Engineering with Computers 6, 1–15 (1990)

Engineering with Computers

Object-Oriented Programming for Engineering Software Development

Gregory L. Fenves
Department of Civil Engineering, University of California at Berkeley, Berkeley, CA, USA

Abstract. The representation of engineering systems in a manner suitable for computer processing is an important aspect of software development for computer aided engineering. The process of abstraction is a well-known technique for developing data representations. Objects are a mechanism for representing data using abstraction, and object-oriented languages are languages for writing programs to manipulate objects. The paper shows through examples the advantages of object-oriented programming for developing engineering software. Mathematical graphs are used as an abstraction for two problems: (1) sorting activities in a schedule and (2) ordering nodes and elements in a finite element mesh. Classes of objects are developed for generic graphs, activity procedence graphs, and graphs of finite elements meshes. Object-oriented program development leads to modular programs and a substantial reuse of code for the two problems.

1 Introduction

Computers can aid in a large variety of activities in the planning, design, simulation, fabrication, and operation of engineering systems. Recent developments in artificial intelligence, such as knowledge-based expert systems, have produced aids for planning and design, particularly in the conceptual and preliminary phases. In contrast, evaluation (also called analysis) is based on mathematical models of a system, for which algorithmic solutions are usually appropriate. There are similar mixture of approaches for computer aids in fabrication (or construction) and operation. Although very different problem-solving methodologies are needed for these activities, the common factor is the necessity of representing an engineering system in a manner suitable for computer processing. Most approaches to computer solutions emphasize the problem-solving methodology, whether by numerical methods, inference strategies, or logic, to the detriment of the data representation. This is not to argue that one is more important than the other, but rather, to emphasize that data representation is equally important in engineering software.

A widely advocated approach for developing data representations is through the process of abstraction. Abstraction is the separation of a concept from the implementation of that concept. It allows development of data representations that emphasize important properties of a system while hiding the details of how the properties are stored in the computer. The use of data representations produced by the abstraction process results in software that is flexible to changing needs and requirements.

Objects are a mechanism for developing data representations using abstraction. Objects are convenient for representing engineering systems, and object representations can be readily translated into computer programs using object-oriented programming languages. The purpose of this paper is to describe object representations and to demonstrate object-oriented program development for solving two engineering problems. The paper will contrast object-oriented programming with procedural programming (as implemented in languages such as Fortran, C, and Pascal) traditionally used to develop engineering software.

There are relatively few applications of object representations to engineering problems. Keirouz et al. [1] developed an object model for constructed facilities. The model includes primitive objects representing irreducible physical entities, domain objects as aggregates of other objects, and connection objects. Varghese [2] and Fenves [3] demonstrated the use of object representations for structural design and analysis, in particular for reinforced concrete beams. Powell [4] has outlined the use of objects in an abstraction model for engineering information and its use in data bases, and Spooner et al. [5] presented an object-oriented data model. Object representations have been used for other engineering systems, such as very large scale integrated circuits [6].

Offprint requests: Gregory L. Fenves, Department of Civil Engineering, University of California at Berkeley, Berkeley, CA 94720, USA.

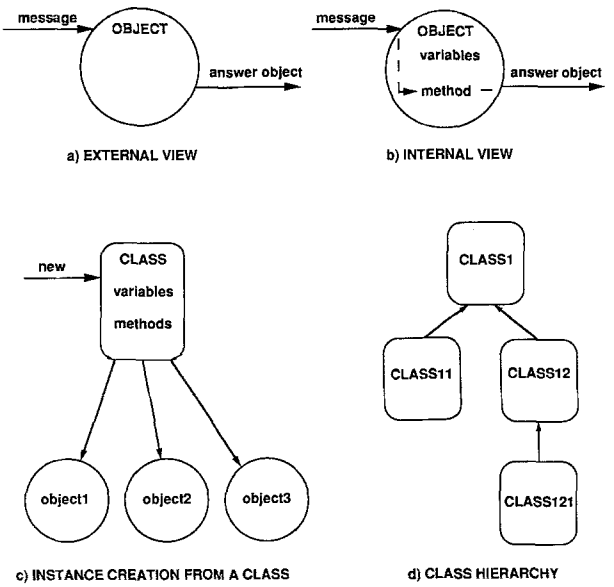

Fig. 1. Schematic views of objects and classes.

2 Objects and Programming Languages

2.1 *Definition of Objects and Classes*

An object has a name and the ability to perform operations on itself. The name allows reference to the object in the computer's memory. An object performs an operation on itself when the object is sent a *message*. Upon receiving the message, the object executes a procedure (also called a *method*) and answers the message with, in general, another object. The process by which an object responds to a message, whether by algorithm, inference, or logic, is unknown and unavailable to the sender; only the answer to the message is available. The external view of an object, messages and answers, is depicted schematically in Fig. 1a.

The internal view of an object consists of variables and methods, as denoted in Fig. 1b. An object can have variables, the values of which are other objects. An object's variables are only for use by the object; the variables are not accessible from other objects. A method is a sequence of instructions that when executed perform an operation on the object. The instructions for a method typically involve sending messages to the variables in the object.

Several objects may have the same variables and methods but represent distinct entities with different values of the variables. To avoid defining similar but distinct objects, an object is created as an *instance* of a *class*, where all instances have the same variables and methods. Classes are used to create new objects that respond to messages in the same way. The relationship between a class and the instances created from that class is indicated in Fig. 1c.

The manner in which a class of objects responds to messages is called the behavior of the objects. The *specification* defines the behavior of the objects by a precise description of the effect of each message. The *implementation* of a class involves writing the methods needed to provide the specified behavior. This separation enhances the process of abstraction because the specification defines the behavior of the objects and the implementation defines how the behavior is accomplished.

2.2 *Object-Oriented Programming Languages*

Object-oriented programming languages support the concept of objects and message sending. An object-oriented program consists of objects sending messages to each other. The program transforms the state of the objects, as encoded in the objects' variables, and the final objects represent the solution. The progam development process involves selecting the classes of objects needed to represent the problem and solution procedure, specifying the messages for the classes, and implementing the classes in a programming language [7].

Object-oriented languages are an advance in expressing the representation of data in programming languages. Early procedural languages provided control statements (e.g., loop, conditional, and case), but the data representation was relatively limited. As an example, Fortran allows integer, real, complex, logical, and character data types; arrays of one type are the only data structure available. Modern languages allow defined data types, such as a `typedef` in C and `type` in Pascal, and aggregate data structures. Although defined data types are useful, the languages do not allow operations on variables of the data types. Operations are only possible knowing the detailed contents of the variables of the defined types, and subprograms are the only mechanism for isolating operations on the variables. Unfortunately, this requires passing a large number of arguments to subprograms or declaring important variables globally accessible to many (or all) subprograms. Reference [8] gives a comprehensive comparison of procedural languages for engineering software development.

