Engineering with Computers (1995) 11: 173–184

Engineering with Computers

A Generic Model for Building Design

J. D. Biedermann* and D. E. Grierson**

Civil Engineering Department, University of Waterloo, Ontario, Canada

Abstract. *The design of building structures has benefited considerably through computer automation, but further developments in this field are still required. This paper presents a generic approach to computer automation of the detailed design of building structures. Because of its high level of abstraction, the resulting model is applicable to a wide range of structure types. Other advantages include the use of a consistent data model for software design and implementation, abstract data types for the representation of engineering data, the ability to represent heuristic knowledge, and the ability to evaluate design results in an intelligent manner.*

Keywords. Building structures; Detailed design; Object-oriented; Knowledge representation

1. Introduction

The detailed design of building structures begins with the structure topology and construction material chosen during the preliminary design stage. In performing the detailed design the engineer must determine the member sizes and connection details that are structurally adequate for safety and serviceability, as well as ensuring that the design is economical. Usually this involves an iterative process of member sizing, structural analysis and conformance checking. The design engineer may use available software which encompasses the entire process (e.g. [1]) or which performs only one or more of the subprocesses (e.g. [2]). The resulting design is then critiqued using engineering judgment and heuristics and, if changes are made to the design, the synthesis cycle is usually repeated. Finally, upon convergence, design drawings are produced, often using computer aided drafting tools.

Correspondence and offprint requests to: J. D. Biedermann, School of Engineering, University of Guelph, Guelph, Ontario, Canada, N1G 2W1.

* Assistant Professor. ** Professor.

In the past, structural engineers mainly used computers for only the analysis portion of the detailed design process. Member sizing and conformance checking were done by hand, and previous experience and design heuristics were relied upon heavily. As computers became more prevalent, software became available which helped to reduce the workload of conformance checking and member sizing. Today there are software packages available which provide analysis, member sizing and conformance checking for particular types of structures, e.g. steel or reinforced concrete frames [1, 3]. In addition, there are graphical routines for pre- and post-processing the data, thereby allowing engineers to verify toplogy and loadings, and to view the structural response, such as moment and shear diagrams and deflected shapes [1].

It may appear that computer automation for structural design has reached a state that design engineers are content with. However, this is not the case, as the increasing power and availability of computers only serve to increase the demands made by the engineers relying on them. Researchers and practising engineers alike have identified areas for improvement [4, 5]. Tyson [5], as a practising engineer, suggests that what is needed now is an effective way to integrate the analysis, member sizing, conformance checking and drawing processes into one uniform system. He makes the observation that the current practice of representing engineering data is based on the analysis model rather than the actual geometric representation, and identifies the problems with this approach. The idea of a complete integrated design environment which addresses all stages of structural design and the various parties involved is being addressed by a number of researchers [6–8]. These researchers identify the many benefits of an integrated system for structural design but, owing to the large scope and complexity of the problem, must focus their efforts on small subsystems while developing the theoretical foundation needed for further development. This is admirable work but falls short

of being able to meet the requirements identified by Tyson [5] in the near future.

This paper presents research into a generic approach to the computer automation of the design of structures. The research focuses on detailed design and the development of a generic model which is applicable to all types of building structures. The purpose of the model is to provide: (a) an enhanced capability for simulating the detailed design process; (b) the ability to evaluate design results in an intelligent manner; and (c) a wide range of applicability in the area of detailed design of building structures. The approach addresses many of the concerns and needs identified by engineers for computer automated structural design [4, 5], especially with respect to the representation of design data and the ability to integrate the data representation with the design activities. The paper begins with a definition of a building and provides a model for representing the design data and relationships. The software environment requirements are then discussed. Finally, work on the development and implementation of the model is described and an example is presented.

2. Definition of a Building

Structural engineers involved in building design encounter a wide variety of building types, which vary from one another in their topology, their construction material and in their load resisting systems. Figures 1 to 6 help illustrate these differences. Figure 1 depicts a low-rise residential building which would typically be constructed of light wood framing with a concrete foundation. The sloping roof could be designed as beams or trusses. A low-rise office building is shown in Fig. 2. Office buildings must be designed to provide as much open space as possible as well as an adequate amount of window area. A popular structural system design for this type of building is a rigid frame on the perimeter with well-spaced interior columns, and a construction material of either steel or reinforced concrete. The penthouse which is required for mechanical services adds irregularity to the building topology. The medium rise office building of Fig. 3 has many of the same characteristics and requirements of the low-rise building of Fig. 2. However, the structural system design places more importance on the lateral force resisting system because of the increase in height. Again, topological irregularities are present owing to the penthouse and lower level.

Buildings are not always constructed in one material. Often the structural system is a combination of materials such as reinforced concrete, steel, wood and masonry. Figure 4 depicts a vertically mixed system where the lateral force resisting system of the high-rise building changes from a rigid exterior frame and interior shear wall system made of reinforced concrete to a rigid exterior frame and braced core made of structural steel. Figures 5 and 6 give two further examples of high-rise buildings, each with yet another type of structural system for lateral load resistance; an outrigger system is used for Fig. 5 and a framed tube with core in Fig. 6. The cut-off corner of the building in Fig. 6 makes the structure unsymmetrical and hence the designer must have increased concern for torsion.

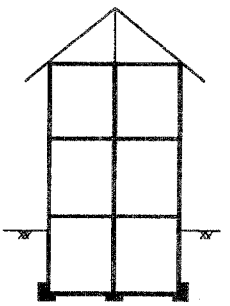

Fig. 1. Low-rise residential building.

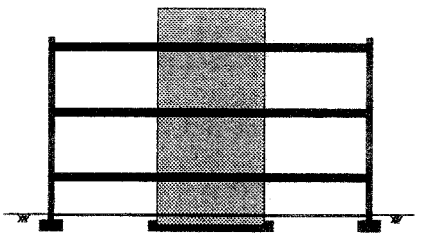

Fig. 2. Low-rise office building.

