A Survey of Object-Oriented Concepts   *

Oscar Nierstrasz

University of Geneva†

Abstract

The object-oriented paradigm has gained popularity in various guises not only in programming languages, but in user interfaces, operating systems, databases, and other areas. We argue that the fundamental object-oriented concept is *encapsulation*, and that all object-oriented mechanisms and approaches exploit this idea to various ends. We introduce the most important of these mechanisms as they are manifested in existing object-oriented systems, and we discuss their relevance in the context of modern application development.

1   Introduction

The explosion of interest in object-oriented approaches in the last few years has led to a proliferation of definitions and interpretations of this much-used and much-abused term. As a consequence, it can be very difficult for a newcomer to understand and evaluate what is meant when it is claimed that a programming language or a piece of software or a user interface is "object-oriented." Do they all mean the same thing or not?

In this chapter we shall survey object-oriented approaches as they are manifested in programming languages and systems today. We shall see that what these approaches have in common is that they all exploit *encapsulation* or "packaging" in various interesting ways. Encapsulation has been traditionally important in computer science for the simple reason that it is necessary to decompose large systems into smaller encapsulated subsystems that can be more easily developed, maintained and ported. Object-oriented languages and systems formalize encapsulation and encourage programming in terms of "objects" rather than "programs" and "data." Each of these approaches adopts a particular object model, depending on which properties of objects they need to encapsulate. A complete definition of what it means to be object-oriented is therefore not possible, though we can perhaps judge when one system or language is "more" object-oriented than another.

In the following sections we shall first survey object models as manifested by various object-oriented programming languages. Object-oriented concepts such as instantiation via object classes, class inheritance, polymorphism, genericity, and strong-typing in object-oriented languages will be shown to depend ultimately on object encapsulation. We shall then briefly consider systems that provide run-time support for objects and for programmers building object-oriented applications.

2   Object-Oriented Programming Languages

The first appearance of the notion of an *object* as a programming construct was in *Simula*, a language for programming computer simulations [8]. This is not so surprising, since it is quite natural to directly model the objects of a simulation as software "objects." More surprising is the discovery that software objects can be useful not only for programming simulations, but also for prototyping and application development. This is the direction that was pursued by the Smalltalk system [16], building upon the concept of an *object class* introduced in Simula.

Since object-oriented programming has been popularized mainly through the Smalltalk effort, it is extremely tempting to adopt a *de facto* definition of an object-oriented programming language as one that supports both object classes and class inheritance (discussed below). We feel that this view is too restrictive, however, since there are many arguably object-oriented approaches that do not depend on class inheritance. We therefore suggest that any programming language that provides mechanisms that can be used to exploit encapsulation is (at least to some degree) object-oriented. By using such a loose definition, we do not feel obliged to answer difficult questions like, "Are Ada and Modula object-oriented?" Instead, we say that you should ask, "In what *ways* are Ada or Modula object-oriented (or not)?" By analogy, it is not so interesting whether Prolog is "declarative" or "procedural," but in which ways it is declarative, and where the paradigm breaks down.

In passing, we should point out that another important way in which one language can be "more" object-oriented than another is in how *homogeneous* the object model is. Is "everything" an object? Are object classes themselves objects? Is there a distinction between "user objects" and "system objects," or between "active objects" and "passive objects"? These distinctions are important if we wish to apply object-oriented mechanisms (like class inheritance) and discover that they are not valid for certain kinds of "objects."

Object models for programming languages often encapsulate objects in terms of a set of *operations* as a visible interface, while hiding the object's *realization* (i.e., its data structures and the implementation of the operations). To emphasize object independence, one often speaks of objects as communicating by *message passing*. This is not so much an implementation strategy as it is a paradigm for communication: one may not manipulate or view an object's hidden data; instead one sends a "mes-

sage" to an object, and the object itself selects the *method* by which it will react to the message.

Once objects are encapsulated in this fashion, we can exploit encapsulation to provide, for example, the possibility of multiple object instantiation, behavioural sharing through various inheritance mechanisms, verification of correct object usage through strong-typing, and structuring of resources in concurrent applications. Object-oriented programming languages make it *easier* to program with objects by providing language constructs for defining useful kinds of objects. Our discussion will center in turn on the issues of software reusability, object types, and concurrency.

2.1 Reusability

Encapsulation of procedures, macros and libraries has been exploited for many years to enhance the reusability of software. Object-oriented techniques achieve further reusability through the encapsulation of programs and data. The techniques and mechanisms we shall discuss here are primarily concerned with paradigms for packaging objects in such a way that they can be conveniently reused without modification to solve new problems.

