorhic reference whose static type is X, but whose dynamic type may be X or Y. Suppose further that the procedure "do_it" is an operation defined in X but redefined in Y as a part of the inheritance mapping. The call x.do_it now depends on the dynamic type of x. If its dynamic type is X, as in statement 3 of Figure 5, the call will be bound to the code defined in class X. If its dynamic type is Y, as in statement 5 of Figure 5, the call will be bound to the code as redefined in class Y.

The graphics example, continued below, will illustrate the usefulness of this concept. In the next section we elaborate on the benefits of this concept relative to extensibility.

Suppose the class "Figure" in Figure 4 contains the procedure "draw" (with perhaps a null implementation). By inheritance, every graphics primitive in Figure 4 will have a procedure draw. The draw algorithm is, however, unique to each graphical shape, and so the draw procedure will be redefined in each class that defines a graphic primitive.

Returning to the polymorphic array of the previous section, one can now see the usefulness of polymorphism combined with dynamic binding. To redraw the entire graphics window the following code would suffice: For i: = 1 to Number_of_ Shapes do current_figures.item(i).draw;

At each pass through the loop, the code matching the dynamic type of current_figure.item(i) will be called. Note that if additional kinds of shapes are added to the system, this code segment remains unchanged. Contrast the resulting simplicity and extensibility as compared with a traditional case statement design.

Basic Design Concepts

This section presents the fundamentals of the object-oriented design phase of the object-oriented software life cycle. For a reference point object-oriented design is compared to the procedural design paradigm (top-down functional decomposition). The designs produced using the object-oriented paradigm are measured against a set of qualities of well-designed software.

A Different Point of View

Most computer professionals know the syntax of several programming languages. Most know several diagramming techniques for representing design details. Yet most know only one approach to system design. It is perhaps harder to learn a new system development technique than it is to learn a new language. This is because learning a language involves memorizing a few keywords and their valid arrangements while learning a new system development technique requires a fundamental change in our way of thinking.

A design paradigm is characterized by its view of the decomposition process. The procedural paradigm takes a task-oriented point of view, which begins its support for the design process when a solution to the target problem is proposed. The proposed solution is decomposed by breaking it into a sequence of tasks. These tasks form the basic building blocks for a procedural application. Information developed as part of the analysis process serves as input into the design phase but it is in a different terminology and represents a different perspective from the design phase. For this reason, the traditional view of an application life cycle includes what may be an unnecessary boundary between analysis and design. This boundary results from a shift from the problem domain in analysis to the solution domain in design.

The object-oriented design paradigm takes a modeling point of view. The analysis and design phases of the traditional life cycle, while remaining distinctly separate activities in the object-oriented life cycle, work together closely to develop a model of the problem domain. The model is constructed by viewing the problem domain as a set of interacting entities. The software-based models of entities and the relationships between them are assembled to form the basic architecture of the application. The information developed in the analysis phase becomes an integral part of the design rather than simply providing input into the phase. This smooth transition is facilitated by the homogeneity of the "pieces" being used by each process. As discussed in Coad and Yourdon [4] this homogeneity is in stark contrast to the difference in point of view between Structured Analysis and Structured Design.

The pieces produced by the procedural design paradigm are procedures that perform tasks. These pieces are artifacts of the design process and relate to the proposed solution. The pieces produced by the object-oriented paradigm are entity descriptions, classes. Many of these descriptions can be related back directly to the original problem. Although many of the classes do not represent physical objects, they are conceptual entities which can be stated in the terminology of the problem domain.

An Example for Comparison

Consider an example that will illustrate the different points of view of the procedural and object-oriented paradigms. Figure 6 presents an abbreviated statement of requirements for a software system which would control the traffic lights at the intersection of two streets. The application would have software interfaces to the hardware pieces like the car sensors and the traffic lights.

The procedural paradigm approaches the development of the traffic intersection control system by considering the sequence of tasks that must be performed. A functional decomposition of the proposed solution provides a set of tasks such as those in Figure 7. This hierarchy chart illustrates the process of step-wise refinement. Each layer in the hierarchy represents the identification of more detailed functionality. The tasks are sequenced by reading from left to right across the chart.

The object-oriented paradigm approaches systems development by identifying the entities that are present in the problem. The physical entities involved in this problem include the sensors, traffic lights, and the controller. The paradigm is not limited to physical entities. For this problem, one non-physical entity that can be identified is a "lane" entity. This entity associates a sensor with a particular traffic light. The lane entity bridges between the problem domain and the solution domain. It is named and discussed in the terminology of the problem, but may arise either through viewing the problem or the implementation of the controller which must have information about which light is associated with which sensor.

Abstractions of the entries will become the classes that form the foundation of the system. The specifications for these classes will supplement the information from the traditional requirements document. The object-oriented requirements document uses objects and classification to blend traditional analysis information about the required systems functionality with descriptions of the objects it manipulates. The systems specifications are useful to both the designers and users.

To the latter because the descriptions are not pieces of a solution but pieces of the problem.

To the former because the design will retain and build directly upon the objects identified during analysis, and this facilities communication between users and designers since users are better able to relate to the problem than to the computer-based solution to that problem.

The entities identified above and several others are illustrated in Figure 8. This figure uses an entity-relationship diagram to present the entities and the relationships between them. The diagram is the representation of a semantic data model of the problem. A semantic data model is so named because it allows representation of a broader range of information than other data models such as the relational model. Peckham and Maryanski [8] provide a survey of several semantic data models. Other techniques have been proposed for capturing this information (e.g., Wirfs-Brock [28]).

A comparison of the information in Figures 7 and 8 illustrates the difference in points of view of the two paradigms. Neither technique concentrates on one type of abstraction to the exclusion of all others. The procedural paradigm, while giving priority to procedural abstraction, must consider the structures necessary for data representation at some level in the design process. Likewise, the object-oriented paradigm may use functional decomposition in designing the operators needed to manipulate an object.