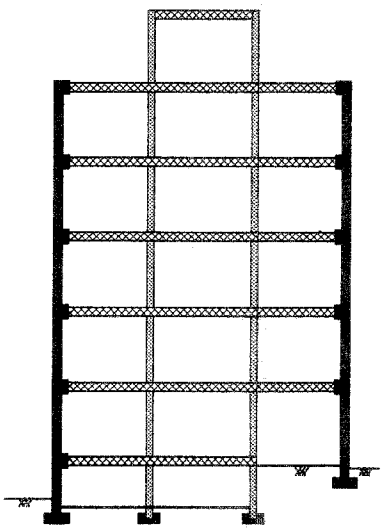

Fig. 3. Medium-rise office building.

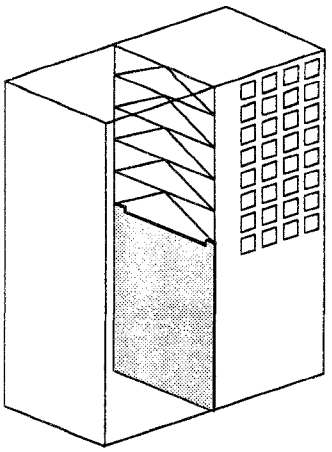

Fig. 4. Vertically mixed system.

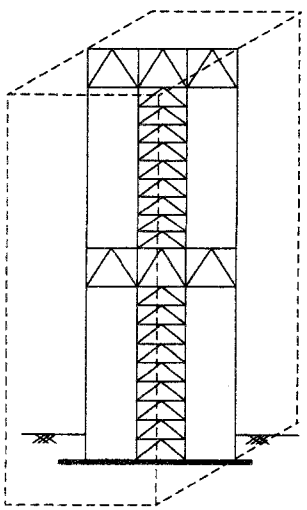

Fig. 5. Outrigger system.

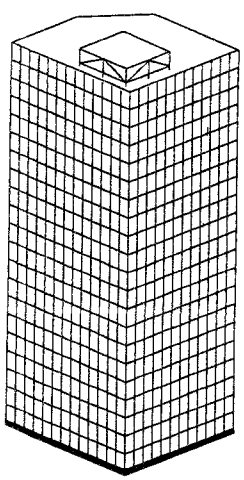

Fig. 6. Framed tube with core.

The examples presented in the foregoing cover a wide range of building design from simple low-rise buildings to complicated high-rise buildings. It is apparent that the structural designer will encounter different construction materials, irregular geometry and many types of structural elements from beams and columns to walls and floors. In addition, the design often proceeds in stages where the engineer initially selects a structural subsystem for detailed design, such as the lateral load resisting system for tall buildings, and then other structural subsystems or elements are progressively designed over time.

The foregoing discussion has noted the many differences that exist among buildings but did little to elaborate on their similarities. A computer-based generic approach to detailed design requires defining the minimum information that the engineer must provide as input data so that any type of building can be designed. In other words, it is necessary to identify the common characteristics of all buildings. At this point we are not concerned with how this information is to be specified but rather with what information is required to be specified. Table 1 defines the general information required for all types of buildings.

All buildings are constructed of one or more materials and must be designed according to a governing design standard. A system of units must be specified (e.g. metric or imperial). As indicated by the square brackets in Table 1, default values can often be used.

The format in Table 1 allows for specifying any number of construction materials, for each of which additional information must be specified as outlined in Table 2.

The simplest topology a building can have is that of a three-dimensional (3D) shape with regular storey heights and bay widths. Many skeletal buildings can be defined in this manner by providing the number of bays in two horizontal directions, the bay widths and the joint types, the number of storeys, the storey height and the joint types in the vertical direction. The regular 3D skeletal shape which results is a good

Table 1. General information.

Identifier
Material(s)
[Design code]
[Design units]
[Force]
[Length]
[Braced or unbraced]
[First order or P – Δ]

Table 2. Material-specific information.

Steel	Reinforced concrete	Composite
[fy]	f'c	[fy]
[fu]	[density]	[fu]
[standard sections]	[γ]	f'c
Prestressed concrete	**Masonry**	**Wood**
f'c	f'm	wet or dry
f'ci	[masonry unit type]	[treatment]
% fpu		[sawn timber or glue-lam]
Aps		
pre- or post-tensioned		
[grouted or not]		

starting point for the generic approach. Initially, it could be assumed that the material of the resulting beams and columns is the first material that the user specified (Table 1), and that there are fixed supports at all column bases.

Modifications and enhancements of the regular 3D topology will most likely be required. Irregular shapes may be created by modifying storey heights or bay widths, or by deleting beams and columns. In addition, joints may be added, deleted or moved, while supports may be added, deleted or changed to a different kind. Bracing members may be added to the structural system. Figure 7 illustrates just a few of the wide variety of bracing patterns which engineers often use. Vertical and horizontal areas may be specified to be solid walls and floors of specified thickness and material, with perhaps openings specified within them. Once all the structural components are defined, the construction material for any of them may then be changed in order to represent mixed or composite construction systems.

The specified loadings that a building must resist are dependent upon the geographic location of the building, the intended occupancy and use of the building, and the governing design standard. It is the engineer's responsibility to ensure that the building is designed for the correct loads though software may exist to help in the specification of these loads. In general these loadings may be defined as a combination of nodal loadings and member loadings with applicable load factors. A nodal loading consists of point loads, and/or moments, and/or prescribed displacements acting at the building joints (nodes). Each nodal loading is given a name (for load combination and identification purposes) and is described by a list of nodes and their load descriptions. Similarly, member loadings representing distributed, and/or point loads, and/or self-stressing effects (e.g. thermal, prestress etc.) acting on building members are described by a name, and a list of members and their load descriptions.

As an example, consider the skeletal steel building illustrated in Fig. 8a. Adopting the generic model approach just described, this building is initially defined as a regular skeletal building with three 10 m bays in the Z direction, two 5 m bays in the X direction and six 5 m high storeys in the Y direction, with all connections rigid and supports fixed, resulting in the topology shown in Fig. 8b. Transforming from Fig. 8b into Fig. 8a is accomplished simply by deleting the beams and columns in the shaded area of Fig. 8b and modifying the width of the third bay in the Z direction to 15 m. Figure 8c shows the nodal and member loadings acting on a typical interior frame. The remaining design information required to define the design problem includes the material and its design properties, the governing design standard, the design units, whether the building is braced or not and whether first- or second-order analysis is required.