Instantiation and Object Classes

Instantiation is perhaps the most basic object-oriented reusability mechanism. Every programming language provides some built-in data types (like integers and floating-point numbers) that can be instantiated as needed. Objects may either be statically or dynamically instantiated. Statically instantiated objects are allocated at compile-time and exist for the duration that the program executes. Dynamically instantiated objects require run-time support for allocation and for either explicit deallocation or some form of garbage collection.

The next step is to provide a way for programmers to define and instantiate their own objects. This can be done by providing the programmer with a facility to define *object classes*, as is the case in Smalltalk. An object class specifies a set of visible operations, a set of hidden *instance variables* and a set of hidden *methods* which implement the operations. The instance variables can only be modified indirectly by invoking the operations. When a new instance of an object class is created, it has its own set of instance variables, and it shares the operations' methods with other instances of its class.

A simple example is the class *ComplexNumber*. The programmer would define an interface consisting of the arithmetic operations that complex numbers support, and provide the implementation of these operations and the internal data structures. It would be up to the programmer to decide, for example, whether to use a representation based on Cartesian or polar coordinates.

An alternative approach to instantiation is to use *prototypical objects* [19] rather than object classes as the "template" from which new instances are forged. This is exactly what we do when we make a copy of a text file containing a document composed in a formatting language like TeX ~ or troff: we reuse the structure of the old document, altering its contents, and possibly refining the layout. This approach is useful to avoid a proliferation of object classes in systems where objects evolve

rapidly and display more differences than similarities. The difference between object classes and prototypical objects is brought out sharply when viewed in terms of applicable inheritance mechanisms (discussed next).

Inheritance

Inheritance has many forms depending on what we wish to inherit and when and how the inheritance takes place. In most cases, however, inheritance is strictly a *reusability* mechanism for sharing behaviour between objects, not to be confused with *subtyping*, which will be discussed in a later section. (Many of the "problems" with inheritance arise from the discrepancy between these two notions.) The differences between the various forms of inheritance can be loosely summed up in terms of the following issues:

- Does inheritance occur statically or dynamically (at runtime)?
- What are the clients of the inherited properties? (I.e., classes or instances of classes?)
- What properties can be inherited? (E.g., instance variables, methods, rules, values, etc.)
- Which inherited properties are visible to the client?
- Can inherited properties be overridden or suppressed?
- How are conflicts resolved?

We shall proceed with a brief overview of the more common kinds of inheritance, starting with class inheritance, and concluding with what we will call *dynamic inheritance*.

Class inheritance is often represented as the fundamental feature that distinguishes object-oriented from other programming languages. Although this may be a useful and simple guide it over-emphasizes the importance of just one aspect of object-oriented programming and therefore undercuts the contributions of other languages that do not provide an explicit mechanism for class inheritance. Nevertheless, class inheritance is an important mechanism which, when properly applied, can simplify large pieces of software by exploiting the similarities between certain object classes.

The key idea of class inheritance is to provide a simple and powerful mechanism for defining new classes that inherit properties from existing classes. With *single inheritance*, a *subclass* may inherit instance variables and methods of a single *parent* class, possibly adding some methods and instance variables of its own. Suppose, for example, that we want to display our complex numbers on a two-dimensional grid. We could then define a subclass *GraphicComplexNumber* that inherits from *ComplexNumber* and adds a *display* operation.

A natural extension to simple inheritance is *multiple inheritance*, that is, inheritance of a subclass from multiple parent classes. In this case we would view our *GraphicComplexNumber* as, say, a subclass of both *GraphicObject* and *ComplexNumber*. Some languages that support multiple inheritance include Lisp with flavors [26], Mesa with Traits [11], Trellis/Owl [32], and Eiffel [25].

At this point we get into some interesting fine points concerning class inheritance. First, not all languages with class inheritance support multiple inheritance (Smalltalk-80, for ex-

ample). Although multiple inheritance is not frequently required, it can be quite clumsy to make do without it. Second, it is important to be able to override inherited methods. A *display* operation is quite specific to an object, and may have to be re-implemented or altered for a subclass that inherits it.

Next, subclasses may or may not be permitted direct access to inherited instance variables. Should a subclass, as a client of the parent class it inherits from, be allowed to see what is normally hidden from regular clients of its parent? When a subclass adds a method that accesses inherited instance variables, it effectively violates encapsulation of that parent. Consider our *GraphicComplexNumber*. If its *display* operation makes use of inherited instance variables, then we are no longer free to alter the internal representation of the parent *ComplexNumber*. On the other hand, if *display* only makes use of inherited operations, such as *xvalue* and *yvalue* (which may in fact be computed from polar coordinates), then we achieve greater independence between subclass and parent.

