Some Notes on Programming Paradigms

The control structures available in a given high-level language are very much dictated by the programming paradigm followed by that language. Of course, regardless of the language or its approach to programming, the fundamental control construct is the application of operations to operands. However, the way in which an individual can combine operations to produce a program may differ widely from one group of languages to another. The approach to programming as dictated by the programming language is often what we mean when we refer to a programming paradigm.

A *programming paradigm*, or world view, is an approach to solving programming problems. In practice, this approach may be dictated more by the particular programming language used than by the orientation of the programmer except, perhaps, in the programmer's choice of a language. In a sense, given the history of the major programming paradigms, they may be considered abstractions based on the features of the particular programming languages in which they arose. If this sounds rather haphazard, it was, and is. To some extent, the "Tower of Babel" of programming languages can be characterized as many different languages doing the same thing. Often, however, we find ourselves bumping up against a small number of radically different languages that do things in quite different ways. Still, if there is at least one unifying concept in all paradigms, it is that *operations* are ultimately applied to *operands.* The differences lie in the ways in which the operations and operands may be organized and described.

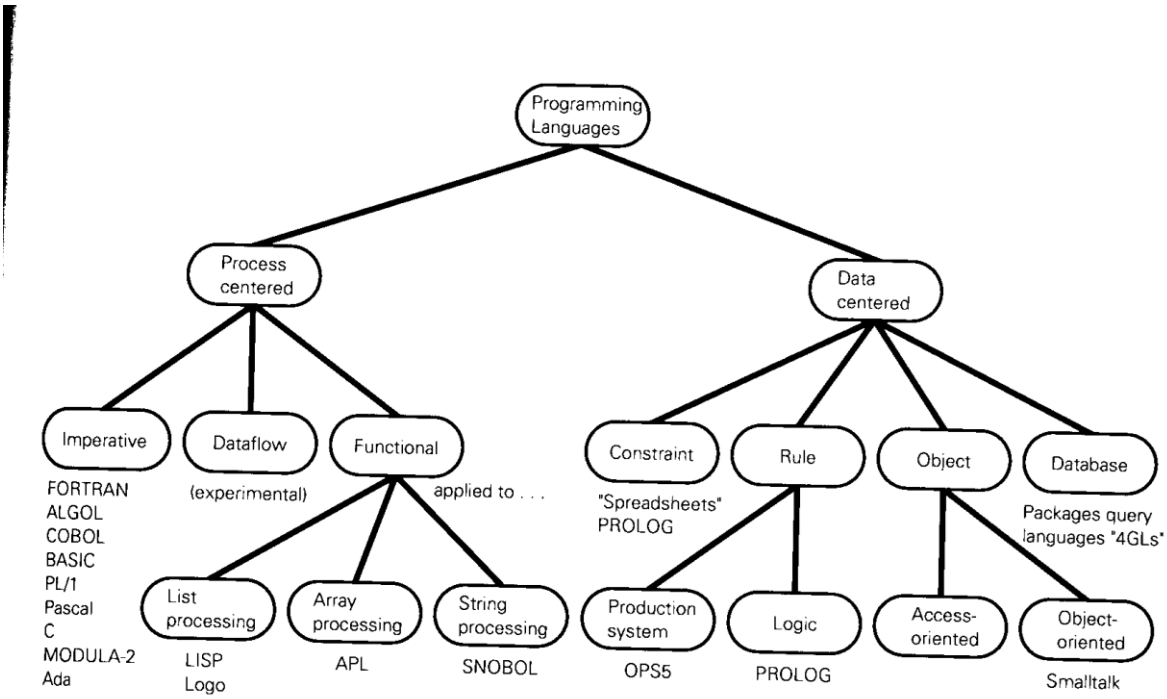

Figure 6.23 Programming paradigms

A classification of programming paradigms is presented in the figure above. Languages listed are associated with a particular programming paradigm as the major organizing principle. Some of the languages listed are not widely known and, in fact, may be somewhat esoteric. Others are experimental. Each of these programming orientations represents not just a different way of coding a program, but often a totally different way of approaching a problem conceptually and organizing the solution before any coding takes place. The orientation of a programming language may be more procedural (concerned with *how)* or more declarative (concerned with *what);* it may be more process-centered or more data-centered. And, among the data-centered languages, the programming world view will differ drastically depending upon the form of the data in question.

