European Journal of Business & Social Sciences
Available at https://ejbss.org/
ISSN: 2235-767X
Volume 07 Issue 05
May 2019

An Object-Oriented Approach to Structural Analysis and Design

[1]Rupesh Kumar Singh, [2] Dr. Tryambak Hirwakar

[1]Research Scholar, Department of Computer Science, Sri Satya Sai University of Technology & Medical Sciences, Sehore , Madhya Pradesh

ABSTRACT

This paper considers the application of an object-oriented perspective in developing software for structural engineering applications. A basic introduction to object-oriented programming is given followed by an overview of how issues of concurrency and persistence can be addressed and exploited within an object-oriented framework. A more in-depth presentation is then given of a structural analysis program being developed using an object-oriented approach, which provides a further illustration of the usefulness and potential of the object-oriented perspective.

Keywords : Phenomenon, Engineering, Fundamental, Topology, Object-Oriented

INTRODUCTION

Current structural engineering software is based largely on computational perspectives and methods that have changed little since the days when batch mode interaction with computers was the norm, although this is generally hidden from the user by means of pre-and postprocessing software. For example, one of the most fundamental of the traditional assumptions concerning how structural analysis is done is that analysis is performed on well-defined, essentially static objects. Thus, most analysis packages are written in a language (e.g. FORTRAN) and style (procedural) well suited to this type of computing. The analysis itself remains essentially a batch job, with preprocessing and post-processing modules serving as the users. In design contexts, however, one is typically dealing with evolving, inherently dynamic data, with model objects having many attributes beyond those that are purely numerical. The trend in treating such issues generally has been to link independently-developed programs, achieving a measure of integration by limited or indirect communication using shared data (e.g. [!]). This is clearly a sensible manner in which to proceed for the short and medium range future, since it provides a degree of integration using an existing, albeit heterogeneous code. This integration, however, is locked in to the granularity, distribution (over time and processors), modularity, and fundamental perspectives of what has traditionally existed, rather than what should or could exist. There is no reason to expect that the best way to do several tasks together is the same as doing the tasks independently. It therefore would appear sensible to consider starting over with a clean slate, developing fundamentally integrated tools and methods which can allow better and more direct use of newer hardware and software technologies. This paper considers one approach to

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

achieving this type of comprehensive redevelopment of structural engineering software: object-oriented programming [2]. There is a rich and growing collection of object-oriented programming languages and concepts that can be used to support a wide class of features, including inheritance (sharing), data abstraction, concurrency, and persistence.

OBJECT-ORIENTED PROGRAMMING

There is no general consensus concerning a complete definition of the meaning of the term 'object oriented programming' (see (3) for an interesting formal definition scheme and clear treatment of the issues), and aspects of object-oriented programming are becoming increasingly common in a variety of languages. For the purposes of this paper, we will adopt the viewpoint that object-oriented programming is more than simply a set of tools and constructs to be added to existing languages, but is actually its own distinct perspective on (or paradigm for) computing [4]. Other computational perspectives include: procedural programming, in which a program is viewed as a set of algorithms; functional programming, in which a program is viewed as a flow of data through a lattice of function nodes; and logic or constraint-oriented programming, in which a program is viewed as a set of relations or constraints between input and output. Object-oriented programming can be considered as the viewing of a program as an analog model of a real-world phenomenon. A program th11s builds an environment containing objects (such as beams, floors, and columns) with attributes analogous to the real-world counterparts (such as weight, length, and material properties) and analogous behaviors (deformation under load, adding self weight to a structure). Furthermore, objects communicate with each other and the user, and can be given necessary skills and 'intelligence' for carrying out requested tasks. The problem solving domain thus consists of independent agents interacting and performing tasks. This perspective has the

desirable feature of allowing problem solution and program development to use natural modeling of physical and abstract entities within a given problem domain.