A fourth point is the issue of name clashes in the presence of multiple inheritance. If we inherit two *display* operations, which method do we take? Actually, this is a non-problem that can easily be resolved by indicating precisely what is to occur when a *display* operation is invoked. Either the system provides default rules for selecting one method or for combining inherited methods, or it requires the programmer to explicitly disambiguate. Similarly, inheritance of identically-named instance variables from multiple parents poses no real problem provided we separately inherit each variable, and we have a means to distinguish them in new methods (for example, by prefixing them with their parent class name).

The difficulties with access to "hidden" inherited properties and with resolving inheritance clashes are most pronounced when we must consider the possibility of changes to the class hierarchy. If we wish to change the definition or implementation of an object class, how will this affect inheriting subclasses? Will the rules for resolving name clashes adversely affect subclasses? Principles for evaluating "good" and "bad" inheritance mechanisms are discussed in [35]. Supporting modifications to class definitions is especially problematic when the existing instances of the modified classes and subclasses must be preserved. This problem is known as *schema evolution* in object-oriented databases, by analogy with schema evolution in relational and other database systems. An approach to schema evolution used in the Orion object-oriented database is described in [7].

Inheritance has a slightly different flavour in the field of knowledge representation. Object classes may then represent knowledge or beliefs rather than software packages. (See also [31] in this book.) An instance of a subclass, then, has all the properties of its parents, and possibly more. A subclass is viewed as a *specialization* of its parents. For example, everything that we know to be true about mammals also holds for humans, but not vice versa. Note that this means that every instance of a subclass is also effectively a member of its parent classes. This points out how specialization is distinct from *aggregation*. It is *not* valid to define a class *Car* that inherits from *Body*, *Frame*, *Wheels* etc., since a car is not a wheel. A *Graph-*

icComplexNumber, however, is at once both a *GraphicObject* and a *ComplexNumber*. Inheritance in this case serves not only as a reusability mechanism, but also as a conceptual structuring mechanism.

An interesting variation on class inheritance is what we call *partial inheritance*. In this case we inherit some properties and suppress others. For example, we may define a *Queue* to inherit from a *List* by inheriting instance variables and a *length* operation. We suppress, however, the *insert* and *delete* operations, replacing them with, say, *getfirst* and *putlast*. In this case neither *Queue* nor *List* are subclasses of one another, but they are undeniably related. Partial inheritance is therefore arguably convenient for code sharing, but it can create a mess of a class hierarchy. This mechanism is provided by both C++ [36] and CommonObjects [34].

Class inheritance is essentially a *static* form of inheritance: new classes inherit properties when they are defined rather than at run-time. Once a class has been defined, the properties of its instances (instance variables and methods) are determined for all time. Note that if we permit the re-definition of object classes at run-time (i.e., schema evolution), then instances and subclass instances will effectively inherit new properties. We do *not* consider this an example of dynamic inheritance, however, because class re-definition is not an operation on objects. By analogy, modifying a database schema is not normally considered a database transaction. We must temporarily step out of the object model in order to make dynamic changes to the inheritance hierarchy.

We will use *dynamic inheritance* to refer mechanisms that permit objects to alter their behaviour in the course of normal interactions between objects. Dynamic inheritance, as opposed to schema evolution, occurs *within* the object model. We can distinguish two fundamentally different forms of dynamic inheritance, which we will call *part inheritance* and *scope inheritance*. The key difference is that the former occurs when an object explicitly changes its behaviour by accepting new parts from other objects, whereas the latter occurs indirectly through changes in the environment.

Basically, part inheritance is nothing more than an exchange of value between objects: an object that modifies an instance variable necessarily changes its behaviour, though in a way that is limited by its object class. But part inheritance is far more interesting if we consider instance variables and methods themselves as values. In such a model, an object may dynamically inherit new instance variables and methods from other objects. An example of this kind of inheritance occurs in a system for distributed problem-solving using "knowledge objects" [39][10]. Evolving active objects can acquire new rules and methods in response to events occurring in their environment.

More common is scope inheritance. In this case an object's behaviour is determined in part by its environment or its acquaintances. When changes in the environment occur, the behaviour of the object changes. A simple example is that of a paragraph in a document that inherits its font, type style, point size and line width from its enclosing environment. If the same paragraph is moved to a footnote or a quotation, new properties will be inherited.

Both forms of dynamic inheritance are possible within systems based on prototypical objects [19]. An object may have instance variables and methods, but it may also *delegate* certain messages to an acquaintance, called a prototypical object. Part inheritance occurs when an object replaces an acquaintance to which it delegates messages. Scope inheritance occurs when a prototype changes its behaviour, implicitly affecting all the objects that delegate to it. Modification of a prototype is analogous to modifying a class in a class-based language, but requires no changes to the inheriting instances.