A defined data type with operations on variables of the type is essentially the concept of a class. An instance of a class corresponds to a variable of a defined data type, and the messages are operators

on the variables. Sending a message to an object is comparable to calling a subprogram, but there is an important difference between the two. Sending a message implies no knowledge of the implementation of the receiving object nor of the arguments for the message. In contrast, a subprogram call generally requires knowledge of the argument types and even content. To the extent that the data structures are hidden from subprograms [9], the more the data representation in a procedural program has the characteristics of objects and classes.

The language SIMULA introduced the idea of a class that can create instances that respond to procedures in the same way [10]. A project at the Xerox Palo Alto Research Center built upon the ideas in SIMULA, and the results is the language Smalltalk [11]. The latest version, Smalltalk-80, provides a very uniform application of the objected-oriented programming paradigm [12].

The important features of Smalltalk that enhance the use of objects for data representation and object-oriented programming are now described.

Inheritance

Because the behavior of objects in two classes may be related, a class can have a superclass and *inherit* the specification and implementation of its superclass. Applying the inheritance recursively, a class inherits all classes superior to it in the hierarchy. A single inheritance hierarchy of classes is shown in Fig. 1d, in which a class has at most one superclass. Usually a subclass is created to provide specialized operations compared to the superclass. Important operations tend to move up the hierarchy so that they are available to a larger number of classes. This can result in substantial reuse of class implementations by classes lower in the hierarchy.

Dynamic binding

The method invoked by a message is determined (or bound) at execution time. The selection of the method depends on the class of the object receiving the message and its position in the class hierarchy. When a message is sent to an object, the system first determines whether the method is defined for the receiver's class. If the method is defined, it is executed. If not, the search proceeds up the class hierarchy until a method corresponding to the message is defined in a superior class, at which point the method is executed. If the top of the class hierarchy is reached without finding a method, the system signals an error. This process of selecting a method depending on the class of the object contrasts sharply with procedural languages. In most procedural languages the compiler and linker determine which subprogram is executed based only on the name of the subprogram, not on the type of the arguments. The programmer is responsible for ensuring that the correct subprogram is called for particular types of the arguments, although the compiler may check that the type of the arguments and the formal parameters in the subprogram are equivalent.

Automatic memory allocation

Most languages rely on the programmer to allocate and manage storage of variables in memory; memory allocation is a burdensome responsibility and a frequent source of difficult to detect errors. Smalltalk frees the programmer and user from concerns of where to store an object and how much memory to allocate. Creating an object from a class automatically allocates memory to store the object. If sufficient memory is unavailable, the system undertakes garbage collection to reclaim storage. Memory is reclaimed only from objects that have no references in the system. Because it is not possible to delete an object to which other objects refer, it is not possible to send a message to an object that does not exist.

The features of dynamic binding and automatic memory allocation impose a penalty on execution speed. The benefit is that the combination of inheritance and dynamic binding provide a powerful mechanism for creating compact and modular programs that are amenable to incremental development and modification. The final section of the paper offers additional comments on the trade-off between development and programming flexibility on one hand and execution speed on the other hand.

A description of Smalltalk would be incomplete without mentioning its comprehensive computing and programming environment [13]. The system provides browsers for defining classes, compilers for methods, debuggers, and project managers. All are available as views through windows, with control determined by pop-up menus and other graphical controllers. The Smalltalk system provides and is implemented using many standard classes, including classes for number and collection objects in addition to classes for windows and other graphical objects [12]. Application programs in Smalltalk take advantage of the standard classes, which greatly reduces the development effort.

Since the introduction of Smalltalk other languages with object representations have become available, such as C++ [14, 15], Objective C [16],

LOOPS [17] and Flavors [18]. The first two are extensions of C for objects, and the latter two allow objects in Lisp. Hybrid languages, such as C++, have restricted binding and memory allocation features that can be determined at the time of compiling and linking. LOOPS combines objects with the functional approach of Lisp, and in particular, methods can be associated with any change of variables stored in an object to provide coordinated interaction between objects. (Smalltalk-80 has a more limited facility.) Object-oriented features are incorporated in Ada through packages [19], and the proposed Fortran 8X standard includes defined data types and operators on variables of the data types [20].

This paper concentrates on developing object representations for engineering problems and implementing the representations in Smalltalk. However, the representations are independent of the language, and the implementation could be done in any programming language that supports the use of objects.

3 Smalltalk Syntax

The only instruction in a Smalltalk program is sending a message to an object. For example, the expression

```
aNumber addTwo
```

sends the message addTwo to the object aNumber, the receiver of the message. A message invokes a method defined for the class of the object. The method for addTwo is

```
addTwo
    "answer the sum of 2 and the receiver"
  ↑ self + 2
```

In the method self is a psuedo-variable that represents the receiver of the message addTwo. The statement in the method sends the message + to self with 2 as an argument. Presumably, + is defined for the receiver (which it is if the receiver is an instance of a number class). The value is then returned (or answered), as indicated by the ↑ symbol, so the expression aNumber addTwo is an object whose value is the sum of aNumber and two. Unless a method answers an object, the default answer is the receiver of the message itself.

Variables and assignments

Variables are named by an identifier and variables can be assigned values, as, for example,

```
aNumber ← 4
result ← aNumber addTwo
```

The first statement assigns the integer object 4 to the variable aNumber. Then the object aNumber addTwo, which has a value of 6, is assigned to the variable result.

As indicated, self is a psuedo-variable, a variable that cannot be assigned a value. Other psuedo-variables are nil, true, and false, which are objects that have no value, true value, and false value, respectively. The latter two Boolean objects are frequently used to control execution.

Messages

Smalltalk has unary, keyword, and binary messages. A message without a argument, such as addTwo, is a unary message. A message with one or more arguments is a keyword message. For example, an expression to obtain the minimum of two numbers is

```
aNumber min: 3
```

The message min: is sent to the object aNumber with one argument, the integer object 3. The answer is the lesser of aNumber and 3. A binary message is a message with one argument that is normally used for operators with noncharacter symbols. The statement in the method for addTwo uses the binary message +. Another use of a binary message is in the expression

```
anItem < anObject
```

which evaluates to true or false.

Expressions are evaluated according to the precedence of the messages. The order of evaluation in decreasing precedence is unary, binary, and keyword messages. Messages of equal precedence are evaluated left to right. Parentheses override the precedence, and expressions are evaluated from inner to outer pairs of parentheses.

Blocks

A block is a class that allows deferred execution of statements. An instance of a block is indicated syntactically by enclosing the statements in square brackets. In the example

```
aBlock ← [aNumber ← 4. result ← aNumber addTwo]
```

aBlock is assigned a block that consists of two statements. Sending the message value to a block executes the statements in the block; the answer to the

message value is the result of the last statement. The expression aBlock value will evaluate to 6 in addition to assigning values to aNumber and result. Blocks can also have one or two arguments, as will as demonstrated in the examples later in the paper.