The object model supports the complete software development life cycle. Beginning in the requirements phase, objects are identified. By developing specifications of the entities found in the problem domain a clear and well-organized statement of the problem is actually built into the application. These objects form a high-level layer of definitions that are written in the terminology of the problem domain. During the refinement of the definitions and the implementation of the application entities, other entities, or classes, are identified. Figure 9 illustrates the layers of object specifications that result from this process.

The object-oriented analysis and the object-oriented design phases work more closely together because of the commonality of the object model. In one phase the analyst identifies problem domain objects while in the next phase, the designer specifies additional objects necessary for a specific computer-based solution. The design process is repeated for these implementation-level objects. The position of object-oriented design in the software development process is further detailed in the paper by Henderson-Sellers [9] appearing in this special issue.

entities

Object-oriented design is first concerned with entities--things. These things may be tangible objects such as traffic lights, chairs, or airplanes. The entities may be abstract concepts such as roles, interactions, or incidents. For example, an airplane and a pilot are tangible objects but these two entities may be bound together by the incident of a flight. The flight is also an entity

which has a state that describes the particular airplane and the particular pilot as well as passengers, cargo, etc.

Relationships

The relationships between entities are the incarnations that occur between objects in the problem space. A pilot flies an airplane. Flies is one relationship between pilot and plane. This relationship is an application-level relationship and is expressed in the terminology of the problem. Two entities may have more than one relationship between them. The pilot is_in_command_of the airplane. An entity may have relationships with several other entities. The pilot is_employed_by the airplane.

An important design relationship in the object-oriented paradigm is the inheritance relation. In order to describe a class of pilots, the designer might wish to begin with the description of class person. Using the inheritance mechanism, all the behavior and attributes of a person would be added to the pilot class definition. Linking two classes via the inheritance relationship means that future improvements to the person class will become improvements to the pilot class automatically.

A second design relationship in the object-oriented paradigm is the component relation. The description of person probably includes a name attribute. This attribute can be provided by declaring an instance of the string class in the definition of person. This instance of string is a component of the person class definition.

The distinction between these two relationships is an important one. Inheritance represents a specialization of the existing definition. That is, a pilot is a special kind of person, one who flies planes, but is still a person. A component is used to provide a service in the implementation of a class. A person is not a special kind of string. The instance of string is used to represent an attribute of person, not to define what a person is. The characteristics of a person become part of the characteristics of a pilot when added by inheritance, while the characteristics of a string are hidden in the implementation of a person.

Complete Data Model

Object-oriented design has two separate components which must be blended in the development of an application: class design and application design. Class design is "wrapped inside" of application design. The design of an application includes the identification of the kinds of entities in the problem domain plus those specific to implementing the solution such as menu systems or pointing devices. Each kind of entity leads to a class description which should be a complete model of one concept. Once these descriptions have been developed, the application can be designed. The application is developed by connecting instances of the

classes--modeling the real world so that they interact with each other resulting in a problem solution.

The class descriptions include three parts: definitions of attributes, description of the class interface, and the set of valid transitions between the possible states for an instance of the class. Consider the design of a stack class. Figure 10 shows the three parts of the class design. Part (a) of the figure shows that the data attributes for the stack class include a data store for the items contained in the stack and some means of locating the top, or visible, item in the stack. Part (b) gives the public interface of the class. Part (c) uses a state diagram to illustrate the sequence of valid states for the class. This diagram not only provides information about the states of the class, it provides constraints on the sequence of use of the operators from the class interface.

The pattern of interaction between instances of the classes provides the structure of the application. These interactions are models of the relationships between the classes. These relationships are modeled in one of several ways but they all involve messages. A message is a communication between two objects. The messaging may be either synchronous or asynchronous depending upon the execution model chosen. An object may send a message to any other object whose identity is available. The message can correspond to any operator in the public interface of the receiving object. The receiving object may be an object at the same design level or it may be one that is declared as part of the implementation of the sending object.

Semantic data modeling provides support for developing a complete data model of an application which incorporates both high-level application relationships and low-level class definition relationships. Figure 11 provides an entity-relationship diagram that illustrates the pieces of the pilot/plane application used above. All the entities and the relationships between them can be captured in this one model.

Thus, the development of an object-oriented application is a blend of class description and application configuration. Much of the process is guided by the real-world problem domain rather than our view of the problem solution. This results in an application design that, at a high level, contains a model of the problem domain. The application design process begins at a top level and proceeds through class identification to a low level and then moves upward as low-level classes are designed based on lower-level definitions.

Support for Good Design

Modularity. The object-oriented paradigm provides natural support for decomposing a system into modules. In this paradigm classes are the modules. This means that not only does the

design process support modularity, but the implementation process supports it as well through the class definition. Figure 12 illustrates a class definition for the pilot class.

Classes as module represent a fine-grained approach to system composition. Others have proposed a more coarse-grained approach. Meyer [15] defines a cluster as a set of classes that are conceptually related. This cluster of classes is a natural grouping to consider because an application, which includes a class representation of a concept, will usually need to include instances of a set of classes in order to fully implement the concept. Wirfs-Brock [29] defines a subsystem grouping and the work of Johnson, reported in the Wirfs-Brock article describes frameworks as other approaches to grouping classes. These groupings collect classes which as a group form an abstraction. For example, the concept of a linked list might be represented by three classes: a class that is the list itself; a class that describes the links; and a class that defines iterators for the list. Individually the classes are not very useful but together they provide the designer with a commonly used abstraction.

Information Hiding. The class construct supports information hiding through the separation of the class interface and the class implementation. The separation allows the class interface (i.e., specification) to be mapped to several different implementations. It also allows much maintenance activity to be hidden from users of the class.

This separation is in parallel to the design activity of separating the behavior of an object from the attributes of the object. The operators in the public interface represent the possible behaviors of the object. One of the responsibilities of these operators is to provide for the controlled access to the attributes of the object. The representation of these attributes and the corresponding implementation of the operators should be hidden from users of the class.