You will note that most of the traditional, widely known languages and most of the languages discussed are classified under the imperative paradigm, all in one branch of the classification tree. In a sense, then, this tree is deceptive. Although many different paradigms exist or have been studied, most languages in current use are still imperative languages in the **FORTRAN** line of descent. That is why the imperative branch of the classification tree is relatively heavy with representative languages while the rest of the tree is rather sparsely populated. In some other paradigms, the distinction among these various types of structures is not as clear. For example, in the object-oriented programming paradigm, data abstraction, program abstraction, and control abstraction are all contained within the same structure.

From a theoretical point of view, the choice of an appropriate programming paradigm for a particular application is independent of the language used to implemented it and, in fact, may or may not precede the selection of an appropriate programming language. Of course, in reality, the programmer wi11 be more than a little limited by the organizing principle of the particular programming language used. However, it is sometimes possible to design a solution using one world view and then mold a programming language with a different organizing principle to that world view. The language should be highly extensible and, naturally, the fit would never be complete. In addition, there are some languages available and others under development that support more than one programming paradigm. Some of these, for example, C++ and **LOOPS** will be discussed at the end of this section. In fact, much of the research going on today in the area of programming paradigms involves efforts to make several of these approaches available in a single programming language or environment.

Imperative Programming

The imperative programming paradigm is a statement-oriented approach reflected in all the languages that have their roots in FORTRAN, COBOL, and/or ALGOL. Clearly, then, many languages take this approach. These languages are also known as *algorithmic languages,* because they facilitate the construction of programs as *algorithms,* or as sequences of instructions for computing.

The imperative world view supports sequential *imperatives* that effect changes to the contents of internal storage locations by assignment. Here, the word imperative (a noun) means *command.* It may also be used as an adjective, meaning *in the manner of a command* (e.g., an *imperative statement).* The imperative approach is perhaps seen most clearly in COBOL statements, each of which begins with an action verb (that is, a command). For example:

READ, WRITE, PERFORM, COMPUTE, DIVIDE.

The most important elements of imperative languages are *variables*, the *assignment operator,* and *procedures.* The imperative programming paradigm, the most commonly used programming world view, draws its major characteristics from the most prevalent machine architecture, the von Neumann architecture.

The von Neumann machine. The statements of imperative languages are closely related to the built-in instruction set of the underlying machine, Even high-level imperative statements translate in a fairly simple, straightforward manner into low-level machine code. Imperative statements reflect an assumption of and dependence on the von Neumann type of computer architecture-namely, a central processor that executes one primitive instruction at a time tied very closely to an internal memory unit containing a large number of unique addressable storage locations that may be retrieved one at a time, and a one-word bus for transporting data between the two components. Thus, this architecture and consequently these languages are fairly low level and highly variable dependent.

Imperative programming languages are characterized by their variables, the assignment operation, and the iteration control construct. Other paradigms are typically characterized by the absence of these features.

Variables. Programmer-defined variables correspond to, and are used to access, individual storage locations containing values. Changes are made by explicit assignment of values to variables. The values of the collection of variables represent the state of computation: as assignments are made to these variables over the course of program execution, so the state of the program changes over time. Thus, an imperative program is *history sensitive*, meaning that the result of the program depends a great deal on the state of computation when execution starts up. Another criticism of this programming approach is that too much programmer effort is spent keeping track of individual variables. This includes naming conventions and the detail work of managing every change by assignment.

Assignment. The most important statement in an imperative language is the assignment statement, since it is the only one that does any real, productive work. The assignment statement, which is often of the form <var> := <expression>, retrieves a value using instructions on the right side of the assignment operator and stores that value in the memory location designated by the variable named on the left side of the assignment operator. Thus, what the program actually accomplishes, it does by means of the assignment statement. With the exception of input/output statements-which are, after all, only there because it is necessary for the computer to communicate with the outside world-all the other types of imperative statements are involved in control (management) not assignment (production).

Iteration. An imperative program will typically execute a statement or sequence of statements repeatedly. This repetition may be controlled by a counter or index variable. The iteration control construct itself contains a branching instruction (a **goto),** which is also a characteristic of the imperative programming paradigm.