To illustrate that dynamic inheritance is not limited to prototypical objects, let us consider a new mechanism called *dynamic subclassing*. Suppose we have an instance of a *Complex-Number*, and we want to display it. Unfortunately it does not have a *display* method. What we really want is an instance of *GraphicComplexNumber*. With dynamic subclassing, we would temporarily re-package the original *ComplexNumber* as a *GraphicComplexNumber*, perform the *display* operation, and then discard the shell. Although dynamic subclassing is not supported by any object-oriented language, it can easily be simulated by implementing a *GraphicComplexNumber* as an object containing the identifier of a *ComplexNumber*, and delegating all messages other than *display*. When we enter the scope in which we want to display the object, we create a new *GraphicComplexNumber* initialized to point to the old *ComplexNumber*, and simply release it when we are done. (The point of dynamic subclassing is to be able to provide a standard mechanism for extending the behaviour of objects at run-time in way that is independent of the application that uses it: the new *display* operation can be added to any *ComplexNumber* anywhere.)

Polymorphism and overloading

A polymorphic function is one that can be applied uniformly to a variety of objects. For example, the same notation may be used to add two integers, two floating point numbers, or an integer and a float. Similarly, the addition function for a programmer-defined complex number type may also be able to cope with the addition of complex numbers to integers or floats, provided that the handling of these combinations is defined. In these cases the "same" operation maintains its behaviour transparently for different argument types.

On the other hand, the operation *open* may apply to both data streams and windows. Here we are concerned with two operations that coincidentally share a name, and otherwise have completely different behaviour. This is *ad hoc* polymorphism, or "mere" overloading of operation names [9]. This kind of polymorphism is nevertheless useful, but can lead to unpleasantness if abused. It is up to the programmers to choose meaningful names for operations, and to avoid reusing names that can be misinterpreted.

Class inheritance is closely related to polymorphism. The same operations that apply to instances of a parent class also apply to instances of its subclasses. Of course, it is possible to have support for polymorphism without class inheritance. In Unix, for example, the paradigm of a file (or data stream) is omnipresent: the operations *open*, *read*, *write* and *close* apply polymorphically to any "stream" object. In each case different methods are used to implement these operations. In an object-oriented Unix, every stream object class would inherit methods from a generic stream class, and tailor those specific to the new kind of stream.

Polymorphism enhances software reusability by making it possible to implement generic software that will work not only for a range of existing objects, but also for objects to be added later. A *Sorter* will sort any list of objects that support a comparison operator, just as software written for Unix streams will continue to work if we add a new kind of stream object.

Polymorphism may or may not impose a run-time overhead depending on whether *dynamic binding* is permitted by the programming language. If all objects are statically bound to variables, we can determine the methods to be executed at compile-time. For example, the addition of an integer expression to a floating-point expression is normally detected by the compiler which then generates the appropriate code for that kind of addition. In this case polymorphism is little more than a syntactic convenience. On the other hand, if variables can be dynamically bound to instances of different object classes, some form of *run-time method lookup* must be performed. In Smalltalk, for example, it may be necessary to search through the class hierarchy at run-time to find the method of an inherited operation. The cost of dynamic binding can be much lower, of course, with the result that there will be more work involved when modifying a method inherited by many subclasses. In both Simula and C++, the designer of an object class may decide that dynamic binding is to be permitted, and thus declare certain operations as *virtual functions*. Subclasses can specify implementations of virtual functions, and invocation of these functions on instances will be resolved at run-time depending on the class to which the instance belongs.

Generic classes

Whereas the mechanism of class inheritance achieves software reusability by factoring out common properties of classes in parent classes, generic object classes do so by partially describing a class and parameterizing the unknowns. (For a good discussion of the relationship between inheritance and genericity in a strongly-typed setting, see [25].) These parameters are typically the classes of objects that instances of the generic classes will manipulate. There are basically two categories of generic object: homogeneous "container" objects, like arrays and lists, that operate on any kind of object, and "tool" objects, like sorters and editors, that can only operate on certain object classes.

In the case of tool objects, the parameter must be constrained to indicate the required parent class of the parameter. A generic sorter, for example, could only sort objects with a comparison operator, that is, instances of some subclass of the class *TotallyOrdered*. The sorter object can then exploit polymorphism to apply uniformly to all objects that satisfy the constraint. This idea of constrained parameters works well in a typed object-oriented language, as we shall see shortly.