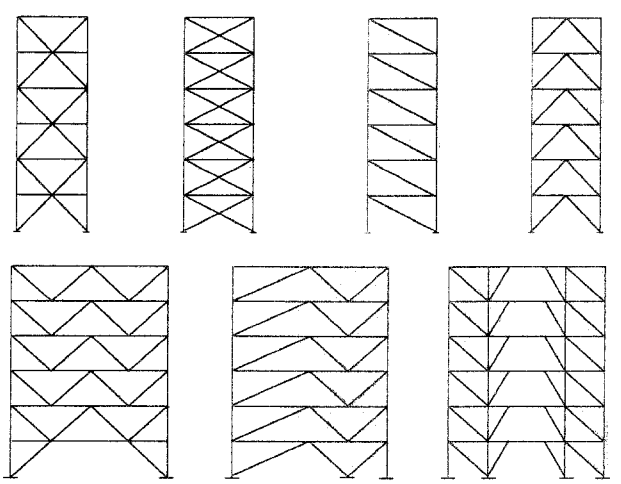

Fig. 7. Bracing systems.

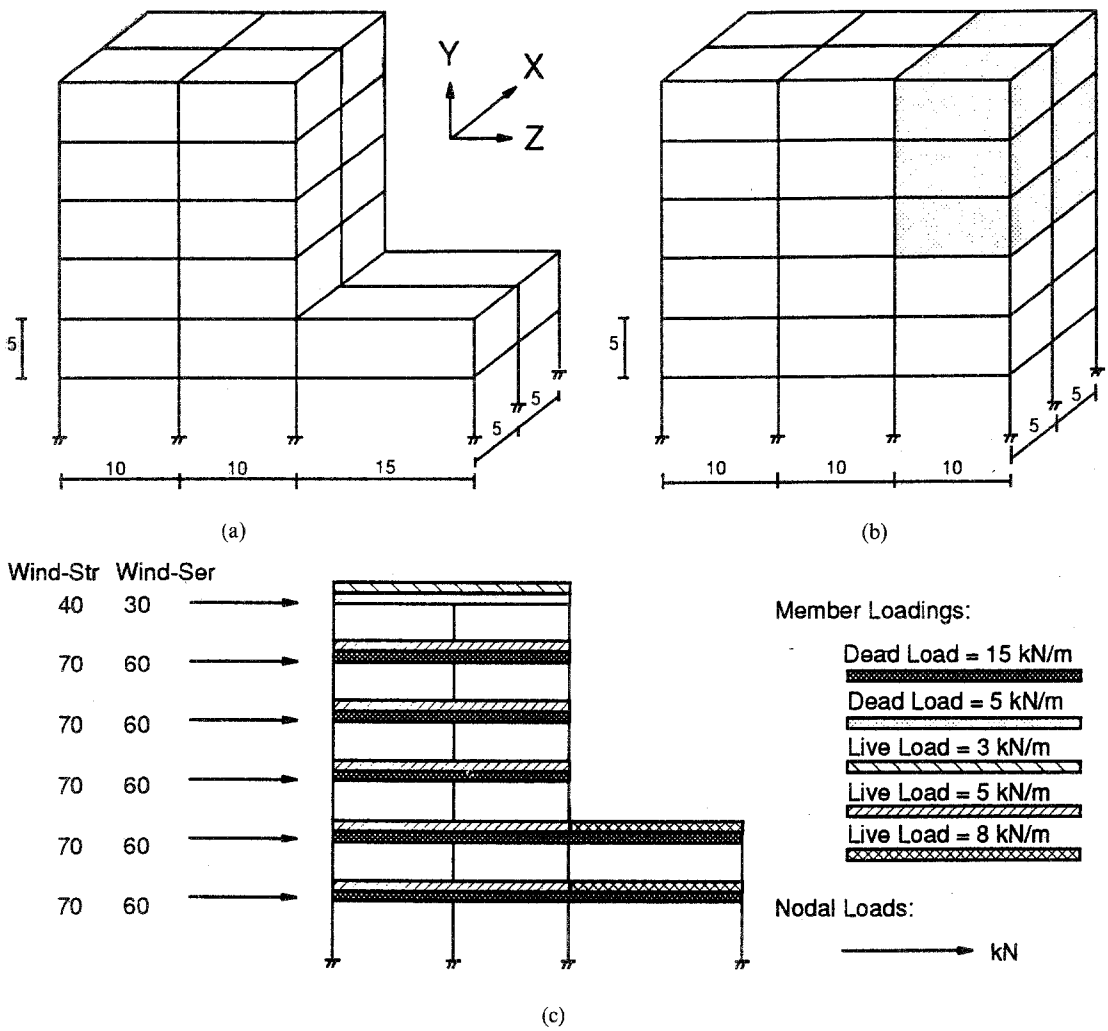

Fig. 8. Rigid frame with mezzanine.

3. Building Data Model

Figure 9 illustrates the data model proposed by this study. The model identifies, at a high level of abstraction, the objects and their interrelations involved in the detailed design of any structure. The objects and relationships shown are derived directly from the modelling of a structural design, which is a necessary part of the detailed design process whether it is done manually or with the aid of a computer.

A ***structure**** is designed using one or more ***materials***. The spatial coordinates of joints, supports and load application points of the ***structure*** are represented as ***nodes***. The topology is further defined by the ***structComps*** (structural Components). Typically in detailed design the ***structComps*** are collected into design ***groups***, where each member of a ***group*** is given the same design properties. The various objects of the model are interrelated as indicated in Fig. 9. For example, each ***structComp*** will have a number of ***nodes*** associated with it. Figure 9 also shows that a ***structure*** is usually designed for a number of different ***loadCases***. Each ***structComp*** and each ***node*** will have a different response for each ***loadCase*** and these are represented as a ***strCmpLCR*** (structural Component Load Case Result) and ***nodalLCR***, respectively.

Apart from simply identifying the objects and their relationships as shown in Fig. 9, it is also necessary to identify the attributes and behaviour for each class of objects. This study adopts an object-oriented approach [9] for the design and implementation of the generic software for detailed design of structures. Object-oriented programming languages allow for the creation of abstract data types (objects) which link attributes and behaviour together, thereby facilitating the representation of engineering data based on actual physical entities [10, 11]. For example, the class

* The following notation is used in the text of this paper: **Class** ***anObject*** *method/message/behaviour* attribute `C++ code`.