The execution of a program is controlled using blocks, by either sending a message to a block or passing blocks as arguments of a message. Conditional execution occurs by sending the message ifTrue:ifFalse: to a Boolean object. In the following example

```
anItem < anObject ifTrue: [anItem ← anObject] ifFalse:
  [anObject ← anItem]
```

the binary message < is sent to anItem with anObject as an argument. The answer of anItem < anObject is true or false, to which the message ifTrue:ifFalse: is sent. The first or second argument block is evaluated depending on whether the receiver is true or false, respectively. Blocks also respond to messages such as whileTrue: which evaluate the argument block while the receiver is true. An example is

```
[anItem < anObject] whileTrue: [anItem addTwo]
```

Additional information on Smalltalk and programming examples are available in refs. [12] and [21].

4 Object-Oriented Program Development

Object-oriented program development primarily involves specifying and implementing data representations for a given problem. There are three steps in the development:

- *Selection of classes.* Determine the classes of objects necessary to represent the problem and its solution; use subclasses to provide an increasingly specific representation of the problem.
- *Specification of the classes.* Define the operations on the objects in a class by specifying what each message does.
- *Implementation of the classes.* Select the variables for objects; for each message program a method to perform the specified operation.

As we will illustrate in the example problems, the three steps are performed in a top-down manner in two regards. First, the specification and implementation of a class are completely separate. The specification allows use of a class only knowing the messages that objects in the class understand. The development of object representations is also top-down. Decisions about the representation are deferred using subclasses, proceeding from the general to the specific as the depth of the class hierarchy increases.

4.1 *Example of Object-Oriented Program Development*

This section presents the object-oriented development of programs for solving two engineering problems: sorting activities for critical path method (CPM) scheduling and ordering nodes in a finite element mesh. Although the two problems seem unrelated, the abstraction of the problems as mathematical graphs can be directly represented using objects and the solutions implemented in an object-oriented programming language. Classes of objects represent generic graphs and subclasses specialize the information for graphs of activity precedences and finite element meshes.

An activity in a schedule has a duration and successor relationships with other activities. A successor of an activity is one that cannot begin until the activity finishes. Figure 2a shows examples of suc-

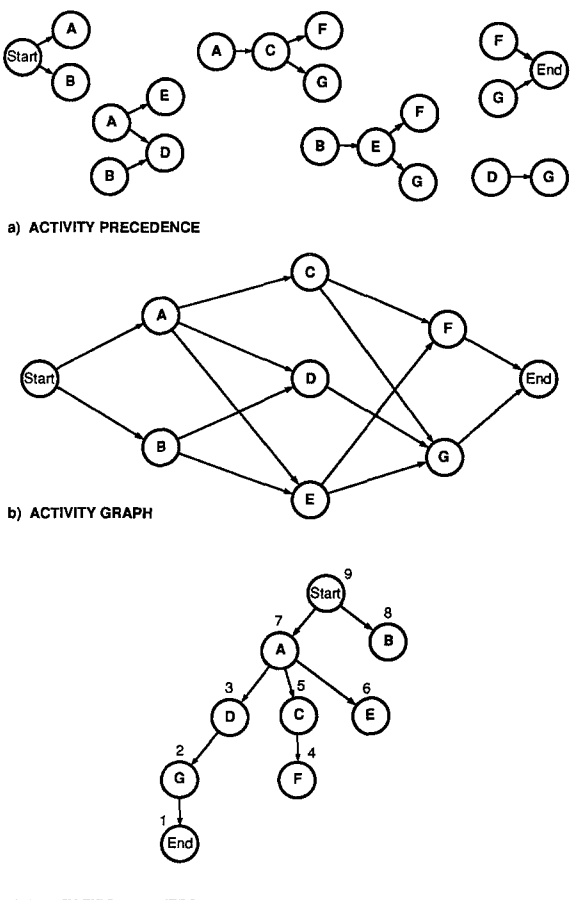

Fig. 2. Example of topological sorting of activities in a schedule.

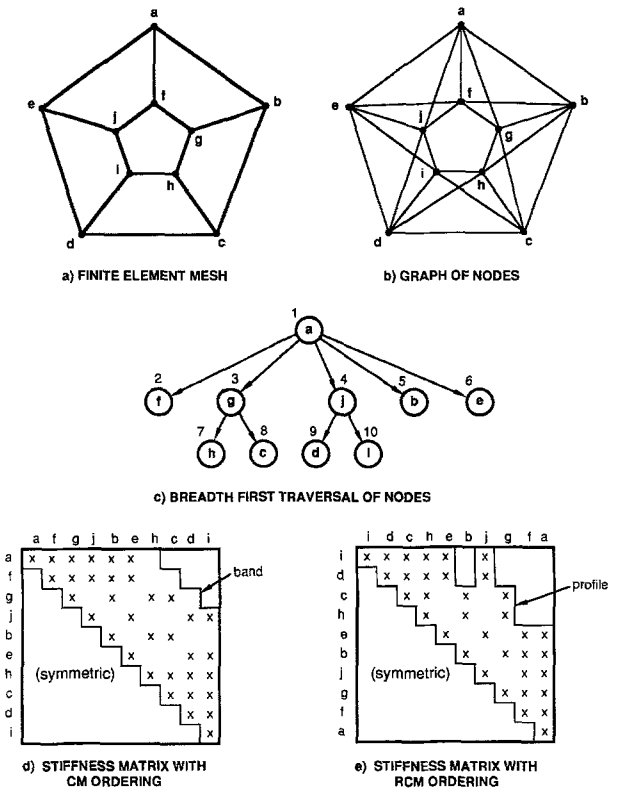

Fig. 3. Example of ordering nodes in a finite element mesh

cessor relationships, where a circle represents an activity and an arrow points to a successor activity. One way to schedule activities is first to sort the activities topologically into a list such that each activity appears in the list before its successor activities. After activities are in topological sort order, forward traversal of the list gives the earliest start and finish times and backward traversal gives the latest start and finish times, from which the critical path and floats can be determined. Only the precedences between activities are considered in this example; computation of the schedule and allocation of resources are not included. The example problem is to sort the activities topologically whose precedence is shown in Fig. 2a.

A finite element mesh consists of elements connected by edges, where nodes define the edges. The order of nodes determines the amount of memory required to store the symmetric stiffness matrix from a finite element mesh and the number of operations during Gauss reduction (or a variant) of the matrix. The nodes in a mesh should be ordered to minimize the band or profile of the coefficient in the stiffness matrix, which are two measures of matrix sparsity. Figure 3a shows an example of a mesh that consists of five quadrilateral elements, and the problem is to order the nodes such that the band or the profile of the stiffness matrix is reduced from a full matrix.