Procedures. Procedures represent an enhancement to imperative programming language statements. In the procedural programming paradigm, statements may be grouped together into a module that serves as an abstraction device. Thus, control abstraction is facilitated by modules forming the sequence, selection, and iteration constructs. Program abstraction is built by modular decomposition and stepwise refinement. Data abstraction may be provided by specialized modules encapsulating data and procedure description. When procedures and modules can be nested, static scoping rules are important. These languages are often called block structured languages.

Examples of imperative languages are COBOL, Pascal, MODULA-2, and Ada. BASIC is an imperative language that is not typically procedural.

Imperative programming languages are still the most efficient ones on von Neumann-type machines, and this type of computer architecture is still the most prevalent today. Indeed, the other programming paradigms that we will discuss are typically compiled to low-level imperative programs in the interest of efficient execution.

Functional Programming

The *functional programming paradigm* involves the mathematical specification of the solution to a programming problem. In mathematics, the major organizing principle is the function and, in programming, this mathematical entity is translated as a subprogram returning a single value. A complete functional program defines an *expression* which, when used with specific operands, solves a particular problem. The use of functions to construct programs can often result in an elegant, concise algorithm.

In a functional programming language, a program is a function that is built up from smaller functions. The functional programming paradigm allows a programmer to solve a programming problem by means of a hierarchy of functional abstractions, in other words, with top-down functional decomposition by stepwise refinement. Functions are combined, producing more powerful functions. Functional programming languages, also called *applicative languages*, achieve their major effect by the application of functions to arguments, which may themselves be functions. This is easily distinguished from imperative programs, in which state transitions are accomplished by means of assignment statements.

Functional programming languages are characterized by powerful data structuring capabilities. These languages are not constrained by variables tied to storage locations. A large, and often complicated, data structure is treated simply as if it were a single value. Data in these languages are often organized into such structural forms as lists, strings, and arrays.

There are three major control constructs used to build up functional programs:

- *Functional composition* is used to define a function in terms of other functions that are the operands.
- The *conditional expression* is used for selecting a function based on the value of another (possibly Boolean) function.
- *Recursion* is used in much the same manner as it is in mathematics, replacing the iteration construct of imperative languages.

Both functional and imperative languages allow specification and application of functions. The difference is that, in an imperative language, a function is merely another type of subprogram and, as such, allows assignment of values to parameters and to global variables. Along with the problems of side effects, then, imperative functions lack referential transparency.

Referential transparency. An important distinguishing feature of functional programming, referential transparency may be described as follows: The value of a function is a unique value that is determined solely by the values of the arguments to the function. This corresponds to the mathematical definition of a function. Thus, two applications of the same function using the same arguments must produce exactly the same result. In other words, the solution is unique. This may be characterized as a many-to-one transformation. Imperative languages, with their high degree of dependence on sequential assignments of values to variables, are history sensitive and not referentially transparent. Relational and rule-based languages, on the other hand

(PROLOG, for example), do not have referential transparency because they are characterized by the many-to-many transformation.

Lazy evaluation. In the functional programming paradigm, an expression or function is evaluated only when some other expression needs its output. Since a functional program is implemented as a series of functions applied to arguments, this "lazy," or delayed, evaluation of expressions is translated into a parameter-passing mechanism known as *call by need*, very much like ALGOL's call by name, in which an argument is evaluated only when its value is needed and not immediately at the point of call. This provides the advantages of both call by name and call by value-since when an argument's value is not required, it is not evaluated; and if it is required it is evaluated once only without the possibility of unwanted changes.

The oldest functional programming language, in fact, the archetype for functional languages, is **LISP.** The **mathematical foundation of LISP** is in lambda calculus, and the unifying data structure (and program structure) is the list. LISP is not a "pure" functional programming language since, in the interests of efficiency, it was augmented early on with some features of imperative languages, for example, variables and the assignment operation. Over the years, there have been a number of derivatives of LISP, for example, SCHEME, CommonLISP, and Flavors. APL, although it does have an assignment operation, may be considered a functional programming language due to its heavy reliance on expressions combining powerful functions.

In order to implement a functional approach in an imperative language, we would need to: Avoid the use of global variables and side effects and use call by value exclusively as a parameter-passing mechanism.

Functional languages provide many benefits. Since variables are not a feature of the functional paradigm, there is greater control over the flow of data, there is no threat from "side effects," and programs are referentially transparent. In addition, it may be possible to mathematically prove program correctness. However, functional languages are not based on the von Neumann architecture and are inefficient on most of today's computing machinery. Specialized computers (such as LISP machines) may be required.