Even when the parameter is not important for the generic object itself, it can be useful for maintaining homogeneous collections. A *List* object, for example, may be capable of storing any kind of object, but when used by the *Sorter* object, it is im-

portant to guarantee that only *TotallyOrdered* objects are inserted in the list.

Depending on the nature of the parameters, it may or may not be possible to compile generic classes before the parameters are bound. For example, the code for a *List* object could be pre-compiled if implemented using pointers to the elements, but the *Sorter* object might need to statically bind the comparison operator in order to achieve reasonable performance. In the latter case, a generic class is similar to a macro.

2.2 Object Types

An *object type* is superficially the same thing as an object class. The difference is that when we manipulate typed objects, we would like to statically verify that we are doing so in a consistent fashion. With static type-checking we can eliminate the need for objects to protect themselves from unexpected messages.

Languages like Clu [22] and Ada [1] support the definition of *abstract data types* but provide no mechanisms for class inheritance. In other words, there are languages that are legitimately "object-oriented" (i.e., oriented towards programming in terms of objects), yet support a very different style of programming from that encouraged by languages like Smalltalk or Lisp with flavors. More recently several attempts have been made to unify object classes and object types. Languages like C++, Trellis/Owl and Eiffel are both object-oriented in the Smalltalk sense as well as being strongly-typed.

The traditional approach to type-checking in languages with user-defined object types is to insist that the types of expressions supplied to operation invocations and to assignments must correspond exactly to the expected type. In object-oriented languages with polymorphic operations and dynamic binding, we must cope with the fact that some types may be equivalent, or included in other types. In this case, the declared types of variables and of arguments to operations serve as *specifications* for valid bindings and invocations. Informally, one type *conforms* to a second if some subset of its interface is identical to that of the second. We also say that the first is a *subtype* of the second. They are *equivalent* if they conform to one another. What constitutes the interface of an object type depends on the particular type model chosen for a language, but normally includes operation names and the types of the arguments and return values. These issues are discussed in detail in the context of functional programming languages in [9].

We can more clearly interpret the difference between object classes and object types if we view the latter purely as specifications. In the presence of dynamic binding it is (in general) impossible to statically determine the class of a variable, but with the appropriate type rules, we can still perform type-checking. For example, if we consider the expression:

> x <- y + z

then we can statically determine whether this expression is type-correct, without knowing the classes of the instances that x, y and z will be bound to (they may change). First, we examine the declared type of y and see if it supports the operator +. If so, we check if the type of z is valid for an argument. Then, the type information of y will tell us the type of y+z (but not its

class). If this type conforms to the type of x (i.e., if y+z supports *at least* the interface required by x), then the expression is type-correct.

Many variations on this basic scheme are possible. If dynamic binding is not supported then an object type will always uniquely determine an object class. For primitive objects (like integers) one may also insist on information about the representation of instances. At the opposite extreme in an "untyped" object world all objects have the same type, *object*, and will accept any message, though their response may be unpredictable.

Note that class hierarchies are *not* the same as type hierarchies, although they may overlap. Two classes may be equivalent as types, though neither inherits anything from the other.

Type information can be extremely useful for generic object classes. For example, our generic *Sorter* object will only be able to sort *TotallyOrdered* classes. This constraint is in fact a type constraint, since we do not care what class the objects to be sorted belong to, only whether a total order is defined. Our programming language should then verify that whenever a *Sorter* is instantiated, the type parameter must be bound to an object class that satisfies the constraint, e.g.:

> **var** s : Sorter of integer ;

where the class *integer* conforms to the type *TotallyOrdered*.

2.3 Concurrency

There are two ways in which concurrency and communication have traditionally been dealt with in programming languages:

1. Active entities (processes) communicate indirectly through shared passive objects.
2. Active entities communicate directly with one another by message passing.

The first approach is typical of languages like Modula-2 [40], whereas the second is adopted by, for example, Thoth [15] and various Actor languages [2]. (See [5] for an excellent survey of concurrent programming languages and notations.) These same approaches are used in object-oriented programming languages in order to structure concurrent applications, though they result in different object models.

If we adopt the first approach, it is quite natural to structure the shared memory as a collection of passive objects and to view a process as a special kind of active *Process* object. We require that actions on the passive objects be performed according to their declared interface. For this approach to work, we must have some mechanism whereby the active objects may synchronize their accesses to the shared objects. This may be through the use of semaphores or locks as in Smalltalk and Trellis/Owl [27], or through the use of monitors (as in Modula-2) or transactions as in Avance (formerly OPAL) [3]. This approach is necessarily non-homogeneous, that is, the object model contains two fundamentally different kinds of objects: active and passive. Furthermore, it is not possible to directly interact with the active objects, at least not using the same paradigms for interaction that apply to passive objects: two active objects can only communicate through a passive intermediary. Finally, it is not possible to extend this model to a distributed environment without employing some form of hidden message

passing. This suggests that the second approach is in some sense more general.