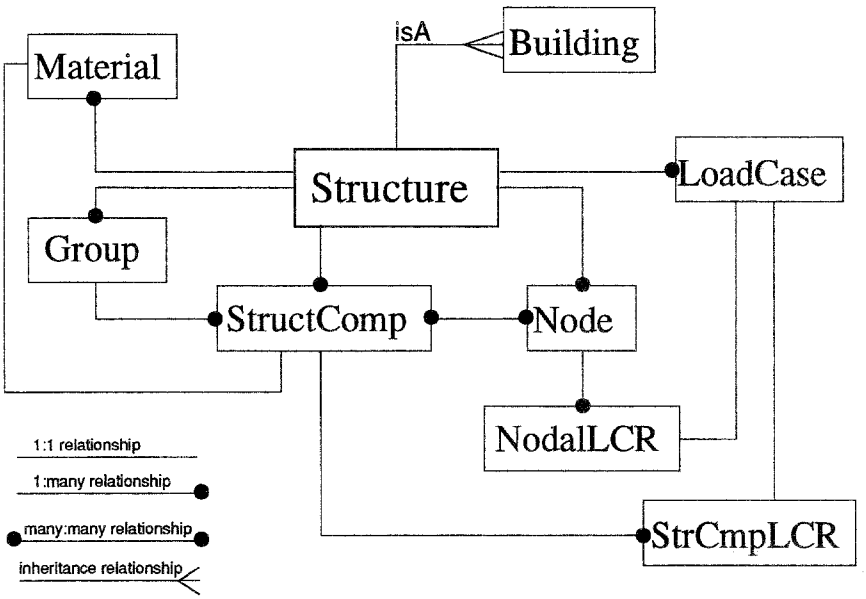

Fig. 9. Generic data model for structural design.

Table 3. Attributes and behavior of class structure.

Attributes	Behaviour
name	setStructuralSubSystem
materials	getLoadCaseResults
nodes	addNodes
members	addMembers
groups	addGroups
loadings	checkNodeConnections
loadCases	design
task	analyze
designTool	verify
strSubSystem	critique
	earlyDesign
	groupMembers
	saveObjects

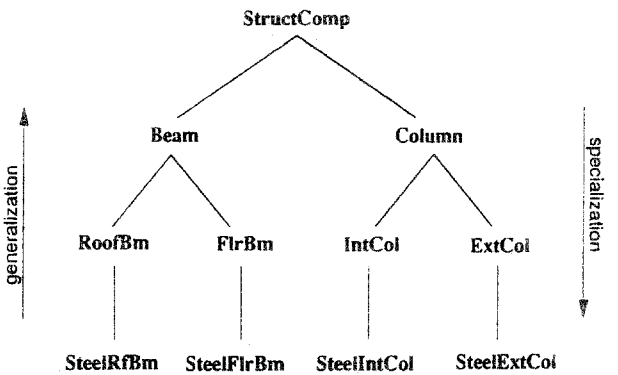

Fig. 10. Structural component class hierarchy.

Structure, which is a generic class applicable to all types of engineering structures, has the (partial) list of attributes and behaviour given in Table 3. Many of the attributes are objects themselves or constitute lists of references to other objects, e.g. nodes is a list of references to all the ***nodes*** that belong to a ***structure***.

The data model of Fig. 9 is generic and developed at a high level of abstraction. In order to apply this model to an actual design problem, some of the objects of the model must be specialized. This is done in a hierarchical manner that allows for inheritance of common attributes and behaviour. For example, Fig. 10 depicts an (incomplete) hierarchy for the class **StructComp** as would be required in the design of a building, and Table 4 provides a brief list of some of the attributes and behaviour that corresponds to this hierarchy. At the top of the hierarchy is the parent class **StructComp** that defines the attributes and behaviour common to all ***structComp*** objects. For example, each ***structComp*** has the attribute nodes which is a list of references to the ***nodes*** to which it is connected. Common behaviour includes storing the results of a loadcase (*putLCR*) and then identifying which ***loadCase*** is most critical and saving this information as the attribute govLCR (governing load case result). A ***structComp*** is then able to use this information when exhibiting some expected behaviour, e.g. returning the worst response (*getWorstResponse*). Further down the hierarchy the class **StreelFlrBm** is identified. At this deepest level of specialization all the attributes and behaviour required for detailed design have been defined.

What is not apparent from the listing given in Table 5 are the heuristics embedded in the subclass

Table 4. Attributes and behaviour of the **StructComp** class hierarchy.

Class **StructComp**	Class **Beam**: subclass to **StructComp**
attributes:	attributes:
name	span
nodes	topBracing
group	bottomBracing
loadCaseResults	behaviour:
govLCR	getSpan
selfDL	
delfLimitRatio	Class **RoofBm**: subclass to **Beam**
behaviour:	inherits from **Beam** and sets values
getName	specializes **Beam** behaviour
getWorstResponse	
getGovLCR	Class **SteelRoofBm**: subclass of **RoofBeam**
saveSelf	behaviour:
putLCR	getMaterialType

behaviour associated with each type of **StructComp.** The need to represent these heuristics is partially responsible for the fact that the **StructComp** hierarchy is specialized by material type. That is, the default values for some of the attributes and many of the design heuristics are material dependent (for example, the span-to-length ratio heuristic defining the maximum allowable deflection for a beam span), and it is computationally advantageous to attach these heuristics directly to a class of objects such as **SteelFlrBm**.

4. Software Environment Requirements

The data model discussed in the previous section was developed through an object-oriented decomposition of the problem where the objects both identify entitites in the real world and encapsulate their attributes and behaviour. Each object is autonomous and collaborates with other objects to achieve the solution to the detailed design problem being modelled. This approach is quite different from the procedural decomposition of a problem that is most often used in developing engineering software. Procedural decomposition involves breaking a large problem into several smaller subproblems. Each subproblem becomes a subprogram in a larger program which operates by systematically calling the subprograms and sharing common data.