Creating an object-oriented program can be considered a three-step process: first, one decides what types of objects are needed; second, one decides what these objects need to be able to do; and finally, the actual code is written which teaches au object how to carry out it tasks. This approach is very powerful for several reasons. First, since the details of carrying out tasks and handling data are distributed locally to each type of object, these details can be hidden from other objects and the global environment. This means creating and controlling interactions between objects can be done through intercommunication of what to do, as opposed to detailed instructions of how to do it. Also, since objects present themselves to the external world in a directly meaningful way (e.g. 'I am a beam on the third floor connecting such and such ... ' as opposed to 'I am array element 5 in Element Array'), it is relatively easy to identify and reason about them, regardless of whether the observer is a programmer, a user, or another object. This is particularly useful for program development, since the details of how an object carries out its tasks can be changed without side effects, but it is not difficult to see how this can also be useful while the program is in use, since an object can dynamically change the internal details of its behavior to suit the circumstances.

One of the more powerful features of object· oriented programming is the capability for sharing attributes and skills (by means of class inheritance, as in SIMULA-like languages [5] or by delegation [61): new object types can be built up from existing types, and the new types obtain the capabilities and knowledge to do everything the subtypes can do without additional coding. For example, assume one has developed some basic graphical entities (lines, points, etc.) that are capable of drawing

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

themselves, moving themselves, and so on. Then, in developing a structural element for analysis, such as a truss element, it is possible to obtain the required geometric information and capabilities by inheriting them from the line type directly. This can be carried further by considering the subsequent development of a beam element, which could inherit both its geometric characteristics and a large proportion of its structural characteristics from the truss element type. For purposes of design, one could imagine various member objects which know their own detailed section prop· erties, and which know how to check themselves for code compliance. By building an additional object which inherits both from elements that are able to analyze themselves and from elements that are able to design themselves, the result is a member that can both analyze itself and design itself, and it could be told to do these tasks at whatever time or place was convenient. In effect, by means of inheritance, the code that goes into defining each type of object and the corresponding behaviors can be reused. In addition to providing programming convenience, this capability can also be exploited by users and a dynamically adaptive environment by providing flexible extensibility of the object environment. This capability for user-extensibility can be very useful in design office use.

There are an ever-increasing number of object-oriented programming languages or language extensions available. A partial list includes fully object-oriented languages such as SmallTalk-80[7], Trellis/Owl [81 and BETA [9J; LISP language extensions such as LOOPS [IO], Flavors [11}, and the newly adopted Common LISP Object System [12); and procedural language extensions such as C++ [13], Objective·C [14], and Object Pascal [15]. Each of these languages/language extensions has its own particular features and limitations.. The program described below uses the LISP extension Flavors, and will migrate to the Common LISP Object System (CLOS) when it becomes available. The principal computational model in both Flavors and CLOS is that of generic functions operating on objects. In essence the generic function represents a high-level generic task to be performed, such as 'draw'. The actual result of applying this same function to different types of object differs according to the object type, since each object type has its own method for carrying out the generic task. For example, if a line object and a triangle obje.ct are each given the generic task of drawing themselves (i.e. operated on by the 'draw' function), the line object will respond by drawing a line, while the triangle will respond by drawing a triangle. This illustrates one technique for making it possible to program from the perspective of what to do (draw objects) as opposed to how to do it (get each object; if it is a line draw a line, if it is a triangle, draw a triangle.).

CLOS has the further capability of directly handling generic functions whose actual completion method depends on more than one object type (multimethods). An example of where this can be useful in a structural setting is provided by considering the various people who are typically involved in the creation of a structure. The architect, the structural engineer and the construction engineer all work with the same structure, but their individual views of the structure and its components are typically very different. The architect may consider a beam as an architectural element with a physical presence and a load bearing capability. The structural engineer is interested in how the beam will bear its load and therefore is more interested in qualities such as the beam's section modulus and yield strength. The construction engineer is interested in a third set of qualities which includes all activities and materials associated with fabricating and erecting the beam. Clearly the proper method for carrying out the generic task of having a structural object describe itself depends on the receiver of the description. By defining different types of screen objects for each type of user, this can be handled nicely by means of a generic function of two arguments: 'describe this object on that screen·. The details of selecting which method to use for a particular object/screen combination do not need to be programmed: in essence the objects themselves take care of this.