Figure 12 illustrates a usual division between public interface and private implementation. The interface operator tell_age hides the fact that the operator uses the person's date_of_birth and the system date as the basis for answering the request. In this case, the object's attribute age has been mapped to a representation using date_of_birth and associated operators to calculate current age.

Weak Coupling. Classes are designed as collections of data and the set of allowable operations on that data. Therefore, the interface operators of a class are inward-looking in the sense that they are intended to access or modify the internal data of the class. This leads to fewer connections between classes.

Interactions between classes come from two sources. The component relationship discussed previously results in a coupling between the two classes. If an instance of class A is declared within the implementation of class B, then instances of class B contain an instance of class A in their implementation. Operators of class B send messages to (invoke the interface operators of)

the instance of class A. In Figure 12, the name attribute is an instance of the string class. Person's print_name operator would carry out its task by sending a message to the name instance to print.

Interface functions of a class may take as parameters instances of another class. This coupling assumes the instance parameter will either provide information for the operation of that function or the instance will be modified by having one of its interface functions invoked.

Strong Cohesion. A class is a naturally cohesive module because it is a model of some entity. Object-oriented style guidelines require that for a function to be a member of a class it must either access or modify data defined within the class.

The inheritance mechanism could be viewed as weakening the cohesion of the module. The data and functions inherited from another class form a natural group separate from those data values and functions defined within the new class. However, the ultimate test of cohesion is met by the fact that all these pieces are brought together to represent one concept.

Abstraction. Liskov and Guttag [12] present two methods of abstraction: abstraction by specification and abstraction by parameterization. Object-oriented languages support both of these methods to varying degrees.

Abstraction by specification abstracts the specification of an entity from its implementation. This type of abstraction is supported by virtually every object-oriented language. The public interface of a class constitutes the specification of that class. The interface specifies the legitimate operators of the data contained in instances of the class. In most object-oriented languages, the implementation of these operators and the exact representation of the class's data elements are hidden from access if not from the view of those outside the class definition.

Abstraction by parameterization abstracts the type of data to be manipulated from the specification of how it is to be manipulated. This type of abstraction is supported by most object-oriented languages at the operator level but by only a few languages at the class level. Eiffel, for example, allows a class definition to be parameterized. Consider the definition of a stack class. The class can be totally specified without regard to the type of the items to be "stacked." The type of the items can be identified at compile time through the use of a parameter. Figure 13 shows a parameterized class definition in Eiffel syntax. This type of abstraction is only of interest in a typed-language environment.

Extensibility. The object-oriented paradigm produces designs that are easily extended. The inheritance mechanism supports extending designs in two ways. First, the inheritance relation facilitates the reuse of existing definitions to ease the development of new definitions. As the inheritance structure becomes progressively deeper, the amount of specification and

implementation inherited by a new class definition grows. This usually means that as the inheritance structure grows, the amount of effort to develop a new class decreases.

Second, the polymorphic property of the typing system in object-oriented languages also supports extensible designs. Figure 14 shows a simple inheritance hierarchy. Consider the operator x, shown in Figure 15, that takes an instance of class A as a parameter. The polymorphic behavior of object-oriented typing systems allows the substitution of instances of classes B, C, D, E, or F as the actual argument to operator x.

This polymorphic property supports the extension of existing systems in the following way. Assume further analysis reveals another specialization, G, would provide the new feature needed by the current application. Instances of this new class can be used as actual arguments to operator x even though the class G had not been thought of before the x operator was designed. Thus a new feature is added with little or no modification to the remainder of the application.

Integratable. The object-oriented design process produces designs which facilitate the integration of individual pieces into complete designs. The narrow clearly defined interface of a class supports integration with other software components. The set of allowable operations and their required parameters are easy to identify. The narrow interface corresponds naturally to the observable behaviors of the real-world entity modeled by the class. The interfacing of two classes then is a model of the natural interactions of the two entities.

Objects are also very integratable at the implementation level. The encapsulation provided by a class hides the implementation details from the remainder of the system. The hiding of data declarations, function-naming conventions, and complex control flows prevents interactions. Thus, integration of prewritten components is much less likely to require modification of existing code in either component.

Support for Reuse

The object-oriented paradigm combines design techniques and language features to provide strong support for reuse of software modules. The reuse comes in a variety of forms. Some of the reuse in the object-oriented paradigm is much the same as that in the procedural paradigm but the object-oriented paradigm adds an additional type of reuse.

Every time an instance of a class is created, reuse occurs. This is similar to the declaration of a variable of a specific type. The major difference is that the resulting class instance is a much more complex structure than a simple variable. An instance of a class provides a combination of data structures and operators on those data structures. Declaring the instance of the string class to represent the name attribute in the pilot class is an example of this type of reuse. Using

an instance of the string class provides operators for copying, concatenating, and, perhaps, even editing the string value.

Inheritance provides two levels of support for reuse. As part of the high-level design phase, inheritance serves as a means of modeling generalization/specialization relationships. These relationships appear in the form of classifications. A chair may be viewed as a special type of furniture as well as a general description of the more specific categories of rocking chairs, straight chairs, and reclining chairs. This high-level use of inheritance encourages the development of meaningful abstractions which, in turn, encourages reuse.

Often in actual design the presence of mid-level abstractions such as table and chair will be recognized and considered separately. The availability of an inheritance relation enables the designer to "push higher" and to identify commonality among abstractions and to produce higher-level abstractions, (e.g., furniture), from this commonality. By identifying this commonality and removing it to a higher abstraction, it becomes available to be reused later in the current design or in future designs. Filing cabinets and bookcases may be identified later. Much of their description (attributes such as height, weight, color, etc.), may already be available from the furniture abstraction. The benefits of this reuse prompt the designer to search for higher and higher levels of abstraction.

In the low-level design phase, inheritance supports the reuse of an existing class as the basis for the definition of a new class. An existing piece of code can be copied to a new file and modified to fit its new purpose. This "editor inheritance" does not establish any connection between the old piece of code and the new code. If a bug is discovered in the old code and repaired, knowledge of this change may or may not reach the person responsible for the new code. Inheritance provides an important improvement in this process.