Each decomposition technique has advantages and disadvantages [9, 12]. The advantages of the object-oriented decomposition technique are fully realized by implementing the model using an object-oriented programming language (such as C++). For such a programming language, the data model created during the software design stage and the data model actually implemented to solve the problem at hand are essentially the same. Descriptions of the object-oriented paradigm and various object-oriented languages can be found in a number of references [9, 12, 13–17] and are not included in this paper. It is sufficient to say that the object-oriented approach allows for defining classes of objects which encapsulate attributes and behaviour, and that these classes can be defined in a hierarchical manner.

The object-oriented approach has a number of advantages [9, 12]. The objects which result are easily identified and related to by the engineer doing the design [11, 18]. The attributes and behaviour of each class of objects can be defined to meet the many different tasks that may be required during the course of a design, e.g. drafing, cost estimating, analysis etc. [7]. The inheritance feature of the object-oriented approach allows for a layered approach to software design [8], permits attacking the problem at different levels of abstraction [10] and provides for efficient software development through code reuse.

Finally, an efficient and friendly graphical user interface is required in order to initially define the structure and monitor the design process [5]. While the implementation of such an interface is not addressed by the present study, it would proceed directly from the information provided earlier in Section 2 of this paper.

5. Prototype Development and Implementation Details

An initial prototype named GOOD_B (Generic Object-Oriented Detailed design of Buidings) has been developed in the C++ language using the data model discussed in section 3. The detailed design of steel building frameworks is presented for illustration, where the software package SODA [1] is used for the structural analysis and design stages while GOOD_B performs pre- and post-processing of the data. The prototype can be easily modified to use other available design and/or analysis packages (e.g. for wood, reinforced concrete, etc., structures).

As an object-oriented program, GOOD_B is essentially a compilation of the definitions of the classes required for the detailed design of buildings. These are the classes identified in Fig. 9 and those resulting from specialization of any one of those classes, such as are presented in Fig. 10 for structural components. The class definition provides the attributes and methods or behaviour common to all objects of that class and, therefore, can be thought of as a template from which an object can be created (instantiated). In this way

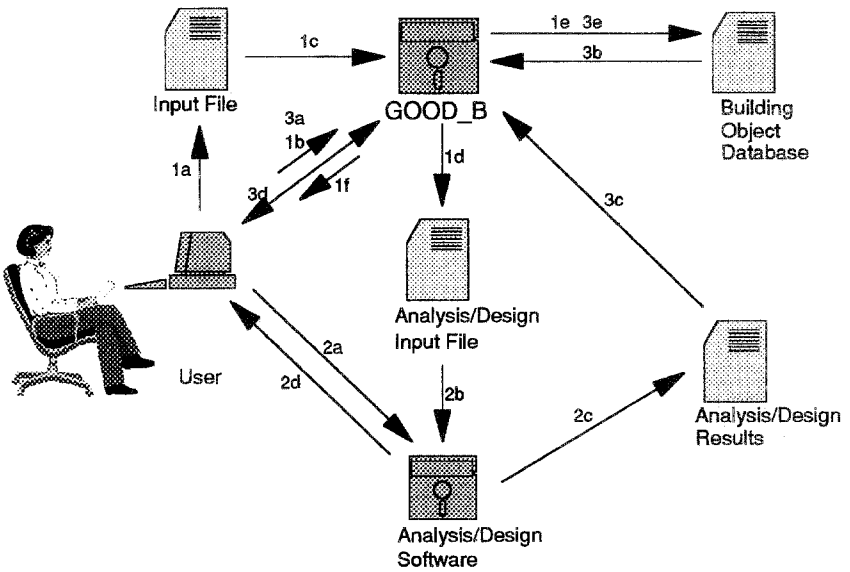

Fig. 11. Conceptual overview of the GOOD_B system.

each object of a class has its own values for the attributes, but shares the methods which define its behaviour. Program control is through message passing, where an object responds to a message by invoking the appropriate class method.

Figure 11 provides a conceptual overview of the GOOD_B system. Initially the user must produce a text file (Fig. 11; 1a) with the required object data which defines the building (as discussed in section 2). An example of such data is given in Table 5. Ideally this data would be created through an intelligent graphical user interface. The user then runs GOOD_B (Fig. 11; 1b), which is implemented in the C++ language and, therefore, has a `main()` function to initiate program execution. The primary purpose of `main()` is to create an object of the class **Building** and then begin the design process. A listing of `main()` follows:

```
main() {
  ifstream infile("bldg.in",ios::in);
  char input[81];
  int line=80;
  infile.getline( input, line, ';' );
  Building theBldg( input, infile );
  theBldg.doTask();
  }
```

The text file "bldg.in", which holds the required input data, is opened and is associated with the **ifstream** (input file stream) object ***infile***. The identifying name of the building is read from the text file and stored in the variable `input`. The object ***theBldg***, of the class **Building**, is then created by calling the class constructor*

Table 5. Example input file.

```
ID: RFwithMezz;
2D; ClsD;
KN; metre;
Braced; First Order; design;
Materials; 1;
steel; 400 450;
Nodes: 24;
:
joint; J4; 35 5 0;
:
FixedSupport; S2; 10 0 0;
:
SteelIntCol: IC5; 2; J13; 1; J16; 1; material:
1;
:
Load cases:
Node loadings: 2;
Wind-Str; 6;
J1; 70 0 0 0 0 0;
J5; 70 0 0 0 0 0;
J9; 70 0 0 0 0 0
J12; 70 0 0 0 0 0;
J15; 70 0 0 0 0 0;
J18; 40 0 0 0 0 0;
:
Load combinations: 4;
#1; 2; DL; 1.25; LL; 1.5;
#2; 3; DL; 1.25; LL; 1.125; Wind-Str; 1.125;
#3; 2; DL; 1.25; Wind-Str; 1.5;
#4; 2; DL; 1; Wind-Ser; 1;
```