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

One of the chief criticisms of object-oriented programming is that its overhead makes it inherently inefficient with respect to speed. The principal speed penalty is related to the speed of function calls as compared to method dispatches, and to the late (runtime) binding typically associated with object· oriented environments. (The latter is not really an inherent property of object-oriented computing, and can in fact be overcome by introducing strong typing when necessary [8].) Although from one perspective these criticisms are legitimate, from a larger perspective they are not as important as they first appear. In the first place, the overall efficiency penalties associated with object-oriented programming can be modest: the Symbolic Flavors extension induces a penalty of the order of 10-15% [16]. Second, as computing is done in increasingly interactive modes, the perception of and need for speed is somewhat different than in the case of batch mode processing. A computer can borrow many hundreds of microseconds from a user over time without any perception on the user's part. Thus, by means of incremental and background processing, response times can be kept sufficiently close to 'instantaneous' for many tasks. For example, in the structural analysis program described later, there need be no distinct solution phase, which means there is no point at which the user must wait for a large number crunch to occur. The most important point, however, is that those issues directly addressed by object-oriented programming-software development, maintainability, and usability-are becoming dominant over raw speed issues. Computer time is no longer the precious commodity it once was; human time is still the increasingly precious commodity it always has been.

CONCURRENCY ISSUES

As indicated by the multimethod example in the previous section, there are many different types of people involved in the creation of a structure. It is becoming increasingly clear that the more interaction and communication between the various parties involved the better, and the earlier this communication takes place the better [l 7]. Providing a computational environment in which such interaction can occur requires mechanisms for concurrency and distributed processing. With respect to concurrent and distributed processing, the object-oriented approach is again particularly attractive, as object-oriented programming encourages localized, distributed groupings of data and tasks. It is thus relatively natural to ually how tasks and data can be distributed over time and processors. Furthermore, objects can be embedded in other objects in hierarchical fashion, allowing great flexibility in defining and altering granularity both with respect to data and tasks.

Given this conceptual attraction of the object oriented approach, there has been a large amount of effort given to developing concurrent and distributed object-oriented programming languages and theories. This development has proceeded largely along two lines: (1) defining all computations in terms of independent, asynchronously-communicating computational agents, such as the Actor model of Hewitt [18, 19]; and (2) extending existing object oriented systems to include concurrent and distributed capabilities by various means as described below.

The Actor model has spawned languages such as Act 3 [20] and ABCL/R [21]. The Actor model is based on asynchronously communicating entities called actors, which are defined by a mail address, a behavior, and a set of acquaintances to whom each actor can send messages [19]. Computations are performed by creating actors with appropriate (generally changeable) behavior, which respond to received messages according to their behavior, creating new actors, passing more messages, altering their own behavior, and so on. In the actor model, objects are intentionally distinct and independent: thus there is no direct support for inheritance of behavior, since this requires a hierarchical dependency among object types. On the other hand, the complete independence of objects is ideally suited for concurrent and distributed implementation, particularly in the case of large numbers of tightly coupled processors.

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

Extending existing object-oriented systems to include concurrent and distributed features requires preserving the desirable features of such systems while overcoming the difficulties inherently associated with these features. For example, the feature of sharing implies inter-object dependence, which immediately presents constraints in terms of arbitrarily distributing objects across different address spaces. Some examples of languages and extensions treating such issues include Concurrent SmallTalk [22], Distributed SmallTalk [23], Hybrid [24), Parallel Objects [25], and Emerald [26). ln the spirit of object oriented programming, each of these systems/ extensions attempts to support concurrency and/or distribution of tasks in as transparent a manner as possible.

Both the Actor-based languages and the extended object-oriented languages are essentially experimental in nature. Nevertheless, the basic concepts and methods can be used to explore different ways in which concurrency and distribution can be used productively in structural engineering applications. Although there has been much research on specific applications of parallel and distributed processing in structural applications, this research has typically focused on local methods of speeding up particular processes (27, 28). The focus has not been on doing new kinds of things, but rather on doing the same types of things faster. Although these local algorithms can still be implemented within an object oriented environment, an object-oriented approach can provide a natural formalism for performing traditionally distinct tasks (such as design, analysis, and construction scheduling) concurrently. Some aspects of this are discussed further below.

DATABASE ISSUES

There is a clear need for persistent storage of information in engineering applications, along with the need for capabilities to retrieve and manipulate this information . It is the role of database systems and languages to provide such storage and retrieval capabilities. Given that engineering information generally involves physical or conceptual objects and their interrelations, object-oriented concepts applied to databases can again provide a natural modeling mechanism. In fact, it has been suggested that object oriented databases might be the ideal way to provide persistent storage and synchronized sharing of data regardless of the area of application [29). Due to the need for persistence and the attractiveness of the object-oriented approach, there have been a variety of efforts made both to embed (transparently) persistence within object-oriented languages and extensions [30, 31], and to build independent object oriented database systems [32). In each case the goals arc similar, and it is likely the two approaches will converge in the middle, resulting in object-oriented programming environments with extremely large memory space and build-in persistence.