Considering the second approach, we permit any object to communicate with any other object. Objects become "active" in response to a communication. In effect, threads of control are determined implicitly by message passing, whereas in the first approach each thread of control was localized in an explicit *Process* object. Explicit synchronization is not required since message passing packages both communication and synchronization, but we must make a choice as to what style of message passing to adopt. For example, message passing may be synchronous as in POOL-T [4] or buffered, as between top-level objects in *Hybrid* [28]. Similarly, we may permit uni-directional message passing as in Act-1 [20], or we may insist on a *call/return* protocol as in Hybrid and ConcurrentSmalltalk [41]. We may also find it useful to provide an *express* mode of message passing for interrupting active objects as in ABCL/1 [42].

In either case, it is possible to accommodate the various reusability mechanisms we have discussed as well as an extendible type system. With the message-passing model we interpret strong-typing to mean that a message-passing expression is type-correct if the message being sent is guaranteed to be valid for the recipient. Similarly, we can support an untyped view, if all objects are prepared to handle any message sent to them. Run-time support for concurrent applications can be quite different for these two object models, as we shall see in the following section. For a more detailed look at approaches to concurrency in object-oriented languages, see [38] in this book.

3   Object-Oriented Systems

Our discussion thus far has focussed on object-oriented programming language constructs and mechanisms, without consideration for run-time support for objects, or for tools to aid programmers in constructing object-oriented applications. In this section we shall provide a brief overview of two important kinds of object-oriented system: those that provide run-time support for object-oriented applications, and those that comprise an environment for object-oriented software development.

3.1   Object management

Object management refers to a mixed bag of run-time issues such as object-naming, persistence, concurrency, distribution, version control, security, and so on. The amount of support provided or required depends very much on the intended application domain. At the low end we have single-user, single-thread applications with minimal persistence requirements, and at the high-end we have distributed, concurrent, multi-user applications with support for evolving software. In either case objects reside in a "workspace" which may be local and private, or distributed and shared.

Minimal object management support is provided for C++ objects. Objects may be allocated and freed in virtual memory. Memory addresses serve as object identifiers. There is no support for garbage collection, persistence, concurrency or distribution. The result is a very lean language and object environment that imposes very little run-time overhead for objects,

without preventing the programmer from defining various extensions (for persistence, concurrency, etc.).

Smalltalk and Lisp additionally provide for automatic garbage collection, and implement a trivial form of persistence by permitting users to save the (single-user) object workspace. Since there is no provision for communication between objects in different workspaces, there is no need to worry about maintaining global consistency in a distributed environment. Persistence for distributed object applications must cope with the possibility of local failures: if a message sent between workspaces is lost, or if either the sender or receiver is accidentally destroyed, then we risk a global inconsistency. Means for dealing with these problems are suggested by two traditional fields: operating systems and database systems.

An object-oriented operating system may provide support for persistence, resilience, reliable communication or distributed object-naming at a low level. For example, Chorus [43] and Mach [17] provide kernel support for distributed object-oriented systems. Argus is a programming language with operating system support for persistence, encapsulation and distribution through the concept of *guardians* [21]. LOOM provides a large object-oriented memory for Smalltalk systems [18].

Most of the object management issues we have mentioned are addressed in some way or another by traditional database technology. It is therefore natural to try and see whether this technology can be transferred to problem of managing objects. An *object-oriented database* is therefore a system that provides database-like support (i.e., for persistence, transactions, querying, etc.) for *objects*, that is, encapsulated data and operations. Some examples are the GemStone system from Servio Logic [24][30], Orion from MCC [6], and Iris from Hewlett-Packard Labs [12][14].

Although object-oriented databases are being built and have clearly practical applications, there are several open problems in this area. First of all, there is no agreement as to a standard data model for object-oriented databases. We do not have the equivalent of relational algebra for an object-oriented data model, and we therefore have no standard guidelines for designing object-oriented databases. This is to be expected, since we have no corresponding agreement as to what mechanisms belong in an object-oriented programming language, or what the rules for encapsulation or inheritance should be. For this reason it is also difficult to decide on a standard query language for objects. Should we be permitted to query on attributes (i.e., instance variables), and if so, how does that square with the principle of encapsulation? (With typical applications for which object-oriented databases have been designed, like CAD/CAM, querying on attributes may be precisely what we want to do, but what about other application domains?) Finally, should we consider object-oriented databases as providing a complete picture of executing applications, or are they better seen as repositories for persistent objects? In particular, can we view active objects as executing *within* the database (as they do inside a Smalltalk workspace) or should we adopt a more traditional database view in which running applications (i.e., threads) are explicitly *outside* the database?

