12-1-1993

Object-Oriented Structural Analysis Using Substructures

Jun Song

Richard Sause

Recommended Citation
Song, Jun and Sause, Richard, "Object-Oriented Structural Analysis Using Substructures" (1993). ATLSS Reports. ATLSS report number 93-16:.
http://preserve.lehigh.edu/engr-civil-environmental-atlss-reports/195

ADVANCED TECHNOLOGY FOR
LARGE
STRUCTURAL SYSTEMS

Lehigh University

Object-Oriented Structural Analysis Using Substructures

by

Jun Song
Research Assistant

Richard Sause
Assistant Professor of Civil Engineering

ATLSS Report No. 93 -16

December 1993

ATLSS Engineering Research Center
Lehigh University
117 ATLSS Dr., Imbt Laboratories
Bethlehem, PA 18015-4729
(215) 758-3525

An NSF Sponsored Engineering Research Center

ACKNOWLEDGEMENTS

This research has been supported in part by the Engineering Research Center for Advanced Technology for Large Structural Systems (ATLSS) at Lehigh University under NSF grant ECD-8943455. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of NSF.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

ABSTRACT

The theory for structural analysis using multiple levels of substructures and computer programs based on this theory are well established. Thus, the concepts for these computer programs have changed little in recent years. Recent research has indicated that a new, object-oriented approach to the development of computer programs has many advantages for engineering software. In addition, recent developments in the automated modeling of structural systems for analysis have relied on object-oriented concepts, and the use of multiple levels of substructures. Thus, research has been undertaken to investigate the potential for object-oriented structural analysis using substructures.

This report presents the results of this research. Previous research on the application of object-oriented concepts in structural engineering is reviewed. Substructure analysis theory is reviewed in detail, and extended to provide more general types of degrees-of-freedom in a substructure. Data abstraction concepts and other object-oriented programming concepts are also reviewed. Based on this background, an object-oriented structural analysis model is developed, based on the use of multiple levels of substructures. Object classes to support the model and algorithms to carry out an analysis using these object classes are outlined. Finally, the results of the research are summarized, and future work needed to fully develop and validate the object-oriented structural analysis concepts is discussed.

Chapter 1

Introduction

1.0 Introduction

Structural analysis using substructures is a technique that is several decades old [14]. The use of multiple levels of substructures is somewhat more recent [6]. However, the concepts for computer programs that analyze a structure based on these methods have changed little in recent years. Recent research has indicated that a new, object-oriented approach to the development of computer programs has many advantages for engineering software [5]. In addition, recent developments in the automated modeling of structural systems for analysis have relied heavily on object-oriented concepts and the use of multiple levels of substructures. Thus, research has been undertaken to investigate the potential for object-oriented structural analysis using substructures. This research extends the substructure analysis theory and develops an object-oriented approach to structural analysis based on this theory.

1.1 Objectives

The purpose of this research to develop an object-oriented approach for the analysis for large structural systems, which is based on multiple level substructure analysis theory. This approach was selected because it appears to be the best approach for a structural analysis program that can support an automated structural analysis modeling scheme developed by An-Nashif [1] for use in computer integrated design.

The specific objectives of this research are:

(1) to extend multiple level substructure analysis theory by introducing specified-

displacement/unknown-force degrees-of-freedom within a substructure in addition to the usual specified-force/unknown-displacement degrees-of-freedom;

(2) to develop an object-oriented structural analysis model based on multiple level substructure analysis theory;

(3) to develop object classes that support the analysis model; and

(4) to provide algorithms that use these object classes to carry out object-oriented structural analysis.

1.2 Scope

The research described in this report reviews and extends multiple level substructure analysis theory; presents a structural analysis model for analyzing large structural systems based on multiple levels of substructures; defines ingredients of the model and outlines object classes for these ingredients to support the structural analysis model; and develops algorithms for structural analysis using these object classes.

The research considers only a linear theory of substructure analysis. Nonlinear behavior is not considered. A standard beam element is the only type of element considered. Other element types, such as plate and truss elements are not considered, although these elements could be easily incorporated into the approach that is developed. Kinematic constraints among the degrees-of-freedom within substructures are also not considered. The research develops concepts for the object classes needed for object-oriented structural analysis using substructures, but does not develop these classes in full detail. As a result, the analysis algorithms that are developed are not yet implemented in a computer program.

1.3 Report Organization

The remainder of the report is organized as follows: Chapter 2 presents a brief discussion of the background information related to the research. This chapter reviews previous research on the application of object-oriented concepts in structural engineering. Chapter 3 reviews and extends the substructure analysis theory on the basis of earlier work by Przemiemiecki [14] and Furnike [6]. Chapter 4 reviews the concepts of data abstraction which are essential to the object-oriented approach, and reviews other object-oriented programming concepts. In addition, this chapter briefly reviews the C++ programming language and the NIH (National Institute of Health) Class library which provide some detailed concepts used in Chapter 5 and Chapter 6. Chapter 5 presents an object-oriented structural analysis model, outlines the ingredients of the model, and outlines object classes for these ingredients to support the model. Chapter 6 outlines one more object class, the Global object, which constructs the other objects and conducts the analysis process. Algorithms for carrying out object-oriented structural analysis are also presented in this chapter. Finally, Chapter 7 presents some concluding remarks.

Chapter 2

Background

2.0 Introduction

This chapter reviews previous research related to the report. Three research areas are reviewed: structural analysis using substructure techniques, object-oriented programming, and structural analysis based on the object-oriented approach. Some of this research will be discussed further in the following chapters. In particular, substructure analysis theory will be discussed in Chapter 3 and object-oriented programming will be discussed in Chapter 4.

2.1 Structural Analysis Using Substructures

The substructure technique was first introduced by Przemieniecki in 1963 [14] because the capacity of computers at that time was not sufficient to analyze large structural systems. In the substructure analysis method, each substructure is analyzed separately under the effects of specified external loading, assuming that the displacements at the common degrees of freedom (DOF) with adjacent substructures (i.e. boundary degrees of freedom) are fixed; the boundary displacement DOF between substructures are then relaxed simultaneously and the boundary displacements are determined. The results for the substructures are then obtained by superposition of the effects of the specified external loading and the boundary displacements. Although substructure techniques can be applied to force methods of structural analysis [15], the discussion in this thesis is limited to the displacement method and this method.

The substructure technique introduced by Przemieniecki was extended to multiple level substructure analysis by Furuike [6]. At the first (lowest) level substructures, the boundaries of each substructure are assumed to be fixed. Substructures are analyzed by releasing the interior DOF and keeping the boundary DOF. Substructure stiffness matrices for these substructures are condensed to form substructure boundary stiffness matrices (also known as superelement stiffness matrices). Then the reduced first or lowest level substructures are combined to form second level substructures. In each second level substructure the boundaries of the first level substructures within the second level substructure are relaxed except those that serve as boundaries for the second level substructure. Each of the second level substructures is then reduced by releasing the interior DOF and keeping the boundary DOF, and the reduced second level substructures are combined to form third level substructures. The procedures are repeated until the last (highest) level substructure (i.e. the complete structure) is reached.

A constrained substructure technique, such as that proposed in [8], is based on applying kinematic constraints to the boundaries of individual substructures to reduce the number of independent boundary DOF. The constrained substructure procedure provides additional modeling flexibility with which reasonably accurate solutions can be obtained with a savings in computational effort.

Dodds described the development of substructure techniques, and outlined two advantages of substructuring with respect to data entry and computational efficiency [3]:

(1) Substructuring has proved to be efficient in the analysis of certain structure classes. "It can reduce computer cost by a factor of from 2 to 100."

(2) Since a structure analysis model may assembled by a number of substructures and some of the substructures may have the same geometry, loads, and boundary conditions due to economic and construction considerations, engineers can take advantage of these similarities to "eliminate duplicate input and computational effort".

The second advantage obtained from similarities among substructures is of particular interest since taking advantage of these similarities can greatly reduce the structural analysis modeling effort.

2.2. Object-Oriented Programming for Structural Engineering Problems

Object-oriented programming is a technique used to organize object classes and to utilize their common features in order to reduce programming effort [7]. Object-oriented programs manipulate of objects. An object can represent any engineering concept: a node, an element, or even an algorithm segment. An object consists of the data elements needed to describe the object, together with the set of permissible operations on the data.

Fenves [4] states that "objects are a mechanism for representing data using abstraction, and object-oriented languages are languages for writing programs to manipulate objects." Procedures for developing an object-oriented program are divided into three steps [4]:

(1) Selection of classes: determining what classes of objects are needed to represent the problem and its solution;

(2) Specification of classes: defining the operations on the objects in each class;

(3) Implementation of classes: selecting the instance variables, and programming the functions needed to perform the specified operations.

Forde *et al.* [5] developed a finite element analysis program using object-oriented programming. The object-oriented language, Object NAP, which also contains Pascal and C-routines, was used. An algorithm for banded matrix storage was implemented in the C language, and object-oriented Pascal was used to create the object-oriented portion of the program. Fundamental components of the finite element method, such as node, element, force, and displacement, were represented by various objects.

Keirouz *et al.* [9] created an object-oriented data model for constructed facilities. The model includes: primitive objects, which can not be further divided and represent basic physical entities; domain objects, which represent complex entities and may contain other objects such as primitive objects; and connection objects, each of which describes the relationship between two other objects.

Scholz [18] developed and implemented a typical finite element analysis program on the basis of the Timoshenko beam using the object-oriented programming language C++. The merits of object-oriented programming are shown by providing two examples of the design, implementation, and application of the object classes "Vector" and "Matrix". Scholz indicates that an object-oriented approach to engineering problems leads to easier validation and maintenance of programs than procedural languages, and that the implementation of an object-oriented program requires less time and produces smaller programs compared with conventional programming techniques.

2.3. Object-Oriented Structural Analysis

Some earlier work has introduced the object-oriented approach into structural analysis. Miller [13] presented a general algorithm for solution of a system of linear equations represented by object-oriented data structures. This matrix-free solution algorithm computes an equivalent to the inverted stiffness matrix for an arbitrary assemblage of linearly related DOF. In addition, Miller described the use of object-oriented data structures in the context of structural analysis. The idea is to view a structure as an assemblage of objects, both physical and mathematical in nature: to define the data associated with each object, and to construct the necessary operations that each object should have. Several fundamental structural analysis object types are defined: DOF objects, node objects, and element objects. A prototype program was developed in the LISP language to implement these objects and the corresponding analysis algorithm.

Miller [12] further described the analysis program from reference [13] with several substantial changes, including an object type "Local DB", which is a simple database with the capability to store simple key/value databases. The database provides key-driven look-up of values, and insertion, deletion, and alteration of values. "DB-able" objects are defined on the basis of "Local DB" as objects that can be stored in the "Local DB" object. DOF objects are the base objects of the model. DOF objects are derived from "DB-able" objects so that DOF objects can be stored in and managed by the "Local DB" object. Each Joint object provides an interface between its associated DOF and the rest of the structure. It contains information about related elements, constraints, and loads. The model has two types of element objects: simple element objects, which are basic beam elements, and complex element objects, which are substructures.

An-Nashif [1] suggested that the modeling process, as well as the numerical computations must be automated in order to implement automated structural analysis. This work [1] assumes that a central database, based on a component-connection design model, describes the structure as an assemblage of components such as beams, columns, substructures, and connections. This database contains a description of the structure geometry, properties, and loading in terms of the physical objects (beams, columns, slabs, etc.) which comprise the structure. A modeler program extracts an analysis model from the central database by transforming the component-connection model into a node-element model for structural analysis. The analysis model produced by the modeler is significantly different from conventional structural analysis models since it makes extensive use of substructuring and kinematic constraints.

Although the work by An-Nashif [1] is limited to the modeling of frame structures made of beam-column type components connected at discrete points, the principles are applicable to more complex structures. The programming of the component-connection modeling algorithm is addressed using object-oriented concepts, which maintains a clear mapping between the component-connection design model and the node-element analysis model.

Chapter 3

Substructure Analysis Theory

3.0. Introduction

In this chapter, substructure analysis theory is presented. A matrix method of linear structural analysis is described for calculating displacements and reaction forces for a structure composed of a number of substructures. Several decades ago, the necessity for dividing a structure into substructures was identified because, in some cases, different analysis methods were needed for different parts of a structure, or more often, because the capacity of the available computers was not adequate for analysis of the complete structure [6]. In this report, the substructure method is utilized to develop concepts for a structural analysis program that can address the needs of the automated modeling approach described by An-Nashif [1], and as an effective way to introduce object-oriented concepts into the structural analysis problem.

In the substructure method of analysis, the complete structure is partitioned into a number of substructures [2], whose boundaries are arbitrary but not overlapping, including the possibility of one substructure being inside another. It should be emphasized that the partitioning affects the subsequent matrix operations and the substructure boundaries must be selected carefully. The individual substructures are combined by matching the corresponding nodes on the boundaries.

The structure is decomposed into substructures based on geometry and on modeling convenience. A substructure can be further decomposed into other substructures. Thus the analysis model can include multiple levels of substructures, and a hierarchy of substructures can be constructed (Fig. 3.1). A substructure that is divided into smaller substructures is called a *higher level substructure. Lower level substructures* are parts of a higher level substructure. Each entity in the hierarchy is referred to as a substructure. Expanding this concept, the complete structure is the *highest level substructure,* and the simplest substructures at the bottom of the hierarchy are the *lowest level substructures* (Fig. 3.1).

The lowest level substructures are composed of nodes and elements. Higher level substructures consist of nodes, elements, and *superelements.* The nodes and elements have the same characteristics as those in the lowest level substructures. Superelements are created from lower level substructures, as discussed later on, to be included in higher level substructures.

In the substructure method, each substructure is analyzed separately under the effects of specified external loading, assuming that displacements at the common degrees of freedom (DOF) with adjacent substructures (i.e. boundary DOF) are fixed; the boundary displacement DOF between substructures are then relaxed simultaneously (during the analysis of a higher level substructure) and the boundary displacements are determined. The results for the substructures are then obtained by superposition of the effects of the specified external loading and the boundary displacements [14].

In summary, the substructure method of analysis consists of:

(1) a separate analysis of each substructure with the displacement DOF on the boundaries with other substructures completely restrained;

(2) a relaxation of the boundary DOF to ensure equilibrium of the boundary nodes and to calculate the substructure boundary displacements;

(3) a separate analysis of each substructure under the known boundary displacements and superposition of these results with the results of (1).

The substructure method is discussed in the following three sections. Section 3.1 discusses the analysis of a single substructure. Section 3.2 discusses the creation of a superelement from a substructure, for inclusion in a higher level substructure. Section 3.3 discusses the analysis of a higher level substructure which includes superelements.

3.1 Analysis of Single Substructure

The complete set of equilibrium equations for a substructure may be written in matrix form as:

$$F_0+KU=F \qquad \textbf{Eq. 3.1}$$

where:

U is a vector of displacements (DOF);

F is a vector of external forces acting on the DOF of the substructure;

K is the substructure stiffness matrix;

$\mathbf{F_0}$ is a vector of initial forces required to equilibrate forces applied within elements, to restrain element displacements arising from forces applied within elements, and to restrain element displacements due to temperature, shrinkage, creep, and other strains.

It should be noted that **K** and $\mathbf{F_0}$ in Eq. 3.1 are assembled from multiple sets of equilibrium equations that relate element forces and element displacements (and superelement boundary forces and boundary displacements). The element forces and element displacements of the single element shown in Fig. 3.2 in an element local coordinate system are related as follows [17]:

$$P^{(e)}=K_p^{(e)}p^{(e)}+P_0^{(e)} \qquad \textbf{Eq. 3.2}$$

where:

(e) denotes a single element;

$\mathbf{p^{(e)}}$ is a vector of displacements of the element ends;

$\mathbf{P^{(e)}}$ is a vector of total forces acting on the element ends including both initial forces and the effects of displacements of the ends;

$\mathbf{K}_p^{(e)}$ is the element stiffness matrix with respect to the element coordinate system;

$\mathbf{P}_0^{(e)}$ is a vector of initial forces required to equilibrate forces applied within the element, to restrain element displacements arising from forces applied within the element, and to restrain element displacements due to temperature, shrinkage, creep, and other strains (i.e. element initial forces).

Before assembling each element into the substructure, a rotation of coordinate directions is carried out by coordinate transformation [17]. After transformation, as Fig. 3.2 shows, the relationship between element forces and displacements in the substructure local coordinate system is:

$$Q^{(e)}=K_q^{(e)}q^{(e)}+Q_0^{(e)} \qquad \textbf{Eq. 3.3}$$

where:

$\mathbf{q}^{(e)}$ is a vector of displacements of the element ends, which is equivalent to $\mathbf{p}^{(e)}$;

$\mathbf{Q}^{(e)}$ is a vector of total forces acting on the element ends, which is equivalent to $\mathbf{P}^{(e)}$;

$\mathbf{K}_q^{(e)}$ is the element stiffness matrix with respect to the substructure coordinate system;

$\mathbf{Q}_0^{(e)}$ is a vector of element initial forces, which is equivalent to $\mathbf{P}_0^{(e)}$.

Then, $\mathbf{K}_q^{(e)}$ and $\mathbf{Q}_0^{(e)}$ can be assembled into **K**, and $\mathbf{F}_0$ respectively.

In the following discussion it is assumed that the substructure displacement DOF are divided into boundary displacement DOF and interior displacement DOF. Usually, all the displacements at a boundary node are considered to be boundary displacement DOF (Fig. 3.3), although it is possible for some of these DOF to be considered to be interior displacement DOF. The displacement DOF are divided into three types: **r**, which are unknown displacements of the interior DOF of the substructure; **y**, which are known displacements of the interior DOF; and **p**, which are unknown displacements of the boundary DOF of the substructure, as shown in Fig. 3.3. The force DOF are also divided into three types: **R**, which are known applied external forces on interior DOF corresponding to **r**; **Y**, which are unknown external reaction forces on interior DOF corresponding to **y**; **P**, which are unknown external reactions on boundary DOF corresponding to **p**, and $\mathbf{P}_e$, which are known applied external forces on the boundary DOF corresponding to **p**. $\mathbf{P}_e$ will be neglected in this chapter, but could easily be added to the equations below.

Based on these definitions, Eq. 3.1 may be written in partitioned form as:

$$\begin{Bmatrix} R_0 \\ Y_0 \\ P_0 \end{Bmatrix} + \begin{bmatrix} K_{rr} & K_{ry} & K_{rp} \\ K_{yr} & K_{yy} & K_{yp} \\ K_{pr} & K_{py} & K_{pp} \end{bmatrix} \begin{Bmatrix} r \\ y \\ p \end{Bmatrix} = \begin{Bmatrix} R \\ Y \\ P \end{Bmatrix} \qquad \textbf{Eq. 3.4}$$

where:

r is a vector of unknown interior displacements;

y is a vector of known interior displacements;

p is a vector of boundary displacements assumed to be fixed initially and released in a higher level substructure analysis;

R is a vector of known applied external forces (corresponding to **r**);

Y is a vector of unknown external reaction forces (corresponding to **y**);

P is a vector of unknown reaction forces (corresponding to **p**);

$\mathbf{R}_0$, $\mathbf{P}_0$, $\mathbf{Y}_0$ are vectors of initial forces which are subvectors of $\mathbf{F}_0$ (corresponding to **r**, **p**, and **y** respectively).

It will now be assumed that the displacements of the substructure may be determined from the superposition of two vectors [6]:

$$U=U^{(\alpha)}+U^{(\beta)} \qquad \textbf{Eq. 3.5}$$

where:

$\mathbf{U}^{(\alpha)}$ are the displacements with $\mathbf{p}$=0;

$\mathbf{U}^{(\beta)}$ are the displacements $\mathbf{U}$ corresponding to the actual boundary displacements $\mathbf{p}$.

Eq. 3.5 may also be written as follows:

$$U^{(\alpha)} = \begin{Bmatrix} r^{(\alpha)} \\ y^{(\alpha)} \\ p^{(\alpha)} \end{Bmatrix} = \begin{Bmatrix} r^{(\alpha)} \\ y \\ 0 \end{Bmatrix} \qquad \textbf{Eq. 3.6.a}$$

$$U^{(\beta)} = \begin{Bmatrix} r^{(\beta)} \\ y^{(\beta)} \\ p^{(\beta)} \end{Bmatrix} = \begin{Bmatrix} r^{(\beta)} \\ 0 \\ p^{(\beta)} \end{Bmatrix} \qquad \textbf{Eq. 3.6.b}$$

$$U = \begin{Bmatrix} r^{(\alpha)} \\ y^{(\alpha)} \\ p^{(\alpha)} \end{Bmatrix}_{\substack{\textit{boundary} \\ \textit{displacements} \\ \textit{fixed}}} + \begin{Bmatrix} r^{(\beta)} \\ y^{(\beta)} \\ p^{(\beta)} \end{Bmatrix}_{\substack{\textit{boundary} \\ \textit{displacements} \\ \textit{relaxed}}} = \begin{Bmatrix} r^{(\alpha)} \\ y \\ 0 \end{Bmatrix} + \begin{Bmatrix} r^{(\beta)} \\ 0 \\ p^{(\beta)} \end{Bmatrix} \qquad \textbf{Eq. 3.6.c}$$

where, the first term on the right-hand side of Eq. 3.6.c denotes the displacements when the boundary displacements **p** are fixed, and the second term on the right-hand side denotes the displacement corrections when the boundary displacements **p** are relaxed.

Corresponding to the displacements $\mathbf{U}^{(\alpha)}$ and $\mathbf{U}^{(\beta)}$, the forces **F** can be separated into two parts:

$$F = F^{(\alpha)} + F^{(\beta)} \qquad \textbf{Eq. 3.7}$$

Eq. 3.7 may be written as follows:

$$F^{(\alpha)} = \begin{Bmatrix} R^{(\alpha)} \\ Y^{(\alpha)} \\ P^{(\alpha)} \end{Bmatrix} = \begin{Bmatrix} R \\ Y^{(\alpha)} \\ P^{(\alpha)} \end{Bmatrix} \qquad \textbf{Eq. 3.8.a}$$

$$F^{(\beta)} = \begin{Bmatrix} R^{(\beta)} \\ Y^{(\beta)} \\ P^{(\beta)} \end{Bmatrix} = \begin{Bmatrix} 0 \\ Y^{(\beta)} \\ P^{(\beta)} \end{Bmatrix} \qquad \textbf{Eq. 3.8.b}$$

$$F = \begin{Bmatrix} R^{(\alpha)} \\ Y^{(\alpha)} \\ P^{(\alpha)} \end{Bmatrix}_{\substack{\textit{boundary} \\ \textit{displacements} \\ \textit{fixed}}} + \begin{Bmatrix} R^{(\beta)} \\ Y^{(\beta)} \\ P^{(\beta)} \end{Bmatrix}_{\substack{\textit{boundary} \\ \textit{displacements} \\ \textit{relaxed}}} = \begin{Bmatrix} R \\ Y^{(\alpha)} \\ P^{(\alpha)} \end{Bmatrix} + \begin{Bmatrix} 0 \\ Y^{(\beta)} \\ P^{(\beta)} \end{Bmatrix} \qquad \textbf{Eq. 3.8.c}$$

where:

$\mathbf{R}^{(\alpha)}$ is a vector of known applied external forces equal to $\mathbf{R}$;

$\mathbf{Y}^{(\alpha)}$ is a vector of unknown external reaction forces on the interior DOF, assuming the boundary displacements $\mathbf{p}^{(\alpha)}$ are fixed;

$\mathbf{Y}^{(\beta)}$ is a vector of unknown external reaction forces on the interior DOF due to the actual boundary displacements $\mathbf{p}^{(\beta)}$;

$\mathbf{P}^{(\alpha)}$ is a vector of unknown external reaction forces on the boundary DOF required to equilibrate forces applied within the substructure and to keep the boundary displacements $\mathbf{p}^{(\alpha)}$ fixed;

$\mathbf{P}^{(\beta)}$ is a vector of unknown external reaction forces on the boundary DOF due to the actual boundary displacements $\mathbf{p}^{(\beta)}$.

Based on these definitions, the analysis of a single substructure consists of two parts [6]:

(1) the α case, in which the boundary displacements are fixed;

(2) the β case, in which the actual boundary displacements are considered.

It should be noted that the initial forces $\mathbf{F}_0$ are included in the α case.

For the α case, the substructure boundary displacements are fixed. The first row of Eq. 3.4 is:

$$R_0 + K_{rr} r^{(\alpha)} + K_{ry} y^{(\alpha)} + K_{rp} p^{(\alpha)} = R^{(\alpha)}$$

The boundary displacements are assumed fixed ($\mathbf{p}^{(\alpha)}$=0) and $\mathbf{y}^{(\alpha)}$=$\mathbf{y}$ is known, so the unknown displacements $\mathbf{r}^{(\alpha)}$ can be determined:

$$r^{(\alpha)} = K_{rr}^{-1} [R - R_0 - K_{ry} y] \qquad \textbf{Eq. 3.9}$$

The second row of Eq. 3.4 is:

$$Y_0 + K_{yr} r^{(\alpha)} + K_{yy} y^{(\alpha)} + K_{yp} p^{(\alpha)} = Y^{(\alpha)}$$

Substituting $\mathbf{r}^{(\alpha)}$ into this equation with $\mathbf{p}^{(\alpha)}$=0 and $\mathbf{y}^{(\alpha)}$=$\mathbf{y}$, the external reaction forces on the interior DOF are:

$$Y^{(\alpha)} = Y_0 + K_{yr} K_{rr}^{-1} [R - R_0 - K_{ry} y] + K_{yy} y \qquad \textbf{Eq. 3.10}$$

The third row of Eq. 3.4 is:

$$P_0 + K_{pr} r^{(\alpha)} + K_{py} y^{(\alpha)} + K_{pp} p^{(\alpha)} = P^{(\alpha)}$$

Substituting $\mathbf{r}^{(\alpha)}$ into this equation with $\mathbf{p}^{(\alpha)}$=0 and $\mathbf{y}^{(\alpha)}$=$\mathbf{y}$, the external reaction forces on the boundary DOF are:

$$P^{(\alpha)} = P_0 + K_{pr}\ K_{rr}^{(-1)}\ [R - R_0 - K_{ry}\ y] + K_{py}\ y \qquad \textbf{Eq. 3.11}$$

It should be noted that $\mathbf{P}^{(\alpha)}$ are the external boundary reactions on the boundary DOF necessary to equilibrate forces applied within the substructure and to maintain $\mathbf{p}^{(\alpha)}$=0 (i.e. to keep the boundary displacements fixed).

After the substructure boundary displacements have been determined, the results of the β case can be determined from Eq. 3.4.

The first row of Eq. 3.4 is:

$$K_{rr} r^{(\beta)} + K_{ry} y^{(\beta)} + K_{rp} p^{(\beta)} = R^{(\beta)}$$

With $\mathbf{R}^{(\beta)}=0$, $\mathbf{y}^{(\beta)}=0$ and $\mathbf{p}^{(\beta)}$ known, the unknown displacements $\mathbf{r}^{(\beta)}$ can be determined:

$$r^{(\beta)} = -K_{rr}^{(-1)} [K_{rp} p^{(\beta)}] \qquad \textbf{Eq. 3.12}$$

The second row of Eq. 3.4 is:

$$K_{yr} r^{(\beta)} + K_{yy} y^{(\beta)} + K_{yp} p^{(\beta)} = Y^{(\beta)}$$

substituting $\mathbf{r}^{(\beta)}$ into this equation with $\mathbf{y}^{(\beta)}=0$ and $\mathbf{p}^{(\beta)}$ known, the external reaction forces on the interior DOF are:

$$Y^{(\beta)} = [-K_{yr} K_{rr}^{(-1)} K_{rp} + K_{yp}] \, p^{(\beta)} \qquad \textbf{Eq. 3.13}$$

For the β case, the stiffness relationship between the reaction forces and the displacements on the boundary DOF is:

$$K_p^* p^{(\beta)} = P^{(\beta)} \qquad \textbf{Eq. 3.14}$$

where:

$$K_p^* = [\, K_{pp} - K_{pr} K_{rr}^{(-1)} K_{rp} \,] \qquad \textbf{Eq. 3.15}$$

$\mathbf{K}_p$* is known as the *condensed stiffness matrix*, the *boundary stiffness matrix*, or the *superelement stiffness matrix*, depending on the context.

The total external force on the boundary DOF is:

$$P = P^{(\beta)} + P^{(\alpha)}$$

which can be expressed as:

$$P = K_p^* p^{(\beta)} + P^{(\alpha)} \qquad \textbf{Eq. 3.16}$$

When the boundary displacements are fixed ($\mathbf{p}^{(\alpha)}=0$), each substructure is completely isolated from the others so that the interior displacements $\mathbf{r}^{(\alpha)}$ and reactions $\mathbf{Y}^{(\alpha)}$, and the boundary reactions $\mathbf{P}^{(\alpha)}$ can be calculated for each substructure separately using Eq. 3.9, 3.10, and Eq. 3.11. The determination of boundary displacements $\mathbf{p}^{(\beta)}$ involves the next higher level substructure. The boundary stiffness matrix $\mathbf{K}_p{}^*$ and the reaction forces on the boundary DOF $\mathbf{P}^{(\alpha)}$ are calculated for each substructure independently.

When the boundary displacements $\mathbf{p}^{(\beta)}$ are known, then the interior displacements $\mathbf{r}^{(\beta)}$ and the reactions $\mathbf{Y}^{(\beta)}$, and the boundary reactions $\mathbf{P}^{(\beta)}$ can be calculated for each substructure using Eq. 3.12, 3.13, and 3.14.

3.2 From Substructure to Superelement

Eq. 3.1 and 3.4 can be written for the rth substructure as follows:

$$(F_0)_{(r)} + (K)_{(r)}(U)_{(r)} = (F)_{(r)} \qquad \textbf{Eq. 3.17}$$

and:

$$\begin{Bmatrix} R_0 \\ Y_0 \\ P_0 \end{Bmatrix}_{(r)} + \begin{bmatrix} K_{rr} & K_{ry} & K_{rp} \\ K_{yr} & K_{yy} & K_{yp} \\ K_{pr} & K_{py} & K_{pp} \end{bmatrix}_{(r)} \begin{Bmatrix} r \\ y \\ p \end{Bmatrix}_{(r)} = \begin{Bmatrix} R \\ Y \\ P \end{Bmatrix}_{(r)} \qquad \textbf{Eq. 3.18}$$

where **(r)** denotes the rth substructure.

It should be noted that in selecting the known interior displacement DOF **y** and boundary displacement DOF **p** of each substructure, these known displacements must be sufficient to restrain rigid body displacement of a substructure. That is, the rigid body modes should not be included in the displacement DOF **r** so that $\mathbf{K}_{rr}$ can be inverted. A typical substructure with fixed boundary displacements is shown in Fig. 3.3.

The external reaction forces on the boundary DOF required to equilibrate forces applied within the substructure and to restrain the boundary displacements (i.e. with **p**=0) can be determined from Eq. 3.11:

$$(P^{(\alpha)})_{(r)} = P_0 + (K_{pr} K_{rr}^{(-1)} [R - R_0 - K_{ry} y] + K_{yp} y)_{(r)} \qquad \textbf{Eq. 3.19}$$

The substructure boundary stiffness matrix associated with the displacement $\mathbf{p}_{(r)}$ from Eq. 3.15 is:

$$(K_p^{*})_{(r)} = (K_{pp} - K_{pr} K_{rr}^{(-1)} K_{rp})_{(r)} \qquad \textbf{Eq. 3.20}$$

This stiffness will be used subsequently in assembling the stiffness matrix for the next higher level substructure.

The results of Eq. 3.19 and Eq. 3.20 define a *superelement*, shown in Fig. 3.4. By comparing Fig. 3.3 and Fig. 3.4, it is seen that a superelement is created by releasing the interior DOF and keeping the boundary DOF. A superelement is treated as a structural element in a higher level substructure. The relationship between forces and displacements of the superelement is given by Eq. 3.16:

$$(P)_{(r)} = (K_p^{*})_{(r)} (p)_{(r)} + (P_0)_{(r)} \qquad \textbf{Eq. 3.21}$$

where:

$(\mathbf{p})_{(r)}$ is a vector of superelement boundary displacements which is equivalent to $\mathbf{p}^{(\beta)}$ in Eq. 3.16;

$(\mathbf{P})_{(r)}$ is a vector of total forces acting on the superelement boundary DOF;

$(\mathbf{K}_p^*)_{(r)}$ is the superelement stiffness matrix equivalent to the substructure boundary stiffness matrix $\mathbf{K}_p^*$ given by Eq. 3.15;

$(\mathbf{P}_0)_{(r)}$ is a vector of initial forces required to equilibrate forces applied within the superelement and to restrain the superelement boundary displacements, which is equivalent to $\mathbf{P}^{(\alpha)}$ in Eq. 3.16 (i.e. a vector of superelement initial forces).

Eq. 3.21 and Eq. 3.2 are similar to each other. Eq. 3.2 represents the relationship between element forces and element displacements, and Eq. 3.21 represents the relationship between external forces on the superelement boundary DOF and superelement boundary displacements. It should be noted that Eq. 3.21 is an equilibrium relationship with respect to the superelement local coordinate system, and a rotation of coordinate directions from the superelement local coordinate system to the local coordinate system for the next higher level substructure must be carried out. The result is similar to Eq. 3.3:

$$(Q)_{(r)} = (K_q^*)_{(r)}\,(q)_{(r)} + (Q_0)_{(r)} \qquad \textbf{Eq. 3.22}$$

Where:

$(\mathbf{q})_{(r)}$ is a vector of superelement displacements, which is equivalent to $(\mathbf{p})_{(r)}$ in Eq. 3.21;

$(\mathbf{Q})_{(r)}$ is a vector of total forces acting on the superelement boundary DOF, which is equivalent to $(\mathbf{P})_{(r)}$ in Eq. 3.21;

$(\mathbf{K}_q^*)_{(r)}$ is the superelement stiffness matrix with respect to next higher level substructure coordinate system;

$(\mathbf{Q}_0)_{(r)}$ is a vector of superelement initial forces, which is equivalent to $(\mathbf{P}_0)_{(r)}$ in Eq. 3.21.

3.3 Analysis of Higher Level Substructure

Having determined the superelement stiffness matrix $(\mathbf{K}_q^*)_{(r)}$ and the superelement initial forces $(\mathbf{Q}_0)_{(r)}$, each superelement can be treated as an element and assembled into a higher level substructure for analysis. For example in Fig. 3.5, superelement 1 and superelement 2 are derived from lower level substructures. Their boundary nodes are then relaxed simultaneously when displacements in the higher level substructure (substructure 3) are calculated.

To calculate these displacements, the higher level substructure is regarded as an assembly of superelements (such as superelement 1 and superelement 2) and ordinary elements, subjected to external loading. For the boundary nodes of a superelement (e.g. node b1, b2, b5, b6 for superelement 1 in Fig. 3.5) Eq. 3.22 holds. As Fig. 3.5 shows, the superelement boundary nodes are nodes in a higher level substructure. As shown in Fig. 3.6, the equilibrium relationship for a node connecting several superelements in a higher level substructure is:

$$\Big(\sum_{r} (Q_0)_{(r)}\Big)_{(n)} + \Big(\sum_{r} (K_q^{\star})_{(r)}\,(q)_{(r)}\Big)_{(n)} = (F_{(s)})_{(n)} \qquad \textbf{Eq. 3.23}$$

where:

(n) denotes the rows of the vectors that pertain to the particular node;

$(\mathbf{q})_{(r)}$ is a vector of unknown superelement boundary displacements, which are equal to certain displacements in the vector **U** for the higher level substructure;

$\mathbf{F}_{(s)}$ is a vector of external forces acting on the DOF of the higher level substructure;

$(\mathbf{K}_q^{*})_{(r)}$ is superelement stiffness matrix;

$(\mathbf{Q}_0)_{(r)}$ is a vector of superelement initial forces.

Note that if the higher level substructure contains elements as well as superelements, then $\mathbf{K}_q^{(e)}$, $\mathbf{q}^{(e)}$, and $\mathbf{Q}_0^{(e)}$ from Eq. 3.3 are included with $(\mathbf{K}_q^{*})_{(r)}$, $(\mathbf{q})_{(r)}$, and $(\mathbf{Q}_0)_{(r)}$ in Eq. 3.23. Note also that the summations indicate a superposition of internal forces (both the forces due to the displacements of the superelement boundaries and the superelement initial forces) over the superelements (or elements) connected to the node.

In Fig. 3.6, a typical node (b5) in the higher level substructures which is on the common boundary between superelement 1 and superelement 2 is shown. From Eq. 3.23:

$$((Q_0)_{(1)})_{(b5)} + ((Q_0)_{(2)})_{(b5)} + ((K_q^{\star})_{(1)}\, q_{(1)})_{(b5)} + ((K_q^{\star})_{(2)}\, q_{(2)})_{(b5)} = (F_{(3)})_{(b5)} \qquad \textbf{Eq. 3.24.a}$$

or:

$$((Q_0)_{(1)})_{(b5)} + ((Q_0)_{(2)})_{(b5)} + ((K_q^{\star})_{(1)}\, U^{(1)})_{(b5)} + ((K_q^{\star})_{(2)}\, U^{(2)})_{(b5)} = (F_{(3)})_{(b5)} \qquad \textbf{Eq. 3.24.b}$$

Where $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$ represent subvectors of **U** that correspond to the boundary DOF of superelement 1 and superelement 2 respectively, and **b5** indicates that only those rows of the vectors pertaining to node b5 are considered.

Eq. 3.23 can be written for all of the nodes of the higher level substructure without the summation in the form of Eq. 3.1:

$$F_0 + K\,U = F \qquad \textbf{Eq. 3.25}$$

where **K** is obtained by assembling the superelement stiffness matrices $(\mathbf{K}_q{}^*)_{(r)}$ and the element stiffness matrices $\mathbf{K}_q{}^{(e)}$ into their positions in the stiffness matrix for the substructure, and $\mathbf{F}_0$ is obtained by assembling the superelement initial forces $(\mathbf{Q}_0)_{(r)}$ and the element initial forces $\mathbf{Q}_0{}^{(e)}$ in their position into the initial force vector of the substructure. When the higher level substructure stiffness matrix is assembled and partitioned to form Eq. 3.18, a higher level superelement can be created.

Through the same process, from lower level to higher level substructures, the highest level substructure (i.e., the complete structure) can be formed and equilibrium equations can be written as:

$$(F_0)_{struc.} + (K)_{struc.}\,(U)_{struc.} = (F)_{struc.}$$

or:

$$\begin{Bmatrix} R_0 \\ Y_0 \end{Bmatrix}_{struc.} + \begin{bmatrix} K_{rr} & K_{ry} \\ K_{yr} & K_{yy} \end{bmatrix}_{struc.} \begin{Bmatrix} r \\ y \end{Bmatrix}_{struc.} = \begin{Bmatrix} R \\ Y \end{Bmatrix}_{struc.} \qquad \textbf{Eq. 3.26}$$

It should be noted that **p** does not appear in this equation because there are no boundary DOF defined for the highest level substructure (i.e. the complete structure).

Having determined the displacements **U** (including **r** and **y**) for the highest level substructure, the boundary displacements of the next lower level superelements can be obtained ($\mathbf{q}_{(r)}$ are equivalent to certain elements of **U** for the boundary DOF and $(\mathbf{p})_{(r)}$ are obtained by coordinate transformation from $\mathbf{q}_{(r)}$). The substructures from which these superelements are derived can now be analyzed separately under the known boundary displacements $(\mathbf{p})_{(r)}$. From Eq. 3.9 and Eq. 3.12 , it follows that the unknown interior displacements of the rth substructure are:

$$(r)_{(r)} = (r^{(\alpha)})_{(r)} + (r^{(\beta)})_{(r)} \qquad \textbf{Eq. 3.27}$$

where:

$$(r^{(\alpha)})_{(r)} = ([K_{rr}]^{(-1)}\,[R - R_0 - K_{ry}\,y])_{(r)}$$

$$(r^{(\beta)})_{(r)} = (-[K_{rr}]^{(-1)}\,K_{rp}\,p)_{(r)}$$

and the complete set of displacements for the rth substructure is:

$$(U)_{(r)} = \begin{Bmatrix} r \\ y \\ p \end{Bmatrix}_{(r)} = \begin{Bmatrix} r^{(\alpha)} \\ y \\ 0 \end{Bmatrix}_{(r)} + \begin{Bmatrix} r^{(\beta)} \\ 0 \\ p \end{Bmatrix}_{(r)} \qquad \textbf{Eq. 3.28}$$

From. 3.10 and Eq. 3.13, it follows that for the rth substructure, the reaction forces on the interior DOF corresponding to $\mathbf{y}_{(r)}$ are:

$$(Y)_{(r)} = (Y^{(\alpha)})_{(r)} + (Y^{(\beta)})_{(r)}$$

where:

$$(Y^{(\alpha)})_{(r)} = (Y_0 + K_{yr}[K_{rr}]^{(-1)}[R - R_0 - K_{ry}y] + K_{yy}y)_{(r)}$$

$$(Y^{(\beta)})_{(r)} = ([-K_{yr}(K_{rr})^{(-1)}K_{rp} + K_{yp}]p)_{(r)}$$

and the complete set of external forces acting on the DOF of the r^th substructure is:

$$(F)_{(r)} = \begin{Bmatrix} R \\ Y \\ P \end{Bmatrix}_{(r)} = \begin{Bmatrix} R \\ Y^{(\alpha)} \\ P^{(\alpha)} \end{Bmatrix}_{(r)} + \begin{Bmatrix} 0 \\ Y^{(\beta)} \\ P^{(\beta)} \end{Bmatrix}_{(r)} \qquad \textbf{Eq. 3.29}$$

where:

$$(P)^{(\beta)} = (K_p^{*}p)_{(r)}$$

The substructure displacements obtained from Eq. 3.28 are then used to determine the element forces using Eq. 3.3:

$$Q^{(e)} = k_e^{(e)}q^{(e)} + Q_0^{(e)} \qquad \textbf{Eq. 3.30}$$

where for each element the displacements $\mathbf{q}^{(e)}$ are equivalent to certain elements of the substructure displacement vector $\mathbf{(U)}_{(r)}$ from Eq. 3.28.