Inheritance establishes a dependency between the existing class and the new class. This technique is nonintrusive in that the existing code is not modified. The new code in the subclass can not cause the existing code to "break." The inherited code is included in the new definition automatically as the new class definition is compiled. Any modifications to the original class, bug fixes or feature additions, are incorporated into the newer class at the next compilation. This technique allows a class to serve as the basis for many new definitions without propagating the errors of the original definitions throughout the system.

Much research focuses on support for reuse. Several of the research efforts presented in Wirfs-Brock [29] include consideration of reuse. Reenskaug's work with Role Models and Johnson's work with Frameworks address the need for conceptual groupings of classes. As libraries of classes grow very large, locating the classes that are needed to represent a given concept becomes a difficult task. These conceptual models associate classes forming larger more readily

recognized groupings. Lieberherr's work on the Law of Demeter approaches reuse from the point of view of the interactions between classes. The Law addresses the allowable dependencies between classes. For example, it explicitly eliminates the direct access by one object of another object's implementation. This reduces the number of dependencies between classes, increasing the design's flexibility.

Software Base

The set of classes together with the application and implementation relationships between them form a reusable resource for the designer which has been referred to as a software base. This software base further promotes reuse by overcoming one of the real-world obstacles to reuse: finding the appropriate class.

Large companies have communications problems. How, in a company of 5,000 designers, can information about existing resources be shared? The following section on development environments discusses tools that facilitate the location and use of specific classes. The article by Gibbs et al [6] in this issue discusses the problem of class management.

Design Guidelines

In any object-oriented application, instances of classes make up the majority of the system and, if a pure object-oriented approach is used, all of the system consists of instances of classes. Therefore the design of the individual classes has a major impact on the overall quality of the application. In this section a set of guidelines is presented for the design of classes. The intent is to illustrate the factors which should be considered when designing a class without discussing each guideline in detail. Figure 16 presents the list of guidelines which are briefly discussed below.

The first four guidelines address the proper form and use of the class interface. The information hiding required by the first guideline reinforces the development of representation-independent designs. This encapsulation is further specified in the second guideline which prohibits accessing class instances used as part of the representation of this class. These guidelines enforce the idea that a class is characterized by its set of operations and not by its representation. The third guideline defines the public interface as containing the complete set of public operations on the representation of the class. "Helper" functions are deferred to the protected area in a class definition. The fourth guideline requires that to belong to the class, each operator must represent a behavior of the concept being modeled by the class. Together these four guidelines give the designer directions for developing and separating the class interface and the class representation.

The second four guidelines consider the relationship of this class to other classes. Guideline number five constrains the designer to link a class with a few other classes as possible. If a class being designed will need many of the services of another class, perhaps that functionality should be part of the representation of the new class. The sixth guideline is intended to reduce, and perhaps eliminate, global information. Any information needed by one class should be explicitly passed to it in a parameter from another class. Guideline number seven prohibits the use of inheritance to develop the representation of a new class rather than its interface. The preferred method for utilizing an instance of a class as part of the representation of another class is to declare an instance of the supporting class in the representation of the newly designed class. Finally, the last guideline encourages the designer to develop inheritance structures of classes which are specializations of an abstraction. These abstractions lead to more reusable subclasses and to clear-cut differences between the subclasses. Articles by Johnson and Foot [10] and Lieberherr and Holland [11] expand on each of these guidelines.

Software Design and Development Environments

The object-oriented design process produces a large number of software components that are interconnected by a network of relationships. Use of the object-oriented paradigm relies much more heavily on the use of supporting tools and environments compared to procedural design techniques. This section presents information about some existing support systems and considers how these systems may look in the future. Many of the features of support systems discussed here are not limited to the object-oriented domain, but characteristics of object-oriented design make particularly good use of the support provided by such an environment.

Conceptual Tools

We have already described a number of conceptual tools for object-oriented design. We will summarize them briefly and include others that have not been explicitly discussed previously in this article.

Generalization/Specialization. These closely related concepts provide much of the power of the object-oriented paradigm. Specialization guides the designer in the reuse of an existing abstraction by defining a new class that is more specific than the existing class. The specification of the new class includes the specification of the existing class as a proper subset. Generalization supports the exploitation of commonalities between classes. When two or more classes are representing overlapping sets of attributes, the common attributes may be factored out of both classes and used to create a new class. The new class then becomes the superclass for those previously overlapping classes. The inheritance mechanism of an object-oriented language implements both of these tools.

Specialization does not require any modification of existing definitions. The new definition simply includes a reference to the existing one. Generalization usually does require modification of existing definitions. The definitions and declarations that are common among the classes are removed to a new higher-level definition. The inheritance mechanism then makes these pieces in the new high-level definition available to the definitions at the lower levels.

Components. One class may be used as part of the representation of another class. This type of reuse of design simply looks for a class that already represents some subset of the attributes of the new class. An instance of this class can then be used to provide that representation. Instances of several classes may provide much of the representation of a new class, making its realization a relatively quick process.

Contracts. This is a device defined in Wirfs-Brock [29] and Meyer [14] to describe the division of responsibility between two classes that interact. The contract specifies what capabilities one class will provide and what the other class can expect to receive when requesting one of the capabilities. The client/server model is a type of contract that has been used to group capabilities even in the procedural paradigm.

Clusters/Subsystems/Frameworks. Each of these concepts is a technique for recognizing sets of classes that are conceptually related. It is often logical to divide the representation of a concept among several classes. Some of these sub-concepts may be of use in the development of other concepts. In the above example of the linked list, by separating the list iterator into a separate class, we are able to have multiple iterators on the same list at the same time.

No syntactic unit exists that indicates this relationship. This is the most important place to apply a tool for support. The designer should be able to designate clusters in the software base. If a user calls up the linked list class, the environment should be able to inform the user that these other class are probably of interest as well.