Rule-oriented Programming

In the rule-oriented paradigms, one may: specify facts and rules about objects and their relationships; query the system regarding these objects and relationships; combine facts to express them as a single rule; easily integrate new facts and rules into a program, This approach is important in artificial intelligence in which it is used to build knowledge bases. A knowledge base is simply a database composed of facts and rules.

A rule is a condition-action statement, also called a *production*, that may be of the form: if <condition> then <action>. Rule-oriented programming has two personas: Pro*duction-system programming* views knowledge as a production system, a rule base composed of independent rules. A rule-based system must be able to deal with uncertainty and explain itself as well. *Logic*

programming is grounded in the mathematical basis of predicate calculus. Its rules are in a more restrictive form. This approach does not attempt to deal with uncertainty (too informal) and does not include an explainer facility as part of its definition. Logic programming seeks to satisfy a goal along with its subgoals. Production-system programming, with its less formal underpinning, is able to look for a second-best or "good enough" solution. Logic programming is referentially transparent; production-system programming is referentially opaque. Both approaches share a number of features.

A distinguishing feature of the rule-oriented paradigms is the explicit and complete separation of data and control elements in the program. The knowledge base is a separate unit operated on by the control component. This control mechanism differs in the logic and rule-based paradigms, The data is not only independent of the control element, but there is a large degree of independence within the database as well. Rules (or, at least, groups of rules) are independent of each, and can be added to the knowledge base incrementally or modified as new knowledge becomes available, without modifying the control elements of the program. Clearly, this type of system would be a never-ending nightmare to implement and maintain in conventional imperative languages.

Both logic programming and production-system programming call for a decision regarding a reasoning mechanism and a search strategy. (Of course, this "decision" may be part of the language design and, as such, out of the hands of the programmer.) The reasoning mechanism can move either forward (i.e., inductive) or backward (i.e., deductive), referring to the order in which the rules are examined. The search strategy may be either depth-first or breadth-first.

The *backward-chaining* reasoning mechanism begins with a solution or goal and tries to prove it correct. Backward-chaining systems are also called *top-down* or *goal-directed systems.* The system selects one goal at a time and sets about trying to prove it.

Each goal may consist of a number of subgoals, each of which must be proven in order for the hypothesis to be correct. *Forward chaining* begins with the facts and rules at hand and uses that as a starting point to search for a solution. Forward-chaining systems are also described as *bottom-up, data-driven,* or *event-driven,* In this case, the system examines one data item at a time, making whatever inferences it can from it. After analyzing some facts in this manner, the system may query the user for other facts it needs in order to reach a conclusion. Of the two reasoning mechanisms, backward chaining is more efficient and is more commonly applied, although the two may sometimes be combined. Backward chaining is appropriate when the solutions are known, and there are not too many of them. Forward chaining works better when the number of starting points is small relative to the number of possible solutions.

A *search strategy is* employed to search the knowledge base. The *depth-first* search strategy is the most commonly used. This is a search along a single search path resulting in ever-increasing detail. The search continues to a possible goal before attempting to explore another path leading to another goal. The *breadth-first* strategy searches across paths, examining all rules on a given level, eliminating as many as possible, before going to the next, more

detailed, level. It does not commit the system to completely examine only a single goal before looking at any other goal.

The rule-oriented programming paradigms represent a "many-to-many" transformation, i.e., there is a *set* of solutions for a particular application. For example, if Jennifer, Donna, and Mary have been defined as sisters, the query Sister<Donna> might produce as a response either Mary or Jennifer, and either response would be correct.

Production-system programming. A production is a condition-action rule, A set of such rules, called a *production system, is* the oldest way of representing human knowledge to the computer. Production systems, first formulated by Allen Newell and Herbert Simon at Carnegie-Mellon University as models of human cognition, are sets of independent knowledge modules containing these condition-action rules, which represent knowledge. Today, they are commonly called *knowledge bases.* Since the rules (or, at least, groups of rules) are independent, it is possible for such systems to "learn" incrementally. This provides the basis for the production-system programming paradigm, which is perhaps more commonly known as the *rule-based programming paradigm.*

The elements of a rule-based system are:

- the production system, equivalent to a program structure;
- working memory, containing values that are the system's state of current knowledge and are thus somewhat like global data; and
- the inference engine, which is the interpreter and provides the control structures for the system.