Through the same process from higher level to lower level substructures, displacements, reactions, and element forces are obtained.

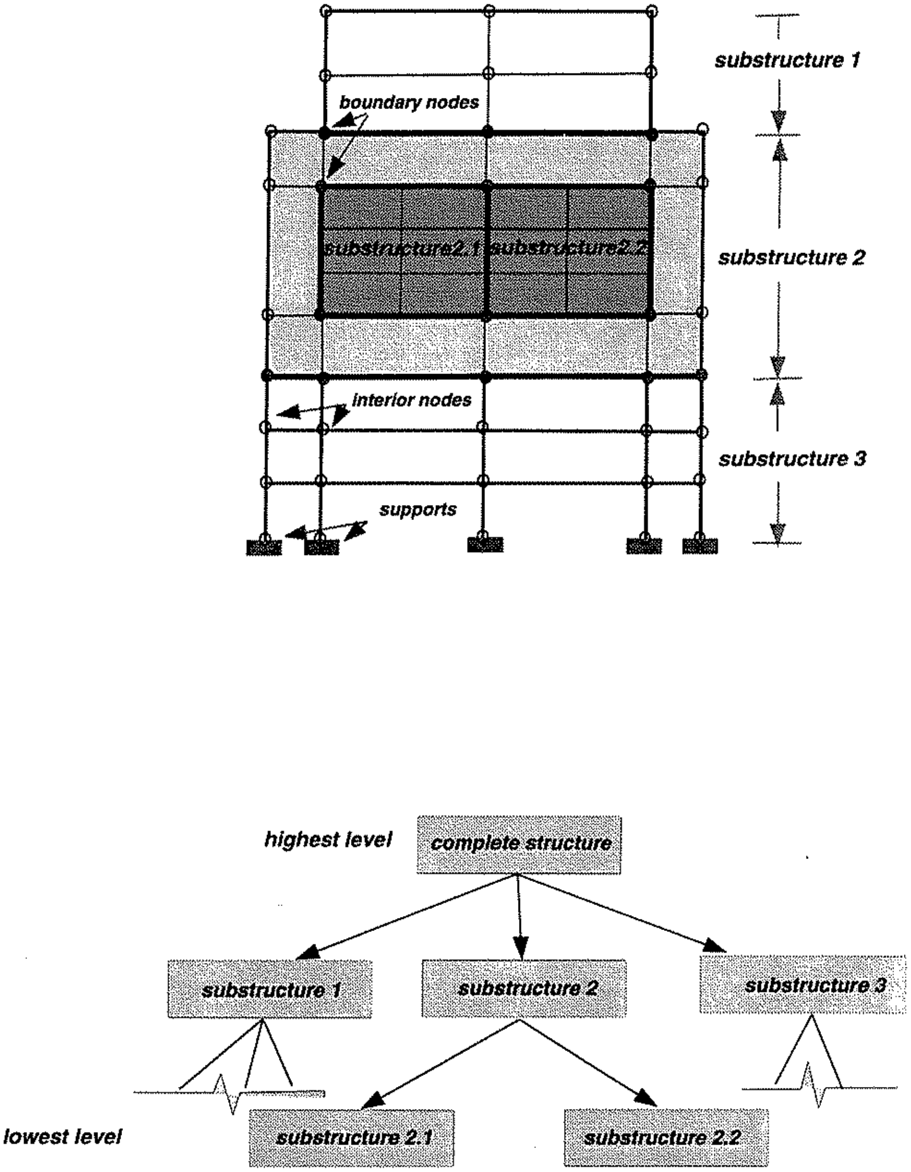


Fig. 3.1. Substructure-based analysis model.

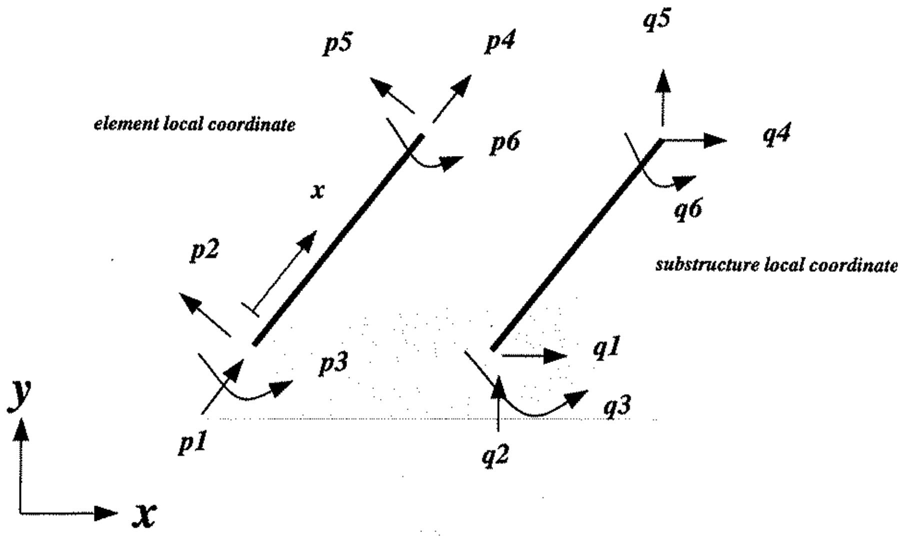


Fig. 3.2. Rotation of coordinate directions.

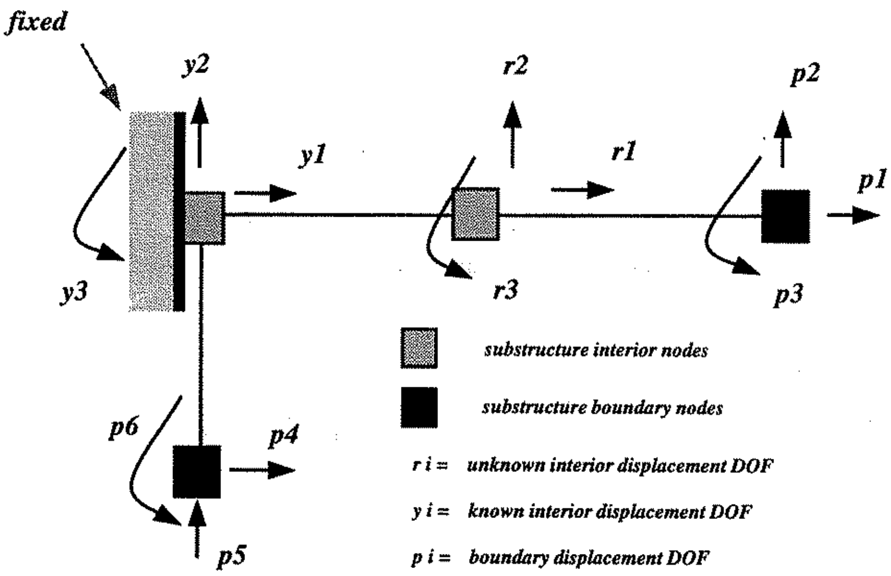

fixed
y2
r2
p2
y1
r1
p1
y3
r3
p3
substructure interior nodes
substructure boundary nodes
p6
p4
r i = unknown interior displacement DOF
y i = known interior displacement DOF
p5
p i = boundary displacement DOF

Fig. 3.3. DOF types for a typical substructure.

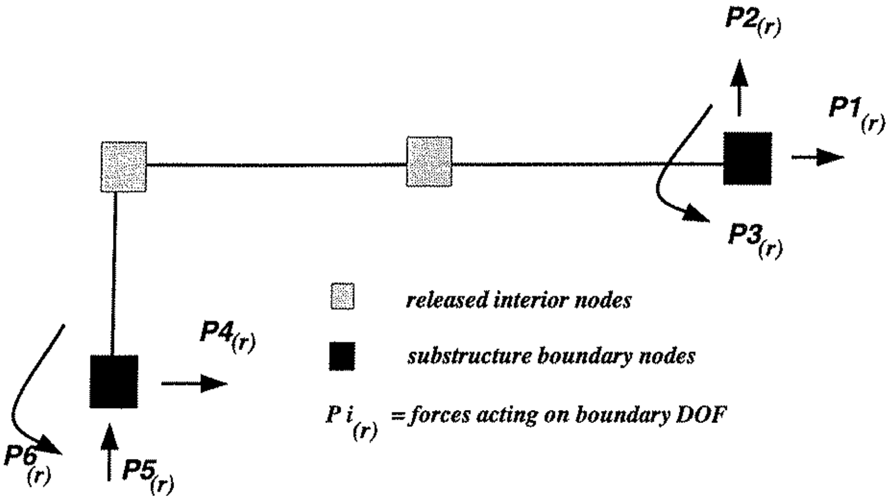


Fig. 3.4. Superelement developed from substructure.

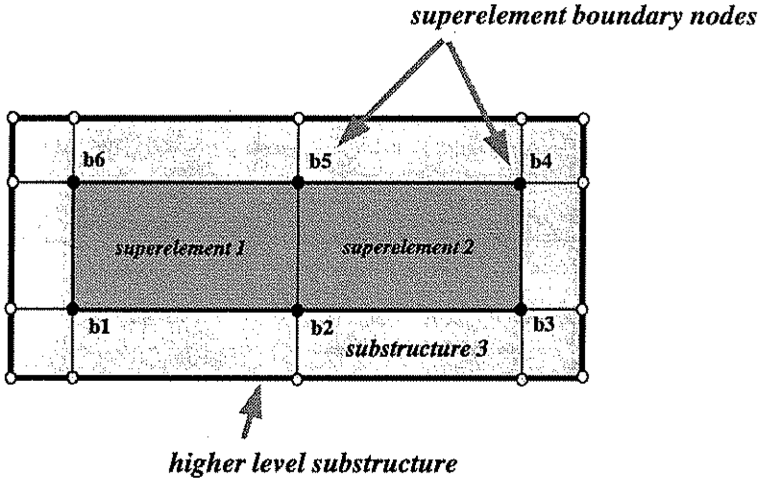
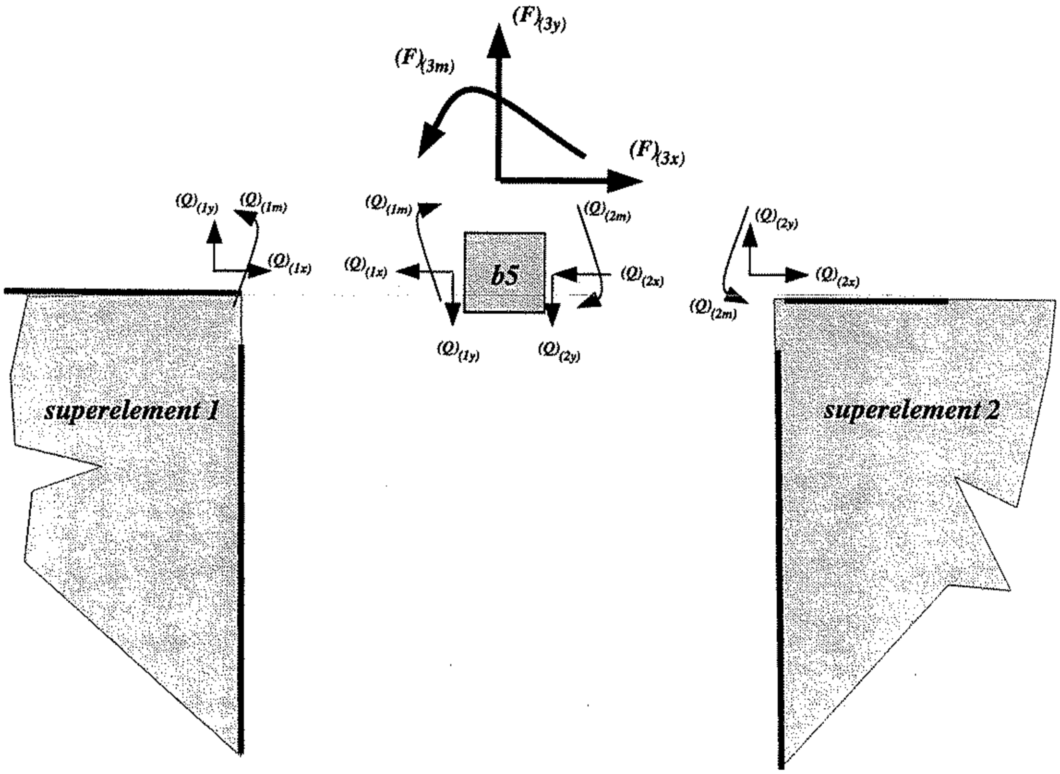


Fig. 3.5. Substructure with two superelements.



$(Q)_{(1)}$ = forces from superelement 1 **$(Q)_{(2)}$ = forces from superelement 2**

$(F)_{(3)}$ = total external forces acting on node b5 in higher level substructure 3

Fig. 3.6. Forces acting on the node b5 in Fig. 3.5.

Chapter 4

Data Abstraction and Object-Oriented Programming

4.0. Introduction

Data abstraction concepts and object-oriented programming techniques are recent developments in the area of computer programming that offer opportunities for improving computer software. Data abstraction, which is the use of abstract data types (ADT), is discussed first in this chapter. Then, object-oriented programming techniques are discussed, including an introduction to the C++ programming language. In addition, the NIH (National Institute of Health) class library is briefly reviewed, since it provides object classes that have been designed to reduce the effort of object-oriented programming in C++.

4.1. Data Abstraction Concepts

Data abstraction is the process inventing new data types that are well suited to a particular programming problem, and that make a complicated algorithm easier to program. The creation of new data types is a powerful, general-purpose technique which, when properly used, results in shorter, and more readable and flexible programs [7].

Most programming languages treat program variables and constants as *instances* of *data types*. A data type tells the program compiler how much memory to allocate, how to interpret the data in the memory, and what operations are permissible on the data. For example, the declaration:

INTEGER X

in FORTRAN declares an instance named *X* of the data type *integer*. The data type *integer* instructs the compiler to reserve sufficient memory for an integer and to use the machine's integer instructions to manipulate this memory, and defines operations such as "add" and "multiply". Other possible operations such as, "the result of the summation from 1 to the integer X" or "shifting from the integer position X to the integer position 100X on an uni-dimension coordinate", are not defined by the integer data type but could be implemented by a programmer. Data types that are built into a compiler in this way are known as *fundamental data types*.

Some programming languages, particularly the C++ language, have features that allow the language to be extended by adding data types. A programmer-defined data type is called as *abstract data type* (ADT) to distinguish it from a *fundamental data type*. According to Fenves, "the term 'abstract' refers to the way in which a programmer abstracts some concepts from a mass of programming detail and unifies these concepts to create a new data type" [4]. An ADT has an internal data representation and a number of legal operations that manipulate the data representation.

In order to illustrate the concepts of data abstraction in more detail, a programming problem, associated with structural analysis and implemented in C++, is presented in Fig. 4.1. C++ allows us to implement an abstract data type by creating a C++ *class* (as discussed in detail later). A class implements an ADT and is conceptually equivalent to INTEGER in FORTRAN. Classes, once defined, are similar to the fundamental data types built into the C++ compiler.

Figure 4.1 defines the Node class which is an ADT that can be used for structural analysis. It contains two parts, the private part and the public part. The private part contains several *instance variables* that comprise the internal representation of a Node. The instance variables x, y, and z define the position of the node in a coordinate system. nodeLabel is an identifier for the node, and dofArray is an array to store six labels for the six degree of freedom (DOF) associated with the node. The public part contains *functions* that comprise the legal operations on a Node. *Node (x_coordinate, y_coordinate, z_coordinate, Node_Label)* is a constructor function to create a Node by defining coordinates and assigning an identifier to the node. *Node (anotherNode)* is a function to create a Node by copying another available Node. The remaining functions define other permissible operations on a Node, for example, the x coordinate of the Node can be retrieved by calling the function *get_x ( )*. Each of these functions returns information from a Node, so a return type (e.g. String& or float &) is shown.

Once it has been defined, the Node ADT (i.e. the Node class) is a reusable software component for different programming applications. The Node ADT may serve different purposes in different programming problems. However, a Node performs only certain pre-defined operations (functions) on its internal data representation (instance variables), which includes both a fundamental data type (float) and two abstract data types (String and ArrayOb). The two ADTs String and ArrayOb are defined by other classes. The Node ADT encapsulates its internal data representation and operations so that a client program using the Node ADT is independent of the implementation details of the Node ADT.

A client program can use the Node ADT in the manner shown in Figure 4.2. The first two statements in main( ) declare two Node instances, Node A and Node B. Node A has been completely created with a Node label and a position in a three-dimensional coordinate system. The second statement creates Node B without any data stored in its internal data representation. The third statement assigns the Node A to Node B using "=". The assignment operator is carried out so that Node B holds the same data as Node A and the assignment uses the function *Node (anotherNode)* implicitly. The fifth statement retrieves the x coordinate from Node A using the function *get_x ( )* and the result is copied to an instance of a fundamental data type (*float x_coordinate* declared in the fourth statement).

In summary, the most important characteristic of an ADT is that it defines an external view of its operations and the associated semantics of the operations, and encapsulates the implementation of these operations in the internal data representation and the details of its operations. The client programs of the ADT see only the external level, allowing its implementation to be modified without adverse effects [10].

4.2 Object-Oriented Programming Concepts

Object-oriented programming organizes related ADTs and exploits their common features in order to reduce programming effort [7]. On object-oriented programming, ADTs are usually called classes, and instances of ADTs are called objects. Object-oriented programming is characterized by [16]:

- *Encapsulation* of data and operations inside instances. This isolates client programs from the implementation details of the data and operations of an ADT (a class). Client programs are not impacted by future changes in the encapsulated implementation, and programs are easier to debug and maintain because the details of the data and operations are isolated from client programs.
- *Inheritance* through an ADT (class) hierarchy simplifies the task of creating a new ADT (class) that is similar to an existing ADT (class). The new ADT inherits data and operations from the existing ADT (superclass), and only the *differences* between the new ADT and the existing ADT are specified.
- *Dynamic binding* allows each ADT (class) of a group of related ADTs (classes) to have a different implementation of a particular operation. Client programs can apply the operation to an instance (object) from one of the ADTs (of the group of related ADTs) without concern for the specific ADT (class) of the instance (object).

4.3 The C++ Language

The C++ programming language was designed by Bjarne Stroustrup of AT&T Bell Laboratories as a successor to the C language [7]. C++ borrows several key ideas from the Simula 67 and Algol 68 programming languages, but retains compatibility with existing C programs [7]. C++ is an interesting language because of its relationship to C, and its potential use for graphical user interfaces, for systems programming, and for supporting large-scale software development [19].

Figure 4.1 and 4.2 include some of the features which C++ provides to support data abstraction [11]:

- *Class* is the C++ construct for creating ADTs. *Class Node* defines an ADT. And Node A and Node B are instances of the ADT (class) and are called *objects.*
- *Instance variables* describe the data (i.e. the internal data representation) within an ADT (class). The data representation can include both fundamental data types and other ADTs which have been previously defined. In Fig. 4.1, the statement *float x, y, z* indicates that x, y, and z coordinates are instances of the fundamental data type float, whereas the statement *ArrayOb dofArray* indicates that dofArray is an instance of the ADT ArrayOb.
- *Functions* define the permissible operations on the data associated with an ADT (class). In Fig. 4.1, *String& getNodeLabel (), float& get_x (),* and so on, define operations on x, y, z, and nodeLabel, such as retrieving the nodeLabel or x,y,z coordinates. These functions allow client programs to access the data which is encapsulated in the ADT (class), but prevent adverse modification of the data.
- *Operator overloading* provides additional meaning to certain operators so that programmers can use them with ADTs. In Fig. 4.2, the third statement *B=A* overloads the equal sign (=) to assign the information in Node A to Node B. This makes the operation similar to the assignment of a fundamental data type (integer or float).

In addition, C++ provides other features for object-oriented programming:

- C++ supports encapsulation by providing classes with private and public parts. A class defines an ADT, and the declaration of an instance from this ADT creates an object. An object has the ability to perform operations on itself. The operations it performs are defined by the functions described in the public part of its class. Therefore, the external view of an object (i.e. the operations it performs) is defined by the public functions it executes. Internally, an object has instance variables and functions so that it has the ability to carry out the operations defined by its public functions. This is called as object's internal view. In object-oriented programming terminology, the external view of an object is said to be defined by the messages it receives and the responses it generates. In C++, the messages and responses are the names of the public functions, and the return values of the public functions. The external and internal view of an object are shown schematically in Fig. 4.3 [4].

- C++ supports inheritance by means of derived classes [4]. It is common for two classes to be similar with only a few different instance variables or functions. Each class in C++ can have one or more superclasses, and inherits the functions and instance variables of its superclasses. The relationship of these classes is shown in Fig. 4.4. For example, class Node, presented in Fig. 4.1, could be classified further into subclasses, such as Beam-element End Node (BENode), which specifically describes the nodes at the ends of a beam-element, or Superelement Boundary Node (SBNode), which describes the nodes located on a superelement boundary. A subclass usually includes specialized data and operations (instance variables and functions) compared to its superclass. Superclass data (such as *float x, y, z;* in Fig. 4.1) and operations (such as function *float& get_x ();* in Fig. 4.1) are inherited (and reused) by the subclasses.
- C++ supports dynamic binding by means of virtual functions [4]. A virtual function lets a subclass redefine a function inherited from a superclass. Through dynamic binding, the run time system executes the function that is appropriate for the class of a particular object. In Fig. 4.5, the Node class has two subclasses, BENode and SBNode. Each class has the function *move ( )* in its public part, however, the function has a different implementation in each class. In the Node class, the *move ( )* function is a virtual function. It changes the x, y, and z coordinates to enable an object from the Node class to move

in three dimensions. The *move ( )* function in the BENode changes the x coordinate of an object from the BENode class to enable it to move only along the element local x axis. The *move ( )* function in the SBNode changes the x and y coordinates within a two dimensional superelement local coordinate system. A function *move_It (Node& mover)* which uses the *move ( )* function is also shown (Fig. 4.5). The function is designed to move an object from the Node class or one of its subclasses. Two objects, BENode1 from the BENode class and SBNode2 of the SBNode class are declared. The two objects are used as arguments of the function *move_It.* It should be noted that the argument aNode of the move_It function is a pointer to an object from the Node class. At execution time, aNode can point to an object from the Node class or from any subclass (BENode or SBNode) of the Node class. C++ uses dynamic binding to determine which implementation of *move ( )* to execute. Since the *move ( )* function of the Node class is a virtual function, C++ looks for a subclass implementation of the move function to execute. As a result, when BENode1 is used as the aNode argument, the *move ( )* function of the BENode class is executed, when SBNode2 is the argument, the *move ( )* function of the SBNode is executed [7]. Note that the move function of the SBNode class is not virtual. If an object from a subclass of the SBNode class was used as the aNode argument, the SBNode version of *move ( )* would be executed.

4.4 Introduction to NIH Class Library

C++ lacks a class library, which makes it difficult to take advantage of its object-oriented features [7]. A programmer has to start without any pre-defined classes; only the fundamental data types are available. To overcome this limitation, the NIH (National Institute of Health) class library was developed by K. E. Gorlen to provide classes for programmers to use [7].

The NIH class library has been designed to simplify object-oriented programming using C++. It provides several general classes, such as *String* and *Float*, and a set of classes similar to the Smalltalk-80 container classes, such as *Collection* and *KeySortCltn.*

Certain NIH classes, such as the *Object* class and some of its subclasses, as well as the container classes are outlined below because they are used in this research. In Chapter 5, object classes for object-oriented structural analysis using substructures are developed from these NIH classes.

4.4.1 Class Object

The *Object* class is the most general NIH class. It is the root of the NIH class inheritance hierarchy, so that its functions are inherited by all the NIH classes that are its descendants. These functions define operations that apply to any class of objects, such as copying, printing, storing, reading, and comparing objects. However, the Object class does not actually implement these functions. Classes that are derived from the Object class implement these functions using instance variables and functions that implement specialized operations.

There are three categories of member functions declared by class Object that are particularly important [7]:

- Functions that identify and test the class of an object; that is functions that determine the information regarding which class the object is from;
- Functions that compare objects;
- Functions that print the contents of an object, including the functions *void printOn (iostream& strm=cout) const;* and *void dumpOn (iostream& strm=cerr) const.*

4.4.2. Class String, Integer and Float

Class *String* is a subclass of the Object class with ability to handle string operations [7]. It overloads many operators, such as *&*, *&=*, *<,<=*, and *==*, to concatenate character strings, to compare character strings, and to address the individual characters from a string.

Class *Integer* and class *Float* are subclasses of the Object class that provide operations on integer and float data types respectively [7]. Although these classes deal with ordinary fundamental data types, they are implemented as derived classes from Object class. Thus these three classes not only hold a position in the NIH class hierarchy, but also are compatible with any other NIH class.

4.4.3. The NIH Class Library Container Classes

The NIH class library container classes are outlined below without describing each of the public functions. A container class is a class of objects that stores multiple objects from other classes.

4.4.3.1. Class Collection

The *Collection* class serves as the base class of all container classes in NIH Library. It declares (but does not implement) virtual member functions required of all container classes [7]:

- insert an object into a container (*add( )*);
- delete an object from a container (*remove( )*);
- copy objects from one container to another (*addContentsTo( )*);
- test for the presence of an object in a container (*includes( )*);
- test for an empty container (*isEmpty( )*);
- report the number of objects in a container (*Size( )*);
- report the capacity of a container (*Capacity( )*);

4.4.3.2. Class SeqCltn

The *seqCltn* class is designed for containers that order objects in a certain sequence [7]. In the class, objects are arranged with the first object numbered 0, the second object numbered 1, and so on. The objects can be accessed by specifying their index number.

4.4.3.3. Class OrderedCltn

The *OrderedCltn* class is derived from class SeqCltn (Fig. 4.6). It differs from SeqCltn by the implementation of a variable-length array of pointers to objects. Instead of placing multiple objects into an array directly, the OrderedCltn class generates an array of pointers, with each pointer corresponding to an object. This is equivalent to inserting each object into the array. In addition, the OrderedCltn class automatically resizes itself (i.e. resizes the array) if it needs more room for storing pointers.

4.4.3.4. Class SortedCltn

The *SortedCltn* class is derived from class OrderedCltn (Fig. 4.6). It differs from OrderedCltn by the implementation of a sorted array of object pointers. It sorts the objects as it adds the pointer to a new object in its array. The objects are sorted according to the function *Compare ( )* that compares the objects. Each object in the container can be accessed with an integer index.

4.4.3.5. Class KeySortCltn

The *KeySortCltn* class sorts *associations* between *key objects* and *value objects.* An association is an object that holds:

- a pointer to a key object (e.g. a label for a Node object);
- a pointer to a value object (e.g. a Node object itself).

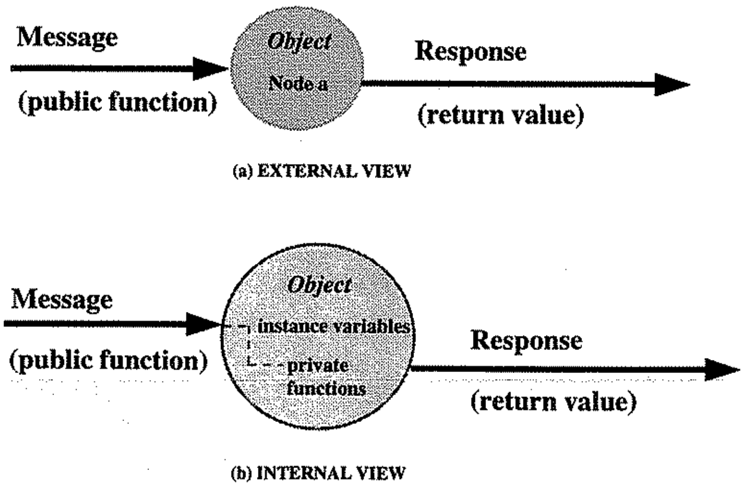
KeySortCltn uses the key object when performing comparisons. For example, a KeySortCltn can be used to sort Node objects (Fig. 4.1) through node labels (i.e. nodeLabel). The KeySortCltn will create a new association for the key (*nodeLabel*) and the value object (*Node)* for each Node object added to the container. When the association is added into a container, it compares the nodeLabels, and inserts the association into a list of sorted Node objects (sorted by nodeLabel).

```
Class Node {

  private:
     float        x, y, z;          // x, y, z coordinate;
     String       nodeLabel;        // Node Label;
     ArrayOb      dofArray;         // Labels of six DOF associated with Node;
     ... ...

  public:
     Node (x_coordinate, y_coordinate, z_coordinate, Node_Label);
                // to construct a Node by defining coordinates and label;

     Node (anotherNode);                    // to assign another Node to a Node;
     String&    getNodeLabel ( );           // to retrieve Node label;
     float&     get_x ( );                  // to retrieve x coordinate;
     float&     get_y ( );                  // to retrieve y coordinate;
     float&     get_z ( );                  // to retrieve z coordinate;

     ... ...  }
```

Fig. 4.1. C++ code that defines the Node class.

```
main ( ) {

        Node A (1.101, 2.54, 3.96, "node_1");
        Node B;
        B = A;
        float  x_coordinate;
        x_coordinate = A.get_x ( );
        ... ...
        }
```

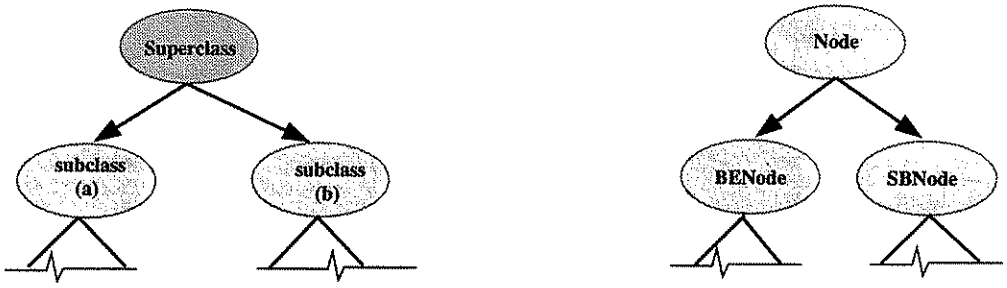
Fig. 4.2. C++ code that uses the Node class.



(a) EXTERNAL VIEW

(b) INTERNAL VIEW

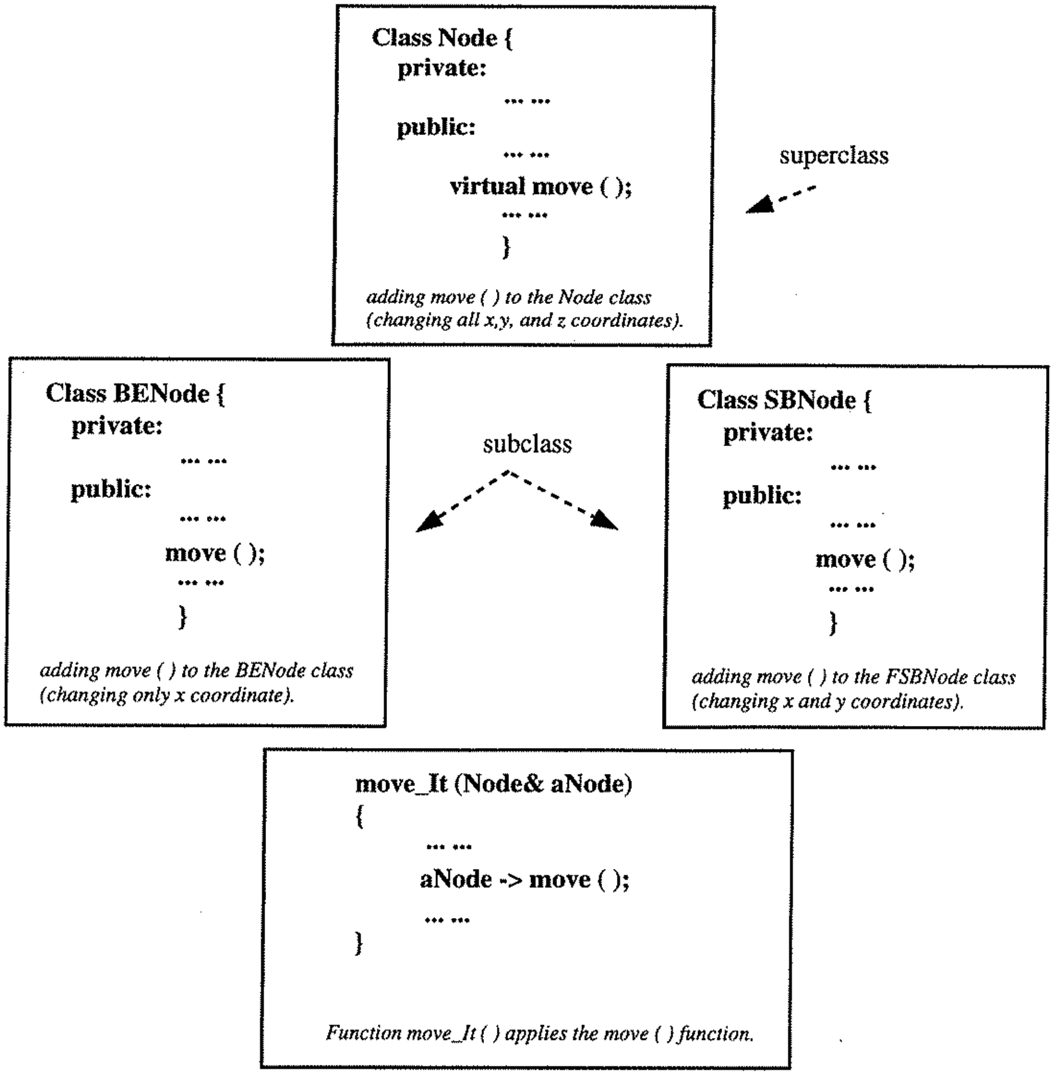
Fig. 4.3. Encapsulation - two views of an object.



A superclass can be further classified into subclasses and each subclass may also be further classified (omitted here).

For example, two types of subclasses, such as Beam-element_Node and Superelement_Boundary _Node can be derived from Node.

Fig. 4.4. Class hierarchy.



```
Class Node {
   private:
            ... ...
   public:
            ... ...
        virtual move ( );
            ... ...
            }
```

adding move ( ) to the Node class (changing all x,y, and z coordinates).

```
Class BENode {
   private:
            ... ...
   public:
            ... ...
        move ( );
            ... ...
            }
```

adding move ( ) to the BENode class (changing only x coordinate).

```
Class SBNode {
   private:
            ... ...
   public:
            ... ...
        move ( );
            ... ...
            }
```

adding move ( ) to the FSBNode class (changing x and y coordinates).

```
move_It (Node& aNode)
{
        ... ...
        aNode -> move ( );
        ... ...
}
```

Function move_It ( ) applies the move ( ) function.

```
main ( )
{

      BENode    BENode1 ( 1.11, 0, 0, "Beam-element_Node1");
      FSBNode   FSBNode2 ( 1.12, 1.13, 0, "Floor_Substructure_Node2" );
                        // declairing two objects of two subclasses.
      move_It (BENode1);
                        // dynamically bound---the function move ( ) of the BENode1 is called.
      move_It (FSBNode2);
                        // dynamically bound---the function move ( ) of the FSBNode1 is called.

      ... ...
}
```

Fig. 4.5. Dynamic binding.

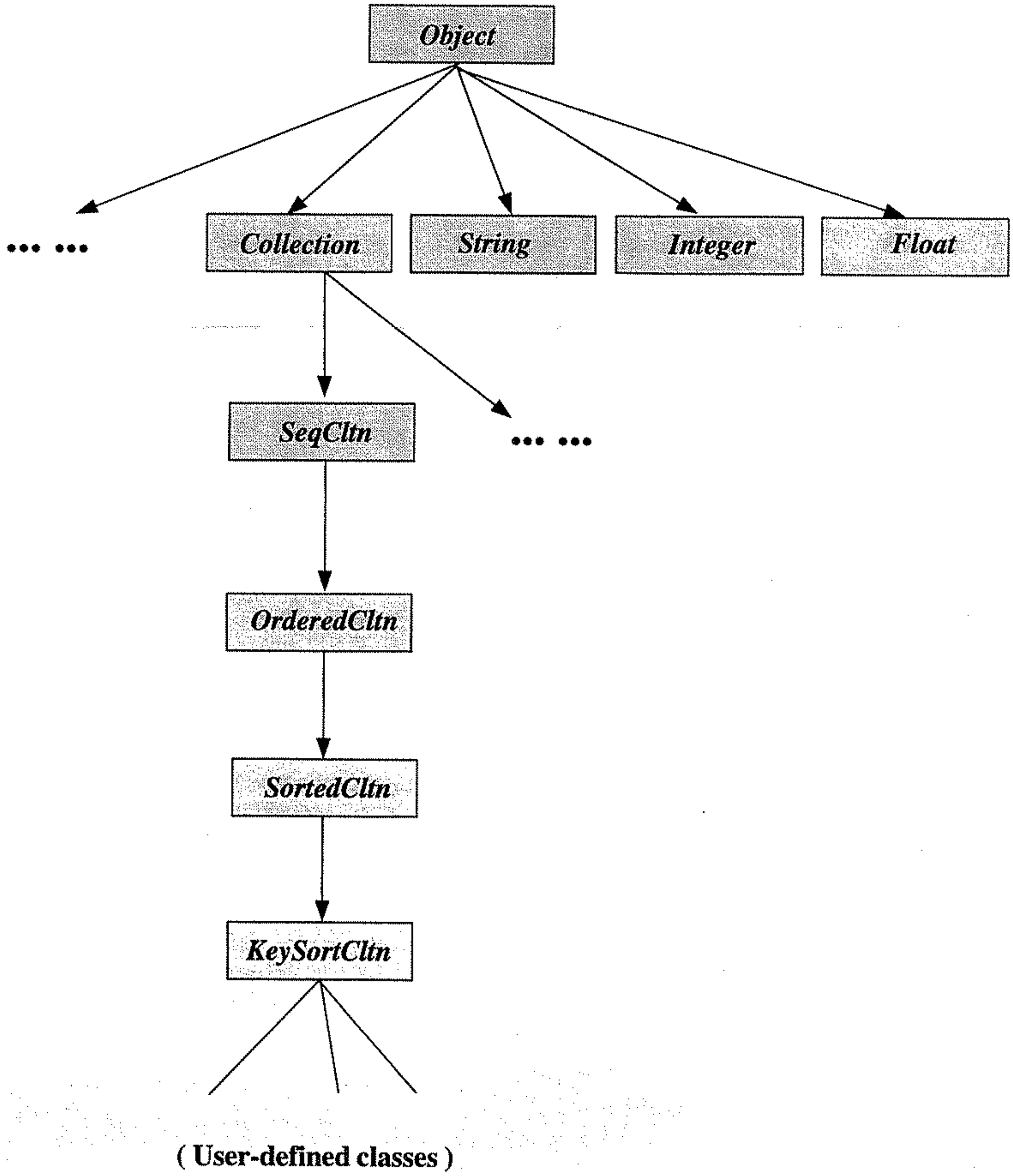


Fig. 4.6 Part of NIH class hierarchy.