Graphs are abstractions that emphasize the topology of a problem and suppress information about the content. A graph consists of a vertex set and an edge set. An edge connects two distinct vertices and it may be undirected or directed to a vertex. In the graph of a schedule a vertex represents an activity and a directed edge connects to a successor activity [see Fig. 2b]. In the graph of a finite element mesh a vertex represents a node and an undirected edge represents a connection to another node through a common element [see Fig. 3b]. Traversal of a graph, where vertices are visited in a specified order, is a basic operation. Topological sorting and node ordering involve traversal of the graph representing an activity schedule and finite element mesh, repectively. A useful tool for working with a graph is a tree rooted at one vertex of the graph. The root for a tree of an activity graph is the start activity, and the root for a graph of a finite element mesh is the node selected as the first node in the ordering.

4.2 Selection of Classes

The first class, called Graph, represents rooted trees of graphs and defines operations for topological sorting and node ordering. Another class, called Vertex, represents vertices in a graph. Operations on an instance of Vertex allow visiting adjacent vertices in the graph. However, the representation of the adjacent vertices and the content of the vertices depends on the problem. An activity vertex must know the successor activities and information about the activity (e.g., duration); a node vertex must know the connected nodes and information about the node (e.g., geometric coordinates). The specialized information about a vertex, both adjacency and content, is represented by subclasses of Vertex.

Class Activity is a subclass of Vertex, whose instances represent activities in a schedule. An instance of Activity knows its successor activities. The class NodeAndElement contains common information about nodes and elements in a finite element mesh; classes Node and Element are subclasses of NodeAndElement that provide specific information. An instance of Node knows the elements connected to it; an instance of Element knows the nodes defining that element. A node can determine its adjacent nodes implicitly from this information, as can an element determine its adjacent elements.

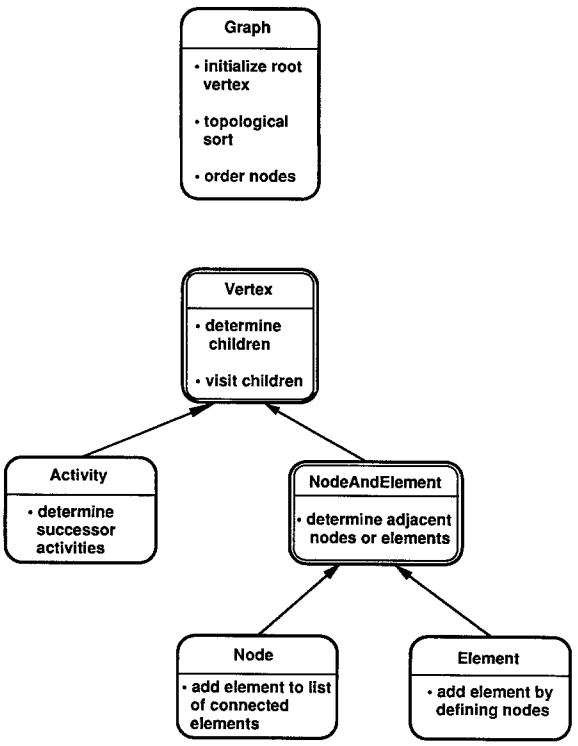

Fig. 4. Class hierarchy for example problems

The class hierarchy for the example problems is illustrated in Fig. 4. Classes are shown as boxes with a general description of operations defined for instances of the class. A class inherits the operations of all superior classes in the hierarchy. Vertex is termed an abstract class because its subclasses inherit common behavior, but no subclass is properly a subclass of another [12]. NodeAndElement is also an abstract class. Instances are only created from subclasses of an abstract class, never from the abstract class itself.

4.3 Specification of the Classes

4.3.1 Class graph

Data flow diagrams are useful for describing object-oriented programs. The data consists of objects and the transformations are accomplished by sending messages to objects. The transformations necessary to sort activities and order a finite element mesh are:

- Create a tree rooted at one vertex in the graph.
- Traverse the graph from the root vertex producing a list of vertices.

The data flow diagram corresponding to these transformations is shown in Fig. 5a. The specializations of the data flow diagram for topological sorting and node ordering are shown in Fig. 5b and 5c, respectively.

As an example, if object start is the start activity, the following operations will perform a topological sort of an activity graph:

```
aGraph ← Graph on:start
list ← aGraph topologicalSort
```

The first statement sends the message on: to class Graph, which creates an instance of the class; the instance is assigned to the object aGraph. The message on: has one argument, which is the root vertex for aGraph. The message topologicalSort is sent to aGraph and the answer is a list of the activity objects in topological sort order with start as the first object. The list is assigned to the object list, which for the problem in Fig. 2 contains objects for activities: start, b, a, e, c, f, d, g, and end, in that order.

There are a several orderings of nodes in a finite element mesh that reduce the band or profile of a stiffness matrix. Two widely used orderings are given by the Cuthill-McKee (CM) [22] and Reverse Cuthill-McKee (RCM) algorithms. They give the same band, but the RCM ordering usually gives a smaller profile [23]. If an instance, a, of class Node

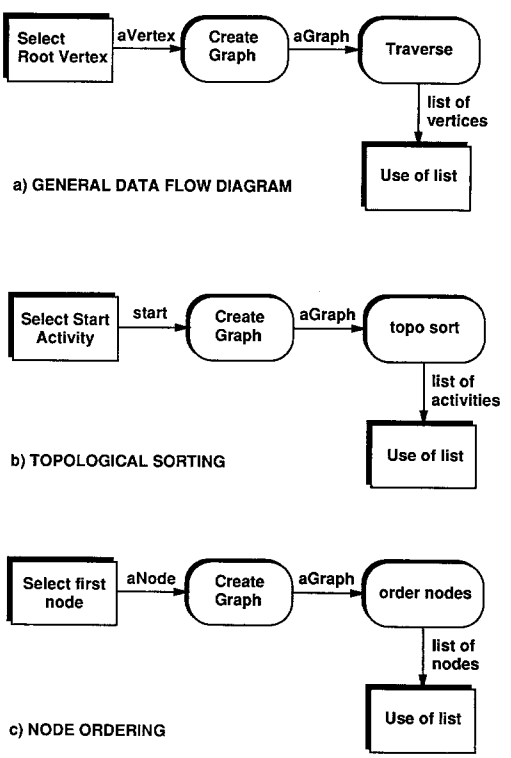

Fig. 5. Data flow diagrams for example problems

Table 1. Specification of class Graph

(a) Graph class messages	
on: aVertex	Answer an instance of the receiver with aVertex as the root vertex for the graph.
(b) Graph instance messages	
topologicalSort	Answer a collection of vertices in topological sort order with the root of the receiver first.
numberCM	Answer a collection of vertices in Cuthill-McKee order with the root of the receiver first.
numberRCM	Answer a collection of vertices in reverse Cuthill-McKee order with the root of the receiver last.

is the first node, the following operations will order the remaining nodes in the CM and RCM orderings:

list1 ← (Graph on:a) numberCM
list2 ← (Graph on:a) numberRCM

The messages numberCM and numberRCM are sent to distinct instances of class Graph that have a as the root vertex. The answer to numberCM is the object list1, which contains a list of node objects: a, f, g, j, b, e, h, c, d, and i, in that order. The object list2 is a list that contains the same node objects but in the reverse order. The band of the symmetric stiffness matrix for the CM ordering is shown in Fig. 3d, and the profile for the RCM ordering is shown in Fig. 3e.