The *control mechanism* of a rule-based system is its key distinguishing characteristic. The inference engine works on a match-select-execute cycle.

Match: The inference engine finds all the rules that meet a match to the goal-that is, the left-hand side evaluates to true. All rules meeting this condition are maintained in a conflict set.

Select: One rule is selected from the conflict set according to a particular conflict resolution mechanism.

Execution: The interpreter processes the clauses of the right-hand side of the selected rule to modify working memory.

The inference engine cycles over these processes until the conflict set is empty.

Explanation *module.* An important part of a rule-based system is the "explainer" facility, which keeps track of which rules have been satisfied, and can output to the user the specific chain of reasoning that led to the particular conclusion reached and action taken. In this way it is said to generate explanations of program behavior.

The first production -system programming language, and still the definitive language for this programming paradigm, is OPS (named, somewhat tongue in cheek, as the Official Production System), designed at Carnegie Mellon in 1970. The current version is OPS5.

Logic programming. The rules of the logic programming paradigm are *Horn clauses.* These are more restrictive than the productions of the rule-based paradigm: the left-hand side of the rule may contain at most a single clause. This reduces the search space, improving the efficiency of the system.

In the logic programming paradigm, the programmer provides a description of a problem in the form of predicate logic statements. The system then is interpreted by a mechanism based on resolution logic. Logic programming languages may be used to set up databases of rules called *knowledge bases* and, therefore, they fall under the rule-oriented paradigm. In fact, the logic programming approach may be considered to be a more formal subset of the rule-based paradigm.

A logic program evaluates theorems to see if they are true. Constraints of a problem are specified rather than the algorithm for solution. A logic program is composed of:

- a knowledge base, the program structure;
- a goal to be proved, often, in the form of input data; and
- an inference mechanism, or control structure.

The inference mechanism is built into the language or system used.

Logic programming languages suffer from a "closed world" view. In other words, any fact not in the knowledge base is automatically not true. This means that if a relation is disproved it may not in reality be false; it may simply be absent from the program.

The prototypical logic programming language is **PROLOG;** others are, for the most part, extensions and enhancements of this language. The particular approach taken by **PROLOG** is also a very specific type of constraint-orientation (see below) since it satisfies logical constraints. Since the rules of **PROLOG** are relations, it is also sometimes classified as a relational language (along with query languages like SQL, which are used to access data from relational databases).

Object-oriented Programming

Object-oriented programming makes good on the promise of structured programming. It implements in a very practical way the principles of program decomposition, data abstraction, and information hiding. It ties together and provides a framework for abstraction and structure at all levels: data, program, and control. As such, it is the natural culmination of everything we have studied in Part I of this book.

Object-oriented programming picks up where structured programming methodology leaves off. Dijkstra's concept of structured programming, Wirth's stepwise refinement, and Parnas's information hiding all contributed to a software development milieu that promised to become

increasingly systematic and scientific. Object-oriented programming, to a great extent, fulfills that promise. It takes the concepts of data abstraction, modular decomposition, and information hiding and refines them into a cohesive world view in which the organizing principle is the data object. In this world view, data objects are active entities. Instead of passing data to procedures, the user asks data objects to perform operations on themselves. A complex problem is viewed as a network of objects that communicate with each other.

Large programming projects have problems of complexity that structured programming alone cannot alleviate. Structured programming methodology appears to break down when applications exceed about 100,000 lines of code. At that point, the assumption that the programmer will be able to "get it right the first time" is no longer valid. Bugs abound, programmer productivity plummets, and the term "software maintenance" becomes about as popular as "plague." Furthermore, as applications increase in size and complexity, even getting it right the first time is no longer good enough since these applications are frequently evolutionary in nature. The object-oriented programming paradigm promotes and facilitates software evolution.

The object-oriented approach provides a natural framework for easily modularizing programs and data, eliminating much of the headache of structured-design techniques. Programming can be carried out in an incremental fashion. In a very real sense, an object-oriented program is a simulation of the real world. It is therefore a relatively natural way to program. In object-oriented programming, we would: identify objects and their attributes; identify operations on the objects; establish the interfaces between objects.