Chapter 5

Object-Oriented Structural Analysis Model

5.0 Introduction

This chapter and Chapter 6 describe an approach to structural analysis based on the substructure analysis method and on an object-oriented model for the related mathematical and engineering concepts. In contrast to conventional approaches, the object-oriented model lets a program developer focus on the mathematical and engineering concepts before dealing with many of the low-level programming details and internal representations. The approach is presented in four parts. The first two parts are presented in this chapter. First, a structural analysis model is outlined. Then the object classes needed to implement this model are presented. InChapter 6, one additional object class called the Global object, which constructs the other objects and controls the analysis process, and a number of structural analysis algorithms that implement the object-oriented approach are presented.

5.1 Structural Analysis Model

5.1.1 Introductory Example

Fig. 5.1 shows a frame structure. The structure is composed of columns, footings, girders, floor beams, and floor slabs. A traditional model of the structure, shown in Fig. 5.1, consists of elements and nodes. However, a traditional model often neglects the advantages obtained from the repeated geometry of the subsystems and components of the structure that are dictated by economic and construction constraints. The geometric similarity of subsystems and components can be exploited to reduce the effort required to model a complex structure for analysis [3].

The structural analysis model used for this research is a hierarchical subdivision of the structure into *substructures*. Each substructure is composed of basic elements or lower level substructures. Fig. 5.2 shows one of several possible subdivisions of the structure in Fig. 5.1.

At the lowest level are *basic elements* such as individual beams and segments of girders and columns. As shown in Fig. 5.3, multi-element girder and column substructures are composed from basic elements, by placing nodes where the girders and columns are connected to other elements in the structure. A multi-element frame substructure is assembled from the girder and column substructures. A multi-element floor substructure is composed of basic elements that model the floor beams. A multi-element balcony substructure consists of elements that are similar to those of the floor substructures. Finally, the floor, balcony, and frame substructures are assembled to form the complete structure.

5.1.2 Ingredients of the Object-Oriented Model

From the example outlined above, we can see that higher level substructures are composed of lower level substructures, and perhaps, some additional nodes and basic elements. The lowest level substructures are composed from only nodes and elements. For the example in Figs. 5.1, 5.2, and 5.3, the model of the structure (the highest level substructure) consists of several nodes and eight substructures, including a roof substructure, four floor substructures, two frame substructures, and one balcony substructure. Each floor substructure consists of a number of nodes and beam elements. Each frame substructure consists of several nodes, four girder substructures, and two column substructures. Each girder and column substructure consists of several nodes and beam elements. The balcony substructure also consists of a number of nodes and beam elements.

In addition to nodes, elements, and substructures, the structural analysis model considered in this research also includes *superelements*. A superelement is derived from a substructure by carrying out the substructure analysis presented in Chapter 3. A superelement is used to represent a substructure within the next higher level substructure. Certain advantages can be achieved by introducing the concepts of superelements and substructures into the structural analysis model:

(1) A structure is decomposed into several substructures and each substructure is analyzed separately. This can be repeated through several levels. A structural analysis program with multiple levels of substructures can support the automated modeling approach proposed by An-Nashif [1];

(2) The effort needed to model and analyze a large structural system may be greatly reduced if there are many substructures with similar geometry, materials, loads, and boundary conditions. For example, in a multi-story building, it is usual for one floor subsystem to be identical to other floor subsystems in terms of geometry, materials, loads, and support conditions. Using the analysis method presented in Chapter 3, several superelements can be developed for these subsystems, and these superelements can be used in the higher level substructure that represents the complete structural system. However, because the floor subsystems are identical, it is possible to obtain superelements for several identical floor subsystems by describing the geometry, materials, loads, and boundary conditions, and analyzing only once, thus reducing the effort needed to model and analyze the structure.

The structural analysis model used for this research utilizes similarities between substructures during the course of developing superelements from substructures. One advantage is obtained from the geometric similarity of substructures. For example, in Fig. 5.1, the ground floor, first floor, second floor, and third floor are geometrically identical. Thus, a *Substructure Geometry Type* (SGT) is included as part of the structural analysis model. A SGT defines the *geometry* and *materials* of a substructure in terms of nodes, elements, and superelements, and is used to represent the geometry and materials of one or more substructures whose geometry and materials are identical (Fig. 5.4).

If two or more substructures have identical geometry, materials, and *loads*, they have the same *Substructure Type* (ST). A ST allows multiple substructures to be defined using one description of geometry, materials, and loads. A ST is differentiated from a SGT by specifying the loads. Two substructures with the same SGT have a different ST if there is a dissimilarity in their loading. For the example in Fig. 5.1, the ground floor and the third floor are not only geometrically identical, but are also subjected to the same loads, thus one ST is needed to represent them. However, as shown in Fig. 5.5, a different ST is needed to represent the second floor substructure since it has different loads.

If two or more substructures have identical *boundary DOF*, in addition to geometry, materials, and loads, they have the same *Superelement Type* (SET). The term "boundary DOF" refers to the DOF where the substructure connects to other substructures. A SET is differentiated from a ST by specifying these boundary DOF. Two substructures that have the same ST have a different SET if there is a dissimilarity in their boundary DOF. In Fig. 5.6, three floor systems are shown. Although they have the same ST, the ground floor and the third floor have a different SET due to their different boundary DOF (i.e. they are connected to other substructures in different ways).

A *superelement* (SE) is obtained from a SET. One superelement is needed for each substructure in the model with the exception of the highest level substructure. Similar to basic elements, superelements are used in creating a model for the next higher level substructure.

Each ingredient of the structural analysis model outlined in this section has a corresponding object in the object-oriented structural analysis program envisioned by this research. These objects (DOF, node, element, SGT, ST, SET, and SE) are described in the following two sections. The user of the object-oriented structural analysis program uses these objects to create a structural analysis model for the structure in Fig. 5.1 as shown in Fig. 5.7. Other related objects in the structural analysis program are also described in the following sections.

5.2 Basic Objects

5.2.1 DOF Objects

A DOF object models an individual DOF of the structural analysis model. Each DOF object contains the following instance variables and functions (Fig. 5.8).

Instance variables:

(1) *DOF_Label* is an instance of the String class of the NIH class library, and is the identifier of the DOF object. Each DOF within a substructure has a unique label. The DOF_Label can be established automatically. For example, the labels of the six DOF attached to "Node1" may be named from "DOF_1-1" to "DOF_1-6", where 1 through 6 are the translations and rotations along the local x, y, and z coordinate directions within the substructure. The labels of the DOF can also be specified by the analyst.

(2) *DOF_Type* identifies the type of the DOF. The DOF within a substructure are divided into three categories, "r" type, for which the external force acting on an interior DOF is known and the analysis will determine the corresponding displacement; "y" type, for which the displacement of an interior DOF is known and the corresponding reaction force will be determined; and "p" type, for which the displacement of a boundary DOF is unknown. In most cases, there are more "r" type DOF than "y" type or "p" type DOF, therefore, "r" type is initially assigned to every DOF. Then, the DOF types are reset from "r" type to "y" type for those DOF whose displacements are known, and from "r" type to "p" type for the boundary DOF.

(3) *DOF_Index* points to a position in the force and displacement vectors that corresponds to the DOF.

Public functions:

(1) *DOF (DOF_Label, DOF_Type, DOF_Index);*

To construct a DOF object by providing the DOF label, the DOF type and the DOF index.

DOF (anotherDOF);

To construct a DOF object by copying another DOF object.

(2) *getDOF_Label ( );*

To return the DOF label.

(3) *getDOF_Type ( );*

To return the DOF type.

(4) *getDOF_Index ( );*

To return the DOF index.

(5) *setDOF_Type (newDOF_Type);*

To reset DOF type (if its type should be reset).

(6) *setDOF_Index (newDOF_Index);*

To set DOF index.

5.2.2. Node Objects

Each *Node* object contains the following instance variables and functions (Fig. 5.9).

Instance variables:

(1) *Node_Label* is an instance of the String class of the NIH class library, and is the identifier of the Node object. Every node in a substructure has an unique label. The label enables a Node object to be manipulated by other objects.

(2) *x_Coordinate, y_Coordinate, z_Coordinate* describe the node position in the substructure local x, y, and z coordinate system. It should be noted that each substructure has its own coordinate system. Consequently, the DOF directions will vary from one substructure to another.

(3) *DOF_Array* is an object derived from the ArrayOb class of the NIH class Library.

The array holds six label strings corresponding to six DOF objects associated with the Node object (each node has six DOF). The DOF_Array object included in the Node object establishes a link between the node and its six DOF so that any of the six DOF objects can be accessed through the Node object.

Public functions:

(1) *Node (Node_Label, x_coordinate, y_coordinate, z_coordinate);*

To construct a Node object by providing the Node_Label and x, y, and z coordinates.

Node (anotherNode);

To construct a Node object by copying another Node object.

(2) *getNode_Label ( );*

To return the label of the Node object.

(3) *get_x ( );*

To return the x coordinate.

(4) *get_y ( );*

To return the y coordinate.

(5) *get_z ( );*

To return the z coordinate.

(6) *getDOF_LabelAt (anInteger);*

To report one of the six DOF labels indicated by a number (integer 1 to 6).

(7) *set_x (new_x);*

set_y (new_y);

set_z (new_z);

To move the Node object from one location to another by changing its x, y, or z coordinates.

5.2.3. Element Objects

The *Element* object represents the deformable material of the structure. Here it is assumed that the Element is a beam element, although many of the functions developed here could be developed similarly for other types of elements. Each Element object contains the following instance variables and functions (Fig. 5.10).

Instance variables:

(1) *Element_Label* is an instance of the String class of the NIH class library, and is the identifier for the Element object.

(2) *Node_Labels* are instances of the String class of the NIH class library. There are three Node_labels: i_Node_Label, j_Node_Label, and reference_Node_Label. They represent three Node objects associated with the Element. Two of them (i_Node and j_Node) are the nodes located at each end of the element, and the length and location of the element are determined from these node coordinates. The reference_Node is a reference node used to determine the element local coordinate system.

(3) *E, A, and I* are material and geometry properties including Young's modulus (E), cross section area (A), and corss section moment of inertia (I). These properties vary from element to element, and should be provided when an Element object is constructed.

(4) *Element_Stiffness_Matrix* depends on the element's material and geometry properties, and the element length. The matrix is initially constructed by the Element object in element local coordinates. Then, the matrix is transformed to the substructure local coordinate system.

(5) *Element_Transformation_Matrix* depends on the coordinates of the three nodes (described in (2)). The matrix is a 3-by-3 transformation matrix between the element local coordinate system and the substructure local coordinate system.

(6) *Element_Initial_Forces* is a vector of forces acting on the element ends. The initial forces are calculated from span loads, w_x, w_y, w_z, m_x, m_y, and m_z by the Element object, and are transformed to the substructure local coordinate system.

(7) *Element_Displacements* is a vector of displacements at the element ends. The displacements are calculated by a substructure in the substructure local coordinate system.

Public functions:

(1) *Element (anElement_Label, i_node_Label, j_node_Label, reference_Node_Label, E, I, A);*

To construct an Element object by specifying the Element_Label, the labels of the i_Node, j_Node, and reference_Node, E (Young's modulus), A (area of cross section), and I (moment of inertia of cross section).

Element (anotherElement);

To construct an Element object by copying another Element object.

(2) *getLabel ( );*

To return the Element_Label.

(3) *getNumberOfNodes ( );*

To return the number of boundary nodes for the Element object (i.e. 2).

(4) *getNode_Label (anInteger);*

To return the Node_Label of a boundary node for the Element object by indicating an integer (i.e. 1 or 2).

(5) *formStiffness_Matrix ( );*

To form the element stiffness matrix in the substructure local coordinate system. The element stiffness matrix is formed with respect to the element local coordinate system and transformed to the substructure coordinate system using Element_

Transformation_Matrix. A coefficient of the stiffness matrix can be retrieved by indicating the row number and column number. The transformation algorithm is presented in Chapter 6.

(6) *getStiffness_Coefficient (anInteger_i, anInteger_j);*

To return a transformed element stiffness coefficient.

(7) *form_Initial_Forces (anElement_Load);*

To form the element initial force vector from specified element span loads. The force vector is also transformed from the element coordinate system to the substructure coordinate system by this function.

(8) *get_Initial_Force_Coefficient (anInteger);*

To return a coefficient from the element initial force vector by indicating the position of the coefficient in the vector.

(9) *calculateInternalForces (anElement_Load, theLoad_Case_Label);*

To calculate the element internal forces given the end displacements in the Element_Displacements vector, and specified element span loads. This function is used by a ST after a substructure analysis is complete and the element internal forces can be determined for a load case. The algorithm is presented in Chapter 6.

(10) *addDataToDisplacements (anInteger, aValue);*

To add data to the Element_Displacements vector.

5.3 Substructure and Superelement Objects

5.3.1. SGT Objects

A *Substructure Geometry Type* (SGT) object represents the geometry and materials of a group of one or more substructures whose geometry and material properties are identical. Although the substructures in the group have identical nodes and elements, they may have different load cases and boundary conditions. A SGT contains three types of objects: Nodes, Elements, and SEs. Three *containers* are implemented inside the SGT object to hold the Node, Element, and SE objects. A SGT object contains the following instance variables and functions (Fig. 5.11).

Instance variables:

(1) *SGT_Label* is an instance of the String class of the NIH class library, and is the identifier of the SGT.

(2) *theNode_Container* is an instance of the KeySortCltn class of the NIH class library. It is a container of the Node objects defined for the substructure. Instead of holding each Node object, it holds associations of *key objects*, which are Node_Label objects, with *value objects*, which are Node objects. It sorts only the key objects; the value objects are sorted by being associated with the key objects. The associated Node_Label object and Node object are stored in the container and are located and retrieved by using the key object, which is the Node_Label.

(3) *theElement_Container* is also an instance of the KeySortCltn class. It is a container of Element objects. Every Element object within the substructure is stored and sorted within this container. The container holds associations of key objects, which are Element_Label objects, with value objects, which are the Element objects. Each Element object in the group is accessed by using its label.

(4) *theSE_Container* is also an instance of the KeySortCltn class. It is a container of SE objects and is similar to theElement_Container.

Public functions:

(1) *SGT (aSGT_Label);*

To construct a SGT object by providing a SGT label.

SGT (anotherSGT);

To construct a SGT object by copying another SGT object.

(2) *getSGT_Label ( );*

To return the SGT label.

(3) *isTheNode_In (aNode_Label);*

To check if a Node object is available in theNode_Container using a Node_Label.

(4) *isTheElement_In (anElement_Label);*

To check if an Element object is available in theElement_Container using an Element_Label.

(5) *isTheSE_In (aSE_Label);*

To check if a SE object is available in theSE_Container using a SE_Label.

(6) *storeNode (aNode_object);*

To store a Node object into theNode_Container.

(7) *storeElement (anElement_object);*

To store an Element object into theElement_Container.

(8) *storeSE (aSE_object);*

To store a SE object into theSE_Container.

(9) *getNumberOfNodes ( );*

To report the number of Node objects held in theNode_Container.

(10) *getNumberOfElements ( );*

To report the number of Element objects held in theElement_Container.

(11) *getNumberOfSEs ( );*

To report the number of SE objects held in theSE_Container.

(12) *getNodeWithLabel (aNode_Label);*

To return a Node object from theNode_Container by label.

(13) *getElementWithLabel (anElement_Label);*

To return an Element object from theElement_Container by label.

(14) *getSE_WithLabel (aSE_Label);*

To return a SE object from theSE_Container by label.

(15) *getNode_At (anInteger);*

To get a Node object by an index indicating the position of the Node object in theNode_Container.

(16) *getElement_At (anInteger);*

To get an Element object by an index indicating the position of the Element object in theElement_Container.

(17) *getSE_At (anInteger);*

To get a SE object by an index indicating the position of the SE object in theSE_Container.

5.3.2. ST Objects

A *Substructure Type* (ST) object represents the geometry, materials, and loads of a group of one or more substructures. As shown in Fig. 5.5, the ground floor and the third floor of the example structure are modeled using the same ST object, called "Substructure Type 1", because of their similarity in the layout of nodes and elements, as well as in loads. The second floor is modeled using a different ST object.

A ST object is differentiated from a SGT object by adding load cases. A SGT object is included in the ST object to describe its geometry and materials. Operations to analyze a substructure are implemented and sets of results are obtained using the analysis method presented in Chapter 3. Each load case consists of specified forces and specified displacements. DOF with specified forces are "r" type DOF, and the specified force on a "r" type DOF becomes an element of the **R** vector. DOF with specified displacements are "y" type DOF, and the specified displacement of a "y" type DOF becomes an element of the **y** vector. Boundary DOF are "p" type DOF, and the displacement of a "p" type DOF becomes an element of the **p** vector. The displacement of a "p" type DOF is assumed to be zero in the current level substructure analysis and is determined in a higher level substructure analysis. It should be noted that the specification of "p" type DOF is performed by a SET object, as discussed later, however, the definition of the "p" type DOF is stored by the ST object. The division of DOF into "r" type, "y" type, and "p" type is the same for all load cases.

The ST object manages the DOF and organizes them into the three categories: "r" type DOF, "y" type DOF, and "p" type DOF and assembles a substructure stiffness matrix and substructure load and displacement vectors. In order to create the stiffness matrix (and to create **R**, **y**, and **p** vectors), each DOF is assigned a DOF index. To do this, a DOF_Manager object is included in the ST object to manage the "r", "y", and "p" type DOF and assign the DOF indices. In addition, a Load_Case_Manager object is included in the ST object to manage the load cases acting on the substructure. These objects, shown in Fig. 5.12, are described in more detail later. A ST object contains the following instance variables and functions (Fig. 5.12).

Instance variables:

(1) *ST_Label* is an instance of the String class of the NIH class library, and is the identifier of the ST object.

(2) *theSGT* is a SGT object that describes the geometry of the ST.

(3) *theDOF_Manager* is a DOF_Manager object that manages the DOF held in the ST.

(4) *theLoad_Case_Manager* is a Load_Case_Manager object that manages the load cases acting on the ST.

(5) *SubstrucStiffness* is the stiffness matrix for the ST.

Public functions:

(1) *ST (ST_Label, aSGT);*

To construct a ST object by providing a ST_Label and a SGT object with which the ST is associated.

ST (anotherST);

To construct a ST object by copying another ST object.

(2) *getST_Label ( );*

A SET object uses this function to get the ST_Label.

(3) *getNodeWithLabel (aNode_Label);*

A SET object uses this function to retrieve a Node object by indicating its label. The Node object is retrieved from theSGT.

(4) *getStiffness_Matrix ( );*

A SET uses this function to get a condensed substructure stiffness matrix.

(5) *buildDOF_Container ( );*

This function is used by the Global object to build a DOF container in the theDOF_Manager to store DOF objects. The algorithm is presented in Chapter 6.

(6) *establish_y_DOF ( );*

The Global object uses this ST function to identify the "y" type DOF for the ST. The algorithm is presented in Chapter 6.

(7) *setDOF_TypeFrom_r_to_p (aDOF_Label);*

A SET object uses this ST function to set the type of a DOF to "p" from the initially assumed "r" type. The algorithm is given in Chapter 6.

(8) *set_r_TypeIndices ( );*

This function is used by a SET object to assign an index to each "r" type DOF object.

(9) *getDOF_TypeWithLabel (aDOF_Label);*

To get the DOF_Type of a DOF object from theDOF_Manager by a DOF_Label.

(10) *getDOF_IndexWithLabel (aDOF_Label);*

To get the DOF_Index of a DOF object from theDOF_Manager by a DOF_Label.

(11) *buildLoad_Cases ( );*

The Global object uses this function to construct and fill theLoad_Case_Container, which is within theLoad_Manager, within theLoad_Case_Manager object. The algorithm is given in Chapter 6.

(12) *buildLoad_Case_Vectors ( );*

The Global object uses this function to construct and fill theLoad_Case_Vector_Container, which is within theVector_Manager, within theLoad_Case_Manager. The algorithm is given in Chapter 6.

(13) *getInitial_Forces (aCurrent_Level_Load_Case_Label);*

A SET object uses this function to retrieve a P_Vector under a load case by indicating its label.

(14) *SubstructureAnalysis ( );*

A SET object and the Global object use this function to carry out the first part of the substructure analysis (the α case). The SET object uses it for a lower level substructure and the Global object uses it for the highest level substructure. The algorithm is shown in Chapter 6.

(15) *LowerLevelSubstructureAnalysisCompletion (p_Vector, Current_Level_Load_Case _Label, Higher_Level_Load_Case_Label);*

A SET object uses this function to produce the substructure analysis results for the boundary displacements in p_Vector obtained from a higher level substructure analysis. The displacements in p_Vector are for a higher level load case identified by the Higher_Level_Load_Case_Label. The Current_Level_Load_Case_Label describes the current level load case which corresponds to these displacements. The current level load case is copied and stored under a new label (the Higher_Level_Load_Case_Label) and the analysis is completed for this new load case. The algorithm is presented in Chapter 6.

(16) *HighestLevelSubstructureAnalysisCompletion ( );*

The Global object uses this function to complete the substructure analysis for the highest level substructure. The algorithm is shown in Chapter 6.

(17) *get_p_number ( );*

A SET object uses this function to obtain the size of the p_Vector and P_Vector of theST.

Other functions:

The following ST functions are used to assemble the substructure stiffness matrix and retrieve the internal forces of the elements in the substructure. The algorithms are presented in Chapter 6.

(18) *assembleStiffnessMatrix ();*

This function assembles the substructure stiffness matrix from Element and SE contributions. The algorithm is shown in Chapter 6.

(19) *assembleElemStiffMatrix (anElement);*

This function assembles one Element or SE stiffness matrix into the substructure stiffness matrix *SubstrucStiffness.* The algorithm is shown in Chapter 6.

(20) *calculateElemInternalForces (theElement, theLoad_Case_Label);*

This function is used to calculate element (both Element and SE) internal forces. The algorithm is shown in Chapter 6.

(21) *getMatrixIndexWithDOF_Label (aDOF_Label);*

To report the matrix index for a DOF so that the DOF can be mapped to coefficients of the substructure stiffness matrix. For example, if a DOF has the "r" type and its DOF_Index in the "r" type DOF category is "i", then its row/column position is "i". If a DOF has the "y" type and its DOF_Index in the "y" type DOF category is "j", then its row/column position is "r_number+j". Similarly, if a DOF has the "p" type and its DOF_Index in the "p" type category is "k", then its row/column position is "r_number+ y_number+k".

5.3.3 DOF_Manager Objects

A *DOF_Manager* object manages the DOF of a ST object. It maintains the DOF objects in a DOF container, sorts the DOF into three groups ("y" type, "r" type, and "p" type groups based on the DOF type), and sorts DOF separately within these groups. Each DOF is assigned a DOF index which indicates its location in either the "y" type group, "r" type group, or "p" type group.

In addition, the dimension of the force vector **R** and corresponding displacement vector **r** for "r" type DOF, the dimension of the specified displacement vector **y** and corresponding reaction force vector **Y** for "y" type DOF, and the dimension of the boundary displacement vector **p** and corresponding reaction force vector **P** for "p" type DOF are stored by the DOF_Manager object. The number of "r" type DOF, number of "y" type DOF, and number of "p" type DOF are determined during the process of sorting the DOF. After sorting the DOF, the DOF_Manager object will retain these numbers, which are called *r_number* (the number of "r" type DOF), *y_number* (the number of "y" type DOF), and *p_number* (the number of "p" type DOF). A DOF_Manager object includes the following instance variables and functions (Fig. 5.13).

Instance variables:

(1) *theDOF_Container* is an instance of the KeySortCltn class, which serves as a container to hold DOF objects. As a KeySortCltn object, it holds associations of key objects with value objects. The key objects are the DOF labels (Section 5.2.1), such as "DOF_1.1", for the x translation DOF of node 1, or "DOF_3.4", for the x rotation DOF of node 3. The value objects are the DOF objects themselves. By establishing these associations, two lists are formed: one list is the *key object* list, the other list is the *value object* list (Fig. 5.14). For example, key object "DOF_1.1" corresponds to a value object, which is the x translation DOF of node 1. When the key objects (DOF labels) are sorted, the value objects (DOF objects) are also sorted.

(2) *r_number, y_number,* and *p_number* indicate the number of "r" type DOF, "y" type DOF, and "p" type DOF respectively. As indicated in Section 5.2.1, a DOF object

includes a type flag, either "r"type, "y" type, or "p" type. Based on these three types, each DOF object is assigned to one of the three groups and an index is assigned to indicate its position in the group. The *DOF_Manager* object determines these numbers during the process of establishing the DOF_Container as discussed in Chapter 6.

Public functions:

(1) *DOF_Manager (r_number, y_number, p_number);*

To construct a DOF_Manager object by setting the r_number, y_number, and p_number. Usually these are set to zero.

DOF_Manager (anotherDOF_Manager);

To construct a DOF_Manager object by copying another DOF_Manager object.

(2) *createDOF_Object (aDOF_Label);*

This function is used by the ST object to create a DOF object based on its DOF_Label. It uses the DOF object constructor, and inserts the DOF object into theDOF_Container. Part of the algorithm is presented in Chapter 6.

(3) *setDOF_TypeFrom_r_to_y ( );*

This function is used by the ST object to set the type of a DOF to "y" type from the initially assumed "r" type.

(4) *setDOF_TypeFrom_r_to_p ( );*

This function is used by the ST object to set the type of a DOF to "p" type from the initially assumed "r" type.

(5) *set_r_TypeIndices ( );*

This function is used by the ST object to assign an index to each "r" type DOF object. The algorithm is presented in Chapter 6.

(6) *get_y_number ( );*

This function is used by the ST object to obtain the number of "y" type DOF objects.

(7) *get_r_number ( );*

This function is used by the ST object to obtain the number of "r" type DOF objects.

(8) *get_p_number ( );*

This function is used by the ST object to obtain the number of "p" type DOF objects.

(9) *getDOF_TypeWithDOF_Label (aDOF_Label);*

This function is used by the ST object to obtain the DOF type of a DOF object using the DOF_Label to locate the DOF object.

(10) *getDOF_TypeAt (anInteger);*

This function is used by the ST object to obtain the DOF type of a DOF object by indicating the position of the DOF object in the DOF container.

(11) *getDOF_IndexWithDOF_Label (aDOF_Label);*

This function is used by the ST object to obtain DOF index of a DOF object using the DOF_Label to locate the DOF object.

5.3.4 Load_Case_Manager Objects

A *Load_Case_Ma*nager object manages loads acting on a ST, as well as the corresponding sets of force and displacement vectors. It contains two lower-level management objects: *theLoad_Manager* object, which manages multiple load cases composed of individual loads, and *theVector_Ma*nager object, which holds multiple sets of force and displacement vectors corresponding to these load cases. A Load_Manager object stores and organizes multiple *Load_Case* objects and each Load_Case object holds multiple *Load* objects. Therefore, many Load objects can be indirectly managed by the Load_Case_Manager object (Fig. 5.15).

For each Load_Case considered in the substructure analysis, there is a corresponding set of vectors. This set includes six vectors: the displacement vector **r** and corresponding force vector **R** for "r" type DOF, the displacement vector **y** and corresponding reaction force vector **Y** for "y" type DOF, and the boundary displacement vector **p** and corresponding reaction force **P** for "p" type DOF. Each set of vectors is managed by a *Load_Case_Vector* object. Multiple Load_Case_Vector objects are managed by a *Vector_Manager* object, and through the Vector_Manager object, the Load_Case_Manager object can indirectly manage many sets of load and displacement vectors (Fig. 5.16). The Load_Manager and Vector_Manager objects are discussed later. A Load_Case_Manager object contains the following instance variables and functions (Fig. 5.17).

Instance variables:

(1) *theLoad_Manager* is an instance of the Load_Manager class. It manages loads acting on the ST.

(2) *theVector_Manager* is an instance of the Vector_Manager class. It manages force and displacement vectors.

Public functions:

(1) *Load_Case_Manager (the_r_number, the_y_number, the_p_number);*

To construct a Load_Case_Manager object.

Load__Case_Manager (anotherLoad_Case_Manager);

To construct a Load_Case_Manager object by copying another Load_Case_Manager object.

(2) *getNumberOfLoadCases ( );*

To return the number of load cases.

(3) *getLoad_Case_LabelAt (anInteger);*

To return a Load_Case_Label by a position in theLoad_Case_Container (see section 5.3.5).

(4) *copyLoad_Case (Load_Case_Label_1, Load_case_label_2);*

To copy one load case to another, including the vectors in the Load_Case_Vector object managed by theVector_Manager, and the Load objects managed by theLoad_Manager.

The following Load_Case_Manager functions are used by the ST object to send information that is managed by *theVector_Manager.*

(5) *storeWhole_R_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);*

To store updated R_Vector with Load_Case_Label.

(6) *storeWhole_r_VectWithLoad_Case_Label (aLoad_Case_Label, r_Vector);*

To store updated r_Vector with Load_Case_Label.

(7) *storeWhole_Y_VectWithLoad_Case_Label (aLoad_Case_Label, Y_Vector);*

To store updated Y_Vector with Load_Case_Label.

(8) *storeWhole_y_VectWithLoad_Case_Label (aLoad_Case_Label, y_Vector);*

To store updated y_Vector with Load_Case_Label.

(9) *storeWhole_P_VectWithLoad_Case_Label (aLoad_Case_Label, P_Vector);*

To store updated P_Vector with Load_Case_Label.

(10) *storeWhole_p_VectWithLoad_Case_Label (aLoad_Case_Label, p_Vector);*

To store updated p_Vector with Load_Case_Label.

(11) *buildLoad_Case_Vectors (theSGT, theDOF_Manager);*

To construct and fill the Load_Case_Vector_Container. The algorithm is given in Chapter 6.

The following Load_Case_Manager functions are used by the ST object to retrieve information that is managed by *theVector_Manager.*

(12) *getWhole_R_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire R_Vector under a load case by indicating its label.

(13) *getWhole_r_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire r_Vector under a load case by indicating its label.

(14) *getWhole_Y_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire Y_Vector under a load case by indicating its label.

(15) *getWhole_y_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire y_Vector under a load case by indicating its label.

(16) *getWhole_P_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire P_Vector under a load case by indicating its label.

(17) *getWhole_p_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire p_Vector under a load case by indicating its label.

(18) *getDataFrom_R_VectAt (aLoad_Case_Label, anInteger);*

To retrieve an element of the R_Vector under a load case by specifying an index pointing to the position of the element in the R_Vector and the Load_Case_Label.

(19) *getDataFrom_r_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the r_Vector under a load case by specifying an index pointing to the position of the element in the r_Vector and the Load_Case_Label.

(20) *getDataFrom_Y_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the Y_Vector under a load case by specifying an index pointing to the position of the element in the Y_Vector and the Load_Case_Label.

(21) *getDataFrom_y_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the y_Vector under a load case by specifying an index pointing to the position of the element in the y_Vector and the Load_Case_Label.

(22) *getDataFrom_P_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the P_Vector under a load case by specifying an index pointing to the position of the element in the P_Vector and the Load_Case_Label.

(23) *getDataFrom_p_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the p_Vector under a load case by specifying an index pointing to the position of the element in the p_Vector and the Load_Case_Label.

The following Load_Case_Manager functions are used by the ST object to send and retrieve information that is managed by *theLoad_Manager.* These functions are used by the ST object as shown in Chapter 6.

(24) *storeLoad (aLoad_Case_Label, aLoad_Label, aLoad);*

To store a Load object by its key (aLoad_Label) and its value (aLoad).

(25) *createLoad_Case (aLoad_Case_Label);*

To create a Load_Case object by its key (aLoad_Case_Label).

(26) *getLoadWithLabel (aLoad_Case_Label, aLoad_Label);*

The function is used by ST to calculate Element_Internal_Forces.

Other Functions:

The following Load_Case_Manager function is used to process the loads within an Element or SE object into contributions to the force vectors for the load case. The algorithm is presented in Chapter 6.

(27) *processElemLoad (theElement, theLoad, theLoad_Case_Label, theSGT, theDOF_ Manager);*

To process an Element or SE load into contributions to the force vectors, given the element, a Load object, and a Load_Case_Label, and theSGT, and theDOF_ Manager for theST.

5.3.5 Load_Manager Objects

A *Load_Manager* object manages multiple load cases acting on the substructure. More than one load case is usually considered in a structural analysis, so a Load_Manager object includes a Load_Case_Container object (an instance of the KeySortCltn class) to store load cases. A Load_Manager object contains the following instance variables and functions (Fig. 5.18).

Instance variables:

(1) *theLoad_Case_Container*, an instance of the KeySortCltn class, works as a container

to hold multiple Load_Cases in the substructure. It holds associations of Load_Case_Labels with Load_Case objects (Fig. 5.19).

Public functions:

(1) *Load_Manager (anotherLoad_Manager);*

To construct a Load_Manager object by copying another Load_Manager object.

Load_Manager ( );

To construct an empty Load_Manager object.

(2) *getNumberOfLoadCases ( );*

This function is used by the Load_Case_Manager object to obtain the number of load cases.

(3) *getLoad_CaseWithLabel (aLoad_Case_Label);*

This function is used to obtain a Load_Case object by its label.

(4) getLoad_CaseAt (anInteger);

This function is used by the Load_Case_Manager object to obtain a Load_Case object by indicating its position in the Load_Case_Container.

(5) *createLoad_Case (aLoad_Case_Label);*

This function is used by the Load_Case_Manager object to create a new Load_Case object in the Load_Case_Container.

(6) *getLoad_Case_LabelAt (anInteger);*

This function is used by the Load_Case_Manager object to retrieve a Load_Case_Label by indicating its position in the Load_Case_Container.

(7) *storeLoad (aLoad_Case_Label, aLoad_Label, aLoad);*

This function is used by the Load_Case_Manager object to store a Load object. The Load_Manager stores the Load object in the correct Load_Case.

(8) *getNumberOfLoads (aLoad_Case_Label);*

This function is used by the Load_Case_Manager object to obtain the number of load objects in a Load_Case object. Using the Load_Case_Label, a particular Load_Case can be retrieved and the number of Load objects in the Load_Case can be determined.

(9) *getLoadWithLabel (aLoad_Case_Label, aLoad_Label);*

This function is used to retrieve a Load object from a particular Load_Case. Using the Load_Case_Label, the Load_Case object can be retrieved from the Load_Case_Container, and the Load object can be retrieved by its label from the Load_Case object.

(10) *getLoadAt (aLoad_Case_Label, anInteger);*

This function is used by the Load_Case_Manager object to retrieve a Load object from a particular Load_Case. Using the Load_Case_Label, the Load_Case object can be retrieved from the Load_Case_Container, and the Load object can be retrieved by indicating its position in the Load_Case.

(11) *copyLoad_Case (Load_Case_Label_1, Load_Case_Label_2);*

This function is used to copy one load case, with Load_Case_Label_1, to a new load case, with Load_Case_Label_2. If the load case corresponding to Load_Case_Label_1 is undefined or does not exist (there is no Load_Case_1), the function makes a new Load_Case object that is empty (has no Load objects in it).

5.3.6 Load_Case Objects

A *Load_Case* object is a container to store a number of Load objects. For example, if a load case contains ten individual loads (e.g. ten concentrated loads), then the Load_Case object creates ten associations, each association includes the label of a Load object and the Load object (Fig. 5.20). A Load_Case object contains the following instance variables and functions (Fig. 5.21):

Instance Variables:

(1) *Load_Case_Label* is an instance of the String class of the NIH class library, and is an identifier for the Load_Case object.

(2) *theLoad_Container* is an instance of the KeySortCltn class of the NIH class library. The container holds all the Load objects acting on the substructure under a load case.

Public functions:

(1) *Load_Case (aLoad_Case_Label);*

To construct a Load_Case object by providing a Load_Case_Label.

Load_Case (anotherLoad_Case);

To construct a Load_Case object by copying another Load_Case object.

(2) *getLoad_Case_Label ( );*

This function is used by the Load_Manager object to obtain the Load_Case_Label.

(3) *getNumberOfLoads ( );*

This function is used by the Load_Manager object to obtain the number of Load objects held in theLoad_Container.

(4) *getLoadWithLabel (aLoad_Label);*

This function is used by the Load_Manager object to retrieve a Load object by label.

(5) *getLoadAt (anInteger);*

This function is used by the Load_Manager object to retrieve a Load object by indicating its position in theLoad_Container.

(6) *storeLoad (aLoad_Label, aLoad);*

This function is used by the Load_Manager object to store a Load object in theLoad_Container using its label.

(7) *removeLoad (aLoad_Label);*

This function is used by the Load_Manager object to remove a Load object from theLoad_Container by indicating its label.

5.3.7 Load Objects

A *Load* object represents a single load. Three types of load objects are considered: Node_Loads, which are loads acting directly on nodes, Element_Loads, which are distributed loads acting on elements, and SE_Loads, which are initial boundary forces for superelements, required to equilibrate forces applied within superelements and restrain the superelement boundary displacements. A SE_Load corresponds to a Load_Case defined for the lower level ST from which the SE is developed. Fig 5.22 shows the three load types. The three load types (Fig. 5.23) are discussed in the following three sections.

5.3.7.1 Node_Load Objects

A Node_Load object contains the following instance variables and functions.

Instance variables:

(1) *Node_Label* is an instance of the String class of the NIH class library. It is both the identifier of the Node object which is subjected to the Node_Load, and the identifier of the Node_Load object.

(2) F_x, F_y, F_z, M_x, M_y, *and* M_z represent the six components acting on the six DOF of the Node. If all of the DOF of the Node are "r" type, or "p" type, then F_x, F_y, and F_z are three forces and M_x, M_y, and M_z are three moments acting on the Node in the direction of the substructure coordinate axes. If any of the DOF are "y" type, then the corresponding component is taken as the specified displacement for that DOF.

Public functions:

(1) *Node_Load (aNode_Label, F_x, F_y, F_z, M_x, M_y, M_z);*

To construct a Node_Load object by providing a Node_Label and six components.

Node_Load (anotherNode_Load);

To construct a Node_Load object by copying another Node_Load.

(2) *getLabel ( );*

The function is used by the Load_Case object to obtain the Node_Label for the load.

(3) *getComponent (aninteger);*

The function is used to obtain the magnitude of a force or a specified displacement component by indicating an integer (1-6).

5.3.7.2 Element_Load Objects

A Element_Load contains the following instance variables and functions.

Instance variables:

(1) *Element_Label* is the identifier of both the Element_Load and the Element object on which it acts.

(2) w_x, w_y, w_z, m_x, m_y and m_z represent the force and moment magnitudes acting on the span of the Element object.

Public functions:

(1) *Element_Load (anElement_Label, w_x, w_y, w_z, m_x, m_y, m_z);*

To construct an Element_Load object by providing an Element_Label and six span loads.

Element_Load (anotherElement_Load);

To construct an Element_Load object by copying another Element_Load.

(2) *getLabel ( );*

The function is used by the Load_Case object to obtain the Element_Label for the load.

(3) *getForce (aninteger);*

The function is used to obtain the magnitude of a force component by indicating an integer (1-6).

5.3.7.3 SE_Load Objects

A SE_Load contains the following instance variables and functions.

Instance variables:

(1) *SE_Label* is the identifier of both the SE_Load and the SE object on which it acts. It indicates the SE that produces the SE_Load for the higher level ST.

(2) *Lower_Level_Load_Case_Label* indicates the Load_Case in the lower level ST from which the SE_Load is obtained.

Public functions:

(1) *SE_Load (aSE_Label, aLower_Level_ Load _Case_Label);*

To construct a SE_Load object by providing aSE_Label as well as the label of the Load_Case in the lower level ST that produces the SE_Load.

SE_Load (anotherSE_Load);

To construct a SE_Load by copying another SE_Load.

(2) *getLabel ( );*

This function is used by the Load_Case object to obtain the SE_Label for the load.

(3) *get_Lower_Level_Load_Case_Label ( );*

This function is used to retrieve the label of the lower level Load_Case from which the SE_Load is calculated. The purpose of this function is to identify a lower level Load_Case (stored in the ST corresponding to the SET from which the SE is developed). The initial boundary forces for this load case are taken as the SE_Load.

5.3.8 Vector_Manager Objects

A *Vector_Manager* object stores multiple *Load_Case_Vector* objects, each of which corresponds to a *Load_Case* object. A Vector_Manager object includes an instance of the KeySortCltn class of the NIH class library to store *Load_Case_Vector* objects (Fig. 5.24).

Instance variable:

(1) *theLoad_Case_Vector_Container* is an instance of KeySortCltn object, and holds associations between key objects, which are *Load_Case_Label* objects, and value objects, which are *Load_Case_Vector* objects, each Load_Case_Vector object includes six vectors (**R**, **r**, **Y**, **y**, **P**, **p**), and corresponds to a Load_Case object.

Public functions:

(1) *Vector_Manager (r_number, y_number, p_number);*

To construct a Vector_Manager object using the number of "r" type DOF, "y" type DOF, and "p" type DOF.

Vector_Manager (anotherVM);

To construct a Vector_Manager by copying another Vector_Manager object.

(2) *getLoad_Case_VectorWithLabel (aLoad_Case_Label);*

This function is used by the Load_Case_Manager object to get a Load_Case_Vector by its Load_Case_Label.