Relational databases are often used in design contexts since they provide direct support for handling relations between data, and since these relations are not static; that is, they can be dynamically altered as relations evolve. Nevertheless, the topology of relational databases remains flat , and the operations that can be performed on data are typically removed from the data themselves. Object-oriented databases, on the other hand, allow for much more expressive power in defining relations and dependencies, and also make it possible to group data and operations together. Thus data manipulations can be physically as well as conceptually encapsulated with the data. Queries to the database can thus be kept more abstract, and generally more concise [33].

ANALYSIS ISSUES

Applying object-oriented programming to structural analysis and finite element analysis has been considered recently by Rehak and Baugh (34, 35), Fenves [36) and Miller (37). Although some of the directions taken in each of these papers differ , there a shared conceptual foundation: using object

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

oriented programming to make programs more closely model the concepts under consideration, both for aiding program development and for enhancing program use. The program to be desired here is essentially a further development of the program described in [37], although there are several fairly substantial changes that have been made. An overview of the program can best be gained by following the same three-step pattern described above for developing object oriented programs: consideration of the types of objects the program defines; examination of the tasks the objects are able to perform; and finally looking at the details of task implementations.

The principal structural object types are Degrees of Freedom (DOFs}, Joints, and Elements. Information concerning these types is summarized in Tables 1-3. Each table provides a listing of object types from which the object in question inherits (Inherits From), a listing of the information associated with the object (Knows About; these are known as instance variables) along with an indication, where appropriate, of the type of the information (Implemented As), and a listing of the tasks the object is capable of performing (Skills). The program is built from the perspective that for purposes of analysis a structure can be viewed as a collection of DOFs associated with Joints connected by Elements.

The fundamental structural object is the DOF as described in Table I. The object type 'Local DB' referred to in Table I corresponds to a simple database object which is capable of storing simple key /value tables, and a 'DB-able Object' is an object that can be stored in such a table: any object can be made storable by simply inheriting from the DB-able type. The reason for introducing these types is to allow for more flexible and intuitive ways of treating inter-object relations than the purely positional

Table 1. Degree-of-freedom description

Inherits from:	DB-able object
Knows about (implemented as):	Connected DOFs (Local DB) Self in terms of (Local DB) Home Joint (Joint) Coordinate System (Coord Sys) Direction (symbol/index) Eliminated-p (Boolean) Input (number/Local DB) Output (number/Local DB)
Basic skills:	Add Connection Remove Connection Get Connection Modify Connection Induced DOFs Eliminate Self Eliminate if Necessary Input Elimination Output calculation

method of using arrays (matrices). The Local DB type itself can be implemented in any of several ways, such as an association list (this was used explicitly in [37]), a hash table, or similar structure, including an array. The important point to note here is the distinction between the implementation (the

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

how) and the abstract notion of the object itself (the what). What the Local DB does is to provide a basic table tool; how it does it is not important to the rest of the program, and in fact it could be allowed to alter its underlying implementation dynamically depending on its size or other considerations. The basic skills the database must possess include key-driven look-up of values, insertion, deletion and alteration of values, and applying arbitrary functions to each element in the database. A DB-able Object knows how to find/insert itself as a key in a local database, and can be imbued with indices to aid in this process.

Since DOF is built on the type DB-able object, it can be stored and retrieved from Local DBs. This provides the fundamental manner in which a structure is represented for purposes of analysis: a set of degrees of freedom, each of which keeps track of to whom it is connected (key) and the nature of the connection (value) by means of a local database (Connected DOFs). To date, only numerical values have been used to define the interconnections (these numbers are identical to the normal stiffness matrix components), but more general connections are possible as well, such as variables, functions, or rules. In addition to the basic connection information, each DOF also keeps track of the inverse connection information as it becomes available, i.e., how to express itself in terms of the connected DOFs (Self in terms of). For the case of numerical connections, this information corresponds to the components of the inverted stiffness matrix. Other information associated with DOFs includes knowledge of the joint to which it belongs (Home Joint), the coordinate system and direction in which it is defined (Coordinate System, Direction), whether the DOF has been eliminated or not (Eliminated-p, see below), and its input and output values which correspond to loads/load cases and displacements respectively.