3.2 Object-oriented programming environments

Another significant category of object-oriented system is that of tools and environments for application development. Object-oriented programming promises a great deal in terms of easing the cost of building applications; our ability to realize this promise depends largely on whether or not objects are truly reusable. This implies that we not only need powerful mechanisms and paradigms for reusability (such as those provided by object-oriented languages), but we need tools to help us design objects, to select and reuse objects, and to manage an evolving software base.

Object design is a software engineering issue: how should we decompose our application into objects in such a way as to best exploit the object-oriented paradigms available to us? This task can be supported to some extent by applying conceptual modeling techniques from the database area to object design. An example of such an approach is described in [23]. See also [13] in this book.

The next major problem is that of selecting objects from the software base that may be useful for building your application (ideally this would be carried out in tandem with the design task). Here a programmer would normally rely on three things: his personal expertise, software documentation, and *browsing* tools. Smalltalk, Trellis [29] and Cedar [37] are examples of programming environments that provide tools for browsing the available software base. The real difficulty is that human expertise cannot realistically cope with a large, evolving software base. Perhaps one can manage to remember the 300 fundamental Smalltalk object classes, or even a couple of thousand generally useful object classes, but how are we to cope with tens of thousands of object classes available perhaps from different vendors? The object selection problem is similar to that of performing a literature search. We depend heavily on services to classify objects and to maintain cross-references in the face of updates.

Finally, an object-oriented programming environment must cope with evolution of the software base. There are two ways in which it can do so. The first is by providing software management tools that maintain global consistency. When changes are made to the software base, it is important to ensure that these changes are properly distributed. The problem of managing software evolution in object-oriented systems is especially interesting in the face of class inheritance and subtyping [33]. As long as the interface to an object class is not modified, we have considerable freedom in modifying its realization. When the interface is changed, however, we fall into a snake pit of invalidated references between object classes.

The second problem with evolution is essentially Darwinian: how do we encourage "survival of the fittest"? If prototyping and application development are really much easier with a well-designed software base, how do we make sure that the right objects end up there? This suggests that we should take more of a long range view of application development: whenever we can't find the objects we need to solve our problem in the software base, either we need new objects, or we need to modify old ones. What we do not know is how to make sure that the new objects will not only solve our problem, but will also give us a "better" software base.

Summary

We have put forward the proposition that the term *object-oriented* is best interpreted as referring to any approach that exploits encapsulation or "packaging" in the process of designing and building software. With this premise in mind, we have surveyed object-oriented techniques in programming languages to enhance software reusability, to enhance maintainability and robustness through extendible type systems, and to ease the development of concurrent and distributed applications. We have also given a brief overview of the issues in providing run-time support for objects, and in providing programming environments for the development of object-oriented software.

We have not discussed other applications of object-oriented concepts, for example, in the area of user interfaces. (Direct manipulation interfaces provide the user with the illusion that the objects of the application are being "directly" manipulated by a set of polymorphic operators: move, copy, delete, resize, etc.)

Object-oriented languages and systems are a developing technology. There can be no agreement on the set of features and mechanisms that belong in an object-oriented language since the paradigm is far too general to be tied down. (What features belong in a declarative language?) The idea of using objects to model software is a natural one that will inevitably appear and reappear in various forms; we can expect to see new ideas in object-oriented systems for many years to come.

References

[1] American National Standards Institute, Inc., *The Programming Language Ada Reference Manual*, Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[2] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[3] M. Ahlsen, A. Bjornerstedt and C. Hulten, "OPAL: An Object-Based System for Application Development," IEEE Database Engineering, vol. 8, no. 4, pp. 31-40, Dec 1985.

[4] P. America, "POOL-T: A Parallel Object-Oriented Language," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 199-220, The MIT Press, Cambridge, Massachusetts, 1987.

[5] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," ACM Computing Surveys, vol. 15, no. 1, pp. 3-43, March 1983.

[6] J. Banerjee, H. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou and H. Kim, "Data Model Issues for Object-Oriented Applications," ACM TOOIS, vol. 5, no. 1, pp. 3-26, Jan 1987.

[7] J. Banerjee, W. Kim, H-J Kim and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proceedings ACM SIGMOD '87, vol. 16, no. 3, pp. 311-322, Dec 1987.

[8] G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard, *Simula Begin*, Auerbach Press, Philadelphia, 1973.

[9] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, vol. 17, no. 4, pp. 471-522, Dec 1985.