(3) *getLoad_Case_VectorAt (anInteger);*

This function is used by the Load_Case_Manager object to get a Load_Case_Vector by indicating its position in the Load_Case_Vector_Container.

(4) *createLoad_Case_Vector (aLoad_Case_Label);*

This function is used by the Load_Case_Manager object to create a

Load_Case_Vector object, and store the object in the Load_Case_Vector_Container.

(5) *copyLoad_Case (Load_Case_Label_1, Load_Case_Label_2);*

This function is used to copy one load case, with Load_Case_Label_1, to a new load case, with Load_Case_Label_2. If the load case corresponding to Load_Case_Label_1 is undefined or does not exist there is no Load_Case_1), the function makes a new Load_Case object that is empty (has no Load objects in it).

The following *Vector_Manager* functions are used by the Load_Case_Manager to send information that is managed by the *Load_Case_Vector* objects.

(6) *storeWhole_R_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);*

To store updated R_Vector with Load_Case_Label.

(7) *storeWhole_r_VectWithLoad_Case_Label (aLoad_Case_Label, r_Vector);*

To store updated r_Vector with Load_Case_Label.

(8) *storeWhole_Y_VectWithLoad_Case_Label (aLoad_Case_Label, Y_Vector);*

To store updated Y_Vector with Load_Case_Label.

(9) *storeWhole_y_VectWithLoad_Case_Label (aLoad_Case_Label, y_Vector);*

To store updated y_Vector with Load_Case_Label.

(10) *storeWhole_P_VectWithLoad_Case_Label (aLoad_Case_Label, P_Vector);*

To store updated P_Vector with Load_Case_Label.

(11) *storeWhole_p_VectWithLoad_Case_Label (aLoad_Case_Label, p_Vector);*

To store updated p_Vector with Load_Case_Label.

The following Vector_Manager functions are used by the Load_Case_Manager object to retrieve information that is managed by the *Load_Case_Vector* objects.

(12) *getWhole_R_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire R_Vector under a load case by indicating its label.

(13) *getWhole_r_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire r_Vector under a load case by indicating its label.

(14) *getWhole_Y_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire Y_Vector under a load case by indicating its label.

(15) *getWhole_y_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire y_Vector under a load case by indicating its label.

(16) *getWhole_P_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire P_Vector under a load case by indicating its label.

(17) *getWhole_p_VectByLoad_Case_Label (aLoad_Case_Label);*

To retrieve an entire p_Vector under a load case by indicating its label.

(18) *getDataFrom_R_VectAt (aLoad_Case_Label, anInteger);*

To retrieve an element of the R_Vector under a load case by specifying an index pointing to the position of the element in the R_Vector and the Load_Case_Label.

(19) *getDataFrom_r_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the r_Vector under a load case by specifying an index pointing to the position of the element in the r_Vector and the Load_Case_Label.

(20) *getDataFrom_Y_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the Y_Vector under a load case by specifying an index pointing to the position of the element in the Y_Vector and the Load_Case_Label.

(21) *getDataFrom_y_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the y_Vector under a load case by specifying an index pointing to the position of the element in the y_Vector and the Load_Case_Label.

(22) *getDataFrom_P_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the P_Vector under a load case by specifying an index pointing to the position of the element in the P_Vector and the Load_Case_Label.

(23) *getDataFrom_p_VectAt (aLoad_Case_Label, anIndex);*

To retrieve an element of the p_Vector under a load case by specifying an index pointing to the position of the element in the p_Vector and the Load_Case_Label.

The following Vector_Manager functions are used by the Load_Case_Manager object to modify the force and displacement vectors:

(24) *addDataTo_R_Vector (aLoad_Case_Label, aDOFIndex, ForceValue);*

To add data to an element of the R_Vector under a load case by indicating the Load_Case_Label and a position in the vector.

(25) *addDataTo_r_Vector (aLoad_Case_Label, aDOFIndex, DisplacementValue);*

To add data to an element of the r_Vector under a load case by indicating the Load_Case_Label and a position in the vector.

(26) *addDataTo_Y_Vector (aLoad_Case_Label, aDOFIndex, ForceValue);*

To add data to an element of the Y_Vector under a load case by indicating the Load_Case_Label and a position in the vector.

(27) *addDataTo_y_Vector (aLoad_Case_Label, aDOFIndex, DisplacementValue);*

To add data to an element of the y_Vector under a load case by indicating the Load_Case_Label and a position in the vector.

(28) *addDataTo_P_Vector (aLoad_Case_Label, aDOFIndex, ForceValue);*

To add data to an element of the P_Vector under a load case by indicating the Load_Case_Label and a position in the vector.

(29) *addDataTo_p_Vector (aLoad_Case_label, aDOFIndex, DisplacementValue);*

To add data to an element of the p_Vector under a load case by indicating the Load_Case_Label and a position in the vector.

5.3.9 Load_Case_Vector Objects

A *Load_Case_Vector* object corresponds to a Load_Case object and contains six vectors:

- The **R** vector stores known forces acting on "r" type DOF;
- The **r** vector stores displacements of "r" type DOF;
- The **Y** vector stores reaction forces acting on "y" type DOF;
- The **y** vector stores known displacements of "y" type DOF;
- The **P** vector stores forces acting on "p" type DOF;
- The **p** vector stores displacements of "p" type DOF.

A Load_Case_Vector object contains the following instance variables and functions (Fig. 5.25).

Instance variables:

(1) *Load_Case_Label* is an instance of the String class of the NIH class library. This label is the identifier of the Load_Case_Vector and the corresponding Load_Case object.

(2) *R_Vector* stores known forces acting on "r" type DOF.

(3) *r_Vector* stores displacements of "r" type DOF.

(4) *Y_Vector* stores reaction forces acting on "y" type DOF.

(5) *y_Vector* stores known displacements of "y" type DOF.

(6) *P_Vector* stores forces acting on "p" type DOF.

(7) *p_Vector* stores displacements of "p" type DOF.

Public functions:

(1) *Load_Case_Vector (r_number, y_number, p_number);*

To construct a Load_Case_Vector object by specifying the number of "r" type, "y" type, and "p" type DOF.

Load_Case_Vector (anotherLCV);

To construct a Load_Case_Vector by copying another Load_Case_Vector object.

The following Load_Case_Vector functions are used by the Vector_Manager object to send information to a Load_Case_Vector object:

(2) *storeWhole_R_Vect (R_Vector);*

To store updated R_Vector.

(3) *storeWhole_r_Vect (r_Vector);*

To store updated r_Vector.

(4) *storeWhole_Y_Vect (Y_Vector);*

To store updated Y_Vector.

(5) *storeWhole_y_Vect (y_Vector);*

To store updated y_Vector.

(6) *storeWhole_P_Vect (P_Vector);*

To store updated P_Vector.

(7) *storeWhole_p_Vect (p_Vector);*

To store updated p_Vector.

The following Load_Case_Vector functions are used by the Vector_Manager object to retrieve information from a Load_Case_Vector object:

(8) *getWhole_R_Vect ( );*

To retrieve an entire R_Vector.

(9) *getWhole_r_Vect ( );*

To retrieve an entire r_Vector.

(10) *getWhole_Y_Vect ( );*

To retrieve an entire Y_Vector.

(11) *getWhole_y_Vect ( );*

To retrieve an entire y_Vector.

(12) *getWhole_P_Vect ( );*

To retrieve an entire P_Vector.

(13) *getWhole_p_Vect ( );*

To retrieve an entire p_Vector.

(14) *getDataFrom_R_VectAt (anIndex);*

To retrieve an element of the R_Vector by specifying an index pointing to the position of the element in the R_Vector.

(15) *getDataFrom_r_VectAt (anIndex);*

To retrieve an element of the r_Vector by specifying an index pointing to the position of the element in the r_Vector.

(16) *getDataFrom_Y_VectAt (anIndex);*

To retrieve an element of the Y_Vector by specifying an index pointing to the position of the element in the Y_Vector.

(17) *getDataFrom_y_VectAt (anIndex);*

To retrieve an element of the y_Vector by specifying an index pointing to the position of the element in the y_Vector.

(18) *getDataFrom_P_VectAt (anIndex);*

To retrieve an element of the P_Vector by specifying an index pointing to the position of the element in the P_Vector.

(19) *getDataFrom_p_VectAt (anIndex);*

To retrieve an element of the p_Vector by specifying an index pointing to the position of the element in the p_Vector.

The following Load_Case_Vector functions are used by the Vector_Manager object to modify force and displacement vectors:

(20) *addDataTo_R_Vector (anIndex, ForceValue);*

To add data to an element of the R_Vector by indicating a position.

(21) *addDataTo_r_Vector (anIndex, DisplacementValue);*

To add data to an element of the r_Vector by indicating a position.

(22) *addDataTo_Y_Vector (anIndex, ForceValue);*

To add data to an element of the Y_Vector by indicating a position.

(23) *addDataTo_y_Vector (anIndex, DisplacementValue);*

To add data to an element of the y_Vector by indicating a position.

(24) *addDataTo_P_Vector (anIndex, ForceValue);*

To add data to an element of the P_Vector by indicating a position.

(25) *addDataTo_p_Vector (anIndex, DisplacementValue);*

To add data to an element of the p_Vector by indicating a position.

5.3.10 R_Vector Objects

A *R_Vector* object stores known forces acting on "r" type DOF. The size of R_Vector is determined by *r_number*, which is the number of "r" type DOF. To reduce the number of vectors that are managed by a Load_Case_Vector object, the R_Vector includes both external forces and initial forces (i.e. it includes $\mathbf{R}$-$\mathbf{R}_0$). The R_Vector is initialized to zero when it is constructed and is then filled by the Load_Case_Manager function *build_Load_Case_Vectors ( )* described in Chapter 6, which processes Node_Load objects (that contribute to **R**), and Element_Load and SE_Load objects (that contribute to $\mathbf{R}_0$).

Instance variables (Fig.5.26):

(1) *r_number* determines the dimension of the R_Vector.

(2) *R* is an array to store data.

Public functions:

(1) *R_Vector (r_number);*

To construct a R_Vector object by specifying the number of "r" type DOF.

R_Vector (another_R_Vector);

To construct a R_Vector object by copying another one.

(2) *addData (forceValue, aDOF_Index);*

This function is used by the Load_Case_Vector object to add data to an element of the R_Vector by specifying the position in the vector (aDOF_Index).

(3) *zeroData (aDOF_Index);*

This function is used by the Load_Case_Vector object to zero an element of the R_Vector by specifying the position in the vector (aDOF_Index).

(4) *getData (aDOF_Index);*

This function is used by the Load_Case_Vector object to return the data from an element of the R_Vector by specifying the position in the vector (aDOF_Index).

5.3.11 r_Vector Objects

A r_Vector stores the displacements determined for the "r" type DOF. The size of r_Vector is determined by r_number. The r_Vector is initialized to zero when it is constructed, and its values are determined by the substructure analysis (Chapter 6).

Instance variables (Fig.5.27):

(1) *r_number* determines the dimension of the r_Vector.

(2) *r* is an array to store data.

Public functions:

(1) *r_Vector (r_number);*

To construct a r_Vector object by specifying the number of "r" type DOF.

r_Vector (another_r_number);

To construct a r_Vector object by copying another one.

(2) *addData (displacementValue, aDOF_Index);*

This function is used by the Load_Case_Vector object to add data to an element of the r_Vector by specifying the position in the vector (aDOF_Index).

(3) *zeroData (aDOF_Index);*

This function is used by the Load_Case_Vector object to zero an element of the r_Vector by specifying the position in the vector (aDOF_Index).

(4) *getData (aDOF_Index);*

This function is used by the Load_Case_Vector object to return the data from an element of the r_Vector by specifying the position in the vector (aDOF_Index).

5.3.12 Y_Vector Objects

A Y_Vector stores reaction forces on "y" type DOF. The size of Y_Vector is determined by y_number. The Y_Vector is initialized to zero when it is constructed, and then it is filled by the Load_Case_Manager function *build_Load_Case_Vectors ( )*, described in Chapter 6, which processes Element_Load objects and SE_Load objects (that contribute to $\mathbf{Y}_0$). The substructure analysis (Chapter 6) then determines additional reaction forces and adds these into the Y_Vector. To reduce the number of vectors that are managed by the Load_Case_Vector object, the Y_Vector includes the effect of initial forces directly at the beginning of the analysis (i.e. it includes $\mathbf{Y}_0$).

Instance variables (Fig.5.28):

(1) *y_number* determines the dimension of the R_Vector.

(2) *Y* is an array to store data.

Public functions:

(1) *Y_Vector (y_number);*

To construct a Y_Vector object by specifying the number of "y" type DOF.

Y_Vector (another_Y_Vector);

To construct a Y_Vector object by copying another one.

(2) *addData (forceValue, aDOF_Index);*

This function is used by the Load_Case_Vector object to add data to an element of the Y_Vector by specifying the position in the vector (aDOF_Index).

(3) *zeroData (aDOF_Index);*

This function is used by the Load_Case_Vector object to zero an element of the Y_Vector by specifying the position in the vector (aDOF_Index).

(4) *getData (aDOF_Index);*

This function is used by the Load_Case_Vector object to return the data from an element of the Y_Vector by specifying the position in the vector (aDOF_Index).

5.3.13 y_Vector Objects

A *y_Vector* object stores known displacements of "y" type DOF. The size y_Vector is determined by *y_number*.

Instance variables (Fig.5.29):

(1) *y_number* determines the dimension of the y_Vector.

(2) y is an array to store data.

Public functions:

(1) *y_Vector (y_number);*

To construct a y_Vector object by specifying the number of "y" type DOF.

y_Vector (another_y_number);

To construct a y_Vector object by copying another one.

(2) *addData (displacementValue, aDOF_Index);*

This function is used by the Load_Case_Vector object to add data to an element of the y_Vector by specifying the position in the vector (aDOF_Index).

(3) *zeroData (aDOF_Index);*

This function is used by the Load_Case_Vector object to zero an element of the y_Vector by specifying the position in the vector (aDOF_Index).

(4) *getData (aDOF_Index);*

This function is used by the Load_Case_Vector object to return the data from an element of the y_Vector by specifying the position in the vector (aDOF_Index).

5.3.14 P_Vector Objects

A P_Vector stores reaction forces on "p" type DOF. The size of P_Vector is determined by p_number. The P_Vector is initialized to zero when it is constructed, and then it is filled by the Load_Case_Manager function *build_Load_Case_Vectors ( )*, described in Chapter 6, which processes Element_Load objects and SE_Load objects (that contribute to $\mathbf{P}_0$). The substructure analysis (Chapter 6) then determines additional reaction forces and adds these into the P_Vector. To reduce the number of vectors that are managed by the Load_Case_Vector object, the P_Vector includes the effect of initial forces directly at the beginning of the analysis (i.e. it includes $\mathbf{P}_0$).

Instance variables (Fig. 5.30):

(1) *p_number* determines the dimension of the P_Vector.

(2) *P* is an array to store data.

Public functions:

(1) *P_Vector (p_number);*

To construct a P_Vector object by specifying the number of "p" type DOF.

P_Vector (another_P_Vector);

To construct a P_Vector object by copying another one.

(2) *addData (forceValue, aDOF_Index);*

This function is used by the Load_Case_Vector object to add data to an element of the P_Vector by specifying the position in the vector (aDOF_Index).

(3) *zeroData (aDOF_Index);*

This function is used by the Load_Case_Vector object to zero an element of the P_Vector by specifying the position in the vector (aDOF_Index).

(4) *getData (aDOF_Index);*

This function is used by the Load_Case_Vector object to return the data from an element of the P_Vector by specifying the position in the vector (aDOF_Index).

5.3.15 p_Vector Objects

A *p_Vector* object stores the displacements determined for "p" type DOF. The size of p_Vector is determined by p_number. As discussed in Chapter 3, when the current level substructure analysis is carried out (α case), the elements of p_Vector are assumed to be zero. The values in p_Vector are determined in a higher level substructure analysis.

Instance variables (Fig.5.31):

(1) *p_number* determines the dimension of the p_Vector.

(2) p is an array to store data.

Public functions:

(1) *p_Vector (p_number);*

To construct a p_Vector object by specifying the number of "p" type DOF.

p_Vector (another_p_number);

To construct a p_Vector object by copying another one.

(2) *addData (displacementValue, aDOF_Index);*

This function is used by the Load_Case_Vector object to add data to an element of the p_Vector by specifying the position in the vector (aDOF_Index).

(3) *zeroData (aDOF_Index);*

This function is used by the Load_Case_Vector object to zero an element of the p_Vector by specifying the position in the vector (aDOF_Index).

(4) *getData (aDOF_Index);*

This function is used by the Load_Case_Vector object to return the data from an element of the p_Vector by specifying the position in the vector (aDOF_Index).

5.3.16 Summary of ST Object And Objects Associated With ST Object

A SGT object describes a group of substructures with identical geometry and materials, and a ST represents substructures with identical geometry, materials, and loading. A ST works with several other objects to carry out a substructure analysis:

- *SGT* describes the substructure geometry and materials in terms of Node, Element, and SE objects;
- *DOF_Manager* stores and organizes the DOF;
- *Load_Case_Manager* stores and organizes loads acting on the ST, using a Load_Manager and a Vector_Manager;
- *Load_Manager* manages multiple Load_Case objects;
- *Load_Case* manages multiple Load objects;
- *Load* represents one load event;
- *Vector_Manager* manages multiple Load_Case_Vector objects;
- *Load_Case_Vector* stores a set of vectors (**R**, **r**, **Y**, **y**, **P**, and **p**) and corresponds to a Load_Case object;
- *R_Vector* stores known external forces acting on "r" type DOF;
- *r_Vector* stores displacements of "r" type DOF;
- *Y_Vector* stores reaction forces acting on "y" type DOF;

- *y_Vector* stores known displacements of "y" type DOF;
- *P_Vector* stores reaction forces acting on the boundary DOF ("p" type DOF);
- *p_Vector* stores displacements of "p" type DOF.

In Fig. 5.32, a diagram illustrating the organization of these objects is shown. According to their functions, these objects are divided into several levels. The lower level objects are designed to support the higher level objects.

5.3.17 SET Objects

Two substructures that have the same ST have a different *Superelement Type* (SET) if there is a dissimilarity in their boundary DOF. As shown in Fig. 5.6, the ground floor and the third floor have the same ST but a different SET because of their different boundary DOF. A SET object is differentiated from a ST by specifying the boundary DOF (i.e., specifying the "p" type DOF). This implies that the substructure analysis for an ST object will not be conducted until it is associated with on SET object. As discussed below, one or more SE objects can be developed from a SET.

A SET object contains the following instance variables and functions (Fig. 5.33)

Instance variables:

(1) *theST* is an instance of the ST class.

(2) *SET_Label* in an instance of the String class of the NIH class library, and is the identifier for the SET.

Public functions:

(1) *SET (SET_Label, aST);*

To construct a SET object by providing a SET_Label and a ST object with which the SET object is associated.

SET (anotherSET);

To construct a SET object by copying another SET object.

(2) *getNodeWithLabel (aNode_Label);*

A SE object uses this function to retrieve a Node object by indicating its label. The Node object is obtained from theST object.

(3) *establish_p_DOF ( );*

This function is used by the Global object to set the DOF type for the "p" type DOF from the initially assumed "r" type to "p" type. This function uses the corresponding function of the associated ST object (theST). The effect is that in theDOF_Manager in theST, three groups of DOF ("r" type, "y" type, and "p" type) are set up in the DOF_Container.

(4) *set_r_Type_Indices ( );*

The Global object uses this function to set the indices for "r" type DOF in the DOF_Container, held within theST.

(5) *getSET_Label ( );*

This function is used by a SE object to return the SET_Label.

(6) *getInitial_Forces (Lower_Level_Load_Case_Label);*

This function is used by a SE object to obtain the P_Vector (initial substructure boundary DOF forces) from theST object.

(7) *calculateInternalForces (p_Vect, aLower_Level_Load_Case_Label, Load_Case_Label);*

This function is used by a SE object to have the lower level ST object (theST) complete the substructure analysis using the boundary displacements in p_Vector. The displacements in p_Vect correspond to the load case identified by Load_Case_Label in the higher level substructure which contains the superelement. The Lower_Level_Load_Case_Label identifies the load case in the lower level substructure that corresponds to the load case identified by Load_Case_Label in the higher level substructure.

(8) *getStiffness_Matrix ( );*

A SE object uses this function to get the condensed stiffness matrix from theST.

(9) *getLowerLevelDOF_TypeWithLabel (aDOF_Label);*

A SE object uses this function to get the DOF_Type of a DOF from a boundary node that belongs to lower level ST associated with the SET from which the SET is developed.

(10) *getLowerLevelDOF_IndexWithLabel (aDOF_Label);*

A SE object uses this function to get the lower level DOF_index of a DOF that belongs to boundary node of the lower level substructure (theST) from which the SE is developed.

(11) *SubstructureAnalysis ( );*

This function is used by the Global object to carry out the first part of the substructure analysis (the α case). The algorithm is shown in Chapter 6.

(12) *get_p_number ( );*

This function is used by a SE object to obtain the size of the p_Vector for the lower level substructure from which the SE is developed (theST). This is used to create a new p_Vector to hold the boundary displacements determined by the higher level substructure. This p_Vector is then returned to the lower level substructure through the function *calculateInternalForces (p_Vect, aLower_Level_ Load_Case_Label, Load_Case_Label)* as shown in Chapter 6.

5.3.18 SE Objects

A *superelement* (SE) is developed from a SET. One SE object is needed for each substructure in the model. SEs are used in creating the next higher level substructure from lower level substructures.

A SE object contains the following instance variables and functions (Fig. 5.34):

Instance variables:

(1) *theSET* is a SET object and from which the SE object is developed.

(2) *SE_Label* is an instance of the String class of the NIH class library, and is the identifier of the SE.

(3) *Boundary_Node_Label_Array* is an instance of the KeySortCltn class of the NIH class library. The array holds associations of labels of the SE boundary nodes. Each association includes a key object, which is the label of the Node object that represents a node in the higher level substructure, with a value object, which is the label of the Node object that represents the same node in the lower level substructure. Both Node_Labels are assigned to a particular node, however, one is for the lower level ST and the other is for the higher level ST.

(4) *SE_Stiffness_Matrix* is the superelement stiffness matrix shown in Eq. 3.22. The matrix is initially obtained by the SE object from the ST object associated with theSET with respect to the superelement (lower level substructure) coordinate system. Then, the matrix is transformed to the higher level substructure coordinate system as shown in Chapter 6.

(5) *SE_Transformation_Matrix* is a 3-by-3 transformation matrix between the superelement (lower level substructure) coordinate system and the higher level substructure coordinate system.

(6) *SE_Initial_Forces* is a vector of forces required to restrain superelement displacements and equilibrate forces applied within the superelement (lower level substructure) coordinate system. The force are initially obtained by the SE object from the ST object associated with theSET with respect to the superelement (lower level substructure) coordinate system. Then, they are transformed to the higher level substructure coordinate system as shown in Chapter 6.

(7) *SE_Displacements* is a vector of displacements of the SE boundary DOF. The

displacements are calculated by a higher level substructure in the higher level substructure local coordinate system.

Public functions:

(1) *SE (SE_Label, theSET, Boundary_Node_Label_Array);*

To construct a SE object by providing a SE_Label, a SET object with which the SE object is associated and the labels of the boundary nodes, including labels for the higher level substructure and labels for the lower level substructure.

SE (anotherSE);

To construct a SE object by copying another SE object.

(2) *getLabel ( );*

The function is used to obtain the SE_Label.

(3) *formStiffness_Matrix ( );*

To form the SE stiffness matrix in the higher level substructure local coordinate system. The matrix is obtained from the lower level substructure (ST) associated with theSET and is transformed to the higher level substructure coordinate system. The transformation procedures are carried out by this function. The algorithm is presented in Chapter 6.

(4) *getStiffness_Coefficient (anInteger_i, anInteger_j);*

This function is used by a higher level ST to obtain a coefficient of the transformed superelement stiffness matrix.

(5) *getNumberOfNodes ( );*

This function is used by a higher level ST object to obtain the number of boundary nodes of the SE.

(6) *getLowerLevelNode_Label (anInteger);*

This function is used by a higher level ST object to obtain a lower level boundary

Node_Label through an index indicating the position of the Node_Label in the Boundary_Node_Label_Array.

(7) *getNode_Label (anInteger);*

This function is used by a higher level ST object to obtain a current level boundary Node_Label through an index indicating the position of the Node_Label in the Boundary_Node_Label_Array.

(8) *form_Initial_Forces (aSE_Load);*

To form the SE initial forces vector. The forces are initially given to the SE with respect to the superelement (lower level substructure) coordinate system by the lower level ST. The forces are transformed to the higher level substructure coordinate system by this function, as shown in Chapter 6.

(9) *get_Initial_Force_Coefficient (anInteger);*

To return a coefficient from the SE initial force vector by indicating the position of the coefficient in the vector.

(10) *calculateInternalForces (aSE_Load, aLoad_Case_Label);*

This function is used to calculate the SE internal forces given the boundary displacements in the vector Element_Displacements, and aSE_Load which specifies the lower level load case which corresponds to these displacements. This function is used by a higher level substructure (ST) after a higher level substructure analysis is completed.

(11) *addDataToDisplacements (anInteger, aValue);*

To add data to the SE_Displacements vector.

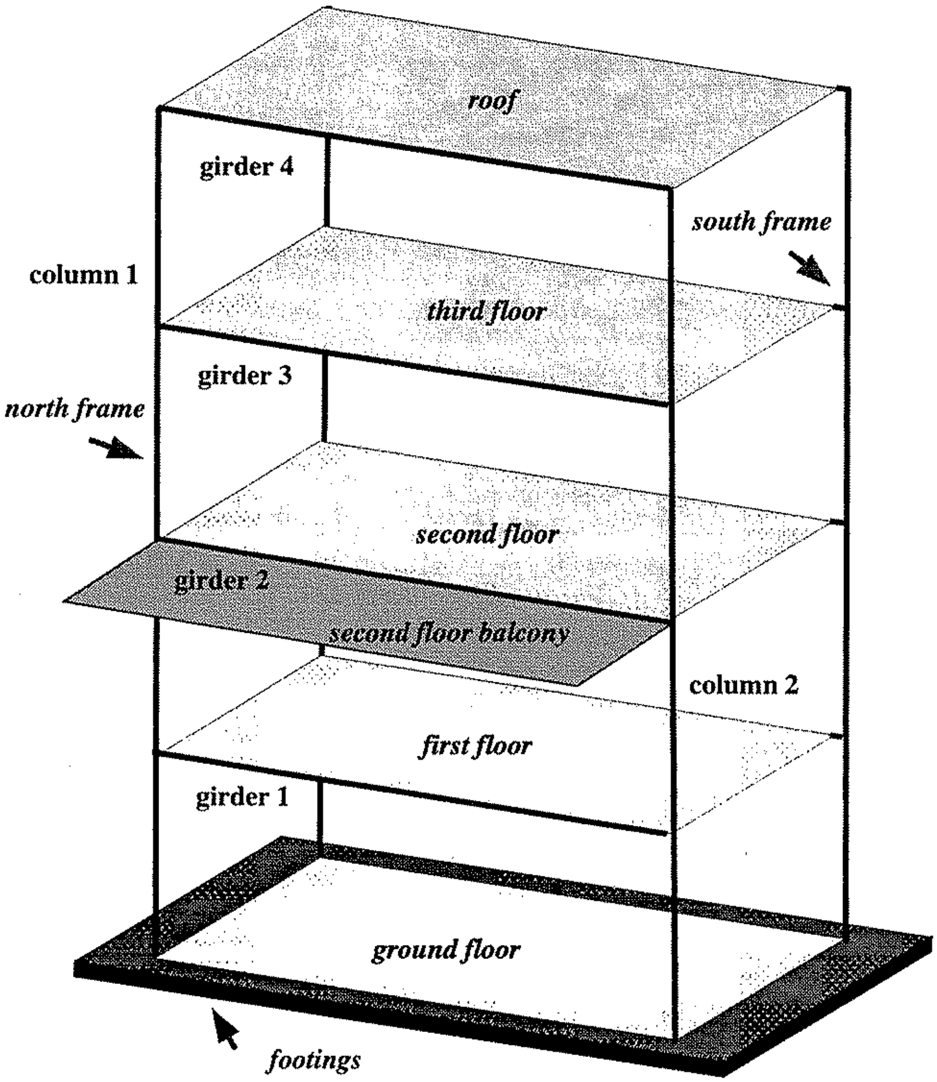
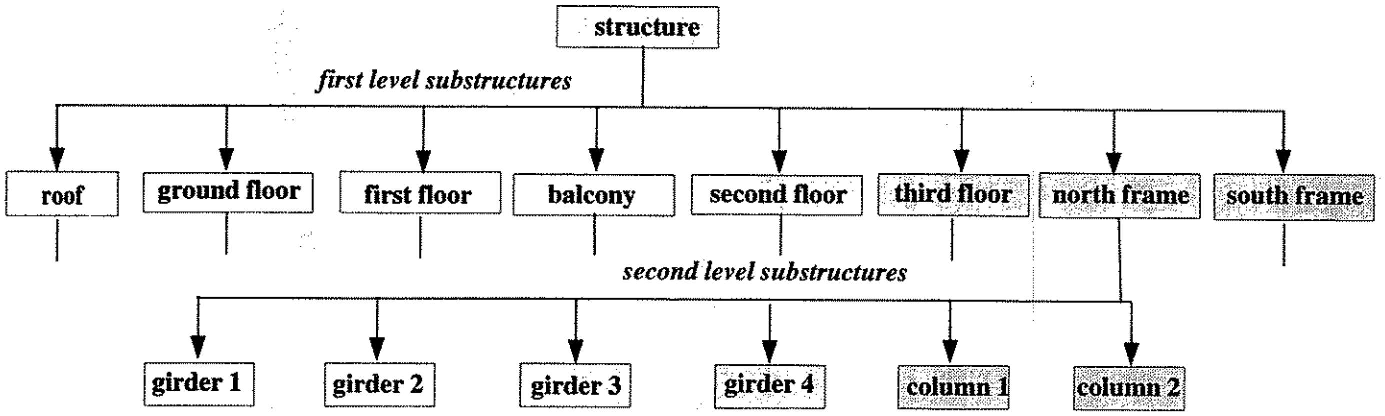


Fig. 5.1. Frame structure.

Structural Analysis Model



The arrows indicate that a higher level substructure can be divided into several lower level substructures

Fig. 5.2. Substructure subdivision hierarchy.

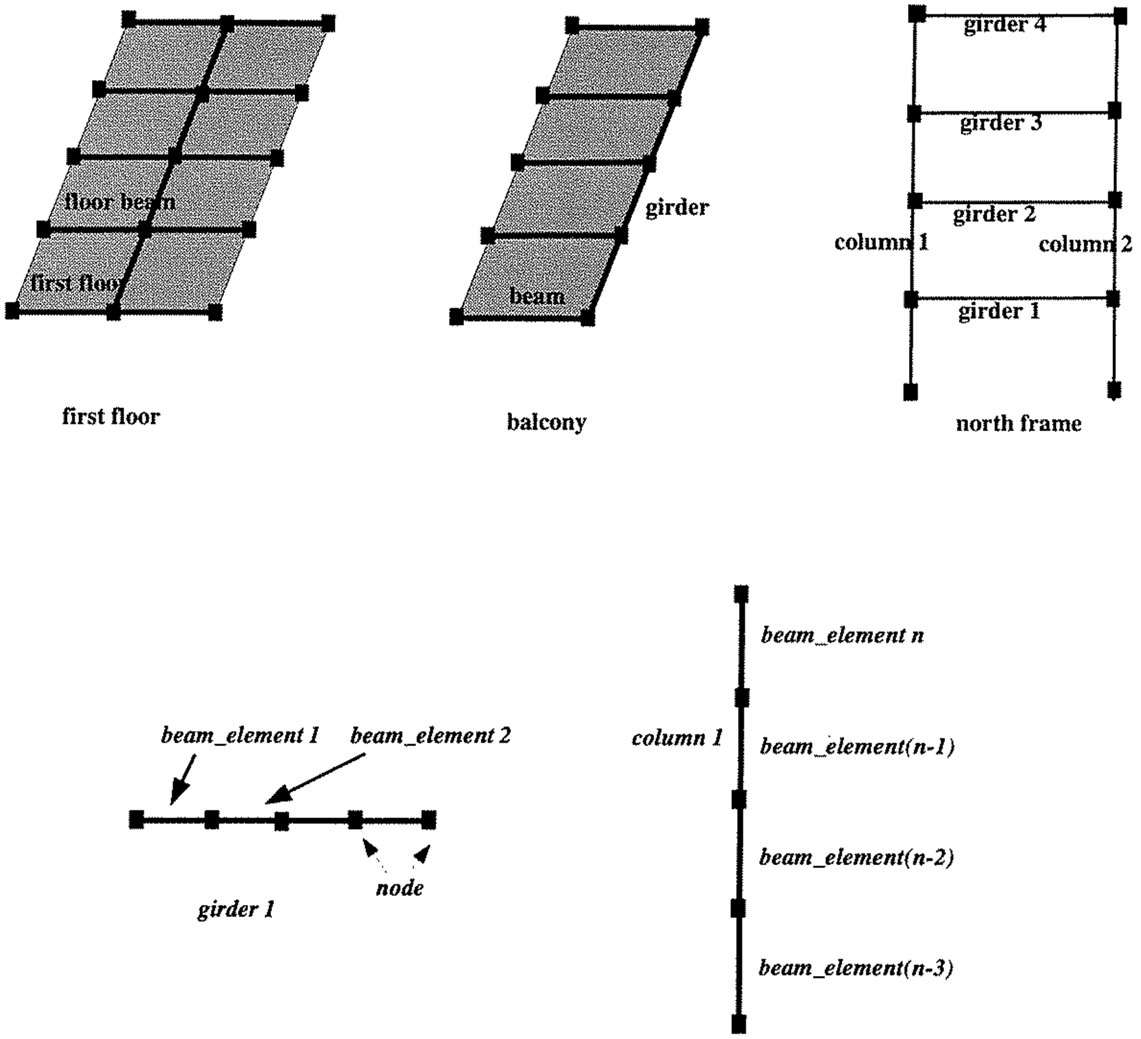


Fig. 5.3. Substructures for frame structure.

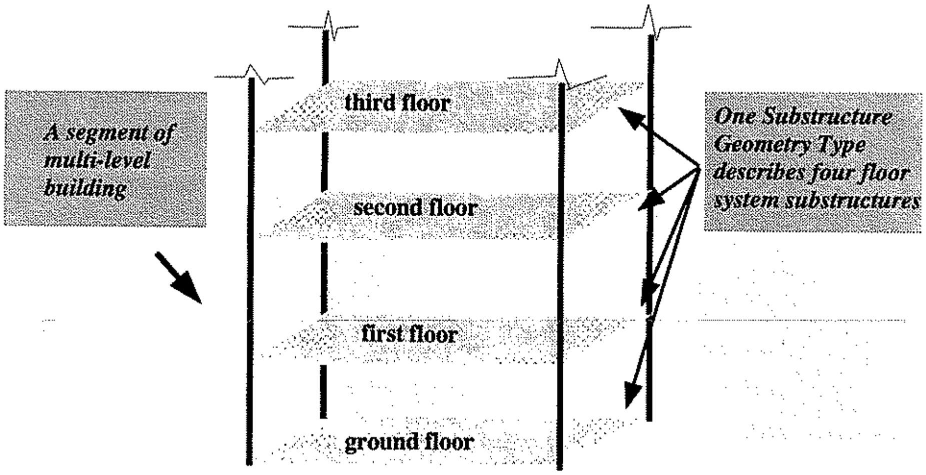

A segment of multi-level building
third floor
second floor
first floor
ground floor
One Substructure Geometry Type describes four floor system substructures

Fig. 5.4. Four substructures with identical geometry represented by one SGT.

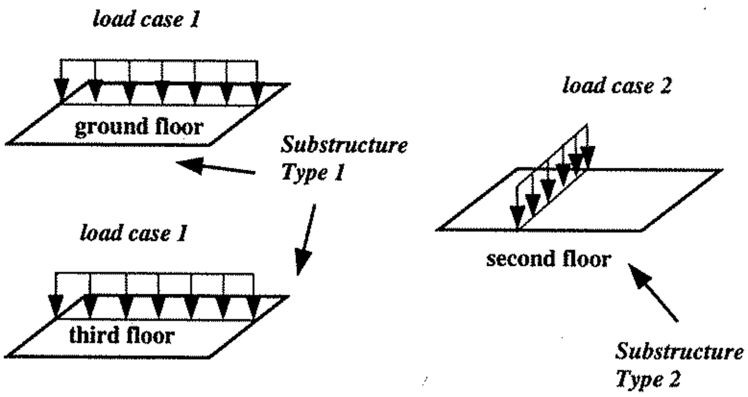


Fig. 5.5. Two Substructure Types (ST) developed from one Substructure Geometry Type (SGT).

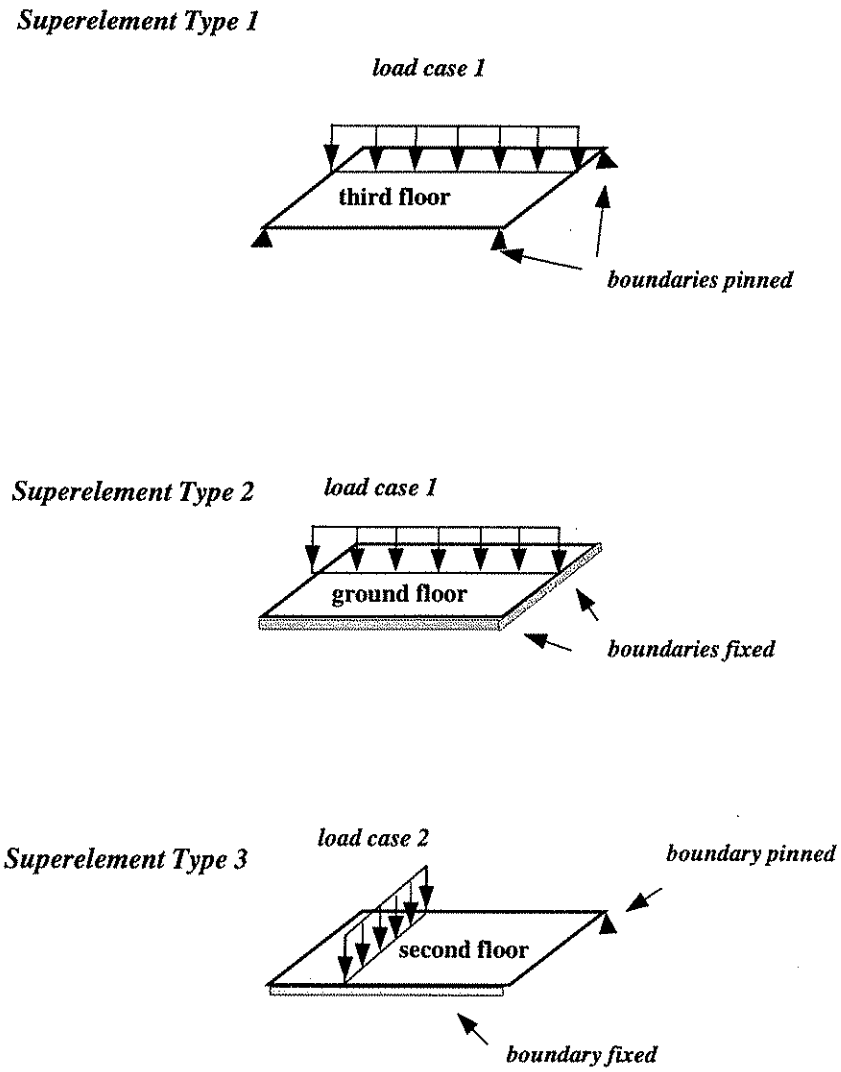
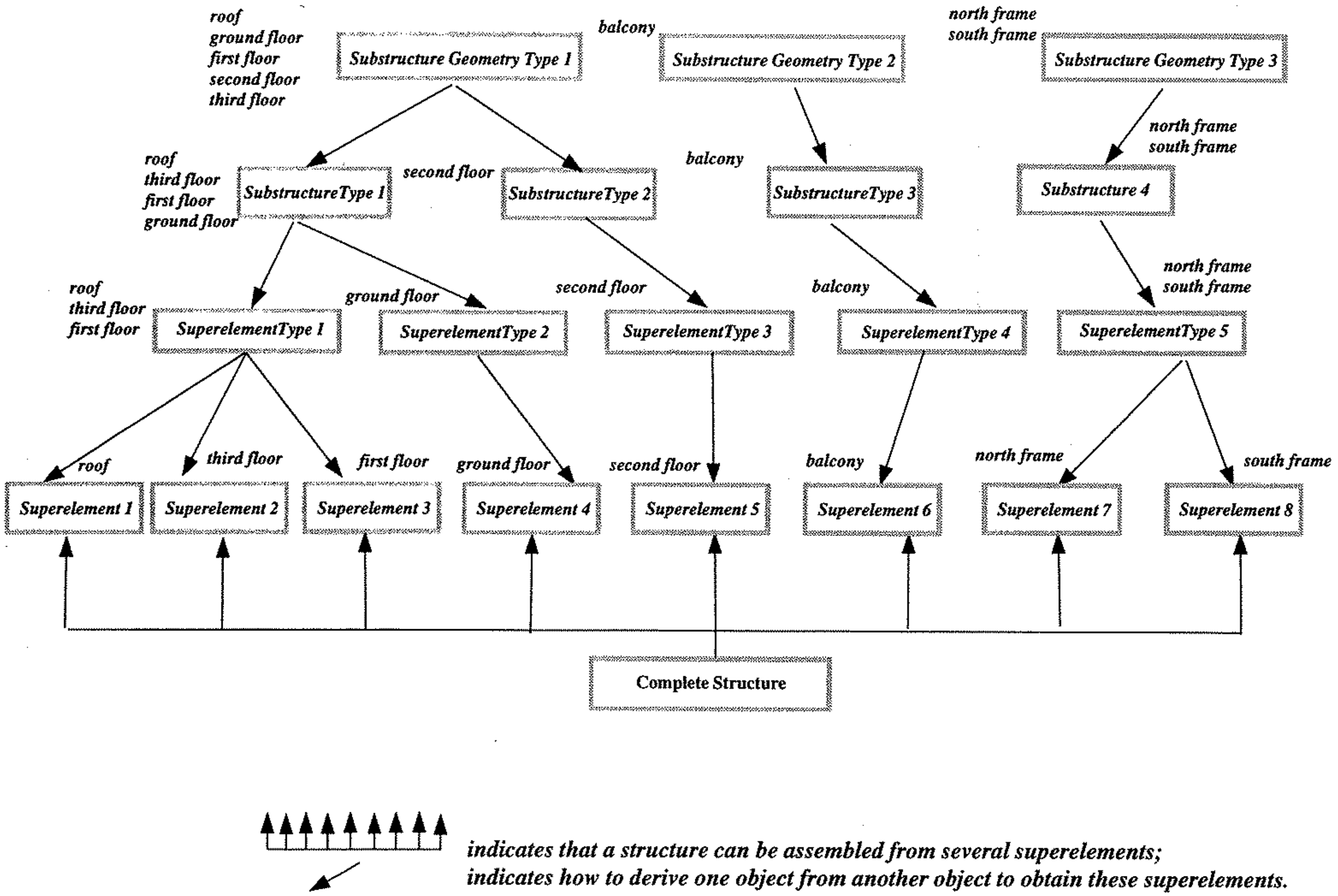


Fig. 5.6. Three Superelement Types (SET) developed from two Substructure Types (STs).



indicates that a structure can be assembled from several superelements;
indicates how to derive one object from another object to obtain these superelements.