Table 1 presents the specification of class Graph. The specification contains a precise description of the operation invoked by each message. The specification includes messages that are sent to the class, typically to create an instance of the class (Table 1a), and messages that are sent to instances of the class, in this case to obtain the desired solutions (Table 1b).

4.3.2 Class vertex

The class Vertex contains a minimum information required to visit adjacent vertices. Information about the adjacency and content of vertices is relegated to subclasses of Vertex as shown in Fig. 4. Only two types of operations on vertex objects are presented here: determination of adjacent vertices and traversal of the graph from the vertex itself.

A vertex can determine its adjacent vertices (called children) using the messages specified in Table 2a. If aVertex is an instance of a subclass of Vertex, the expression

aVertex children

gives the set of vertices adjacent to aVertex. Because information about adjacency is relegated to subclasses of Vertex, each subclass must implement a method for children.

An important operation on a vertex is to traverse the graph beginning at the vertex itself. There are two orders in which vertices may be visited during traversal of a graph:

- *Depth first traversal.* If vertex v is just visited and vertex w is adjacent to v, do a depth first traversal on each adjacent vertex to w.
- *Breadth first traversal.* If vertex v is just visited, visit all vertices w adjacent to v before visiting vertices adjacent to any w.

Table 2c gives the specification of the traversal messages. It is usual to perform an operation on each vertex visited during the traversal of a graph. One

Table 2. Specification of class Vertex

(a) Vertex instance messages for children	
children	Answer children of receiver.
childrenNotVisited	Answer a set of unvisited children of receiver.
degree	Answer the number of children of receiver.
(b) Vertex instance messages for visiting	
setPreVisited	Set receiver pre-visited.
setPostVisited	Set receiver post-visited.
notVisited	Answer true if receiver has not been visited.
preVisited	Answer true if receiver has been pre-visited.
(c) Vertex instance messages for traversal	
doDepth: aBlock	Traverse children of receiver in depth first order. Evaluate aBlock at post-visit of vertex.
doDepth: aBlock ifCycle: exBlock	Traverse children of receiver in depth first order. Evaluate aBlock at post-visit of vertex. If a cycle is found, evaluate exBlock.
doBreadth: aBlock	Traverse children of receiver in breadth first order. Evaluate aBlock at visit of each vertex.
doBreadth: aBlock visitBy: srtBlock	Traverse children of receiver in breadth first order. Visit children in order determined by srtBlock. Evaluate aBlock at visit of each vertex.

argument of the messages in Table 2c is aBlock, a block that is evaluated for each vertex visited. For example, the following expression prints vertices in a depth first traversal of a graph beginning at an instance of Node:

```
aNode doDepth: [:current|current printIt]
```

Because Node is a subclass of Vertex, it inherits the doDepth: message. The argument for doDepth: is a one parameter block, where the parameter current is a vertex visited in the traversal. The message printIt presumably prints information about current. The specification of doDepth: in Table 2c states that the argument block is evaluated during a post-visit of each vertex; a pre-visit could easily be defined. It is important for some graphs to be acyclic, so the message doDepth:ifCycle: specifies a depth first traversal, but the second block is evaluated if a cycle is detected. Typically, the second block contains an error notification.

Breadth first traversal of a graph is performed by sending the message doBreadth: to a vertex object. In some applications it is desirable to traverse the vertices at a given level in an order determined by a property of the adjacent vertices. This is done by the message doBreadth:visitBy:, where the second argument is a two parameter block to determine the order in which adjacent vertices are to be visited. The expression

```
aNode doBreadth: [:current|current printIt] visitBy:
  [:a :b|a degree <= b degree]
```

prints vertices in breadth first order beginning at aNode, where for each node the adjacent nodes are visited in order of increasing degree.

The traversal methods for instances of class Vertex are control operators that iterate over the vertices in a graph. In object-oriented programming languages, Smalltalk in particular, there is no distinction between messages that transform objects and messages that control execution.

4.3.3 Subclasses of vertex

Because an instance of Vertex does not know its adjacent vertices, the subclasses of Vertex are responsible for representing adjacent vertices. The message children to an instance of Activity answers the successor activity objects. The class NodeAndElement implements children for its subclasses. The message children to an instance of a Node answers nodes connected through common elements; the message children to an instance of Element answers elements connected by common nodes. Table 3 gives the specification of the subclasses of Vertex.

Table 3. Specification of subclasses of Vertex

(a) Activity instance messages for children	
children	Answer successor activities of receiver.
(b) NodeAndElement instance messages for children	
children	Answer adjacent objects with the same class as receiver.

4.4 Implementation of the Classes

This section presents the implementation of the classes specified in Tables 1 to 3. The implementation of a class involves defining the variables for objects in the class and programming a method for each message in the specification.

4.4.1 Class graph

The only variable required for an instance of class Graph is the root vertex. The class definition in Table 4a states that the superclass of Graph is Object (a class that defines generic behavior of objects), and each instance of Graph has one variable, rootVertex. The class message on:, which creates an instance of Graph with a root vertex, is implemented in Table 4b. All objects understand the message new, which answers a new instance of the receiver, class Graph in this case. The new instance is sent the message setOn: to assign the instance variable. The message, implemented in Table 4c, is private because it should only be invoked by methods for the class.

According to the specification in Table 1, methods for the messages topologicalSort, numberCM and numberRCM must be implemented. Each message invokes a specific graph traversal operation. In this example it is assumed that initially all vertices are not visited.