An *object* is a bundle of data and related functions. In an object-oriented programming language, every data element is an object. This includes literals and all sorts of data structures. Objects are defined by a hierarchy of *classes* capable of *inheritance.* Objects communicate with each other by passing *messages.* The object receiving the message activates a *method* corresponding to that message and may, in turn, send messages to other objects.

A *class* is an independent program module representing a class of data object, much in the same way that Ada's package implements an abstract data type. However, the class concept, which originated with **SIMULA,** goes beyond data abstraction. A class encapsulates the data declarations, called *instance variables,* and the function specifications, called *methods,* necessary for the definition of the data object. An *object* is an instance of a class. There may be many instances of the same class coexisting at any moment in time. A class, then, may be viewed as a template for generating objects. An object's data is private to it, and the only way to access that data is to ask the object to invoke one of its methods

Methods are operations defined within the class. A method is invoked when an object receives an appropriate *message* (which may contain arguments and typically returns a value to the sender of the message). Objects communicate with each other by means of *message passing.* A message is a request to the object to carry out an operation and is roughly (very roughly) equivalent to a subprogram call. The operation must be contained in the group of procedures in the definition of the object. These methods are given in the class definition and specify how objects of the class will respond to any particular message. Thus a method is similar to a function

specification and a message is equivalent to a function call. Every object is associated with a *class protocol* as specified in its class definition. This is the external interface and contains the messages to which the object can respond, and for which the object has matching methods.

The state of an individual object is maintained and represented by its *instance variables,* which are not deallocated when execution of a method terminates. This differentiates method invocation from a standard subprogram call. Instance variables are roughly equivalent to the **STATIC** variables of PL/l. Only the methods, akin to local procedures, can directly manipulate this data. This is the implementation of the principle of information hiding that is important in all modular programming in reducing the interdependencies between modules. Objects do have the capability to declare purely local variables, called *temporary variables,* that only exist during execution of the method subprogram. *Class variables* contain common data values, which may be shared by all objects of the class.

If we were going to draw a very rough comparison between the features of object-oriented programming and imperative programming, we would say that an object is somewhat equivalent to a record, a class to a record type or an abstract data type, an instance variable to a field on the record, and a message to a subprogram call. This comparison is not terribly accurate but it is close enough to be a good starting point for a programmer grounded in the imperative paradigm to cross over to object-oriented programming. For example, defining a class in ObjectPascal is very much like a Pascal-type definition:

```
type
      ClassName = object(SuperclassName)
      <instance var declarations>
      <method header declarations>
end;
```

One key characteristic that separates classes from abstract data types is *inheritance.* Classes and their objects are organized into a hierarchy. The variables and methods of a class are inherited by its subclasses and do not have to be recoded in the subclass definitions. The subclasses then need code only for the structure and behavior that is unique. Thus, a class may be seen as a specialization of a superclass and may either override or provide additional definition to its superclass. This makes for reusable code, since common structures and methods do not need to be recoded from scratch for each new class of objects, only that which is different from the existing class. Inheritance relationships enable the customizing of objects to a user's specific needs without costly and time-consuming coding of custom software.