Fig. 5.7. Object-oriented structural analysis model.

Instance Variables		
DOF_Label;	DOF_Type;	DOF_Index;

Public functions	

DOF (DOF_Label, DOF_Type, DOF_Index);
DOF (anotherDOF);
	getDOF_Label( );
	getDOF_Type( );
	getDOF_Index( );

setDOF_Type (newDOF_Type);
setDOF_Index (newDOF_Index);	

Fig. 5.8. DOF object.

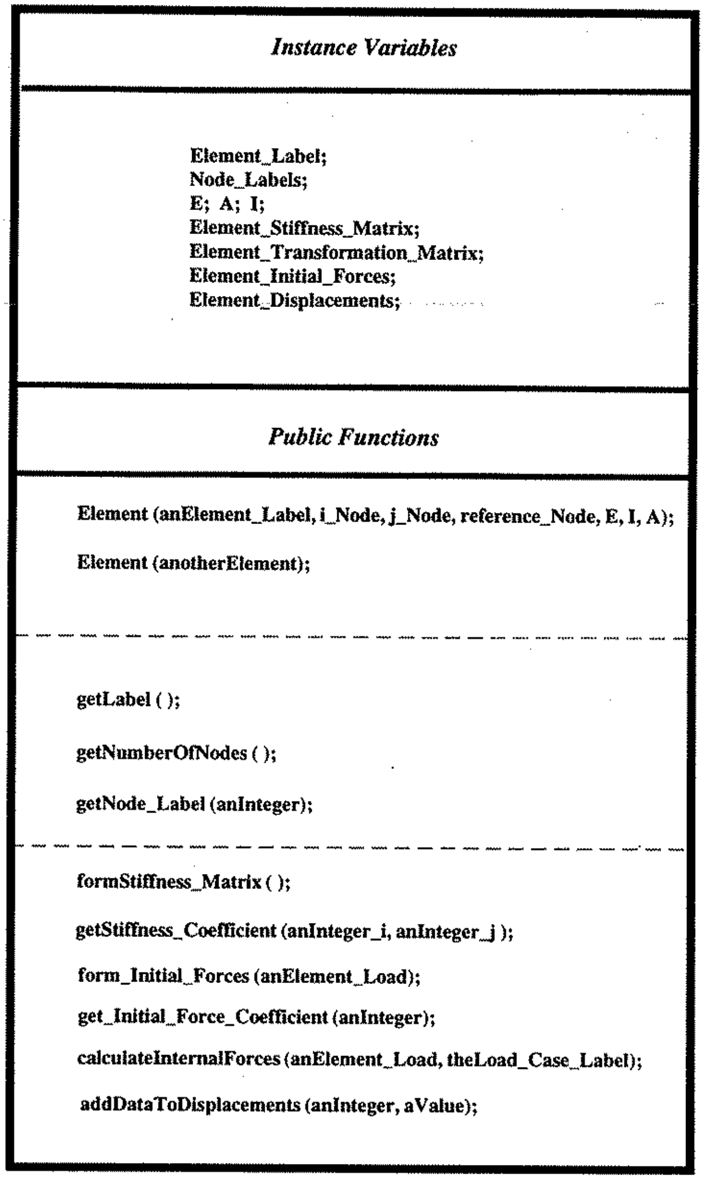
Instance Variables		
x_Coordinate;	y_Coordinate;	z_Coordinate;
Node_Label;	DOF_ Array;	

Public functions	

Node (Node_label, x_coordinate, y_coordinate, z_coordinate);
Node (anotherNode);	

getNode_Label ( ) const;
get_x ( ) const;
get_y ( ) const;
get_z ( ) const;
getDOF_LabelAt (anInteger) const;
	set_x (new_x);
	set_y (new_y);
	set_z (new_z);

Fig. 5.9. Node object.



Instance Variables

Element_Label;
Node_Labels;
E; A; I;
Element_Stiffness_Matrix;
Element_Transformation_Matrix;
Element_Initial_Forces;
Element_Displacements;

Public Functions

Element (anElement_Label, i_Node, j_Node, reference_Node, E, I, A);

Element (anotherElement);

getLabel ( );

getNumberOfNodes ( );

getNode_Label (anInteger);

formStiffness_Matrix ( );

getStiffness_Coefficient (anInteger_i, anInteger_j);

form_Initial_Forces (anElement_Load);

get_Initial_Force_Coefficient (anInteger);

calculateInternalForces (anElement_Load, theLoad_Case_Label);

addDataToDisplacements (anInteger, aValue);

Fig. 5.10. Element object.

Instance Variables

SGT_Label;

theNode_Container;

theElement_Container;

theSE_Container;

Public Functions

SGT (aSGT_Label);
SGT (anotherSGT);

getSGT_Label ( );

isTheNode_In (aNode_Label);
isTheElement_In (anElement_Label);
isTheSE_In (aSE_Label);

storeNode (aNode_object);
storeElement (anElement_object);
storeSE (aSE_object);

getNumberOfNodes ( );
getNumberOfElements ( );
getNumberOfSEs ( );

getNodeWithLabel (aNode_Label);
getElementWithLabel (anElement_Label);
getSE_WithLabel (aSE_Label);
getNode_At (anInteger);
getElement_At (anInteger);
getSE_At (anInteger);

Fig. 5.11. Substructure Geometry Type (SGT) object.

Instance Variables

ST_Label;
theSGT;
theDOF_Manager;
theLoad_Case_Manage;
SubstrucStiffness;

Public Functions

ST (ST_Label, aSGT);
ST (anotherST);

---

getST_Label ( );

getNodeWithLabel (aNode_label);

getStiffness_Matrix ( );

buildDOF_Container ( );

establish_y_DOF ( );

setDOF_TypeFrom_r_to_p (aDOF_Label);

set_r_TypeIndices ( );

getDOF_TypeWithLabel (aDOF_Labal);

getDOF_IndexWithLabel (aDOF_Labal);

buildLoad_Cases ( );

buildLoad_Case_Vectors ( );

getInitial_Forces (aCurrent_Level_Load_Case_Label);

SubstructureAnalysis ( );

LowerLevelSubstructureAnalysisCompletion (p_Vector,
Current_Level_Load_Case_Label, Higher_Level_Load_Case_Label);

HighestLevelSubstructureAnalysisCompletion ( );

get_p_number ( );

Other Functions

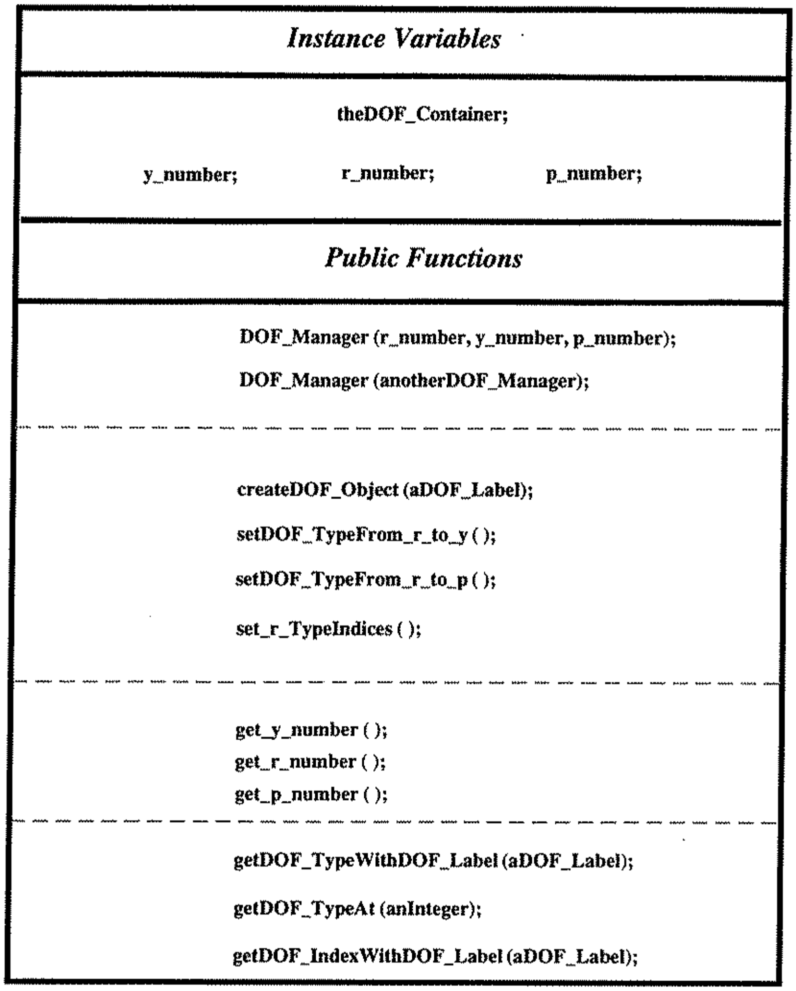
assembleStiffness_Matrix ( );

assembleElemStiffMatrix (anElement);

calculateElemInternalForces (theElement, theLoad_Case_Label);

getMatrixIndexWithDOF_Label (aDOF_Label);

Fig. 5.12. Substructure Type (ST) object.



Instance Variables

theDOF_Container;

y_number; r_number; p_number;

Public Functions

DOF_Manager (r_number, y_number, p_number);
DOF_Manager (anotherDOF_Manager);

createDOF_Object (aDOF_Label);
setDOF_TypeFrom_r_to_y ( );
setDOF_TypeFrom_r_to_p ( );
set_r_TypeIndices ( );

get_y_number ( );
get_r_number ( );
get_p_number ( );

getDOF_TypeWithDOF_Label (aDOF_Label);
getDOF_TypeAt (anInteger);
getDOF_IndexWithDOF_Label (aDOF_Label);

Fig. 5.13. DOF_Manager object.

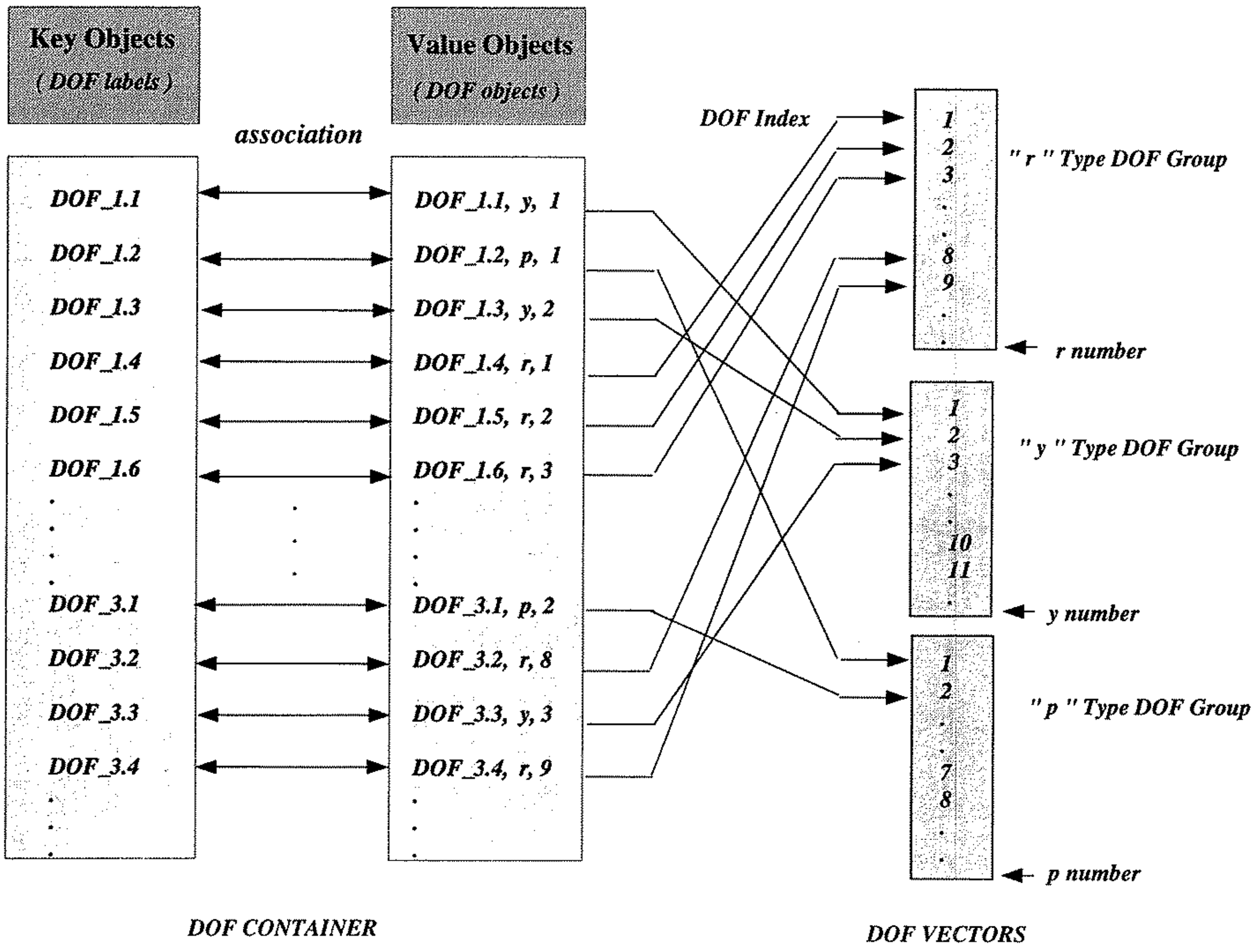


Fig. 5.14. DOF_Container and corresponding vectors.

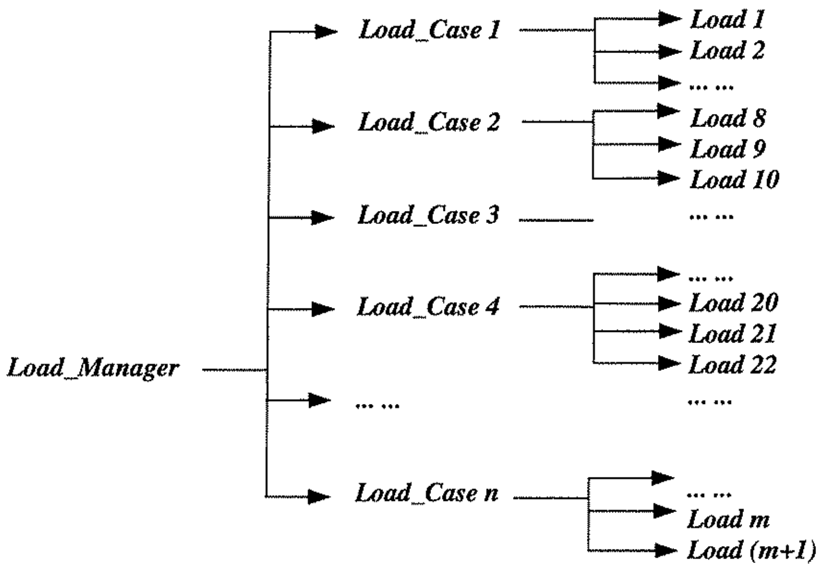


Fig. 5.15. Load management.

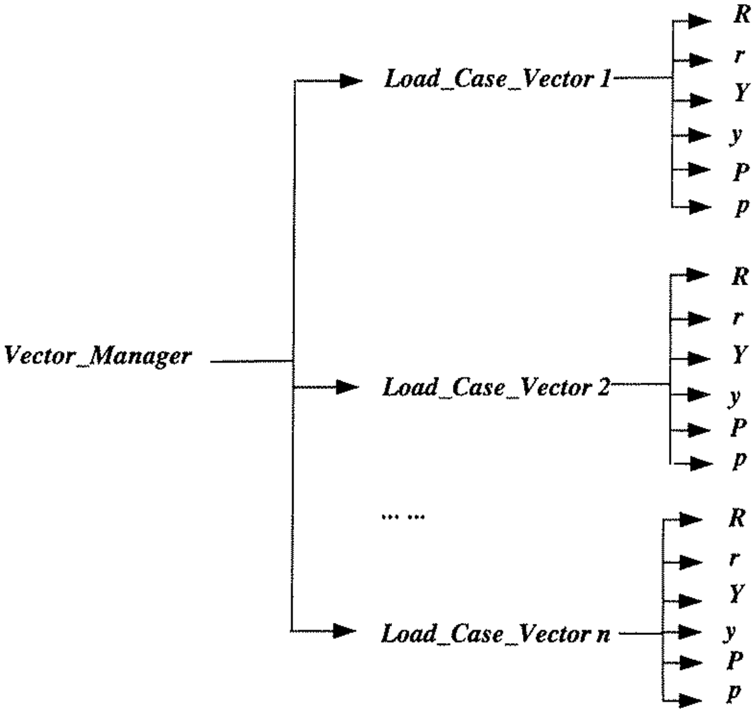


Fig. 5.16. Force and displacement vector management.

Instance Variables

theLoad_Manager;
theVector_Manager;

Public Functions

Load_Case_Manager (the_r_number, the_Y_number, the_p_number);
Load_Case_Manager (anotherLoad_Case_Manager);

getNumberOfLoadCase ( );
getLoad_Case_LabelAt (anInteger);
copyLoad_Case (Load_Case_Label_1, Load_Case_Label_2);

storeWhole_R_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);
storeWhole_r_VectWithLoad_Case_Label (aLoad_Case_Label, r_Vector);
storeWhole_Y_VectWithLoad_Case_Label (aLoad_Case_Label, Y_Vector);
storeWhole_y_VectWithLoad_Case_Label (aLoad_Case_Label, y_Vector);
storeWhole_P_VectWithLoad_Case_Label (aLoad_Case_Label, P_Vector);
storeWhole_p_VectWithLoad_Case_Label (aLoad_Case_Label, p_Vector);
buildLoad_Case_Vectors (theSGT, theDOF_Manager);

getWhole_R_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_r_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_Y_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_y_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_P_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_p_VectByLoad_Case_Label (aLoad_Case_Label);
getDataFrom_R_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_r_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_Y_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_y_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_P_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_p_VectAt (aLoad_Case_Label, anInteger);

storeLoad (aLoad_Case_Label, aLoad_Label, aLoad);
createLoad_Case (aLoad_Case_Label);
getLoadWithLabel (aLoad_Case_Label, aLoad_Label);

Other Functions

processElemLoad (theElement, theLoad, theLoad_Case_Label, theSGT
theDOF_Manager);

Fig. 5.17. Load_Case_Manager object.

Instance Variables

theLoad_Case_Container;

Public Functions

Load_Manager (anotherLoad_Manager);
Load_Manager ( );

getNumberOfLoadCase ( );
getLoad_CaseWithLabel (aLoad_Case_Label);
getLoad_CaseAt (anInteger);
createLoad_Case (aLoad_Case_Label);
getLoad_Case_LabelAt (anInteger);
storeLoad (aLoad_Case_Label, aLoad_Label, aLoad);
getNumberOfLoads (aLoad_Case_Label);
getLoadWithLabel (aLoad_Case_Label, aLoad_Label);
getLoadAt (aLoad_Case_Label, anInteger);
copyLoad_Case (Load_Case_Label_1, Load_Case_label_2);

Fig. 5.18. Load_Manager object.

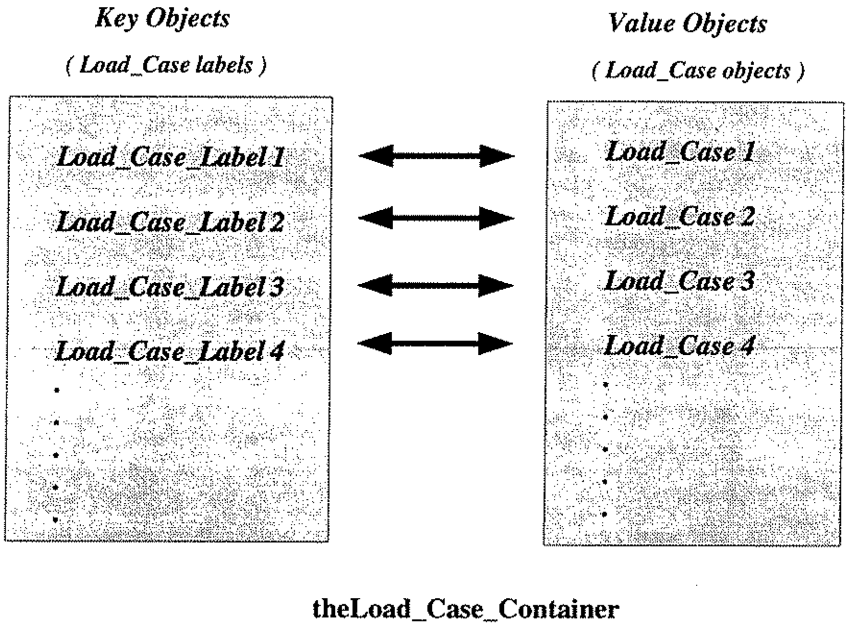


Fig. 5.19. Load_Case container object.

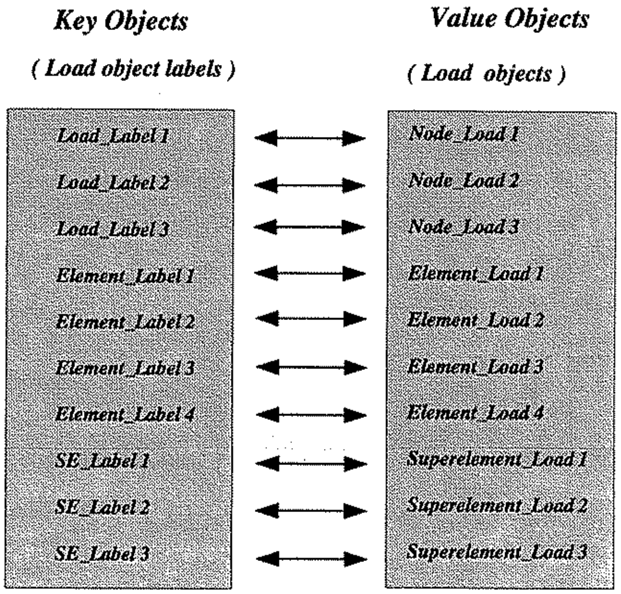


Fig. 5.20. Load_Container object.

Instance Variables

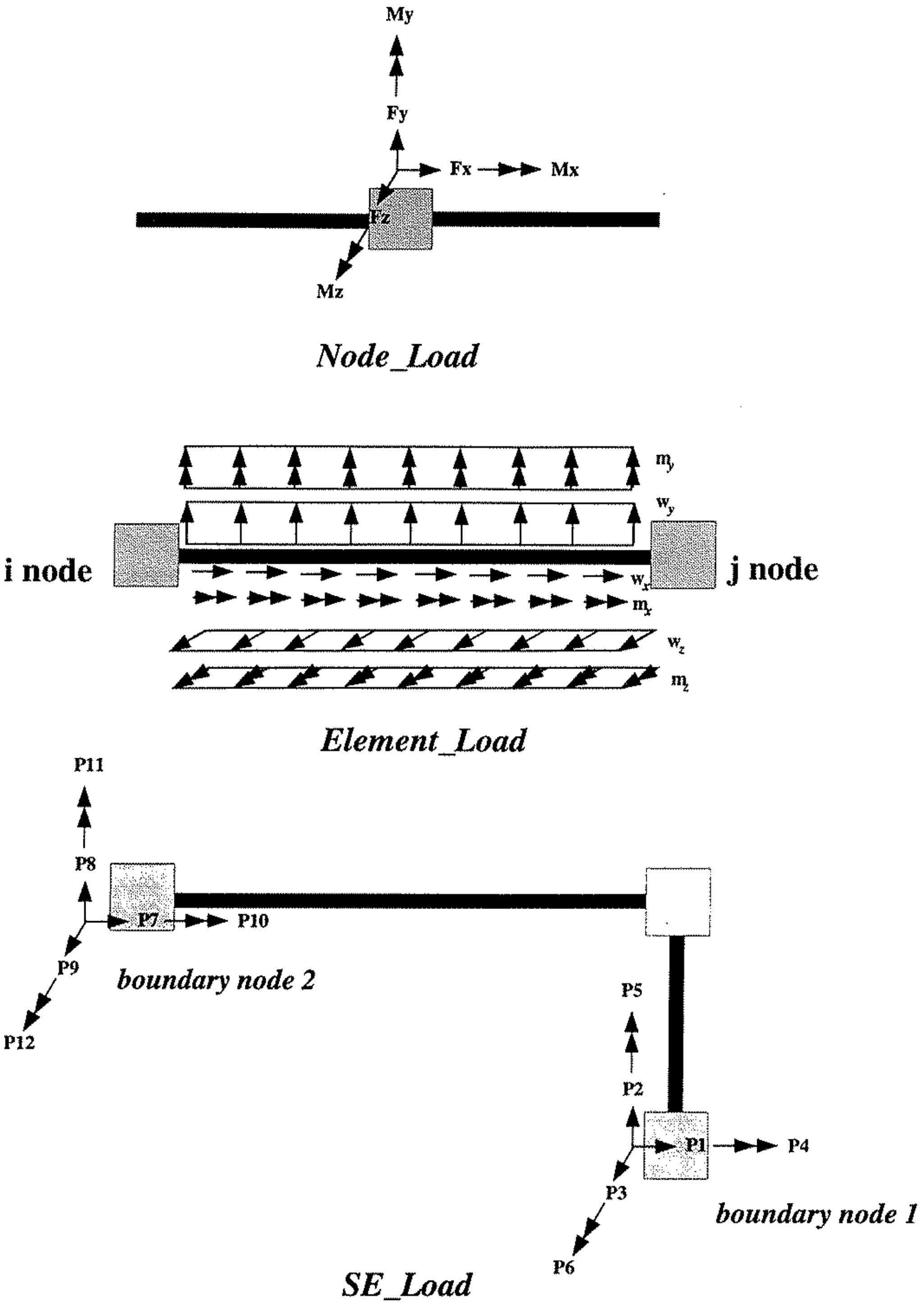
Load_Case_Label; theLoad_Container;

Public Functions

Load_Case (aLoad_Case_Label);
Load_Case (anotherLoad_Case);

getLoad_Case_Label ( );
getNumberOfLoads ( );
getLoadWithLabel (aLoad_Label);
getLoadAt (anInteger);
storeLoad (aLoad_Label, aLoad);
removeLoad (aLoad_Label);

Fig. 5.21. Load_Case object.

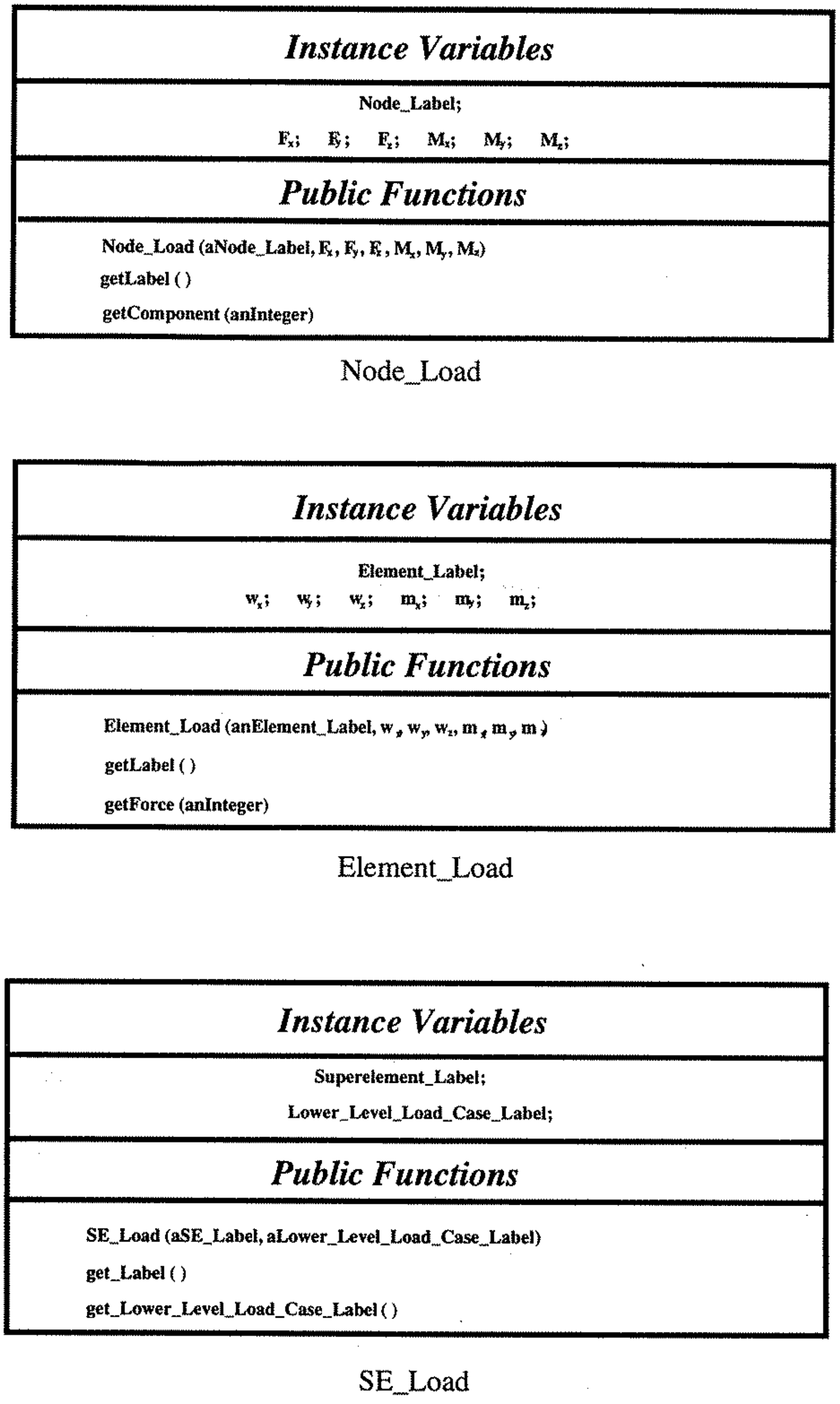


Node_Load

Element_Load

SE_Load

Fig. 5.22. Three types of loads.



Instance Variables

Node_Label;

F_x; F_y; F_z; M_x; M_y; M_z;

Public Functions

Node_Load (aNode_Label, F_x, F_y, F_z, M_x, M_y, M_z)
getLabel ( )
getComponent (anInteger)

Node_Load

Instance Variables

Element_Label;

w_x; w_y; w_z; m_x; m_y; m_z;

Public Functions

Element_Load (anElement_Label, w_x, w_y, w_z, m_x, m_y, m_z)
getLabel ( )
getForce (anInteger)

Element_Load

Instance Variables

Superelement_Label;

Lower_Level_Load_Case_Label;

Public Functions

SE_Load (aSE_Label, aLower_Level_Load_Case_Label)
get_Label ( )
get_Lower_Level_Load_Case_Label ( )

SE_Load

Fig. 5.23. Load objects.

Instance Variables

theLoad_Case_Vector_Container

Public Functions

Vector_Manager (r_number, Y_number, p_number);
Vector_Manager (anotherVM);

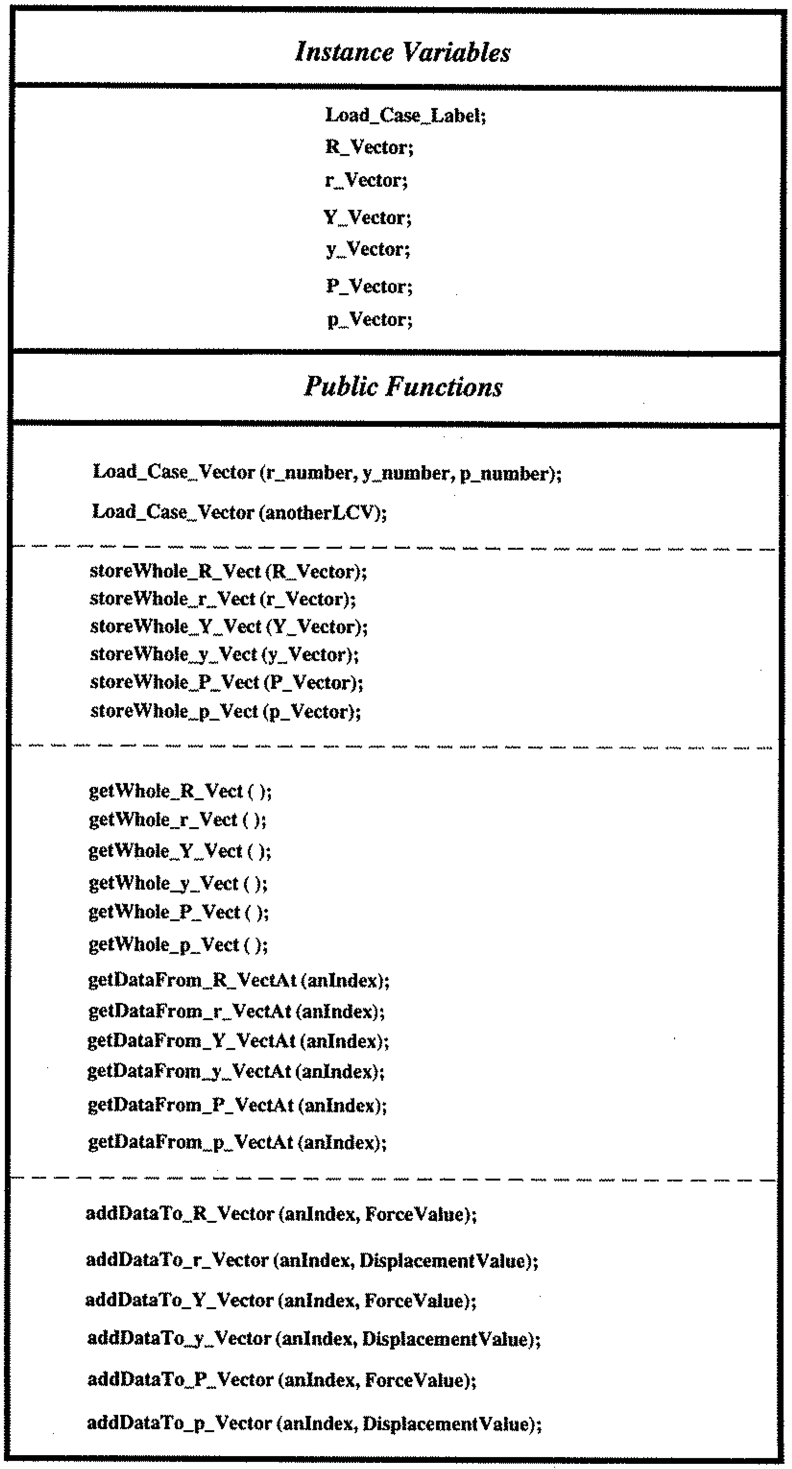
getLoad_Case_VectorWithLabel (aLoad_Case_Label);
getLoad_Case_VectorAt (anInteger);
createLoad_Case_Vector(aLoad_Case_Label);
copyLoad_Case(Load_Case_Label_1, Load_Case_Label_2);

storeWhole_R_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);
storeWhole_r_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);
storeWhole_Y_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);
storeWhole_y_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);
storeWhole_P_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);
storeWhole_p_VectWithLoad_Case_Label (aLoad_Case_Label, R_Vector);

getWhole_R_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_r_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_Y_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_y_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_P_VectByLoad_Case_Label (aLoad_Case_Label);
getWhole_p_VectByLoad_Case_Label (aLoad_Case_Label);
getDataFrom_R_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_r_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_Y_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_y_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_P_VectAt (aLoad_Case_Label, anInteger);
getDataFrom_p_VectAt (aLoad_Case_Label, anInteger);

addDataTo_R_Vector (aLoad_Case_Label, aDOFIndex, ForceValue);
addDataTo_r_Vector (aLoad_Case_Label, aDOFIndex, DisplacementValue);
addDataTo_Y_Vector (aLoad_Case_Label, aDOFIndex, ForceValue);
addDataTo_y_Vector (aLoad_Case_Label, aDOFIndex, DisplacementValue);
addDataTo_P_Vector (aLoad_Case_Label, aDOFIndex, ForceValue);
addDataTo_p_Vector (aLoad_Case_Label, aDOFIndex, DisplacementValue);

Fig. 5.24. Vector_Manager object.



Instance Variables

Load_Case_Label;
R_Vector;
r_Vector;
Y_Vector;
y_Vector;
P_Vector;
p_Vector;

Public Functions

Load_Case_Vector (r_number, y_number, p_number);
Load_Case_Vector (anotherLCV);

storeWhole_R_Vect (R_Vector);
storeWhole_r_Vect (r_Vector);
storeWhole_Y_Vect (Y_Vector);
storeWhole_y_Vect (y_Vector);
storeWhole_P_Vect (P_Vector);
storeWhole_p_Vect (p_Vector);

getWhole_R_Vect ( );
getWhole_r_Vect ( );
getWhole_Y_Vect ( );
getWhole_y_Vect ( );
getWhole_P_Vect ( );
getWhole_p_Vect ( );
getDataFrom_R_VectAt (anIndex);
getDataFrom_r_VectAt (anIndex);
getDataFrom_Y_VectAt (anIndex);
getDataFrom_y_VectAt (anIndex);
getDataFrom_P_VectAt (anIndex);
getDataFrom_p_VectAt (anIndex);

addDataTo_R_Vector (anIndex, ForceValue);
addDataTo_r_Vector (anIndex, DisplacementValue);
addDataTo_Y_Vector (anIndex, ForceValue);
addDataTo_y_Vector (anIndex, DisplacementValue);
addDataTo_P_Vector (anIndex, ForceValue);
addDataTo_p_Vector (anIndex, DisplacementValue);

Fig. 5.25. Load_Case_Vector object.