For an acyclic graph the order in which vertices are post–visited in a depth first traversal is the topological sort order in reverse [24]. The method for topologicalSort is implemented by sending the message doDepth:ifCycle: to the root vertex, as shown in Table 4d. If the graph has a cycle, the second argument block is evaluated in which error: sends the receiver an error notification. A list of vertices is maintained in the local variable index during traversal of the graph. As each vertex is visited, it is placed at the beginning of the list. After traversing the graph, the list contains the vertices in topologi-

Table 4. Implementation of class Graph

(a) Class definition

```
class Graph
  superClass: Object
  instanceVariableNames: rootVertex
```

(b) Class method

```
on: aVertex
    "answer an instance rooted at
    aVertex"
  ↑ (self new) setOn: aVertex
```

(c) Private instance method

```
setOn: aVertex
    "set root vertex of receiver"
  rootVertex ← aVertex
```

(d) Instance methods

```
topologicalSort
    "answer collection of vertices in
    topological sort order with the root of
    receiver first"
  |index|
  index ← OrderedCollection new.
  rootVertex
    doDepth:[:aVertex|index addFirst:
              aVertex]
    ifCycle:[↑ self error: 'Graph contains a
              cycle.'].
  ↑ index
```

(d) Instance methods (continued)

```
numberCM
    "answer collection of vertices in
    Cuthill-McKee order with the root of
    the receiver first"
  |index|
  index ← OrderedCollection new.
  rootVertex
    doBreadth:[:aVertex|index addLast:
              aVertex]
    visitBy:[:a:b|a degree <= b degree].
  ↑ index

numberRCM
    "answer collection of vertices in
    Reverse C-M order with the root of the
    receiver last"
  |index|
  index ← OrderedCollection new.
  rootVertex
    doBreadth:[:aVertex|index addFirst:
              a Vertex]
    visitBy:[:a:b|a degree <= b degree].
  ↑ index
```

cal sort order. In the method the list of vertices, index, is an instance of class OrderedCollection, a class for maintaining ordered lists of objects. Smalltalk-80 supplies this class; it allows inserting an object at the beginning and end of a list and enumerating the objects. Table 5 gives part of the specification of class OrderedCollection.

The Cuthill-McKee (CM) ordering of vertices is obtained by a breadth first traversal of the graph starting at the root vertex [22]. Vertices adjacent to a given vertex are visited in order of increasing degree. The method for numberCM involves sending the message doBreadth:visitBy: to the root vertex, as given in Table 4d. A list, index, is maintained during the traversal of the graph, and when a vertex is visited, it is placed at the end of the list. The method for numberRCM is similar to numberCM but visited vetices are placed at the beginning of index.

The implementation of class Graph illustrates two aspects of object-oriented programming that are beneficial for software development. The implementation only requires specification of other classes, not their implementation. The methods use classes Vertex and OrderedCollection, but only through their specifications as given in Tables 2 and 5. The implementations of Vertex and OrderedCollection can change without affecting their use in class Graph. This separation of specification and implementation supports top-down refinement and is a substantial benefit of object-oriented programming.

Table 5. Partial specification of class OrderedCollection[a]

(a) OrderedCollection class messages

new	Answer an empty instance of the receiver.
new: anObject	Answer an instance of the receiver with anObject as the first element.

(b) OrderedCollection instance messages

addLast: anObject	Add anObject as the last element in the receiver.
add First: anObject	Add anObject as the first element in the receiver.
removeFirst	Remove the first element of the receiver and answer it.
isEmpty	Answer true if the receiver is empty.

[a] Ref. [12].

Table 6. Implementation of the class Vertex

(a) Class definition

```
class Vertex
  superClass: Object
  instanceVariableNames: visited
```

(b) Instance methods for children

```
children
  self subclassResponsibility
childrenNotVisited
    "answer set of children not visited"
  ↑ (self children)
    select:[:aChild|aChild notVisited]
degree
    "Answer the number of children"
  ↑ (self children) size
```

(c) Instance methods for visiting

```
setPreVisited
    "set receiver pre-visited"
  visited←'pre'
setPostVisited
    "set receiver post-visited"
  visited←'post'
notVisited
    "answer true if receiver not visited"
  ↑ visited='no'
preVisited
    "answer if receiver pre-visited"
  ↑ visited='pre'
```

(d) Instance methods for traversal

```
doDepth: aBlock ifCycle: exBlock
    "traverse children of receiver in depth
    first order. Post-visit each child and
    evaluate aBlock. If a cycle is found
    evaluate exBlock"
  self setPreVisited.
  self children do:[:aChild|
    aChild preVisited ifTrue:
          [exBlock value].
    aChild notVisited ifTrue:
              [aChild doDepth: aBlock
                ifCycle: exBlock]
  ].
  aBlock value: (self setPostVisited)
doBreadth: aBlock visitBy: srtBlock
    "traverse children of receiver in
    breadth first order. Visit children in
    order determined by srtBlock. At
    each child evaluate aBlock."
      |q v|
      q←OrderedCollection with:
        (self setPreVisited).
      [q isEmpty]
        whileFalse:[
          v←q removeFirst.
          aBlock value: (v setPostVisited).
          (v childrenNotVisited
          asSortedCollection: srtBlock) do:
            [:aChild|q addLast:
                (aChild setPreVisited)]
        ]
```

The second aspect has to do with software reusability. Many applications use ordered lists of arbitrary objects and the availability of the class OrderedCollection means that the behavior does not have to be developed for each use. This generality is awkward or impossible in procedural programming languages.

4.4.2 Class vertex

The implementation of Class Vertex must provide the operations specified in Table 2. To keep track of a traversal, each vertex has a state of not-visited, pre-visited, or post-visited. The state is stored in one instance variable, visited, as given by the class definition in Table 6a. The messages specified in Table 2b query or update the state of a vertex and Table 6c gives the implementation. The method for children cannot be implemented in Vertex because adjacent vertices must be determined by the subclasses. As shown in Table 6b, if children is sent to an instance of Vertex, the message subclassResponsibility invokes an error indicating that children must be implemented in a subclass. In the method for degree the object self children is a collection of the adjacent vertices of the receiver. Instances of collection classes understand the message size, which answers the number of objects in the collection. Collections also understand select:, which answers a collection of the objects in the receiver that evaluate the argument block as true.

In a depth first traversal of a graph with post-visit each adjacent vertex is visited before the vertex itself. A vertex is in a pre-visit state when at least one adjacent vertex has been visited, but not all. If a vertex is visited while a pre-visit is still pending on it, the graph contains a cycle. The algorithm shown here uses recursion to perform a depth first traversal with post-visit, including detection of cycles, assuming all vertices are initially not visited:

```
depthFirstTrav on v
  Set v pre-visited
  For each child w of v
    If w pre-visited, then a cycle exists
    If w not visited, then depthFirstTrav on w
  Visit v
  Set v visited
```

Table 7. Implementation of the subclasses Vertex

(a) Activity class definition

```
class Activity
  superClass: Vertex
  instanceVariableNames: followers
```

(b) Activity instance method

```
children
    "answer successors of receiver"
  ↑ followers
```

(c) NodeAndElement class definition

```
class NodeAndElement
  superClass: Vertex
  instance VariableNames: list
```

(d) NodeAndElement instance methods

```
children
      "answer children of receiver through
      indirect reference"
  |allRefs|
  allRefs ← Set new.
  (self list) do:[:aRef|
      allRefs addAll: (aRef list)].
  allRefs remove: self.
  ↑ allRefs
list
      "answer objects connected to
      receiver"
  ↑ list
```

The method for doDepth:ifCycle:, given in Table 6d, implements the algorithm with almost a line-by-line equivalence between the algorithm and the program code. If cycle detection is not necessary, doDepth: sends the message doDepth:ifCycle: with an empty exception block.