These conceptual tools are very useful in the design process. They provide guidelines for the design and direction for the designer. Most existing development environments require that the designer be responsible for respecting the guidelines presented by these conceptual tools. A set of software tools that supports the use of these conceptual tools is needed. The work of Lieberherr reported in Wirfs-Brock [29] is one effort to develop tools that include a policy for design.

Levels of Access

The reuse of software components is a strong feature of the object-oriented paradigm. This reuse will only take place if there are good tools which adapt to the various ways that classes

are used. These tools will provide the designer with access to the class definitions, implementations, and structures of the software base. For our purposes, three levels of access to a class definition can be identified:

Access for generating instances. Users often reuse an existing class by creating instances of the class for an application they are designing. Accessing a class for this purpose requires that the designer have access to the public interface of the class. However, it should be the complete interface. That is, the attributes that are inherited from super classes up the inheritance structure should be presented as part of the complete interface. A design environment should be capable of combining information from multiple class specifications to present to the class user.

Access for creating new subclasses. Users accessing a class in order to use it as the basis for a new class definition may need more information than those users simply creating instances from the definition. In particular, some languages give operators written in the subclass access to some or all of the data or protected operators declared in the super classes.

The person creating the subclass may need to see the data declarations as well as the interface for the class(es) being accessed. Some of this "data" are actually instances of other classes declared as part of the representation of the current class. This implies that the user may need access to definitions for these classes to fully understand how a class functions.

Access for maintaining the class. Users accessing a class, in order to repair or extend it, need the maximum amount of information. Information is needed on all of the following: the interface, the representation, and the implementation. The environment should allow for wide-ranging browsing that lets the user select an operator to be edited, a data declaration to be examined, or a relationship to be followed. The environment should provide integrated editing and as much static checking as possible.

Existing Tools

Tools for design. A number of design tools are available which facilitate some part of the object-oriented design process. These tools can be divided into two categories: those that provide high-level design tools and those that assist with low-level design. No commercial tool that we are aware of supports both the development of a high-level design and takes that design to the lower levels of development. The work of Reenskaug reported in the research survey of this special issue [29] is an attempt to develop a methodology for the complete life cycle.

Tools such as Excelerator from Index Technology support the development of Entity-Relationship Diagrams. The Entity-Relationship Model is one of the semantic data models mentioned previously. An E-R Diagram provides a very flexible framework in that it can capture

any relationship that exists between classes. Excelerator provides an integrated data dictionary which allows much information about the classes and the relationships to be stored, managed, and manipulated. A number of other existing computer-assisted software engineering (CASE) tools support the development of the E-R diagrams.

Most of these tools do not "understand" the object-oriented paradigm. That is, they do not treat the application-specific relationships such as is_employed_by, any differently than the paradigm-specific relationships such as inheritance. It is the designer's responsibility to differentiate between the relations. While a tool dedicated to the paradigm can provide more intelligent support and "critique" the design as it progresses, a more general tool can support multiparadigm integration of systems.

Interactive Development Environments, Inc. has developed and markets a product which supports a variant of the object-oriented paradigm. Their design technique, reported in [26], is termed Object-Oriented Structured Design (OOSD). OOSD incorporates the features of object-oriented design with the functional approach of Structured Design. It is beyond the scope of this article to present a detailed critique of this methodology but the automated design tool, Software Through Pictures, developed to support their work is one example of a useful design tool.

Software Through Pictures provides a range of CASE tools. A series of editors provides the designer with the popular graphical notations: E-R diagrams, Data Flow diagrams, and a data structures editor which supports Jackson hierarchical data structures. The environment also supports automatic documentation and data dictionary facilities. All facilities are integrated and information entered in one tool is accessible in others.

The designs developed in Software Through Pictures can be "brought to life" using the automatic code generation facilities. Unfortunately, the product does not support generation in any object-oriented language. It does provide code generation in Ada, C, and Pascal. The code is not managed or manipulated by the environment. Because the system does not provide low-level design support, the designer has very little control over the code that is generated.

Tools for implementation. A number of languages are accompanied by toolkits which support the low-level design and implementation process. Smalltalk provides an integrated environment which combines a class browser, editor, and compiler. The class browser treats each class discretely and does not present the complete interface of a class. The browser does provide a structured list which shows the class inheritance structure. While navigating the class inheritance structure, a class may be selected, viewed, and modified in the integrated editor. When the designer completes the editing process, a pop-up menu in the editor window allows

the new code to be immediately checked compiled, and linked into the system. Development of a similar system for C++ (minus the compiler), has been reported in [20].

Tools of the Future

The two design environments described above are passive graphics editors in the sense that they record what the designer enters and display the information when needed. The language toolkit discussed in the same section is more active but does not include capabilities for high-level design functions. The useful tools of the future will integrate high- and low-level design processes. The tools will provide intelligent support for the design paradigms being used and will be an active "assistant" to the designer.

At a high level, in the object-oriented paradigm, this means support for recognizing and recording the relationships that exist between entities in the problem space. These relationships represent the interactions between complex entities and as a result are infinite in variety. This requires the flexible approach of the entity-relationship semantic data model that allows user-defined relations rather than a limited number of "standard" relations. The result at this level is a complex structure of dependencies. Many environments allow the designer to draw this class/application structure in the form of E-R diagrams or some other notation and retain only the attributes of the picture rather than the interconnections. This is insufficient. The tool should provide a representation for this structure which can be the basis for active manipulation. Baldassari et al. [1] present a CASE tool which uses a variant of Petri nets to provide this capability. PROTOB is directed at distributed systems but provides a hint of what a general-purpose tool should include.

The concept of a software base [24, 25] of reusable components will require the support of a comprehensive set of tools. These tools should provide the user with a software base navigator that can follow any of the relation links developed between classes. Such a tool must be very general because the relationships present in an object-oriented design span from application-oriented relationships to standard relationships such as inheritance.

The toolkit should include low-level tools which operate on both the classes and the relationships between them. This includes the "hierarchy flatteners" described above as well as in-line editors and syntax checkers. All of these tools should be incorporated into the overall environment.