Instance Variables

r_number; R;

Public Functions

R_Vector (r_number);
R_Vector (another_R_Vector);
addData (forceValue, aDOF_Index);
zeroData (aDOF_Index);
getData (aDOF_Index);

Fig. 5.26. R_Vector object.

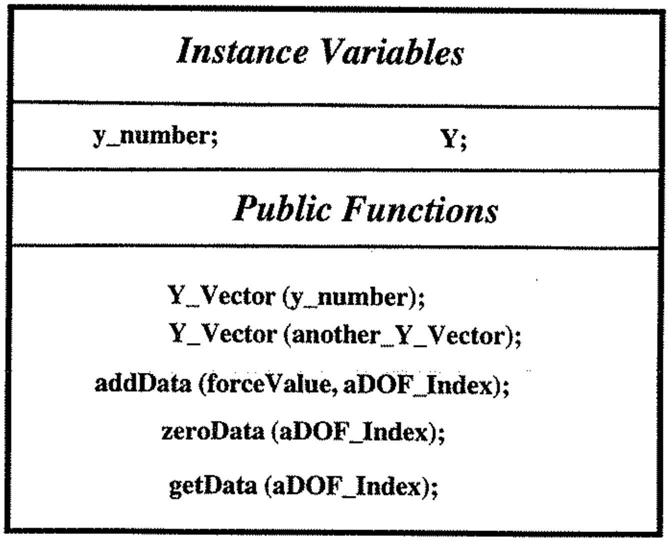
Instance Variables

r_number; r;

Public Functions

r_Vector (r_number);
r_Vector (another_r_number);
addData (displacementValue, aDOF_Index);
zeroData (aDOF_Index);
getData (aDOF_Index);

Fig. 5.27. r_Vector object.



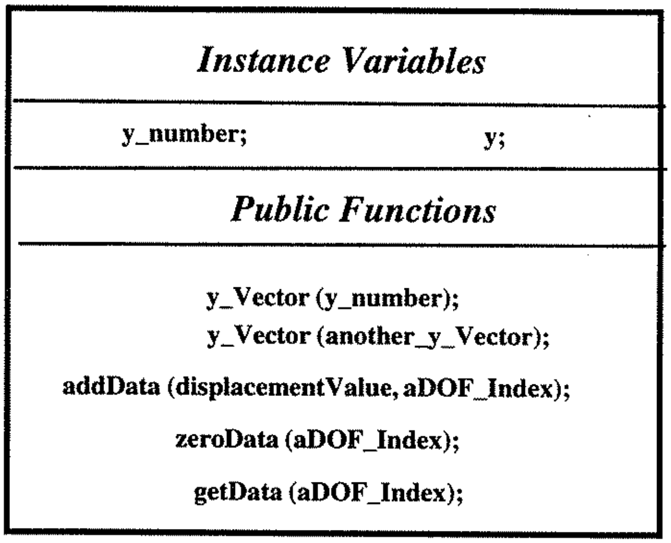
Instance Variables

y_number; **Y;**

Public Functions

Y_Vector (y_number);
Y_Vector (another_Y_Vector);
addData (forceValue, aDOF_Index);
zeroData (aDOF_Index);
getData (aDOF_Index);

Fig. 5.28. Y_Vector object.



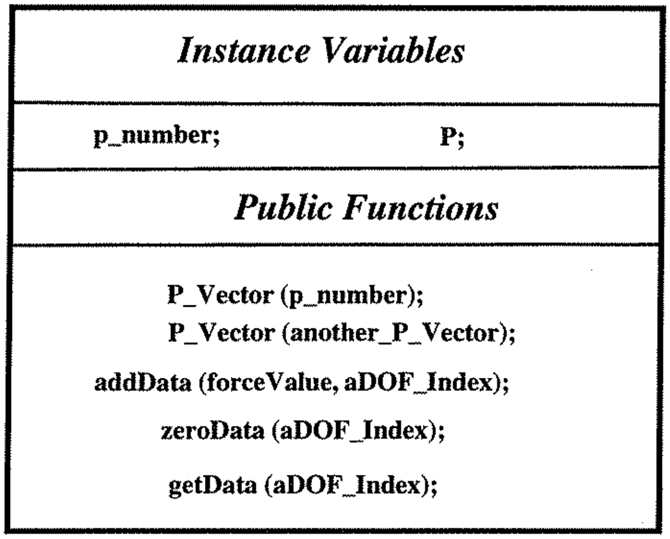
Instance Variables

y_number; **y;**

Public Functions

y_Vector (y_number);
y_Vector (another_y_Vector);
addData (displacementValue, aDOF_Index);
zeroData (aDOF_Index);
getData (aDOF_Index);

Fig. 5.29. y_Vector object.



Instance Variables

p_number; P;

Public Functions

P_Vector (p_number);

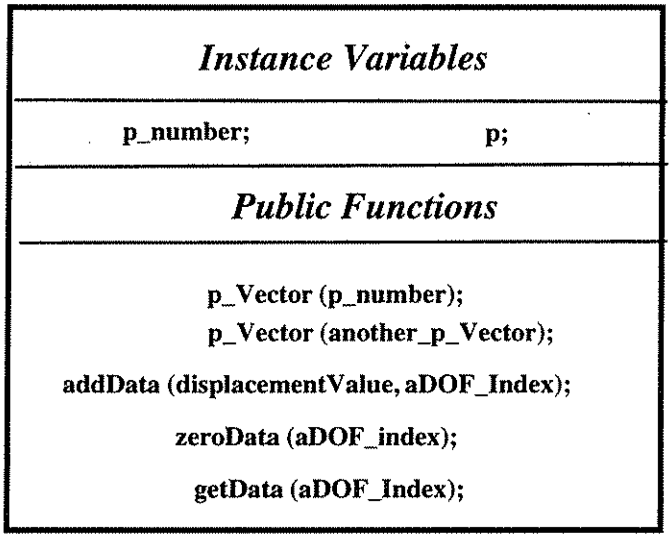
P_Vector (another_P_Vector);

addData (forceValue, aDOF_Index);

zeroData (aDOF_Index);

getData (aDOF_Index);

Fig. 5.30. P_Vector object.



Instance Variables

p_number; p;

Public Functions

p_Vector (p_number);

p_Vector (another_p_Vector);

addData (displacementValue, aDOF_Index);

zeroData (aDOF_index);

getData (aDOF_Index);

Fig. 5.31. p_Vector object.

The object at the end of arrow (lower level object) is held by the object at the start of arrow (higher level object).

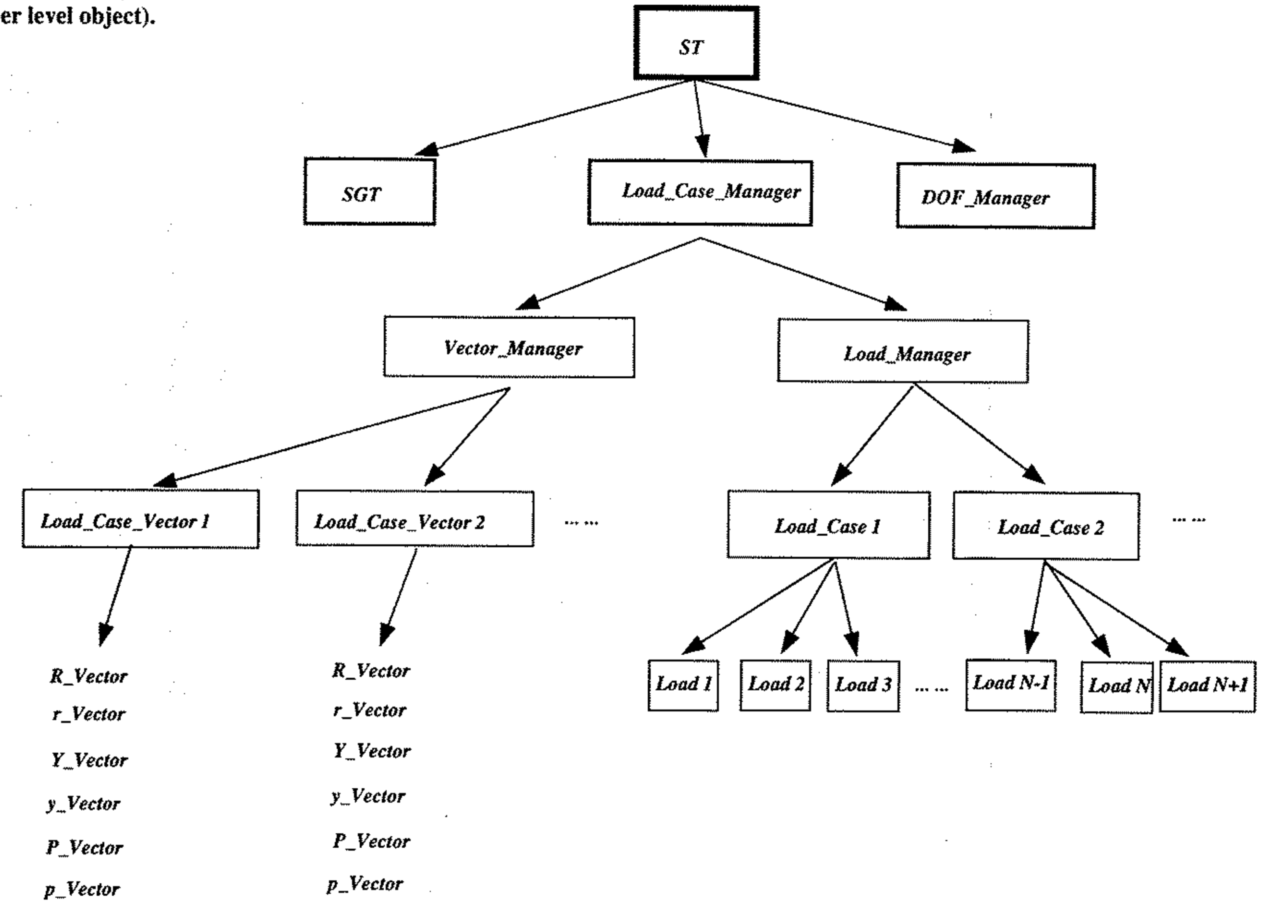


Fig. 5.32. Organization of objects related to the ST object

Instance Variables

theST

SET_Label

Public Functions

SET (SET_Label, aST);

SET (anotherSET);

getNodeWithLabel (aNode_Label);

establish_p_DOF ( );

set_r_Type_Indices ( );

getSET_Label ( );

getInitial_Forces (Lower_Level_Load_Case_Label);

**calculateInternalForces (p_Vect, aLower_Level_Load_Case_Label,
Load_Case_Label);**

getStiffness_Matrix ( );

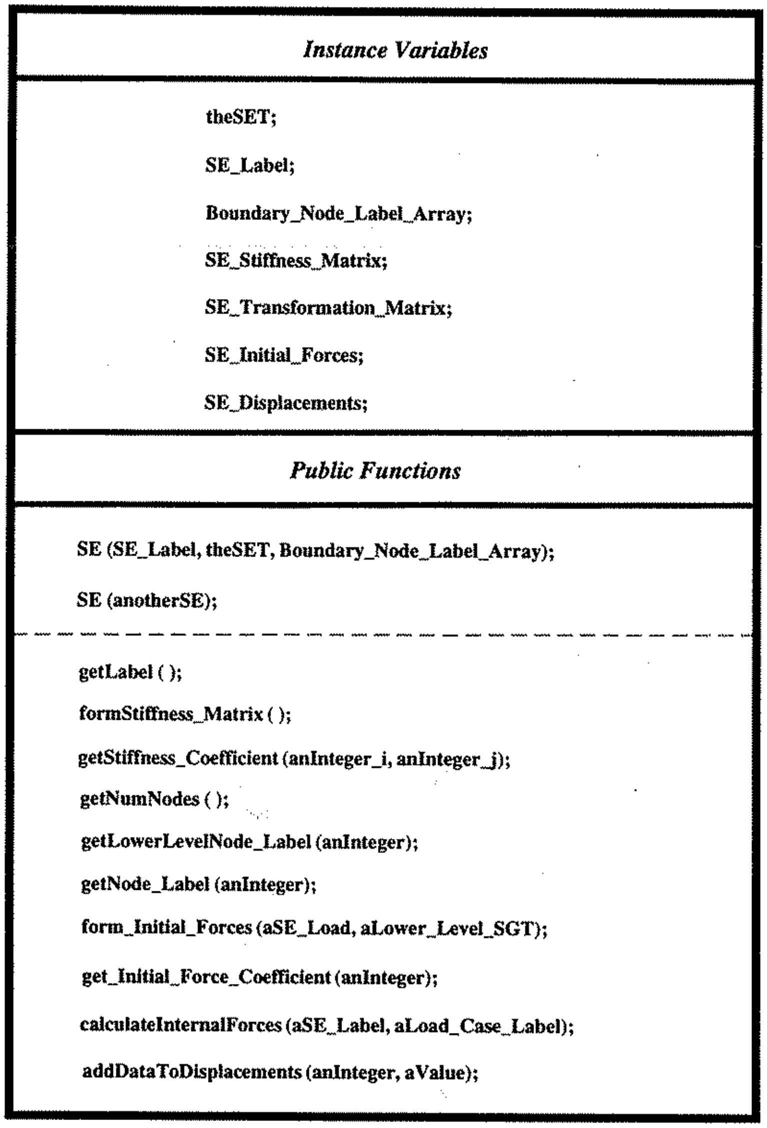
getLowerLevelDOF_TypeWithLabel (aDOF_Label);

getLowerLevelDOF_IndexWithLabel (aDOF_Label);

SubstructureAnalysis ( );

get_p_number ( );

Fig. 5.33. SET object.



Instance Variables

theSET;

SE_Label;

Boundary_Node_Label_Array;

SE_Stiffness_Matrix;

SE_Transformation_Matrix;

SE_Initial_Forces;

SE_Displacements;

Public Functions

SE (SE_Label, theSET, Boundary_Node_Label_Array);

SE (anotherSE);

getLabel ( );

formStiffness_Matrix ( );

getStiffness_Coefficient (anInteger_i, anInteger_j);

getNumNodes ( );

getLowerLevelNode_Label (anInteger);

getNode_Label (anInteger);

form_Initial_Forces (aSE_Load, aLower_Level_SGT);

get_Initial_Force_Coefficient (anInteger);

calculateInternalForces (aSE_Label, aLoad_Case_Label);

addDataToDisplacements (anInteger, aValue);

Fig.5.34. SE object.

Chapter 6

Object-Oriented Structural Analysis Algorithm

6.0 Introduction

In Chapter 5, an object-oriented analysis model, including several object classes, is presented for structural analysis using substructures. This chapter introduces one additional object, called the *Global* object, which constructs the other objects and conducts the analysis process, and also outlines a number of algorithms for object-oriented structural analysis. The objects presented in Chapter 5 range from relatively simple objects (i.e. basic objects), such as DOF objects, Node objects, and Element objects, to relatively complicated objects, such as SGT objects, ST objects, and SET objects. The Global object manages the interface between the input data file and these objects.

The chapter is organized as follows: the organization of the input data is discussed in Section 6.1, the Global object is described in Section 6.2, and the algorithm performed by the Global object to handle the input data and establish the other objects is outlined in Section 6.3. The structural analysis algorithms that use the objects of Chapter 5 are described in Section 6.4 through 6.11.

6.1 Input Data File

The structural analysis model presented in Chapter 5 (Fig. 5.2) consists of multiple levels of substructures, with multiple substructures at each level (except the highest level). Each substructure (except the highest level substructure) is modeled by a SE object. A SE object is developed from a SET object, which is developed from a ST object, which is developed from a SGT object. The highest level substructure is modeled by a ST object which is developed from a SGT object. Each object is described by a data set in the input file (Fig. 6.1): one data set is needed to describe each SGT object which represents the geometry and materials of a group of substructures; one data set is needed to describe each ST object which represents the geometry, materials, and loads of a group of substructures (with the same SGT); and one data set is needed to describe each SET object which represents the geometry, materials, loads, and boundary DOF of a group of substructures (with the same ST).

In the data file, the lowest level substructures are considered first. That is, the data sets for constructing the lowest level SGTs, STs, and SETs are first. For each SGT, the SGT data set contains a SGT_Label, which is an identifier of the SGT, and data for defining the Node and Element objects of the SGT (Fig. 6.1.a). For each ST, the ST data set contains a ST_Label which is an identifier of the ST, the label of an already-defined SGT object, a list of the DOF_Labels of the DOF that are "y" type DOF, and a description of the external forces acting on the "r" type DOF and the specified displacements of the "y" type DOF (Fig.6.1.b). For each SET, the SET data set contains a SET_Label, which is an identifier of the SET, the label of an already-defined ST object, and a list of DOF_Labels of the boundary or "p" type DOF (Fig. 6.1.c).

Higher level substructures include superelements (represented by SE objects) that are developed from lower level substructures (represented by SET objects). For a higher level substructure, the data needed to create the SE objects are included in the SGT data set. Therefore, the SGT data set for a higher level SGT includes the data that describes Node, Element, and SE objects (Fig. 6.1.a). Higher level ST and SET data sets are similar to these of the lowest level.

For the highest level substructure (i.e. the complete structure), the data for a SGT and a ST object are included in the input file. There is no SET developed for the highest level substructure.

6.2 Global Object

The Global object has the responsibility to read data from the input data file, and to create and interface with the other objects. The Global object contains the following instance variables and functions (Fig. 6.2):

Instance variables:

(1) *theSGT_List* is an instance of the KeySortCltn class of the NIH class library. It is a container of the SGT objects defined for the structural analysis model. The container holds associations of key objects, which are SGT_Label objects, with value objects, which are SGT objects.

(2) *theST_List* is also an instance of the KeySortCltn class. It is a container of the ST objects defined for the structural analysis model. The container holds associations of key objects, which are ST_Label objects, with value objects, which are ST objects.

(3) *theSET_List* is also an instance of the keySortCltn class and is similar to theSGT_List and theSET_List.

Functions:

(1) *is_The_SGT_In (aSGT_Label);*

To check if a SGT object is available in the SGT_List using the SGT_Label.

(2) *is_The_ST_In (aST_Label);*

To check if a ST object is available in the ST_List using the ST_Label.

(3) *is_The_SET_In (aSET_Label);*

To check if a SET object is available in the SET_List using the SET_Label.

(4) *storeSGT (aSGT_object);*

To store a SGT object into the SGT_List.

(5) *storeST (aST_object);*

To store a ST object into the ST_List.

(6) *storeSET (aSET_object);*

To store a SET object into the SET_List.

(7) *getSGT_With_Label (aSGT_Label);*

To return a SGT object from the SGT_List by its label.

(8) *getST_With_Label (aST_Label);*

To return a ST object from the ST_List by its label.

(9) *getSET_With_Label (aSET_Label);*

To return a SET object from the SET_List by its label.

6.3. Algorithm to Establish Objects and Conduct Analysis

The Global object interfaces with the data file shown in Fig. 6.1 to construct substructure and superelement objects, and to conduct the structural analysis. The discussion of the algorithm used to construct these objects is divided into three parts: (1) objects for the lowest level substructures, (2) objects for higher level substructures, and (3) objects for the highest level substructure. An overview of the analysis procedure is given in the discussion of the highest level substructure.

6.3.1 Objects for the Lowest Level Substructures

SGT Objects

For each SGT object, the Global object reads the SGT_Label from the data file, and constructs the SGT object without filling theNode_Container and theElement_Container of the SGT object. Then, data describing the Node objects of the SGT are read from the data file, and each Node object is constructed by the Global object from the Node_Label, and the x, y, and z coordinates in the data file. The Node constructor function *Node (Node_Label, x_coordinate, y_coordinate, z_coordinate)* is used. The Node object is stored in theNode_Container of the SGT object using the SGT function *storeNode (aNode_object).* The procedure is repeated until all Node objects are constructed and stored. Then, the data describing the Element objects of the SGT are read from the data file, and each Element object is constructed by the Global object, using the Element constructor function *Element (anElement_Label, i_node_Label, j_node_Label, reference_ node_Label, E, I, A).* It should be noted that the arguments i_node_Label, j_node_Label, and reference_node_label, are the labels of three Node objects that can be retrieved from theNode_Container of the SGT object. The node labels and other arguments, such as anElement_ Label, E, I, and A (the label of the Element, material, and geometry properties) are read from the data file. The Element object is stored in theElement_Container of the SGT object using the SGT function *storeElement (anElement_object).* The procedure is repeated until all Element objects are constructed and stored. After the Global object constructs and stores the Node and Element objects of the SGT, the Global object stores the SGT object in theSGT_List.

ST Objects

For each ST object, the Global object reads the ST_Label and the SGT_Label of the SGT that describes the geometry and materials of the ST from the data file. It constructs the ST object using the ST constructor function *ST (ST_label, aSGT).* The second argument in the function is a SGT object that is retrieved from theSGT_List using the SGT_Label, and is copied using the SGT constructor function *SGT (anotherSGT)* before being sent to the ST.

The DOF_Container within the DOF_Manager of the ST object is established from the information stored in the SGT using the ST function *buildDOF_Container ( ).* Then the DOF_Types of the specified displacement DOF of the ST are set to "y" type (as discussed in Section 5.3.3 and Section 6.4), using the ST function *establish_y_DOF ( ).* This function reads the DOF_Labels of the "y" type DOF from the data file. Then, the Global object uses the ST function *buildLoad_Cases ( ),* which reads data describing loads acting on the ST from the data file (see Section 5.3.3 and Section 6.5). All loads are grouped into load cases. Each load case is read and constructed, and the loads in the load case are read and constructed. After the load data are read, the Global object stores the ST object in theST_List.

SET Objects

For each SET object, the Global object reads the SET_label and the ST_Label of the ST object that describes the geometry, materials, and loads of the SET from the data file, and constructs the SET object using the SET constructor function *SET (SET_Label, aST)*. The second argument in the function is a ST object that is retrieved from theST_List using the ST_Label, and is copied using the ST constructor function *ST(anotherST)* before being sent to the SET. Then the Global object uses the SET function *establish_ p_DOF ( )* which reads data describing the boundary DOF (i.e. it reads DOF_Labels of the "p" type DOF) from the data file. After the "p" type DOF are established, the Global object uses the SET function *SubstructureAnalysis ( )*, which performs the first part of the substructure analysis (the α case) described in Chapter 3. This function is discussed further in Section 6.9 After the substructure analysis function is completed, the Global object stores the SET object in theSET_List.

6.3.2. Objects for Higher Level Substructures

SGT Objects

For each SGT object, the Global object reads data and constructs the SGT object and the Node and Element objects as described for the lowest level substructures. A higher level substructure will also include SE objects. For each SE in a higher level SGT, the Global object uses the SE constructor function *SE (SE_Label, theSET, Boundary_Node_Label_Array)* to construct the SE. The second argument in the function is a SET object that is retrieved from theSET_List using the SET_Label, and is copied using the SET constructor function *SET (anotherSET)* before being sent to the SE. The third argument is an array of the Node_Labels of the boundary nodes of the SE. This array contains a list of associations. Each association consists of two Node_Labels, one is the Node_Label of a boundary node in the current level SGT, the other is the Node_Label of the boundary node in the lower level SGT. The Global object reads the Node_Labels of the boundary nodes from the data file and constructs the Boundary_Node _Label_Array.

ST and SET Objects

Each ST and SET object for a higher level substructure is constructed as described earlier for the lowest level substructures.

6.3.3. Objects for the Highest Level Substructures

SGT Object

The SGT object for the highest level substructure is constructed as described earlier for the higher level substructures.

ST Object and Analysis Overview

To construct the ST object for the highest level substructure (the complete structure), the Global object performs operations similar to those for the lowest level substructures. However, for the highest level substructure, a SET is not developed and all the DOF of the structure in this level are either "r" type DOF, or "y" type DOF. The Global object stops reading data and constructing objects when an existing ST is identified as the highest level substructure. At this point, the structure analysis model is complete and the analysis can begin. The Global object conducts the analysis by using the ST function *SubstructureAnalysis ( )* which solves for the unknown displacement vector **r** and the unknown reaction force vector **Y** for the highest level substructure (see Section 6.9). Note that the operations of this function perform the first part of the substructure analysis (the α case) for a typical lower level substructure which has boundary or "p" type DOF, however, the same operations perform a complete analysis of the displacement and force vectors of the highest level substructure which has no "p" type DOF. The Global object completes the analysis (see Section 6.11) using ST function *HighestLevelSubstructureAnalysisCompletion ( ).*

6.4 Algorithm to Establish DOF_Container

The algorithm to construct the DOF_Container (shown in Fig. 5.14) is divided into four parts. Each part is implemented in a different function which is executed independently.

(1) The ST function *buildDOF_Container ( )* forms the DOF_Container. This function is used by the Global object immediately after the ST object is constructed. The function uses information held by the Node objects of the SGT object associated with the ST to construct DOF objects. The DOF objects are constructed with an assumed DOF_Type="r", and DOF_Index=0. For the example in Fig. 5.14., the first association in the DOF container is initially (DOF_1.1 <==> DOF_1.1, "r", 0). However, as indicated in Fig. 5.14, the correct type for the DOF_1.1 object is "y" type and the correct index is 1. These are set later as discussed below. The algorithm for *buildDOF_Container ( )* is shown in Fig. 6.4.

(2) The ST function *establish_y_DOF ( )* is used by the Global object to modify the DOF_Container by changing the type of certain DOF objects to "y" type. The *establish_ y_DOF ( )* function reads the labels of the "y" type DOF from the data file. It resets the DOF_Type and DOF_Index. For the example in Fig. 5.14, the first association is changed to (DOF_1.1 <==> DOF_1.1, "y", 1). Fig. 6.5 shows the algorithm for *establish_y_DOF ( ).*

(3) The SET function *establish_p_DOF ( )* is used by the Global object to modify the DOF_Container by changing the type of certain DOF objects to "p" type . The *establish_p_DOF ( )* function reads the labels of the "p" type DOF from the data file. It resets the DOF_Type and DOF_Index. For the example in Fig. 5.14, the second association is changed to (DOF_1.2 <==> DOF_1.2, "p", 1). Fig. 6.6 shows the algorithm for *establish _p_DOF ( )*. It should be noted that the process of establishing the "p" type DOF is part of defining a SET object, rather than a ST object. However, the SET object uses a function of the associated ST object to establish the "p" type DOF as shown in Fig. 6.6.

(4) The SET function *set_r_Type_Indices ( )* is used by the Global object to set correct indices for the "r" type DOF after the "y" type and "p" type DOF are established. These indices

were initially set to 0 as discussed above. Fig. 6.7 shows the algorithm used to assign indices to the "r" type DOF by the SET function *set_r_Type_Indices ( )*.

6.5 Algorithm to Establish Load_Case Objects and Load_Case_Vector Objects

The loads acting on a ST are read from the data file immediately after the DOF_Container of the ST is established. As discussed in Chapter 5, each load is represented by a Load object. A structural analysis usually considers more than one load case. Each load case is represented by a Load_Case object, which contains one or more Load objects. Each Load_Case object is stored in a Load_Case_Container, which is part of a Load_Manager object. Each Load_Case object in the Load_Case_Container is processed to establish a Load_Case_Vector object, which corresponds to a Load_Case and holds six vectors (i.e. R_Vector, r_Vector, Y_Vector, y_Vector, P_Vector, and p_Vector). Each Load_Case_Vector object is stored in a Load_Case_Vector_Container, which is part a Vector_Manager object. The complete algorithm consists of two steps:

(1) The ST function *buildLoad_Cases ( )*, shown in Fig. 6.8, establishes the Load_Case objects and stores them in a Load_Case_Container, which is part of the Load_Manager object, which is part of the Load_Case_Manager object of the ST. A ST object reads the number of load cases for the substructure from the input file. A Load_Case_ Label is obtained for each load case, and the Load_Case_Manager function *createLoad_ Case (aLoad_Case_Label)* uses the Load_Manager function *createLoad_Case (aLoad_Case_Label)* to create a Load_Case object for the load case. The Load_Case object is inserted into the Load_Case_Container within the Load_Manager. The function *buildLoad_Cases ( )* then reads data describing each load in the load case from the data file and constructs a Load object. The type of Load object and the required data depends on the type of load specified. Three types of loads are considered: (1) Node loads, (2) Element loads, and (3) SE loads. A Load_Case_Manager function *storeLoad (aLoad_Case_Label, aLoad_Label, aLoad)* stores the Load object using the Load_Manager

function *storeLoad (aLoad_Case_Label, aLoad_Label, aLoad).* The Load_Manager retrieves a Load_Case object from its Load_Case_Container based on the Load_Case_Label and stores the Load object in the Load_Container of the Load_Case object using the Load_Case function *storeLoad (aLoad_Label, aLoad).* This procedure is repeated until Load objects are constructed and stored in a Load_Case object for all the loads of a Load_Case, and until all load cases are processed. The algorithm is shown in Fig. 6.8.

(2) The ST object uses the Load_Case_Manager function *buildLoad_Case_Vectors (theSGT, theDOF_Manager),* shown in Fig. 6.9, to establish the Load_Case_Vector objects and store them in a Load_Case_Vector_Container, which is part of the Vector_Manager object, which is part of the Load_Case_Manager object. The Load_Case_Manager object uses the Load_Manager function *getNumberOf Load_Cases ( )* to obtain the number of load cases in the analysis. Using the Load_Manager function *getLoad_CaseAt (anInteger),* each Load_Case object is retrieved from the Load_Case_Container, which is part of the Load_Manager. The number of Load objects contained in the Load_Case can be determined by the Load_Case function *getNumberOfLoads ( ).* Each Load object is retrieved from the Load_Container of the Load_Case object and one of the following procedures is used:

i. If the Load object is a Node_Load, the SGT function *getNodeWithLabel (aNode_Label)* is used to retrieve the Node object that is subjected to the load. The labels of the six DOF associated with the Node are obtained from the Node. Each DOF_Label is used to obtain the DOF_Type and DOF_Index of the corresponding DOF object from the DOF_Manager object. Using this information, the force coefficient (or specified displacement coefficient) for that DOF is inserted into the correct position of the correct vector using the Vector_Manager functions *addData_to_R_Vector (aDOFIndex, aForceValue), addData_to_y_Vector (aDOFIndex, aDisplacementValue),* or *addData_to_P_Vector (aDOFIndex, -aForceValue).* Note that a minus sign is used for a node force applied to a "p" type DOF. This is the case of a $\mathbf{P}_e$ force that was neglected in Chapter 3.

ii. If the Load object is an Element_Load, the SGT function *getElementWithLabel (anElement_Label)* is used to retrieve the Element object that is subjected to the load. The Load_Case_Manager function *processElemLoad (anElement, aLoad, aLoad_Case_Label, theSGT, theDOF_Manager)*, shown in Fig. 6.10, uses the Element function *form_Initial_Forces (aLoad)* to construct the element initial force vector. Then, for each of the Node objects associated with the element, the Node_Label is obtained from the Element object, and using the Node_Label, the Node object is obtained from the SGT object. The DOF_Label objects of each of the six DOF associated with the Node are obtained, and using the DOF_Label, the DOF_Type and DOF_Index are obtained from the DOF_Manager. Then the corresponding coefficient of the element initial force vector is retrieved using function *get_initial_Force_Coefficient (anInteger)* and is inserted into the correct position of the correct vector, as discussed in (i).

iii. If the Load object is a SE_Load, the SGT function *getSE_WithLabel (aSE_Label)* is used to retrieve the SE object that is subjected to the load. The Load_Case_Manager function *processElemLoad (theSE, aLoad, aLoad_Case_Label, theSGT, theDOF_Manager)* is also used for the SE object. The function works on a SE object and SE_Load as discussed in (ii), except that it uses SE functions rather than Element functions.

The procedure is repeated to establish a Load_Case_Vector object for each Load_Case object included in a Load_Case_Container. The algorithm for establishing the Load_Case _Vector objects is shown in Fig. 6.9.

6.6 Algorithm to Transform Element Initial Force Vector to Substructure Coordinate System

The element (both the Element object and the SE object) initial force vectors discussed in Section 6.4 are assembled with respect to the element (Element or SE) local coordinate system. In Chapter 3, the element initial force vectors are denoted as $\mathbf{P}_0^{(e)}$ and $\mathbf{P}_{0(r)}$ for a beam element and a superelement respectively. However, when they are included in the substructure analysis, they have to be transformed to the substructure coordinate system by:

$$Q_0^{(e)} = T^T P_0^{(e)}$$

and:

$$(Q_0)_{(r)} = T^T (P_0)_{(r)}$$

where $\mathbf{Q}_0^{(e)}$ and $(\mathbf{Q}_0)_{(r)}$ are the initial force vectors in the substructure coordinate system for a beam element and a superelement respectively, and **T** is a transformation matrix for the beam element or superelement. Both the Element object and the SE object include a transformation matrix as an instance variable (Element_Transformation_Matrix for the Element object and SE_Transformation_Matrix for the SE object).

The transformation algorithm for the Element object is presented in Fig. 6.11, and the transformation algorithm for the SE object is presented in Fig. 6.12. The results of these two algorithms are initial force vectors (Element_Initial_Forces for the Element object, and SE_Initial_Forces for the SE object) in the substructure coordinate system.

6.7 Algorithm to Transform Element Stiffness Matrix to Substructure Coordinate System

The element (both the Element object and the SE object) stiffness matrices are assembled with respect to the element local coordinate system. In Chapter 3, the element stiffness matrices are denoted as $\mathbf{K}_p^{(e)}$ and $(\mathbf{K}_p^*)_{(r)}$ for a beam element and a superelement respectively. These matrices are transformed to the substructure coordinate system as follows:

$$K_q^{(e)} = T^T K_p^{(e)} T$$

and:

$$(K_q^*)_{(r)} = T^T (K_P^*)_{(r)} T$$

where $\mathbf{K}_q^{(e)}$ and $(\mathbf{K}_q^*)_{(r)}$ are the stiffness matrices in the substructure coordinate system for a beam element and a superelement respectively, and **T** is a transformation matrix as discussed earlier.

The transformation algorithm for the Element object is presented in Fig. 6.13, and the transformation algorithm for the SE object is presented in Fig. 6.14. The results of these two algorithms are stiffness matrices (Element_Stiffness_Matrix for the Element object, and SE_Stiffness_Matrix for the SE object) in the substructure coordinate system.

6.8 Algorithm to Assemble Substructure Stiffness Matrix

The substructure stiffness matrix is shown in Eq. 3.4 as:

$$\begin{bmatrix} K_{rr} & K_{ry} & K_{rp} \\ K_{yr} & K_{yy} & K_{yp} \\ K_{pr} & K_{Py} & K_{pp} \end{bmatrix}$$

The matrix is partitioned into nine submatrices which reflects the fact that a typical substructure has three types of DOF: "r" type, "y" type, and "p" type (Fig. 5.14). Each DOF has a position among the DOF of the same type, given by DOF_Index. Within a substructure, the nine submatrices are kept within a single matrix, *substrucStiffness*, and the position in the stiffness matrix for a DOF is determined by the function *getMatrixIndexWithDOF_Label (aDOF_Label).* A typical substructure has two types of elements: beam elements (represented by Element objects) and superelements (represented by SE objects). Each element has its own stiffness matrix, either a beam element stiffness matrix (12 x 12) or a superelement stiffness matrix ((number of boundary nodes x 6) x (number of boundary nodes x 6)). The algorithm for stiffness assembly maps the element DOF (12 for the Element object or (number of boundary nodes) x 6 for the SE object) to the substructure DOF. This is done through the nodes associated with the element. Each node has six DOF, and the element DOF are sequence; the first six for the first node, the second six for the second node, and so on.

The substructure stiffness matrix assembly is carried out by the ST using the ST function *assembleStiffnessMatrix ( )*. The algorithm is separated into two parts. Fig. 6.15 shows the algorithm for the ST function *assembleStiffnessMatrix ( )* which assembles the substructure stiffness matrix from both Element and SE contributions using the ST function *assembleElem StiffnessMartrix (anElement)*. Fig. 6.16 shows the algorithm for the ST function *assembleElem StiffMatrix (anElement)*, which assembles one Element or SE stiffness matrix into the substructure stiffness matrix.

6.9 Substructure Analysis Algorithm

As presented in Chapter 3, the substructure analysis for a typical ST consists of two parts:

(1) the α case, in which the boundary displacements are fixed (i.e. the elements of p_Vector are assumed to be zero).

(2) the β case, in which the actual boundary displacements (the elements of p_Vector) are obtained from a higher level ST, and used to complete the analysis.

α **case**:

The α case of the substructure analysis is performed by a ST object immediately after the "p" type DOF have identified, and the indices for the "r" type DOF are set. This analysis is initiated by the Global object using the SET function *SubstructureAnalysis ( )*, which uses the ST function *SubstructureAnalysis ( )* as shown in Fig. 6.11. This ST function forms the substructure stiffness matrix, retrieves the force vectors (R_Vector, Y_Vector, and P_Vector) and the displacement vectors (r_Vector and y_Vector), sets up the equilibrium relationship between forces and displacements shown in Eq. 3.4, and solves for $\mathbf{r}^{(\alpha)}$, $\mathbf{Y}^{(\alpha)}$, and $\mathbf{P}^{(\alpha)}$ using Eq. 3.9, Eq. 3.10, and Eq. 3.11. These results are placed in r_Vector, Y_Vector, and P_Vector. It should be noted that if the ST is subjected to more than one Load_Case, then multiple sets of vectors are analyzed, one set of vectors for each Load_Case. When the analysis is over, the ST stores the sets of vectors. The α case algorithm is shown in Fig. 6.17.

β **case**:

The β case of the substructure analysis is performed by a ST object for a single load case, as directed by a SE object through its associated SET object. The β case analysis is part of the calculation of internal forces for the SE object, as discussed in Section 6.10. The SET object uses the ST function *LowerLevelSubstructureAnalysisCompletion (p_Vector, Current_Level_Load_Case_Label, Higher_Level_Load_Case_Label)*, shown in Fig. 6.18, to initiate the β case analysis. The first argument p_Vect contains the displacements of the boundary ("p" type) DOF obtained from the analysis of a higher level substructure for the load case identified by Higher_Level_Load_Case_Label (the third argument). The second argument Current_Level_Load_Case_Label is the label of the load case of the ST which contributes boundary forces ($\mathbf{P}^{(\alpha)}$) as SE_Initial_Forces to the load case identified by Higher_Level_Load_Case_Label of the higher level substructure through the associated SET and SE objects. Current_Level_Load_Case_Label is undefined if the ST object did not contribute boundary forces to the load case identified by Higher_Level_Load_Case_Label (i.e. if the ST, through the associated SET and SE objects, contributes a condensed stiffness as a SE_Stiffness_Matrix to the higher level substructure, but does not contribute loads as SE_Initial_Forces because a SE_Load for the associated SE object is not included in the load case identified by Higher_Level_Load_Case_Label).

The ST function *LowerLevelSubstructureAnalysisCompletion (p_Vect, Current_Level_Load_Case_Label, Higher_Level_Load_Case_Label)* carries out the β case analysis for a new load case for the ST that is created by copying the load case identified by Current_Level_Load_Case_Label to a new load case (i.e. new for the current level substructure, but already existing in the higher level substructure) which is given Higher_Level_Load_Case_Label as its label. The β case analysis is carried out for a load case which combines the loads applied to the ST in the load case identified by Current_Level_Load_Case_Label with the boundary displacements determined by a higher level substructure in the load case identified by Higher_Level_Load_Case_Label. If Current_ Level_Load_Case_Label is undefined, the new load case consists of an empty current level load case (i.e. no loads applied to the ST and the results of the α case consist of vectors of zero reaction forces and displacements) combined with the boundary displacements determined by a higher level substructure in the load case identified by Higher_Level_Load_Case_Label.

The *LowerLevelSubstructureAnalysisCompletion (p_Vect, Current_Level_Load_Case_Label, Higher_Level_Load_Case_Label)* function first creates the new load case, and then retrieves r_Vect, Y_Vect, and P_Vect from this load case (i.e. $\mathbf{r}^{(\alpha)}$, $\mathbf{Y}^{(\alpha)}$, and $\mathbf{P}^{(\alpha)}$). Then, based on the p_Vect (i.e. $\mathbf{p}^{(\beta)}$) that is provided by the higher level substructure, the β case results are calculated using Eq. 3.12, 3.13, and 3.14, producing $\mathbf{r}^{(\beta)}$, $\mathbf{Y}^{(\beta)}$, and $\mathbf{P}^{(\beta)}$. The results are superposed as discussed in Chapter 3:

$$\begin{Bmatrix} r \\ R \\ y \\ Y \\ p \\ P \end{Bmatrix} = \begin{Bmatrix} r^{(\alpha)} \\ R^{(\alpha)}-R_0 \\ y^{(\alpha)} \\ Y^{(\alpha)} \\ 0 \\ P^{(\alpha)} \end{Bmatrix} + \begin{Bmatrix} r^{(\beta)} \\ 0 \\ 0 \\ Y^{(\beta)} \\ p^{(\beta)} \\ P^{(\beta)} \end{Bmatrix}$$

with the exception that R_Vector contains $\mathbf{R}^{(\alpha)}\text{-}\mathbf{R}_0$ which is the applied external node forces and the element initial force combined. After the final force and displacement vectors have been obtained for the ST (i.e. both the α and β cases are completed), the internal forces in each element within the ST (in the associated SGT) are calculated. This is discussed in the following section. The β case algorithm is shown in Fig. 6.18.