The first five skills listed for the DOF type in Table 1 are all associated with building the structural representation, and essentially correspond to assembly. Each DOF knows how to create connections, properly accounting for induced degrees of freedom and transformations due to differing coordinate systems. By viewing coordinate transformations as a summing up of component contributions as opposed to a matrix multiplication, it is possible for one-dimensional objects such as lines to be modeled locally as such, without artificially embedding them in two or three-dimensional spaces until necessary. Thus a single representation can be used efficiently for objects without regard for the ultimate dimensionality of the space in which they will be embedded. This ability to have an object's dimensionality readily apparent is also useful for identification and reasoning about these objects. A coordinate system itself is an object type ('Coordinate Sys'), and can be modeled using nested Local DBs. Again, one can mix two and three-dimensional coordinate systems defined in this manner quite easily.

One of the primary skills of a DOF is the ability to eliminate itself in the mathematical sense; that is, to condense itself out of its own Local DB and out of each of the Local DBs of the DOFs to which it is connected (Eliminate Self). This in essence corresponds to a frontal solution technique that can be carried out at the degree-of-freedom level [38]. (In discussions of using a frontal solution technique, one of the primary issues generally pointed out is that it is difficult to program. The reason for this is that the method is by nature object-oriented: it is in fact very simple to program in an object-oriented environment.) Since such inversion calculations can be carried out largely independently of the global nature of the structure, it is in effect possible to analyze a structure at the same time as it is being developed. This makes it possible to create conveniently a 'WYSIWYG' environment in which a structure can be loaded (interactively and in real time) and observed at any point in its development, without disturbing the flow of the creation process. The additional skills listed for DOFs are all part of the solution process, with the capability for smart recalculation when desired (Eliminate if Necessary).

European Journal of Business & Social Sciences
Available at https://ejbss.org/
ISSN: 2235-767X
Volume 07 Issue 05
May 2019

The next principal object type is the Joint, as presented in Table 2. A Joint inherits its geometric properties and capabilities from the type 'Point', thereby providing basic location and drawing skills. In the present model, the role of a joint is to provide an intermediate interface between its associated degrees-of-freedom (DOFs) and the rest of the structure. Thus, information concerning neighboring elements (Connected Elements), constraints (Constraints), and nodal loads (Loads) is organized and processed here.

The skills the Joint class possesses are fairly straightforward, consisting of basic geometric and structural tasks. In addition to the basic drawing skills inherited from Point, however, it is not difficult

Table 2. Joint description

Inherits from:	Point DB-able Object
Knows about (implemented as):	DOFs (DOFs) Loads (numbers, . . .) Connected Elements (Local DB) Constraints (Boolean, . . .)
Skills:	Incremental Invert Show Self Constrain Relocate Load Displace

Table 3. Element description

Inherits from:	DB-able Object
Knows about (implemented as):	Joints (Ordered List, . . .) Stiffness (Local DB) Containing Element (Complex Element) Coordinate System (Coord Sys)
Skills:	Local to Global Show Self Show Displaced Self Internal Forces Load Modify Copy

to imagine how a joint should be able to do more involved depiction. For example, a constrained joint should look different to an unconstrained joint, and a joint should be able to indicate what its loads are. This is accomplished by defining a separate drawing method for the Joint for carrying out the generic task of Showing. Self. When a particular joint is told to show itself, it will use the appropriate

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

Joint method in place of the Point method. Overriding an inherited method is one example of method combination· a different mechanism is used below in the case of elements.

Modeling the structural elements themselves are objects of type 'Element' as given in Table 3. This is an example of an abstract class or Flavor, i.e. an object type that is not intended to be instantiated on its own. The role of this kind of definition is to provide a compendium of attributes and skills that are shared by all more specialized subtypes of the class. Thus all elements will know which joints they connect (Joints), what their local stiffness is (Stiffness), what structure or substructure they belong to (Containing Element), and so on, regardless of the details of their subtype. Nevertheless, to actually create an instance of an element, it is necessary to provide these details. In terms of skills, elements know how to insert themselves into a structure or substructure (Local to Global), how to depict themselves in displaced and undisplaced configurations (Show Self, Show Displaced Self), how to calculate and display their internal forces (Internal Forces. Show Internal Forces), how to transfer any load applied to them to the relevant DOFs by way of the connected joints (Load), and how to modify their internal properties (Modify).