The existing tools for object-oriented design are inadequate. A number of simple tools assist in recording design decisions but these tools do not facilitate the design decisions themselves. Although the object-oriented paradigm simplifies the solution of complex problems, a comprehensive set of tools will greatly assist the designer.

Implementing an Object-Oriented Design

The software community has long realized that iteration is inherent in the software life cycle. Since the software development team will be constantly moving back and forth across the different phases, the joints between what we have traditionally called analysis, design and implementation should be seamless.

Therefore it is important to have object-oriented languages and development environments that directly support the object-oriented design paradigm and that make a smooth transition between design and implementation.

The object-oriented design paradigm exists independently from any specific implementation environment, and yet there are implementation issues that affect the design as well as the development process. This section explores some of these implementation issues. In addition, the sidebars by Jordan and Kilian provide case studies on the implementation decisions taken by two major object-oriented languages: C++ and Trellis.

As a parenthetical remark we note that in some environments one may wish to do an object-oriented design, but implement in a non-object-oriented language. High-level design should be language independent but it is not paradigm independent. A truly object-oriented design can be directly implemented only in an object-oriented language. What can be done in this situation is to transform the object-oriented design into an object-based design (remove inheritance) and then implement in a traditional language. We view this transformation as part of the design process, but others view it as an implementation technique.

Newer languages like Ada or Modula 2 have language constructs that directly support the encapsulation of an ADT. In other languages like C or COBOL the implementation of an object-based design requires programmer adherence to a set of coding standards that are unenforceable by the compiler.

Specific Issues

As stated in the introduction to this section of this article, there are many facets to the object world. Some techniques and concepts such as delegation are often considered under the object-oriented umbrella, but this section considers only the variations within traditional class-based inheritance systems.

Typing. Programming languages are commonly classified according to the extent of type checking provided at compile time. Object-oriented languages vary widely on this axis. On the one extreme Smalltalk is essentially a typeless language, whereas Eiffel is a very strongly typed language.

In object-oriented languages, type checking is applied at the level of an object. Objects not only have data, but associated operations. Type checking in the object world must therefore be associated not only with the interpretation of data, but with determining which operations may be applied to an object.

In a strongly typed object-oriented system, the only messages that are allowed to be passed to an object are those that can, from a syntactic analysis of the source code, be guaranteed to be resolvable at runtime.

This is not true of Smalltalk. The absence of a type system means that any message may be attempted with any object. If, at runtime, a message is applied to an object that does not know how to respond, then a runtime error occurs.

The weakly typed, interpreted environments exemplified by the Common Lisp Object System (CLOS) and Smalltalk have traditionally been associated with rapid prototyping [17] whereas the strongly typed, compiled languages such as Eiffel and C++ are usually chosen for production environments.

Dynamic binding. In the previous section, the relationship between a language's typing system and dynamic binding were considered. Some typed languages, however, impose further constraints. C++ requires that a function be declared as "virtual" in order for it to be a candidate for dynamic binding. This limitation may have some efficiency benefits, but makes for less-modifiable software components by restricting which operations on a type can be modified in a sub-type. Polymorphism. Even in a strongly typed object-oriented language, the typing system is more flexible than in a language like Ada or Pascal. As explained in section 2, entitled "Basic Concepts of Object-Oriented Programming," the user-defined inheritance hierarchy can be taken by the compiler to define a sub-typing hierarchy.

Suppose that class Y inherits from class X. if the language treats Y as a sub-type of X, then compiler will allow an instance of Y wherever an instance of X is expected. This allows for an entry in the polymorphic array of menu_items in Figure 2 to refer to any of the specialized menu items (including a menu).

Not all languages treat every user-defined inheritance relation as defining a sub-type. This imposes constraints on the use of polymorphism. These variations and their impact on polymorphism are further explored in the next sub-section.

Inheritance. Issues here include single versus multiple inheritance, single versus multiple inheritance graph(s), information hiding in inheritance, and sub-typing versus implementation.

Multiple Inheritance: Many object-oriented languages allow for a class to have more than one direct parent. This facility is especially useful when a class belongs to more than one classification scheme. Suppose that a system was being developed for the National Wildlife Federation, and a "Leopard" class was being defined. The designers may already have class definitions for "cat" and "endangered." Since a leopard is both a cat and an endangered species, it would be natural to inherit from both classes. Eiffel has always had multiple inheritance, but only the most recent versions of C++ have added this ability. Some languages such as Turbo Pascal V5.5 still only allow for single inheritance.

Number of Inheritance Graphs: Some languages define a single root class from which all other classes automatically inherit. This language-defined root class thus bind all user-defined classes together into a single inheritance graph structure and is used to provide a set of services common to all classes in the system. One such service, provided in Eiffel's root class adds the ability for each object to make a deep (recursive) clone of itself.

In a strongly typed language, the dynamic type of an object reference is constrained by the programmer-defined class inheritance hierarchies in that system. In a language where every class automatically inherits from a single root class there is no constraint on the dynamic type of a reference whose static type is the root class.

The presence of an automatic parent to every class provides a simple, elegant solution to some otherwise messy problems. It allows a language to remain small and conceptually clean, yet all the same time extensible. Smalltalk has a single root class, C++ does not.

Information Hiding in Inheritance. Inheritance is a form of coupling between classes. Normally modules in a software system should be loosely coupled. Should inheritance be an exception to this rule? Should inheritance represent tight coupling, or loose coupling? Is information hiding incompatible with inheritance?

In Eiffel, inheritance represents tight coupling with no information hiding. Other languages, such as C++, allow for information hiding within inheritance, and for variable levels of coupling between a base and derived class. This issue has been widely discussed in the literature [14, 22].

Sub-Typing versus Implementation. Inheritance can be used for either sub-typing or implementation. All of the examples used thus far have demonstrated the use of inheritance for sub-typing: A leopard is a sub-type of a cat. A toggle_item is a sub-type of a menu_item. Inheritance for sub-typing corresponds to the "is a" nature of inheritance described earlier.