6.10 Algorithm to Calculate Element Internal Forces

After the substructure analysis (both the α case and the β case) is completed for a substructure, the internal forces in the elements (Element objects and SE objects) within the substructure must be determined. These forces depend, of course, on the displacements of the substructure. The element (Element object or SE object) displacements in the substructure coordinate system can be obtained from the substructure displacement vectors by mapping the substructure DOF to the element (Element object or SE object) DOF. The displacements are then transformed from the substructure local coordinate system to the element (Element object or SE object) local coordinate system:

$$p = Tq$$

where **p** is a vector of displacements of the boundary nodes of an element (either an Element object or a SE object) in the element local coordinate system, **q** is a vector of displacements of the boundary nodes in the substructure local coordinate system, and **T** is the element transformation matrix.

The element (Element or SE) internal force vector **P** can be obtained by:

$$P = K_p p + P_0$$

where $\mathbf{K}_p$ is the element stiffness matrix, and $\mathbf{P}_0$ is the element initial force vector. The element forces given by this expression are adequate results for an Element object, however, since a SE object is developed from a lower level substructure, the displacements (**r**) and reactions (**Y**) of the interior DOF, and the element forces within the lower level substructure are also needed.

The algorithm to calculate element internal forces begins in the ST function *LowerLevel SubstructureAnalysisCompletion (p_Vect, Current_Level_Load_Case_Label, Higher_Level_Load_ Case_Label)*, shown in Fig. 6.18, which loops over the Elements and SEs within the associated SGT object and uses the ST function *calculateElemInternalForces (theElement, theLoad_Case_*

Label). This ST function, shown in Fig. 6.19, uses two other functions, the Element function *calculate InternalForces (anElement_Load, aLoad_Case_ Label)*, shown in Fig. 6.20, and the SE function *calculate InternalForces (aSE_Load, aLoad_Case_Label)*, shown in Fig. 6.21.

The ST function *calculateElemInternalForces (theElement, theLoad_Case_Label)* obtains the element (Element object or SE object) displacements in the substructure coordinate system from the substructure displacement vectors, and sends them to the element using the Element object function *addDataToDisplacements (anIndex)* or the SE object function *addDataTo Displacements (anIndex).* The ST function also obtains the element (Element or SE) load (if one exists) for the current load case. Then the Element object function *calculateInternalForces (anElement_Load, aLoad_Case_Label)* or the SE object function *calculateInternalForces (aSE_ Load, aLoad_Case_Label)* is used to obtain the internal forces. Both of these functions transform the displacements from the substructure coordinate system to the element coordinate system. For an Element object, the expression given above is used to obtain the element internal forces. A SE object uses the SET function *calculateInternalForces (p_Vect, aLower_Level_Load_Case_Label, Load_Case_Label)* for the associated SET object. This function then uses the ST function *LowerLevelSubstructureAnalysisCompletion (p_Vect, Current_Level_Load_Case_Label, Higher_ Level_Load_Case_Label)* for the associated ST object (lower level substructure) to carry out the β case substructure analysis and produce the displacements (**r**) and reactions (**Y**) of the interior DOF, and the element forces within the lower level substructure.

6.11 Highest Level Substructure Analysis Completion

The Global object uses the ST function *HighestLevelSubstructureAnalysisCompletion ( ),* shown in Fig. 6.22, to calculate element internal forces for the highest level substructure. Prior to this, the Global object must use the ST function *SubstructureAnalysis ( )* to calculate the displacements (**r**) and reactions (**Y**). Since the highest level substructure does not have "p" type DOF, the results for the α case determined by the *SubstructureAnalysis ( )* function are the total displacements of the substructure. Thus the ST function *HighestLevelSubstructureAnalysis Completion ( )* does not carry out a β case analysis, but immediately calculates the element internal forces. Note that this function loops over all load cases of the highest level ST. All of the analysis results for the lower level substructures will be given for these load cases because the lower level STs will create new load cases with labels to match those of the highest level substructure as discussed in Section 6.9.

```
SGT_Label

Node 1  x_coordinate1  y_coordinate1  z_coordinate1      Data for constructing Node  objects
Node 2  x_coordinate2  y_coordinate2  z_coordinate2
... ...
                                                          Data for constructing Element objects
Element 1  i_node_Label 1  j_node_Label 1  reference_node_Label 1  E1  I1  A1
Element 2  i_node_Label 2  j_node_Label 2  reference_node_Label 2  E2  I2  A2
... ...

SE 1  aSET_Label  Lower_B_Node1  Upper_B_Node1     Data for constructing SE objects
                  Lower_B_Node2  Upper_B_Node2     (for higher level SGT only)
                  ... ...
SE 2  aSET_Label  Lower_B_Node1  Upper_B_Node1
                  Lower_B_Node2  Upper_B_Node2
                  ... ...
```

a. A SGT data set

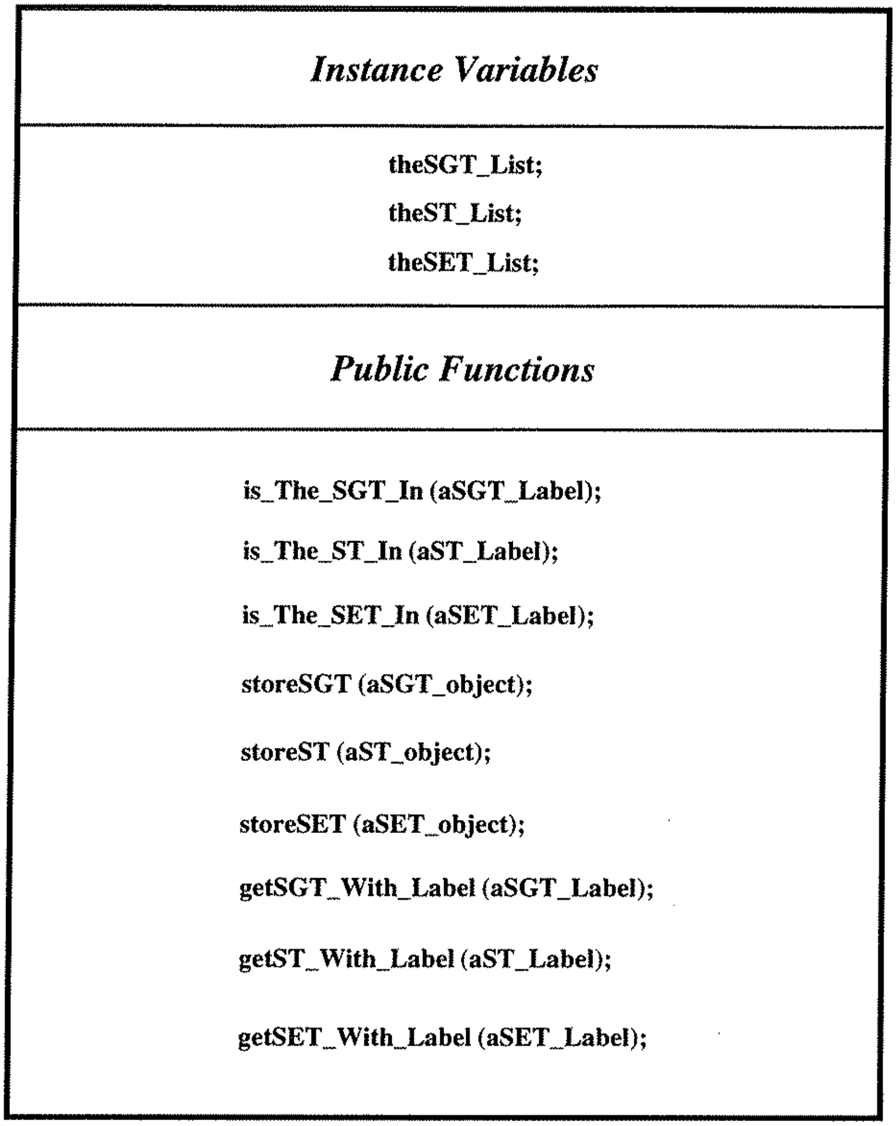
```
ST_Label     aSGT_Label

NumberOf_y_TypeDOF

y_Type_DOF_Label1                 DOF_Labels for "y" type DOF objects
y_Type_DOF_Label2
y_Type_DOF_Label3
... ...
NumberOfLoad_Cases
                                  Data for constructing Load_Case objects
Load_Case_Label 1
NumberOfLoads
Load 1    Load 2   ... ...

Load_Case_Label 2
NumberOfLoads
Load 1    Load 2   ... ...

... ...
```

b. A ST data set

```
SET_Label     aST_Label

NumberOf_p_TypeDOF
                                      DOF_Labels for "p" type DOF objects
p_Type_DOF_Label1
p_Type_DOF_Label2
p_Type_DOF_Label3
... ...
```

c. A SET data set

Fig. 6.1. Data file.



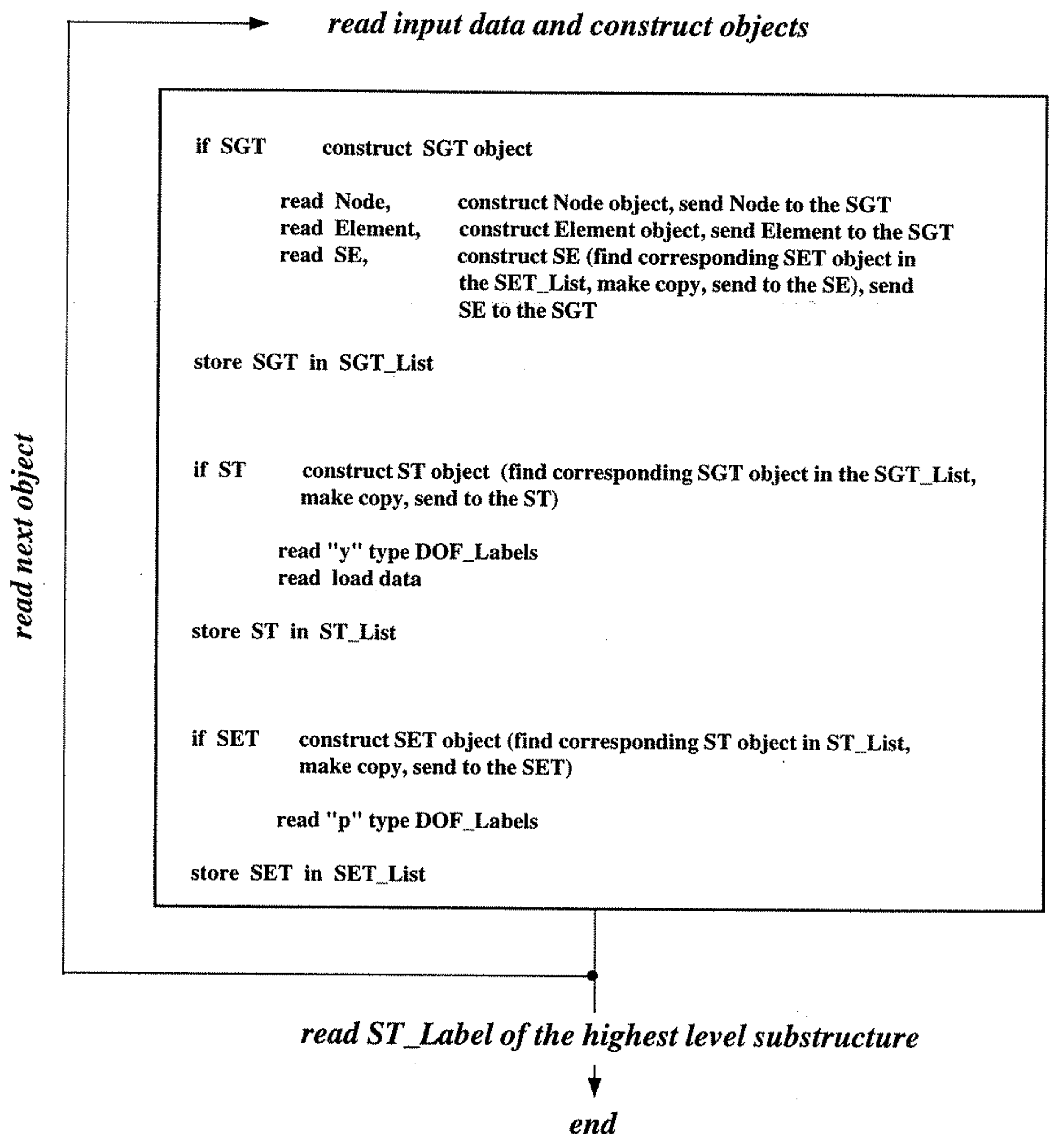
Instance Variables

theSGT_List;
theST_List;
theSET_List;

Public Functions

is_The_SGT_In (aSGT_Label);
is_The_ST_In (aST_Label);
is_The_SET_In (aSET_Label);
storeSGT (aSGT_object);
storeST (aST_object);
storeSET (aSET_object);
getSGT_With_Label (aSGT_Label);
getST_With_Label (aST_Label);
getSET_With_Label (aSET_Label);

Fig. 6.2. Global object.



```
read input data and construct objects

if SGT      construct SGT object

        read Node,       construct Node object, send Node to the SGT
        read Element,    construct Element object, send Element to the SGT
        read SE,         construct SE (find corresponding SET object in
                         the SET_List, make copy, send to the SE), send
                         SE to the SGT

store SGT in SGT_List


if ST       construct ST object (find corresponding SGT object in the SGT_List,
            make copy, send to the ST)

        read "y" type DOF_Labels
        read load data

store ST in ST_List


if SET      construct SET object (find corresponding ST object in ST_List,
            make copy, send to the SET)

        read "p" type DOF_Labels

store SET in SET_List

read next object

read ST_Label of the highest level substructure

end
```

Fig. 6.3 Global object algorithm to construct objects.

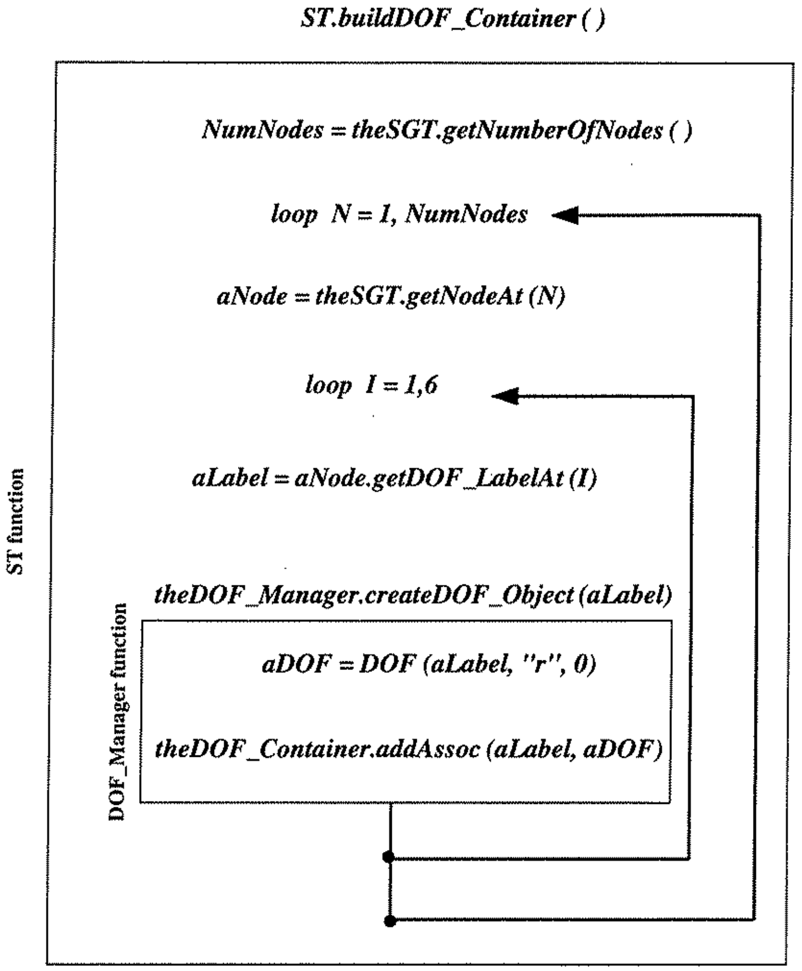
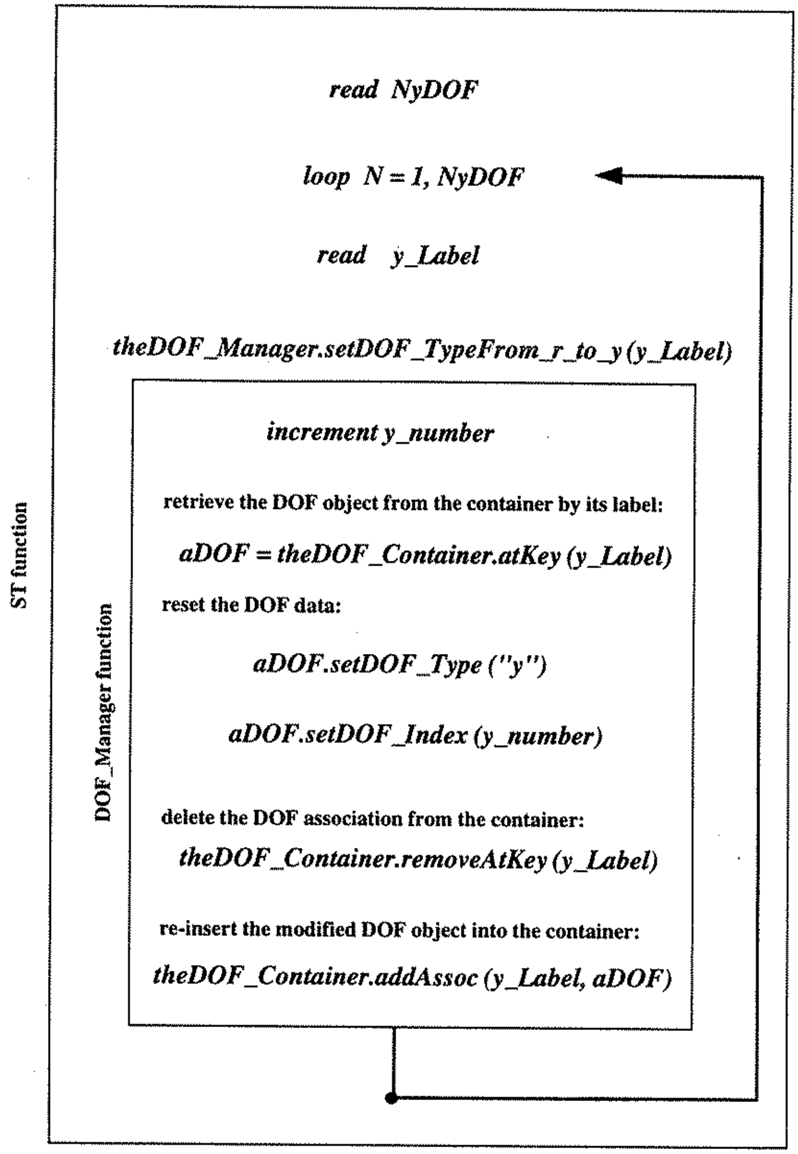

ST.buildDOF_Container ( )
ST function
NumNodes = theSGT.getNumberOfNodes ( )
loop N = 1, NumNodes
aNode = theSGT.getNodeAt (N)
loop I = 1,6
aLabel = aNode.getDOF_LabelAt (I)
theDOF_Manager.createDOF_Object (aLabel)
DOF_Manager function
aDOF = DOF (aLabel, "r", 0)
theDOF_Container.addAssoc (aLabel, aDOF)

Fig. 6.4. Construction of DOF_Container.

ST.establish_y_DOF ( )



```
read NyDOF
loop N = 1, NyDOF
    read y_Label
    theDOF_Manager.setDOF_TypeFrom_r_to_y (y_Label)
        increment y_number
        retrieve the DOF object from the container by its label:
        aDOF = theDOF_Container.atKey (y_Label)
        reset the DOF data:
            aDOF.setDOF_Type ("y")
            aDOF.setDOF_Index (y_number)
        delete the DOF association from the container:
        theDOF_Container.removeAtKey (y_Label)
        re-insert the modified DOF object into the container:
        theDOF_Container.addAssoc (y_Label, aDOF)
```

Fig. 6.5. Algorithm to modify "y" type DOF objects.

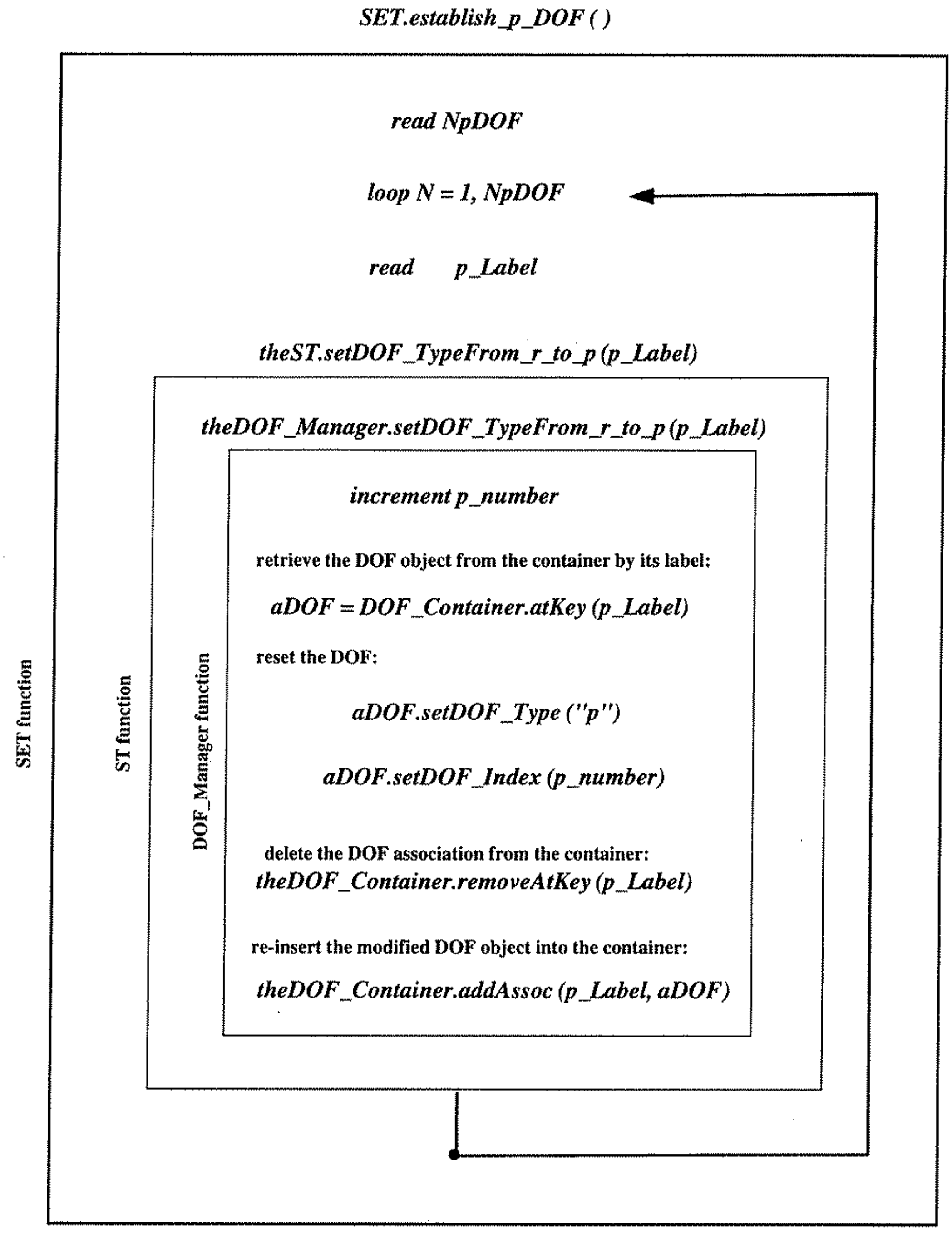
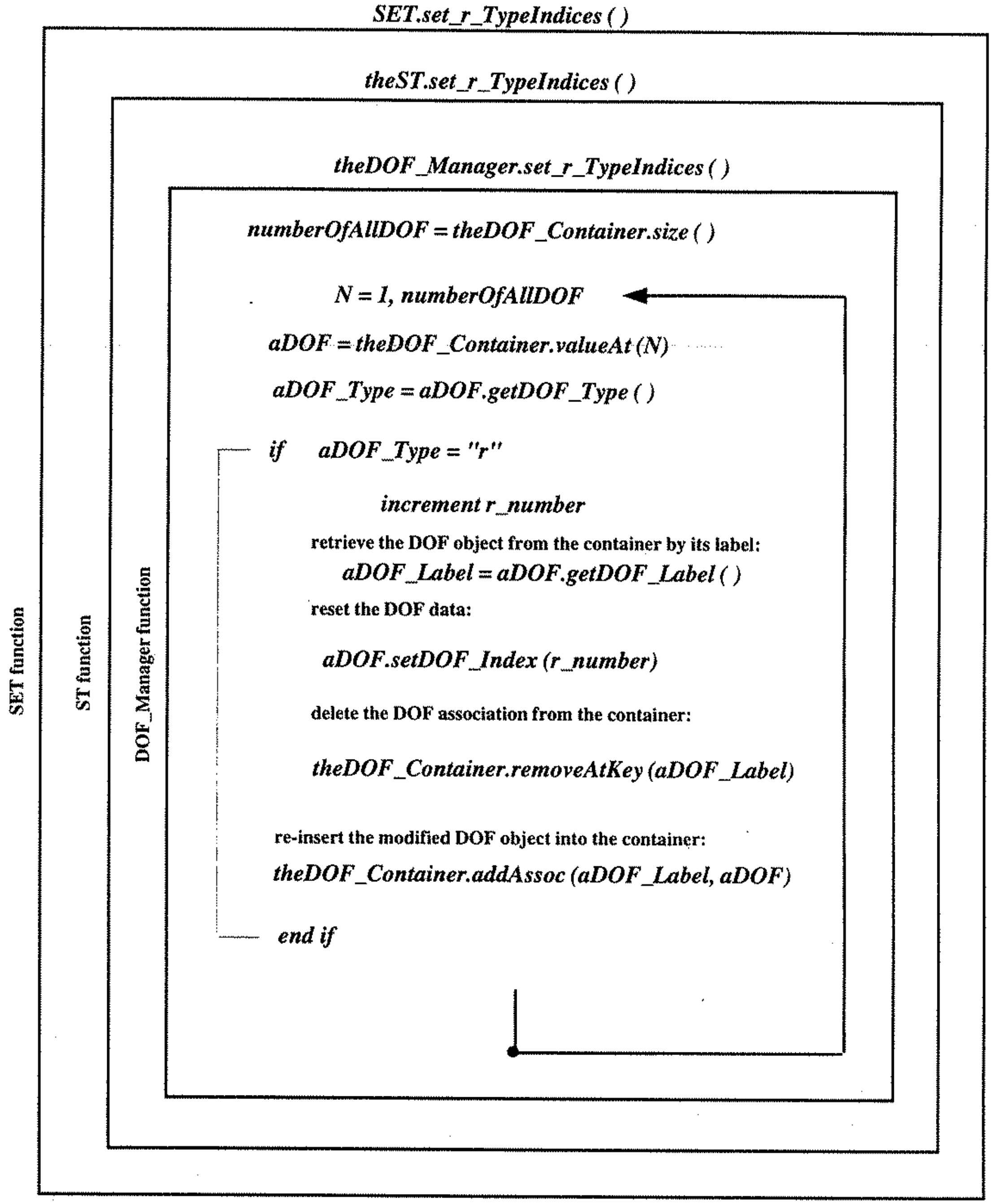
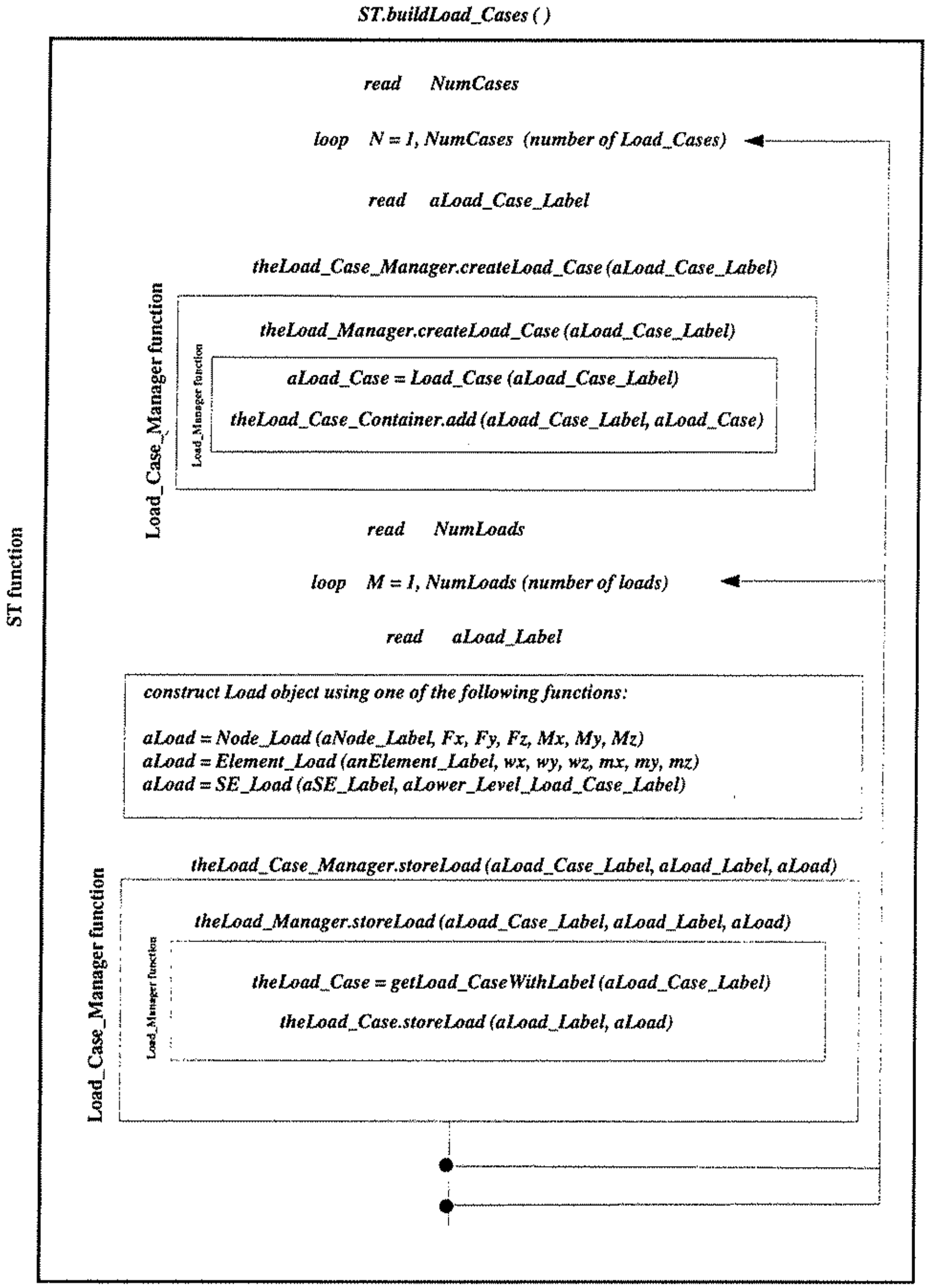

SET.establish_p_DOF ( )
read NpDOF
loop N = 1, NpDOF
read p_Label
theST.setDOF_TypeFrom_r_to_p (p_Label)
theDOF_Manager.setDOF_TypeFrom_r_to_p (p_Label)
increment p_number
retrieve the DOF object from the container by its label:
aDOF = DOF_Container.atKey (p_Label)
reset the DOF:
aDOF.setDOF_Type ("p")
aDOF.setDOF_Index (p_number)
delete the DOF association from the container:
theDOF_Container.removeAtKey (p_Label)
re-insert the modified DOF object into the container:
theDOF_Container.addAssoc (p_Label, aDOF)
SET function
ST function
DOF_Manager function

Fig. 6.6. Algorithm to modify "p" type DOF objects.



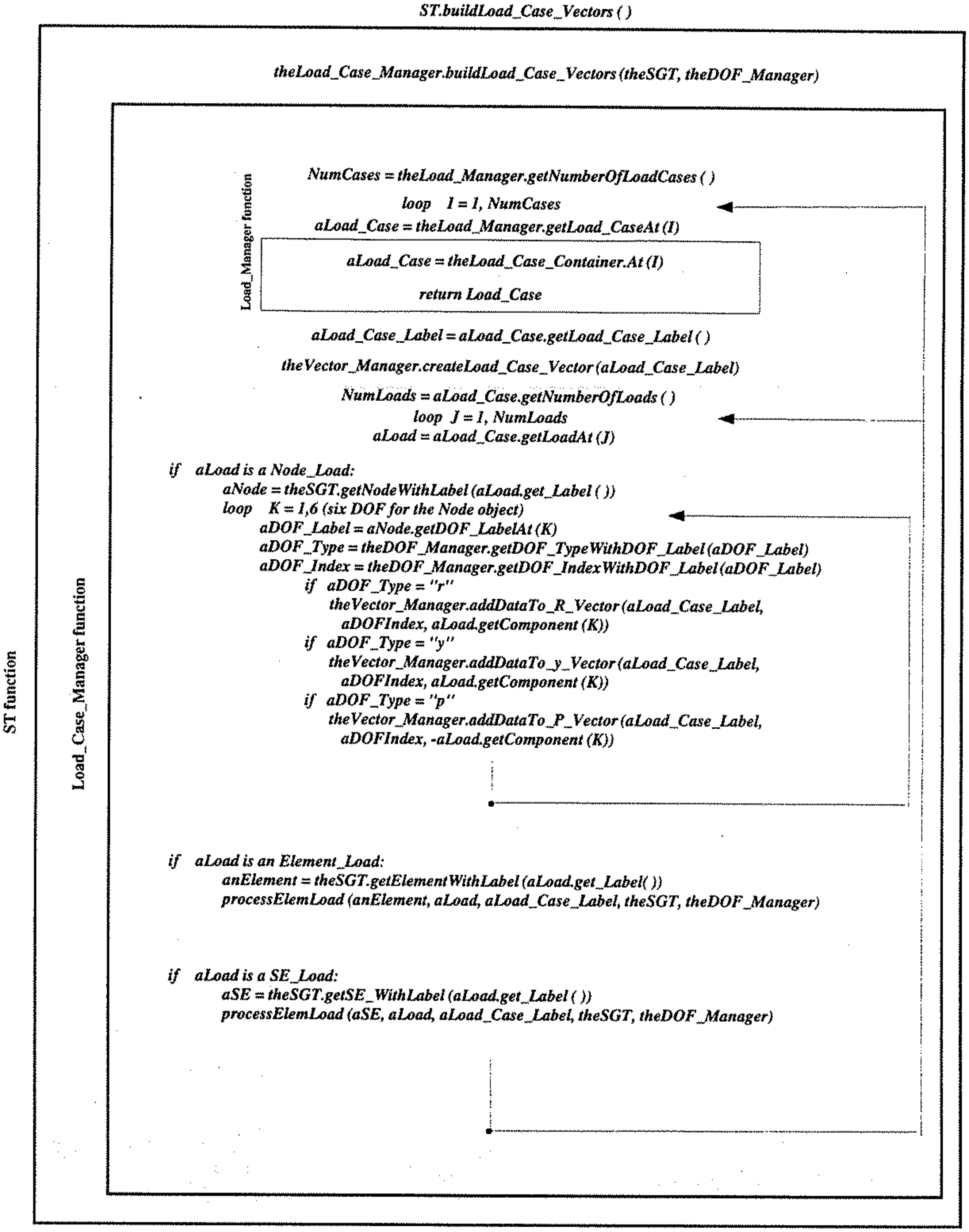
```
SET.set_r_TypeIndices ( )
  theST.set_r_TypeIndices ( )
    theDOF_Manager.set_r_TypeIndices ( )
      numberOfAllDOF = theDOF_Container.size ( )

          N = 1, numberOfAllDOF
        aDOF = theDOF_Container.valueAt (N)
        aDOF_Type = aDOF.getDOF_Type ( )

        if    aDOF_Type = "r"

                increment r_number
            retrieve the DOF object from the container by its label:
                aDOF_Label = aDOF.getDOF_Label ( )
            reset the DOF data:

                aDOF.setDOF_Index (r_number)

            delete the DOF association from the container:

            theDOF_Container.removeAtKey (aDOF_Label)

          re-insert the modified DOF object into the container:
          theDOF_Container.addAssoc (aDOF_Label, aDOF)

        end if
```

Fig. 6.7. Algorithm to set DOF indices for "r" type DOF objects.



```
ST.buildLoad_Cases ( )
ST function
    read NumCases
    loop N = 1, NumCases (number of Load_Cases)
        read aLoad_Case_Label
        theLoad_Case_Manager.createLoad_Case (aLoad_Case_Label)
        Load_Case_Manager function
            theLoad_Manager.createLoad_Case (aLoad_Case_Label)
            Load_Manager function
                aLoad_Case = Load_Case (aLoad_Case_Label)
                theLoad_Case_Container.add (aLoad_Case_Label, aLoad_Case)
        read NumLoads
        loop M = 1, NumLoads (number of loads)
            read aLoad_Label
            construct Load object using one of the following functions:
                aLoad = Node_Load (aNode_Label, Fx, Fy, Fz, Mx, My, Mz)
                aLoad = Element_Load (anElement_Label, wx, wy, wz, mx, my, mz)
                aLoad = SE_Load (aSE_Label, aLower_Level_Load_Case_Label)
            theLoad_Case_Manager.storeLoad (aLoad_Case_Label, aLoad_Label, aLoad)
            Load_Case_Manager function
                theLoad_Manager.storeLoad (aLoad_Case_Label, aLoad_Label, aLoad)
                Load_Manager function
                    theLoad_Case = getLoad_CaseWithLabel (aLoad_Case_Label)
                    theLoad_Case.storeLoad (aLoad_Label, aLoad)
```

Fig. 6.8. Algorithm to establish Load_Case objects.



```
ST.buildLoad_Case_Vectors ( )

theLoad_Case_Manager.buildLoad_Case_Vectors (theSGT, theDOF_Manager)

        NumCases = theLoad_Manager.getNumberOfLoadCases ( )
                loop   I = 1, NumCases
        aLoad_Case = theLoad_Manager.getLoad_CaseAt (I)
                aLoad_Case = theLoad_Case_Container.At (I)
                return Load_Case
        aLoad_Case_Label = aLoad_Case.getLoad_Case_Label ( )
        theVector_Manager.createLoad_Case_Vector (aLoad_Case_Label)
        NumLoads = aLoad_Case.getNumberOfLoads ( )
                loop  J = 1, NumLoads
        aLoad = aLoad_Case.getLoadAt (J)

if   aLoad is a Node_Load:
     aNode = theSGT.getNodeWithLabel (aLoad.get_Label ( ))
     loop   K = 1,6 (six DOF for the Node object)
          aDOF_Label = aNode.getDOF_LabelAt (K)
          aDOF_Type = theDOF_Manager.getDOF_TypeWithDOF_Label (aDOF_Label)
          aDOF_Index = theDOF_Manager.getDOF_IndexWithDOF_Label (aDOF_Label)
               if  aDOF_Type = "r"
                   theVector_Manager.addDataTo_R_Vector (aLoad_Case_Label,
                     aDOFIndex, aLoad.getComponent (K))
               if  aDOF_Type = "y"
                   theVector_Manager.addDataTo_y_Vector (aLoad_Case_Label,
                     aDOFIndex, aLoad.getComponent (K))
               if  aDOF_Type = "p"
                   theVector_Manager.addDataTo_P_Vector (aLoad_Case_Label,
                     aDOFIndex, -aLoad.getComponent (K))

if   aLoad is an Element_Load:
     anElement = theSGT.getElementWithLabel (aLoad.get_Label( ))
     processElemLoad (anElement, aLoad, aLoad_Case_Label, theSGT, theDOF_Manager)

if   aLoad is a SE_Load:
     aSE = theSGT.getSE_WithLabel (aLoad.get_Label ( ))
     processElemLoad (aSE, aLoad, aLoad_Case_Label, theSGT, theDOF_Manager)
```

Fig. 6.9. Algorithm to establish Load_Case_Vector objects.