In the example of topological sorting of activities, Fig. 2c shows the rooted tree obtained from the depth first traversal of the activity graph. The numbers in the figure are the order in which the vertices are post-visited (the topological sort order in reverse).

Breadth first traversal of a graph can be implemented by storing vertices to be visited in a queue (a queue is first-in, first-out list). The algorithm given here performs a breadth first traversal [24]:

breadthFirstTrav on s
  Add s to queue
  While queue is not empty
    Remove v from queue
    Visit v
    Set v visited
    For each child w of v
      If w not visited and w not in queue, then add
        w to queue

The method for doBreadth:visitBy: in Table 6d implements the breadth first traversal algorithm. The local variable q is an instance of OrderedCollection used as a queue. The message addLast: adds to the queue, and removeFirst removes from the queue (see Table 5). A vertex is set pre-visited when put in the queue, so the message childrenNotVisited answers children that are not visited nor in the queue. The unvisited children are added to the queue in the order specified by the argument srtBlock. If the order in which adjacent vertices are visited is not important, doBreadth: sends the message doBreadth: visitBy: with a sort block that always evaluates true.

The sorting is performed by the class SortedCollection, a subclass of OrderedCollection that orders objects based on a sorting criterion. For example, the expression

```
aSortList ← aList asSortedCollection: srtBlock
```

sorts the collection aList according to the two parameter block srtBlock that evaluates true if the values of the two parameters are in the sorted order. The answer is an instance of SortedCollection assigned to aSortList.

In the example of finite element node ordering Fig. 3c shows the rooted tree obtained from the breadth first traversal of the graph of nodes. The numbers in the figure are the order in which the vertices are visited (the CM ordering).

4.4.3 Subclasses of vertex

Table 3 gives the specification of the subclasses of Vertex. The definition of class Activity is shown in Table 7, along with the implementation of the message children. An instance of Activity has one variable, followers, which is the collection of successor activity objects. Table 7 also gives the definition of class NodeAndElement. In the method for children the class Set, a collection of nonduplicate objects, is

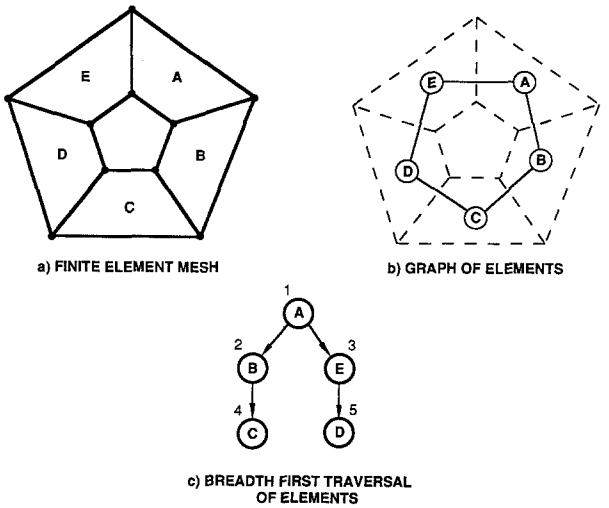

Fig. 6. Example of ordering elements in a finite element mesh

used so that no reference is repeated in the connectivity of nodes or elements.

4.5 Extensions of the Node-Ordering Example

The ordering of nodes in a finite element mesh is important for band, profile, and active column reduction of the stiffness matrix. The frontal method is an alternative procedure for reducing the stiffness matrix [25]. The number of active equations in the front depends on the ordering of the elements in the mesh. The Cuthill-McKee algorithm applied to the graph of elements in a mesh provides a convenient ordering of elements that reduces the front size. The graph of the element connectivity for the mesh in Fig. 3 is shown in Fig. 6b. The element ordering can be obtained using the previously developed classes. If A is an instance of Element that is the first element, then the expression

```
list3 ← (Graph on:A) numberCM
```

gives the list of element objects in CM order, which for the example in Fig. 6 is A, B, E, C, and D, in that order. The breadth first traversal of the element graph is shown in Fig. 13c. The preceding expression is valid because Element is a subclass of Vertex, and the messages for class Graph work with any root vertex that is a subclass of Vertex. The correct connectivity of elements is provided because instances of Element and Node respond to the message children, using the method in class NodeAndElement.

The number of operations to traverse a graph is of the order of the number of edges, so traversing the graph of elements is faster than traversing the graph of nodes (compare Fig. 3c with Fig. 6c), particularly for meshes with higher-order elements. This observation has led to two-step procedures for node ordering [26, 27], where the elements are ordered to reduce the front and then the nodes connected to the elements are ordered. The two-step procedure can be specified as an operation on instances of Graph, where a breadth first traversal begins with an element as the root vertex and the visit operation consists of enumerating the nodes attached to each element. The following expression implements this node ordering procedure:

```
rootElem doBreath:[:aElem|
      aElem do:[:aNode|index addLast:aNode]
      visitBy:[:a:b|a valency >= b valency] ]
    visitBy:[:a:b|a degree <= b degree]
```

This expression uses a message do:visitBy:, which visits each node directly connected to an element in the order specified by the second argument block. The message valency answers the number of elements connected to a node [26] or a weighting of the number of elements [27].

If a frontal method is desired, it is not necessary to number the elements because the breadth first traversal of the element graph gives the order in which elements are processed into the front. Consequently, the visit block for the traversal can be the operations necessary for the frontal solution. This can be implemented as

```
fr ← Front new.
rootElem doBreadth:[:aElem | fr reduceWith:aElem]
  visitBy: [:a:b|a degree <= b degree].
```

The object `fr` is an instance of class `Front`, which has operations, such as reduceWith:, for maintaining and reducing the equations. In the processing of an element the element object would need to understand a message to form its stiffness matrix for assembly into the front.

5 Discussion

Characteristics of object-oriented programs

The examples demonstrate that object-oriented development and programming results in software that has two important characteristics: modularity and reusability. Modularity is enhanced because an object hides its data in instance variables only available to the object. The concept of data hiding has

been shown to be an important characteristic of modular software [9]. Modularity is accomplished by separating the specification from implementation of the classes. A class can be used only knowing the specification; the implementation can be changed without affecting the use as long as it provides the behavior in the specification.