Figure 17 shows inheritance for implementation. A stack is not an array, but an array can be used for the implementation of a stack. Some authors [8, 13] discourage the use of this type of

inheritance because it introduces complex implementation dependencies into the software system. Furthermore, this practice, in the presence of polymorphism and dynamic binding, leads to a design in which information hiding can be violated.

Notice, in Figure 17, that by the use of polymorphism the stack protocol can be subverted. C++ avoids this kind of problem by allowing a base class to be declared as "private" in the inheritance clause. This signifies to the compiler that inheritance for implementation is intended. The compiler then disallows any corresponding backward polymorphic assignments.

Figure 13 shows an alternate implementation of an array-based stack. In C++ in-line functions can be used to avoid the extra procedure calls and thus achieve the same efficiency that was sought by inheritance in Figure 17.

Objects. Object-oriented languages differ in the kind of run-time objects that they manipulate, and in how these objects are created and destroyed.

Objects as Instances of Classes: Class instances are the basic objects in an object-oriented system. All object-oriented languages allow for dynamic object creation. Dynamically created objects are referenced through a pointer (either explicitly or implicitly) and have the associated problems of aliasing, etc. This means that object-oriented systems must deal with the issue of the management of objects in memory.

Some languages allow objects to be created statically or dynamically. In C++, this results in having a separate syntax for pointers to objects. Although statically created objects may save one level of indirection, many systems, such as MODSIM II, do not even allow for them, and in most of those that do they cannot be associated with polymorphism and dynamic binding.

For dynamically created objects, the language system automatically handles allocation and deallocation of memory for the object. Initialization of the object's data, and memory management within an object are treated differently from one system to another.

C++ has class constructors which deal with an object's data initialization and initial dynamic allocation of any memory within an object. Class constructors are implicitly called whenever a class variable declaration is encountered. When an object passes out of scope or is explicitly deleted, the class destructor is automatically called and must de-allocate any dynamically allocated memory internal to the object. Eiffel has no class destructors, because it has automatic garbage collection. Eiffel constructors must be explicitly called to create and initialize an object.

Other "Objects": In addition to class instances, an object-oriented language may allow other types of run-time entities. In the second section of this article we introduced the idea of a class-

object. This object is not a class instance, but more like an accessible class descriptor. Another way of looking at it is that each class has a corresponding meta-class that describes the class. A class-object is an instance of this meta-class. Uses for such a class-object were given in that section.

In Smalltalk all run-time entities are objects. Even integers are instances of the integer class. Other languages allow some primitives such as an integer to be implemented in the traditional way. This saves memory space and execution overhead as every integer does not have to have a pointer to its class descriptor, etc. Other systems treat integers conceptually as objects, but handle them differently at the compiler level for efficiency reasons.

Encapsulation

Most object-oriented languages provide direct support for some form of information hiding within a class. Figure 13 shows the use of the export clause to support information hiding. In Eiffel, every feature of a class is automatically private, and thus unavailable outside of the class, unless it is specifically included in the class' export clause. C++ addresses information hiding with the language constructs of private, public, protected, and friend.

Full support for encapsulation requires that the programmer have complete control over what features of a class are a part of the public interface. In some versions of Smalltalk, encapsulation is limited in that all methods are automatically public and all data private.

Encapsulation is such an important part of the object-oriented paradigm that, according to some definitions, a language is not object-oriented unless it provides for encapsulation.

Persistence: Some object-oriented languages, such as C++ and Turbo Pascal V5.5, have no direct way to store an object. Users of these languages must either manage their own object i/o (a nontrivial task) or purchase a commercial object-oriented database system.

In Smalltalk and CLOS the entire current execution state can be saved to disk. Yet, individual objects cannot be saved to an external file through the facilities of the basic language.

Other languages provide object i/o to disk in the same way that Pascal and COBOL provide record i/o. An environment can add this functionality without altering the language definition or adding keywords by adding an object storage management class to the class library. An object needing persistence would then inherit from the "storable" class. This is the approach taken by Eiffel.

The OOPS class library [7] uses a single root class to provide basic persistence capabilities and guarantee uniform i/o semantics.

Keyed access to persistent object may also be provided in a language.

Pure versus hybrid. Trellis, Eiffel and Small talk are pure object-oriented languages. They are pure in that they not only support the paradigm, they enforce it.

One way in which some languages enforce the use of the object-oriented paradigm is by requiring that all functions, procedures and variables be declared internal to some class. In such a system there are no global variables, and no free-standing functions or procedures. All state is encapsulated in an object, and all code is executed relative to some specific object. The main program becomes the constructor in the class that defines a specific application.

C++, Object-Pascal, CLOS are hybrid languages. They are object-oriented extensions to base languages that were designed to support a different paradigm. As such, C++ is a multi-paradigm language whereas Smalltalk is a single paradigm language.

"Purists" claim tha the hybrid languages are a detriment to the paradigm in that groups using a hybrid language may claim to be doing object-oriented programming when, in fact, they may be doing 90 percent of their design in the paradigm of the base language. When the benefits of the object-oriented paradigm are not realized on such projects, managers will become disillusioned with the "object-oriented" approach.

Proponents of hybrid languages see them as a natural migration path into the paradigm. Programmers skilled in the base language can immediately use the object-oriented version and gradually learn the new features and paradigm. Proponents also claim that no all problems are best solved by a single tool, thus the availability of multiple paradigms is desirable.

History Lessons

There is much that has been learned about software development that is orthogonal to the object-oriented paradigm but that nevertheless ought to be included in any modern programming language and environment.

All object-oriented languages provide classes and objects, but some of them do not provide any construct for information hiding within an object. Turbo Pascal 5.5 is particularly bad in this aspect.

Eiffel provides a generic facility. This is such an important construct for supporting reuse that every strongly typed language ought to support it. Unfortunately, few do. Some of the compilers are very fast, others are very slow. Some, like Smalltalk have large libraries and are integrated into a sophisticated development and testing environment. Others provide only a bare-bones compiler.