Load_Case_Manager function

Load_Case_Manager.processElemLoad (theElement, theLoad, theLoad_Case_Label, theSGT, theDOF_Manager)

```
theElement.form_Initial_Forces (theLoad)
NumNodes = theElement.getNumberOfNodes ( )
loop    N = 1, NumNodes
    aNode_Label = theElement.getNode_Label (N)
    aNode = theSGT.getNodeWithLabel (aNode_Label)
    loop    K = 1, 6    (six DOF for the Node object)
        aDOF_Label = aNode.getDOF_LabelAt (K)
        aDOF_Type = theDOF_Manager.getDOF_TypeWithDOF_Label (aDOF_Label)
        aDOF_Index = theDOF_Manager.getDOF_IndexWithDOF_Label (aDOF_Label)
        aForceValue = theElement.get_Initial_Force_Coefficient ((N-1)*6+K)
        if      aDOF_Type = "r":
                theVector_Manager.addDataTo_R_Vector (theLoad_Case_Label, aDOFIndex, -aForceValue)
        if      aDOF_Type = "y":
                theVector_Manager.addDataTo_Y_Vector (theLoad_Case_Label, aDOFIndex, aForceValue)
        if      aDOF_Type = "p":
                theVector_Manager.addDataTo_P_Vector (theLoad_Case_Label, aDOFIndex, aForceValue)
```

Fig. 6.10. Algorithm to include initial forces for one Element or SE object in a Load_Case_Vector object.

Element.form_Initial_Forces (aLoad)

Element function

- ***obtain the element object initial force vector in the element coordinate system from the span loads included in aLoad object.***
- ***this is denoted by P_Vect.***
- ***transform forces and moments at each end separately.***

```
loop  N = 1, 4
    loop  I = 1, 3
        aValue = 0
        loop  J = 1, 3
            aValue = aValue + Element_Transformation_Matrix (J,I) * P_Vect ( (N-1)*3 + J)
        Element_Initial_Forces ( (N-1)*3 + I) = aValue
```

Fig. 6.11. Algorithm to form and transform Element object initial force vector.

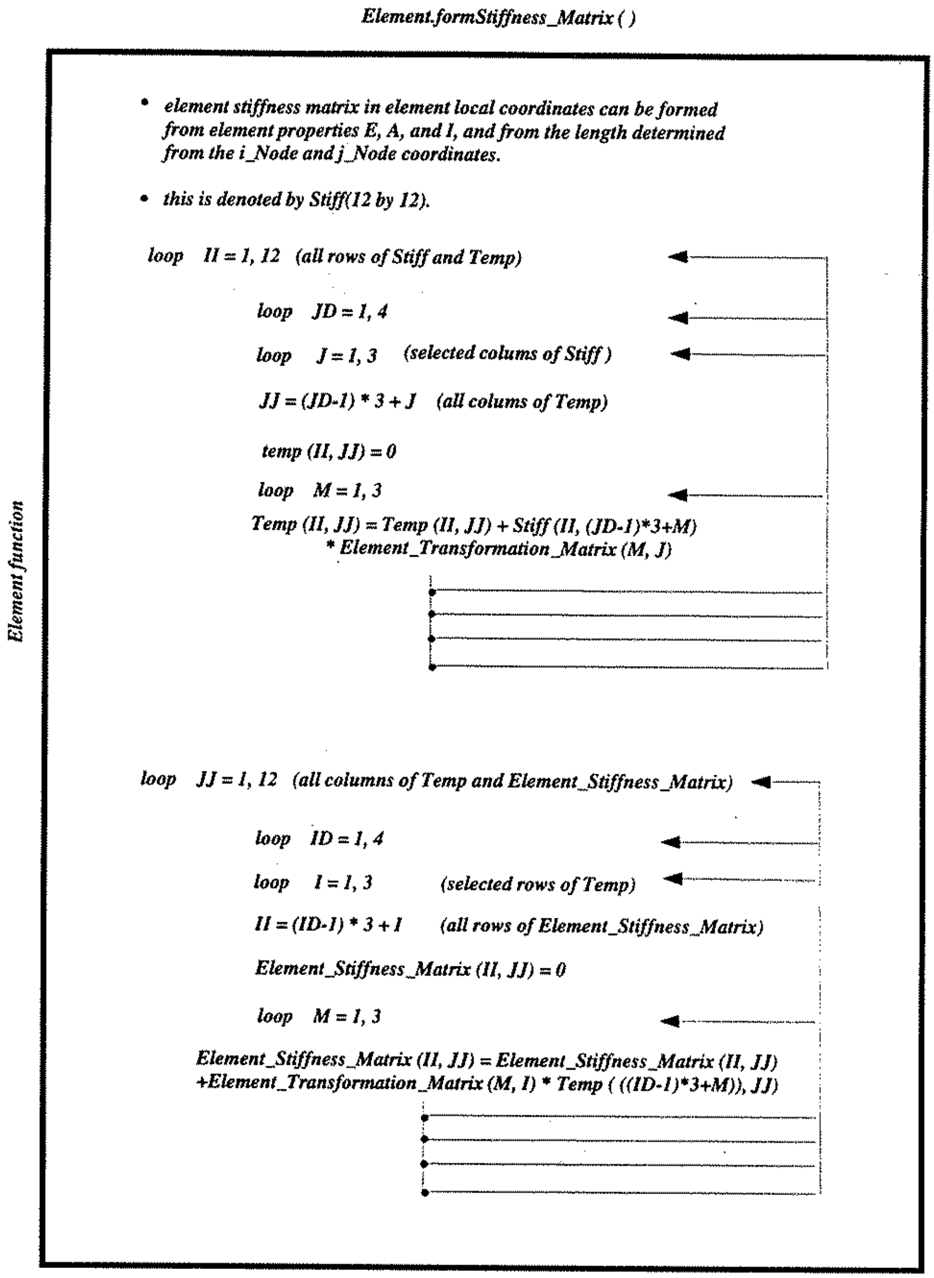
SE.form_Initial_Forces (aLoad)

SE function

- *obtain the SE object initial force vector in the superelement (lower level substructure) coordinate system from the lower level ST object through the SET object.*
- *this is denoted by P_Vect.*

```
aLower_Level_Load_Case_Label = aLoad.get_Lower_Level_Load_Case_Label( )
P_Vect = theSET.getInitial_Forces (aLower_Level_Load_Case_Label)
NumBoundaryNodes = getNumberOfNodes ( )
loop   N = 1 to NumBoundaryNodes
    Node_Label = getLowerLevelNode_Label (N)
    aLower_Level_Boundary_Node = theSET.getNodeWithLabel (Node_Label)
    loop   ID = 1, 2   (first translation DOF, then rotation DOF)
        IS = (ID-1)*3
        loop   II = 1, 3   (three DOF at a time)
            IC = IS + II   (six DOF total per node)
            aDOF_Label = aLower_Level_Boundary_Node.getDOF_LabelAt (IC)
            aDOF_Type = theSET.getLowerLevelDOF_TypeWithLabel (aDOF_Label)
            if   aDOF_Type = "p"
                aDOF_Index = theSET.getLowerLevelDOF_IndexWithLabel (aDOF_Label)
                aP_Value = P_Vect.getData (aDOF_Index)
                loop   I = 1, 3   (three translation DOF or three rotation DOF)
                    IQ = (N-1)*6 + IS + I
                    SE_Initial_Forces (IQ) = SE_Initial_Forces (IQ) + SE_Transformation_Matrix (II, I) * aP_Vect
            end if
```

Fig. 6.12. Algorithm to form and transform SE object initial force vector.



Element.formStiffness_Matrix ( )

Element function

- element stiffness matrix in element local coordinates can be formed from element properties E, A, and I, and from the length determined from the i_Node and j_Node coordinates.
- this is denoted by Stiff(12 by 12).

```
loop   II = 1, 12   (all rows of Stiff and Temp)
        loop   JD = 1, 4
        loop    J = 1, 3    (selected colums of Stiff )
        JJ = (JD-1) * 3 + J    (all colums of Temp)
        temp (II, JJ) = 0
        loop   M = 1, 3
        Temp (II, JJ) = Temp (II, JJ) + Stiff (II, (JD-1)*3+M)
                 * Element_Transformation_Matrix (M, J)


loop   JJ = 1, 12   (all columns of Temp and Element_Stiffness_Matrix)
        loop   ID = 1, 4
        loop    I = 1, 3          (selected rows of Temp)
        II = (ID-1) * 3 + I      (all rows of Element_Stiffness_Matrix)
        Element_Stiffness_Matrix (II, JJ) = 0
        loop   M = 1, 3
    Element_Stiffness_Matrix (II, JJ) = Element_Stiffness_Matrix (II, JJ)
    +Element_Transformation_Matrix (M, I) * Temp ( ((ID-1)*3+M)), JJ)
```

Fig. 6.13. Algorithm to transform Element stiffness matrix.

SE.formStiffness_Matrix ( )

SE function

```
• initialize the SE_Stiffness_Matrix.

• SE stiffness matrix in SE local coordinates is retrieved from the lower level
  ST object through theSET object.

P_Stiff = theSET.getStiffness_Matrix ( )

NumBoundNodes = getNumberOfNodes ( )
loop  IBN = NumBoundNodes
    Node_Label = getLowerLevelNode_Label (IBN)
    I_LowerLevelNode = theSET.getNodeWithLabel (Node_Label)
    loop  JBN = 1, NumBoundNodes
        Node_Label = getLowerLevelNode_Label (JBN)
        J_LowerLevelNode = theSET.getNodeWithLabel (Node_Label)
        loop  ID = 1, 2
            IS = (ID-1) * 3
            loop  II = 1, 3
                IC = IS + II        (six DOF for Node I)
                I_DOF_Label = I_LowerLevelNode.getDOF_LabelAt (IC)
                I_DOF_Type = theSET.getLowerLevelDOF_TypeWithLabel (I_DOF_Label)
                if  I_DOF_Type = "p"   (DOF of boundary Node I is "p" type)
                    I_DOF_Index = theSET.getLowerLevelDOF_IndexWithLabel (I_DOF_Label)
                    loop  JD = 1, 2
                        JS = (JD-1) * 3
                        loop  JJ = 1, 3
                            JC = JS + JJ        (6 DOF for Node J)
                            J_DOF_Label = J_LowerLevelNode.getDOF_LabelAt (JC)
                            J_DOF_Type = theSET.getLowerLevelDOF_TypeWithLabel (J_DOF_Label)
                            if  J_DOF_Type = "p"   (DOF of boundary Node J is "p" type)
                                J_DOF_Index = theSET.getLowerLevelDOF_IndexWithLabel (J_DOF_Label)
                                loop  I = 1, 3
                                    IQ = (IBN-1)*6+IS+I
                                    loop  J = 1, 3
                                        JQ = (JBN-1)*6+JS+J
                                        SE_Stiffness_Matrix (IQ, JQ) = SE_Stiffness_Matrix (IQ, JQ) +
                                          SE_Transformation_Matrix (II, I)*P_Stiff (I_DOF_Index, J_DOF_Index*
                                          SE_Transformation_Matrix (JJ, J)
                            end if
                end if
```

Fig. 6.14. Algorithm to transform SE stiffness matrix.

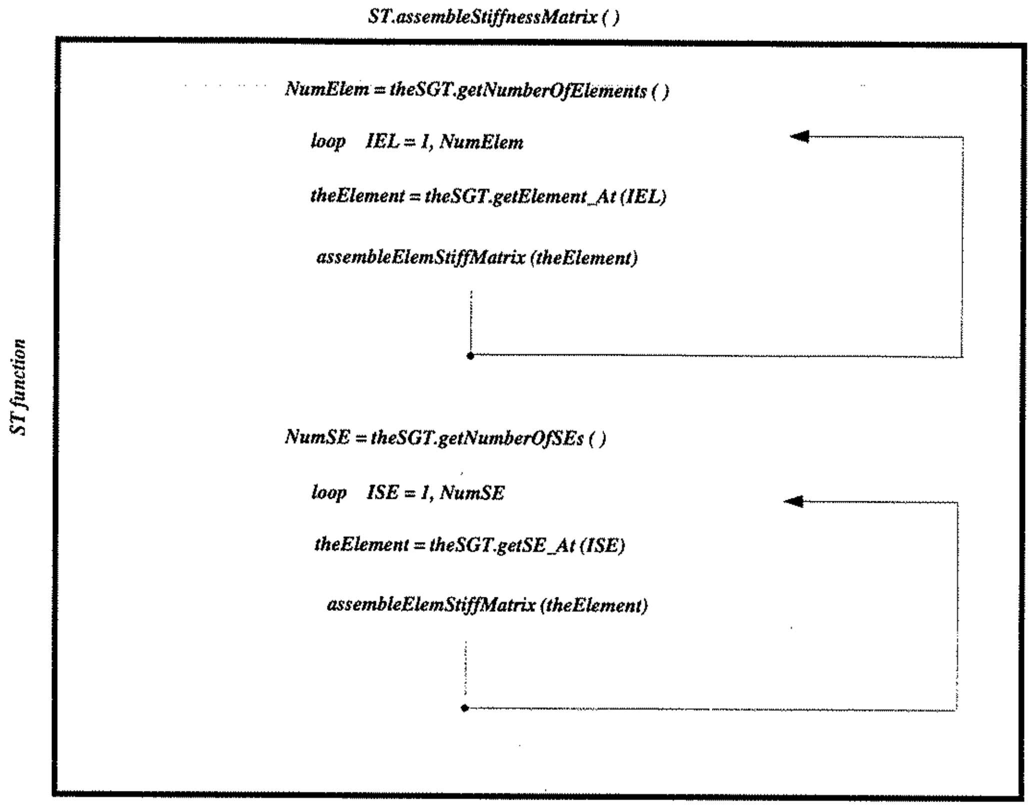
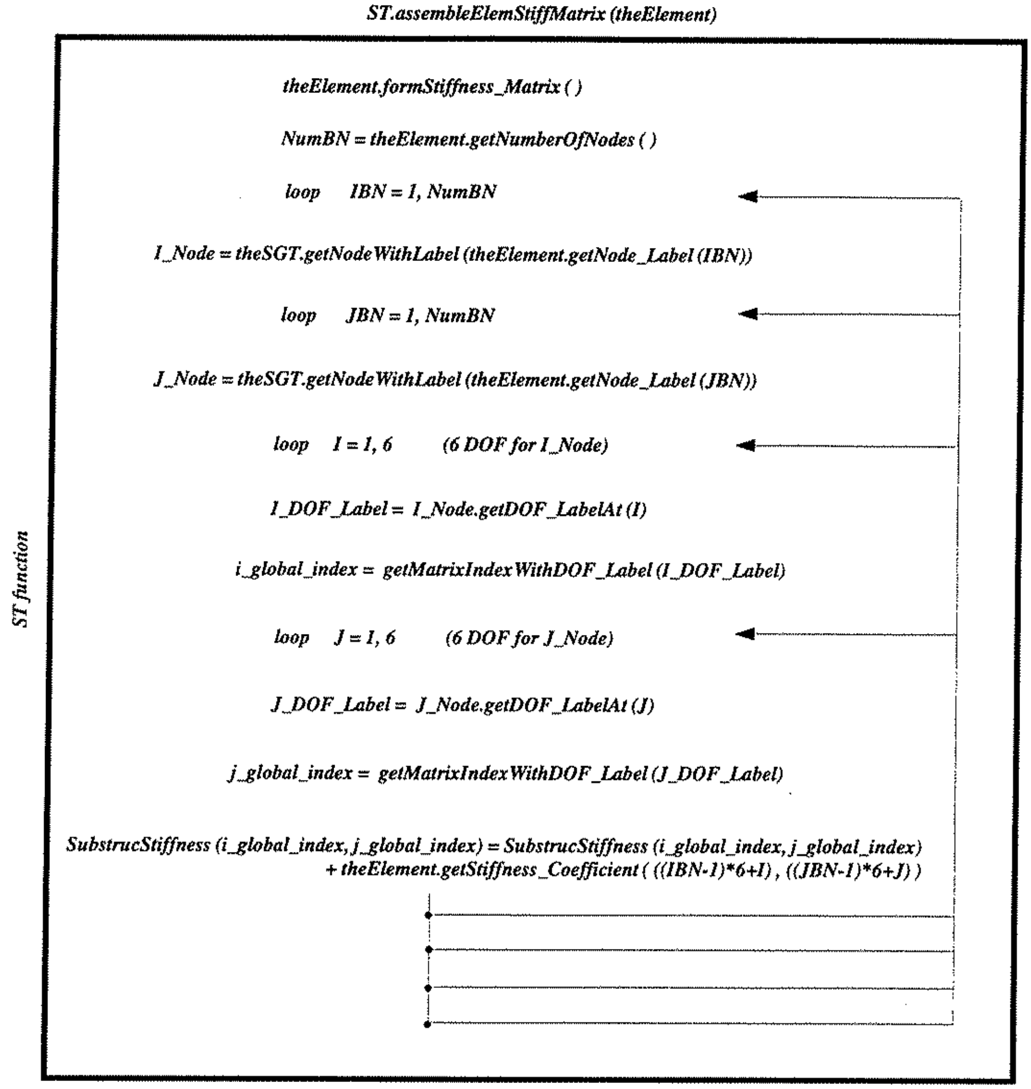
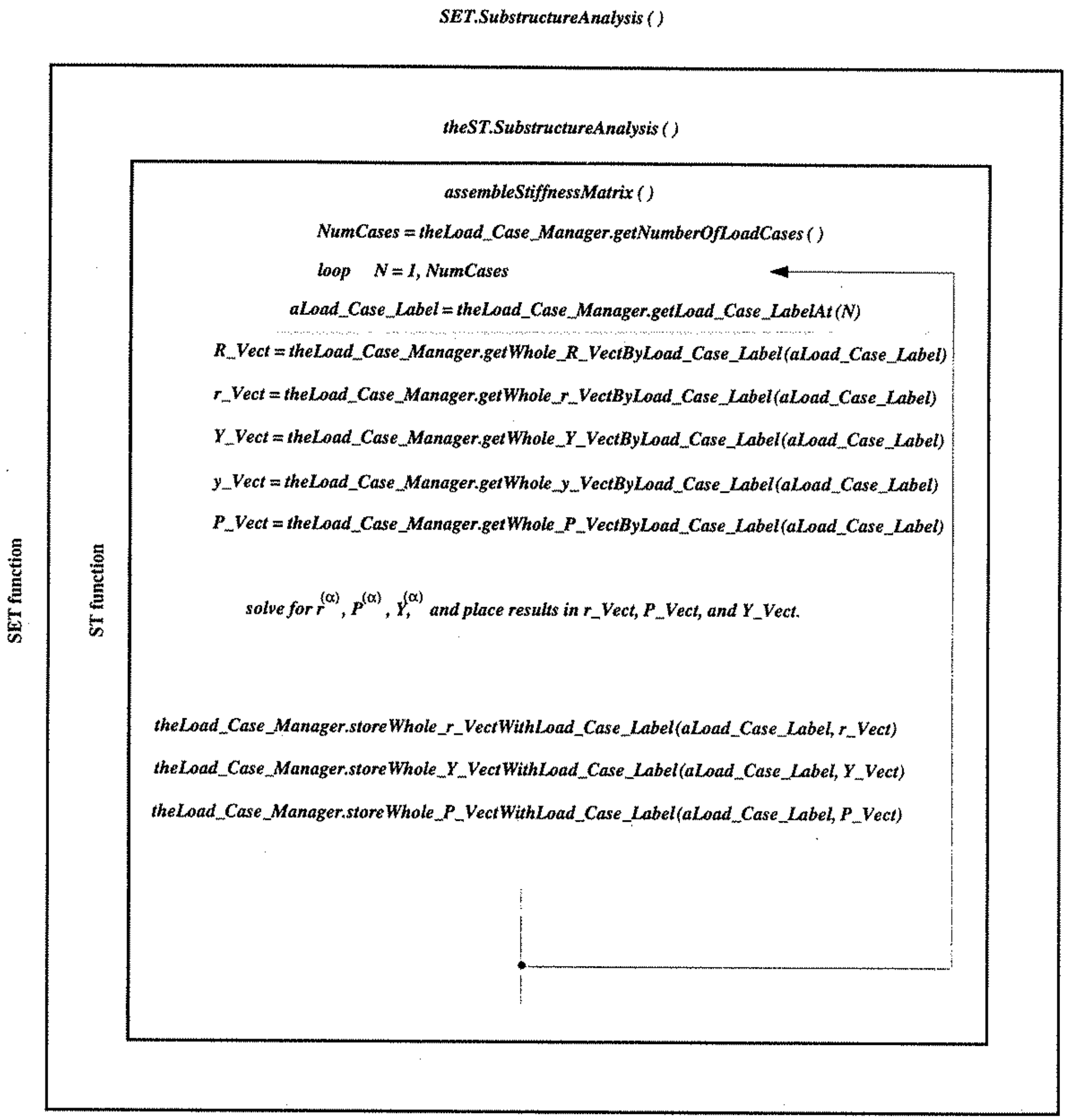

ST.assembleStiffnessMatrix ( )
ST function
NumElem = theSGT.getNumberOfElements ( )
loop IEL = 1, NumElem
theElement = theSGT.getElement_At (IEL)
assembleElemStiffMatrix (theElement)
NumSE = theSGT.getNumberOfSEs ( )
loop ISE = 1, NumSE
theElement = theSGT.getSE_At (ISE)
assembleElemStiffMatrix (theElement)

Fig. 6.15. Algorithm to assemble substructure stiffness matrix from Element and SE contributions.



```
ST.assembleElemStiffMatrix (theElement)

theElement.formStiffness_Matrix ( )
NumBN = theElement.getNumberOfNodes ( )
loop    IBN = 1, NumBN
    I_Node = theSGT.getNodeWithLabel (theElement.getNode_Label (IBN))
    loop    JBN = 1, NumBN
        J_Node = theSGT.getNodeWithLabel (theElement.getNode_Label (JBN))
        loop    I = 1, 6        (6 DOF for I_Node)
            I_DOF_Label = I_Node.getDOF_LabelAt (I)
            i_global_index = getMatrixIndexWithDOF_Label (I_DOF_Label)
            loop    J = 1, 6        (6 DOF for J_Node)
                J_DOF_Label = J_Node.getDOF_LabelAt (J)
                j_global_index = getMatrixIndexWithDOF_Label (J_DOF_Label)
                SubstrucStiffness (i_global_index, j_global_index) = SubstrucStiffness (i_global_index, j_global_index)
                        + theElement.getStiffness_Coefficient ( ((IBN-1)*6+I) , ((JBN-1)*6+J) )
```

Fig. 6.16. Algorithm to assemble stiffness matrix for Element or SE object.



```
SET.SubstructureAnalysis ( )
    theST.SubstructureAnalysis ( )
        assembleStiffnessMatrix ( )
        NumCases = theLoad_Case_Manager.getNumberOfLoadCases ( )
        loop    N = 1, NumCases
            aLoad_Case_Label = theLoad_Case_Manager.getLoad_Case_LabelAt (N)
            R_Vect = theLoad_Case_Manager.getWhole_R_VectByLoad_Case_Label(aLoad_Case_Label)
            r_Vect = theLoad_Case_Manager.getWhole_r_VectByLoad_Case_Label(aLoad_Case_Label)
            Y_Vect = theLoad_Case_Manager.getWhole_Y_VectByLoad_Case_Label(aLoad_Case_Label)
            y_Vect = theLoad_Case_Manager.getWhole_y_VectByLoad_Case_Label(aLoad_Case_Label)
            P_Vect = theLoad_Case_Manager.getWhole_P_VectByLoad_Case_Label(aLoad_Case_Label)

            solve for $r^{(\alpha)}$, $P^{(\alpha)}$, $Y^{(\alpha)}$ and place results in r_Vect, P_Vect, and Y_Vect.

            theLoad_Case_Manager.storeWhole_r_VectWithLoad_Case_Label(aLoad_Case_Label, r_Vect)
            theLoad_Case_Manager.storeWhole_Y_VectWithLoad_Case_Label(aLoad_Case_Label, Y_Vect)
            theLoad_Case_Manager.storeWhole_P_VectWithLoad_Case_Label(aLoad_Case_Label, P_Vect)
```

Fig. 6.17. The algorithm to begin a substructure analysis (α case).

ST function

```
ST.LowerLevelSubstructureAnalysisCompletion (p_Vect, Current_Level_Load_Case_Label,
                                             Higher_Level_Load_Case_Label)

    theLoad_Case_Manager.copyLoad_Case (Current_Level_Load_Case_label,
                                        Higher_Level_Load_Case_Label)

    theLoad_Case_label = Higher_Level_Load_Case_Label

    r_Vect = theLoad_Case_Manager.getWhole_r_VectByLoad_Case_Label(theLoad_Case_Label)

    Y_Vect = theLoad_Case_Manager.getWhole_Y_VectByLoad_Case_Label(theLoad_Case_Label)

    P_Vect = theLoad_Case_Manager.getWhole_P_VectByLoad_Case_Label(theLoad_Case_Label)

        • given p_Vect, solve for $r^{(\beta)}$, $P^{(\beta)}$, $Y^{(\beta)}$.
        • add $r^{(\beta)}$, $P^{(\beta)}$, $Y^{(\beta)}$ to existing values in r_Vect, P_Vect, and Y_Vect.

    theLoad_Case_Manager.storeWhole_p_VectWithLoad_Case_Label(theLoad_Case_Label, p_Vect)
    theLoad_Case_Manager.storeWhole_r_VectWithLoad_Case_Label(theLoad_Case_Label, r_Vect)
    theLoad_Case_Manager.storeWhole_P_VectWithLoad_Case_Label(theLoad_Case_Label, P_Vect)
    theLoad_Case_Manager.storeWhole_Y_VectWithLoad_Case_Label(theLoad_Case_Label, Y_Vect)

        NumElem = theSGT.getNumberOfElements ( )

        loop    IEL = 1, NumElem
            theElement = theSGT.getElement_At (IEL)
            calculateElemInternalForces (theElement, theLoad_Case_Label)

        NumSE = theSGT.getNumberOfSEs ( )

        loop    ISE = 1, NumSE
            theElement = theSGT.getSE_At (ISE)
            calculateElemInternalForces (theElement, theLoad_Case_Label)
```

Fig. 6.18. Algorithm to complete a lower level substructure analysis (β case).

ST function

ST.calculateElemInternalForces (theElement, theLoad_Case_Label)

```
NumBN = theElement.getNumberOfNodes ( )
loop   IBN = 1, NumBN
    aNode_Label = theElement.getNode_Label (IBN)
    aNode = theSGT.getNodeWithLabel (aNode_Label)
    loop   N = 1, 6     (six DOF for the Node object)
        aDOF_Label = aNode.getDOF_LabelAt (N)
        aDOF_Type = theDOF_Manager.getDOF_TypeWithDOF_Label (aDOF_Label)
        aDOF_Index = theDOF_Manager.getDOF_IndexWithDOF_Label (aDOF_Label)

        if    aDOF_Type = "r"
            theElement.addDataToDisplacements ((IBN-1)*6+N,
              theLoad_Case_Manager.getDataFrom_r_VectAt (theLoad_Case_Label, aDOF_Index))

        if    aDOF_Type = "p"
            theElement.addDataToDisplacements ((IBN-1)*6+N,
              theLoad_Case_Manager.getDataFrom_p_VectAt (theLoad_Case_Label, aDOF_Index))

        if    aDOF_Type = "y"
            theElement.addDataToDisplacements ((IBN-1)*6+N,
              theLoad_Case_Manager.getDataFrom_y_VectAt (theLoad_Case_Label, aDOF_Index))

aLoad = theLoad_Case_Manager.getLoadWithLabel (theLoad_Case_Label, theElement.getLabel ( ))
    theElement.calculateInternalForces (aLoad, theLoad_Case_Label)
```

Fig. 6.19. Algorithm to calculate internal forces for Element or SE object.

Element.calculateInternalForces (anElement_Load, theLoad_Case_Label)

Element function

- *the vector of element displacements in the substructure local coordinate system is already given to the element.*
- *this is denoted by Element_Displacements.*
- *transform translations and rotations at each end separately.*

```
loop  N = 1, 4
    loop  I = 1, 3
        aValue = 0
        loop  J = 1, 3
            aValue = aValue + Element_Transformation_Matrix (I, J)
                              *Element_Displacements ( (N-1)*3 + J)
        p_Vect ( (N-1)*3 + I) = aValue
```

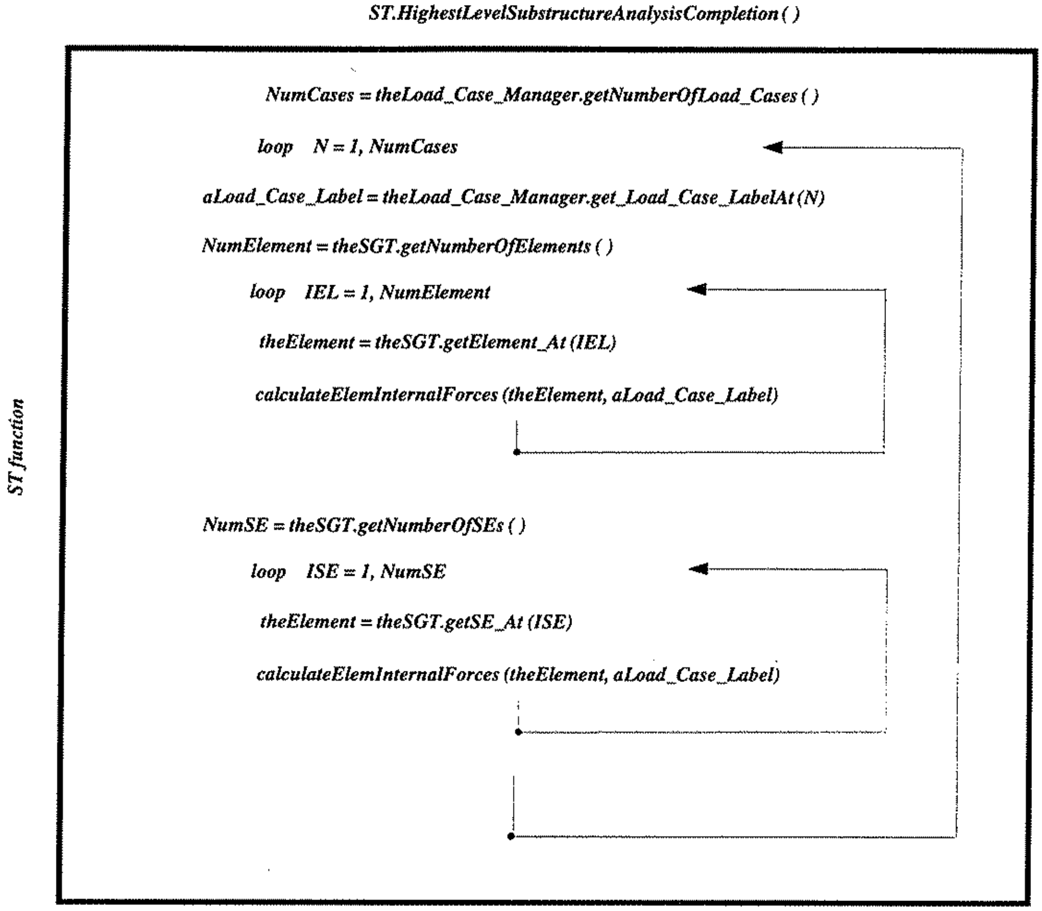
Element Internal Forces (denoted by P_Vect) is calculated by:

- *multiply p_Vect by stiffness coefficients determined from E, A, I, and the length which is determined from the i_Node and j_Node coodinates.*
- *add effect of the span loads given in anElement_Load.*

Fig. 6.20. Algorithm to calculate Element object internal forces.

```
SE.calculateInternalForces (aSE_Load, aLoad_Case_Label)

SE function

• the vector of SE boundary displacements in the higher level substructure coordinate system is already given to the SE.
• this is denoted by SE_Displacements.
    aLower_Level_Load_Case_Label = aSE_Load.getLower_Level_Load_Case_Label( )
    p_number = theSET.get_p_Number( )
• create p_Vect with dimension p_number to hold lower level substructure boundary displacement values.
      NumBoundNodes = getNumberOfNodes ( )

      loop   N = 1, NumBoundNodes
      Node_Label = getLowerLevelNodeLabel (N)
      aLower_Level_Bound_Node = theSET.getNodeWithLabel (Node_Label)
      loop   ID = 1, 2
          IS = (ID-1) * 3
      loop   II = 1, 3
          IC = IS + II     (six DOF for Node object)
      aDOF_Label = aLower_Level_Bound_Node.getDOF_LabelAt (IC)
      aDOF_Type = theSET.getLowerLevelDOF_TypeWithLabel (aDOF_Label)
    ┌─ if     aDOF_Type = "p"
    │    aDOF_Index = theSET.getLowerLevelDOF_IndexWithLabel (aDOF_Label)
    │    a_p_Value = 0
    │    loop   I = 1, 3
    │    IQ = (N-1)*6 + IS + I
    │    a_p_Value = a_p_Value + SE_Transformation_Matrix (II, I)
    │                  * SE_Displacements (IQ)
    │    p_Vect.storeData (a_p_Value, aDOF_Index)
    └─ end if
    theSET.calculateInternalForces (p_Vect, aLower_Level_Load_Case_Label, Load_Case_Label)

    SET function
      theST.LowerLevelSubstructureAnalysisCompletion (p_Vect, Current_Level_Load_Case_Label,
                                                      Higher_Level_Load_Case_Label)
        ST function
            see Fig. 6.18
```

Fig. 6.21. Algorithm to calculate SE object internal forces.



```
ST.HighestLevelSubstructureAnalysisCompletion ( )

NumCases = theLoad_Case_Manager.getNumberOfLoad_Cases ( )
loop   N = 1, NumCases
    aLoad_Case_Label = theLoad_Case_Manager.get_Load_Case_LabelAt (N)
    NumElement = theSGT.getNumberOfElements ( )
    loop   IEL = 1, NumElement
        theElement = theSGT.getElement_At (IEL)
        calculateElemInternalForces (theElement, aLoad_Case_Label)
    NumSE = theSGT.getNumberOfSEs ( )
    loop   ISE = 1, NumSE
        theElement = theSGT.getSE_At (ISE)
        calculateElemInternalForces (theElement, aLoad_Case_Label)
```

Fig. 6.22. Algorithm to complete the highest level substructure analysis.

Chapter 7

Summary and Conclusions

7.0 Summary

This report presents research on an object-oriented approach to structural analysis using substructures. Background information related to this research is presented in Chapter 2. In Chapter 3, substructure analysis theory is reviewed in detail, and the theory is extended to include degrees-of-freedom with specified displacements (and unknown external reaction forces) in addition to degrees-of-freedom with unknown displacements (and specified external applied forces) within a substructure. It is shown that these so-called "y" type degrees-of-freedom are easily included in a substructure analysis. Chapter 4 outlines data abstraction and object-oriented programming concepts that are used in developing the object-oriented structural analysis approach. The chapter also provides an overview of the C++ programming language and the NIH (National Institute of Health) class library which provide detailed programming concepts that are also used in the research.

Chapters 5 and 6 provide the major results of this research. Chapter 5 presents an object-oriented structural analysis model based on multiple levels of substructures. The model uses Substructure Geometry Types (SGTs), Substructure Types (STs), Superelement Types (SETs), and Superelements (SEs) as the major ingredients to represent substructure at various steps in a substructure analysis. Chapter 5 also outlines object classes needed to carry out structure analysis based on the model. The main instance variables and functions of the object classes are outlined using notation that is compatible with the C++ programming language. Chapter 6 describes data needed to describe a structural analysis model based on the concepts in Chapter 5, and introduces one additional object class, called the Global object, which constructs the other objects and conducts the analysis. A number of structural analysis algorithms based on the objects in Chapter 5, are also presented.

7.1 Conclusions

The research presented in the report indicates that the object-oriented approach is effective for developing programming concepts for multiple level substructure analysis. The object-oriented approach enables a clear mapping between the ingredients of the analysis model and the major components of the computer program. The primary ingredients of the model, SGTs, STs, SETs, and SEs, as well as the standard structural analysis concepts of degree-of-freedom (DOF), node, element, load, and load case map directly to the primary object classes of the program. Of course, as shown in Chapter 5, a large number of secondary object classes are needed to support these primary object classes. However, as shown in Chapter 6, the required input data describes the analysis model in terms of the primary ingredients and standard concepts, and this data is used to construct objects from the corresponding primary object classes. Thus, the analyst is concerned with only the primary object classes.

The research presented in the report has several limitations. Only linear structural analysis is considered. Nonlinear analysis would complicate substantially the interactions between the multiple levels of substructures. Only a standard beam element is considered in this report, and other types of elements such as plate and truss elements should be studied. The functions outlined for the Element (beam element) and SE (superelement) object classes are very similar. Thus, it appears that a generic element superclass could be developed that would define the functions required of object classes for all element types, and would provide a uniform element interface for use by other objects in the program. Kinematic constraints between degrees-of-freedom within a substructure are also not considered in this report.

A computer program based on the concepts presented in the report has not yet been implemented. This is a notable limitation of the work. The object classes and algorithms that are presented in the report must be developed in full detail in order to implement the program. Some simplification of the object classes is needed, and an increased use of inheritance from the NIH classes (rather than using objects from these classes as instance variables) should be considered. A generic element superclass, and additional element object classes (for element types other than the beam element) should be developed. These new element object classes, as well as the existing Element (beam element) and SE (superelement) classes would inherit required functions from the generic element superclass. Algorithms are also needed for the linear matrix algebra needed to carry out the substructure analysis using the object classes outlined in the report. The programming concepts for object-oriented structural analysis using substructures presented in this report should be verified by developing and testing a computer program that implements these concepts.

References

(1) An-Nashif, H.N. (1989) "Automated Structural Analysis for Computer Integrated Design," Ph.D. dissertation, University of California at Berkeley.

(2) Benscoten, S.U. (1948) "The Partitioning of Matrices in Structural Analysis," Journal of Applied Mechanics 15 pp303-307.

(3) Dodds, R.H. Jr. (1980) "Substructuring in Linear and Nonlinear Analysis," International Journal for Numerical Methods in Engineering 15 pp583-597.

(4) Fenves, G.L. (1990) "Object-Oriented Programming for Engineering Software Development," Engineering with Computers 6 pp1-15.

(5) Forde, B.W.A., Foschi, R.O., and Stiemer, S.F. (1990) "Object-Oriented Finite Element Analysis," Computers with Structures 34(3) pp355-374.

(6) Furuike, T. (1972) "Computerized Multiple Level Substructuring Analysis," Computers and Structures 2 pp1063-1073.

(7) Gorlen, K.E., Orlow, S.M., and Plexico, P.S., (1990) Data Abstraction And Object-Oriented Programming in C++ , John Wiley & Sons.

(8) Kamel, H.A., Liu, D., and McCabe, M.W. (1972) "Some Developments in the Analysis of Complex Ship Structures," Advances in Computational Methods in Structural Mechanics and Design , pp703-726.

(9) Keirouz W.T. and Rehak D.R. (1987) "Development of An Object-Oriented Domain Model for Constructed Facilities," Artificial Intelligence in Engineering: Tools and Techniques, pp259-271.

(10) Lee, H.H. and Arora, J.S. (1991) "Object-Oriented Programming For Engineering Applications," Engineering with Computers 7(4) pp226-231.

(11) Lippman, S.B. (1991) C++ Primer , 2nd Edition, Addison-Wesley Publishing Company.

(12) Miller, G.R. (1991) "Object-Oriented, Concurrent Structural Analysis," Sixth Conference on Computing in Civil Engineering , ASCE, pp36-43.

(13) Miller, G.A. (1988) "A Lisp-Base Object-Oriented Approach to Structural Analysis," Engineering with Computers 4 pp197-203.

(14) Przemieniecki, J.S. (1963) "Matrix Structural Analysis of Substructures", AIAA Journal 1(1) pp138-147.

(15) Przemieniecki, J.S. and Denke, P.H. (1966) "Joining of Complex Substructures by the Matrix Force Method," Journal of Aircraft 3(3) pp236-243.

(16) Sause, R., Martini, K., and Powell, G.H. (1992) "Object-Oriented Approaches For Integrated Engineering Design Systems," Journal of Computing in Civil Engineering 6(3) pp248-265.

(17) Sause, R. (1992) CE418 Lecture Notes , Department of Civil Engineering, Lehigh University, Bethlehem, PA 18015

(18) Scholz, S.-P. (1992) "Elements of An Object-Oriented FEM++ Program in C++," Computers & Structures 43(3) pp517-529.

(19) Stroustrup, B. (1988) "What Is Object-Oriented Programming?" IEEE Software May pp10-20.