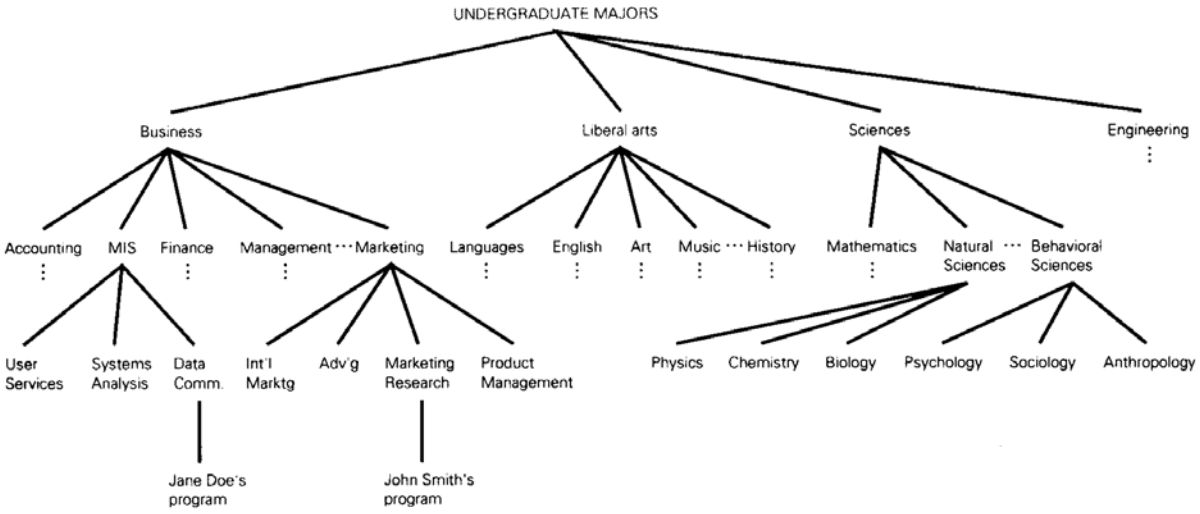

Figure 6.24 Classification of objects

The figure gives an example of a class hierarchy structure for a registration and record-keeping system in a particular (hypothetical) university. Each class contains information, such as required and elective courses, specific to its own majors. The class MIS contains information important to all MIS majors, such as required MIS core courses. Jane Doe is a data communication major. Jane Doe's program is an object, a class instance. Any student's program is a specific instance of a major. Each class inherits all the characteristics from the one above it and then adds its own particulars. MIS majors must take the required business core curriculum, and there may be certain core requirements across all undergraduate majors.

The concepts of class and inheritance implement two special relationships called is-a and a-kind-of. The *is-a* relationship relates an object to its defining class and the *a-kind-of* relationship relates a class to a superclass. For example, from Fig. 6.24, John Smith's program *is-a* MarketingResearch major's program; MarketingResearch is *a-kindof* Marketing major, which is *a-kind-of* Business major.

Dynamic binding, also called *late binding,* means that the association of a data element with a particular property occurs during run time, rather than at compile time (that would be *early binding).* Thus, for example, stack definition may be used for integers, characters, or records. This is true data abstraction. With late binding, the same name can be used for similar operations (methods) on different kinds of objects. This is called *polymorphism. A* polymorphic operation may be applied to a variety of different types of data objects. The message looks the same but it invokes a different method depending on the type of object receiving the message. Traditional languages call this *operator overloading.* This tends to eliminate a lot of selection constructs from the control structure of the program.

A *garbage collection* facility, which originated fairly early on with LISP, automatically reclaims storage that is no longer needed by the program. This eliminates the need for programmer control over memory management and may represent a large savings in a large, complex application. However, garbage collection facilities are often time consuming and space inefficient and thus serve to increase the overhead in a large software system. One method of garbage collection, which is used in Smalltalk-80, is *reference counting.* In the reference -counting method of garbage collection, the system automatically keeps track of the number of pointers to each object in the system. When the number of pointers drops down to zero, the object is deallocated, and the space may be used for another object. Another garbage collection method, called *mark and sweep,* works in this way: When the system runs out of memory, a two-pass procedure is triggered. The "mark" pass traces and marks all objects that are not garbage. The "sweep" pass reclaims storage space from everything else.

The archetypal object-oriented programming language is Smalltalk, designed at the Xerox Palo Alto Research Center (PARC) in the early 1970s. Object-oriented methodology has also been incorporated into and combined with a range of different languages, producing ObjectPascal, ObjectiveC, C++, LOOPS.

Some of the benefits of object-oriented programming are probably obvious. There is a high degree of modularity in object-oriented programs. Since objects are relatively self-contained and encapsulate data and related code segments together in a single package, the object-oriented world view is better equipped to handle complexity in a natural manner and can be better for the design of very large programs. Also, objects are less susceptible to the kind of programming error that occurs when a data structure or its component is changed. When code is changed, it only has to be changed in one place in the program. The inheritance characteristic means that object-oriented programs will have a great deal of reusable code. This too reduces the chance of programming error. And then, there is the naturalness and readability that goes along with polymorphism. Finally, the object-oriented approach is well suited for such control mechanisms as parallel processing and coroutines.

On the negative side, object-oriented programming is often inefficient due to dynamic binding and garbage collection. In addition, message passing requires more processing time than does calling subprograms. In general, with this approach, the emphasis will be on greater programmer productivity at the (possible) expense of machine efficiency.

Other Paradigms

The programming paradigms discussed so far are the most widely used as organizing principles for coding programs. They are represented by established languages that are not experimental and have an experienced user base. There are also a number of programming paradigms that are not as widely known or used. Some of these are experimental or used primarily in conjunction with another paradigm.