The final skill listed in Table 3, Copy, introduces the question of what the best means to copy an element is. The most straightforward method would be to reproduce the position-independent information , while recalculating the connection-dependent information. Such a procedure could save the recalculation of element stiffness components. A different approach would be to spawn a dependent copy which directly shares position-independent quantities, rather than reproducing them. Such a copy would respond to a request for a stiffness component by passing along (delegating)

Table 4. Prototype description

Inherits from:	DB-able Object Element
Knows about (implemented as):	Copies (List, . . .)
Skills:	List Copies

the request to its source (prototype), which would provide the appropriate response . A corollary to this approach is that if the prototype is modified, so are all its copies. This could be a particularly useful approach for structural design applications in which there are relatively few unique elements in a given structure. Rather, there is typically a great deal of standardization of beams and columns. For those cases where the second method of copying is desirable, a specialized subtype object can be defined: a Prototype Element (Table 4). This is a special case of Element, with the additional data and skills necessary to handle the dependency relation. Although it is not necessary to define a dependent object type, it might be desirable to do so. In this case it would probably be useful to break the original Element type into two parts: a position-independent part and a connection-dependent part.

An additional useful object type built on Element is Complex Element as described in Table 5. This type of object has both internal and external (public and private) Joints and may be viewed as a substructure composed of sub-elements (Contained Elements). A structure is then composed of one or more substructures, which can themselves be simple or complex elements. This leads naturally to a hierarchical model of the structure , which lends itself quite well to distributed or concurrent processing and rapid updating [39], as well as providing a consistent scheme useful for conceptual

European Journal of Business & Social Sciences
Available at https://ejbss.org/
ISSN: 2235-767X
Volume 07 Issue 05
May 2019

design [40]. The additional skills a Complex Element possesses are used to manage internal bookkeeping.

An example of an element that can actually be instantiated is a Truss Element, as presented in Table 6. As described earlier, such an element can obtain its basic geometric attributes from a type Line. In addition to the structural data and skills inherited due to the fact it is built on Element, a Truss Element also has its own specific properties: area, material stiffness, length, and an axial force. The Truss Element also has its own methods for carrying out the various generic tasks any Element must be able to do, as well as its own specific task of showing axial forces.

Table 5. Complex element description

Inherits from:	Element
Knows about (implemented as):	Contained Elements (List, . . .) Public Joints (List, . . .)
Skills:	Add Element Remove Element Internalize Joint

Table 6. Truss element description

	Line
Inherits from:	Element
Knows about:	Area Young's Modulus Length Axial Force
Skills:	Show Axial Load

Upon adding an instance of a Truss Element object to a structure or substructure (Complex Elements), it is straightforward to incrementally update the corresponding DOF interconnection databases, and to subsequently update the inverse relations as well. This again illustrates the ease with which a structure can be analyzed in parallel with its definition.

The final object type to be considered is another specific element, namely a Beam Element (Table 7). Note that a Beam Element inherits from Truss Element rather than directly from Element. This is because a Beam Element can be considered a further specialization of a Truss Element: it has the same geometry, and it has axial properties along with the additional bending properties. Rather than redefining the axial properties, the Beam type can simply inherit them from Truss Element. Consider the implications of this with respect to computing element stiffness components. In defining methods

for carrying out the element stiffness tasks it is not desirable in this case to have the bending code override the axial code (cf. the earlier example of Joint methods overriding Point methods). Instead, both the existing axial code and the new bending code should be executed whenever a Beam Element object computes its element stiffness components, thereby calculating both the bending and axial components. This simply requires the specification of the appropriate method combination mechanism (after-daemon in this case), and the desired result is obtained. In fact there are many other method combination options available, making it possible to model quite complicated composite objects.