[10] E. Casais, "An Object-Oriented System Implementing KNOs," Proceedings of the Conference on Office Information Systems (COIS), pp. 284-290, Palo Alto, March 1988.

[11] G. Curry, L. Baer, D. Lipkie and B. Lee, "TRAITS: an Approach for Multiple Inheritance Subclassing," Proceedings ACM SIGOA, SIGOA Newsletter, vol. 3, no. 12, Philadelphia, June 1982.

[12] N. Derrett, W. Kent and P. Lyngbaek, "Some Aspects of Operations in an Object-Oriented Database," IEEE Database Engineering, vol. 8, no. 4, pp. 66-74, Dec 1985.

[13] J. Diederich and J. Milton, "Objects, Messages and Rules for Database Design," in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988.

[14] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan and M.C. Shan, "Iris: An Object-Oriented Database Management System," ACM TOOIS, vol. 5, no. 1, pp. 48-69, Jan 1987.

[15] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," Software – Practice and Experience, vol. 11, pp. 435-466, 1981.

[16] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[17] M.B. Jones and R.F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 67-77, Nov 1986.

[18] G. Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading MA, 1983.

[19] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 214-223, Nov 1986.

[20] H. Lieberman, "Concurrent Object-Oriented Programming in Act 1," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 9-36, The MIT Press, Cambridge, Massachusetts, 1987.

[21] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," ACM TOPLAS, vol. 5, no. 3, pp. 381-404, July 1983.

[22] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, The MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1986.

[23] M.E.S. Loomis, A.V. Shah and J.E. Rumbaugh, "An Object Modeling Technique for Conceptual Design," Proceedings of the European Conference on Object-oriented Programming, pp. 325-335, Paris, France, June 15-17, 1987.

[24] D. Maier, J. Stein, A. Otis and A. Purdy, "Development of an Object-Oriented DBMS," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 472-482, Nov 1986.

[25] B. Meyer, "Genericity versus Inheritance," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 391-405, Nov 1986.

[26] D.A. Moon, "Object-Oriented Programming with Flavors," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 1-8, Nov 1986.

[27] J.E.B. Moss and W.H. Kohler, "Concurrency Features for the Trellis/Owl Language," Proceedings of the European Conference on Object-oriented Programming, pp. 223-232, Paris, France, June 15-17, 1987.

[28] O.M. Nierstrasz, "Active Objects in Hybrid," ACM SIGPLAN Notices Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243-253, Dec 1987.

[29] P.D. O'Brien, D.C. Halbert and M.F. Kilian, "The Trellis Programming Environment," ACM SIGPLAN Notices Proceedings OOPSLA '87, vol. 22, no. 12, pp. 91-102, Dec 1987.

[30] A. Purdy, B. Schuchardt and D. Maier, "Integrating an Object-Server with Other Worlds," ACM TOOIS, vol. 5, no. 1, pp. 27-47, Jan 1987.

[31] D. Russinoff, "Proteus: a Frame-based Non-monotonic Inference System," in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988.

[32] C. Schaffert, T. Cooper, B. Bullis, M. Killian and C. Wilpolt, "An Introduction to Trellis/Owl," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 9-16, Nov 1986.

[33] A.H. Skarra and S.B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 483-495, Nov 1986.

[34] A. Snyder, "CommonObjects: An Overview," ACM SIGPLAN Notices, vol. 21, no. 10, pp. 19-28, Oct 1986.

[35] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 38-45, Nov 1986.

[36] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[37] D. Swinehart, P. Zwellweger and R. Beach, "A Structural View of the Cedar Programming Environment," ACM TOPLAS, vol. 8, no. 4, pp. 419-490, Oct 1986.

[38] C. Tomlinson and M. Scheevel, "Concurrent Object-Oriented Programming Languages," in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988.

[39] D.C. Tsichritzis, E. Fiume, S. Gibbs and O.M. Nierstrasz, "KNOs: KNowledge Acquisition, Dissemination and Manipulation Objects," ACM TOOIS, vol. 5, no. 1, pp. 96-112, Jan 1987.

[40] N. Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin, 1983.

[41] Y. Yokote and Mario Tokoro, "The Design and Implementation of ConcurrentSmalltalk," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 331-340, Nov 1986.

[42] A. Yonezawa, J-P Briot and E. Shibayama, "Object-Oriented Concurrent Programming in ABCL/1," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 258-268, Nov 1986.

[43] H. Zimmermann, M. Guillemont, G. Morisset and J. Banino, "Chorus: A Communication and Processing Architecture for Distributed Systems," Research report no. 328, INRIA, Rocquencourt, Sept 1984.