* A constructor is the C++ class initialization function. It is used to initialize the data members (attributes) of the class to some values and can also perform other tasks. Reference [19] provides further information on programming in C++.

```
Building theBldg( input, infile );
```

and passing to it the variables (in parentheses) which have the value of ***theBldg***'s name and the input file. A partial listing of the class **Building** constructor follows:

```
Building::Building( char* id, ifstream& inputFile )
{
  getSimpleData( inputFile );
  getMaterialData( inputFile );
  getNodeData( inputFile );
  setTopDispNode();
  setInterStoryNodes();
  getGroupData( inputFile );
  getMemberData( inputFile );
  checkNodeConnections();
  getLoadCases( inputFile );
  if ( task==CRITIQUE )
    getLoadCaseResults();
}
```

The purpose of the constructor is to process the input file (Fig. 11; 1c) and assign the attributes of ***theBldg*** their appropriate values. Many of ***theBldg***'s attributes are objects of other classes, e.g. **Group**, **Node**, **StructComp**, and each of them have constructors which also read the input file in a similar manner. For example, the method *getMemberData* is responsible for creating all the ***structComp***s that are a part of ***theBldg***. The statement

```
case STEELROOFBM:
  members → add( *(new SteelRoofBm( inputFile,
                                    this )) );
```

found in *getMemberData* results in the creation of an object of the class **SteelRoofBm** by calling its constructor and passing the input file to it. The methods *getMaterialData*, *getNodeData* and *getGroupData* behave similarly. In this way all the objects required in the design process are created.

GOOD_B then sends the message *doTask* to ***theBldg*** (the last statement in the listing of `main()`) that instructs it to perform its given task, which is to do detailed design using an available software package. This results in ***theBldg*** collecting its members into ***designGroups*** and then producing the input file used by the analysis/design software specified in the input file (Fig. 11; 1d).

Because of the many activities involved in detailed building design, the objects are required to be persistent beyond the duration of any one executable program in which they may be used (e.g. analysis, conformance checking, drafting etc.). Therefore, all the objects are provided with the ability to read and write their current state to a Building Object Database (BOD). The objects do this once the input file for the analysis/design software is produced (Fig. 11; 1e).

Control now returns to the user (Fig. 11; 1f), who runs the analysis/design package (Fig. 11; 2a). Once complete, the results are written to a file (Fig. 11; 2c) and again the user runs GOOD_B (Fig. 11; 3a). This again creates ***theBldg*** but this time the BOD (Fig. 11; 3b) is used as the input file by the constructor and the task has changed from 'design' to 'critique'. The last statement given in the listing of the **Building** constructor causes ***theBldg*** to read the results stored in the results file (Fig. 11; 3c) produced by the analysis/design software. Again, the message *doTask* is sent to ***theBldg*** but this time the method *critique* is invoked. At this point, ***theBldg*** critiques the resulting design. Interaction with the user (Fig. 11; 3d) takes place as GOOD_B suggests changes to the design. Upon completion of the critique stage, the objects are again written to the BOD (Fig. 11; 3e) and control returns to the user. If the suggested changes to the design are accepted by the user, then GOOD_B produces the necessary input file for the analysis/design software and the user goes through the process again. Otherwise, the design is deemed complete and its description is located in the BOD.

Figure 12 provides a representation of how a design proceeds by illustrating the classes, their methods, some of the message passing that occurs and some of the implementation details of the methods. As stated earlier, ***theBldg*** is created in the `main()` function of GOOD_B and is sent the message to invoke the method *doTask* inherited from the class **Structure**. Initially the value of the attribute task is DESIGN, and, therefore, the method *design* is invoked. The class **Structure** method *design* calls the methods *earlyDesign* and *designSolution* for the purpose of first making some early design decisions and then finding the detailed design solution. One of the early design decisions to make involves the grouping of the structural components into design groups for reasons of economy and ease of construction. This grouping is done by the class **Building** method *groupMembers*. Once all the members have been collected into design groups, each ***group*** is sent the message *pickInitSection* which results in an initial design section being chosen for ***theGroup*** based on heuristics, e.g. all the beams of a similar span and the same material are grouped together. The method *designSolution* is a specialized method defined by the class **Building**. For steel structures, it results in the production of the input file for the SODA program [1] by calling the method *makePopFile*. Before control returns to the user to then run SODA (Fig. 11; 2a), ***theBldg*** changes the value of task to CRITIQUE in anticipation of post-processing of the design results and saves itself to the BOD through the method *saveObjects*. The method

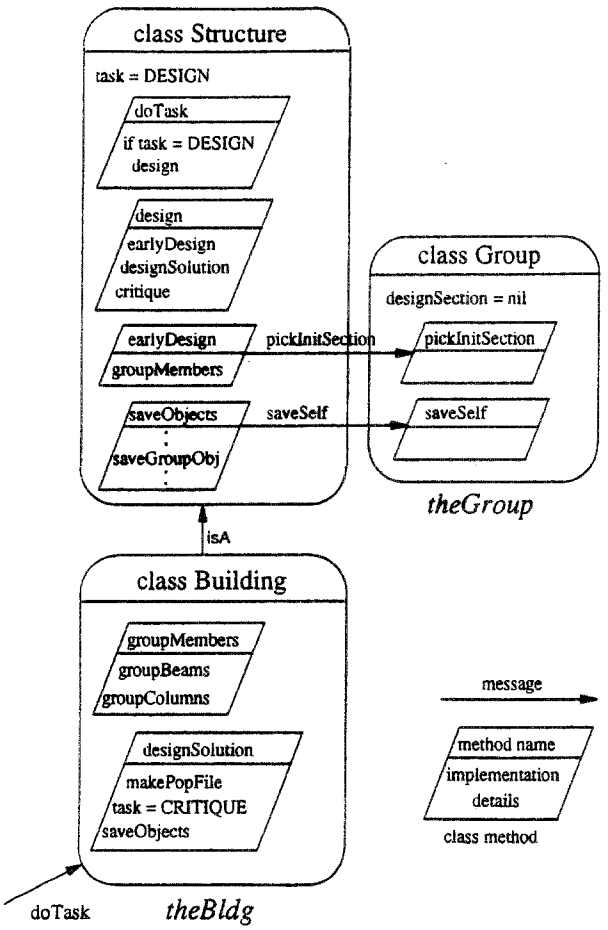

Fig. 12. Object interaction for Task = DESIGN.