Having defined and implemented the required structural objects, the remainder of the programming chore consists of providing tools to allow the users to create, manipulate, and examine the various types of objects. Although this is a nontrivial task, it is relatively straightforward to do, since the objects can be treated in a natural and abstract manner, and since the interface itself can be built using object-oriented tools. Not only does this allow for fairly simple program construction, but it also provides an

Table 7. Beam element description

Inherits from:	Truss Element
Knows about:	Moment of inertia Bending Moment Shear
Skills:	Show Bending Moment Show Shear Force

extensible and maintainable system. In addition, the underlying consistency between the data structures and the systems being modeled is apparent to the user, since the program construction is based on thinking about how structures and components behave as opposed to thinking about how computers compute.

SUMMARY AND CONCLUDING REMARKS

The purpose of this paper has been to give an overview of how object-oriented programming can provide a useful tool for structural engineering applications. Although the central focus here has been on issues of analysis, the eventual goal is to provide unified software tools applicable at all stages of the creation of a structure. This will involve the consideration of structural issues such as dynamic and nonlinear analysis, reliability, optimization, fabrication simulation and construction sequencing, along with the computational issues of concurrency and distribution of processing and persistence of data structures, which were overviewed briefly from an object-oriented perspective in this paper.

REFERENCES

[1] R. J. Zabilski and H. E. Hall, Presented in 3-D. Civil Engng 59(6), 48-50 (1989).

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

[2] M . Stefik and D. G. Bob row, Object-oriented programming: themes and variations. Al Magazine 6(4), 40-62 (1986).
[3] P. Wegner, Dimensions of object-based language design. OOPSLA '87 Proc. SlGPLAN Notices, Vol. 22, pp. 168182 (1987).
[4] 0. L. Madsen and B. Moller-Pederson, What object-oriented programming may be-and what it does not have to be. £COOP '88-Eur. Conf on Object-oriented Programming Oslo, Norway (Edited by S. Gjessing an K. Nygaard), Lecture Notes in Computer Science, Vol. 322. Springer-Verlag, Berlin (1988).
[5] G. Birtw is tle, 0 . Dahl, B. Myrhaug and K . Nygaard, Simula Begin. Petr ocelli/Charter (1973).
[6] H. Lieberman, Using prototypical objects to implement shared behavior in object-oriented systems. Proc. ACM Conj. on Object-oriented Programming Systems, Languages and Applications, Portland, OR, Special Issue of SIGPLAN Notices, Vol. 21 (1986).
[7] A. Goldberg and D. Robson, Sma//Talk-80: The Language and its Implementation. Addison-Wesley, Reading, MA (1983).
[8] P. D. O'Brien, D. C. Halbert and M. F. Kilian , The Trellis programming environment. OOPSLA '87 Proc. SIGPLAN Notices , Vol. 22, pp. 91-102 (1987).
[9] B. B. Kristensen, 0. L. Madsen, B. Moller-Pedersen and K. Nygaard, The BETA programming langu age. In Research Directions in Object-oriented Languages (Edited by B. Shriver and P. Wegner). MIT Press, Cambridge , MA (1987).
[10] D. G. Bobrow and M. Stefik, The LOOPS manual-a data and object oriented programming system for interlisp. Memo KB-VLSI-81-13, Xerox, PARC (1982).
[11] D. A. Moon, Object-oriented programming with Flavors . OOPSLA '86 Proc., SIGPLAN Notices. Vol. 21, pp. 1-8 (1986).
[12] D. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales and D. Moon, Common Lisp object system specification. ANSI XJ.J.13 Document 88-002R, American National Standards Institute, Washington, DC (1988).
[13] B. Stroustrup , The C+ + Programming Language. Addison-Wesley, Reading, MA (1986).
[14] B. Cox and B. Hunt, Objects, icons, and software ICs.
[15] Byte 11 (8) (1986).
[16] K. Schmucker, Object-oriented Programming for the Macintosh . Hayden (1986).
[17] Symbolics Common Lisp-Language Concepts. Symbolics, Inc., Cambridge, MA (1986).
[18] J. C. Perkowski, Technical trends in the E & C business: the next 10 years. ASCE J. Construct. Engng Management 114, 565-576 (1988).
[19] C. E. Hewitt, P. Bishop and R. Steiger, A universal modular ACTOR formalism for artificial intelligence. Proc. 3rd Int. Joint Conf on Artificial Intelligence (1973).
[20] G. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA (1986).
[21] G. Agha and C. E. Hewitt, Concurrent programming using actors . In Object-oriented Concurrent Programming (Edited by A. Yonezawa and M. Tokoro) , pp. 37-54. MIT Press , Camb ridge, MA (1987).