The dynamic binding between messages and methods depending on the class of the receiver and inheritance hierarchy has the effect of producing methods with very few statements because of the substantial reuse of classes. The reuse of classes reduces the effort required to develop and implement new applications. Usually classes previously developed for a problem provide a good starting point for extensions. This was evident in the extensions to the finite element ordering problem, where no additional programming was required to order elements and very little effort was needed to implement the two-step procedure.

Efficiency

There are three types of efficiency to consider when developing and using computer programs: efficiency in design, implementation, and execution. Object representations appear to provide large efficiencies in the design and implementation of programs. The incremental compilation, versatile debuggers, and software tools available in systems such as Smalltalk-80 make programming very rapid. However, the features of object-oriented programming that provide efficiency in development and implementation (inheritance, dynamic binding, and automatic memory allocation) impose penalties on execution efficiency of a program.

The execution penalty is of a significant nature for many engineering problems of realistic scale using the current generation of object-oriented programming languages. Given the advantages of object-oriented programming, compromises between development and execution efficiency are possible. Hybrid languages such as C++ bind at compile and link time, and the programs execute nearly as efficiently as standard C programs. Future versions of object-oriented languages may allow restrictions on dynamic binding, resolvable at the time a method is compiled, for faster execution. The boundary between compile time and execution time binding may be selected depending on the state of the program development, moving from dynamic to static binding as the program is customized for a particular application.

Single versus multiple inheritance

The class hierarchy for the example problems (Fig. 4) requires that Node and Element be subclasses of Vertex. This relationship is only necessary for a limited aspect of what nodes and elements represent. Activity is also a subclass of Vertex, but there is no natural relationship between activities, nodes, and elements, although they are all subclasses of the vertex class for the purpose of graph traversal. Activities, nodes, and elements may need some additional behavior that would be represented by a new class. Because standard Smalltalk-80 allows a class to have only one superclass, the new class would either have to be a sub- or superclass of Vertex, even though it may have no relationship with vertices or graph traversal operations.

Multiple inheritance allows a class to have more than one superclass, recognizing that a class may inherit properties from several classes. Instead of the restriction of the class hierarchy to a tree, the class relationships from a directed acyclic graph. The major difficulty with multiple inheritance is resolving conflicts in names of variables and messages along different paths in the class structure. Recent enhancements to Smalltalk-80 allow limited use of multiple inheritance [28].

6 Conclusions

This paper has presented the use of objects for representing engineering problems. Objects support the process of abstraction, which is important for developing flexible software for aiding in the design, simulation, and operation of engineering systems. The implementation of object representations using object-oriented programming languages is straightforward, and the resulting software has many beneficial characteristics from a software engineering viewpoint. It can be expected that object representations and object-oriented programming techniques will become an important factor in the development of future software for computer aided engineering.

References

1. Keirouz, W.T.; Rehak, D.R.; Oppenheim, I.J. (1987) Development of an object oriented domain model for constructed facilities. In: Artifical Intelligence in Engineering: Tools and Techniques (Eds. D. Sriram; R.A. Adey). pp. 259–271
2. Varghese, K. (1988) An object-oriented representation for design of reinforced concrete beams. Thesis submitted in

partial fulfillment of the requirements for the Master's Degree of Science in Engineering, The University of Texas at Austin, Austin, TX, May
3. Fenves, G.L. (1988) Object representations for structural analysis and design. In: Proceedings of Fifth Conference on Computing in Civil Engineering, ASCE, Alexandria, VA, March, pp. 502–511
4. Powell, G.H.; Bhateja, R. (1988) Data base design for computer-integrated structural engineering. Eng. Comput. 4, 135–143
5. Spooner, D.L.; Milicia, M.A.; Faatz, D.B. (1986) Modelling mechanical CAD data with data abstraction and object-oriented techniques. In: Proceedings, IEEE, pp. 416–424
6. Wolf, W. (1986) An object-oriented, procedural database for VLSI chip design. In: Proceedings, IEEE 23rd Design Automation Conference, pp. 744–751
7. Booch, G. (1986) Object-oriented development. IEEE Trans. Softw. Eng. SE-12(2)(February), 211–221
8. Holtz, N.M.; Rasdorf, W.J. (1988) An evaluation of programming languages and language features for engineering software development. Eng. Comput. 3(4), 183–199
9. Parnas, D.L. (1972) On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058
10. Dahl, O.-J.; Hoare, C.A.R. (1972) Hierarchical program structures. In: Structured Programming (Eds. O.-J. Dahl; E.W. Dijkstra; C.A.R. Hoare). New York: Academic Press, pp. 175–220
11. Ingalls, D.H.H. (1983) The evolution of the Smalltalk virtual machine. In: Smalltalk-80: Bits of History, Words of Advice (G. Krasner, Ed.) Reading, MA: Addison-Wesley, pp 9–28
12. Goldberg, A.; Robson, D. (1983) Smalltalk-80: The Langauge and the Implementation. Reading, MA: Addison-Wesley
13. Goldberg, A. (1984) Smalltalk-80: The Interactive Programming Environment. Reading, MA: Addison-Wesley
14. Stroustrup, B. (1988) What is object-oriented programming? IEEE Softw. 5(3)(May), 10–20
15. Stroustrup, B. (1987) The C++ Programming Language. Reading, MA.: Addison-Wesley
16. Cox, B.J. (1986) Object Oriented Programming. Reading, MA: Addison-Wesley
17. Stefik, M.J.; Bobrow, D.G.; Kahn, K.M. (1986) Integrated access-oriented programming in a multiparadigm environment. IEEE Softw. 3(1)(January), 10–18
18. Symbolics, Inc. (1984) Reference Guide to Symbolics—Lisp. Cambridge, MA
19. Saib, S. (1985) Ada: An Introduction. New York: Holt, Rinehart and Winston
20. Global Engineering Documents Incorporated. (1987) 8X Draft Proposed American National Standard for Fortran. Santa Ana, CA, October
21. Kaehler, T.; Patterson, D. (1986) A Taste of Smalltalk, New York: Norton
22. Cuthill, E.; McKee, J. (1969) Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of 24th National Conference of the ACM, Association for Computing Machinery, pp. 157–172
23. Liu, W.-H.; Sherman, A.H. (1976) Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. SIAM J. Num. Analy. 13(2), 198–213
24. Tarjan, R.E. (1983) Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA
25. Irons, B.M. (1970) A frontal solution program for finite element analysis. Int. J. Num. Meth. Eng. 2, 5–32
26. Fenves, S.J.; Law, K.H. (1983) A two-step approach to finite element numbering. Int. J. Num. Meth. Eng. 19, 891–911
27. Hoit, M.: Wilson, E.L. (1983) An equation numbering algorithm based on a minimum front criteria. Comput. Struct. 16(1–4), 225–239
28. Borning, A.; Ingalls, D.H.H. (1982) Multiple inheritance in Smalltalk-80. In: Proceedings, AAAI National Conference, Pittsburgh, PA