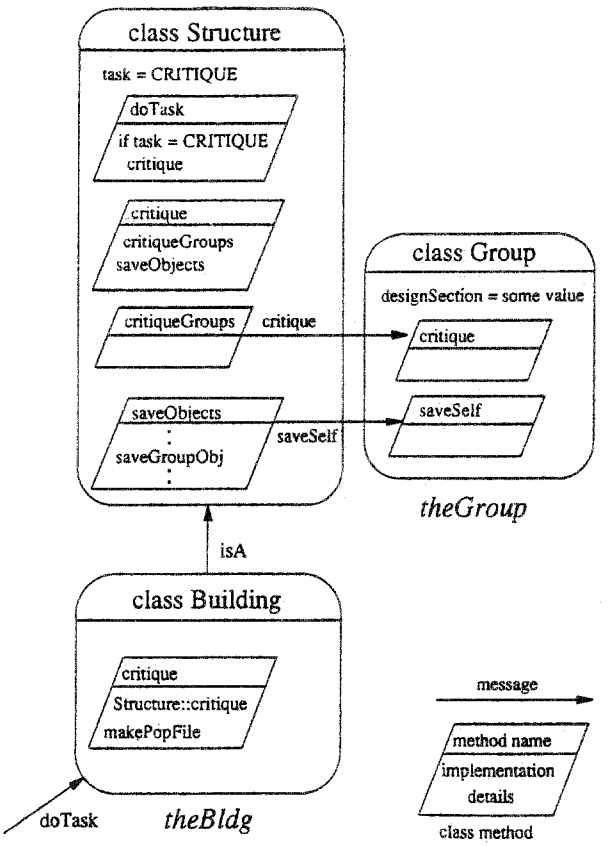

Fig. 13. Object interaction for Task = CRITIQUE.

saveObjects sends a message requesting the objects involved in the design and belonging to ***theBldg*** to save themselves; Fig. 13 illustrates this message passing to an object of the class **Group**.

Once the analysis/design software has completed its task, the user executes GOOD_B again (Fig. 11; 3a). Figure 13 illustrates the sequence of events that result. Once again the message *doTask* is sent to ***theBldg***, but this time task = CRITIQUE. Figure 13 shows that both the **Building** and **Structure** classes define a *critique* method. The **Building** *critique* method is invoked and it in turn invokes the **Structure** *critique* method (*Structure::critique*, as shown in the figure). In this way, after the **Structure** class method *critique* is executed, the class **Building** is able to call the additional method *makePopFile* to create the necessary analysis/design input file. In *Structure::critique*, the method *critiqueGroups* is called which sends the message *critique* to each of the ***groups*** belonging to ***theBldg***. (Each ***group*** then critiques its own design but these details are not shown in Fig. 13). Finally the objects of the design are saved when the method *saveObjects* is invoked and the message *saveSelf* is sent to all the objects involved in the design. Control returns to the class **Building** *critique* method which calls the *makePopFile* method if changes to the design have occurred. With the critiquing of the design complete, program control returns to the user (Fig. 11; 3d).

The GOOD_B system is very flexible and is easily modified and extended because of its object-oriented design. Extending the software to allow for other design and analysis packages (e.g. for wood structures) simply involves adding other methods to create and process the corresponding data files and subclasses of **StructComp**. Incremental software development is easily supported. For example, while the design critiquing presented here has only considered the collection of individual members into common-property design groups, much more can be considered in this regard simply by adding new critiquing methods. For example, each **Group** object could critique the chosen design section by adding the method *improveProfile* to the class **Group** *critique* method. This method would determine if a different cross-section might be more suitable, e.g. a circular

versus a rectangular cross-section for reinforced concrete design.

6. Design Example

Consider the structural steel building framework in Fig. 8. The loadings and analytical model are as shown in Fig. 8c. Other necessary design information is given in Table 6. The first run of the GOOD_B program results in the creation of five ***groups***: all exterior columns, interior columns and roof beams are assembled into the ***groups*** named ***extCol***, ***intCol*** and ***rfBeam*** respectively; the floor beams are grouped using a simple heuristic based on span (e.g. all beams with spans lengths within 20% of one another are grouped together), resulting in the creation of the two beam ***groups*** named ***bmGrp#1*** and ***bmGrp#2***. Figure 14 illustrates this grouping.

The SODA software produces an optimal design based on a minimum weight criterion [1]. (A minimum weight design is often a minimum cost design, which is important to structural designers and their clients.) The design produced by the first SODA run for the initial grouping in Fig. 14 has a weight of 32,915 kg and the design sections given in Table 7. GOOD_B uses the analysis and design results to critique the design groups. The method *splitGroup* of

Table 6. Design information for rigid frame with mezzanine example.

Name: RFwithMezz	Load combinations:
2D braced frame	#1: 1.25*DL + 1.5*LL
Design code: CSA-S16.1-M89	#2: 1.25*DL + 1.125*LL + 1.25*Wind-Str
Units: kN, m	#3: 1.25*DL + 1.5* Wind-Str
first-order analysis	#4: DL + Wind-Ser
Materials:	
steel fy = 400 MPa	
fu = 450 MPa	

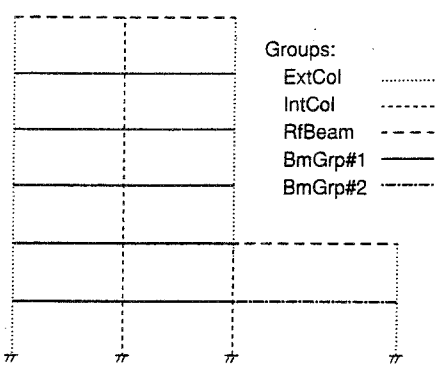

Fig. 14. Initial grouping.