The field is so new, however, that it should not be judged too harshly. Commercially, C++ is only 3 years old, Smalltalk-80 and Objective C only 6 years old. Admittedly, Simula has existed for more than 20 years, but mostly in a research setting.

Conclusion

The purpose of this article has been to introduce terminology, concepts, and basic techniques surrounding the object-oriented paradigm. The emphasis on data in system design has increased greatly over the past several years and has led to a number of data-driven techniques. By considering a definition that includes objects, classes, and class inheritance, we have tried to focus on a set of concepts that we view as a paradigm separate from the object-based paradigm being supported by the Ada and Modula-2 communities.

The design philosophy of the object-oriented paradigm takes a modeling point of view. This allows the designer to work with one approach which begins in the problem domain and transitions naturally into the solution domain. By building a model of the problem into the application system, the resulting design is more responsive to changes in knowledge about the problem situation. The modularity of these design and information-hiding capabilities of most object-oriented languages contribute to the technique's responsiveness to modifications.

A number of languages support the object-oriented paradigm, and the variability in language features affects the design process. Although most object-oriented languages now support multiple inheritance, the inheritance mechanism varies greatly from one language to another. A number of languages are available in a variety of development environments which facilitate the systems development process. We have detailed a number of language differences, but we believe that ultimately it is the richness of this development environment and supporting software base that will have the largest impact on productivity.

The articles in the remainder of this special issue will provide a detailed view of many topics that we have merely introduced. The Proceedings of ACM's Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA) is a rich source of material as well.

References

[1] Baldassari, M., Bruno, G., Russi, V. and Zompi, R. PROTOB, a hierarchical object-oriented CASE tool for distributed systems. In ESEC '89, 2d European Software Engineering Conference Lecture Notes in Computer Science no. 387, Springer-Verlag, 1989.

[2] Boehm, B. A spiral model of software development and enhancement. Computer, (May 1988), 61-72.

[3] Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. Comput. Surv. 17, 4 ACM, N.Y. 471-522.

[4] Coad, P. and Yourdon, E. Object-oriented Analysis. Yourdon Press, 1990.

[5] Cox, B. Object-oriented Programming: An Evolutionary Approach. Addison-Wesley, New York, 1986.

[6] Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O., Pintado, X. Class management for software communities. Commun. ACM 33, 9 (Sept. 1990).

[7] Gorlen, K. An object-oriented class library for C++ programs. Softw. Exp. and Prac. 17, 12, 899-922.

[8] Halbert, Using types and inheritance in object-oriented programming. JEEE Softw. (Sept. 1987), 71-79.

[9] Henderson-Sellers, B. and Edwards, J.M. The object-oriented systems life cycle. Commun. ACM 33, 9 (Sept. 1990).

[10] Johnson, R.E. and Foot, B. Designing reusable classes. J. Object-oriented Program. (June-July 1988), 22-35.

[11] Lieberherr, J.J. and Holland, I.M. Assuring good style for object-oriented programming. IEEE Softw. (Sept. 1989), 38-48.

[12] Liskov, B., and Guttag, J. Abstraction and Specification in Program Development. MIT Press, 1986.

[13] McGregor, J.D. Object-oriented Software Design and Development. Van Nostraad Reinhold. To be published.

[14] Meyer, B. Object-Oriented Software Constraction. Prentice-Hall, 1988.

[15] Meyer, B. The new culture of software development: Reflections on the practice of object-oriented design. In Proceedings of the First International Conference on the Technology of Object Oriented Languages and Systems (1989), pp. 13-23.

[16] Meyer, B. Lessons from the design of the Eiffel libraries. Commum. ACM (Sept. 1990).

[17] Mullin, Object-oriented Program Design with Examples in C++. Addison Wesley, 1989.

[18] Peckham, J. and Maryanski, F. Semantic data models. ACM Comput. Surv. v. 20, n. 3, 1988, 153-189.

[19] Pressman, R.S. Software Engineering. McGraw-Hill, 1987.

[20] Raghavan, R., Ramakrishnan, N., and Strater, S. A C++ class browser. In Proceedings of the C++ Workshop, USENIX Association (1987), pp. 274-281.

[21] Sciore, E. Object specialization. ACM Trans. Inf. Syst., 7, 2 (April 1989), 103-122.

[22] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. In Proceedings of OOPSLA (1986), pp. 38-45.

[23] Sommerville, I. Software Engineering. Addison-Wesley, 1989.

[24] Tsichritzis, D., Ed. Active Object Environments. Centre

Universitaire D'Informatique, Geneve Switzerland, 1987.

[25] Tsichritzis, D., Ed. Object Oriented Development. Centre Universitaire D'Informatique, Geneve Switzerland, 1989.

[26] Wasserman, A.I., Pircher, P.A., Muller, R.J. Concepts of object-oriented structured design. In Proceedings of TOOLS '89 (1989), pp. 269-280.

[27] Wegner, P. Dimensions of object-based language design. In Proceedings of the Conference on Object-Oriented Systems, Languages, and Applications (1987).

[28] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. Designing Object-Oriented Software. Prentice-Hall, 1990.

[29] Wirfs-Brock, R. and Jonnson, R. Surveying current research in object-oriented design. Commun. ACM 33, 9 (Sept. 1990).

TIM KORSON is an assistant professor of computer science at Clemson University in Clemson, South Carolina. He has worked at the Software Engineering Institute as a Visiting Scientist and serves as a consultant to AT&T. His current research interests include developing metrics and accounting systems for the management of information assets with emphasis on managing a corporate transition to object-oriented technology.

JOHN D. McGREGOR is an associate professor of computer science at Clemson University. He has worked at Lawrence Livermore National Laboratory and is a consultant for AT&T Bell Laboratories. He is an ACM Lecturer and recently has been program chair of ACM's Computer Science Conference and general chair of the National Educational Computing Conference.