[22] T. Watanabe and A. Yonezawa , Reflection in an objectoriented concurrent language. OOPSLA '88 Proc. SIGPLAN Notices, Vol. 23, pp. 306-315 (1988).
[23] Y. Yokote and M. Tokoro, Concurrent programming in concurrent SmallT alk. In Object-oriented Concurrent Programming (Edited by A. Yonezawa and M. Tokoro). MIT Press , Cambridge, MA (1987).
[24] J. K. Bennet, Distributed SmallTalk: inheritance and reactiveness in distributed systems. Ph.D. Dissertation, University of Washington, Seattle (1988).
[25] M. Nierstrasz, Active objects in hybrid. OOPSLA
[26] '87 Proc.. SIGPLAN Notices, Vol. 22, pp . 243-253(1987).
[27] A. Corradi and L. Leonardi, How to embed concurrency within an object environment: parallel objects. In Parallel Processing and Applications: Proc. Conf on Parallel Processing and Applications, L'Aquila, Italy, pp. 7984. Elsevier, Amsterdam (1988).
[28] E. Jul, Object mobility in emerald. Ph.D. Dissertation, University of Washington, Seattle (1988).
[29] A. Corana, C. Martini, M. Morando, S. Ridella and
[30] C. Rolando, An LU factorization algorithm with maximum efficiency on parallel-vector computers . In Parallel Processing and Applications: Proc. Int. Conf on Parallel Processing and Applications, L'Aquila, Italy, pp. 197202. Elsevier, Amsterdam (1988).
[31] H. Adeli (Ed.). Parallel and distributed processing in structural engineering . Special Session Proc.: ASCE National Convention, Nashville. ASCE, New York (1988).
[32] D. Maier, Why object-oriented databases can succeed where others have failed. Proc. 1986 Int . Workshop on Object-oriented Database Systems (1986).
[33] W. Kim, N. Ballou, H.-T. Chou, J. F . Garza and D. Woelk, Integrating an object-oriented programming system with a database system. OOPSLA '88 Proc., SIGPLAN Notices , Vol. 23, pp. 142152 (1988).
[34] A. Bjornerstedt and S. Britts, AVANCE: an object management system. OOPSLA '88 Proc. SIGPLAN Notices, Vol. 23, pp. 206-221 (1988).
[35] F. Lochovsky (Ed.), Database Engineering: Special issue on object-oriented systems 8 (4). IEEE Computer Society (1985).
[36] K. Smith and S. B. Zdonik, Intermedia: a case study of the differences between relational and object-oriented database systems. OOPSLA ' 87 Proc., SIGPLAN Notices, Vol. 22, pp. 452-465 (1987).
[37] .D.R. Rehak, Artificial intelligence based techniques for finite element program development. Proc. Symp. on Reliability of Methods for Engineering Analysis, Swansea, U.K. University College of Swansea, University of Wales, pp. 515-532 (1986).
[38] J. W. Baugh and D. R . Rehak , Implementation of a finite element programming system-a declarative ap-
[39] proach. Proc. ASCE Structures Congress, San Francisco (in press).
[40] G. L. Fenves, Object-oriented models for engineering data . Proc. 6th ASCE Conf on Computing in Civil Engineering. Atlanta, GA, (1113 Sept. 1989).
[41] G . R. Miller, A LISP-based, object-oriented approach to structural analysis. Engng Comput. 4, 197-203 (1988).

ISSN: 2235-767X
Volume 07 Issue 05
May 2019

[42] B. M. Irons, A frontal solution program for finite element analysis. Int. J. numer. Meth. 2, 5-32 (1970).
[43] A. Kela and R. Perucchio, Hierarchical incremental finite element analysis through solid modeling. 1988 ASME International Computers in Engineering Conf , San Francisco (31 July4 Aug. I988).
[44] W. R. Spillers and S. L. Newsome, The role of conceptual design within design theory . Proc. 1989 ASCE Water Resources Planning and Management Division Specialty Conf , Sacramento, CA (1989).