Access-oriented programming. Access-oriented programming is, in some respects, a refinement of object-oriented programming. It has historical roots in the language SIMULA and "attached procedures."

In this approach to programming, accessing data, either storing or retrieving, from variable locations in storage can cause procedures or methods to be invoked. As compared to the object-oriented paradigm, in which an object may receive a message to change its own state, in the access-oriented paradigm, whenever the state of an annotated object changes (on access), a message is sent to invoke a procedure. The access-oriented paradigm is often considered the "dual" of object-oriented programming. Access-oriented programming involves the specification of "side effects," sometimes called "demons," which are attached to the manipulation of variables. A "demon" is a program triggered by specific conditions. Access-oriented programming provides a mechanism for monitoring these conditions.

This approach is based on a value called an *annotated value,* which associates annotations with data objects. A kind of annotated value is an *active value,* which monitors a data object and, on its access, triggers the invocation of the appropriate method (demon). There is no need for explicit function calls. Active values are objects and have their own variables for saving state. An active value may be considered to be an abstraction that transforms a variable access to a method invocation. Another kind of annotated value is a *property annotation,* which maintains property lists for data values. The property list can change dynamically over the course of execution and can be used, for example, to store a history of value changes.

One application of access-oriented programming is run time type checking. It is also appropriate for game playing and simulation applications and to implement the windows and iconic menus often found in an interactive user interface.

The access-oriented paradigm is not used on its own in language design, but it may be combined with other world views.

Constraint-oriented programming. In the constraint-oriented programming paradigm, the programmer specifies a set of relations among a set of data objects. The constraint-satisfaction system, then, attempts to find a solution that satisfies the relations. For example, if the system specified is simply A + B = 27, and a value for A is also specified, the system will find the value of *B.* If a value of *B* is given, the system will find the value of A. The one relation alone is sufficient to do both types of analyses and no additional coding is necessary. Thus stating that A = 3 is the same as 3 = A.

A constraint system is a set of constraints on data objects. These constraints may be specified in any sequence. The system is satisfied as long as all of the constraints in the set are satisfied. Constraints may be identified as *basic* or *unconditional* constraints (these are equations), *complex* constraints (which may be reduced to basic constraints), *conditional* constraints, *constraint generators,* and *nested* constraints.

In this type of system, a data object may be defined implicitly based on its relationship with another specified object.

PROLOG is a (very specific) type of constraint-oriented language: It satisfies logical constraints. Constraint-oriented programming is the paradigm of spreadsheet applications. This approach is also used in the areas of graphics, engineering, and knowledge representation.

Dataflow programming. The dataflow programming paradigm specifies the flow of data through a network of operations. In this respect, it is similar to the dataflow analysis at the larger system analysis and design level. Strictly speaking, dataflow programs are intended to run on dataflow machines. True dataflow machines, however, are a rarity and still experimental.

Dataflow analysis determines dependencies among data and the order of execution is in turn determined by these dependencies. Operations are selected for execution once their operands have been computed. For example in the following instruction set:

```
1. X = Y + Z
2. Y = 3 + 5 / 8
3. Z = 9 - 8
```

statement 1 must wait for results from statements 2 and 3, both of which may be executed independently and in parallel. Thus, it is the flow of data between operations, rather than statement number, that actually constitutes the "control structure" of the program.

Once the flow of data between statements is examined, operands necessary for several (>1) operations may be available simultaneously, and the program's inherent parallelism is exposed. The program may, in fact, be executed on a dataflow-oriented machine that takes advantage of the parallelism thus exposed to speed up execution. This is sometimes called a paradigm for multiprocessing since it provides an approach to organizing multiple processors to work together on a single program.

The dataflow program is translated from a dataflow graph, which is a directed graph. The nodes of the graph represent operators; the arcs represent the flow of data between successive operators. The basic operators are illustrated by the nodes in Fig. 6.25. The value generator node is used to insert constants into the data flow. The switch and merge nodes are used together to conditionally select blocks of code. The input to these nodes, "control," is a Boolean value.

Data values are not stored in variables but are either used as they are computed (called *data driven)* or computed as they are needed (called *demand driven).* Therefore, the dataflow paradigm does not have the problem of side effects.

True dataflow languages are largely experimental since they are intended to run on dataflow architecture machines, which are themselves still experimental.