Table 7. SODA run 1: design section results.

rfBeam	W310X158
bmGrp#1	W610X140
bmGrp#2	W310X158
extCol	W460X97
intCol	W310X129

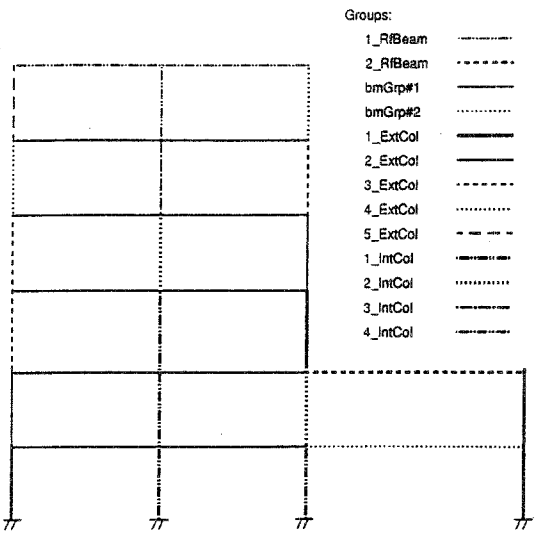

Fig. 15. Grouping after critique stage.

the class **Group** is a simple critiquing routine in which the object ***group*** determines whether it should split itself into possibly more than one ***group*** by inspecting the response ratio of each of the ***structComps*** belonging to it. If appropriate and if the user agrees, new ***groups*** are created. GOOD_B determines that the original ***groups*** should be split for this example. Specifically, ***rfBeam*** is split into two, ***extCol*** into five and ***intCol*** into four new ***groups***. Figure 15 depicts these new ***groups***. A new input file is created and the user runs SODA again.

The second SODA run results in a lighter design representing a 15% reduction in weight to 27,940 kg. The design sections for the different ***groups*** are given in Table 8. Executing GOOD_B results in the ***groups*** being critiqued again. This process continues as long as GOOD_B is able to suggest changes to the design or until the user is satisfied with the results from SODA.

7. Conclusions

The example presented in section 6 was intentionally simplistic with respect to the initial grouping of the

Table 8. SODA run 2: design sec tion results.

1_RfBeam	W610X155
2_RfBeam	W200X52
bmGrp#1	W310X129
bmGrp#2	W610X155
1_ExtCol	W610X155
2_ExtCol	W310X60
3_ExtCol	W310X60
4_ExtCol	W200X36
5_ExtCol	W310X31
1_IntCol	W310X107
2_IntCol	W460X97
3_IntCol	W310X67
4_IntCol	W200X31

structComps and the subsequent splitting of the ***groups*** during the critiquing. The primary intent was to present a working prototype to illustrate the effectiveness of the approach. A much richer set of heuristics can be implemented for grouping ***structComps***, as well as for critiquing the ***groups***. Moreover, other critiquing can include heuristic knowledge pertaining to a wide range of design concerns (e.g. extending columns over two floors, limiting the maximum depth of beams, connectivity of the members etc.).

While presented at but a simple level, this work has shown that a generic approach to the detailed design of building structures is possible. The resulting data model is applicable to a wide range of building structures and can easily be implemented to make use of a variety of different analysis/design software. Other advantages of the approach lie in the consistent data model, which is applicable to the different stages of design (such as preliminary design, detailed design, drawing etc.). Finally, the object-oriented approach allows attributes and behaviour to be linked together in such a way that readily facilitates the implementation of design heuristics, which form an important part of the design process and which are often difficult to account for in other programming environments (such as the traditional procedural approach).

References

1. Grierson, D.E.; Cameron, G.E. (1990) SODA – Structural Optimization Design and Analysis, Release 3.0, User Manual, Waterloo Engineering Software, Waterloo, Canada
2. Wilson, E.L.; Hollings, J.P., Dovey, H.H. (1972) Extended Three-Dimensional Analysis of Building Systems – ETABS, Report No. EERC 72–8, Earthquake Engineering Research Center, University of California, Berkeley, California
3. PCA-FRAME (1992) Proprietary software of Portland Cement Association, © 1992
4. Miller, G.R. (1991) An object-oriented approach to structural analysis and design, Computers and Structures, 40, 1, 75–82
5. Tyson, T.R. (1991) Effective automation for structural design, Journal of Computing in Civil Engineering, 5, 2, 132–140
6. An-Nahish, H.N.; Powell, G.H. (1991). An object-oriented algorithm for automated modelling of frame structures: stiffness modelling, Engineering with Computers, 7, 177–190
7. Abdalla, J.A. (1991) An object-oriented architecture and concept for an integrated structural engineering system, Civil Comp '91, AI and Structural Engineering, Edinburgh, Scotland, 147–155
8. Agbayani, N.; Sriram, D.; Jayachandran, P. (1992) An object oriented framework for steel frame design – implementation issue, Computing Systems in Engineering, 3, 5, 571–587
9. Booch, G. (1991) Object Oriented Design with Application, Benjamin Cummings, Don Mills, Ontario
10. Baugh Jr, J.W.; Rehak, D.R. (1992) Data abstraction in engineering software development, Journal of Computing in Civil Engineering, 6, 3, 282–301
11. Fenves, G.L. (1989) Object-oriented models for engineering data, in Computing in Civil Engineering, Computers in Engineering Practice, Proceedings of the 6th Conference, ASCE, Atlanta, Georgia, 564–571
12. Powell, G.H.; Abdalla, G.A.; Sause, R. (1989) Object-oriented knowledge representations: cute things and caveats, in Computing in Civil Engineering, Computers in Engineering Practice, Proceedings of the 6th Conference, ASCE, Atlanta, Georgia, 1–8
13. Kreutzer, W.; McKenzie, B. (1990) Programming for Artificial Intelligence: Methods, Tools and Applications, Addison-Wesley, Singapore
14. Nelson, M.L. (1991) An object-oriented Tower of Babel, OOPS Messenger: a quarterly publication of the Special Interest Group in Programming Languages, 2, 3, 3–11
15. Thomas, D. (1989) What's in an object?, Byte Magazine, March, 231–240
16. Wegner, P. (1990) Concepts and paradigms of object-oriented programming, OOPS Messenger: a quarterly publication of the Special Interest Group in Programming Languages, 1, 1, 7–87
17. Wegner, P. (1989) Learning the language, Byte Magazine, March, 245–253
18. Fenves, G.L. (1988) Object representations for structural analysis and design, in Computing in Civil Engineering, Microcomputers to Supercomputers, Proceedings of the 5th Conference, Alexandria, Virginia, 502–511
19. Lippman, S.B. (1989) C++ Primer, Addison-Wesley, New York, for AT & T Bell Laboratories [reprinted 1990]